CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY (CHARUSAT)
FACULTY OF TECHNOLOGY AND ENGINEERING (FTE)
SUBJECT: OBJECT-ORIENTED PROGRAMMING (CEUE203)
SEMESTER: 3RD, 2025-26 (ODD)
PRACTICAL LIST

Practical Number	Title	CO/PO
	CLASS FUNDAMENTALS	
1	**Problem Definition:** **Problem Definition:Employee Management System using Class Fundamentals and Constructors** Design a simple Employee Management System to model employees in a company. Each employee should have attributes like ID, name, department, and salary. Use appropriate access modifiers to ensure encapsulation. Implement default and parameterized constructors, and demonstrate constructor overloading. Track the number of employees created using a static variable and showcase the use of the this keyword and this() constructor chaining. Use instanceof to check object types during runtime when dealing with contract vs. permanent employees. **Problem Definition:Student Portal with Inner Classes, Object Class Methods, and Memory Concepts** Create a Student Portal System that stores student details and their enrolled courses. Implement a main Student class and define an inner class Course. Use method-local inner classes to represent semester-wise enrollments, and anonymous inner classes for scholarship eligibility logic. Override the equals(), toString(), and hashCode() methods from the Object class to compare student objects. Demonstrate how objects are created in the heap, how method calls go on the stack, and when objects become eligible for garbage collection. **Problem Definition: Vehicle Registration System using Annotations, Reflection, and Final Keyword** Build a Vehicle Registration System where different types of vehicles (Car, Bike, Truck) are registered. Use final variables for immutable fields, such as vehicle registration numbers. Declare a final class for utility methods and restrict overriding certain methods with final. Implement custom annotations like @VehicleInfo to mark important fields and use reflection to extract metadata at runtime (e.g., print fields annotated with @VehicleInfo). Demonstrate the use of Java meta-annotations in defining your custom annotation. **Key Questions / Analysis / Interpretation:** 1. How do static variables help in tracking class-level data? 2. What is the purpose of using instanceof in polymorphic structures? 3. What's the significance of overriding equals(), hashCode(), and toString()? 4. When and how is memory allocated in the stack vs. the heap? 5. How does reflection help inspect and use metadata at runtime? 6. What are the implications of using custom annotations in large systems?	1

Supplementary Problems:
Def: 1 - Add a subclass ContractEmployee and override a method for tax calculation.
Def: 2 - Add a feature to track attendance with a method-local inner class.
Def: 3 - Add validation rules using custom annotations.

Key Skills to be addressed:

1. Use of this, static variables, and overloading
2. Understanding Java memory management
3. Usage of the final keyword, annotations

Applications:

1. HR software for employee tracking
2. College academic portal systems

Learning Outcome:

1. Understand object creation, constructor overloading, static vs instance variables, and encapsulation using access modifiers.
2. Apply inner classes, object class methods, and analyze memory behavior during object lifecycle.
3. Design and apply annotations, understand reflection, and enforce class-level restrictions using the final keyword.

Dataset/Test Data: NA

Tools/Technology To Be Used:

Java SE, Any IDE (IntelliJ/Eclipse), Console

Total Hours:
Implementation – 04
Total Engagement – 06

Post Laboratory Work Description: Implement supplementary problems

Evaluation Strategy Including Viva:

Feedback on Problem Definition Implementation
(Satisfaction Level 0 to 4, where 0 is lowest,1 is poor,2 is average, 3 is good, 4 is excellent) (This can be asked for a group of practicals belonging to the same tool/concept/technology)

Advanced/Intermediate Extension:
Intermediate: Add a method to calculate tax and net salary based on different employee types.
Advanced: Add file I/O to save and load employee records.

STRING HANDLING

Problem Definition: You are developing a chat application where users may use unnecessary spaces, special characters, or slang. Create a program that sanitizes each input string in real-time by:

- Removing extra spaces
- Replacing inappropriate words with ***
- Formatting the message (capitalization rules)

Key Questions / Analysis / Interpretation:

- How to clean and transform strings effectively?
- How to detect and replace patterns dynamically?
- How to preserve original message tone while sanitizing?

Supplementary Problems:

- Allow emoji placeholders like :smile: → □
- Detect and count abbreviations like “brb”, “idk”

Key Skills to be addressed:

- String pattern matching
- Use of String.replaceAll(), trim(), regex
- Real-time input filtering

Applications:

- Messaging platforms
- Content moderation systems

Learning Outcome:

Students will learn how to manipulate strings using Java methods to clean and format user data in real time.

Dataset/Test Data: Sample chat lines with slang, spaces, symbols

Tools/Technology To Be Used: Java SE, Regular Expressions (Regex)

Total Hours:

Implementation – 2 hrs
Total Engagement – 3 hrs

Post Laboratory Work Description:

- Test against sample chat logs
- Reflect on string method performance and flexibility

Evaluation Strategy Including Viva:

- Regex explanation
- Real-time message simulation demonstration

Advanced/Intermediate Extension:

- Intermediate: Create a dashboard of top used slang words
- Advanced: Use NLP to autocorrect spelling errors in input

Problem Definition: Design a utility for a book reading app that highlights palindrome words/sentences. Read a paragraph and identify all palindromes—words or full sentences. Ignore punctuation and case.

Key Questions / Analysis / Interpretation:

- What defines a palindrome when ignoring punctuation?
- How can string reversal and normalization be used?
- How to handle nested palindromes?

