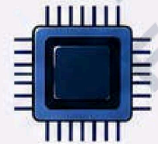
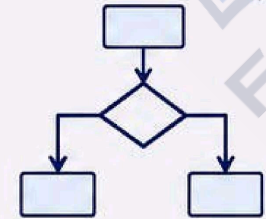


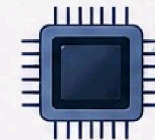
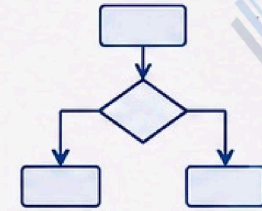
1010101010
0101010101
1010101010
0101010101



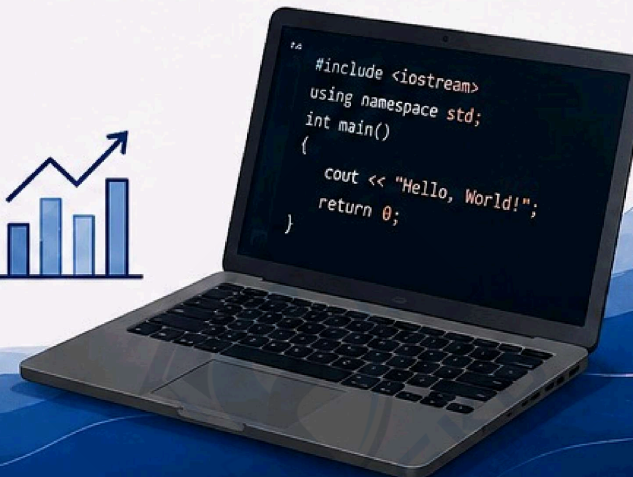
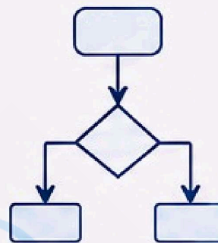
Notes



1010101010
0101010101
1010101010
0101010101



10110
01001
10101



Unit-I

* History and Evolution of Java *

Introduction :

Java is a high level, object oriented, platform independent programming language. It was developed by Sun Microsystems. It is designed to have as few implementation dependencies as possible.

• Origin of Java :

Java was originally developed in 1991 by James Gosling, Patrick Naughton, Mike Sheridan and their team at Sun Microsystems. The project was codenamed "Green". Initially, it was designed for small devices like digital TV, Set Top Box etc.

• Oak to Java :

The name of the language was first "Oak", inspired by an oak tree outside James Gosling's office. Later it was renamed as "Java" in 1995. (Java means coffee in Indonesian)

Java is a name which represents strength, reliability and universal acceptance.



Java

Write Once
Run Anywhere
(WORA)

Key Point :

Java evolved continuously to meet the growing needs of developers and technology.

Evolution / Important Milestones :

- 1991 : Java project started by James Gosling at Sun Microsystems.
- 1995 : Java officially released as a programming language.
- 1996 : JDK 1.0 released.
- 1997 : JDK 1.1 released with inner classes, JavaBeans, RMI etc.
- 1998 : J2SE 1.2 released.
- 2000 : J2SE 1.3 released.
- 2002 : J2SE 1.4 released (with features like assert, regex, NIO).
- 2004 : J2SE 5.0 released (Autoboxing, Generics, Enhanced for loop).
- 2006 : Sun Microsystems was acquired by Oracle Corporation.
- 2011 : Oracle released Java 7.
- 2014 : Oracle released Java 8 (Lambda Expressions, Stream API).
- 2017 : Java 9 released (after modularization).
- 2018 : Java 10 released.
- 2021 : Java 17 (LTS version).
- 2023+ : Latest versions with improvements in performance, security and features.

* Features of Java *

Introduction :

Java is a modern programming language developed by Sun Microsystems. It has many features that make it different from other programming languages.

Key Point :

Java is designed to be simple, secure, portable and platform independent.

- ① **Simple :** → Java is easy to learn and understand. It has a simple syntax similar to C++ and removes complicated features like pointer, operator overloading, etc.
- ② **Object Oriented:** → Java is based on Object Oriented Programming (OOPs) concepts like Class, Object, Inheritance, Polymorphism, Encapsulation and Abstraction.
- ③ **Platform Independent:** → Java programs can run on any platform (Windows, Linux, Mac, etc.) without recompilation. This is achieved by JVM.
- ④ **Secure :** → Java is secure because it does not support pointer, virus free environment, and provides automatic memory management.
- ⑤ **Robust :** → Java is robust and reliable. It has strong memory management and exception handling.
- ⑥ **Portable :** → Java programs can be easily transferred from one system to another.

⑦ **Multithreaded :**

→ Java supports multithreading. It allows a program to perform multiple tasks simultaneously.

⑧ **Distributed :**

→ Java is designed for distributed environment. It provides built-in support for networking and communication.

⑨ **High Performance :**

→ Java is faster than many other interpreted languages because of bytecode and JIT (Just In Time) compiler.

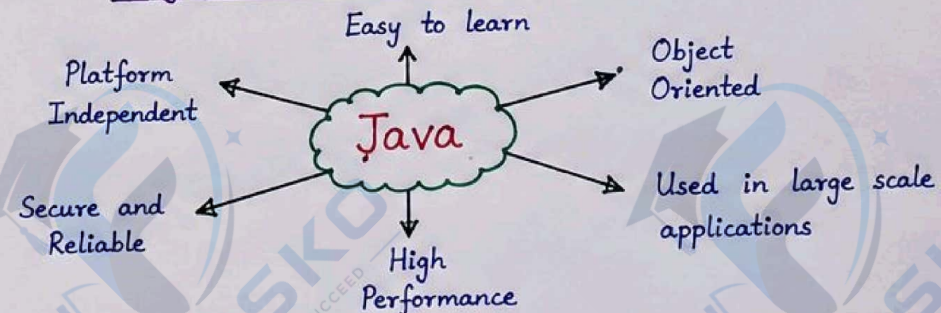
⑩ **Dynamic :**

→ Java is dynamic. It supports dynamic loading of classes. It can adapt to changing environment.

In Short :

Java is Simple, Object Oriented, Platform Independent, Secure, Robust, Portable, Multithreaded, Distributed, High Performance and Dynamic.

Why Java is Popular ?



* JRE, JDK and JVM *

① JRE (Java Runtime Environment)

Definition :

JRE is an environment that provides everything required to run a Java program. It is a part of JDK.

Components :

- JVM (Java Virtual Machine)
- Java Class Library
- Other supporting files

Purpose :

It is used to run (execute) Java programs and applets.

Key Points :

- Does not contain development tools.
- End users need JRE to run Java applications.
- It is platform dependent.

② JDK (Java Development Kit)

Definition :

JDK is a software development kit used to develop, compile, debug and run Java programs. It contains JRE and development tools.

Components :

- JRE (Java Runtime Environment)
- Development Tools (Compiler, Debugger, Javadoc, etc.)
- Standard Class Library
- Other Utility Programs

Purpose :

It is used by developers to create and build Java applications.

Key Points :

- Contains JRE + Development tools.
- Required for development of Java programs.
- It is platform dependent.
- Larger in size than JRE.

③ JVM (Java Virtual Machine)

Definition :

JVM is a virtual machine that acts as an intermediary between the compiled Java program (bytecode) and the operating system. It is the heart of Java.

Working :

- ① Java code is compiled by compiler (javac) into bytecode (.class file).
- ② This bytecode is executed by JVM.
- ③ JVM converts bytecode into machine code (native code) line by line.
- ④ Then OS executes the machine code.

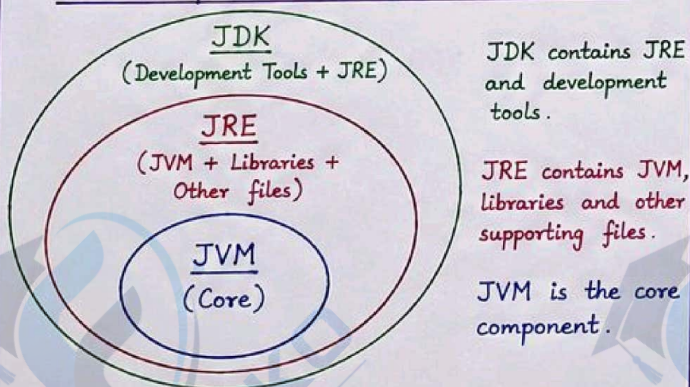
Features :

- Provides platform independence.
- Memory management (automatic garbage collection).
- Security through class loader and bytecode verifier.
- Executes code in a safe environment.

Key Points :

- JVM is part of JRE.
- It is platform dependent.
- Without JVM, Java's "Write Once, Run Anywhere" is not possible.

Relationship between JDK, JRE and JVM



Difference Between JRE and JDK

Feature	JRE	JDK
Full Form	Java Runtime Environment	Java Development Kit
Purpose	To run Java programs.	To develop, compile, debug and run Java programs.
Contains	JVM + Libraries + Other files	JRE + Development Tools
Used By	End Users	Developers
Size	Smaller	Larger
Example	To run .class file	To create .class file

In Short :

JDK is for developers, JRE is for running programs and JVM is the engine that makes Java platform independent. *

* Data Types in Java *

Introduction :

A data type specifies the type of data that a variable can store. Java is a strongly typed language. It supports two types of data types.

Every variable in Java must be declared with a data type.

① Primitive Data Types (Built-in)

Primitive data types are the basic data types. They store single values.

