**Task 1**What are streams?

Answer:- A Stream is a sequence of elements (like a list) that you can process, filter, map, and collect using a set of operations.   
  
  
**Task 2**   
What is Boilerplate code ?  
Answer:-Boilerplat Code refers to code that you have to write again and again in multiple places, even though it’s not directly related to the core logic of your program.   
  
What is Lack of Parallelism?   
Answer:- If your code runs everything one after another (sequentially), it’s slow and not optimized.  
Using Streams with .parallel() or multithreading helps with parallelism.   
  
What is Lack of Composition?   
Answer:- Means your code is not modular or reusable.  
You are writing logic in one big chunk instead of breaking it into small, composable methods or functions.

**Task 3**  
List of Intermediate and terminal operations ?  
Answer:-

**✅ Intermediate Operations**

(These return a new Stream — **lazy** operations)

| **Operation** | **Description** |
| --- | --- |
| filter(Predicate) | Filters elements that match a condition |
| map(Function) | Transforms each element |
| flatMap(Function) | Flattens nested streams (e.g., List<List> to List) |
| distinct() | Removes duplicate elements |
| sorted() | Sorts the stream elements in natural order |
| sorted(Comparator) | Sorts using a custom comparator |
| limit(long maxSize) | Limits stream to the first n elements |
| skip(long n) | Skips the first n elements |
| peek(Consumer) | Performs action on each element (useful for debugging) |
| mapToInt(ToIntFunction) | Converts stream to IntStream |
| mapToLong(ToLongFunction) | Converts stream to LongStream |
| mapToDouble(ToDoubleFunction) | Converts stream to DoubleStream |
| boxed() | Converts primitive stream to object stream |
| parallel() | Converts a sequential stream to parallel |
| sequential() | Converts a parallel stream back to sequential |
| unordered() | Hints that order of elements doesn’t matter |

**✅ Terminal Operations**

(These produce a **result or side-effect** — **triggers stream processing**)

| **Operation** | **Description** |
| --- | --- |
| forEach(Consumer) | Applies action to each element |
| forEachOrdered(Consumer) | Applies action in encounter order |
| toArray() | Converts stream to array |
| reduce(BinaryOperator) | Reduces elements to one value |
| collect(Collector) | Collects result into a collection (List, Set, Map, etc.) |
| min(Comparator) | Finds the minimum element |
| max(Comparator) | Finds the maximum element |
| count() | Counts number of elements |
| anyMatch(Predicate) | Returns true if any match |
| allMatch(Predicate) | Returns true if all match |
| noneMatch(Predicate) | Returns true if none match |
| findFirst() | Returns the first element (Optional) |
| findAny() | Returns any element (mostly used in parallel streams) |

**Task 4**  
interface MyInterface{

    // abstract method

    double getPiValue();

}

public class Main {

    public static void main( String[] args ) {

    // declare a reference to MyInterface

    MyInterface ref;

    // lambda expression

    ref = () -> 3.1415;

    System.out.println("Value of Pi = " + ref.getPiValue());

    }

}

Output:- Value of Pi = 3.1415  
  
**Task 5**package samplePackage2;

interface MyInterface1 {

// abstract method

String reverse(String n);

}

public class Task5June{

public static void main( String[] args ) {

// declare a reference to MyInterface

// assign a lambda expression to the reference

MyInterface1 ref = (str) -> {

String result = "";

for (int i = str.length()-1; i >= 0 ; i--)

result += str.charAt(i);

return result;

};

// call the method of the interface

System.*out*.println("Lambda reversed = " + ref.reverse("Lambda"));

}  
output:- Lambda reversed = adbmaL

}  
**Task 6**

**package** samplePackage2;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StreamMain {

// create an object of list using ArrayList

**static** List<String> *places* = **new** ArrayList<>();

// preparing our data

**public** **static** List getPlaces(){

// add places and country to the list

*places*.add("Nepal, Kathmandu");

*places*.add("Nepal, Pokhara");

*places*.add("India, Delhi");

*places*.add("USA, New York");

*places*.add("Africa, Nigeria");

