Day 11 - 19th June 2025

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Java 8 Features

👍

Streams

Lambda Exp

Functional Interfaces

Concrete methods….we can body of the methods

Default methods

Static methods

Collection API

Fork join methods

Method ref - class name :: method name

Comparable and comparators

Abstract and private methods

NIO

=====================

C++ – pure virtual functions

Void display() = 0;

=========================

Extends Class Name

Implement Interface Name

STreams —-- stream of data..

**Task 1:**

What are streams?

Streams are used when we want to read or write data, like from a file, network connection, or memory. We can use them when dealing with bytes or characters

### **Examples:**

* InputStream – Reads data (bytes)
* OutputStream – Writes data (bytes)
* Reader – Reads text (characters)
* Writer – Writes text (characters)

3min - 09.56 to 10.00

**Task 2:**

Write about :

**Boilerplate code:**

Boilerplate code in Java is to keep writing again and again like class structures, getters, setters, and constructors —even if it's the same every time. It doesn’t do anything unique, just fills space to make Java run properly.

For expml: for loop and while loop.

**Lack of parallelism:**

It means your code runs tasks one after another instead of using multiple CPU cores to run them at the same time. This can slow things down, especially when you're working with large data or heavy processing.

**Lack of Composition:**

Lack of composition means your code is all jammed into one piece instead of being broken into small, reusable parts. Good composition makes your code easier to reuse, test, and understand.

Task 3:

List of Intermediate and terminal operations

1. **Intermediate Operators** :

These are methods that transform or filter the stream they return a new stream, allowing further operations to be chained.

| **Method** | **What it does** |
| --- | --- |
| filter() | Keeps only elements that match a condition |
| map() | Transforms each element |
| sorted() | Sorts the stream |
| distinct() | Removes duplicates |
| limit(n) | Limits to n elements |
| skip(n) | Skips first n elements |
| peek() | Allows debugging (prints/logs) |

2. **Terminal Operators:**

These are the final operations that trigger the execution of the stream pipeline. Once a terminal operation is called, the stream is consumed and can’t be used again.

| **Method** | **What it does** |
| --- | --- |
| **collect()** | **Gathers result into a list, set, etc.** |
| **forEach()** | **Performs an action on each element** |
| **count()** | **Counts elements** |
| **findFirst()** | **Gets the first matching element** |
| **anyMatch()** | **Checks if any element matches** |
| **allMatch()** | **Checks if all match** |
| **reduce()** | **Combines elements into one** |

Intermediate operators prepare or transform the data but don’t actually run anything. Terminal operators are the final action that runs the whole stream and gives the result.

========================================================================

Lambda Expressions

========================================================================

**Task 4:**

import java.lang.FunctionalInterface;

// this is functional interface

@FunctionalInterface

interface MyInterface{

// abstract method

double getPiValue();

}

public class Main {

public static void main( String[] args ) {

// declare a reference to MyInterface

MyInterface ref;

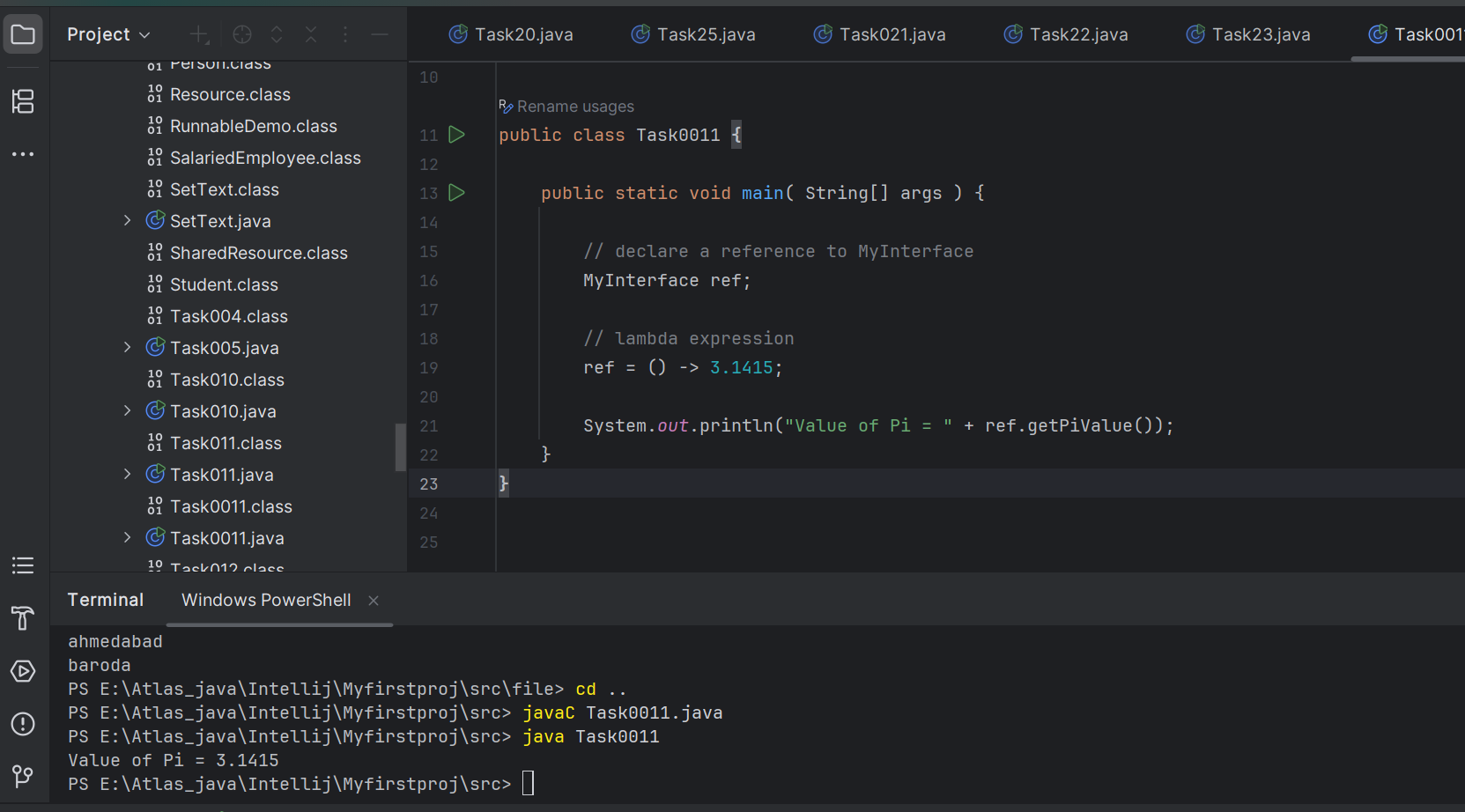
// lambda expression

ref = () -> 3.1415;

