Core Java Notes

Why you must learn Java

1. Wide Usage (Web-apps, backend,Mobile apps, enterprise software).
2. Object Oriented
3. Rich APIs and Community Support

What is a Programming Language

Giving instructions to a computer

What is an Algorithm

An algorithm is a step-by-step procedure for solving a problem or
performing a task

History of Java : Developed by James Gosling at Sun Microsystems (Early 1990s):
Originally named 'Oak', later renamed Java in 1995

Java Features :

Robust: Java is robust due to its strong memory management,exception handling, and type-checking mechanisms, which help in preventing system crashes and ensuring reliable performance.

Multithreaded : Multithreading in programming is the ability of a CPU to execute multiple threads concurrently, allowing for more efficient processing and task management.

Architecture Neutra : Java is architecturally neutral because its compiled code (bytecode) can run on any device with a Java Virtual Machine (JVM), regardless of the underlying hardware
Architecture.

Interpreted and High Performance:Java combines high performance with interpretability, as its bytecode is interpreted by the Java Virtual Machine (JVM), which employs Just-In-Time JIT) compilation for efficient and fast execution. Distributed :Java is inherently distributed,designed to facilitate network-based application development and interaction, seamlessly integrating with Internet protocols and remote method invocation.

Object Oriented Programming :

.JDK

- It's a software development kit required to develop Java applications.
- Includes the JRE, an interpreter/loader (Java), a compiler (javac), a doc generator (Javadoc), and other tools needed for Java development.
- Essentially, JDK is a superset of JRE.

JRE

- It's a part of the JDK but can be downloaded separately.
- Provides the libraries, the JVM, and other components to run applications
- Does not have tools and utilities for developers like compilers or debuggers

.JVM

- It's a part of JRE and responsible for executing the bytecode.

• Ensures Java’s write-once-run-anywhere capability.
• Not platform-independent: a different JVM is needed for each type of OS.

Variables :Variables are like containers used for storing data values.

Variable Declaration:eg(int a=10) here a is variable

Data Types:
Data types in Java specify the size and type of values that a variable can store.

Primitive Data Types:
These are the basic building blocks for data manipulation and are predefined by the language. They store values directly in memory. Java has eight primitive data types:

- **byte:** 8-bit signed integer.
- **short:** 16-bit signed integer.
- **int:** 32-bit signed integer.
- **long:** 64-bit signed integer.
- **float:** 32-bit single-precision floating-point number.
- **double:** 64-bit double-precision floating-point number.
- **boolean:** Represents true or false values.
- **char:** 16-bit Unicode character.

2. Non-Primitive (Reference) Data Types:
These are more complex types that store references to memory locations where data is stored. They include:

- **Classes:** User-defined types that encapsulate data and methods.
- **Interfaces:** Define contracts for classes to implement.
- **Arrays:** Ordered collections of elements of the same type.
- **Strings:** Sequences of characters.

Java Identifier Rules: The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), ‘$‘ (dollar sign) and ‘_‘ (underscore).

2. Can’t use keywords or reserved words

3. Identifiers should not start with digits([0-9]).

4. Java identifiers are case-sensitive.

5. There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.

Keywords: Keywords are the reserved words in java, which as special meaning or purpose of those words.(eg:for,if,while etc).

Operators : operators are special symbols that perform operations on variables or values.

Types of Operators in Java

1. **Arithmetic Operators:** are used to perform simple arithmetic operations on primitive and non-primitive data types.

- * : Multiplication
- / : Division
- % : Modulo
- + : Addition
- - : Subtraction

2. **Relational Operators**:Relational operators compare values and return Boolean results:

- `==` , Equal to.
- `!=` , Not equal to.
- `<` , Less than.
- `<=` , Less than or equal to.
- `>` , Greater than.
- `>=` , Greater than or equal to.

3. **Logical Operators:**Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

4. **Ternary Operator**:condition **?** if true **:** if false

Looping :in programming languages is a feature that facilitates the execution of a set of instructions repeatedly while some condition evaluates to true

for loop

The for loop is used when we know the number of iterations (we know how many times we want to repeat a task). The for statement includes the initialization, condition, and increment/decrement in one line.

