**Core Java Concepts**

**1. OOPs Concepts:**

**i. Class and Object:**

* A class is a blueprint or template that defines the properties (variables) and behaviors (methods) of objects. It does not occupy memory until an object is created.
* An object is a real-world instance of a class that has its own state (data) and behavior (methods).It is created from a class using the new keyword.

**ii. Constructor and it's overloading:**

* A constructor is a special method in a class that is used to initialize objects.It has the same name as the class and does not have a return type.It is called automatically when an object is created.
* Constructor overloading means having more than one constructor in the same class with different parameter lists.It allows you to create objects in different ways (with or without initial values).

**iii. Static Members:**

* Static Variable: A static variable belongs to the class, not to any specific object.It is shared by all objects of the class.
* Static Method: A static method can be called without creating an object of the class. It can access only static variables directly.
* Static Block: A static block runs once when the class is loaded into memory. It is used to initialize static data.

**iv. Final Keyword:**

* Final Variable: A final variable value cannot be changed once assigned. **Example**: final int x = 10;
* Final Method: A final method cannot be overridden in a subclass.**Example:** class A {final void display() {System.out.println("Hello");}}
* Final Class: A final class cannot be inherited by any other class. **Example:** final class A { }class B extends A { }

**v. Access Modifier:**

Access modifiers define the visibility (where a member can be accessed) of classes, variables, and methods.

* public: Can be accessed from anywhere — inside or outside the class, even from other packages.
* private: Can be accessed only within the same class. It is not visible to other classes.
* protected: Can be accessed within the same package and also by subclasses (even in other packages)
* default (no modifier): If no modifier is used, it is accessible only within the same package.

**vi. Encapsulation:**

Encapsulation means binding data (variables) and methods (functions) together in a single unit (class). It is used to protect data by keeping it private and providing public getter and setter methods to access it.

**Example:**

class Student {

private int id;

private String name;

public void setId(int i) {id = i;}

public void setName(String n) {name = n;}

public int getId() {return id;}

public String getName() {return name;}

}

class Test {

public static void main(String[] args) {

Student s = new Student();

s.setId(101);

s.setName("Dharshini");

System.out.println(s.getId() + " " + s.getName());

}

}

**vi. Inheritance:**

Inheritance is the process where one class (child) gets the properties and methods of another class (parent). It helps in code reusability and method overriding.

1. Single Inheritance: One class inherits from another single class.

class A {

void display() {System.out.println("Class A");}

}

class B extends A {

void show() {System.out.println("Class B");}

}

B inherits from A.

2. Multilevel Inheritance: A class inherits from another class, which itself is inherited from another.

class A { void showA() { } }

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

class C extends B { void showC() { } }

C inherits from B, and B inherits from A.

3. Hierarchical Inheritance: Multiple classes inherit from the same parent class.

class A { void showA() { } }

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

class C extends A { void showC() { } }

Both B and C inherit from A.

**vii. Polymorphism:**

Polymorphism means "many forms" — the same method or operation behaves differently based on the object or situation. It allows one name to have multiple behaviors.

1. Compile-time Polymorphism (Method Overloading): Happens when multiple methods in the same class have the same name but different parameters. The method is decided at compile time.

**Example:**

class Calculator {

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

int add(int a, int b, int c) {return a + b + c;}

}

The correct add() method is chosen during compilation.

2. Runtime Polymorphism (Method Overriding): Happens when a child class provides a new implementation for a method that is already defined in its parent class. The method is decided at runtime.

**Example:**

class Animal {

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

}

class Dog extends Animal {

void sound() {System.out.println("Bark");}

}

class Test {

public static void main(String[] args) {

Animal a = new Dog();

a.sound();

}

}

**viii. Abstraction:**

Abstraction means showing only essential features and hiding the internal details. It focuses on what an object does, not how it does it.

Abstraction can be achieved in two ways:

1. Using Abstract Classes
2. Using Interfaces

1. **Abstract Class**: A class declared with the keyword abstract. It can have abstract methods (without body) and non-abstract methods (with body). It cannot be instantiated (objects cannot be created directly). It is used for partial abstraction.

