Day 11 - 21 June 2025

# **Day 11 - 21 June 2025**

**Document Name:**Day 11 - hmuvvala@ - Hari Gopal Muvvala

### Task 1 – Write Byte Array to File

#### Code: Task1\_WriteByte.java

byte[] text = { 'I', ' ', 'L', 'O', 'V', 'E', ' ', 'I', 'N', 'D', 'I', 'A' };

FileOutputStream outFile = new FileOutputStream("FileName01.txt");

outFile.write(text);

#### Output (file):

I LOVE INDIA

#### Notes:

* Uses FileOutputStream to write raw bytes.
* Writes each character as a byte.
* File is created or overwritten in the project folder.

### Task 2 – Read Bytes from File

#### Code: Task2\_ReadingByte.java

FileInputStream infile = new FileInputStream("FileName01.txt");

int b;

while ((b = infile.read()) != -1) {

System.out.println((char) b);

}

#### Output:

Characters printed one by one vertically.

#### Notes:

* Reads 1 byte at a time.
* Converts to character using (char).
* Stops at EOF (-1).

### Task 3 – Take Input and Write to File

#### Code: Task3\_WriteUserInput.java

Scanner sc = new Scanner(System.in);

String s = sc.nextLine();

FileOutputStream outFile = new FileOutputStream("FileName02.txt");

outFile.write(s.getBytes());

#### Output:

Writes whatever the user enters to FileName02.txt

#### Notes:

* Takes user input using Scanner.
* Converts input string to bytes.
* Writes dynamically typed content into file.

### Task 4 – Write Characters Using FileWriter

#### Code: Task4\_CharacterWrite.java

FileWriter fw = new FileWriter("FileName03.txt");

fw.write("ahmedabad\n");

fw.write("baroda\n");

fw.close();

#### Output (file):

ahmedabad

baroda

#### Notes:

* Uses FileWriter for character writing.
* \n for new line.
* Good for writing text-based files.

### Task 5 – Read Characters Using FileReader

#### Code: Task5\_ReadChar.java

FileReader fr = new FileReader("FileName03.txt");

int ch;

while ((ch = fr.read()) != -1) {

System.out.print((char) ch);

}

#### Output:

ahmedabad

baroda

#### Notes:

* Uses FileReader to read one character at a time.
* Works well with text files.
* Print remains inline, unlike println.

### Task 6 – Copy File Byte-by-Byte

#### Code: Task6\_CopyByte.java

FileInputStream inFile = new FileInputStream("FileName01.txt");

FileOutputStream outFile = new FileOutputStream("FileName05.txt");

int byteRead;

while ((byteRead = inFile.read()) != -1) {

outFile.write(byteRead);

}

#### Output (file):

Exact same content as FileName01.txt.

#### Notes:

* Copies bytes one at a time.
* Uses raw byte streams (not characters).
* No decoding or formatting involved.

### Task 7 – Merge Two Files into One

#### Code: Task7\_MergeFiles.java

FileInputStream file1 = new FileInputStream("FileName01.txt");

FileInputStream file2 = new FileInputStream("FileName03.txt");

SequenceInputStream combined = new SequenceInputStream(file1, file2);

BufferedInputStream br = new BufferedInputStream(combined);

BufferedOutputStream bw = new BufferedOutputStream(new FileOutputStream("FileName07.txt"));

int ch;

while ((ch = br.read()) != -1) {

bw.write(ch);

}

#### Output (file):

I LOVE INDIA

ahmedabad

baroda

#### Notes:

* Merges contents of two files using SequenceInputStream.
* Buffered streams make reading/writing efficient.
* Appends content in sequence (not line-by-line logic).

### Task 8 – Rename a File

#### Code: Task8\_RenameFile.java

File oldFile = new File("FileName07.txt");

File newFile = new File("RenamedFile07.txt");

if (oldFile.renameTo(newFile)) {

System.out.println("File renamed successfully.");

} else {

System.out.println("Rename failed.");

}

#### Output:

File renamed successfully.

#### Notes:

* Uses File.renameTo() to rename a file.
* Fails silently if file doesn’t exist or is in use.
* No terminal needed; filenames are hardcoded.

### Task 9

#### Lambda Expression to Check String Length

package day11;

interface Checker {

boolean check(String str);

}

public class Task9\_StringLengthChecker {

public static void main(String[] args) {

Checker checker = (str) -> str.length() > 5;

System.out.println(checker.check("Hello")); // false

System.out.println(checker.check("Greetings")); // true

}

}

#### Output:

false

true

#### Understanding:

* Lambda used to implement functional interface Checker
* (str) -> str.length() > 5 is the lambda expression
* Checks whether string length is greater than 5

### Task 10

#### Reverse a String using Lambda

package day11;

import java.util.function.Function;

public class Task10\_ReverseString {

public static void main(String[] args) {

Function<String, String> reverse = str -> new StringBuilder(str).reverse().toString();

System.out.println(reverse.apply("Java"));