Syntax:

```
for (initialization; condition; increment/decrement) {

// code to be executed

}
```

```
eg:import java.io.*;

class Geeks {

   public static void main(String[] args){

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

while Loop

A while loop is used when we want to check the condition before executing the loop body.

Syntax:

```
        while (condition) {

        // code to be executed

        }
```

```
eg:import java.io.*;

class Geeks {

   public static void main(String[] args)

   {

      int i = 0;

      while (i <= 10) {

         System.out.print(i + " ");

         I++;

          }

   }

}
```

do-while Loop

The do-while loop ensures that the code block executes **at least once** before checking the condition.

Syntax:

```
        do {

        // code to be executed

        } while (condition);
```

```
eg:class Geeks {
    public static void main(String[] args)
    {
        int i = 0;
        do {
            System.out.print(i + " ");
            i++;
        } while (i <= 10);
    }
}
```

Conditional Loops:

`if` Statement: The `if` statement executes a block of code only if a specified condition is true.

syntax:

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

```
eg:if (age >= 18){
   System.out.print(“you can vote”)
}
```

if-else Statement

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't.

Syntax:

```
          if(condition){
          // Executes this block if
          // condition is true
```

```
}else{

// Executes this block if

// condition is false

}
```

```
eg:import java.util.*;

class Geeks {

    public static void main(String args[])

    {

        int i = 10;


        if (i < 15)

            System.out.println("i is smaller than 15");

        else

            System.out.println("i is greater than 15");

    }

}
```

Switch Case

The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax:

```
switch (expression) {

case value1:

// code to be executed if expression == value1

break;

case value2:

// code to be executed if expression == value2
```

```
          break;

          // more cases...

          default:

          // code to be executed if no cases match

          }

eg:import java.io.*;

class Geeks {

    public static void main(String[] args)

    {       int num = 20;

        switch (num) {

        case 5:

            System.out.println("It is 5");

            break;

        case 10:

            System.out.println("It is 10");

            break;

        case 15:

            System.out.println("It is 15");

            break;

        case 20:

            System.out.println("It is 20");

            break;

        default:

            System.out.println("Not present"); } } }
```

Java Methods

A **method** is a block of code which only runs when it is called

Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

Eg:
```
public class Main {
 static void myMethod() {
   System.out.println("I just got executed!");
 }
 public static void main(String[] args) {
   myMethod();
 }      }
```

Arrays in Java

Arrays in Java are one of the most fundamental data structures that allow us to store multiple values of the same type in a single variable. They are useful for storing and managing collections of data

eg:
```
     public class Main {
   public static void main(String[] args)
   {     // declares an Array of integers.
       int[] arr;
// allocating memory for 5 integers.
       arr = new int[5];
// initialize the elements of the array
       // first to last(fifth) element
         arr[0] = 10;
       arr[1] = 20;
```

```
        arr[2] = 30;

        arr[3] = 40;

        arr[4] = 50;

        // accessing the elements of the specified array

        for (int i = 0; i < arr.length; i++)

          System.out.println("Element at index " + i + " : " + arr[i]);

    }     }
```

For-Each Loop in Java

The **for-each loop in** Java (also called the enhanced for loop) was introduced in Java 5 to simplify iteration over arrays and collections.

Syntax of For-each Loop

```
         for (type var : array) {

         statements using var;

         }
```

```
eg:// Java Program to Iterate through an array

// Using for-each loop

import java.io.*;

class Geeks {

     public static void main(String[] args) {

      // Array declaration

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

        for (int e : arr) {

           System.out.print(e + " ");

       }   }         }
```

2D Array:

A 2D array in Java is essentially an array of arrays, visualized as a table with rows and columns.

Eg:

```
// Java Program to Demonstrate

// Multi Dimensional Array

import java.io.*;

public class Geeks

{

   public static void main(String[] args){


       // Multidimensional array declaration

       int[][] arr;


       // Initializing the size of row and column respectively

       arr = new int[1][3];


       // Initializing the values

       arr[0][0] = 3;

       arr[0][1] = 5;

       arr[0][2] = 7;


       // Display the values using index

       System.out.println("arr[0][0] = " + arr[0][0]);

       System.out.println("arr[0][1] = " + arr[0][1]);

       System.out.println("arr[0][2] = " + arr[0][2]);