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

}

}



==================================

Task 5:

@FunctionalInterface

interface MyInterface {

// abstract method

String reverse(String n);

}

public class Main {

public static void main( String[] args ) {

// declare a reference to MyInterface

// assign a lambda expression to the reference

MyInterface ref = (str) -> {

String result = "";

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

result += str.charAt(i);

return result;

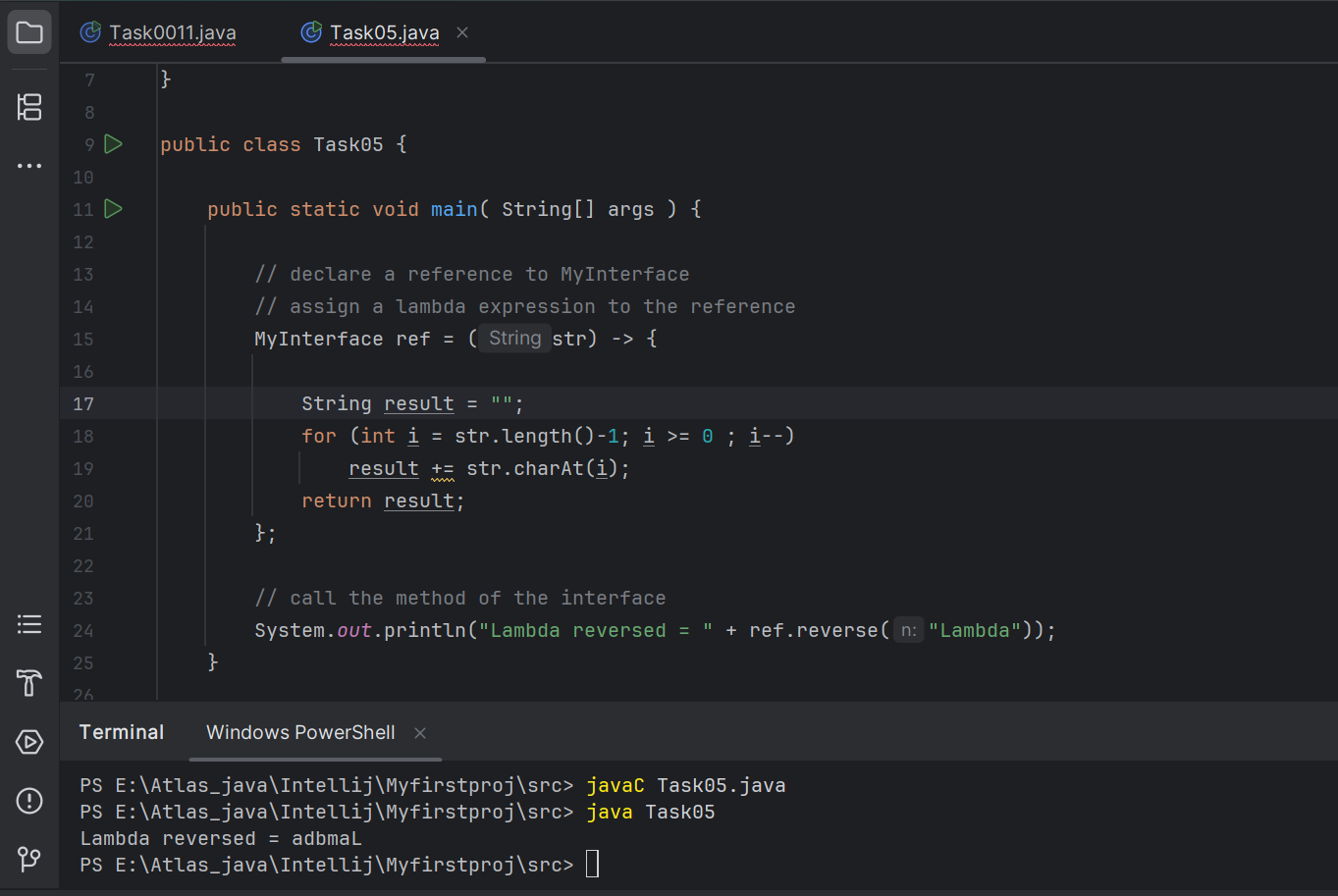
};

// call the method of the interface

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

}

}



==================================

Task 6:

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"))

