Module 1– Core Java

1. Introduction to Java

1.1 History Of Java.

Answer : Java was developed in the early 1990s by James Gosling and his team at Sun Microsystems as part of the “Green Project,” with the goal of creating a platform-independent programming language for consumer electronics. Originally called *Oak*, it was later renamed *Java* in 1995 and officially released with the slogan “Write Once, Run Anywhere,” made possible by the Java Virtual Machine (JVM). Java quickly gained popularity for web and enterprise applications due to its object-oriented nature, strong memory management, security features, and rich standard libraries. Over the years, Java evolved significantly with regular updates, becoming a core technology for enterprise systems, mobile applications (especially Android), web services, and large-scale distributed systems. In 2010, Oracle Corporation acquired Sun Microsystems and became the steward of Java, continuing its development and making it one of the most widely used programming languages in the world.

1.2 Features of Java.

Answer : Features of Java :

1. Simple – Java has an easy-to-learn syntax and removes complex features like pointers and operator overloading.

2. Object-Oriented – Java follows OOP concepts such as inheritance, encapsulation, abstraction, and polymorphism.

3. Platform Independent – Java programs run on any system with a JVM using the principle *Write Once, Run Anywhere*.

4. Secure – Java provides a secure runtime environment using bytecode verification, class loaders, and no direct memory access.

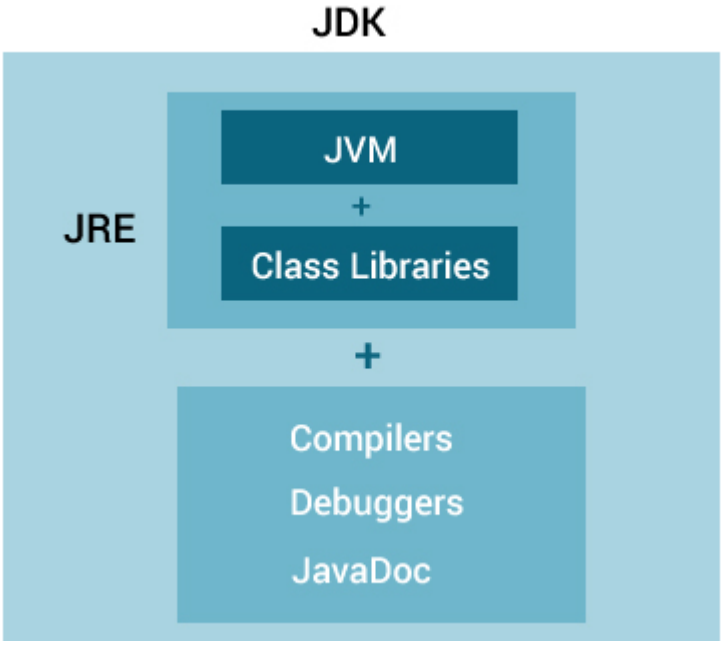
5. Robust – Strong memory management, exception handling, and garbage collection make Java reliable.

6. Multithreaded – Java supports multithreading, allowing concurrent execution for better performance.

7. High Performance – Uses Just-In-Time (JIT) compiler to improve execution speed.

1.3 Understanding JVM,JRE, and JDK.

Answer:



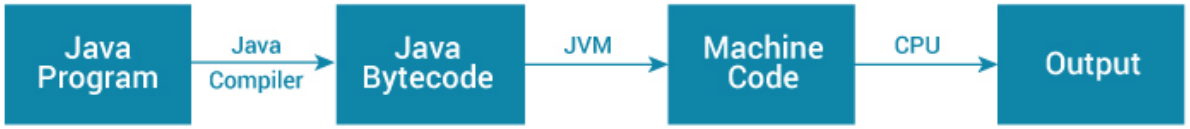
JVM:

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, the Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.

Working Of Java Program :



JRE (Java Runtime Environment) is a software package that provides Java class libraries (java.lang, java.util, etc.), Java Virtual Machine (JVM), and other components that are required to run Java applications.Used by end users. Cannot compile Java code.Only executes compiled `.class` files.

JRE is the superset of JVM.

JDK:

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.
In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).

1.3 Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ).

Step 1: Install JDK

1. Download JDK (Java Development Kit) from Oracle or OpenJDK.
2. Install it on your system.
3. Set environment variables.

Verify Installation:

java -version

javac -version

IntelliJ IDEA

Steps:

1. Download IntelliJ IDEA Community Edition.
2. Install and open it.
3. New Project → Java → Select JDK.
4. Create Class → `main` method.