   }     }
```

Array Methods:

1. `Arrays.toString()` – Convert array to String

```
import java.util.Arrays;

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

System.out.println(Arrays.toString(arr));  // Output: [1, 2, 3, 4]
```

2. `Arrays.sort()` – Sort an array

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

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));  // Output: [1, 2, 4, 5]
```

3. `Arrays.copyOf()` – Copy an array to a new array

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

int[] copy = Arrays.copyOf(original, 5);

System.out.println(Arrays.toString(copy));  // [1, 2, 3, 0, 0]
```

4. `Arrays.equals()` – Compare two arrays

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

int[] b = {1, 2, 3};

System.out.println(Arrays.equals(a, b));  // true
```

5. `Arrays.fill()` – Fill array with a value

```
int[] arr = new int[5];

Arrays.fill(arr, 9);

System.out.println(Arrays.toString(arr));  // [9, 9, 9, 9, 9]
```

```
int[] arr = new int[5];

Arrays.fill(arr, 9);

System.out.println(Arrays.toString(arr));  // [9, 9, 9, 9, 9]
```

6. `Arrays.binarySearch()` – Search for a value (must be sorted)

```
int[] arr = {1, 3, 5, 7, 9};

int index = Arrays.binarySearch(arr, 5);

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

7. Looping Over Arrays

```
String[] names = {"Alice", "Bob", "Charlie"};

for (int i = 0; i < names.length; i++) {

    System.out.println(names[i]);

}

// or enhanced for-loop

for (String name : names) {

    System.out.println(name);

}
```

`String` in Java

A `String` is an object that represents a sequence of characters.

Strings are **immutable** (cannot be changed once created).

Ways to Create Strings

```
String s1 = "Hello";               // String literal
String s2 = new String("Hello");   // Using constructor
```

Common `String` Methods (with Examples)

Method	Description	Example
`length()`	Returns string length	`s.length()`
`charAt(int index)`	Character at position	`s.charAt(2)`
`substring(int start, int end)`	Extract substring	`s.substring(1, 4)`

`equals(String s2)`	Compares content	`s1.equals(s2)`
`equalsIgnoreCase(String s2)`	Ignores case	`s1.equalsIgnoreCase(s2)`
`compareTo(String s2)`	Lexical comparison	`s1.compareTo(s2)`
`toLowerCase()` / `toUpperCase()`	Case conversion	`s.toLowerCase()`
`trim()`	Removes spaces	`" abc ".trim()`
`replace(a, b)`	Replace characters	`s.replace('a', 'o')`
`split(" ")`	Splits string into array	`s.split(" ")`
`contains("text")`	Checks if text exists	`s.contains("Hi")`

Examples:

```
String str = "Java Programming";

System.out.println(str.length());        // 16

System.out.println(str.charAt(5));        // P

System.out.println(str.substring(0, 4));  // Java

System.out.println(str.toUpperCase());    // JAVA PROGRAMMING

System.out.println(str.contains("gram")); // true
```

StringBuilder & StringBuffer (for mutable strings)

```
StringBuilder sb = new StringBuilder("Hello");

sb.append(" Java");

System.out.println(sb);  // Hello Java
```

Constructor in Java (Special Method to Create Objects)

A constructor is a special block of code that is called when an object is created. Its main job is to initialize the object.

```
public class Student {

   String name;

   int rollNumber;

   // Constructor

   Student(String n, int r) {

      name = n;

      rollNumber = r;

   }

   void display() {

      System.out.println(name + " - " + rollNumber);

   }    }
```

`this` Keyword :The `this` keyword is a reference to the **current object** of the class — it helps you **distinguish between instance variables and local variables** or **call current object methods**.

Method Overloading :**Method Overloading** means creating **multiple methods with the same name** but **different parameters** in the **same class**.

```
Eg:   class Calculator {

   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;

   }    }
```

Constructor Overloading :Constructor Overloading means defining multiple constructors with different parameters in the same class.