Data Type	Size	Description	Range / Value
byte	1 byte	Stores whole numbers	-128 to 127
short	2 bytes	Stores whole numbers	-32,768 to 32,767
int	4 bytes	Stores whole numbers	-2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores decimal numbers	1.4E-45 to 3.4028235E+38 (approx)
double	8 bytes	Stores decimal numbers	4.9E-324 to 1.7976931348623157E+308 (approx)
char	2 bytes	Stores a single character	\u0000 to \uFFFF (0 to 65,535)
boolean	1 bit	Stores true or false values	true or false

Examples :

byte b = 10;	float pi = 3.14f;	String name = "Saqib";
short s = 200;	double salary = 12345.6789;	int [] marks = {85, 90, 78};
int age = 20;	char grade = 'A';	Student st = new Student();
long population = 780000000L;	boolean flag = true;	// (user defined class)

Important Points :

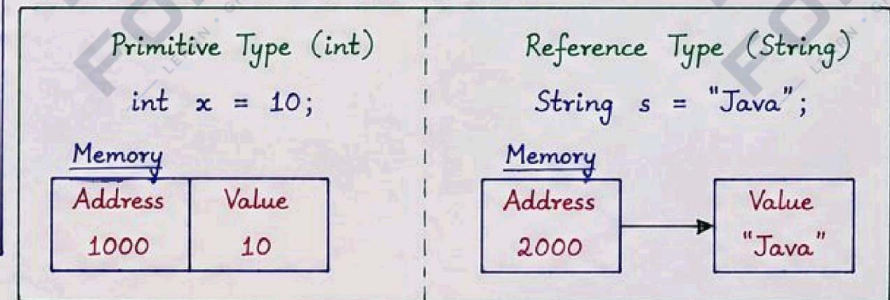
- Java is case sensitive.
- Size of data type may differ from one system to another.
- char is enclosed in single quotes.
- String is enclosed in double quotes.

② Non-Primitive / Reference Data Types

These are not built-in types. They store reference (address) of objects.

Data Type	Description
String	Represents a sequence of characters.
Array	Stores multiple values of same type.
Class	User defined data type.
Interface	Blueprint of a class.
Object	Instance of a class.

Memory Representation (Example) :



Note :

- Use 'L' or 'l' at the end of long values.
- Use 'f' or 'F' at the end of float values.
- double values do not require any suffix.
- boolean values are either true or false.

* Keywords in Java *

Introduction :

Keywords are predefined words in Java that have a special meaning. They are reserved by Java and cannot be used as identifiers (name of variables, classes, methods etc.).

Key Points :

- They are written in lowercase.
- Keywords cannot be used as variable name, class name, method name etc.
- There are 50 keywords in Java (as per latest versions).

Rules for Keywords :

- ① All keywords are in lowercase.
- ② They have a fixed meaning.
- ③ They cannot be used as user defined names (identifiers).
- ④ They are used to define the syntax and structure of a program.

Example :

- ✗ `int class = 10;` → Invalid (class is a keyword)
- ✓ `int classNo = 10;` → Valid

List of Java Keywords :

abstract	double	int	super	true
assert	else	interface	switch	false
boolean	enum	long	synchronized	null
break	extends	native	this	
byte	final	new	throw	
case	finally	package	throws	
catch	float	private	transient	
char	for	protected	try	
class	goto*	public	void	
const*	if	return	volatile	
continue	implements	short	while	
default	import	static		
do	instanceof	strictfp		

* const and goto are reserved but not used in modern Java.

Note :

- Java is case sensitive, but all keywords are in lowercase.
- You cannot use keywords as variable name, method name, class name etc.
- Trying to do so will give compilation error.

Structure of a Java Program

1. Documentation Section	→	<pre>/* This is a simple Java Program */</pre>	} — Used to write comments about the program.
2. Package Statement	→	<pre>package myprogram;</pre>	→ Defines the package to which the class belongs.
3. Import Statements	→	<pre>import java.util.*; import java.io.*;</pre>	→ Used to include required classes from libraries.
4. Interface / Class Definition	→	<pre>public class Hello {</pre>	→ Defines a class. Every Java program must have at least one class.
5. Main Method	→	<pre>public static void main(String[] args) {</pre>	→ Every Java program must have main() method. It is the starting point of execution.
6. Variable Declaration	→	<pre>int number = 10; double price = 99.99; char grade = 'A';</pre>	} → Variables are declared to store data.
7. Executable Statements	→	<pre>System.out.println("Hello, World!"); System.out.println("Welcome to Java!"); number = number + 5;</pre>	} — Statements are executed in sequence.
8. Return Statement / End of Program	→	<pre>return; }</pre>	→ Indicates the end of main() method.

Java Command Line Argument

What is it?

Command line arguments are values passed to a Java program at the time of execution.

How it works?

- Arguments are passed after the program name in the command prompt (or terminal).
- They are received in the main() method as an array of Strings.
- The first argument is stored at args[0], second at args[1] and so on.

Syntax

```
public static void main(String[] args)
{
    // args is an array of strings
}
```

Points to remember

1. If no arguments are passed, args.length will be 0.
2. All command line arguments are treated as Strings.

Example Program

```
public class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Total arguments: " + args.length);
        for (int i = 0; i < args.length; i++)
        {
            System.out.println("args[" + i + "] = " + args[i]);
        }
    }
}
```

args.length gives the number of arguments passed.

Compilation

```
javac Demo.java
```

Execution

```
java Demo Hello 123 "Good Morning"
```

↑ First argument ↑ Second argument ↑ Third argument ↑ Fourth argument

Output

```
Total arguments: 4
args[0] = Hello
args[1] = 123
args[2] = Good Morning
```

Note: We can pass any number of arguments. They are separated by spaces.

Simple Java Program

What is it ?

A simple Java program prints a message on the screen.

Structure

1. Class Declaration

Every Java program must have a class.

2. main() Method

The execution of program starts from main() method.

3. Print Statement

System.out.println() is used to display output.

4. End of Class

Indicates the end of the class.

Program

```
public class SimpleDemo           // Class Declaration
{
    public static void main(String[] args) // main() method
    {
        System.out.println("Hello, World!"); // Print Statement
    }
                                           // End of main() method
}                                           // End of class
```

Output

Hello, World!

→ This will be displayed on the screen.

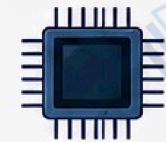
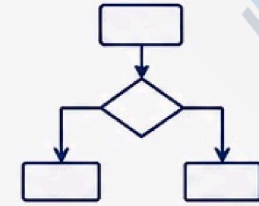
Steps to Run

1. Write the program in a file with .java extension.
2. Compile the program using: `javac SimpleDemo.java`
3. Run the program using: `java SimpleDemo`

Note :

- * Java is case-sensitive.
- * Every statement must end with a semicolon (;).

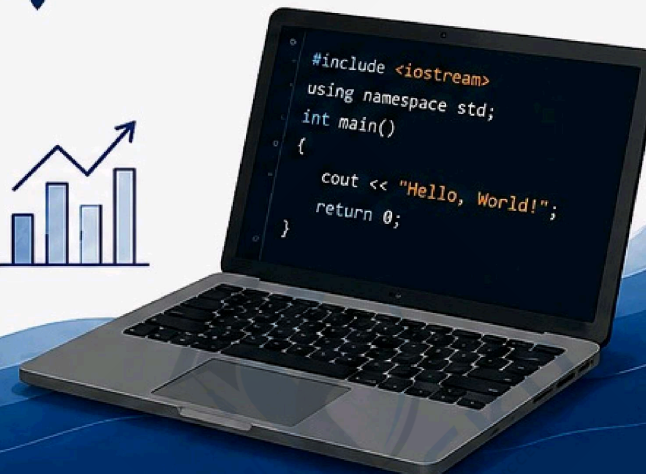
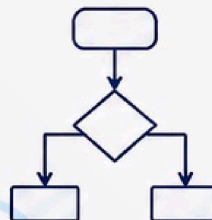
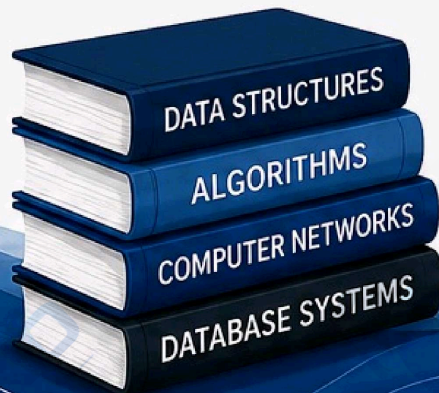
1010101010
0101010101
1010101010
0101010101



Unit - II



10110
01001
10101



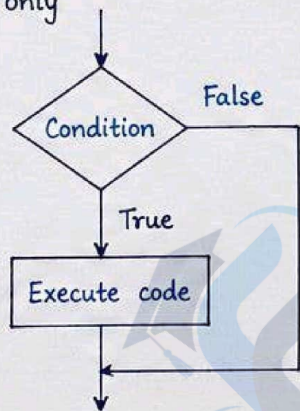
DECISION MAKING AND BRANCHING

Decision making statements are used to execute a block of code based on a certain condition.

1) Simple if statement :

Executes a block of code only if the condition is true.

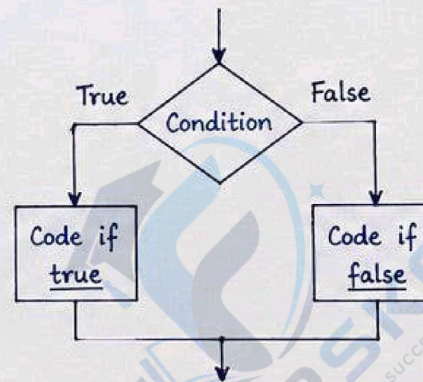
```
if (condition) {  
    // code to be executed  
}
```



2) if...else statement :

Executes one block if condition is true and another block if it is false.