1.4 Java Program Structure (Packages, Classes, Methods).

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

1. Package

- Used to group related classes
- Helps in code organization and access control

  ex...package com.example.demo;

2.Class

- Blueprint of an object
- All Java code must be inside a class

  ex...public class HelloWorld {}

3.Methods

- A block of code that performs a specific task
- Improves reusability and readability

  ex...public static void main(String[] args) {}

4.Statements

- Instructions executed by JVM

  ex... System.out.println("Hello, Java!");

2. Data Types, Variables, and Operators

2.1 Primitive Data Types in Java (int, float, char, etc.)

In Java, Primitive Data Types are the basic data types that store simple values directly in memory (not objects). Java has 8 primitive data types, grouped into integer, floating-point, character, and boolean types.

1. Integer Data Types

Used to store **whole numbers** (positive or negative, without decimals).

Data Type	Size	Range
byte	1 byte	-128 to 127
short	2 byte	-32768 to -32767
int	4 byte	-2^{31} to $2^{31}-1$
long	8 byte	-2^{63} to $2^{63}-1$

Example:

byte a = 10;

short b = 200;

int c = 100000;

long d = 9876543210L;

2. Floating-Point Data Types

Used to store **decimal (fractional) numbers**.

Data Type	Size	Precision
float	4 byte	~7 decimal digits
double	8 byte	~15 decimal digits

Example:

float pi = 3.14f;

double value = 12345.6789;

3. Character Data Type

Used to store **single characters** (letters, digits, symbols).

Data Type	Size	Description
char	2 byte	Store Unicode characters

Example:

char grade = 'A';

char symbol = '@';

4.Boolean Data Type

Used to store **true or false values**.

Data Type	Size	Values
boolean	1 bit	true,false

Example:

boolean isJavaEasy = true;

2.2 Variable Declaration and Initialization.

1.Variable Declaration

Declaration means specifying the data type and variable name.

Syntax:

dataType variableName;

Example:

int number;

double price;

char grade;

boolean isActive;

2.Variable Initialization

Initialization means assigning a **value** to a variable.

Syntax:

variableName = value;

Example:

number = 10;

price = 99.99;

grade = 'A';

isActive = true;

- Rules for Variable Declaration in Java
  - A variable name Must start with a letter (a–z or A–Z), underscore (_), or dollar sign ($). Cannot start with a digit.
  - Variable names cannot be Java reserved keywords.
  - Java is **case-sensitive**, so different cases mean different variables.
  - Except `_` and `$`, special characters are not allowed.
  - Local Variables Must Be Initialized Before Use.

2.3 Operators:Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise.

Operators are special symbols used to perform operations on variables and values.

Java mainly uses the following operators:

1.Arithmetic Operators

Used to perform mathematical calculations.

Operator	Description	Example
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	a*b
/	division	a/b
%	Modulus	a%b

2.Relational (Comparison) Operators

Used to compare two values and return a boolean result.

Operator	Description	Example
==	Equal to	a==b
!=	Not Equal to	a!=b
>	Greater than	a>b
<	Less than	a<b
>=	Greater than or equal	a>=b
<=	Less than or equal	a<=b

3.Logical Operators

Used to combine multiple conditions.

Operator	Name	Description	Example
&&	Logical and	Return true if both statement are true	x<5 && x<10
||	Logical or	Return true if one of the statement are true	x<5 || x<4
!	Logical not	Reverse the result,return false if the result is true.	!(x<5 && x<10)

4.Assignment Operators

Used to assign values to variables.

Operator	Example	Meaning
=	a=10	Assign
+=	a+=5	a=a+5
-+	a-=3	a=a-3
*=	a*=2	a=a*2
/=	a/=7	a=a/7
%=	a%=4	a=a%4

5.Unary Operators

Operate on a single operand.

Operator	Description
+	Unary plus
-	Unary minus
++	Increment
- -	Decrement

6.Bitwise Operators

Used to perform operations bit by bit.

Operator	Description
&	Bitwise And
|	BitWise Or
^	Bitwise X-OR
~	Bitwise Complement
<<	Left Shift
>>	Right Shift

2.4 Type Conversion and Type Casting.

1.Type Conversion (Widening / Implicit Casting)

This conversion is done automatically by the compiler when:

- Converting smaller data type → larger data type
- No data loss occurs

Order of Widening Conversion:

byte → short → int → long → float → double

Example :

```
int a = 10;

double b = a;   // implicit conversion

System.out.println(b); // 10.0
```

2.Type Casting (Narrowing / Explicit Casting)

This conversion is done manually by the programmer when:

- Converting larger data type → smaller data type
- Data loss may occur

Example:

```
double x = 10.75;

int y = (int) x;   // explicit casting

System.out.println(y); // 10
```

3. Control Flow Statements

3.1 If-Else Statements

The if–else statement is a decision-making control statement in Java.

It allows the program to execute different blocks of code based on conditions.

1.Simple `if` Statement

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

Syntax:

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

2.if–else Statement

Used when there are two possible outcomes.

Syntax:

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

3. `if–else if–else` Ladder

Used when there are **multiple conditions**.