**Example:**

abstract class Animal {

abstract void sound();

void sleep() {System.out.println("Sleeping");}

}

class Dog extends Animal {

void sound() {System.out.println("Bark");}

}

**2. Interface:**

An interface is a blueprint of a class that contains abstract methods (methods without a body). It is used to achieve abstraction and multiple inheritance in Java. A class that uses an interface must implement all its methods.

* Declared using the keyword interface.
* All methods are public and abstract by default.
* A class uses the keyword implements to use an interface.
* One class can implement multiple interfaces.
* Interfaces help in loose coupling and code reusability.

**Example:**

import java.util.\*;

interface Calculator

{

double add(double a, double b);

double sub(double a, double b);

double mul(double a, double b);

double div(double a, double b);

}

interface Scientific {double power(double a, double b);}

class BasicCalculator implements Scientific,Calculator­­­

{­­­

public double add(double a, double b){return a+b;}

public double sub(double a, double b) {return a-b;}­

public double mul(double a, double b) {return a\*b;}

public double div(double a, double b){return a/b;}

public double power(double a, double b){ return Math.pow(a,b);}

}

public class App {

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

BasicCalculator app=new BasicCalculator();

System.out.println("Enter first Value: ");

double a=s.nextDouble();

System.out.println("Enter Second Value: ");

double b=s.nextDouble();

System.out.println("Enter Any Symbol(+,-,\*,/,^): ");

char sys=s.next().charAt(0);

double ans;

switch(sys)

{

case '+':

ans=app.add(a,b);

System.out.println("Answer: "+ans);

break;

case '-':

ans=app.sub(a,b);

System.out.println("Answer: "+ans);

break;

case '\*':

ans=app.mul(a,b);

System.out.println("Answer: "+ans);

break;

case '/':

ans=app.div(a,b);

System.out.println("Answer: "+ans);

break;

case '^':

ans=app.power(a,b);

System.out.println("Answer: "+ans);

break;

}}}

**2. Enums:**

An enum is a special type in Java used to define a set of constant values (fixed values that do not change). It helps make the code more readable and type-safe.

* Declared using the keyword enum.
* Enum constants are public, static, and final by default.
* Used when you have a fixed set of related constants — like days, directions, colors, etc.
* You can also add methods and variables inside an enum.

**Example:**

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

}

class TestEnum {

public static void main(String[] args) {

Day d = Day.MONDAY;

System.out.println(d);

}

}

**3. Annotations:**

Annotations in Java are special markers or metadata that provide information to the compiler or JVM about the code. They do not change the actual logic, but they help give instructions or add information. They start with the symbol @.

Commonly Used Annotations

**1. @Override**: Used when a method overrides a method from its parent class.

Helps the compiler check correctness.

**Example:**

class Parent {

void show() { }

}

class Child extends Parent {

@Override

void show() {System.out.println("Overridden method"); }

}

**2. @Deprecated**: Marks a method or class as outdated (not recommended to use). The compiler gives a warning if it’s used.

**Example:**

@Deprecated

void oldMethod() {System.out.println("This method is deprecated");}

**3. @SuppressWarnings**: Used to ignore compiler warnings for specific code parts.

**Example:**

@SuppressWarnings("unchecked")

void example() {

List list = new ArrayList(); // unchecked warning suppressed

list.add("Hello");

}

**4. @SafeVarargs**: Used to suppress warnings when using varargs with generics (from Java 7+). It ensures that the method does not perform unsafe operations.

**Example:**

@SafeVarargs

private final void print(List<String>... lists) {

for (List<String> list : lists) {System.out.println(list);}

}

**5. @FunctionalInterface:** Used to mark an interface that has exactly one abstract method (used in Lambda Expressions).

**Example:**

@FunctionalInterface

interface MyFunction {void display();}

Meta-Annotations: These are annotations that apply to other annotations (used when creating custom annotations).