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



3) Nesting of if...else statement :

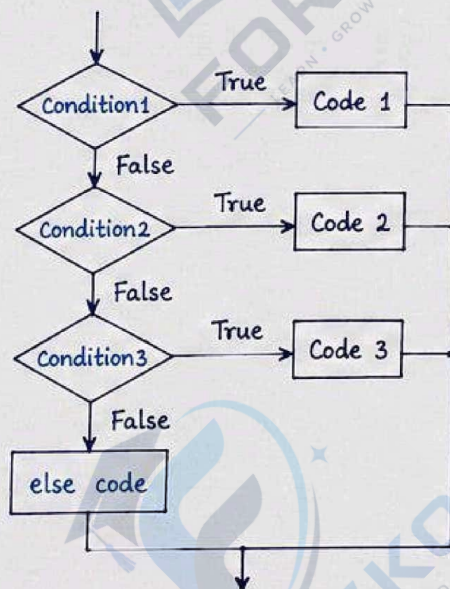
An if or else statement inside another if or else statement.

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

4) Else...if ladder :

Used to test multiple conditions one by one.

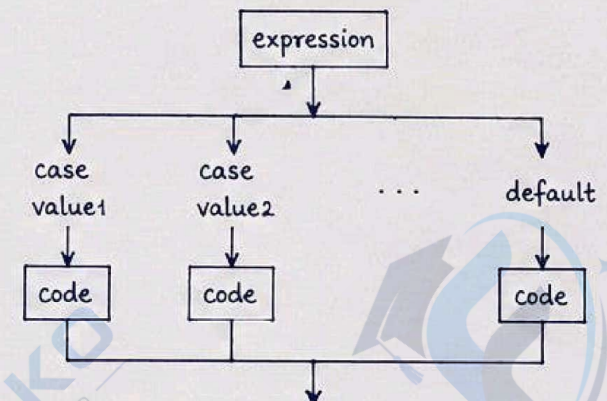
```
if (condition1) {  
    // code  
} else if (condition2) {  
    // code  
} else if (condition3) {  
    // code  
} else {  
    // code  
}
```



5) Switch statement :

Used to execute one block of code from many options based on the value of a variable or expression.

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



Simple if statement

Definition :

The Simple if statement is used to execute a block of code only if the given condition is true.

Syntax :

```
if (condition) {  
    // statements  
}
```

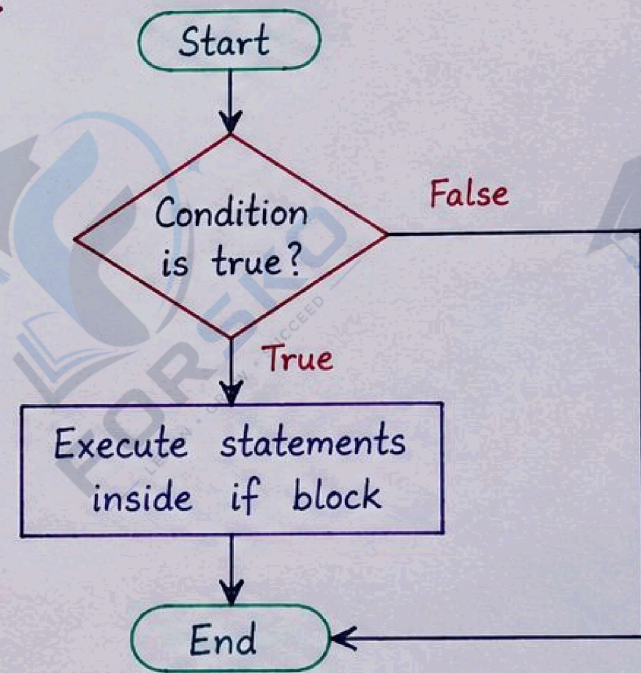
Example :

```
public class SimpleIfExample {  
    public static void main(String[] args) {  
        int age = 20;  
        if (age >= 18) {  
            System.out.println("You are eligible to vote.");  
        }  
    }  
}
```

How it works :

- First, the condition inside the if is evaluated.
- If the condition is true, then the block of statements inside if is executed.

Flowchart :



Key Point :

If the condition is false, the statements inside the if block are skipped.

IF - ELSE STATEMENT

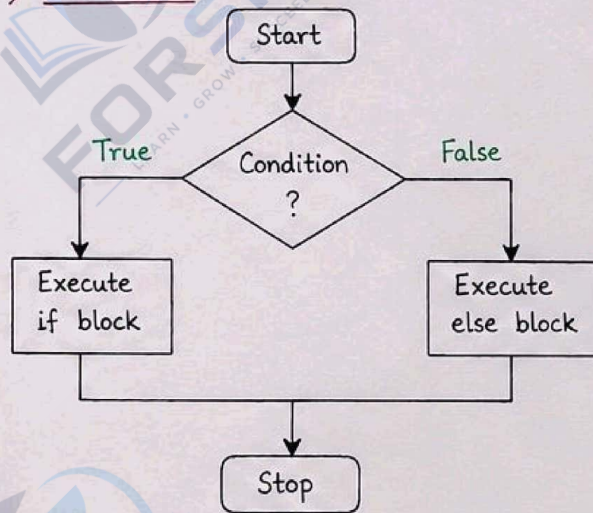
1) Introduction:

The if-else statement is used to execute a block of code among two alternatives.

2) Syntax:

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

3) Flowchart:



4) How it works:

- First, the condition inside if is evaluated.
- If the condition is true, then the statements inside if block are executed.
- If the condition is false, then the statements inside else block are executed.

5) Example Program:

```
import java.util.*;  
class IfElseExample {  
    public static void main(String args[]) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter your age : ");  
        int age = sc.nextInt();  
  
        if (age >= 18) {  
            System.out.println("You are eligible  
for voting.");  
        }  
        else {  
            System.out.println("You are not eligible  
for voting.");  
        }  
    }  
}
```

6) Example with working:

Input : age = 20

Condition : age >= 18 → True

Output : You are eligible for voting.

Input : age = 15

Condition : age >= 18 → False

Output : You are not eligible for voting.

7) Points to remember:

- if condition is a boolean expression (true/false).
- if block executes when condition is true.
- else block executes when condition is false.
- We can have nested if-else inside another if or else block.

8) Nested if-else Example:

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

Summary: if-else statement helps in decision making. It allows our program to choose between two paths based on a condition.

This program is created
by SHID 😊

NESTING OF IF - ELSE STATEMENT

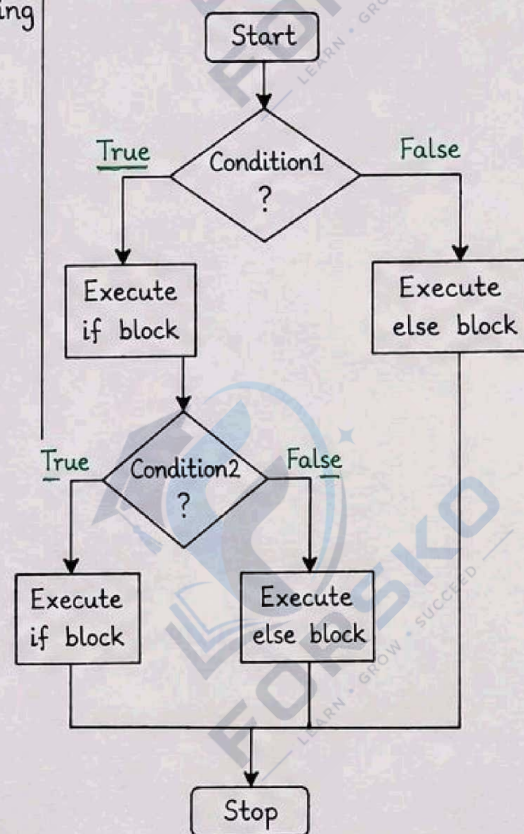
① Introduction :

Nesting of if-else means using one if-else statement inside another if or else block. It is used to test multiple conditions.

② Syntax :

```
if (condition1) {  
    // statements  
    if (condition2) {  
        // statements  
    }  
    else {  
        // statements  
    }  
}  
else {  
    // statements  
}
```

③ Flowchart :



④ Example Program :

```
import java.util.*;  
class NestedIfExample {  
    public static void main(String args[]) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter marks : ");  
        int marks = sc.nextInt();  
  
        if (marks >= 90) {  
            if (marks >= 95) {  
                System.out.println("Grade : A+");  
            } else {  
                System.out.println("Grade : A");  
            }  
        } else if (marks >= 60) {  
            if (marks >= 75) {  
                System.out.println("Grade : B");  
            } else {  
                System.out.println("Grade : C");  
            }  
        } else {  
            System.out.println("Grade : Fail");  
        }  
    }  
}
```

⑤ Explanation :

- First, we check the outer condition (marks ≥ 90).
- If true, then we check inner condition (marks ≥ 95) inside the if block.
- If false, we go to else if (marks ≥ 60) and again check inner conditions.
- If all conditions are false, the else block will execute.

⑥ Example Output :

Enter marks : 96
Grade : A+

Enter marks : 82
Grade : B

Enter marks : 69
Grade : C

Enter marks : 45
Grade : Fail

⑦ Points to remember :

- We can put any number of if-else inside another if or else block.
- Proper indentation (spacing) improves readability.
- Use nested if-else only when necessary, too much nesting makes code complex.