Syntax:

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

4.**Nested `if-else`**

An `if-else` inside another `if-else`.

Syntax:

```
if (condition1) {
    // executes when condition1 is true

    if (condition2) {
        // executes when condition2 is true
    } else {
        // executes when condition2 is false
    }

} else {
    // executes when condition1 is false

    if (condition3) {
        // executes when condition3 is true
    } else {
        // executes when condition3 is false
    }
}
```

3.2 Switch Case Statements.

The `switch` statement is a selection (decision-making) control statement used when you have multiple fixed choices based on a single expression.

Syntax of `switch` Statement:

```
switch (expression) {
    case value1:
        // statements
        break;

    case value2:
        // statements
        break;

    case value3:
        // statements
        break;

    default:
        // statements
```

How `switch` Works

1. The `expression` is evaluated once
2. Its value is matched with `case` values
3. Matching case executes
4. `break` stops execution
5. If no case matches, `default` executes

3.3 Loops(For, While, Do-While)

Loops are used to execute a block of code repeatedly as long as a condition is true**.**

1.`for` Loop

Used when the **number of iterations is known**.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // loop body
}
```

2.`while` Loop

Used when the number of iterations is not fixed and depends on a condition.

Syntax:

```
while (condition) {
    // loop body
}
```

3.`do-while` Loop

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

Syntax:

```
do {
    // loop body
} while (condition);
```

3.4 Break and Continue Keywords.

1.break Keyword

Used to terminate the loop immediately.

Example:

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

Output: 1 2 3 4

2.Continue Keyword

Used to skip the current iteration and move to the next iteration.

Example:

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

Output: 1 2 4 5

4.Classes and Objects

4.1 Defining a Class and Object in Java

In Java, Class and Object are the core concepts of Object-Oriented Programming (OOP).

1.Class in Java

A **class** is a **blueprint or template** used to create objects. It defines:

- Variables (data members / fields)
- Methods (behavior)

Syntax to Define a Class:

```
class ClassName {

    // variables

    // methods

}
```

2.Object in Java

An object is a real-world instance of a class. Objects store actual values and can access class members.

Syntax to Create an Object:

ClassName objectName = new ClassName();

3.Memory Representation:

- Class → no memory allocated for variables
- Object → memory allocated in heap
- Reference variable → stored in stack

4.2 Constructors and Overloading

1.Constructors in Java

A constructor is a special method used to initialize objects.

Features of Constructors:

- Name is same as class name
- No return type (not even `void`)
- Automatically called when an object is created
- Used to initialize instance variables

Example:

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

    Student() {   // default constructor
        id = 0;
        name = "Unknown";
    }
}
```

2.Constructor Overloading

When a class has more than one constructor with different parameter lists, it is called constructor overloading.

Rules:

- Different number of parameters OR
- Different type of parameters

Example:

```
class Student {
    int id;
    String name;
    Student() {
        id = 0;
        name = "NA";
    }
    Student(int i) {
        id = i;
        name = "NA";
    }
    Student(int i, String n) {
        id = i;
        name = n;
    }
}
```

4.3 ObjectCreation, Accessing Members of the Class.

Objects are created using the `new` keyword.

Syntax:

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

Accessing Members of a Class:

Class members (variables and methods) are accessed using the dot (`.`) operator.

Example:

s2.id = 101;

s2.name = "Aman";

s2.display();

4.4 this Keyword.

The `this` keyword refers to the current object.

Distinguish Instance and Local Variables:

```
class Student {
    int id;
    Student(int id) {
        this.id = id; // this.id → instance variable
    }
}
```

`this()` must be the first statement in a constructor.

Pass Current Object as Parameter:

```
void show(Student s) {
    System.out.println(s.id);
}

void call() {
    show(this);
}
```

5.Methods in Java

5.1 Defining Methods in Java

A method is a block of code that performs a specific task and is executed when it is called.

Syntax:

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

5.2 Method Parameters and Return Types

- **Method Parameters**

Parameters are values passed to a method when it is called.

- **Return Type**

A method can return a value using the `return` keyword.
If it does not return anything, use `void`.

Example with Parameters and Return:

```
class Calculation {
    int add(int a, int b) {
        return a + b;
    }
    public static void main(String[] args) {
        Calculation c = new Calculation();
        int result = c.add(10, 20);
        System.out.println(result);
    }
}
```

5.3 Method Overloading

Method Overloading means defining multiple methods with the same name but different parameters.