**return** *places*;

}

**public** **static** **void** main( String[] args ) {

List<String> myPlaces = *getPlaces*();

System.***out***.println("Places from Nepal:");

// Filter places from Nepal

myPlaces.stream()

.filter((p) -> p.startsWith("Nepal"))

.map((p) -> p.toUpperCase())

.sorted()

.forEach((p) -> System.***out***.println(p));

}

}

Output:- Places from Nepal:

NEPAL, KATHMANDU

NEPAL, POKHARA

**Task 7**  
Write a code to create a array list to store 5 integers and display the square of each no..

**package** samplePackage2;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** MapExample {

**public** **static** **void** main(String[] args) {

// Step 1: Create an ArrayList of numbers

List<Integer> numbers = **new** ArrayList<>();

numbers.add(1);

numbers.add(2);

numbers.add(3);

numbers.add(4);

numbers.add(5);

numbers.add(6);

numbers.add(7);

numbers.add(8);

numbers.add(9);

// Step 2: Use map() to square each number and print

System.***out***.println("Squared Numbers:");

numbers.stream()

.map(n -> n \* n) // square each number

.forEach(n -> System.***out***.println(n)); // print each squared number

}

}

Output:- Squared Numbers:

1

4

9

16

25

36

49

64

**Task 8**  
What do you understand by map() in Java?

In Java, the map() method is a stream intermediate operation used to transform each element of a stream. It takes a Function as an argument and applies it to every element, returning a new stream with the modified elements.   
  
  
  
**Task 9**Write a code to create an array list and filter the values which are odd numbers and display them..

**package** samplePackage2;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** OddNumberFilter {

**public** **static** **void** main(String[] args) {

// Create an ArrayList of numbers

List<Integer> numbers = **new** ArrayList<>();

numbers.add(10);

numbers.add(15);

numbers.add(22);

numbers.add(33);

numbers.add(40);

numbers.add(55);

// Display only odd numbers

System.***out***.println("Odd Numbers:");

numbers.stream()

.filter(n -> n % 2 != 0) // Filter condition for odd numbers

.forEach(n -> System.***out***.println(n)); // Print each odd number

}

}

Output:- Odd Numbers:

15

33

55

**Task 10**What do you understand by filter()?

Answer:- filter() is used to remove unwanted elements and keep only those that match the given condition.  
  
**Task 11**Wap to create an array list to remove duplicate values from the List.

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class RemoveDuplicates {

public static void main(String[] args) {

// Step 1: Create an ArrayList with duplicate values

List<Integer> numbers = new ArrayList<>();

numbers.add(10);

numbers.add(20);

numbers.add(10);

numbers.add(30);

numbers.add(20);

numbers.add(40);

// Step 2: Remove duplicates using stream().distinct()

List<Integer> uniqueNumbers = numbers.stream()

.distinct() // removes duplicate values

.collect(Collectors.toList());

// Step 3: Display the result

System.out.println("Original List: " + numbers);

System.out.println("List after removing duplicates: " + uniqueNumbers);

}

}  
output:- Original List: [10, 20, 10, 30, 20, 40]

List after removing duplicates: [10, 20, 30, 40]   
  
**Task 12**What do you understand by distinct()?

Answer:- distinct() helps to filter out duplicates and keep only unique values from a list or stream  
  
sorted()

* Sorts the elements in natural order or with a custom comparator

limit(n)

* Returns only the first n elements of the stream.

skip(n)

* Skips the first n elements and processes the rest.

forEach()

* Performs an action (like printing) for each element.

collect()

* Gathers the elements into a list, set, map, or other result containers.

reduce()

* Reduces the stream to a single value (like sum, min, max, etc.)