⑧ Structure Example (3 levels) :

```
if (condition1) {  
    if (condition2) {  
        if (condition3) {  
            // statements  
        } else {  
            // statements  
        }  
    } else {  
        // statements  
    }  
} else {  
    // statements  
}
```

← Level 1 (outer if)
← Level 2
← Level 3 (innermost if)

This program is created
by SHID 😊

ELSE IF LADDER

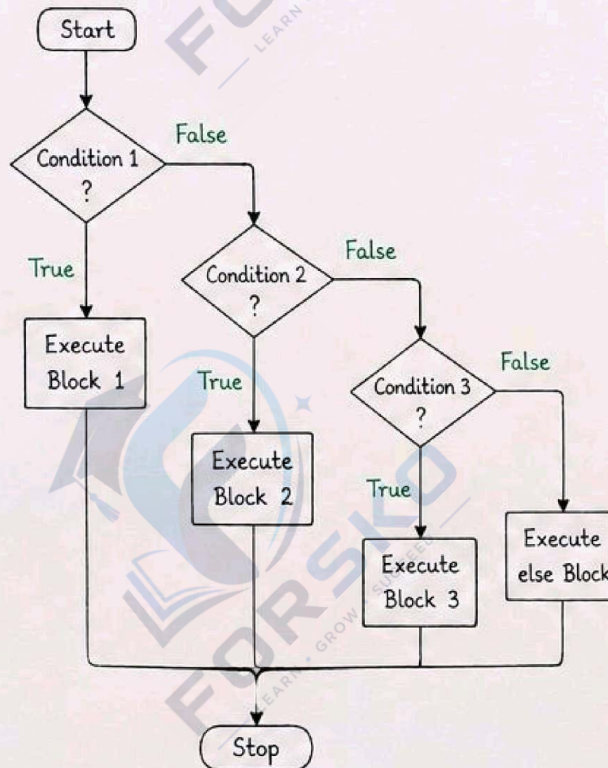
1) Introduction :

Else if ladder is used to check multiple conditions one by one. As soon as one condition is true, its block is executed and remaining conditions are skipped.

2) Syntax :

```
if (condition1) {  
    statement(s);  
}  
else if (condition2) {  
    statement(s);  
}  
else if (condition3) {  
    statement(s);  
}  
.  
.  
else {  
    statement(s); // optional  
}
```

3) Flowchart :



4) Example Program :

```
import java.util.*;  
class ElseIfLadderExample {  
    public static void main(String args[]) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter your marks :");  
        int marks = sc.nextInt();  
  
        if (marks >= 90) {  
            System.out.println("Grade : A+");  
        }  
        else if (marks >= 75) {  
            System.out.println("Grade : A");  
        }  
        else if (marks >= 60) {  
            System.out.println("Grade : B");  
        }  
        else if (marks >= 40) {  
            System.out.println("Grade : C");  
        }  
        else {  
            System.out.println("Grade : Fail");  
        }  
    }  
}
```

5) Explanation :

- First condition is checked.
- If it is true, its block executes.
- If it is false, next else if condition is checked.
- This continues until one condition is true.
- If none of the conditions is true, the else block executes.
- It is called a ladder because conditions are checked step by step like the steps of a ladder.

6) Example Output :

Enter your marks : 92
Grade : A+

Enter your marks : 78
Grade : A

Enter your marks : 63
Grade : B

Enter your marks : 38
Grade : C

Enter your marks : 25
Grade : Fail

7) Points to remember :

- Conditions are checked from top to bottom.
- As soon as one condition is true, the rest are skipped.
- Use else if ladder when you have multiple conditions for one variable/expression.
- The final else block is optional but recommended.

8) Example (Using day number) :

```
int day = 3;  
if (day == 1)        System.out.println("Monday");  
else if (day == 2)    System.out.println("Tuesday");  
else if (day == 3)    System.out.println("Wednesday");  
else if (day == 4)    System.out.println("Thursday");  
else if (day == 5)    System.out.println("Friday");  
else if (day == 6)    System.out.println("Saturday");  
else                  System.out.println("Sunday");
```

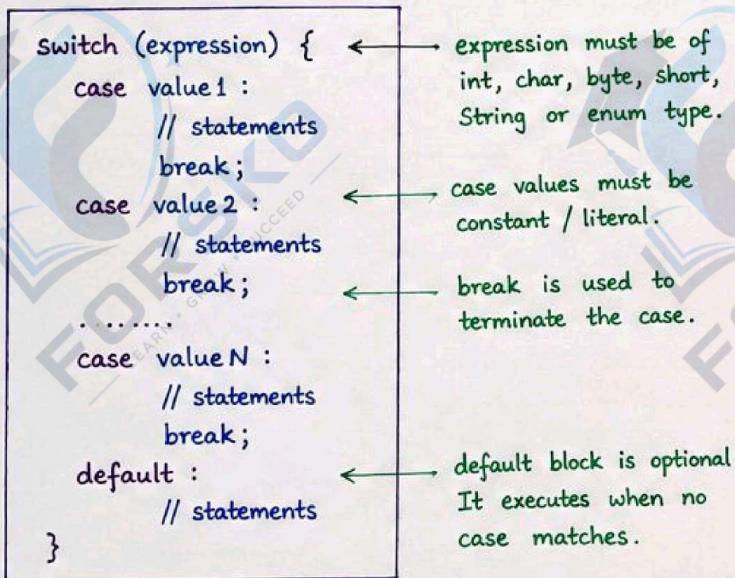
This program is created
by SHID 😊

SWITCH STATEMENT IN JAVA

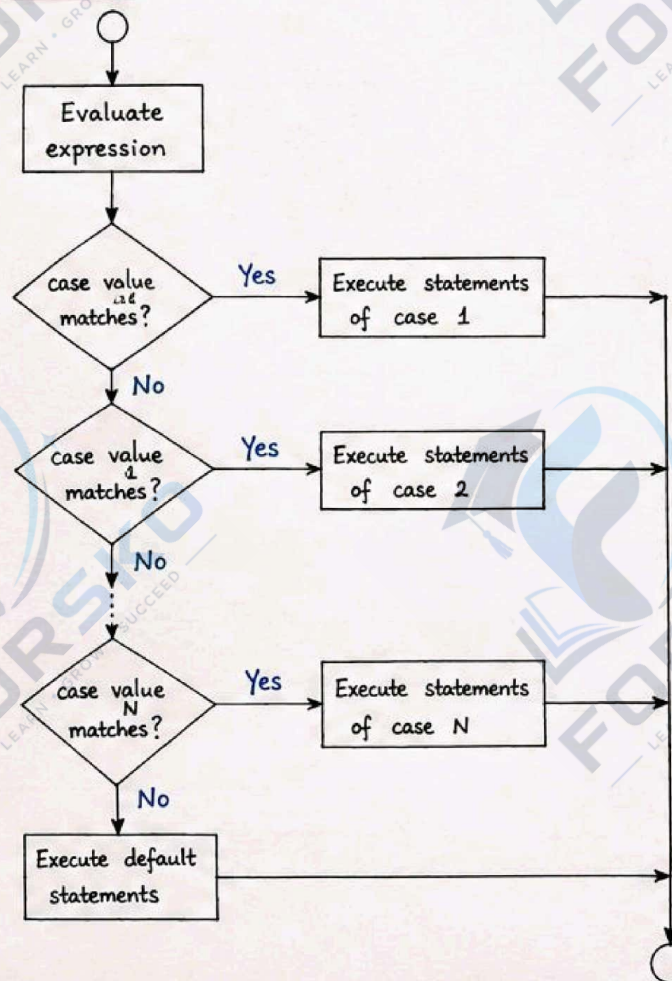
1. Definition :

Switch statement is a multi-way decision making statement. It allows a variable to be tested for equality against a list of values. It is an alternative to multiple if-else if statements.

2. Syntax :



3. Flowchart :



4. Example : Program to print day name using switch.

```
import java.util.*;  
class SwitchExample {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter day number (1-7):");  
        int day = sc.nextInt();  
        switch(day) {  
            case 1:  
                System.out.println("Monday");  
                break;  
            case 2:  
                System.out.println("Tuesday");  
                break;  
            case 3:  
                System.out.println("Wednesday");  
                break;  
            case 4:  
                System.out.println("Thursday");  
                break;  
            case 5:  
                System.out.println("Friday");  
                break;  
            case 6:  
                System.out.println("Saturday");  
                break;  
            case 7:  
                System.out.println("Sunday");  
                break;  
            default:  
                System.out.println("Invalid day number");  
        }  
    }  
}
```

5. Points to Remember :

- Each case must end with break to avoid fall-through.
- If break is omitted, control will move to the next case automatically.
- default case is optional but it is good practice to handle invalid input.
- switch improves readability when there are many possible choices.

Output Example :

Enter day number (1-7): 3
Wednesday

DESIGN MAKING

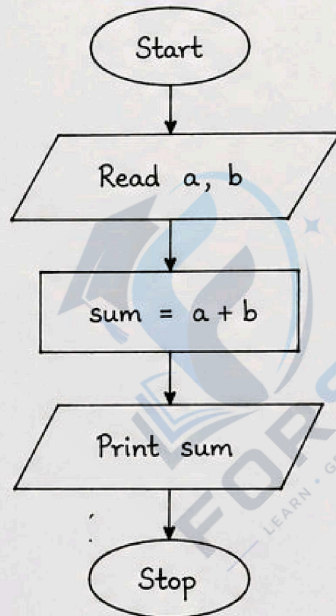
Design Making is the process of preparing the logic or steps to solve a problem before writing the program.

Example Problem : Read two numbers and print their sum.