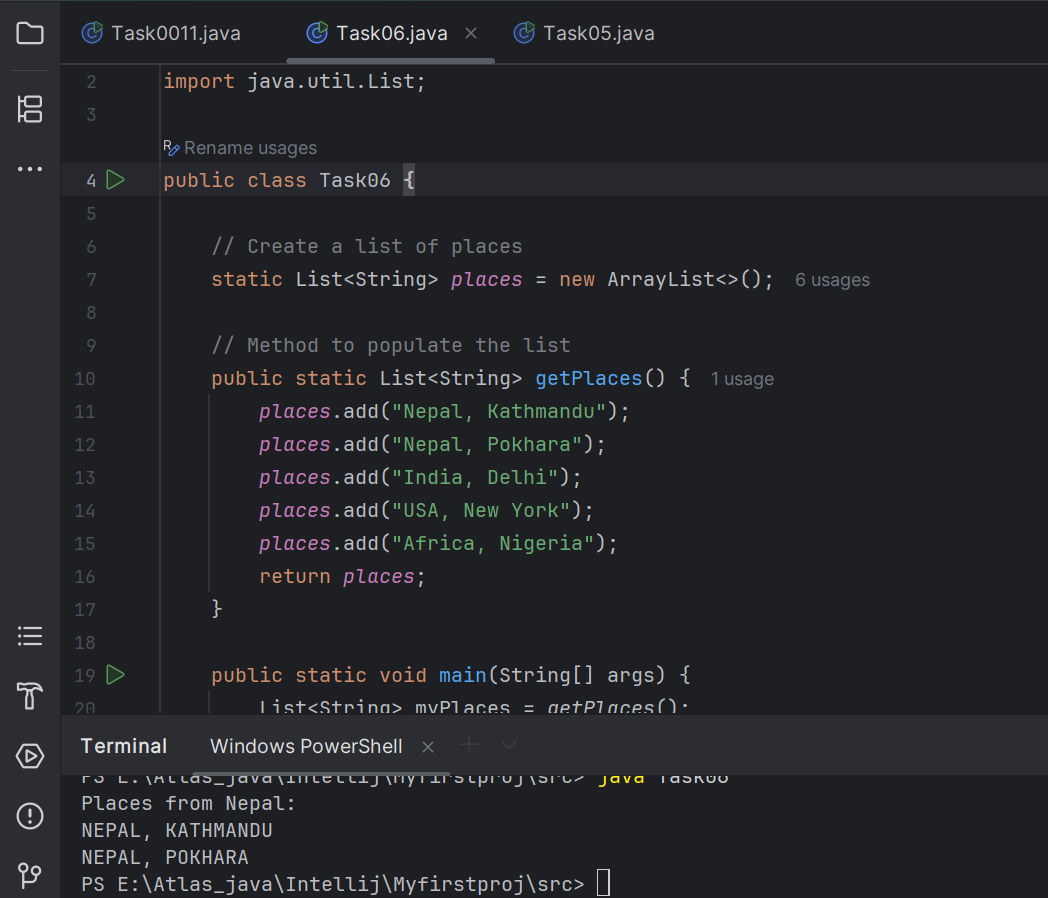
.map((p) -> p.toUpperCase())

.sorted()

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

}

}



**Task 7:**

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

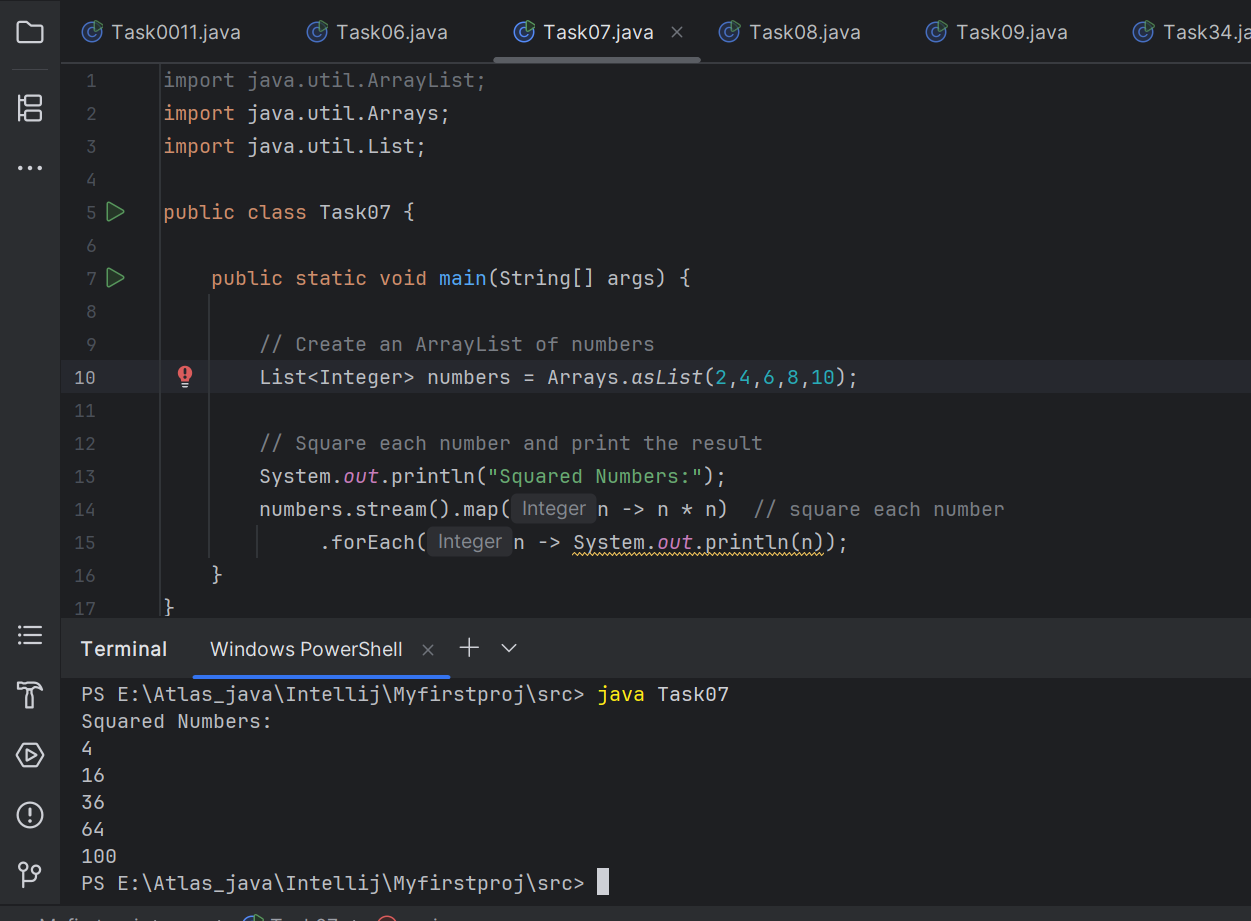
Hint:

List<Integer> squareofNums = numbers.stream()

.map(num->num\*num).

collect(Collectors.toList());

12.01 to 12.05



Task 8:

What do you understand by map()?