**Task 13**  
Wap to create an arrayList of your friends using string and try to sort them and display

**package** samplePackage2;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** FriendListSort {

**public** **static** **void** main(String[] args) {

// Step 1: Create an ArrayList of friend names

List<String> friends = **new** ArrayList<>();

friends.add("Zara");

friends.add("Amit");

friends.add("Nikhil");

friends.add("Ravi");

friends.add("Bina");

// Step 2: Sort and display the friend names using Stream

System.***out***.println("Sorted Friend List:");

friends.stream()

.sorted() // Sort in alphabetical order

.forEach(name -> System.***out***.println(name));

}

}

Output:- Sorted Friend List:

Amit

Bina

Nikhil

Ravi

Zara

**Task 14**Wap to run a loop / iterate()  and limit it to 20 values (1 to 2)

While displaying use for each to limit till 10 numbers.

**package** samplePackage2;

**import** java.util.stream.Stream;

**public** **class** StreamIterateExample {

**public** **static** **void** main(String[] args) {

System.***out***.println("First 10 values (odd numbers):");

// Generate a stream of numbers starting from 1, incremented by 2

Stream.*iterate*(1, n -> n + 2)

.limit(20) // generate first 20 odd numbers

.limit(10) // further limit to display only first 10

.forEach(System.***out***::println); // print each number

}

}

Output:- First 10 values (odd numbers):

1

3

5

7

9

11

13

15

17

19

**Tasks 14**Wap to create an array List skip 15 numbers and print the output using foreach loop

**package** samplePackage2;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** SkipExample {

**public** **static** **void** main(String[] args) {

// Step 1: Create an ArrayList with numbers 1 to 30

List<Integer> numbers = **new** ArrayList<>();

**for** (**int** i = 1; i <= 30; i++) {

numbers.add(i);

}

// Step 2: Skip the first 15 numbers and print the rest

System.***out***.println("Numbers after skipping first 15:");

numbers.stream()

.skip(15) // skips the first 15 elements

.forEach(n -> System.***out***.println(n)); // prints remaining

}

}

Output:- Numbers after skipping first 15:

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

**Task 15**Wap to create an array List skip 15 numbers and print the output using foreach loop

import java.util.ArrayList;

import java.util.List;

public class SkipExample {

public static void main(String[] args) {

// Create an ArrayList with numbers 1 to 30

List<Integer> numbers = new ArrayList<>();

for (int i = 1; i <= 30; i++) {

numbers.add(i);

}

// Skip the first 15 numbers and print the rest using forEach

System.out.println("Numbers after skipping first 15:");

numbers.stream()

.skip(15)

.forEach(n -> System.out.println(n));

}

}  
Numbers after skipping first 15:

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30  
  
**Task 16**Explain limit and skip methods..  
Answer:-  
**limit(n) – Restrict the number of elements**

The limit(n) method is used to take only the **first n elements** from the stream. It helps when you want to fetch a fixed number of records (like top 5 names, first 10 numbers, etc.).

**skip(n) – Ignore the first n elements**

The skip(n) method is used to **ignore or skip** the **first n elements** and continue processing from the (n+1)th item onward. This is useful when you want to ignore some records from the start.

**collect()**

**Purpose:** To **gather** the results of a stream into a collection (like List, Set, Map) or into a single summary result.

**forEach()**

**Purpose:** To **perform an action** (like printing) on each element in the stream.

Does not return anything.  
  
**reduce()**

**Purpose:** To **combine all elements** of the stream into a **single result** using a binary operation (like sum, product, min, max).

It's powerful but can be slightly complex.

**Task 17**   
import java.util.Arrays;

import java.util.List;

import java.util.Optional;

public class ReduceExample {

public static void main(String[] args) {

// List of integers

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Sum of all numbers using reduce

Optional<Integer> sum = numbers.stream()

.reduce((x, y) -> x + y);

System.out.println("Sum of all elements: " + sum.orElse(0));

// Maximum number using reduce

Optional<Integer> max = numbers.stream()

.reduce(Integer::max);

System.out.println("Maximum element: " + max.orElse(0));

// List of strings

List<String> strings = Arrays.asList("Hello", " ", "world", "!");

// Concatenate strings using reduce

Optional<String> concatenatedString = strings.stream()

.reduce((x, y) -> x + y);

System.out.println("Concatenated string: " + concatenatedString.orElse(""));

}

}  
output:- Sum of all elements: 15

Maximum element: 5

Concatenated string: Hello world!   
  