```
class Student {
    String name;
    int age;
    // No-arg constructor
    Student() {
        this.name = "Unknown";
        this.age = 0;
    }    // Constructor with one argument
    Student(String name) {
        this.name = name;
        this.age = 0;
    }   // Constructor with two arguments
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }     }
```

Key Differences:

Feature	**Method Overloading**	**Constructor Overloading**
Purpose	Perform similar tasks differently	Create object in different ways
Name	Same method name	Same constructor name (same as class)
Return type	Can have any return type	No return type (not even `void`)

Basic OOP Concepts (Overview)

Concept	Description
Class	Blueprint/template for creating objects
Object	Real-world entity created from a class
Encapsulation	Wrapping data + behavior into a single unit (class)
Abstraction	Hiding internal details and showing only essential features
Inheritance	One class can inherit fields and methods from another
Polymorphism	Same method can behave differently based on context

Class in Java: A class is a blueprint that defines: Attributes,Behaviors(methods)

```
public class Car {

   // Fields

   String brand;

   int speed;

 // Method

   void drive() {
        System.out.println(brand + " is driving at " + speed + " km/h");
    }    }
```

Object

An object is an instance of a class. It holds actual values**.**

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Object creation
        myCar.brand = "Tesla";
        myCar.speed = 120;
        myCar.drive();  // Output: Tesla is driving at 120 km/h
    }
}
```

`this` vs `super` Keyword in Java

Both are used inside a **class** to refer to objects, but they refer to **different contexts**:

Keyword	Refers To
`this`	**Current class** object
`super`	**Immediate parent class** object

1. `this` Keyword

Refers to the current **class instance**.

✅ **Uses of `this`:**

- Refer to current class fields
- Invoke current class methods
- Call current class constructor (`this()`)
- Return current class object (`return this`)

🔧 **Example:**

```
class Student {
    String name;
    Student(String name) {
        this.name = name;  // resolve conflict
    }
   void display() {
        System.out.println("Name: " + this.name);
    }
}
```

2. `super` Keyword

Refers to the immediate parent class object.

✅ **Uses of `super`:**

- **Access parent class variables**
- **Call parent class methods**
- **Invoke parent class constructor (`super()`)**

🔧 Example:

```
class Animal {

    String name = "Animal";

    void display() {

        System.out.println("I am an Animal");

    }

}

class Dog extends Animal {

    String name = "Dog";

    void printNames() {

        System.out.println(name);        // Dog

        System.out.println(super.name); // Animal

    }

    void display() {

        super.display(); // calls Animal's display()

        System.out.println("I am a Dog");

    }

}
```

`final` Keyword

A `final` variable is **constant** — its value **cannot be changed** once assigned.

```
eg:class Constants {
    final int MAX_USERS = 100;

    void printMax() {
        System.out.println("Max users: " + MAX_USERS);
    }
}
```

A `final` method **cannot be overridden** by any subclass.

```
eg:class Animal {
    final void eat() {
        System.out.println("Animal eats food");
    }
}

class Dog extends Animal {
    // void eat() { ❌ Error: Cannot override final method }
}
```

3. `final` with Classes

```
eg:final class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

// class Car extends Vehicle {} ❌ Error: Cannot inherit from final class
```

Inheritance

Inheritance allows a class (called the **child/subclass**) to acquire fields and methods from another class (called the **parent/superclass**).

This promotes **code reusability**, **modularity**, and implements the **"is-a"** relationship.

```
Example:class Animal {

    void eat() {

        System.out.println("This animal eats food");

    }     }

class Dog extends Animal {

    void bark() {

        System.out.println("Dog barks");

    }     }

public class Main {

    public static void main(String[] args) {

        Dog d = new Dog();
        d.eat();  // inherited from Animal

        d.bark(); // defined in Dog

    }

}
```

Types of Inheritance in Java

Type	Description	Supported in Java?
Single	One subclass inherits from one superclass	✅ Yes
Multilevel	One subclass inherits from another subclass	✅ Yes
Hierarchical	Multiple subclasses inherit from a single superclass	✅ Yes
Multiple (via classes)	One class inherits from multiple classes	❌ No (ambiguity)
Multiple (via interfaces)	One class implements multiple interfaces	✅ Yes

Package in Java

A **package** is a namespace that organizes a set of related classes and interfaces.

Think of it like a **folder** in a computer, where related Java files (classes) are grouped together.