Supplementary Problems:

- Check palindrome phrases (e.g., "Madam In Eden I'm Adam")
- Count frequency of unique palindromes

Key Skills to be addressed:

- String normalization
- Reversal and comparison
- Tokenization

Applications:

- Digital libraries
- Educational reading tools

Learning Outcome:

Students will implement algorithms to detect palindromes in various contexts using Java string functions.

Dataset/Test Data: Paragraphs from books or user input with punctuation

Tools/Technology To Be Used: Java SE, StringTokenizer or split, Collections

Total Hours:

Implementation – 3 hrs
Total Engagement – 4 hrs

Post Laboratory Work Description:

- Submit code with example text
- Highlight trade-offs in sentence vs word-level detection

Evaluation Strategy Including Viva:

- Logic walk-through
- Demo on different paragraphs

Advanced/Intermediate Extension:

- **Intermediate:** Add highlighting of palindromes in output
- **Advanced:** Detect mirrored palindromes (e.g., "abc|cba")

1

Problem Definition:

You are building a utility to scan long input strings to find all occurrences of a given pattern (substring or regex) with performance optimization. This is designed for competitive programming inputs with 10^6+ characters.

Key Questions / Analysis / Interpretation:

- How can we optimize pattern matching in large strings?
- What's the difference between brute-force and regex approaches?
- How do you handle overlapping matches?

Supplementary Problems:

- Count overlapping matches
- Replace pattern with "**" in the original string

Key Skills to be addressed:

- Substring search
- Performance benchmarking

- Regex vs manual search comparison

Applications:

- Log analysis
- Search systems

Learning Outcome:

Students will compare string matching techniques and implement scalable search systems using string methods.

Dataset/Test Data: Large strings (simulate logs or DNA sequences) with hidden patterns

Tools/Technology To Be Used: Java SE, Regex, String.indexOf(), Pattern, Matcher

Total Hours:

Implementation – 3 hrs
Total Engagement – 4 hrs

Post Laboratory Work Description:

- Test and record time for different approaches
- Document regex vs loop comparison

Evaluation Strategy Including Viva:

- Performance discussion
- Regex and Matcher logic explanation

Advanced/Intermediate Extension:

- Intermediate: Add UI to search pattern in a live editor
- Advanced: Implement KMP algorithm for custom pattern matching

INHERITANCE, INTERFACES & PACKAGES

INHERITANCE

Aim: Develop a Hospital Management System using Inheritance and Polymorphism"

A hospital employs various types of staff like Doctors, Nurses, and Administrative Staff, each inheriting from a common base class Staff. Each subclass overrides the work() method to define specific duties. The super keyword is used to invoke parent class constructors or methods. Demonstrate runtime polymorphism by calling overridden methods dynamically.

Design constraints include:

Mark certain classes as final (e.g., FinanceTeam) to restrict further inheritance.
Demonstrate upcasting and downcasting.
Use abstract classes for roles that cannot be instantiated directly (e.g., MedicalStaff).

INTERFACES

Aim: Design a Smart Home Automation System using Interfaces and Segregation Principle

A smart home system consists of devices like Light, Fan, Thermostat, and Camera. Each device supports different features such as turning on/off, scheduling, and motion sensing. Apply interfaces to segregate these capabilities:

Use separate interfaces such as Switchable, Schedulable, and SensorEnabled.
Implement default and static methods in interfaces.
Follow the Interface Segregation Principle to ensure classes implement only the relevant functionality.
Example: Camera implements SensorEnabled and Switchable but not Schedulable

Aim: Develop an Event Notification System using Functional and Marker Interfaces

An event management platform requires a notification system to alert users through multiple channels such as Email, SMS, and PushNotification.

Define a functional interface Notifier with a method send(String message), implemented by all notification types.
Use lambda expressions to dynamically send notifications.
Introduce a marker interface UrgentNotification to classify certain channels (like SMS) as high-priority.
Include private, default, and static methods in the interface to format messages and handle logging.

Example Use Cases:
Trigger multiple notifications using a list of Notifier objects.
Identify and log urgent notifications based on marker interface.

PACKAGES

Problem Definition: Organizing a Java Application Using Packages

Objective:
To understand and apply the concept of packages in Java by dividing a multi-functional application into logically separated modules for improved maintainability, code reuse, and access control.

Problem Statement:
Design a Java-based Student Management System by creating and organizing related classes into different user-defined packages:

student.details – to store and display student personal and academic information.

student.utility – to provide utility functions such as grade calculation or input validation.

student.services – to offer services like student registration, listing all students, or searching by ID.

Implement interaction among classes from different packages using import statements and proper access modifiers.

Key Questions / Analysis / Interpretation:

1. How does inheritance promote code reuse and dynamic behavior in Java?
2. What role do access modifiers and abstract classes play in system design?
3. How can interfaces be used to implement multiple behaviors flexibly?
4. What is the importance of functional and marker interfaces in real-world systems?
5. How do packages improve modularity and access control?
6. How do default, static, and private methods in interfaces improve interface design?
7. Why should functionalities be logically grouped in different packages?

Supplementary Problems:

Def: 1 - Hospital Billing System Extension
Add BillingStaff and PharmacyStaff with additional methods for handling patient bills and medicine inventory. Demonstrate inheritance with domain-specific attributes.

Def: 2 - Smart Home Device Control Panel
Build a control panel that takes a list of devices and uses interface polymorphism to trigger supported operations like turning on, scheduling, or sensing motion dynamically.

Def: 3 - Notification Retry Logic
Enhance the notification system to support retry logic for failed messages using static and private methods inside interfaces, and use lambda expressions for retry strategy selection.