✔ Rules

- Same method name
- Different number or type of parameters
- Return type alone cannot change

Example:

```
class MathOperation {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
    int add(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String[] args) {
        MathOperation m = new MathOperation();
        System.out.println(m.add(10, 20));
        System.out.println(m.add(10.5, 20.5));
        System.out.println(m.add(1, 2, 3));
    }
}
```

5.4 Static Methods and Variables

- Static Variables
  - Belong to the class, not objects
  - Memory is allocated only once
- Static Methods
  - Can be called without creating an object
  - Can access only static data directly

Example:

```
class Student {
    static String college = "TOPS";
    int id;
    String name;
    static void changeCollege() {
        college = "TOPS Technologies";
    }
    void display() {
        System.out.println(id + " " + name + " " + college);
    }
    public static void main(String[] args) {
        Student.changeCollege();
        Student s1 = new Student();
        s1.id = 1;
        s1.name = "Amit";
        s1.display();
    }
}
```

6.Object-Oriented Programming (OOPs) Concepts

6.1 BasicsofOOP:Encapsulation, Inheritance, Polymorphism, Abstraction

1.Encapsulation

Encapsulation means wrapping data (variables) and methods together into a single unit (class) and restricting direct access to data.

✔ Achieved by:

- Making variables `private`
- Providing `public` getters and setters

Example:

```
class Student {
    private int id;
    private String name;
    public void setId(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
}
```

2.Inheritance

Inheritance allows one class to acquire properties and behavior of another class using `extends`.

Example:

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

3.Polymorphism

Polymorphism means one name, many forms.

Types:

- Compile-time (Method Overloading)
- Run-time (Method Overriding)

Compile-time Polymorphism:

Method call is resolved **at compile time** using **method overloading**.

Example:

```
class Demo {
    void add(int a, int b) {
        System.out.println(a + b);
    }
    void add(double a, double b) {
        System.out.println(a + b);
    }
}
```

Run-time Polymorphism:

Method call is resolved at runtime using method overriding and dynamic method dispatch.

Example:

```
class A {
    void show() {
        System.out.println("A class");
    }
}
class B extends A {
    void show() {
        System.out.println("B class");
    }
    public static void main(String[] args) {
        A obj = new B();
        obj.show();
    }
}
```

4.Abstraction

Abstraction means hiding implementation details and showing only essential features.

✔ Achieved by:

- Abstract class
- Interface

Example:

```
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

6.2 Inheritance: Single, Multilevel, Hierarchical

1.Single Inheritance

One child class inherits from one parent class.

```
class A {
    void showA() { }
}
class B extends A {
    void showB() { }
}
```

2.Multilevel Inheritance

Inheritance in multiple levels.

```
class A {
    void showA() { }
}
```

```
class B extends A {
    void showB() { }
}
class C extends B {
    void showC() { }
}
```

3.Hierarchical Inheritance

Multiple child classes inherit from one parent class.

```
class A {
    void showA() { }
}
class B extends A {
    void showB() { }
}
class C extends A {
    void showC() { }
}
```

6.3 MethodOverriding and DynamicMethodDispatch

🔷 Method Overriding

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

✔ Rules

- Same method name
- Same parameters
- IS-A relationship

Example:

```
class Bank {
    int getRateOfInterest() {
        return 5;
    }
}
class SBI extends Bank {
    int getRateOfInterest() {
        return 7;
    }
}
```

🔹 **Dynamic Method Dispatch**

Dynamic Method Dispatch is runtime polymorphism where the method call is resolved at runtime based on the object.

✔ Achieved using:

- Parent class reference
- Child class object

Example:

```
class Bank {
    void interest() {
        System.out.println("Bank Interest");
    }
}
class HDFC extends Bank {
    void interest() {
        System.out.println("HDFC Interest");
    }
}
public class Test {
    public static void main(String[] args) {
        Bank b;
        b = new HDFC();   // Parent reference, child object
        b.interest();    // Calls HDFC method
    }
}
```

7.Constructors and Destructors

7.1 Constructor Types (Default, Parameterized)

🔹 Constructor in Java

A constructor is a special method used to initialize objects.

- Same name as class
- No return type
- Called automatically when an object is created

Default Constructor

A constructor with **no parameters**.
If not defined, Java provides one automatically.

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

Parameterized Constructor

A constructor that accepts **parameters** to initialize data.

```
class Student {
    int id;
    Student(int id) {
        this.id = id;
    }
}
```

7.2 Copy Constructor (Emulated in Java)

Java does not have a built-in copy constructor, but we can emulate it by passing an object as a parameter.

```
class Student {
    int id;
    Student(Student s) {
        this.id = s.id;
    }
}
```

7.3 Constructor Overloading

Defining multiple constructors in a class with different parameters.

```
class Student {
    int id;
    String name;
    Student() {
        id = 0;
        name = "NA";
    }
    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

7.4 ObjectLife Cycle and Garbage Collection

The Object Life Cycle describes the stages an object goes through:

1. Creation – using new keyword
2. In Use – object is accessed
3. Eligible for GC – no reference to object
4. Garbage Collection – memory reclaimed

```
Student s = new Student(); // creation
s = null;                  // eligible for GC
```

Garbage Collection:

Garbage Collection is an automatic memory management process in Java where the JVM identifies and removes unused objects from memory to free heap space.