🔹 Benefits of Using Packages

✅ Code organization
✅ Avoid name conflicts
✅ Easier maintenance
✅ Access control
✅ Reusability

Syntax:

Creating a Package

```
// File: MyClass.java
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("This is a class inside 'mypackage'");
    }
}
```

Using a Package:

```
// File: Main.java
import mypackage.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }    }
```

Access Modifiers with Packages

Modifier	Access Within Package	Access Outside Package
`public`	✅ Yes	✅ Yes
`protected`	✅ Yes	✅ (with inheritance)
`default` (no modifier)	✅ Yes	❌ No
`private`	❌ No	❌ No

Access Modifiers

Access modifiers control **where** a class, method, or variable can be **accessed from** in your program.

Types of Access Modifiers in Java:

Modifier	Class	Package	Subclass (other pkg)	World (anywhere)
`private`	✅	❌	❌	❌
`default` *(no modifier)*	✅	✅	❌	❌
`protected`	✅	✅	✅	❌
`public`	✅	✅	✅	✅

1. `private` – Most restrictive

Accessible only within the same class

```
class Test {
   private int num = 10;

   private void show() {
      System.out.println("Private Method");
   }
}
```

2. *default* (no keyword)

Accessible within the same package only.

Can't be accessed from another package.

```
class Test {
   int num = 20; // default
   void show() {
      System.out.println("Default Method");
   }
}
```

3. `protected`

Accessible:

- In the **same package**
- In **subclasses**, even in other packages

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

4. `public` – Least restrictive

=>Accessible from **anywhere** in the program =>Full access from other packages and projects.

```
Eg:     public class MyClass {
           public void display() {
              System.out.println("This is public");
           }      }
```

Encapsulation

Encapsulation is the process of **hiding internal details** of an object and **only exposing necessary parts** using public methods.

Getter vs Setter in Java

Aspect	Getter	Setter
Purpose	To **read/access** a private variable	To **update/modify** a private variable
Method type	Usually starts with `get`	Usually starts with `set`
Return value	Returns the value of a field	Doesn’t return anything (void)
Parameters	No parameters	Takes one parameter (new value)

Why Use Encapsulation

Benefit	Explanation
🔒 Security	Prevents unauthorized access to fields
🧪 Data validation	Validate values before setting
🔄 Flexibility	Change implementation without affecting users
📦 Modularity	Keeps code organized and modular

```
Example:public class Student {
    // Step 1: Make variables private
    private String name;
    private int age;
    // Step 2: Provide public getter and setter methods
    public String getName() {
        return name;
 }
    public void setName(String newName) {
        this.name = newName;  }
    public int getAge() {
        return age;
    }
    public void setAge(int newAge) {
        if (newAge > 0) {
            this.age = newAge;
        }          }        }
```

Usage:

```
public class Main {

   public static void main(String[] args) {

      Student s = new Student();

      s.setName("Kavya");

      s.setAge(20);

      System.out.println("Name: " + s.getName());

      System.out.println("Age: " + s.getAge());

   }    }
```

Data Hiding

Data hiding is an OOP principle where internal object details (data) are hidden from outside classes. It's closely related to Encapsulation.

✅ Achieved using:

- `private` access modifier on fields
- `public` getters and setters to control access

Example: Data Hiding

```
public class Account {

   private double balance;  // hidden data

   public double getBalance() {

      return balance;

   }

   public void deposit(double amount) {

      if (amount > 0) {

         balance += amount;

}        }        }
```

`static` vs `instance` keywords

Keyword	Belongs to	Accessed by	Memory Allocation	Example Use
`static`	Class (shared by all objects)	`ClassName.member` or `object.member`	Loaded once when class is loaded	Utility methods, constants
none (instance)	Object (each object gets its own copy)	Only via objects	Each time object is created	Instance-specific data

Example:

```
public class Student {

    // instance variable

    String name;

    // static variable (shared by all objects)

    static String college = "ABC University";

    // constructor

    Student(String name) {

        this.name = name;

    }

    void showInfo() {

        System.out.println(name + " - " + college);

    }

}
```

Abstraction

Abstraction means **hiding internal implementation details** and only showing the essential features to the user.

✅ Achieved by:

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

It helps reduce complexity and increase code reusability.

🔹 Example: Abstraction Using Abstract Class

```
abstract class Animal {
    abstract void makeSound();  // abstract method (no body)
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound();  // Dog barks
        d.eat();        // This animal eats food
    }
}
```

Interface in Java

An **interface** is a **fully abstract** type used to define a **contract** – what a class must do, but not how.