Key Skills to be addressed:

1. Class and object design using inheritance
2. Runtime polymorphism and typecasting
3. Interface segregation and multiple interface implementation
4. Functional and marker interfaces with lambda expressions
5. Java package creation, import handling, and access control

Applications:

Real-time event alert and escalation systems

Modular enterprise-grade systems with reusable components

Learning Outcome:

Apply interface-driven development for system extensibility and modularity.
Design scalable applications by structuring code into logical packages with proper encapsulation.

Post Laboratory Work Description:

1. Modify existing classes to add additional features like audit logs or history tracking.
2. Extend the inheritance hierarchy or interface implementations to cover more edge cases.
3. Document the structure and provide class diagrams.
4. Write a brief reflective report highlighting challenges faced, concepts applied, and learnings.

Evaluation Strategy Including Viva

- **Code Implementation (40%)** – Clean structure, correct use of OOP/interface/package concepts.
- **Documentation & Comments (10%)** – Code clarity, use of JavaDocs/comments.
- **Post Lab Submission (10%)** – Completed report with analysis and screenshots.
- **Viva (40%)** – Questions based on:
    - Reason behind inheritance/interface choice
    - Difference between abstract class and interface
    - Explanation of marker/functional interfaces
    - Package structure and access modifiers

Advanced/Intermediate Extension

Intermediate:

- Demonstrate composition in addition to inheritance (e.g., a SmartHome has multiple Device objects).

Advanced:

- Use Java Reflection API to dynamically invoke interface methods or class constructors.

EXCEPTIONS HANDLING

Problem Definition: Robust Command-Line Calculator

Create a simple command-line calculator that takes two numbers and an operator (+, -, *, /) as input from the user. Your program must handle potential runtime errors gracefully. Specifically, it should manage cases where the user enters non-numeric input for numbers (NumberFormatException) and attempts to divide a number by zero (ArithmeticException). Use a try-catch block to handle these specific exceptions and a finally block to print a "Calculation finished" message, regardless of whether an error occurred or not.

Key Questions / Analysis / Interpretation:

What is the main purpose of using a try-catch block?
Explain the difference between a checked exception and an unchecked exception.
Which types did you handle in this problem?
Under what circumstances will the finally block execute? Can it be skipped?
What happens if an exception occurs that you haven't written a catch block for?

Supplementary Problems:

Def: 1 - Add a default case in your operator logic to throw a new IllegalArgumentException if the user enters an invalid operator (e.g., '%', '^').

Key Skills to be addressed:

try-catch-finally block implementation
Handling standard unchecked exceptions (NumberFormatException, ArithmeticException)
Reading user input from the console

Applications:

Validating user input in web forms or applications.
Preventing crashes in simple data entry tools.

Learning Outcome:

Students will be able to implement basic error handling to prevent application crashes and understand the flow of control in try-catch-finally statements.

Dataset/Test Data: NA

Tools/Technology To Be Used:

Java SE, Any IDE (IntelliJ/Eclipse), Console

Total Hours:
Implementation – 01
Total Engagement – 02

Post Laboratory Work Description:
Implement the supplementary problem to handle invalid operators.

Evaluation Strategy Including Viva:
Feedback on Problem Definition Implementation: (Satisfaction Level 0 to 4)
Viva Questions: Focus on the difference between Error and Exception, the exception hierarchy, and the purpose of the finally block.

Advanced/Intermediate Extension:
Intermediate: Modify the program to repeatedly ask for input in a loop until the user enters "exit".
Advanced: Log every exception caught (with its message and timestamp) to a text file named error_log.txt

Problem Definition: Bank Account Validator

Design a simple banking application that simulates withdrawals and deposits. Create a BankAccount class with a balance field. Implement two methods: deposit(amount) and withdraw(amount). This system must handle specific banking errors by using custom exceptions:

InvalidTransactionException: Create this checked exception and throw it if a user tries to deposit or withdraw a negative or zero amount.
InsufficientFundsException: Create this unchecked exception and throw it if a user tries to withdraw an amount greater than the current balance.

Your main class should create a BankAccount object, and perform a series of deposits and withdrawals inside a try-catch block to demonstrate how these custom exceptions are caught and handled.

Key Questions / Analysis / Interpretation:
Why is it beneficial to create custom exceptions like InsufficientFundsException instead of using a standard one like IllegalArgumentException?
What is the difference between the throw and throws keywords in Java? Provide an example from your code.
Explain the difference in handling a checked vs. an unchecked custom exception. Why might you choose one over the other?
How does creating a custom exception improve the readability and maintainability of your code?

Supplementary Problems:
Def: 1 - Implement a try-with-resources block to read initial account balance from a file named balance.txt, ensuring the file resource is closed automatically.