1) Algorithm (Step by Step)

1. Start
2. Declare variables a, b and sum
3. Read two numbers a and b
4. Calculate $\text{sum} = a + b$
5. Print sum
6. Stop

2) Flowchart



3) Pseudocode

// Program to add two numbers

BEGIN

DECLARE a, b, sum

READ a, b

sum = a + b

PRINT sum

END

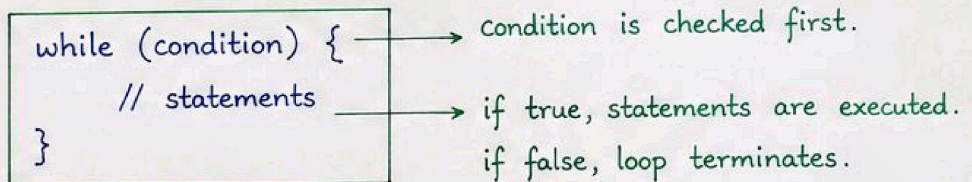
Note :

Good design makes the program simple, easy to understand and error free.

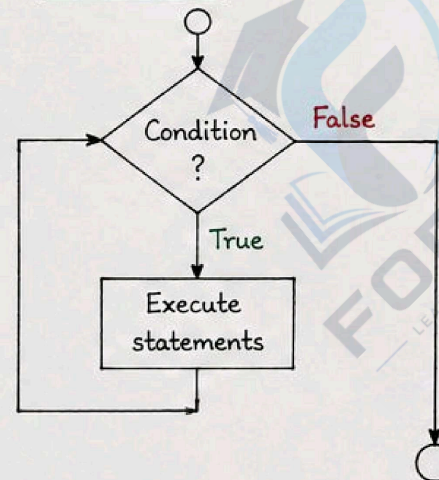
LOOPING : while STATEMENT

while loop is an entry controlled loop. It is used to execute a block of statements repeatedly as long as the given condition is true.

1) Syntax :



2) Flowchart :



Key Points :

- Condition is checked before executing the loop body.
- If condition is false initially, the loop body will not execute even once.

3) Example Program : Print numbers 1 to 5

```
class WhileExample {  
    public static void main(String[] args) {  
        int i = 1;           // initialization  
        while (i <= 5) {     // condition  
            System.out.println(i); // statement  
            i = i + 1;        // update  
        }  
    }  
}
```

Output :

1
2
3
4
5

Note :

If we forget to update the control variable, the loop will become infinite.

* Do - While Statement *

Introduction :

do-while loop is an exit-controlled loop in Java. It executes the block of statements at least once and then checks the condition.

Syntax :

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

Note :

Don't forget the semicolon (;)

Example :

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

Output :

1
2
3
4
5

Special Example :

```
int i = 10;  
do {  
    System.out.println(i);  
} while (i < 5);
```

Output :

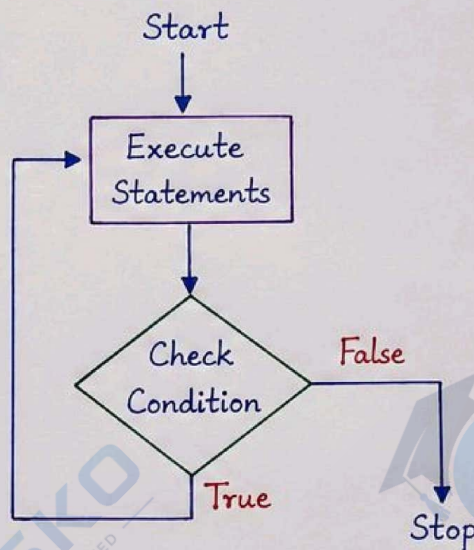
10

Even though condition is false, the block executes once.

Remember :

while → Check → Execute
do-while → Execute → Check

Flow Diagram :



Key Points :

- 1. The body of do-while loop executes at least once.
- 2. The condition is checked after the execution of the block.
- 3. If the condition is true, the statements are executed again.
- 4. If the condition is false, the loop terminates.
- 5. It is called an exit-controlled loop.

Difference between while and do-while :

Feature	while Loop	do-while Loop
Condition check	Checked first	Checked after execution
Execution	May execute 0 times	Executes at least 1 time
Type	Entry-controlled loop	Exit-controlled loop
Syntax	while (condition) { statements }	do { statements } while (condition);

* Use do-while loop when the statements must be executed at least once.

* For Statement *

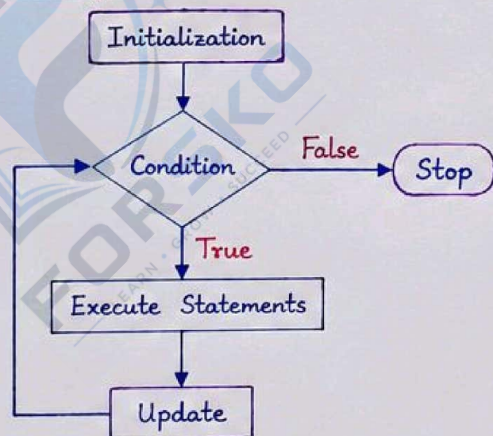
Introduction :

The for loop is an entry-controlled loop in Java. It is used when we know in advance how many times the loop should execute.

Syntax :

```
for ( initialization ; condition ; update ) {  
    // statements  
}
```

Flow Diagram :



Example :

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

Output :

1
2
3
4
5

Key Points :

- 1. Initialization is executed only once at the beginning.
- 2. Condition is checked before each iteration.
- 3. If condition is false, the loop terminates.
- 4. Update is executed after each iteration.
- 5. for loop is best when number of iterations is known.

Note :

Any or all the three parts (initialization, condition, update) of for loop are optional.

* Jumps in Loop *

In loops, we can change the normal flow of execution using break and continue statements.

• break Statement :

- It is used to terminate the loop immediately.
- Control comes out of the loop.

Example :

```
for ( int i = 1; i <= 5; i++ ) {  
    if ( i == 3 )  
        break;  
    System.out.print(i + " ");  
}
```

Output :

1 2

Loop stops
when i == 3

• continue Statement :

- It is used to skip the current iteration.
- Control goes to the next iteration.

Example :

```
for ( int i = 1; i <= 5; i++ ) {  
    if ( i == 3 )  
        continue;  
    System.out.print(i + " ");  
}
```

Output :

1 2 4 5

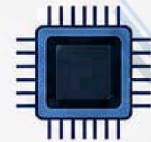
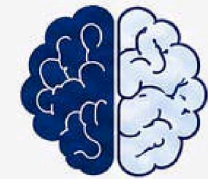
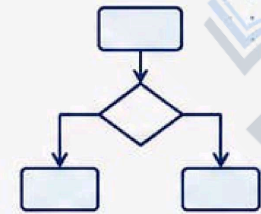
3 is skipped,
loop continues

Difference between break and continue :

Feature	<u>break</u>	<u>continue</u>
Meaning	Terminates the loop immediately.	Skips the current iteration only.
Control goes to	Comes out of the loop.	Next iteration.
Remaining Iterations	Not executed.	Executed.
Example Use	When a specific condition is met and we don't want to continue the loop.	When we want to skip some iterations based on a condition.

* Remember : Use break to exit the loop completely.
Use continue to skip and move to the next iteration.

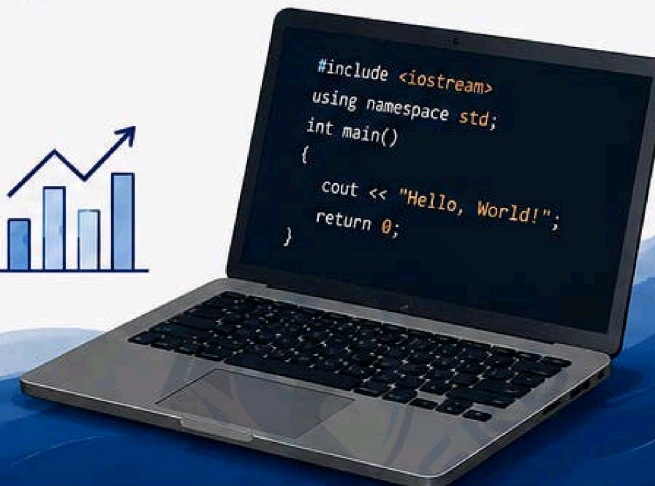
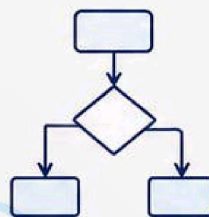
1010101010
0101010101
1010101010
0101010101



Unit - III



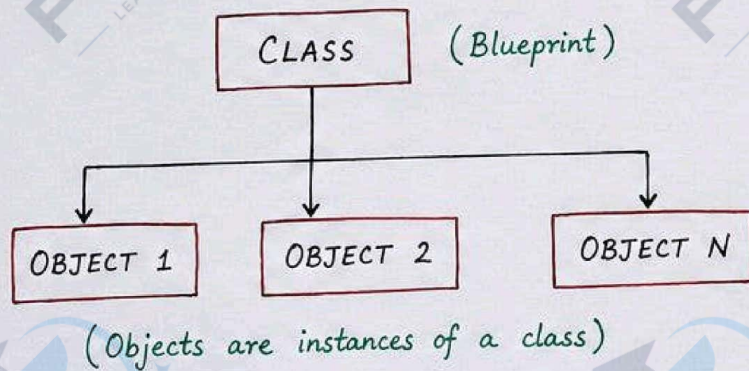
10110
01001
10101



CLASSES, OBJECTS AND METHODS: INTRODUCTION

1. INTRODUCTION

- Java is an object oriented programming language.
- The basic building blocks of Java program are Classes, Objects and Methods.
- Class is a blueprint or template which defines the properties (fields) and behaviors (methods) of objects.
- Object is an instance of a class.
- Method is a block of code which performs a specific task.