=> An object becomes garbage when no reference points to it.

```
Student s = new Student();
s = null;   // object becomes eligible for GC
```

🔹 Java Memory Structure (Important for GC)

JVM Memory Areas:

- Heap Memory → Objects are stored here (GC works here)
- Stack Memory → Local variables & method calls
- Method Area → Class metadata
- PC Register & Native Stack

=> GC works ONLY on Heap Memory

8.Arrays And Strings

8.1One-Dimensional and Multidimensional Arrays

- ### One-Dimensional Array

  ➢ A one-dimensional array stores multiple values of the same type in a single linear sequence.

Example:

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

- **Multidimensional Array**

  ➢ A multidimensional array stores data in row-column (table) form using arrays inside arrays.

Example:

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

8.2 StringHandling in Java: String Class, StringBuffer, StringBuilder

- String Handling in Java

  ➢ String handling deals with creation, manipulation, and storage of character sequences.

- String Class

  ➢ String is immutable, meaning once created its value cannot be changed.

  Ex.. String s = "Java";

- StringBuffer

  ➢ StringBuffer is mutable and thread-safe, used in multi-threaded environments.

  Ex.. StringBuffer sb = new StringBuffer("Java");

- StringBuilder

  ➢ StringBuilder is mutable but not thread-safe, and faster than StringBuffer.
  Ex.. StringBuilder sb = new StringBuilder("Java");

8.3 Array of Objects

An array of objects stores multiple object references of the same class.

Ex.. Student s[] = new Student[3];

8.4 StringMethods (length, charAt, substring, etc.)

- `length()` → **Returns total number of characters in a string**
  - **Ex..**System.out.println("Java".length()); // 4
- `charAt()` → **Returns character at a specific index**
  - **Ex..**System.out.println("Java".charAt(1)); // a
- `substring()` → **Extracts part of a string**
  - **Ex..**System.out.println("Programming".substring(0,4)); // Prog
- `equals()` → **Compares content of two strings**
  - **Ex..**System.out.println("Java".equals("java")); // false
- `equalsIgnoreCase()` → **Compares strings ignoring case**
  - **Ex..**System.out.println("Java".equalsIgnoreCase("java")); // true
- `toUpperCase()` → **Converts string to uppercase**
  - **Ex..**System.out.println("java".toUpperCase()); // JAVA
- `toLowerCase()` → **Converts string to lowercase**
  - **Ex..**System.out.println("JAVA".toLowerCase()); // java
- `trim()` → **Removes leading and trailing spaces**
  - **Ex..**System.out.println(" Java ".trim()); // Java
- `indexOf()` → **Returns index of first occurrence**
  - **Ex..**System.out.println("Java".indexOf('v')); // 2
- `replace()` → **Replaces characters or substring**
  - **Ex..**System.out.println("Java".replace('a','o')); // Jovo
- `contains()` → **Checks if string contains sequence**
  - **Ex..**System.out.println("Java Programming".contains("Java")); // true
- `startsWith()` → **Checks starting characters**
  - **Ex..**System.out.println("Java".startsWith("Ja")); // true
- `endsWith()` →**Checks ending characters**
  - **Ex..**System.out.println("Java".endsWith("va")); // true
- `split()` → **Splits string into array**
  - **Ex..**String[] a = "Java Spring Boot".split(" ");
- `isEmpty()` → **Checks if string is empty**
  - **Ex..**System.out.println("".isEmpty()); // true
- `concat()` → **Joins two strings**
  - **Ex..**System.out.println("Java".concat(" Dev")); // Java Dev

9. Inheritance and Polymorphism

9.1 Inheritance Types and Benefits

Inheritance allows a child class to reuse properties and methods of a parent class using `extends`.

Types (One Line Each)

- Single → One child inherits one parent
- Multilevel → Child inherits from another child
- Hierarchical → Multiple children inherit one parent

Benefits of Inheritance:

➢ Inheritance reduces code duplication, improves reusability, and supports polymorphism.

9.2 Method Overriding

Method overriding occurs when a child class provides a new implementation of a parent class method with the same signature.

9.3 Dynamic Binding (Run-Time Polymorphism)

➢ Dynamic binding means the method call is resolved at runtime based on the object, not the reference.

A obj = new B();

obj.show(); // calls B's show()

9.4 Super Keyword and Method Hiding

super Keyword

➢ `super` is used to access parent class variables, methods, and constructors.