Key Skills to be addressed:

	Creating and using custom checked and unchecked exceptions. Using the throw keyword to trigger an exception manually. Using the throws keyword in method signatures. Understanding the difference between checked and unchecked exceptions. **Applications:** Business applications where domain-specific errors are common (e.g., e-commerce, financial services). APIs and libraries that need to report specific error conditions to the client. **Learning Outcome:** Students will be able to design and implement custom exceptions to handle specific error conditions in an application, making the code more robust and descriptive. **Dataset/Test Data:** A balance.txt file containing an initial numeric value (e.g., 5000.0). **Tools/Technology To Be Used:** Java SE, Any IDE (IntelliJ/Eclipse), Console, File I/O **Total Hours:** Implementation – 02 Total Engagement – 03 **Post Laboratory Work Description:** Implement the supplementary problem to read data using try-with-resources. **Evaluation Strategy Including Viva:** Feedback on Problem Definition Implementation: (Satisfaction Level 0 to 4) Viva Questions: Ask about the process of creating a custom exception, the super() call in the constructor, and the benefits of try-with-resources. **Advanced/Intermediate Extension:** Intermediate: Implement exception chaining. If an InvalidTransactionException occurs, catch it and re-throw it as a new exception that includes the original exception as its cause. Advanced: Modify the BankAccount class to be thread-safe. Use synchronized blocks for deposit/withdraw methods and demonstrate how to handle potential multi-threading issues.	
	Problem Definition: Inventory Management System Create an Inventory system that manages a collection of Product objects (each with an ID, name, and quantity). Implement a method purchaseProduct(productId, quantityToPurchase). This method must use custom exceptions to handle specific business rules: ProductNotFoundException (Checked): Thrown if no product with the given ID exists in the inventory. InsufficientStockException (Unchecked): Thrown if the quantityToPurchase is	

greater than the product's available quantity.

Your main class should demonstrate calling this method and handling only the checked ProductNotFoundException explicitly with a try-catch block, while letting the unchecked InsufficientStockException potentially halt the program if not handled.

Key Questions / Analysis / Interpretation:
Why is it beneficial to create specific exceptions like ProductNotFoundException instead of using a generic one like Exception?
Explain your design choice: why make ProductNotFoundException a checked exception and InsufficientStockException an unchecked one?
What is the difference between throw and the throws keyword in a method signature? Show an example from your code.
How do custom exceptions improve the API of your Inventory class?

Supplementary Problems:
Def: 1 - Add a restockProduct(productId, quantityToAdd) method that throws an IllegalArgumentException if the quantity is zero or negative.

Key Skills to be addressed:
Creating custom checked and unchecked exceptions.
Using the throw keyword for business rule validation.
Using the throws keyword in method signatures.
Understanding the design implications of checked vs. unchecked exceptions.

Applications:
E-commerce platforms for managing product stock.
Supply chain management software.
Booking systems (e.g., SeatUnavailableException).

Learning Outcome:
Students will be able to design and implement custom exceptions to create robust and self-documenting business logic.

Dataset/Test Data: NA

Tools/Technology To Be Used:
Java SE, Any IDE (IntelliJ/Eclipse), Console

Total Hours:
Implementation – 02
Total Engagement – 03

Post Laboratory Work Description:
Implement the supplementary restockProduct method with its validation.

Evaluation Strategy Including Viva:
Feedback on Problem Definition Implementation: (Satisfaction Level 0 to 4)
Viva Questions: Discuss the process of creating a custom exception, inheriting from Exception vs. RuntimeException, and the responsibility of the calling

method.

Advanced/Intermediate Extension:
Intermediate: Implement exception chaining. If an InsufficientStockException occurs, catch it and re-throw it as a TransactionFailedException that includes the original exception as its cause.
Advanced: Make the purchaseProduct method thread-safe and handle potential concurrency issues.

Problem Definition: Corrupt Signal Processor

You are tasked with processing a stream of incoming signals for a satellite. The signals are supposed to be positive integers, but some are corrupted. Your program receives a single line of space-separated strings as input. You must calculate the sum of all valid signals. A signal is valid if it can be parsed into an integer and falls within the inclusive range of [1, 1000]. Your program must handle NumberFormatException for malformed signals (e.g., "abc") and use a manually thrown IllegalArgumentException for out-of-range numbers. The final output should be the sum of valid signals and, on a new line, the total count of corrupt signals encountered.

Key Questions / Analysis / Interpretation:

Why is catching an Exception a poor strategy here compared to catching the specific NumberFormatException?

How does using exception handling in this scenario compare to using conditional checks (e.g., if-else with regex) for performance?

What is the main difference between an error that stops the program and an error that you can recover from, in the context of this problem?

Could this problem be solved without throwing a new IllegalArgumentException? What is the benefit of doing so?

Supplementary Problems:

Def: 1 - Modify the program to also track the count of each specific error type (NumberFormatException vs. IllegalArgumentException) and print them.

Key Skills to be addressed:

Efficient input parsing with error handling.

Using try-catch blocks within a loop.

Distinguishing between recoverable and non-recoverable states.

Manual exception throwing for custom validation logic.

Applications:

Data sanitization and cleaning in data science pipelines.

Robust parsers for competitive programming problem inputs.

Fault-tolerant stream processing.

Learning Outcome:

Understand how to use basic exception handling to build resilient parsers that can process large, imperfect datasets without crashing, a common requirement in competitive programming.

Dataset/Test Data:

Input: 10 50 abc 2000 75 -5 100
Output: 185
4

Tools/Technology To Be Used:

Java SE, Any IDE (IntelliJ/Eclipse), Console

Total Hours:

Implementation – 01

Total Engagement – 02

Post Laboratory Work Description:

Implement the supplementary problem to provide a detailed error breakdown.

Evaluation Strategy Including Viva:

Feedback on Problem Definition Implementation: (Satisfaction Level 0 to 4)

Viva Questions: Focus on the performance cost of exception handling and when it's appropriate to use it in a tight loop versus alternative error-checking methods.

Advanced/Intermediate Extension:

Intermediate: Read a very large number of signals (e.g., 10^6) from a file and compare the performance of the exception-based solution to one using Scanner.hasNextInt().

Advanced: Process the signals in parallel using a stream, aggregating valid results and exceptions separately.