2. CLASS

- A class is a user defined data type which combines data members and member functions.

Syntax :

```
class ClassName {  
    // fields (variables)  
    // methods  
}
```

Example :

```
class Student {  
    int rollNo;  
    String name;  
  
    void display() {  
        System.out.println(rollNo + " " + name);  
    }  
}
```

3. OBJECT

- An object is a real world entity.
- It is an instance of a class.

Syntax :

```
ClassName objectName = new ClassName();
```

Example :

```
Student s1 = new Student();  
// Student → class  
// s1 → object
```

4. METHOD

- Method is a block of code which performs a specific task.

Syntax :

```
returnType methodName(parameters) {  
    // statements  
    return value; // if return type is not void  
}
```

Example :

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

NOTE : 1. Class is a logical entity.

2. Object is a physical entity.

3. Method describes the behavior of an object.

CLASSES, OBJECTS AND METHODS

1. DEFINING A CLASS

- A class is a blueprint or template used to create objects.
- It combines data (fields) and behavior (methods) in a single unit.

Syntax :

```
class ClassName {  
    // fields (variables)  
    // methods  
}
```

Example :

```
class Student {  
    int rollNo;      // field  
    String name;     // field  
  
    void display() { // method  
        System.out.println(rollNo + " " + name);  
    }  
}
```

Here, Student is a class.
rollNo and name are fields.
display() is a method.

2. DECLARATION OF FIELDS

- Fields are variables declared inside a class but outside any method.
- They represent the state or properties of an object.

Syntax :

```
dataType fieldName;
```

Example :

```
class Student {  
    int rollNo;      // integer field  
    String name;     // String field  
    double marks;    // double field  
}
```

Types of Fields :

1. Instance Field — Each object has its own copy.
2. Static Field — Shared by all objects of the class.
e.g. static int count;

3. METHODS

- A method is a block of code that performs a specific task.
- Methods define the behavior or actions of an object.

Syntax :

```
returnType methodName(parameters) {  
    // statements  
    return value; // if return type is not void  
}
```

Example :

```
class Calculator {  
    int add(int a, int b) { // method  
        int sum = a + b;  
        return sum;  
    }  
}
```

Calling the method :

```
Calculator c = new Calculator(); // object creation  
int result = c.add(10, 20);      // method call  
System.out.println(result);      // output: 30
```

Key Points :

- ✓ A class is defined using the class keyword.
- ✓ Fields store data of an object.
- ✓ Methods operate on the data and define the behavior.
- ✓ Together, fields and methods form a complete class.

CLASSES, OBJECTS AND METHODS

3. OBJECTS CREATION

- An object is an instance of a class.
- It represents a real world entity.
- Objects are created using the "new" keyword.

Syntax :

```
ClassName objectName = new ClassName();
```

↓ ↓ ↓
class reference creates object
name variable

Example :

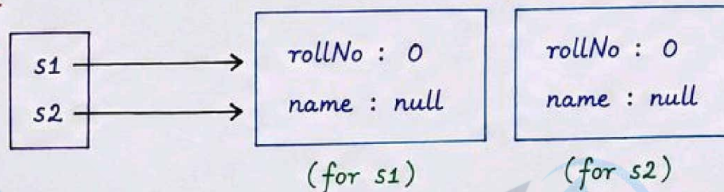
```
class Student {  
    int rollNo;  
    String name;  
}
```

```
Student s1 = new Student();  
Student s2 = new Student();  
// s1 and s2 are objects  
// of class Student
```

Memory Representation :

Objects in Heap memory

Reference
variables



Note : Each object gets its own copy of instance fields.

4. ACCESSING CLASS MEMBERS

- Class members (fields and methods) can be accessed using the dot (.) operator.
- The dot operator links the object with the member.

Syntax :

```
objectName.fieldName;                      // accessing field  
objectName.methodName(arguments);        // accessing method
```

Example :

```
class Student {  
    int rollNo;  
    String name;  
    void display() {  
        System.out.println(rollNo + " " + name);  
    }  
}
```

Accessing members :

```
Student s1 = new Student(); // object creation  
s1.rollNo = 101;                      // accessing field  
s1.name = "Rahul";                      // accessing field  
s1.display();                      // accessing method
```

Here,

- s1.rollNo and s1.name are used to access fields.
- s1.display() is used to call the method.

Important Points :

1. Fields hold data related to the object.
2. Methods perform operations on the data.
3. Both fields and methods are accessed using the object reference and the dot (.) operator.

CLASSES, OBJECTS AND METHODS

5. METHOD OVERLOADING

- Method overloading means having more than one method with the same name but different parameter list in the same class.
- It improves code readability.
- It is also known as compile time polymorphism.

Rules for Method Overloading :

1. Methods must have the same name.
2. Parameters must be different in number, type or order.
3. Return type may be same or different.
4. Changing only the return type does NOT cause overloading.

Example :

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

Explanation :

All three methods have the same name "add" but their parameter list is different.

Here,

- First method takes two integers.
- Second method takes three integers.
- Third method takes two double values.

Calling Overloaded Methods :

```
Calculator c = new Calculator();
```

```
int r1 = c.add(10, 20);           // calls first add(int, int)
int r2 = c.add(10, 20, 30);       // calls second add(int, int, int)
double r3 = c.add(12.5, 3.5);     // calls third add(double, double)
```

Key Point :

Compiler decides which method to call based on the arguments provided in the method call.

Important :

<u>Valid Overloading</u> (✓)	<u>Invalid Overloading</u> (✗)
<pre>int add(int a, int b) double add(int a, int b) ✓ int add(int a, int b, int c) int add(double a, double b)</pre>	<pre>int add(int a, int b) double add(int a, int b) // only return type changed</pre>

CLASSES, OBJECTS AND METHODS

6. CONSTRUCTOR

- A constructor is a special member of a class which is automatically called when an object is created.
- It has the same name as the class name.
- It does not have any return type (not even void).
- Its main purpose is to initialize the object.

Example :

```
class Student {  
    int rollNo;  
    String name;  
  
    Student() {  
        System.out.println("Constructor called");  
    }  
}
```

Usage :

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

↓
Constructor Student()
is called automatically.

Notes :

- Constructors are invoked automatically.
- If we do not write any constructor, Java provides a default constructor.

Difference between Method and Constructor :

<u>Basis</u>	<u>Method</u>	<u>Constructor</u>
1. Name	Can be any name.	Must be same as class name.
2. Return type	Must have return type.	Does not have return type.
3. Invocation	Called explicitly.	Called automatically at object creation.
4. Purpose	Performs a specific task.	Initializes the object.

* 7. TYPES OF CONSTRUCTOR

Constructors are of the following types :

(1) Default Constructor (No-argument Constructor provided by Java)

- If we do not write any constructor, Java provides a default constructor.

```
class Student {  
    int rollNo;  
    String name;  
    // Java provides default constructor Student()  
}
```

Note :

The default constructor has no parameters and an empty body.

(2) No-Argument Constructor (written by programmer)

- A constructor that has no parameters.

```
class Student {  
    Student() {  
        System.out.println("Student object created");  
    }  
}
```

Usage :

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

↓
No-argument constructor
is called.

(3) Parameterized Constructor

- A constructor that accepts parameters.

```
class Student {  
    int rollNo;  
    String name;  
    Student(int r, String n) {  
        rollNo = r;  
        name = n;  
    }  
}
```

Usage :

```
Student s1 = new Student(101, "Rahul");
```

↓
Parameterized constructor
is called and object is initialized.

Important Points :

1. A class can have multiple constructors (Constructor Overloading).
2. Constructors cannot be inherited.
3. Constructors are mainly used to initialize objects.

CLASSES, OBJECTS AND METHODS

8. CONSTRUCTOR OVERLOADING

- Constructor overloading means having more than one constructor with the same name but different parameter list in the same class.
- The constructors must differ in number, type or order of parameters.
- It improves code reusability and allows multiple ways to initialize objects of a class.

Example :

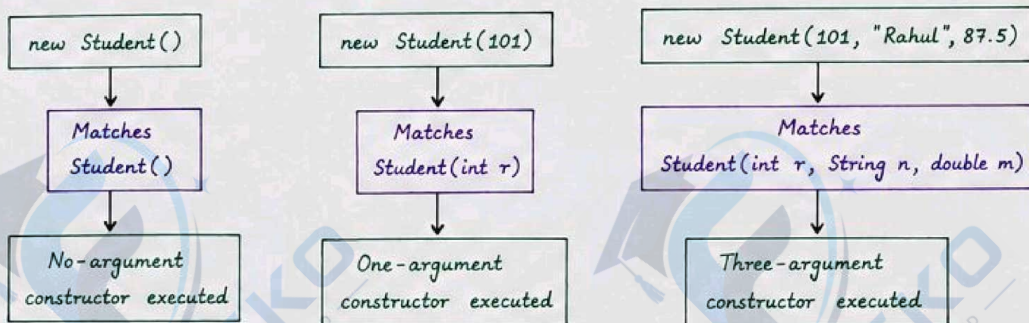
```
class Student {
    int rollNo;
    String name;
    double marks;

    // 1. No-argument constructor
    Student() {
        rollNo = 0;
        name = "Unknown";
        marks = 0.0;
        System.out.println("No-argument
        constructor called");
    }

    // 2. One-argument constructor
    Student(int r) {
        rollNo = r;
        name = "Unknown";
        marks = 0.0;
        System.out.println("One-argument constructor called");
    }

    // 3. Three-argument constructor
    Student(int r, String n, double m) {
        rollNo = r;
        name = n;
        marks = m;
        System.out.println("Three-argument constructor called");
    }
}
```

How Constructor Overloading works ?