**6. @Target:** Specifies where the annotation can be used (e.g., on a method, class, field, etc.).

**7. @Retention**: Specifies how long the annotation should be retained:

SOURCE: discarded after compilation

CLASS: kept in class file but ignored by JVM

RUNTIME: available at runtime (via reflection)

**8. @Documented:** Indicates that the annotation should be included in Javadoc.

**9. @Inherited:** Allows a subclass to inherit an annotation from its superclass.

**10. Custom Annotation:** You can create your own annotation using @interface.

**4. Serialization:**

Serialization is the process of converting an object into a byte stream, so it can be saved to a file, sent over a network, or stored in a database.

Deserialization is the reverse process — converting the byte stream back into an object.

Normally, Java objects exist only in memory (RAM). If the program stops, the data is lost. Serialization lets us save the object’s data and reuse it later.

**Example**

import java.io.\*;

class Student implements Serializable {

int id;

String name;

Student(int id, String name) {

this.id = id;

this.name = name;

}

}

Serialization (Saving object to file)

Student s1 = new Student(101, "Dharshini");

FileOutputStream fos = new FileOutputStream("student.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(s1);

oos.close();

fos.close();

System.out.println("Object serialized!");

Deserialization (Reading object from file)

FileInputStream fis = new FileInputStream("student.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

Student s = (Student) ois.readObject();

ois.close();

fis.close();

System.out.println("ID: " + s.id);

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

**Output:**

ID: 101

Name: Dharshini

**transient Keyword:** If you don’t want a variable to be saved during serialization, mark it as transient:

transient String password;

(This field will be skipped and become null when deserialized.)

**serialVersionUID**: Used to maintain version control of a serialized class.

private static final long serialVersionUID = 1L;

(If the class changes later, this ID ensures deserialization still works.)

**5. Exception Handling:**

Exception Handling is a way to handle runtime errors in Java — so the program doesn’t crash and can continue executing smoothly. We use try–catch–finally blocks for this.

**Example:**

try {

int a = 10 / 0; // ❌ ArithmeticException

} catch (ArithmeticException e) {

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

} finally {

System.out.println("Finally block always runs!");

}

**Multiple Catch Blocks**: Used when different exceptions might occur.

try {

int[] arr = new int[3];

arr[5] = 10; // ArrayIndexOutOfBoundsException

} catch (ArithmeticException e) {

System.out.println("Math error");

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index error");

} catch (Exception e) {

System.out.println("Other exception");

}

Always keep specific exceptions before the general Exception class.

**throw Keyword**: Used to manually throw an exception.

throw new ArithmeticException("Invalid operation");

**throws Keyword**: Used in method declaration to say that the method might throw an exception.

void readFile() throws IOException {

// code that may cause IOException

}

**Custom Exception:** You can create your own exception class by extending Exception or RuntimeException.

class MyException extends Exception {

MyException(String msg) {super(msg);}

}

public class Demo {

public static void main(String[] args) {

try {

throw new MyException("Custom error message!");

} catch (MyException e) {

System.out.println(e.getMessage());

}

}

}

Some of the common exceptions in Java are ArithmeticException (when dividing by zero), NullPointerException (when trying to access a null object), ArrayIndexOutOfBoundsException (when using an invalid array index), and NumberFormatException (when converting a string to a number fails). There are also checked exceptions like IOException (for input/output errors) and FileNotFoundException (when a file is missing).

**6. Threads**

A thread in Java is a lightweight subprocess — the smallest unit of execution. When you run a Java program, there is always at least one thread running called the main thread. Threads allow multiple parts of a program to run concurrently, which helps in performing multiple tasks at the same time.

For example, in a web browser, one thread can handle user input while another loads data in the background.

**Creating Threads**

There are two main ways to create a thread in Java:

1. By Extending the Thread Class: You can create a thread by extending the Thread class and overriding the run() method.

class MyThread extends Thread {

public void run() {

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

}

}

public class Demo {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start(); // starts the thread and calls run()

}

}

start() creates a new thread and calls the run() method internally.

2. By Implementing the Runnable Interface: This is the preferred way because Java does not support multiple inheritance. Using Runnable, your class can extend another class and still define thread behavior.

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable thread is running...");

}

}

public class Demo {

public static void main(String[] args) {

Thread t = new Thread(new MyRunnable());

t.start();

}

}

Here, Runnable is a functional interface, meaning it has only one abstract method (run()).