The map() method is an intermediate operation used to transform each element of a stream.

It takes each item in the stream, applies a function to it, and returns a new stream with the transformed elements.

12.06 to 12.09

Task 9:

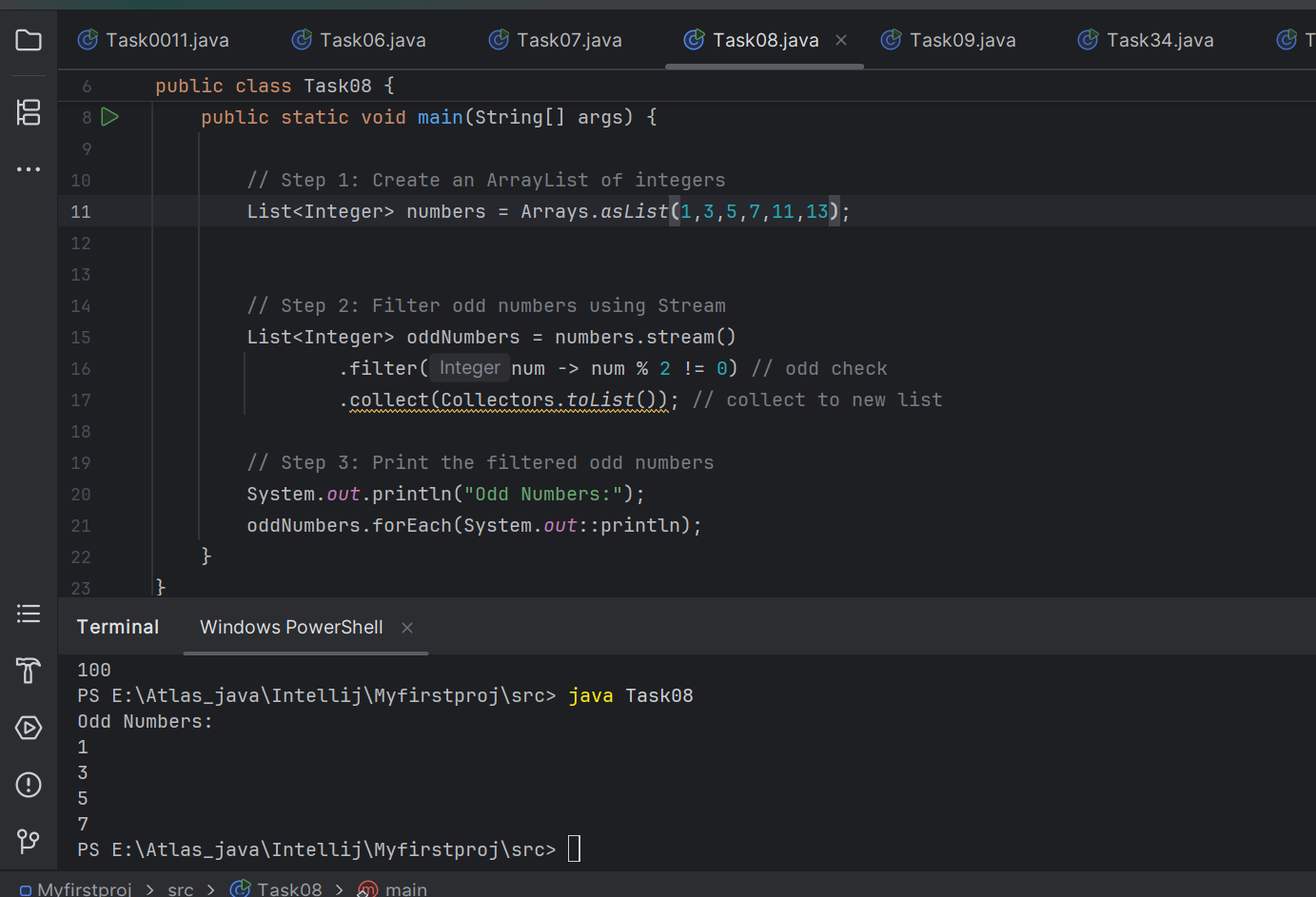
Write a code to create an array list and filter the values which are odd numbers and display them..

Hint:

List<Integer> addNumbers = numbers.stream()

.filter(num -> num % 2 !=0)

.collect(Collectors.toList());



Task 10:

What do you understand by filter()?

It is used to remove unwanted elements from a stream based on a condition (true/false).

12.16 to 12.18

Task 11:

Wap to create an array list to remove duplicate values from the List.