MULTITHREADED PROGRAMMING

CO4

Problem Definition:

Create a Java program where two threads print numbers from 1 to 5 and A to E, respectively, with a 1-second delay. Demonstrate thread creation using both extending Thread class and implementing Runnable interface.

Key Questions / Analysis / Interpretation:

- How is a thread created using Thread vs Runnable?
- What is the purpose of start() and run()?
- Why is sleep() used?

Supplementary Problems:

- Create two threads printing even and odd numbers separately.
- Create a thread to print your name five times with delay.

Key Skills to be addressed:

Thread creation methods, Thread lifecycle basics, Delay using Thread.sleep()

Applications:

Introduction to concurrent task execution, Foundation for multi-tasking applications

Learning Outcome:

Students will be able to create and run basic threads, understand thread lifecycle, apply delays and observe interleaving output.

Dataset/Test Data:

N/A (Simple hardcoded values)

Tools/Technology To Be Used:

Java SE, Any IDE (IntelliJ/Eclipse), Console

Total Hours:

Implementation – 2 hrs

Total Engagement – 3 hrs

Post Laboratory Work Description:

Submit source code, Reflect on observed output order, Answer questions on

	thread execution **Evaluation Strategy Including Viva:** Viva on thread lifecycle, Code explanation and thread creation technique **Feedback on Problem Definition Implementation** (Satisfaction Level 0 to 4, where 0 is lowest,1 is poor,2 is average, 3 is good, 4 is excellent) **Advanced/Intermediate Extension:** Intermediate: Add a third thread that prints symbols. Advanced: Synchronize threads to print alternately in order.	
	Problem Definition: Simulate a bank account accessed by multiple threads representing deposit and withdrawal operations. Ensure the balance is updated correctly using synchronization. Key Questions / Analysis / Interpretation: ● What happens if multiple threads access a shared resource? ● How to use synchronized keyword? ● What are race conditions? Supplementary Problems: ● Add print statements showing race condition without sync. ● Try with and without synchronized keyword. **Key Skills to be addressed:** Synchronization, Shared resource access, Thread-safe operations **Applications:** Online banking systems, Transaction processing **Learning Outcome:** Students will be able to implement synchronized methods in Java and handle concurrent access to shared data. **Dataset/Test Data:** Initial balance: 1000 Thread 1: Withdraw 700 Thread 2: Deposit 500 Thread 3: Withdraw 800 Tools/Technology To Be Used: **Java SE, Console-based application** Total Hours: Implementation – 3 hrs Total Engagement – 4 hrs **Post Laboratory Work Description:** Analyze output for correctness, Answer: “Why is synchronization necessary	CO4

	here?" **Evaluation Strategy Including Viva:** Explain race condition and synchronized method, Demonstrate correct output **Feedback on Problem Definition Implementation** (Satisfaction Level 0 to 4, where 0 is lowest,1 is poor,2 is average, 3 is good, 4 is excellent) **Advanced/Intermediate Extension:** Intermediate: Add logging of transactions. Advanced: Add thread priority and simulate deadlock scenarios.	
	Problem Definition: Write a Java program to read 3 different files using 3 threads simultaneously. Each thread should count the number of lines in the file and print the result. **Key Questions / Analysis / Interpretation:** ● How to perform file I/O in threads? ● How to handle exceptions in threads? ● How do threads improve efficiency? **Supplementary Problems:** ● Count words or characters instead of lines. ● Read same file with multiple threads and merge result. **Key Skills to be addressed:** File handling, Multi-threaded file I/O, Exception handling in threads **Applications:** Log analysis, Data preprocessing pipelines **Learning Outcome:** Students will be able to implement multi-threaded file reading and handle exceptions and concurrent tasks. **Dataset/Test Data:** 3 small text files with random data (provided or created by student) **Tools/Technology To Be Used:** Java SE, Files and BufferedReader classes **Total Hours:** Implementation – 3 hrs Total Engagement – 4 hrs **Post Laboratory Work Description:** Submit code and test files, Describe what happens if file not found **Evaluation Strategy Including Viva:** Code explanation, Thread creation and I/O logic, Discussion on thread efficiency Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4, where 0 is lowest,1 is poor,2 is average, 3 is good, 4 is	CO4

	excellent) **Advanced/Intermediate Extension:** Intermediate: Merge the line counts from all files. Advanced: Add progress bar or status update using GUI.	
	Problem Definition: Design a real-time coding leaderboard simulation where multiple coder threads are continuously submitting their scores. Each thread updates a shared leaderboard (top-10 scores). Ensure thread safety and performance using concurrent collections like ConcurrentSkipListMap or PriorityBlockingQueue. **Key Questions / Analysis / Interpretation:** ● How can we avoid data inconsistency during simultaneous updates? ● What data structures are best suited for concurrent top-k sorting? ● How to prevent race conditions and deadlocks in live systems? **Supplementary Problems:** ● Add a feature for score updates and user removal. ● Visualize leaderboard refresh every few seconds using Timer. **Key Skills to be addressed:** Use of thread-safe data structures, Score simulation with concurrent updates, Live sorting of top-k elements **Applications:** Competitive coding platforms (e.g., Codeforces, LeetCode), Online gaming leaderboards **Learning Outcome:** Students will be able to handle real-time concurrent updates and choose proper concurrent data structures for performance. **Dataset/Test Data:** Simulated user names with randomized scores submitted at varying intervals. **Tools/Technology To Be Used:** Java SE, ConcurrentSkipListMap, ExecutorService, TimerTask **Total Hours:** Implementation – 3 hrs Total Engagement – 5 hrs **Post Laboratory Work Description:** Analyze leaderboard updates under concurrent load, Submit explanation of thread-safety mechanism used **Evaluation Strategy Including Viva:** Explain concurrent collection usage, Demonstrate top-k logic under heavy submissions	CO4

	Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4, where 0 is lowest,1 is poor,2 is average, 3 is good, 4 is excellent) **Advanced/Intermediate Extension:** Intermediate: Track submission time and sort accordingly. Advanced: Create separate leaderboards by coding domain (DSA, Math, etc.)	
	Problem Definition: Write a program where multiple threads compress different chunks of a file (or simulate string blocks). Measure time taken for compression and optimize using thread pools. The goal is to simulate a multithreaded backup tool under time constraints. **Key Questions / Analysis / Interpretation:** ● How to divide file data across threads? ● How to use thread pool for optimal resource usage? ● How does thread creation overhead impact performance? **Supplementary Problems:** ● Try with 2, 4, and 8 threads and compare speed. ● Implement compression simulation using RLE or simple repetition logic. **Key Skills to be addressed:** Task division for I/O operations, Thread pool tuning, Performance benchmarking **Applications:** Backup systems, Log file processors, Archive software **Learning Outcome:** Students will be able to create a parallel processing system for large data files and apply compression logic in a performance-sensitive scenario. **Dataset/Test Data:** Large text files or generated string arrays with repeating patterns **Tools/Technology To Be Used:** Java SE, ExecutorService, Timer utilities **Total Hours:** Implementation – 3 hrs Total Engagement – 5 hrs **Post Laboratory Work Description:** Submit compression performance metrics, Reflect on trade-offs between number of threads and speed **Evaluation Strategy Including Viva:** Discuss how compression was done, Compare performance with different thread counts Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4, where 0 is lowest,1 is poor,2 is average, 3 is good, 4 is	CO4

	excellent) **Advanced/Intermediate Extension:** Intermediate: Merge compressed chunks into a single file Advanced: Implement file splitting and compression with Callable and Future	
	FILE NIO	
	Problem Definition: Design a Java application to read student marks from a file and generate individual grade reports, including average marks and grades based on specified criteria. **Key Questions / Analysis / Interpretation:** ● How to efficiently read and parse data from files? ● How do you map scores to grades? ● How do you write structured data back into a report file? **Supplementary Problems:** ● Handling missing or malformed data. ● Calculating class average and top scorer report. **Key Skills to be addressed:** ● File Reading/Writing ● Data Structures (ArrayList/HashMap) ● String manipulation and Exception Handling **Applications:** Automated report card generation systems & educational analytics. **Learning Outcome:** Students will be able to read and write structured data using File I/O and apply basic data analysis logic. **Dataset/Test Data:** `students.txt` (Name, Roll, Subject1, Subject2, Subject3...) **Tools/Technology To Be Used:** Java, Eclipse/NetBeans, FileReader, BufferedReader, FileWriter **Total Hours:** Implementation – 3 Hours Total Engagement – 4 Hours	CO5

	Post Laboratory Work Description: ● Document test cases. ● Modify the system to export in CSV format. **Evaluation Strategy Including Viva:** ● Code Quality and Execution: 10 marks ● Viva (concepts & logic): 5 marks ● Output file accuracy: 5 marks **Feedback on Problem Definition Implementation:** (Satisfaction Level: ___ / 4) **Advanced/Intermediate Extension:** ● Intermediate: Include grade histogram generation **Advanced:** Add GUI-based file selector and visualisation chart for grades	
	Problem Definition: Create a Java-based file logging system that tracks book issues and returns logs in a library. **Key Questions / Analysis / Interpretation:** ● How do you append logs to an existing file? ● How do you search and filter records from a log file? **Supplementary Problems:** ● Filter logs between two dates ● Count books issued by a student. **Key Skills to be addressed:** ● Append mode in File I/O ● Search/filter from flat files ● Time-stamping with logs **Applications:** Library systems, activity log tracking, basic audit trails **Learning Outcome:** Students will learn how to maintain sequential logs using files and how to perform selective data retrieval. **Dataset/Test Data:** log.txt (StudentID, BookID, Issue/Return, DateTime) **Tools/Technology To Be Used:** Java, BufferedWriter/BufferedReader, SimpleDateFormat **Total Hours:** Implementation – 2 Hours	CO5

	Total Engagement – 3.5 Hours **Post Laboratory Work Description:** ● Create a method to delete a record by Book ID. **Evaluation Strategy Including Viva:** ● Functional logging system: 8 marks ● Record filtering logic: 7 marks ● Viva: 5 marks **Feedback on Problem Definition Implementation:** (Satisfaction Level: ___ / 4) **Advanced/Intermediate Extension:** ● Intermediate: Sort logs by student or book ● Advanced: Export logs in XML or JSON format	
	Problem Definition: Develop a console-based Java application that tracks monthly expenses and writes daily entries to a file with auto-generated monthly summaries. **Key Questions / Analysis / Interpretation:** ● How do you write entries daily and aggregate monthly? ● How to use File I/O for persistent expense tracking? **Supplementary Problems:** ● Display monthly average ● List the highest three expense categories **Key Skills to be addressed:** ● File I/O with Buffered Streams ● Data aggregation ● Monthly file creation using Java I/O **Applications:** Expense Tracking, Budget Management Systems. **Learning Outcome:** Students will learn to build real-world file-based systems with date-wise data organisation and summaries. **Dataset/Test Data:** User input stored in `expenses_YYYYMM.txt` **Tools/Technology To Be Used:** Java, Scanner, FileWriter, Calendar, Formatter **Total Hours:** Implementation – 3 Hours	CO5