**7. Thread States:**

**1. New State**: When a thread is created but not yet started. The thread object exists, but no system resources are allocated yet.

Thread t = new Thread();

**2. Start / Runnable State**: When you call the start() method, the thread moves from new to runnable. It means the thread is ready to run and waiting for the CPU to schedule it. It may start running immediately or after some time depending on the CPU.

t.start();

**3. Running State:** When the CPU picks the thread, it starts executing the code inside the run() method.A thread can move back and forth between runnable and running based on CPU scheduling.

public void run() {

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

}

**4. Sleep / Waiting State**: A thread enters sleeping or waiting state when it temporarily stops execution. It can happen because:

You call Thread.sleep(time) (sleeping)

You call wait() (waiting for another thread)

It’s waiting to acquire a lock (blocked)

The thread goes back to runnable once the waiting time is over or another thread notifies it.

try {

Thread.sleep(1000); // TIMED\_WAITING

} catch (InterruptedException e) {

e.printStackTrace();

}

**5. Dead (Terminated) State:** When the run() method finishes, the thread’s life ends. Once a thread is dead, it cannot be restarted.

System.out.println("Thread finished!");

**Thread Lifecycle:**

NEW → RUNNABLE → RUNNING → WAITING/SLEEPING → DEAD

**Example Flow:**

Thread created → NEW

start() called → RUNNABLE

CPU executes → RUNNING

sleep() or wait() → WAITING

Task finished → DEAD

**8. Comparable vs Comparator**

**1. Comparable – “Default Way to Compare”**

**Definition:**

Comparable is an interface that allows a class to define its own natural way of ordering — how its objects should be compared.

**Syntax:**

public interface Comparable<T> {

int compareTo(T other);

}

**Meaning:**

If a class implements Comparable, it must define the logic in the compareTo() method that explains how one object is compared with another.

**Example:**

class Student implements Comparable<Student> {

int id;

String name;

Student(int id, String name) {

this.id = id;

this.name = name;

}

public int compareTo(Student other) {

return this.id - other.id;

}

}

Now when you do:

Collections.sort(studentList);

It automatically sorts by id, because that’s the natural order defined in the class.

**2. Comparator – “Custom Way to Compare”**

**Definition:**

Comparator is an interface used to write custom comparison logic outside the class — useful when you want different sorting options.

**Syntax:**

public interface Comparator<T> {

int compare(T o1, T o2);

}

**Meaning:**

You can create multiple comparators for different sorting logic — for example, by name, age, or marks.

**Example (Anonymous Class):**

Comparator<Student> nameComparator = new Comparator<Student>() {

public int compare(Student s1, Student s2) {

return s1.name.compareTo(s2.name);

}

};

**Example (Lambda Expression):**

Comparator<Student> nameComparator = (s1, s2) -> s1.name.compareTo(s2.name);

Then use:

Collections.sort(studentList, nameComparator);

This sorts the students by name, not by id.

1. What is Collection Framework?

The Java Collection Framework (JCF) is a unified architecture that provides interfaces, classes, and algorithms to store, retrieve, and manipulate groups of objects efficiently.

i. To handle multiple objects easily (instead of arrays).

ii. To provide ready-to-use data structures like dynamic lists, sets, queues, maps, etc.

iii. To provide built-in methods (add, remove, sort, search, etc.) so you don’t need to code them manually.

**9. What are all Collections in Java?**

**1. List Interface**

i. Ordered collection (insertion order is maintained)

ii. Allows duplicate elements, which makes it different from a Set.

iii. Index-based access

-> The List interface defines many useful methods such as:

\* add() to insert elements

\* get() to retrieve elements by index

\* remove() to delete elements

\* set() to update elements

\* size() to get the number of elements

\* contains() to check if an element exists

-> Because List is an interface, you cannot create an object directly from it.

-> Instead, you create objects from classes that implement it — such as ArrayList, LinkedList, Vector, or Stack.

-> Classes:

**1. ArrayList:**

-> It is fast for accessing elements (because it uses indexes), but slower for insertions or deletions in the middle, since shifting of elements is required.

syntax: List<data-type> names = new ArrayList<>();

**2. LinkedList:**

-> LinkedList is another implementation of the List interface, but it also implements Deque (Double Ended Queue).