Creating objects and calling constructors :

```
Student s1 = new Student(); // calls no-argument constructor
Student s2 = new Student(101); // calls one-argument constructor
Student s3 = new Student(101, "Rahul", 87.5);
// calls three-argument constructor
```

Output :

```
No-argument constructor called
One-argument constructor called
Three-argument constructor called
```

Key Points :

- Constructor overloading provides flexibility in object initialization.
- But changing only the return type does NOT cause constructor overloading.
- Constructors are called automatically at the time of object creation.
- It helps to set default values or accept different values when creating objects.

Important Note :

The Java compiler decides which constructor to call based on the number, types and order of arguments provided in the object creation statement.

CLASSES, OBJECTS AND METHODS

9. USE OF new KEYWORD

- The new keyword is used to allocate memory for an object at runtime.
- It creates an object and returns a reference to it.
- It also calls the constructor of the class.

Syntax :

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



Example :

```
class Student {
    int rollNo;
    String name;
    Student() {
        System.out.println("Object created");
    }
}
```

```
class Test {
    public static void main(String[] args) {
        Student s1 = new Student();
    }
}
```

Output : Object created

Key Points :

- Memory for the object is allocated in heap area.
- A reference variable is created in the stack area.
- The reference variable stores the address of the object.
- Without 'new', no memory is allocated.

10. USE OF delete

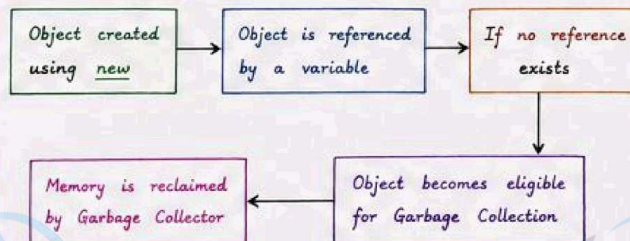
- Java does NOT support the delete keyword.
- In Java, memory is managed automatically by the Garbage Collector.
- Programmer cannot explicitly delete an object.
- An object becomes eligible for garbage collection when there are no reference variables pointing to it.

Example :

```
class Student {
    int rollNo;
}

public class Test {
    public static void main(String[] args) {
        Student s1 = new Student(); // object created
        s1.rollNo = 101;
        s1 = null; // eligible for GC
    }
}
```

How Garbage Collection works ?



Note : Java automatically deletes objects that are no longer in use.

11. USE OF this KEYWORD

- this is a reference variable that refers to the current object.
- It is used to resolve naming conflicts between instance variables and local variables.
- It can be used to call current class method and constructors.

Example 1 : Referring current object's variables

```
class Student {
    int rollNo;
    String name;
    Student(int rollNo, String name) {
        this.rollNo = rollNo; // this → instance variable
        this.name = name; // rollNo, name → parameters
    }
}
```

Example 2 : Calling method

```
class Student {
    void display() {
        System.out.println("Hello");
    }
    void show() {
        this.display(); // calls //current method
    }
}
```

Example 3 : Calling constructor

```
class Student {
    Student() {
        this(101, "Rahul"); // calls //parameterized //constructor
    }
    Student(int rollNo, String name) {
        System.out.println(rollNo + " " + name);
    }
}
```

Summary of this :

1. Refers to the current object.
2. Resolves naming conflicts.
3. Can be used to call current class methods.
4. Can be used to call another constructor in the same class.

CLASSES, OBJECTS AND METHODS

12. ACCESS SPECIFIER : public, private & protected

- Access specifiers are used to set the visibility and accessibility of classes, fields, methods and constructors.
- Java provides four types of access specifiers :
public, private, protected and default (no specifier).

Access Levels :

Access Specifier	Same Class	Same Package	Subclass (Different Package)	Different Package (Non-Subclass)	Explanation
public	✓	✓	✓	✓	Accessible everywhere.
protected	✓	✓	✓	✗	Accessible in same package and in subclasses (even if different package).
default (no specifier)	✓	✓	✗	✗	Accessible only within the same package.
private	✓	✗	✗	✗	Accessible only within the same class.

Examples :

1. public

```
class A {  
    public int data = 10;  
    public void show() {  
        System.out.println("Hello");  
    }  
}
```

Can be accessed in any class anywhere.

2. private

```
class A {  
    private int data = 10;  
    private void show() {  
        System.out.println("Hello");  
    }  
}
```

Can be accessed only within the same class.

3. protected

```
package p1;  
class A {  
    protected int data = 10;  
    protected void show() { }  
}
```

Can be accessed in the same package and by subclasses in other packages.

4. default (no specifier)

```
package p1;  
class A {  
    int data = 10; // no specifier  
    void show() { }  
}
```

Can be accessed only in the same package.

Rules :

- If no access specifier is written, it is treated as default.
- Private members cannot be accessed directly outside the class.
- Protected members can be accessed within the same package and by subclasses in other packages.
- Public members can be accessed from anywhere.

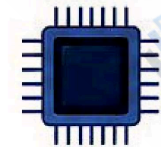
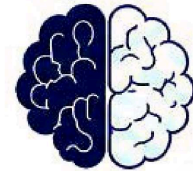
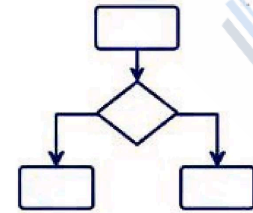
Key Points :

- Use private for data hiding.
- Use protected when inheritance is needed.
- Use public when members need to be accessed everywhere.
- Default access provides package-level access.

Real-time Analogy :

- private → Like a personal diary (only you can read it).
- default → Like room access within your house (only family members).
- protected → Like access for family + relatives.
- public → Like information on a public notice board (anyone can see it).

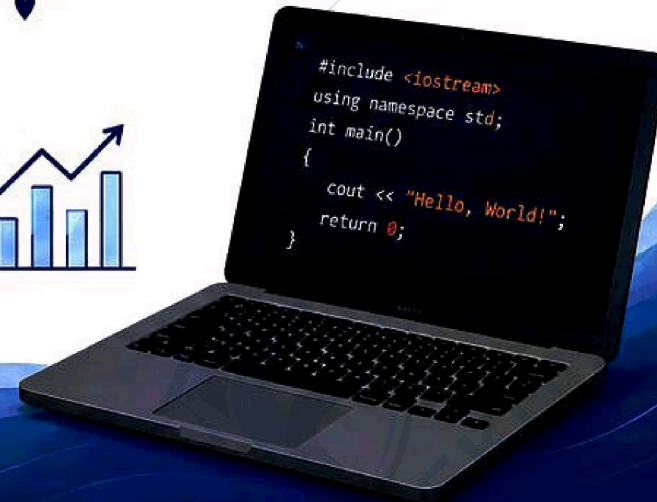
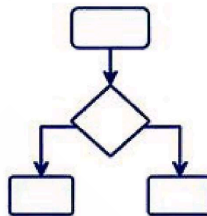
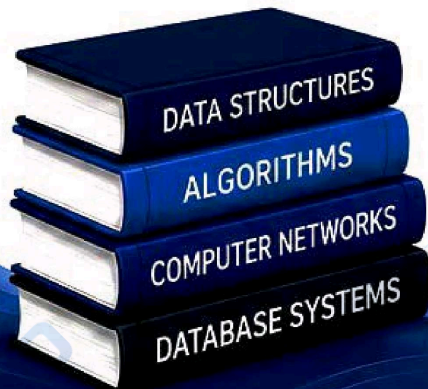
1010101010
0101010101
1010101010
0101010101



Unit - IV



10110
01001
10101



INHERITANCE : INTRODUCTION

- Inheritance is a mechanism in Java by which one class acquires the properties and methods of another class.
- The class whose members are inherited is called **Super class** (Parent class / Base class).
- The class that inherits the members of another class is called **Sub class** (Child class / Derived class).
- It helps in code reusability and establishes a natural IS-A relationship.

Syntax :

```
class SubClass extends SuperClass
{
    // members
}
```

Example :

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

Here,

- Animal is **Super class** (Parent class).
- Dog is **Sub class** (Child class).

Since Dog extends Animal, it inherits **eat()** method from Animal.

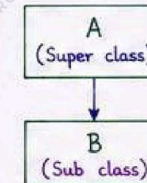
Note : Java does not support multiple inheritance using classes to avoid ambiguity.

TYPES OF INHERITANCE IN JAVA

Java supports the following types of inheritance.

1. Single Inheritance :

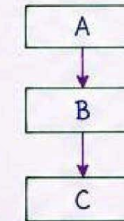
One class inherits another class.



```
class A { }
class B extends A { }
```

2. Multilevel Inheritance :

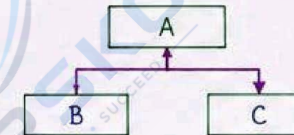
A class inherits from a class which is also derived from another class.



```
class A { }
class B extends A { }
class C extends B { }
```

3. Hierarchical Inheritance :

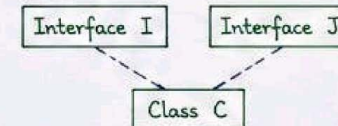
Two or more classes inherit the same super class.



```
class A { }
class B extends A { }
class C extends A { }
```

4. Multiple Inheritance (through interfaces) :

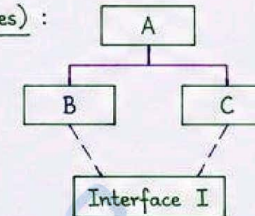
A class implements multiple interfaces.



```
interface I { }
interface J { }
class C implements I, J { }
```

5. Hybrid Inheritance (through interfaces) :

Combination of two or more types of inheritance.



```
class A { }
class B extends A { }
class C extends A { }
interface I { }
class D implements I { }
```

Note : Multiple and Hybrid inheritance are possible only through Interfaces.