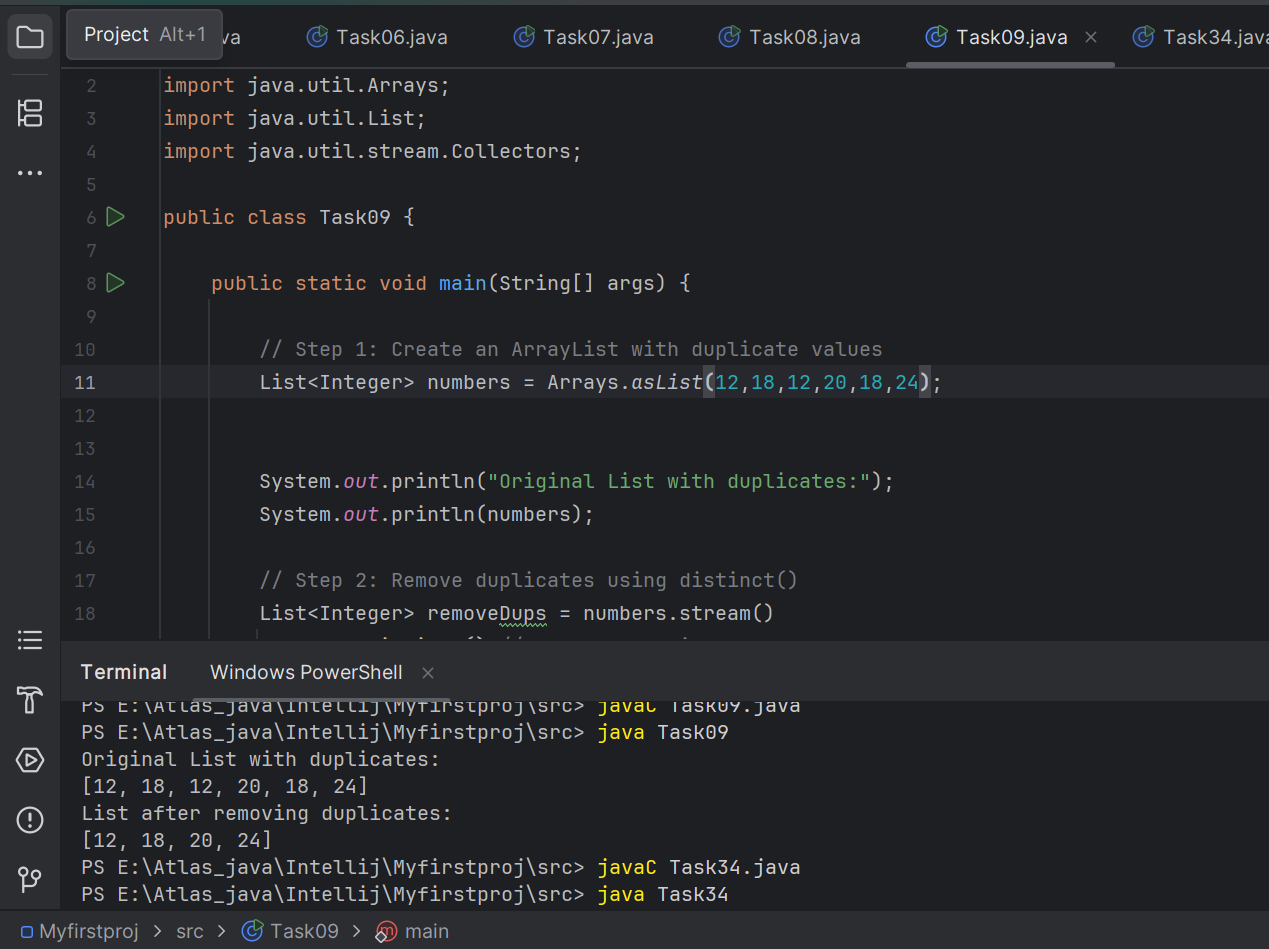
Hint:

List<Integer> RemovDups= numbers.stream()

.distinct()

.collect(Collectors.toList());

12.18 to 12.23



Task12:

What do you understand by distinct()?

distinct() keeps only unique values in the stream and removes all duplicates.

12.23 to 12.25

**Intermediate**

| **Operation** | **Type** | **Description** | **Example** |
| --- | --- | --- | --- |
| sort() | Intermediate | Sorts the elements of the stream | .sorted() |
| limit(n) | Intermediate | Returns only the first n elements of the stream | .limit(3) |
| skip(n) | Intermediate | Skips the first n elements in the stream | .skip(2) |

**Terminal op**

| **Operation** | **Description** | **Returns** | **Example** |
| --- | --- | --- | --- |
| forEach() | Performs an action (like printing) on each element of the stream | void (no return) | stream.forEach(System.out::println) |
| collect() | Gathers the elements into a collection like List, Set, or Map | A collection | stream.collect(Collectors.toList()) |
| reduce() | Reduces all elements to a single value using a binary operation (e.g., sum) | A single value (int, etc.) | stream.reduce(0, (a, b) -> a + b) |

Task 13:

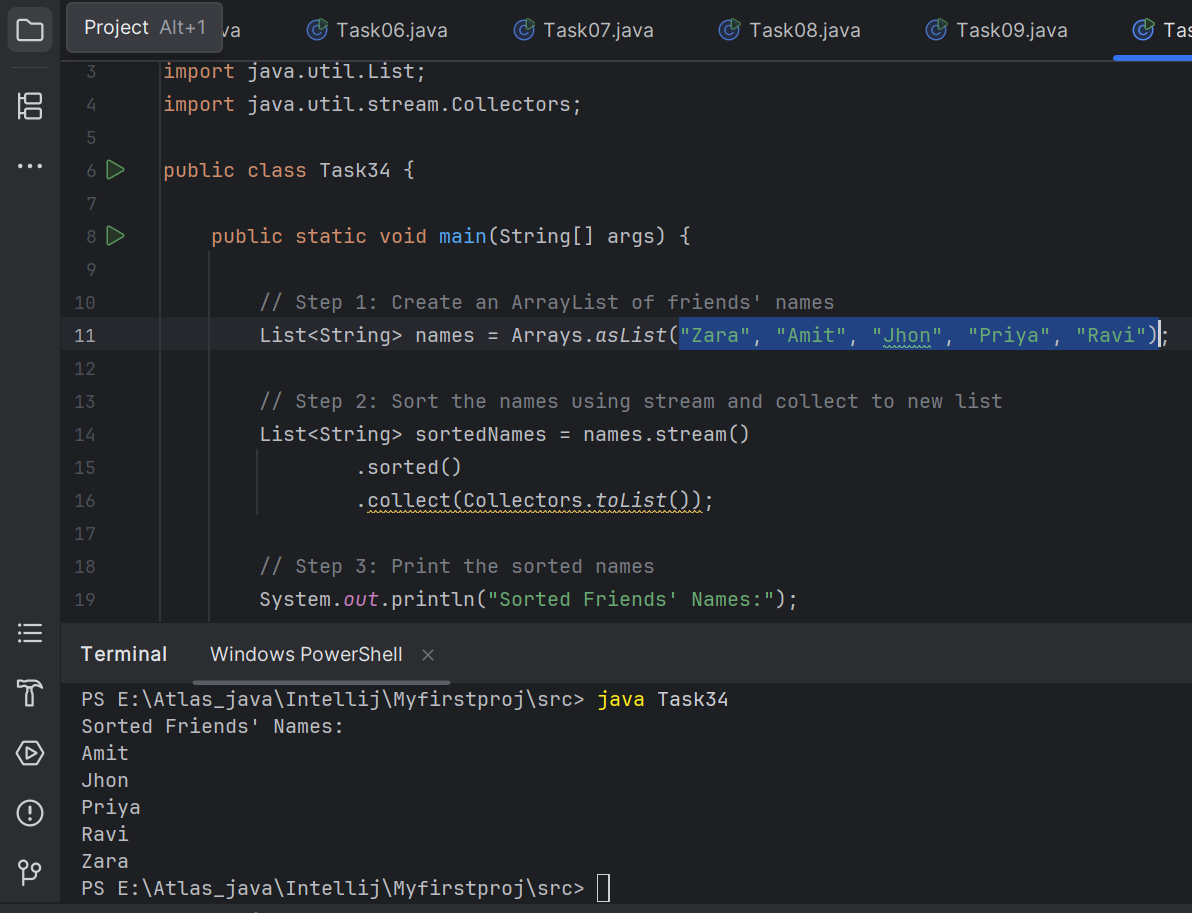
Wao to create an arrayList of your friends using string and try to sort them and display