-> Because of this structure, insertion and deletion operations are faster than in ArrayList, especially when done frequently in the middle or beginning of the list.

-> However, random access using an index (like get(5)) is slower, since it needs to traverse nodes from the start or end.

Syntax: List<data-type> IDs = new LinkedList<>();

**3. Vector:**

-> Vector is very similar to ArrayList — it also uses a dynamic array internally.

-> However, the main difference is that Vector is synchronized, meaning it is thread-safe. Multiple threads can access a Vector object safely without corrupting data.

Syntax: Vector<Integer> v = new Vector<>();

**4. Stack:**

-> Stack is a subclass of Vector that follows the LIFO (Last In First Out) principle.

-> It’s mainly used when you want to store elements where the last inserted element is removed first, like in undo/redo operations, expression evaluation, or backtracking problems.

-> methods such as:

push() → to add elements on top of the stack

pop() → to remove the top element

peek() → to view the top element without removing it

empty() → to check if the stack is empty

-> Although Stack is still available, modern Java recommends using ArrayDeque instead for stack behavior because it’s faster and not legacy-based.

Syntax: Stack<String> stack = new Stack<>();

**2. Set Interface**

-> The Set interface in Java is used to store unique elements — it does not allow duplicates.

i. A set is unordered (order may change).

ii. It can store null (only one null).

iii. It has no index, so you can’t access elements using numbers like in a list.

-> Types of Set:

**1. HashSet**

-> Stores elements in random order (no insertion order).

-> Does not allow duplicates.

-> Allows one null value.

-> Fastest for add, remove, and search.

Syntax: HashSet<String> set = new HashSet<>();

**2. LinkedHashSet**

-> Similar to HashSet, but keeps elements in insertion order.

-> Does not allow duplicates.

-> Allows one null value.

Syntax: LinkedHashSet<Integer> set = new LinkedHashSet<>();

**3. TreeSet**

-> Stores elements in sorted (ascending) order.

-> Does not allow duplicates.

-> Does not allow null.

Syntax: TreeSet<String> set = new TreeSet<>();

**3. Queue Interface**

-> The Queue interface in Java is used when you want to store elements in order and process them one by one — usually in FIFO (First In, First Out) order.

-> Think of it like a line of people — the first person to enter the line is the first to be served.

-> Maintains order (usually FIFO).

-> Can have duplicate elements.

-> Provides special methods for adding and removing elements.

-> Some queues can also work in LIFO order (like stacks) using ArrayDeque.

->Methods:

i. add() → add an element

ii. offer() → add element (returns false if fails)

iii. remove() → removes and returns first element

iv. poll() → removes and returns first element, or null if empty

v. peek() → shows first element without removing it

-> Implementation of Queue Interface

**1. LinkedList (also acts as Queue)**

-> LinkedList implements both List and Queue.

-> When used as a Queue, it follows FIFO order — elements are added at the end and removed from the front.

Syntax: Queue<String> queue = new LinkedList<>();

**2. PriorityQueue**

-> PriorityQueue stores elements based on priority, not in insertion order.

-> By default, it arranges elements in natural ascending order (for numbers or strings).

-> When you remove elements, the smallest element (highest priority) comes out first.

Syntax: PriorityQueue<Integer> pq = new PriorityQueue<>();

**3. ArrayDeque**

-> ArrayDeque (Double Ended Queue) allows you to add or remove elements from both ends — front and back.

-> It can act as both Queue (FIFO) and Stack (LIFO).

-> It’s faster than both LinkedList and Stack.

Syntax: ArrayDeque<String> dq = new ArrayDeque<>();

**4. Map Interface**

-> The Map interface in Java is used to store data in key–value pairs.

-> Each value is stored with a unique key, and you can use that key to quickly get its value.

-> It is not a part of the Collection interface hierarchy but still part of the Java Collection Framework (inside java.util package).

-> Stores key–value pairs.

-> Keys must be unique; values can be duplicate.

-> One null key is allowed (depends on implementation).