Access Parent Method:

➢ super.show();

Call Parent Constructor:

- super();

Super keyword Example:

```
class Parent {
    int x = 10;
    Parent() {
        System.out.println("Parent Constructor");
    }
    void show() {
        System.out.println("Parent show method");
    }
}
class Child extends Parent {
    int x = 20;
    Child() {
        super();   // calls Parent constructor
        System.out.println("Child Constructor");
    }
    void display() {
        System.out.println(super.x); // access Parent variable
        super.show();               // call Parent method
        System.out.println(x);       // access Child variable
    }
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
```

```
    }
}
```

Method Hiding:

- Method hiding occurs when a static method in child class has the same name as static method in parent class.

```
class A {
    static void display() { System.out.println("A"); }
}
class B extends A {
    static void display() { System.out.println("B"); }
}
```

10. Interfaces and Abstract Classes

10.1 Abstract Classes and Methods

Abstract Class

- An abstract class is a class that cannot be instantiated (object cannot be created) and is used as a base class.

Abstract Method

- An abstract method has no body and must be implemented by the child class.

Example:

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

10.2 Interfaces: Multiple Inheritance in Java

What is Multiple Inheritance?

- Multiple inheritance means one child class inherits from more than one parent class.

Why Java Does NOT Support Multiple Inheritance:

->Ambiguity Problem (Diamond Problem)

- The main reason is confusion about which method to call.

->Complexity and Maintenance Issues

->Method Conflict

->Constructor Confusion

What is an Interface?

- ➢ An interface is a blueprint of a class that contains only abstract methods (Java 8+ can have default/static methods).

Why Interface?

- ➢ Java does not support multiple inheritance using classes, so interfaces provide multiple inheritance.

10.3 Implementing multiple interface(multiple inheritance)

A class can implement multiple interfaces at the same time.

Example:

```
interface A {
    void show();
}
interface B {
    void display();
}
class Test implements A, B {
    public void show() {
        System.out.println("Show method");
    }
    public void display() {
        System.out.println("Display method");
    }
}
```

11. Packages and Access Modifiers

11.1 JavaPackages: Built-in and User-Defined Packages

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

✔ Helps in code organization
✔ Avoids name conflicts
✔ Supports access control

Built-in Packages

➢ Packages already provided by Java.

Common Built-in Packages:

- `java.lang` → basic classes (`String`, `Math`, `System`)
- `java.util` → collections, `Scanner`
- `java.io` → input/output
- `java.sql` → database

✔ `java.lang` is **automatically imported.**

User-Defined Packages

➢ Packages created by the developer.

Example:

```
package com.myapp.model;

public class Student {
    public void show() {
        System.out.println("Student class");
    }
}
```

11.2 AccessModifiers: Private, Default, Protected, Public

Access modifiers control the visibility of classes, methods, and variables.

1. Private

- ➢ Accessible **only within the same class**.
- ➢ Ex..private int id;

2.Default (No Keyword)

- ➢ Accessible **within the same package only**.
- ➢ Ex..int age; // default

3.Protected

- ➢ Accessible within same package and subclass in other packages.
- ➢ Ex..protected String name;

4.Public

- ➢ Accessible **from anywhere**.
- ➢ Ex..public void display() { }

11.3 Importing Packages and Classpath

To use classes from another package without writing full package name.

1️⃣ Import Single Class:

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

2️⃣ Import All Classes

```
import java.util.*;
```

What is Classpath?

- ➢ Classpath tells JVM where to find `.class` files while running a program.

12. Exception Handling

12.1 Types of Exceptions: Checked and Unchecked

What is an Exception?

➢ An exception is an unexpected error that occurs during program execution and stops normal flow.

1️⃣ Checked Exceptions

👉 Checked exceptions are checked at compile time and must be handled using `try-catch` or `throws`.

Examples:

- `IOException`
- `SQLException`
- `FileNotFoundException`

Example:

```
import java.io.*;

class Test {

    public static void main(String[] args) throws IOException
{

        FileReader fr = new FileReader("abc.txt");

    }

}
```

2️⃣ Unchecked Exceptions

➢ Unchecked exceptions occur at runtime and are not checked by the compiler.

Examples:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException

Ex..`int a = 10 / 0;   // ArithmeticException`

12.2 try,catch, finally, throw, throws

1.try

➢ try block contains code that may cause an exception.

```
try {
    int a = 10 / 0;
}
```

2.catch

➢ **catch block handles the exception**.

```
catch (ArithmeticException e) {
    System.out.println("Error occurred");
}
```

3.finally

➢ finally block always executes, whether exception occurs or not.

```
finally {

    System.out.println("Always executed");

}
```

4.throw

➢ throw is used to explicitly throw an exception.

Ex..throw new ArithmeticException("Invalid value");

5.throws

➢ throws is used in method declaration to pass exception to caller.