Super & Extended class

- The class whose properties and methods are inherited by another class is called **Super class (Parent class)**.
- The class that inherits the properties and methods of another class is called **Extended class (Child class / Sub class)**.

Syntax :

```
class SubClass extends SuperClass
{
    // body
}
```

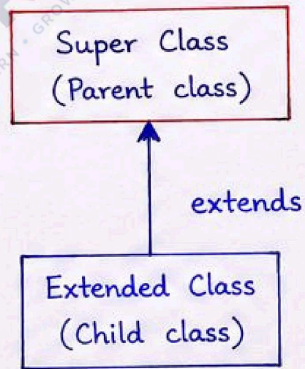
Example :

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

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

// Animal → Super class (Parent class)
// Dog → Extended class (Child class)

Relation :



Example with objects :

```
Animal a = new Animal();
Dog d = new Dog();
```

```
d.eat(); // Inherited method
d.bark(); // Own method
```

Note :

- A child class can access members of parent class.
- Java supports Single inheritance with classes using **extends** keyword.

Final Variable

- A variable declared with **final** keyword is called final variable.
- The value of a final variable cannot be changed once it is initialized.
- It acts as a **constant**.

Syntax :

```
final dataType variableName = value;
```

Example :

```
class Test {
    final int MAX = 100; // final variable
    void show() {
        System.out.println("Max value = " + MAX);
        // MAX = 200; // Error: cannot assign
                       // to final variable
    }
}
```

Important Points :

- It must be initialized at the time of declaration.
- If not initialized at declaration, it must be initialized in constructor.
- By convention, final variables are written in **UPPER_CASE**.

Final Method & Final Class

1) Final Method

- A method which is declared with final keyword is called **final method**.
- A final method cannot be overridden by the subclass.
- It is used to prevent the logic of a method from being changed.

Syntax :

```
final return_type methodName()  
{  
    // body  
}
```

Example :

```
class Parent {  
    final void show() {  
        System.out.println("This is final method");  
    }  
}  
  
class Child extends Parent {  
    // trying to override  
    void show() { // Error  
        System.out.println("Cannot override");  
    }  
}
```

Note :

- final keyword with method **prevents** it from being modified in the child class.

2) Final Class

- A class which is declared with final keyword is called **final class**.
- A final class cannot be inherited (extended).
- It is used to restrict inheritance.

Syntax :

```
final class ClassName  
{  
    // body  
}
```

Example :

```
final class Parent {  
    void display() {  
        System.out.println("I am final class");  
    }  
}  
  
class Child extends Parent { // Error  
    // Cannot inherit final class  
}
```

Important Points :

- A final class cannot have any subclasses.
- All methods of final class are implicitly final.
- **Ex:** String, Integer, System classes are final.

Key Difference :

Final Method

1. Applied on method.
2. Cannot be overridden.
3. Inheritance is allowed.

Final Class

1. Applied on class.
2. Cannot be inherited.
3. Inheritance is not allowed.

Abstract Methods and Classes

1) Abstract Methods

- A method that is declared without implementation is called **abstract method**.
- It is declared using the **abstract** keyword.
- It must be implemented in the subclass (concrete class).
- It cannot be **private**, **final** or **static**.

Syntax :

```
abstract return_type methodName(parameters);  
// no body
```

Example :

```
abstract class Shape {  
    abstract double area(); // abstract method  
}
```

Example with implementation :

```
abstract class Shape {  
    abstract double area();  
    void display() {  
        System.out.println("This is Shape");  
    }  
}
```

```
class Circle extends Shape {  
    double radius;  
    Circle(double r) { radius = r; }  
    double area() {  
        return 3.14 * radius * radius;  
    }  
} // provides implementation
```

Note :

- If a class contains one or more abstract methods, the class must be declared **abstract**.
- If a subclass does not provide implementation of abstract method, it must be declared **abstract**.

Illegal :

- × private abstract void m();
- × final abstract void m();
- × static abstract void m();

2) Abstract Classes

- A class that is declared with abstract keyword is called **abstract class**.
- It cannot be instantiated (object cannot be created).
- It can have abstract methods (without body) as well as non-abstract methods (with body).
- It can have constructors and variables.
- It is used to provide a common base and partial implementation.

Syntax :

```
abstract class ClassName {  
    // abstract method(s)  
    // non-abstract method(s)  
}
```

Example :

```
abstract class Animal {  
    abstract void sound(); // abstract method  
    void eat() {  
        System.out.println("Eating...");  
    }  
}
```

Example with subclass :

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

Test class :

```
class Test {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        d.sound(); // Bark  
        d.eat(); // Eating...  
    }  
}
```

Key Points :

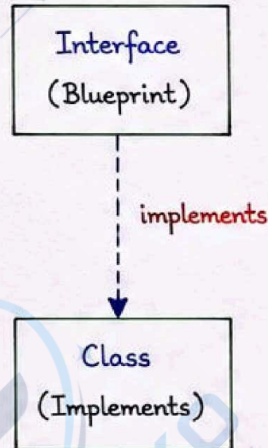
- **Abstract class** cannot be instantiated.
- It may contain 0 to 100% abstract methods.
- Abstract class can have constructors, fields and both abstract and concrete methods.
- Subclass must implement abstract methods or must be declared **abstract**.

Illegal :

```
Animal a = new Animal(); // Error  
  
// abstract class cannot  
// be instantiated
```


Interfaces : Introduction

- An interface is a reference type in Java.
- It is a **blueprint** or template which contains abstract methods and constants.
- It is used to achieve **abstraction** and **multiple inheritance** in Java.
- The interface is declared using the '**interface**' keyword.
- A class implements an interface using the '**implements**' keyword.
- It defines a contract that implementing classes must follow.



Need of Interface :

1. To achieve full abstraction (100% abstraction).
2. To support multiple inheritance.
3. To achieve loose coupling.
4. To provide a common specification for classes.

Key Points :

- All methods in an interface are **public** and **abstract** by default.
- All variables in an interface are **public**, **static** and **final** by default.
- Interface cannot be instantiated.
- A class implementing an interface must provide implementation for all the methods of that interface.
- Interface can extend another interface.

Defining Interfaces

Syntax :

```
interface InterfaceName {  
    // constants  
    // abstract methods  
}
```

By default,

- methods are **public** & **abstract**
- variables are **public**, **static** & **final**

Rules for Defining Interfaces :

1. Interface is declared using '**interface**' keyword.
2. It can contain only abstract methods and constants.
3. It cannot contain instance fields.
4. It can contain static methods (with body) since Java 8.
5. It can contain default methods (with body) since Java 8.
6. It can contain private methods since Java 9.

Example :

```
interface Vehicle {  
    // constants  
    int MAX_SPEED = 120; // public static final by default  
  
    // abstract methods  
    void start();  
    void stop();  
}
```

} Methods are
public & abstract
by default

Note :

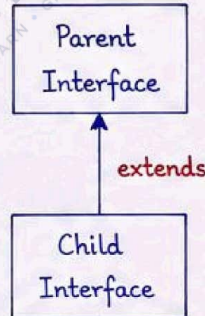
An interface only declares methods. The implementation is provided by the class that implements the interface.

1) EXTENDING INTERFACES

- An interface can extend another interface using the **extends** keyword.
- The child interface inherits all the abstract methods of the parent interface.

Syntax :

```
interface ChildInterface extends ParentInterface
{
    // new abstract methods
}
```



Example 1: (Single Interface Extension)

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

// Parent Interface

```
interface B extends A {
    void display();
}
```

// Child Interface

Example 2: (Multiple Interface Extension)

```
interface X {
    void x();
}
```

```
interface Y {
    void y();
}
```

```
interface Z extends X, Y {
    void z();
}
```

Note :

An interface can extend multiple interfaces.
(Multiple Inheritance of type)

Key Points :

- Interfaces can extend one or more interfaces.
- Child interface inherits all abstract methods of parent interfaces.
- Interfaces cannot extend classes.
- There can be any number of levels of interface inheritance.

Example :

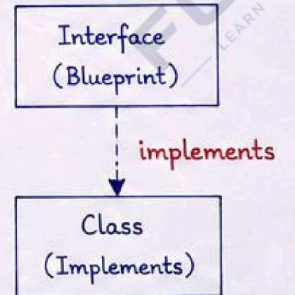
```
A
↓
B extends A
↓
C extends B
↓
D extends C
```

2) IMPLEMENTING INTERFACES

- A class implements an interface using the **implements** keyword.
- The class must provide implementation for all the abstract methods of the interface.

Syntax :

```
class ClassName implements InterfaceName
{
    // implementation of abstract methods
}
```



Example 1: (Implementing a single interface)

```
interface Animal {
    void sound();
}
```

```
class Dog implements Animal {
    public void sound() {
        System.out.println("Barking...");
    }
}
```

// Dog class implements Animal interface

Example 2: (Implementing multiple interfaces)

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

```
interface B {
    void b();
}
```

```
class MyClass implements A, B {
    public void a() {
        System.out.println("Method A");
    }
    public void b() {
        System.out.println("Method B");
    }
}
```

// MyClass implements both A and B interfaces

Important Points :

- A class must implement all abstract methods of the interface.
- A class can implement more than one interface, separated by comma.
- A class can extend another class and implement one or more interfaces simultaneously.
- An interface cannot be instantiated, but implementing class objects can be created.

Example :

```
MyClass obj = new MyClass();
// Valid
```