-> No insertion order guaranteed (depends on the class used).

-> You can get values quickly using keys (very fast lookup).

-> Common methods in Map:

\* put(key, value) → Adds or replaces a key–value pair.

\* get(key) → Returns the value of the given key.

\* remove(key) → Removes the entry for the given key.

\* containsKey(key) → Checks if a key exists.

\* containsValue(value) → Checks if a value exists.

\* keySet() → Returns all keys.

\* values() → Returns all values.

\* entrySet() → Returns all key–value pairs.

-> Implementation

**1. HashMap**

-> HashMap is the most commonly used Map implementation.

-> It stores data in random order using a hash table, so the order of elements is not guaranteed.

-> It allows one null key and multiple null values.

-> It is fast for adding, removing, and searching.

-> However, it is not synchronized (not thread-safe).

Syntax: HashMap<Integer, String> map = new HashMap<>();

**2. LinkedHashMap**

-> LinkedHashMap is similar to HashMap, but it maintains the insertion order of key–value pairs.

-> It is a combination of a HashMap and a linked list, so when you print it, elements appear in the same order they were added.

-> It also allows one null key and multiple null values.

Syntax: LinkedHashMap<Integer, String> map = new LinkedHashMap<>();

**3. TreeMap**

-> TreeMap stores elements in sorted order of keys (ascending order by default).

-> It is based on a Red-Black Tree structure.

-> It does not allow null keys, but allows null values.

-> It is slower than HashMap, because it maintains sorting order.

Syntax: TreeMap<Integer, String> map = new TreeMap<>();

**10. What is the Collections Class?**

-> The Collections class is a utility class in Java (found in java.util package).

-> It contains static methods that you can use to operate on collections (like lists, sets, maps, etc.).

-> Methods:

**1. sort()**

-> Used to sort elements of a list in ascending order (natural order).

-> For custom sorting, you can pass a Comparator.

Ex: Collections.sort(list);

**2. reverse()**

-> Reverse the order of elements in a list.

Ex: Collections.reverse(list);

**3. shuffle()**

-> Rearranges the list elements randomly.

Ex: Collections.shuffle(list);

**4. max() and min()**

-> Find the largest and smallest elements from a collection.

Ex: int max = Collections.max(list);

int min = Collections.min(list);

**5. frequency()**

-> Counts how many times a specific element appears in a collection.

Ex: List<String> names = Arrays.asList("A", "B", "A", "C");

int count = Collections.frequency(names, "A");

**6. swap()**

-> Swaps two elements in a list using their indexes.

Ex: List<String> colors = new ArrayList<>(Arrays.asList("Red", "Blue", "Green"));

Collections.swap(colors, 0, 2);

o/p: After swap: [Green, Blue, Red]

**7. fill()**

-> Replaces all elements in a list with a given value.

Ex: Collections.fill(colors, "Black");

o/p: After fill: [Black, Black, Black]

**8.copy()**

-> Copies elements from one list to another (destination list must be the same size).

Ex: List<String> src = Arrays.asList("A", "B", "C");

List<String> dest = new ArrayList<>(Arrays.asList("X", "Y", "Z"));

Collections.copy(dest, src);

System.out.println("After copy: " + dest);

o/p: After copy: [A, B, C]

**4. What Are Generics in Java?**

-> Generics allow you to write type-safe code — meaning you can tell Java what kind of data a collection will hold.

-> Without generics, a collection can hold any type of object, which can cause runtime errors.

-> With generics, you specify the data type during declaration, so mistakes are caught at compile-time.

**Syntax:** ClassName<Type> objectName = new ClassName<Type>();

**11. What is a Stream in Java?**

-> A Stream in Java is a sequence of elements (like from a collection, array, etc.) that you can process using functional operations — such as filtering, mapping, sorting, or reducing.

-> Think of a stream as a pipeline — data flows through it, and you apply operations to transform or filter it.

**Syntax:** collection.stream()

.intermediateOperation()

.terminalOperation();

**Syntax Pattern:** stream()

.filter(...)

.map(...)

.sorted(...)