	Total Engagement – 4.5 Hours **Post Laboratory Work Description:** ● Add functionality to generate pie charts based on categories. **Evaluation Strategy Including Viva:** ● Completeness and correctness: 10 marks ● Reusability and modularity: 5 marks ● Viva: 5 marks **Feedback on Problem Definition Implementation:** (Satisfaction Level: ___ / 4) **Advanced/Intermediate Extension:** ● Intermediate: Categorize and search by expense type ● Advanced: Import/export data from/to spreadsheet formats	
	Problem Definition: Given a text file, count the frequency of each word and display the top 10 most frequent words. **Key Questions / Analysis / Interpretation:** ● How do you parse large files efficiently? ● How do you use data structures for counting? **Supplementary Problems:** ● Case-insensitive counting ● Exclude common stop words **Key Skills to be addressed:** ● File reading ● HashMap usage ● Sorting by value **Applications:** Text analysis, search engine indexing, NLP **Learning Outcome:** Students gain confidence in problem-solving using File I/O and data structures. **Dataset/Test Data:** `input.txt` with large text content **Tools/Technology To Be Used:** Java, HashMap, Comparator, BufferedReader **Total Hours:** Implementation – 2 Hours Total Engagement – 3 Hours	CO5

	Post Laboratory Work Description: ● Output results to CSV format **Evaluation Strategy Including Viva:** ● Time complexity discussion: 5 marks ● Code efficiency: 5 marks ● Accuracy: 5 marks ● Viva: 5 marks **Feedback on Problem Definition Implementation:** (Satisfaction Level: ___ / 4) **Advanced/Intermediate Extension:** ● Intermediate: Add stop word filter **Advanced:** Support multi-file input and merge results	
	Problem Definition: Read a 9x9 Sudoku puzzle from a file and validate whether it is a correct solution or not. **Key Questions / Analysis / Interpretation:** ● How to represent and validate 2D data from a file? ● What constraints to check in Sudoku? **Supplementary Problems:** ● Validate partially filled board ● Check missing digits **Key Skills to be addressed:** ● Multidimensional arrays ● Constraint-based logic ● File parsing **Applications:** Puzzle validators, grid-based file parsing **Learning Outcome:** Builds logic and data validation skills using 2D data from files. **Dataset/Test Data:** `sudoku.txt` – 9x9 grid **Tools/Technology To Be Used:** Java, BufferedReader, 2D Arrays **Total Hours: Implementation –** 2 Hours **Total Engagement –** 3 Hours	CO5

Post Laboratory Work Description:

- Modify to support 4x4 or 16x16 variants

Evaluation Strategy Including Viva:

- Grid parsing and validation: 10 marks
- Correctness of constraints: 5 marks
- Viva: 5 marks

Feedback on Problem Definition Implementation:
(Satisfaction Level: ___ / 4)

Advanced/Intermediate Extension:

- **Intermediate:** Visual representation of invalid rows/columns
- **Advanced:** GUI-based board validator and puzzle solver

COLLECTION FRAMEWORK AND GENERICS

CO6

Problem Definition:
Create a **Java program** that allows users to **manage a To-Do List.** The program will offer a **menu-driven interface** to perform the following operations:

- **Add** a new task.
- **Display** all tasks.
- **Edit** an existing task.
- **Delete** a task.

Tasks will be stored using an **ArrayList**. Each functionality will be implemented as a **separate method.**

Key Questions / Analysis / Interpretation:

- Why is ArrayList suitable for this application?
- How are tasks added, removed, and updated in an ArrayList?
- What is the impact of removing an element on the indexes of other elements?

Supplementary Problems:

- Enhance the To-Do list to include task priorities (High, Medium, Low) and allow sorting based on priority.
- Implement a simple reminder feature to print all overdue tasks based on a deadline (Date handling)

Key Skills to be addressed:

- ArrayList operations, **Menu-driven programming logic**
- User input handling, Iterating through collections

Applications:
Task Management System, Shopping list , Inventory list

Learning Outcome:
Learn **ArrayList manipulation** methods and handle **user input and exceptions** effectively.

Dataset/Test Data:
User provides task descriptions as **hardcoded text input** via **Scanner** at runtime.

Tools/Technology To Be Used:
Java SE, Eclipse/IntelJ

Total Hours: Implementation – 2 Hours
Total Engagement – 3 Hours

Post Laboratory Work Description:

- Basic operation of add/delete/update/replace on ArrayList.

Evaluation Strategy Including Viva:

- Demonstration of **Add/Edit/Delete/Display** functionality.
- Correctness of constraints: 5 marks
- Viva: 5 marks

Feedback on Problem Definition Implementation:
(Satisfaction Level: ___ / 4)

Advanced/Intermediate Extension:

- **Intermediate:** Add **task priority** (High/Medium/Low) and allow sorting by priority using a **Comparator**.
- **Advanced:** Extend the program to **store tasks in a file** (File I/O) so that tasks are persisted between program runs.

Problem Definition:
Develop a word counter that counts word frequency in a paragraph using HashMap, maintains order using LinkedHashMap, and sorts by key using TreeMap.

Key Questions / Analysis / Interpretation:

- Why Map keys must be unique?
- Difference between HashMap and Hashtable?