Hint:

List<String> SortedNames= Names.stream()

.sorted()

.collect(Collectors.toList());



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.

Hint:

Stream<Integers> nums = Stream

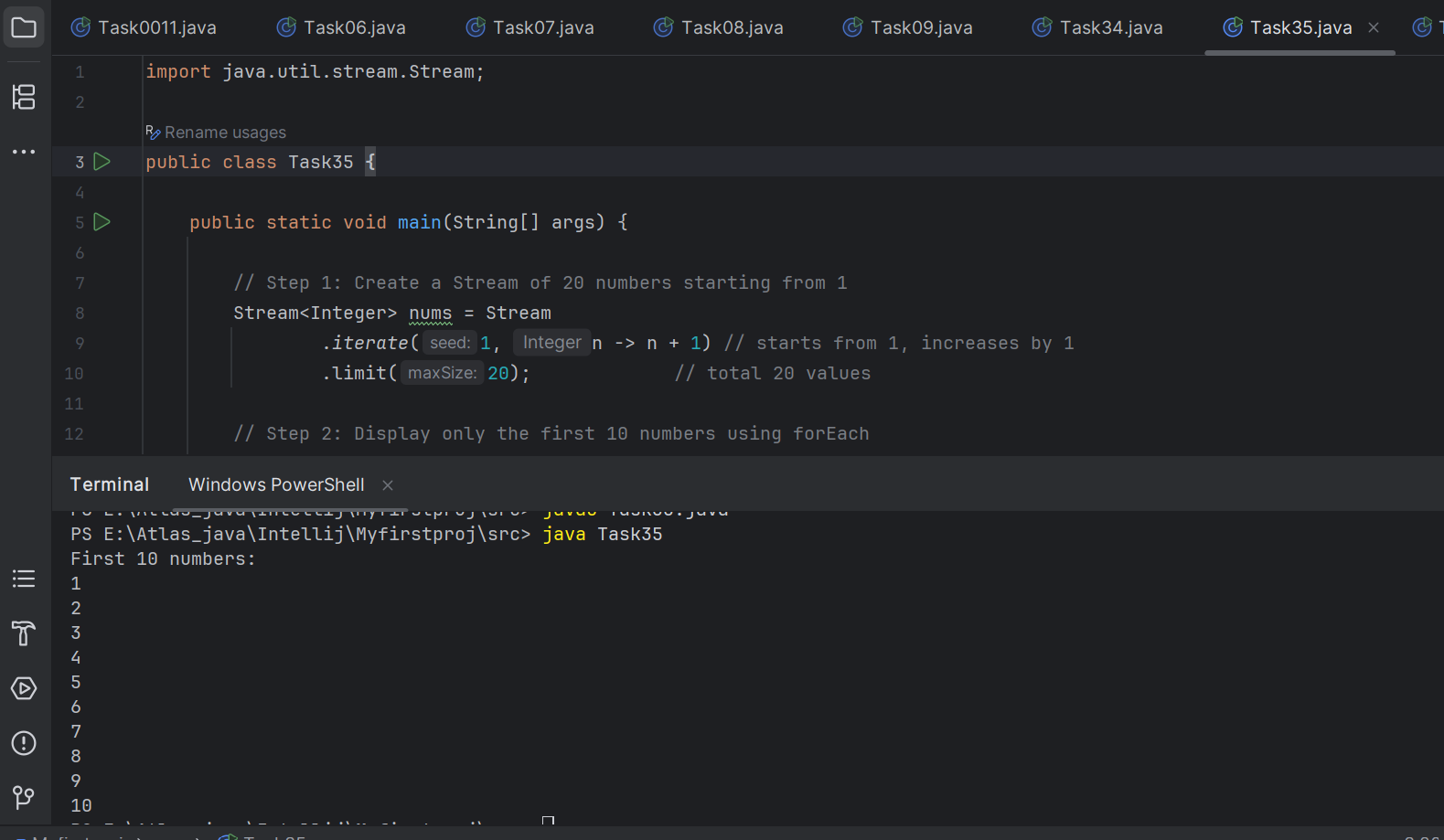
.iterate(1, n -> n+1)

.limit(20);

Nums

.limit(10)

.foreach(System.out::println);



Task 15: (similar to Task 14)

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

HInt:

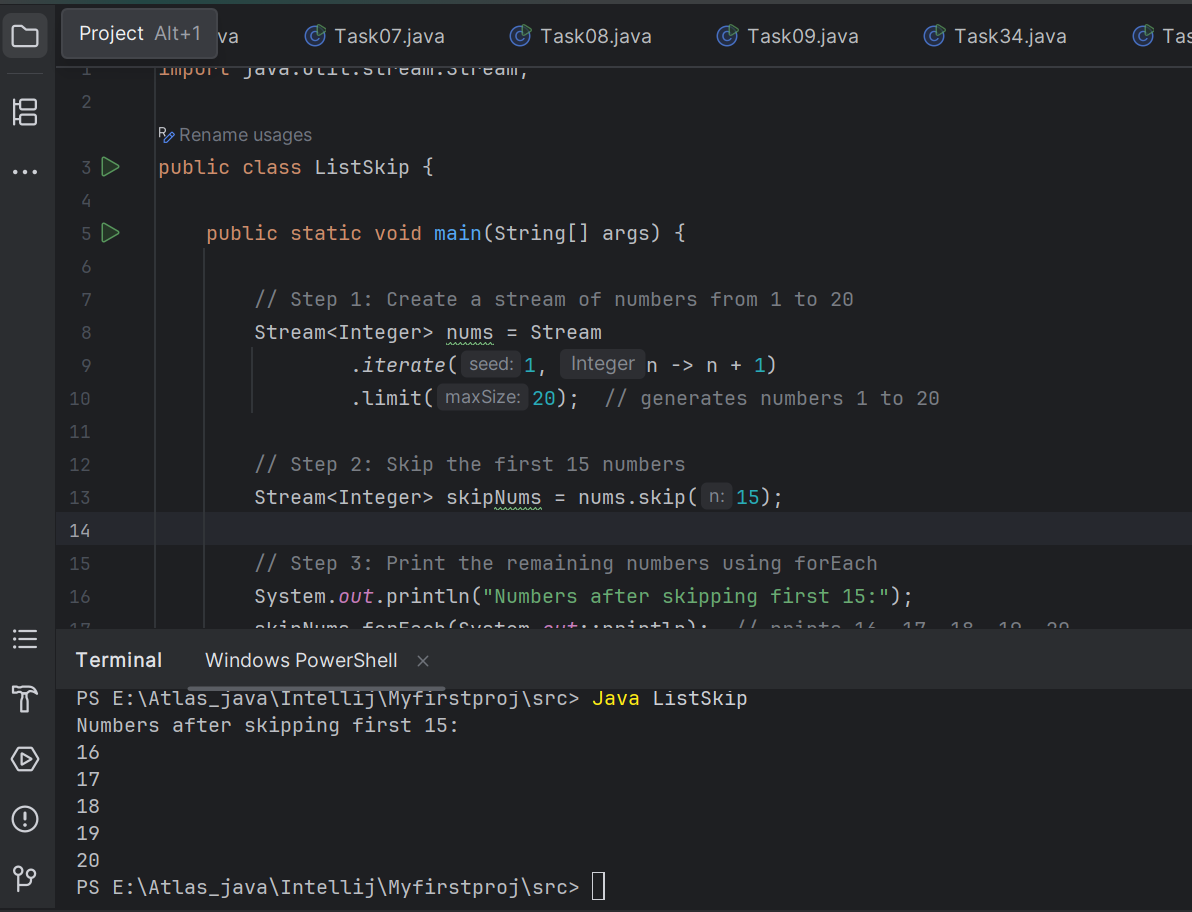
Stream<Integers> nums = Stream

.iterate(1, n -> n+1)

.limit(20);

Stream<Integer> SkipNums = nums.skip(15);

Nums.foreach(System.out::println);



12.36 to 12.40

Task 16:

Explain limit and skip methods..

**Limit method :**

The limit(n) method is used to restrict the stream to only the first n elements

.

Syntax: stream.limit(n)

Expl:

Stream<Integer> stream = Stream.iterate(1, n -> n + 1).limit(5);

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

Op:

1

2

3

4

5

**skip method:**

The skip(n) method is used to skip the first n elements of the stream.

Syntax: stream.skip(n)

Ex: Stream<Integer> stream = Stream.iterate(1, n -> n + 1).limit(10).skip(5);

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

Op:

6

7

8

9

10

12.41 to 12.44

Terminal operators:

Collect 👍

Done in all the above tasks to display the output of intermediate Tasks

For each 👍

You guys are familiar with for each…

reduce(): The reduce() method is a terminal operation that reduces the elements of a stream into a single result like sum, product, string concatenation, etc.

It is immutable.

Task 17:

What is the difference between mutable and immutable?

| **Feature** | **Mutable** | **Immutable** |
| --- | --- | --- |
| Definition | Objects whose state can be changed after creation | Objects whose state cannot be changed once created |
| Modification | You can change, update, or delete values | Any change creates a new object |
| Examples | ArrayList, HashMap, StringBuilder | String, Integer, LocalDate |
| Memory usage | More efficient (in-place updates) | May use more memory (due to new objects) |
| Thread-safety | Usually not thread-safe | Often thread-safe |

Task 18:

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

public class ReduceExample {

public static void main(String[] args) {

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

Optional<Integer> sum = numbers.stream().reduce((x, y) -> x + y);

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

Optional<Integer> max = numbers.stream().reduce(Integer::max);