✅ Key Points:

- All methods are public and abstract by default (until Java 7)
- A class can implement **multiple interfaces**
- From Java 8+, interfaces can have default and static methods too

---

🔹 Example: Interface

```
interface Vehicle {
    void start();
    void stop();
}
class Car implements Vehicle {
    public void start() {
        System.out.println("Car started");
    }
    public void stop() {
        System.out.println("Car stopped");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle v = new Car(); // Interface reference
        v.start();  // Car started
        v.stop();   // Car stopped
    }
}
```

Functional Interface

A **Functional Interface** is an interface that contains **only one abstract method**.

✅ Rules:

- Can have any number of **default** or **static** methods.
- Must have **only one abstract method** (can be annotated with `@FunctionalInterface` for clarity).

🔹 Example: Functional Interface

```
@FunctionalInterface
interface Greeting {
    void sayHello();
}
```

2. Lambda Expressions

A **lambda expression** is a short block of code that takes in parameters and returns a value. It's used primarily to implement functional interfaces.

✅ Syntax:

```
(parameters) -> { statements }
```

Example: **Lambda Expression with Functional**

```
Interface@FunctionalInterface
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greeting greet = () -> System.out.println("Hello,
Kavya!");
        greet.sayHello();
    }     }
```

Benefits of Lambda Expressions

- **Reduces boilerplate** (less code)
- **Improves readability**
- **Perfect for one-time-use functionality**
- Often used with **streams**, **collections**, and **event handling**

Stack vs Heap Memory in Java

Feature	Stack Memory	Heap Memory
Stores	Method calls, local variables	Objects, instance variables
Access	Last-In-First-Out (LIFO)	Random Access
Memory Size	Limited (faster access)	Larger (slower access)
Managed By	Automatically by Java (after method exits)	Garbage Collector
Lifetime	Temporary (method duration)	Until object is no longer used

Example:

```
public class MemoryExample {
    public static void main(String[] args) {
        int x = 10;  // stored in Stack
        String name = new String("Kavya");  // object stored
in Heap, reference in Stack
    }
}
```

Polymorphism

Polymorphism means "many forms." In Java, it lets the same method or object behave differently based on context.

Types:

1. **Compile-Time Polymorphism** (Method Overloading)
2. **Run-Time Polymorphism** (Method Overriding)

Method Overloading (Compile-Time Polymorphism)

Same method name, different parameters (by number or type).

EX:

```
public class Calculator {

    int add(int a, int b) {

        return a + b;

    }

    double add(double a, double b) {

        return a + b;

    }

}

Calculator calc = new Calculator();

System.out.println(calc.add(5, 6));        // int version

System.out.println(calc.add(5.5, 3.2));    // double version
```

Run-Time Polymorphism (Method Overriding)

A subclass provides its own version of a method defined in its superclass.

EX:

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // upcasting
        a.makeSound();  // Output: Dog barks
    }
}
```

Object Class in Java

Every class in Java implicitly extends the **Object** class. It provides common methods:

Method	Purpose
toString()	Returns string representation
equals()	Compares two objects for equality
hashCode()	Returns object’s hash code
getClass()	Returns runtime class of object
clone()	Makes a copy of object
finalize()	Called before object is destroyed

Example: toString() and equals()

```
public class Student {
    String name;

    Student(String name) {
        this.name = name;
    }

    public String toString() {
        return "Student: " + name;
    }

    public boolean equals(Object obj) {
        Student s = (Student) obj;
        return this.name.equals(s.name);
    }
}
```

Java 8 Major Features

Feature	Description
Lambda Expressions	Functions without names (anonymous)
Functional Interfaces	Interfaces with a single abstract method
Stream API	Process data collections in a functional style
Default & Static Methods	Methods inside interfaces
Method References	Shorthand for calling methods
Optional Class	Avoid `null` pointer exceptions
Date & Time API	Better date/time handling (`java.time.*`)

Stream API

Stream API lets you process **collections** (like List, Set) in a **declarative** and **functional** style.

Operation	Type	Description
filter()	Intermediate	Filters elements
map()	Intermediate	Transforms elements
collect()	Terminal	Collects the result
forEach()	Terminal	Iterates and performs actions
sorted()	Intermediate	Sorts elements
count()	Terminal	Counts elements

Example 1: Filter and print names starting with 'K'