Ex..void readFile() throws IOException {}

try–catch–finally Example:

```
try {

    int a = 10 / 0;

}

catch (ArithmeticException e) {

    System.out.println("Cannot divide by zero");

}

finally {

    System.out.println("Program ended");

}
```

12.3 Custom Exception Classes

What is a Custom Exception?

➢ A custom exception is a user-defined exception created by extending Exception or RuntimeException.

Checked Custom Exception:

```
class InvalidAgeException extends Exception {

    InvalidAgeException(String msg) {

        super(msg);

    }

}
```

Usage:

```
if (age < 18) {

    throw new InvalidAgeException("Not eligible");

}
```

Unchecked Custom Exception:

```
class InvalidDataException extends RuntimeException {

    InvalidDataException(String msg) {

        super(msg);

    }

}
```

13.Multithreading

13.1 Introduction to Threads

A thread is a lightweight unit of a process that allows a program to perform multiple tasks at the same time.

13.2 Creating Threads by Extending Thread Class or Implementing Runnable Interface.

1.By Extending Thread Class

- Create a class that extends Thread and override run() method.
- **Example:**

```
class MyThread extends Thread {

    public void run() {

        System.out.println("Thread running");

    }

    public static void main(String[] args) {

        MyThread t = new MyThread();

        t.start();   // start thread

    }

}
```

2.By Implementing Runnable Interface

- Create a class that implements Runnable and pass object to Thread.
- **Example:**

```
class MyTask implements Runnable {
      public void run() {
          System.out.println("Thread running");
      }

      public static void main(String[] args) {
          Thread t = new Thread(new MyTask());
          t.start();
      }
}
```

13.3 Thread Life Cycle

A thread passes through different states during execution.

🔄 **States:**

1. New → Thread created
2. Runnable → Ready to run
3. Running → Executing
4. Waiting / Blocked → Waiting for resource
5. Terminated (Dead) → Finished execution

13.4 Synchronization and Inter-thread Communication

What is Synchronization?

➢ Synchronization prevents multiple threads from accessing shared data at the same time, avoiding inconsistency.

🔹 **Synchronized Method**

```
synchronized void print() {
    // critical section
}
```

🔹 **Synchronized Block**

```
synchronized(this) {
    // critical section
}
```

✔ Avoids race condition
✔ Uses **object-level lock**

What is Inter-thread Communication?

➢ **Allows threads to communicate and cooperate with each other.**

Methods Used:

- **wait()**
- **notify()**
- **notifyAll()**

Example:

```
class Demo {
    synchronized void message() throws InterruptedException {
        wait();   // waiting state
        System.out.println("Notified");
    }
    synchronized void notifyThread() {
        notify(); // wakes waiting thread
    }
}
```

14. File Handling

14.1 Introduction to File I/O in Java (java.io package)

File I/O (Input/Output) in Java is used to read data from files and write data into files.

- Java provides the java.io package for file handling.
- File I/O allows data to be stored permanently on disk.
- Streams are used to perform input and output operations.

Common Classes in java.io:

- File
- FileReader, FileWriter
- BufferedReader, BufferedWriter
- ObjectInputStream, ObjectOutputStream

14.2 FileReader and FileWriter Classes

FileReader

- Used to **read character data from a file**.
- Suitable for **text files**.

```
FileReader fr = new FileReader("data.txt");

int ch = fr.read();
```

FileWriter

- Used to **write character data into a file**.

```
FileWriter fw = new FileWriter("data.txt");

fw.write("Hello Java");

fw.close();
```

14.3 BufferedReader and BufferedWriter

BufferedReader

- Reads text from input stream **efficiently**.
- Reads data **line by line** using `readLine()`.

```
BufferedReader br = new BufferedReader(new
FileReader("data.txt"));

String line = br.readLine();
```

BufferedWriter

- Writes text to file **efficiently**.
- Uses buffer memory to improve performance.

```
BufferedWriter bw = new BufferedWriter(new
FileWriter("data.txt"));

bw.write("Java File Handling");

bw.close();
```

14.4 Serialization and Deserialization

Serialization

- Process of **converting an object into a byte stream**.
- Used to store object in a file or send over network.
- Class must implement a Serializable interface.

```
class Student implements Serializable {
    int id;
    String name;
}
```

```
ObjectOutputStream oos =
    new ObjectOutputStream(new FileOutputStream("obj.txt"));
oos.writeObject(student);
```

Deserialization

- Process of **converting byte stream back into object**.

```
ObjectInputStream ois =
    new ObjectInputStream(new FileInputStream("obj.txt"));
Student s = (Student) ois.readObject();
```

15. Collections Framework

15.1 Introduction to Collections Framework

The Java Collections Framework (JCF) is a set of classes and interfaces used to store, manipulate, and retrieve groups of objects efficiently.