  
**Task 18**Advantages of Streams:

✅ 1. Clean and Readable Code

Streams allow you to process data in a clear and concise way. Instead of writing long for loops and if conditions, you can write one-liner operations like filter(), map(), or sorted() which are easy to read and understand.

✅ 2. Less Boilerplate Code

With streams, there is no need to write extra code for looping, condition checking, or result collection. This reduces clutter and keeps your code short and focused.

✅ 3. Supports Functional Programming

Streams use functional-style programming with lambda expressions. This encourages writing code in a declarative style (what to do) instead of imperative (how to do it).

✅ 4. Chain Multiple Operations

You can easily combine multiple steps like filtering, mapping, and sorting using method chaining, making the code flow naturally from one transformation to the next.

list.stream().filter(...).map(...).sorted().collect(...)

✅ 5. Parallel Processing Made Easy

Streams support parallel execution with parallelStream(), which allows your data to be processed faster by using multiple CPU cores—without writing multithreaded code manually.

✅ 6. Immutable Data Handling

Stream operations do not modify the original data. They return new results, which helps maintain data integrity and avoid bugs caused by unintended data changes.

✅ 7. Lazy Evaluation

Streams are lazy, meaning intermediate operations like filter() or map() are not executed until a terminal operation like forEach() or collect() is called. This helps improve performance.

✅ 8. Built-in Utility Methods

Streams provide powerful built-in methods like min(), max(), sum(), distinct(), limit(), skip(), reduce(), etc., which make data processing quick and easy.

**Task 19**What are the debugging tools in Java.. list down a few..

Debugging tools are software programs or features within IDEs (like Eclipse, IntelliJ, or VS Code) that help developers find and fix bugs or errors in their code. They allow you to track the flow of a program, inspect variable values, and understand what your code is doing step-by-step during execution.  
  
  
**Task 20**Here's a clear explanation of **Task 19: Error Messages in Java**, including **compile-time** and **run-time** errors:

**✅ Compile-Time Errors in Java:**

These are errors that occur **when you try to compile the code**. They are usually **syntax or grammatical mistakes**.

**🔹 Common Compile-Time Errors:**

* Missing **semicolon (;)**
* Mismatched **braces {} or parentheses ()**
* Using **undeclared variables**
* Calling methods or classes that **don’t exist**
* Wrong spelling or case issues (Java is case-sensitive)

**🔸 Example:**

int x = 10

System.out.println(x); // ❌ Missing semicolon above

**✅ Run-Time Errors (Exceptions):**

These happen **while the program is running**, even if the code compiles properly. These are usually **logic or data issues**.

**🔹 Common Run-Time Errors:**

1. **NullPointerException**  
   – Accessing methods/fields on a null object.
2. String name = null;
3. System.out.println(name.length()); // ❌ NullPointerException
4. **ArrayIndexOutOfBoundsException**  
   – Trying to access an invalid array index.
5. int[] arr = {1, 2, 3};
6. System.out.println(arr[5]); // ❌ Index out of bounds
7. **IOException**  
   – File not found, network issues, input/output problems.
8. BufferedReader br = new BufferedReader(new FileReader("file.txt")); // ❌ File may not exist
9. **StackOverflowError**  
   – Caused by infinite or deep recursion.
10. public void call() {
11. call(); // ❌ Stack overflow
12. }

**🧠 Summary:**

| **Type** | **When It Happens** | **Example** |
| --- | --- | --- |
| Compile-Time | While writing code | Missing ;, wrong syntax |
| Run-Time (Exception) | During execution | NullPointer, ArrayIndex, IO etc. |
| Error | Critical failures | StackOverflowError |

**Task 21**  
import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

public class Example {

    private static final Logger logger = LogManager.getLogger(Example.class);

    public static void main(String[] args) {

        int x = 5;

        int y = 7;

        int sum = x + y;

        logger.debug("x = " + x);

        logger.debug("y = " + y);

        logger.debug("sum = " + sum);

    }

}

Output:- DEBUG Example - x = 5

DEBUG Example - y = 7

DEBUG Example - sum = 12