}

}

#### Output:

avaJ

#### Understanding:

* Used Function<T, R> interface
* Lambda reverses the string using StringBuilder

### Task 11

#### ArrayList of City Names with Lambda

package day11;

import java.util.\*;

public class Task11\_CityNamesLambda {

public static void main(String[] args) {

List<String> cities = Arrays.asList("Hyderabad", "Chennai", "Bangalore", "Delhi", "Mumbai");

cities.forEach(city -> System.out.println(city));

}

}

#### Output:

Hyderabad

Chennai

Bangalore

Delhi

Mumbai

#### Understanding:

* Used forEach() to print each city
* city -> System.out.println(city) is the lambda expression

### Task 12

#### Create ArrayList of 5 Friends

package day11;

import java.util.\*;

public class Task12\_FriendsList {

public static void main(String[] args) {

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

friends.add("Hemanth");

friends.add("Divya");

friends.add("Sai");

friends.add("Vaibhav");

friends.add("Ravi");

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

}

}

#### Output:

Hemanth

Divya

Sai

Vaibhav

Ravi

#### Understanding:

* ArrayList stores 5 names
* Printed using forEach + method reference

### Task 13

#### List with Full Names

package day11;

import java.util.\*;

public class Task13\_FullNameList {

public static void main(String[] args) {

List<String> friends = Arrays.asList(

"Hemanth Bommena",

"Divya Bandi",

"Sai Yellanki",

"Vaibhav Raj",

"Ravi Kopparapu"

);

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

}

}

#### Output:

Hemanth Bommena

Divya Bandi

Sai Yellanki

Vaibhav Raj

Ravi Kopparapu

#### Understanding:

* Used List.of() or Arrays.asList() for storing full names
* Readable format to support future stream operations

### Task 14

#### Filter, Map, Sort with Streams

package day11;

import java.util.\*;

public class Task14\_FriendStreamOps {

public static void main(String[] args) {

List<String> fullNames = Arrays.asList(

"Hemanth Bommena",

"Divya Bandi",

"Sai Yellanki",

"Vaibhav Raj",

"Ravi Kopparapu"

);

fullNames.stream()

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

.map(String::toUpperCase)

.sorted()

.forEach(System.out::println);

}

}

#### Output:

DIVYA BANDI

#### Understanding:

* filter() to select names starting with ‘D’
* map() to uppercase
* sorted() and forEach() for final display

### Task 15

#### Square Numbers using map() and collect()

package day11;

import java.util.\*;

import java.util.stream.Collectors;

public class Task15\_SquareOfNumbers {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(2, 4, 6, 8, 10);

List<Integer> squares = numbers.stream()

.map(num -> num \* num)

.collect(Collectors.toList());

System.out.println("Squares: " + squares);

}

}

#### Output:

Squares: [4, 16, 36, 64, 100]

#### Understanding:

* map() transforms each number to its square
* collect() stores result into a new list

### Task 16

#### Filter Odd Numbers

package day11;

import java.util.\*;

import java.util.stream.Collectors;

public class Task16\_FilterOddNumbers {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(11, 22, 33, 44, 55);

List<Integer> oddNumbers = numbers.stream()

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

.collect(Collectors.toList());

System.out.println("Odd Numbers: " + oddNumbers);

}

}

#### Output:

Odd Numbers: [11, 33, 55]

#### Understanding:

* filter() used to select only odd numbers

### Task 17

#### Remove Duplicate Values

package day11;

import java.util.\*;

import java.util.stream.Collectors;

public class Task17\_RemoveDuplicates {

public static void main(String[] args) {

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

List<Integer> uniqueNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println("Unique Numbers: " + uniqueNumbers);

}

}

#### Output:

Unique Numbers: [10, 20, 30, 40, 50]

#### Understanding:

* .distinct() removes repeated values in stream

### Task 18

#### Stream Iterate and Limit

package day11;

import java.util.stream.Stream;

public class Task18\_StreamIterateLimit {

public static void main(String[] args) {

Stream<Integer> numbers = Stream.iterate(1, n -> n + 1).limit(20);

numbers.limit(10).forEach(System.out::println);

}

}

#### Output:

1

2

3

...

10

#### Understanding:

* Stream.iterate() starts from 1
* .limit(20) creates 20 values
* .limit(10) displays only first 10

### Task 19

#### Stream Skip

package day11;

import java.util.stream.Stream;

public class Task19\_StreamSkip {

public static void main(String[] args) {

Stream<Integer> numbers = Stream.iterate(1, n -> n + 1).limit(20);

numbers.skip(15).forEach(System.out::println);

}

}

#### Output:

16

17

18

19

20

#### Understanding:

* .skip(15) skips the first 15 elements
* Displays last 5 values from 16 to 20

### Task 20

#### Reduce (sum, max, concatenation)

package day11;

import java.util.\*;

public class Task20\_ReduceExamples {

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: " + sum.orElse(0));

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

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

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

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

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

}

}

#### Output:

Sum: 15

Max: 5

Concatenated: Hello world!

#### Understanding:

* .reduce() is used for single result: sum, max, or combined string
* Optional used to handle empty streams safely