.forEach(...);

-> You can chain many intermediate operations, but the stream ends only when a terminal operation is used (like forEach, count, collect).

Ex: Find only numbers greater than 20, Double them, Print them

List<Integer> list = Arrays.asList(10, 20, 30, 40, 50);

list.stream()

.filter(n-> n>20)

.map(n-> n\*2)

.forEach(System.out::println);

**12. How Stream Work?**

-> Three Main Steps:

**\* Create a Stream**

way 1: list.stream(); // from a List

way 2: Stream.of(1, 2, 3, 4); // directly

way 3: Arrays.stream(array); // from array

**\* Perform Intermediate Operations**

Common ones:

-> .filter() → Filters elements

-> .map() → Transforms elements

-> .sorted() → Sorts elements

-> .distinct() → Removes duplicates

-> .limit(n) → Takes first n elements

-> .skip(n) → Skips first n elements

Ex: Applying

import java.util.\*;

import java.util.stream.\*;

class Example {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 20, 30, 40, 50, 60, 70);

numbers.stream()

.filter(n -> n > 20) // keep only numbers greater than 20

.distinct() // remove duplicates

.map(n -> n \* 2) // multiply each by 2

.sorted() // sort ascending

.skip(1) // skip the first element

.limit(3) // take next 3 elements

.forEach(System.out::println); // print them

}

}

**\* Perform Terminal Operation**

-> These end the stream and produce a result.

-> Common ones:

\* .forEach() → Iterate and perform an action

\* .collect() → Convert stream to list, set, etc.

\* .count() → Count elements

\* .findFirst() / .findAny() → Get an element

\* .reduce() → Combine elements to a single value

Ex: in practice\StreamTerminal.java file

**13. What is a Parallel Stream?**

-> Normally, a stream runs sequentially — one element is processed at a time using a single thread.

-> A parallel stream, on the other hand, splits the data into multiple parts and processes them simultaneously using multiple threads (through the ForkJoinPool).

-> The name comes from the way it works:

\* Fork → split a big task into smaller subtasks.

\* Join → combine (join) the results of those subtasks when they’re done.

-> So it divides and conquers — that’s the main idea.

Ex: sequential vs Parallel are in practice\streamdiff.java

**14. How to create a Parallel Stream?**

-> There are two ways:

-> From a Collection:

list.parallelStream();

-> From a normal Stream:

list.stream().parallel();

-> Both are same.

**15. When to use Parallel Stream?**

-> Use when:

\* The data set is large (thousands of elements).

\* Each operation is independent (no shared data).

\* CPU-intensive operations (e.g. calculations, transformations).

-> Avoid when:

\* Order matters (parallel may not preserve order).

\* Small data sets (overhead of threads is costly).

\* You modify shared resources (can cause race conditions).

Ex: to understand the usage see practice\ParallelStreams.java

-> we can use forEachOrdered();

-> we can combine with collectors

**16. What is Collectors?**

-> Collectors are used with .collect() to convert a stream back into a collection.

-> toList() → collects stream elements into a List (keeps duplicates and order).

-> toSet() → collects elements into a Set (removes duplicates).

-> toMap(keyFn, valueFn) → collects elements into a Map using key and value functions.

-> These are terminal operations — they end the stream.

**17. What is groupingBy()?**

-> Used to group elements based on a property or condition.

-> Returns a Map<key, List<values>>.

-> Example use: group students by department, or words by length.

-> It’s a collector, so used with .collect().

**18. What is PartitionBy()?**

-> Splits elements into two groups — one where the condition is true, and one where it’s false.

-> Returns a Map<Boolean, List<values>>.

-> Common use: divide data into even/odd, pass/fail, active/inactive, etc.

**19. What is Joining()?**

-> Combines all stream elements (Strings) into one single string.

-> You can add a delimiter, prefix, and suffix.

-> Example: joining names with commas — useful for display or reports.

**20. What are Primitive Streams?**

-> Specialized streams for primitive types → IntStream, LongStream, DoubleStream.

-> Faster and memory-efficient because they avoid auto-boxing.

-> Provide numeric methods like sum(), average(), max(), min().

-> Often used for numeric calculations or performance-critical code.