Supplementary Problems:

- Implement a phonebook directory using Map.
- Frequency analysis in a text file.

Key Skills to be addressed:

- Key-value data structure (Map) usage

- Hashing and collision resolution understanding
- Ordering via TreeMap and LinkedHashMap
- Handling uniqueness in keys

Applications:
Word frequency analysis tools, Search engine indexing, Data categorization (e.g., grouping anagrams)

Learning Outcome:
Understand **Map** types and their characteristics (HashMap, LinkedHashMap, TreeMap, apply hashing concepts for data storage and retrieval.

Dataset/Test Data:
Simple hardcoded text/paragraph, User input or file data for supplementary problems

Tools/Technology To Be Used:
Java SE, Eclipse/IntelJ

Total Hours: Implementation – 2 Hours
Total Engagement – 3 Hours

Post Laboratory Work Description:

- Submit word counter program, Reflect on differences between Map types.

Evaluation Strategy Including Viva:

- Discuss how of insertion order, sorting, thread safety in Map types
- Correctness of constraints: 5 marks
- Viva: 5 marks

Feedback on Problem Definition Implementation:
(Satisfaction Level: ___ / 4)

Advanced/Intermediate Extension:

- **Intermediate:** Extend the word counter to display the top 5 most frequent words in the paragraph.
- **Advanced:** Instead of sorting words alphabetically (TreeMap default), sort the words **by their frequency values** in descending order.

Problem Definition:
Develop a Java program to manage a collection of unique Student IDs in a university. The system must 1) Store unique IDs using HashSet to prevent duplicates. 2) Display IDs in sorted order using TreeSet. 3) Maintain the order of insertion using LinkedHashSet.

Implement menu-driven options: 1) Add a new Student ID.2)Remove an existing ID.3)Check if a particular ID exists.4) Display all IDs in: 4.1) Unordered (HashSet) form 4.2)Sorted (TreeSet) form 4.3) Insertion order (LinkedHashSet) form.

Key Questions / Analysis / Interpretation:

- Why is HashSet used for storing Student IDs?

- What is the difference between HashSet, LinkedHashSet, and TreeSet?

Supplementary Problems:

- Design a system to merge two sets of Student IDs and display the result without duplicates.
- Implement a search feature to check if a specific ID exists and report accordingly.

Key Skills to be addressed:

- Understanding and usage of **HashSet, LinkedHashSet, TreeSet**.
- Performing **basic Set operations**: addition, removal, checking presence.
- Sorting and ordering collections using appropriate Set implementations.

Applications:
Maintaining unique identifiers

Learning Outcome:
Learn differences between HashSet, LinkedHashSet, TreeSet, Understand Set operations in Java

Dataset/Test Data:
Hardcoded integer IDs or user-entered values via Scanner.

Tools/Technology To Be Used:
Java SE, Eclipse/IntelJ

Total Hours: Implementation – 2 Hours
Total Engagement – 3 Hours

Post Laboratory Work Description:

- Submit simple code for ascending and descending of the 10 student name using HashSet and used comparator for custom sorting.

Evaluation Strategy Including Viva:

- Discuss how of insertion order, sorting, thread safety in Map types
- Correctness of constraints: 5 marks
- Viva: 5 marks

Feedback on Problem Definition Implementation:
(Satisfaction Level: ___ / 4)

Advanced/Intermediate Extension:

- **Intermediate:** Count Total Unique Student IDs Entered
- **Advanced:** Allow filtering of Student IDs based on custom conditions (e.g., IDs starting with ‘CS’ only) using Java Stream API with Sets.

Problem Definition:
Create an inventory management system which have three different inventory items (named as InventoryItem<T>) such as electronics, clothing and groceries. This class should support adding, updating, and deleting items, with bounded type parameters ensuring that only items implementing a Sellable interface are included. A method to calculate the total inventory value using wildcards can illustrate the flexibility of generics. To understand type erasure, you have to create an array of a generic type and note the compile-time errors, learning about the runtime limitations of generics.

Key Questions / Analysis / Interpretation:

- Why is a bounded type parameter (T extends Sellable) used in the generic class?
- What is Type Erasure in Generics? Why can't we create an array of type T?

Supplementary Problems:

- Extend InventoryItem class to apply discount calculation on all items using generics.
- Implement a sorting feature to sort items by price or name using generic comparator methods.

Key Skills to be addressed:

- Usage of **Wildcards (? extends Type)** for flexible method arguments**.**
- Understanding of **Type Erasure** and its limitations.

Applications:
Generic data handling in frameworks, libraries, Type-safe generic utility classes, and collections.

Learning Outcome:
Learn to implement and use Java Generics with bounded type parameters and design reusable and type-safe generic classes.

Dataset/Test Data:
Hardcoded inventory items via Scanner.

Tools/Technology To Be Used:
Java SE, Eclipse/IntelJ

Total Hours: Implementation – 2 Hours
Total Engagement – 3 Hours

Post Laboratory Work Description:

- Submit simple code for accessing wild card methods and constructor creation using generics.

Evaluation Strategy Including Viva:

- Discuss regarding of the insertion, deletion & update using generics also handle the wild card methods.
- Correctness of constraints: 5 marks

	• Viva: 5 marks **Feedback on Problem Definition Implementation:** (Satisfaction Level: ___ / 4) **Advanced/Intermediate Extension:** • **Intermediate:** Inventory search feature based on item name using generics. • **Advanced:** Create an inventory report feature generating item-wise stock and total inventory value, formatted and saved as a text file.	