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

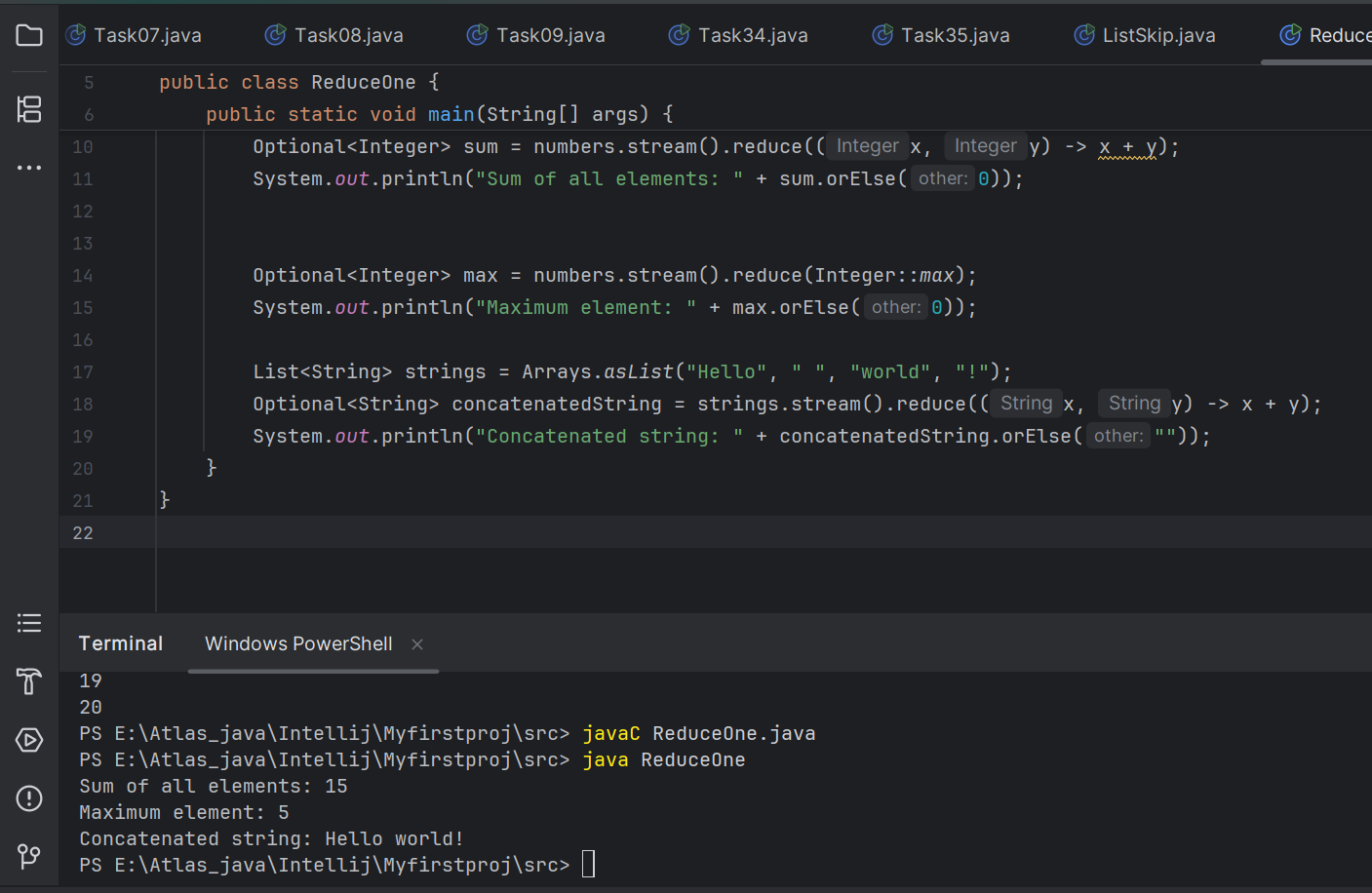
List<String> strings = Arrays.asList("Hello", " ", "world", "!");

Optional<String> concatenatedString = strings.stream().reduce((x, y) -> x + y);

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

}

}



**When to use reduce and when to use collect..**

* Reduce will be used if you are expecting a single result from the stream (eg min, max , sum, product…)
* Collect will be used if you are excepting a list of values… (list, set, map)

**Advantages of Streams:**

| **Advantage** | **Description** |
| --- | --- |
| **1. Clean & Concise Code** | Simplifies complex logic into readable, chainable methods. |
| **2. No Explicit Loops** | Eliminates the need for for/while loops by using internal iteration. |
| **3. Functional Style** | Supports lambda expressions and encourages declarative programming. |
| **4. Lazy Evaluation** | Intermediate operations are only executed when a terminal operation is called. |
| **5. Parallel Processing** | Easy performance boost using .parallelStream() for multi-core execution. |
| **6. Immutable Data Processing** | Streams don’t change the original data — safer and side-effect free. |
| **7. Easy Chaining** | Combine multiple operations (filter, map, sorted, etc.) in a flow. |
| **8. Built-in Operations** | Rich set of methods like filter(), map(), reduce(), collect() etc. |

Task 19:

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

* **db (Java Debugger)** – Command-line debugging tool provided by the JDK.
* **IntelliJ IDEA Debugger** – Feature-rich debugger built into IntelliJ IDE.
* **Eclipse Debugger** – Integrated graphical debugger in the Eclipse IDE.
* **NetBeans Debugger** – Built-in debugger with GUI support in NetBeans.
* **VisualVM** – Tool for monitoring, profiling, and debugging Java applications.
* **jstack** – Command-line tool to generate thread dumps.
* **jconsole** – GUI tool for monitoring JVM performance via JMX.
* **JProfiler** – Commercial tool for profiling and debugging performance issues.
* **YourKit Java Profiler** – Advanced commercial profiler for memory, CPU, and threads.
* **Log4j / SLF4J / java.util.logging** – Logging frameworks often used for tracking application behavior and debugging.

3 min 16.04 to 16

Task 20:

Error Messages in Java

Compile time and run time

Compile time : grammatical mistakes … ;, {} , missing the code

Run time error or exceptions

Stack overflow error

Array index out of bounds

IO exception

Nulpointer exception

**done**

Task 21:

Stack trace.. What will it do?

A stack trace in Java is a report that shows the sequence of method calls that led to an error (usually an exception). It's printed by the JVM when your program throws an uncaught exception.

* Helps locate bugs in your code
* Shows the path of execution leading to the error
* Identifies the exact file and line number where the error occurred
* Gives insight into which method called what — all the way up the call stack

Identify the error

Locate the code

Analyze the code

Solution also