```
List<String> names = Arrays.asList("Kavya", "Ravi", "Kiran",
"Amit");

names.stream()

    .filter(name -> name.startsWith("K"))

    .forEach(System.out::println);
```

Example 2: Map and Collect

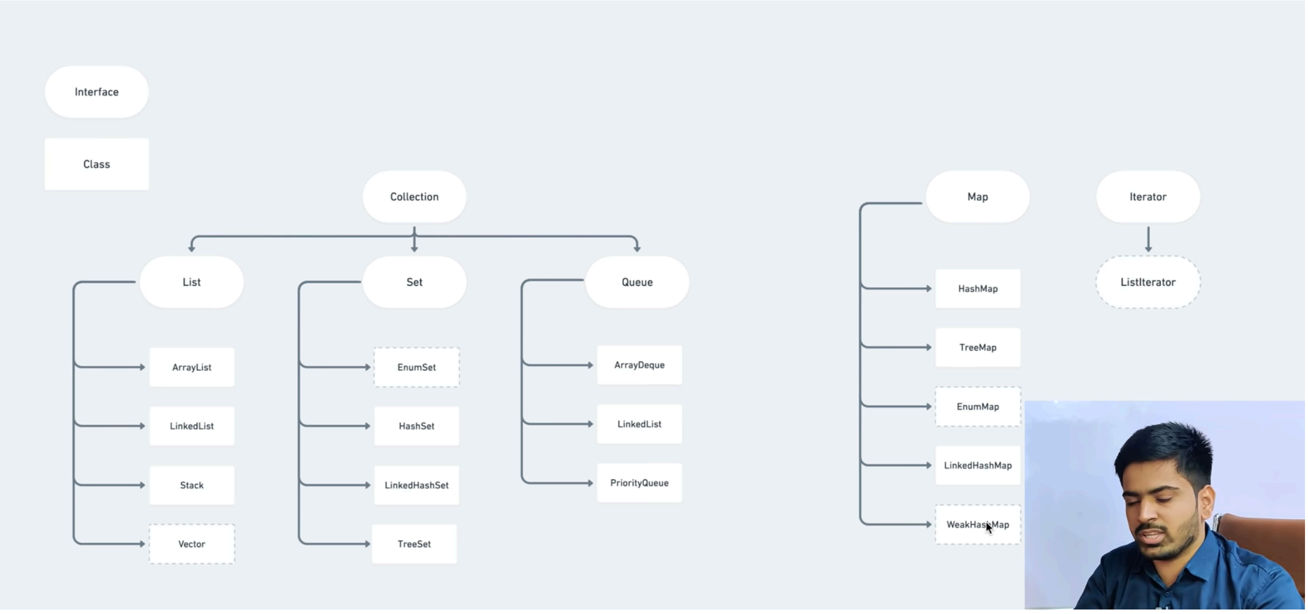
```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

List<Integer> squared = numbers.stream()

    .map(n -> n * n)

    .collect(Collectors.toList());

System.out.println(squared);  // [1, 4, 9, 16]
```

Example 3: Count names with length > 4

```
long count = names.stream()

    .filter(name -> name.length() > 4)

    .count();

System.out.println(count);
```

Java Collections Framework

Java Collections Framework (JCF) provides **interfaces and classes** for storing and manipulating groups of data (like objects).



Common Interfaces:

Interface	**Description**
List	Ordered collection (can have duplicates)
Set	Unordered collection (no duplicates)
Map	Key-value pairs
Queue	Follows FIFO (First In First Out)

1. List Interface

Ordered collection that allows **duplicate elements**.

🔹 Implementations:

- ArrayList: Fast for read, slow for insert/delete.
- LinkedList: Good for frequent insert/delete.

```
Eg :import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Kavya");
        names.add("Ravi");
        names.add("Kavya"); // allows duplicates


        System.out.println("First element: " + names.get(0));
        names.set(1, "Rahul");  // update index 1
        names.remove("Kavya");  // removes first occurrence
        System.out.println(names);
    }

}
```