- Located in the `java.util` package
- Supports dynamic data structures
- Improves performance and code reusability

Benefits:

- Ready-made data structures
- Reduces programming effort
- Improves readability and maintainability

15.2 List, Set, Map, and Queue Interfaces

List Interface

- Allows duplicate elements
- Maintains insertion order
- Access elements by index

Examples: `ArrayList`, `LinkedList`

Set Interface

- Does not allow duplicates
- No index-based access

Examples: **HashSet**, **TreeSet**

Map Interface

- Stores data as key-value pairs
- Keys must be unique

Examples: **HashMap**, **TreeMap**

Queue Interface

- Follows FIFO (First In First Out)
- Used for processing tasks

Example: PriorityQueue

15.3 ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

ArrayList

- Uses dynamic array
- Fast random access
- Allows duplicates

Ex...`ArrayList<Integer> list = new ArrayList<>();`

LinkedList

- Uses **doubly linked list**
- Faster insertion and deletion
- Allows duplicates

Ex...`LinkedList<String> list = new LinkedList<>();`

HashSet

- Stores **unique elements**
- No insertion order
- Faster performance

Ex...`HashSet<Integer> set = new HashSet<>();`

TreeSet

- Stores unique elements in sorted order
- Slower than HashSet

Ex...`TreeSet<Integer> set = new TreeSet<>();`

HashMap

- Stores key-value pairs
- No insertion order
- Allows one null key

Ex...`HashMap<Integer, String> map = new HashMap<>();`

TreeMap

- Stores key-value pairs in sorted order
- Does not allow null keys

Ex...`TreeMap<Integer, String> map = new TreeMap<>();`

Example covered all collections:

```
import java.util.*;
public class CollectionExample {
    public static void main(String[] args) {

        // ArrayList
        ArrayList<String> arrayList = new ArrayList<>();
        arrayList.add("Java");
        arrayList.add("Spring");

        // LinkedList
        LinkedList<String> linkedList = new LinkedList<>();
        linkedList.add("HTML");
        linkedList.add("CSS");
```

```
// HashSet
HashSet<Integer> hashSet = new HashSet<>();
hashSet.add(10);
hashSet.add(20);
hashSet.add(10);   // duplicate not allowed

// TreeSet
TreeSet<Integer> treeSet = new TreeSet<>();
treeSet.add(30);
treeSet.add(10);
treeSet.add(20);   // sorted order

// HashMap
HashMap<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "Amit");
hashMap.put(2, "Rahul");

// TreeMap
TreeMap<Integer, String> treeMap = new TreeMap<>();
treeMap.put(3, "Java");
treeMap.put(1, "Python");
```

```
        // Output
        System.out.println("ArrayList: " + arrayList);
        System.out.println("LinkedList: " + linkedList);
        System.out.println("HashSet: " + hashSet);
        System.out.println("TreeSet: " + treeSet);
        System.out.println("HashMap: " + hashMap);
        System.out.println("TreeMap: " + treeMap);
    }
}
```

15.4 Iterators and ListIterators

Iterator

- Used to traverse elements one by one
- Works with all collections

```
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

ListIterator

- Used only with List interface
- Can traverse in both directions
- Allows add, update, remove

```
ListIterator<Integer> li = list.listIterator();
```

16.Java Input/Output (I/O)

16.1 Streams in Java (InputStream, OutputStream)

A stream in Java is a flow of data from a source to a destination.

- Streams are part of the java.io package
- Used to read or write data sequentially

InputStream

➢ **Used to read data from a file or input source.**

Examples:

- FileInputStream
- BufferedInputStream

```
InputStream in = new FileInputStream("data.txt");
```

OutputStream

➢ Used to **write data** to a file or output destination.

Examples:

- FileOutputStream
- BufferedOutputStream

```
OutputStream out = new FileOutputStream("data.txt");
```

16.2 Reading and Writing Data Using Streams

Reading Data Using InputStream

```
FileInputStream fis = new FileInputStream("data.txt");
int ch;
while ((ch = fis.read()) != -1) {
    System.out.print((char) ch);
}
fis.close();
```

Writing Data Using OutputStream

```
FileOutputStream fos = new FileOutputStream("data.txt");
String msg = "Java Streams";
fos.write(msg.getBytes());
fos.close();
```

16.3 Handling File I/O Operations

File I/O operations involve creating, reading, writing, and closing files.

Common Steps:

1. Create file object
2. Open stream
3. Read / write data
4. Close stream

Using File Class:

File file = new File("data.txt");

System.out.println(file.exists());

Exception Handling in File I/O:

```
try {
    FileInputStream fis = new FileInputStream("data.txt");
}
catch (IOException e) {
    System.out.println("File error");
}
```

- ➢ Use **buffered streams** for better performance
- ➢ Handle exceptions properly
- ➢ Close streams using `finally` or try-with-resources

```
try (FileInputStream fis = new FileInputStream("data.txt")) {
    // auto close
}
```
