Advance Java Programming

Lab Assignments

Uint-1

Object-Oriented Programming (OOP)

✅ Lab Question 1: Encapsulation using Getter and Setter

Objective: Enforce data security through access control.

Task:
Create a class `Student` with private data members:

- `id`, `name`, `marks`

Implement:

- Public getters and setters
- Validation in setter (marks must be between 0 and 100)

Create a `Main` class to:

- Accept and display student details using methods only

Deliverables:

- Proper encapsulation
- No direct access to variables

✅ Lab Question 2: Inheritance and Method Overriding

Objective: Demonstrate IS-A relationship and runtime behavior.

Task:
Create a base class `Employee` with method:

- `calculateSalary()`

Create derived class `Manager` that:

- Overrides `calculateSalary()` to add bonus

In `main()`:

- Use parent reference to call child object method

Deliverables:

- Use of `extends`
- Method overriding
- Runtime polymorphism

✅ Lab Question 3: Constructor Overloading and this Keyword

Objective: Validate object initialization strategies.

Task:
Create class `Book` with variables:

- `id`, `title`, `price`

Implement:

- Default constructor
- Parameterized constructor
- Use `this` to resolve variable conflict

Display book details.

Deliverables:

- Multiple constructors
- Proper use of `this`

✅ Lab Question 4: Abstract Class and Method Implementation

Objective: Enforce contract-based design.

Task:
Create abstract class `Shape` with:

- Abstract method `area()`

Create subclasses:

- `Rectangle`
- `Circle`

Implement `area()` in both classes.
In `main()`, store objects in `Shape` reference and display area.

Deliverables:

- Abstract class usage
- Method implementation
- Polymorphic behavior

✅ Lab Question 5: Interface Implementation and Multiple Inheritance

Objective: Demonstrate capability-based design.

Task:
Create interfaces:

- `Printable` → method `print()`
- `Scannable` → method `scan()`

Create class `MultiFunctionPrinter` that implements both.
In `main()`, invoke both methods.

Deliverables:

- Multiple interface implementation
- Clean method definitions

✔ Practical Evaluation Focus Areas

- Proper class design and access control
- Correct use of `extends` and `implements`
- Constructor logic and object creation
- Runtime polymorphism
- Code readability and structure

Exception Handling

✅ Lab Question 1: Basic Try–Catch Mechanism

Objective: Validate handling of runtime exceptions.

Task:
Write a Java program that accepts two integers from the user and performs division. Handle the following:

- Division by zero using `ArithmeticException`
- Invalid input using `InputMismatchException`

Deliverables:

- Proper `try-catch` blocks
- User-friendly error messages
- Program must not terminate abnormally

✅ Lab Question 2: Multiple Catch Blocks and Finally

Objective: Demonstrate structured exception hierarchy handling.

Task:
Create a program that accesses an array element using a user-provided index. Handle:

- `ArrayIndexOutOfBoundsException`
- `NullPointerException` (by intentionally setting array reference to null in one test case)
  Also implement a `finally` block that always prints: **"Execution completed."**

Deliverables:

- Multiple `catch` blocks
- One `finally` block
- Clean termination in all scenarios

✅ Lab Question 3: Custom Exception for Business Rule Validation

Objective: Implement domain-level exception control.

Task:
Create a custom exception named `InvalidAgeException`.
Write a program that checks eligibility for voting:

- If age < 18, throw `InvalidAgeException`
- Else, print "Eligible to vote"

Deliverables:

- Custom exception class
- Use of `throw` keyword
- Proper exception handling in `main()`

✅ Lab Question 4: Exception Propagation using throws

Objective: Validate exception delegation across layers.

Task:
Create three methods: `method1()`, `method2()`, and `method3()`

- `method3()` should cause `ArithmeticException`
- `method2()` should declare `throws ArithmeticException` and call `method3()`
- `method1()` should call `method2()` inside `try-catch` and handle the exception

Deliverables:

- Use of `throws` keyword
- Centralized exception handling
- Clear console trace of method calls

✅ Lab Question 5: Bank Account Simulation with Checked Exception

Objective: Apply exception handling in transaction control.

Task:
Create a class `BankAccount` with:

- `balance` variable
- `withdraw(amount)` method

Create a checked exception `InsufficientBalanceException`.
If withdrawal amount > balance, throw the exception.
Handle it in `main()` and display proper message.

Deliverables:

- Checked custom exception (`extends Exception`)
- Business-rule enforcement
- Clean transaction flow

✔ Evaluation Criteria (for Lab Assessment)

- Correct identification of exception types
- Proper use of `try`, `catch`, `finally`, `throw`, and `throws`

- Code stability (no abnormal termination)
- Compliance with problem constraints
- Readable and maintainable structure

File Handling

✅ Lab Question 1: Creating and Writing to a File

Objective: Demonstrate basic file creation and writing.

Task:

- Write a Java program to create a text file named `data.txt`.
- Take user input (any text) and write it to the file.
- Close the file properly.

Deliverables:

- Use `FileWriter` or `BufferedWriter`
- Handle `IOException`
- Program should inform the user if the file is successfully created

---

✅ Lab Question 2: Reading from a File

Objective: Demonstrate file reading using streams.

Task:

- Write a program to read the contents of `data.txt` (from previous lab).
- Display the file content on the console line by line.

Deliverables:

- Use `FileReader` or `BufferedReader`
- Handle exceptions properly
- Output should match the input text exactly

✅ Lab Question 3: Counting Words, Lines, and Characters in a File

Objective: Apply file processing logic.

Task:

- Write a program to read a file `data.txt`.
- Count and display:
  - Total number of words
  - Total number of lines
  - Total number of characters

Deliverables:

- Use proper loops and string manipulation
- Handle file not found scenarios (`FileNotFoundException`)
- Display statistics clearly

✅ Lab Question 4: File Copy Program

Objective: Demonstrate file handling with streams.

Task:

- Write a program to copy content from `data.txt` to a new file `copy.txt`.
- Read from the source file and write to the destination file.

Deliverables:

- Use `FileReader`/`FileWriter` or `BufferedReader`/`BufferedWriter`
- Include exception handling
- Confirm successful copy with a console message

✅ Lab Question 5: Random Access File Example

Objective: Demonstrate use of `RandomAccessFile`.

Task:

- Create a program that stores student records (id, name, marks) in a file using `RandomAccessFile`.
- Perform the following operations:
  - Add a new record
  - Read all records
  - Update marks of a specific student using file pointer

Deliverables:

- Use `RandomAccessFile` with `"rw"` mode
- Implement seek and update operations
- Handle exceptions (`IOException`)

✅ Lab Question 6: Serializing a Student Object

Objective: Demonstrate object serialization using `ObjectOutputStream`.

Task:

1. Create a class `Student` implementing `Serializable` with:
   - `int id`
   - `String name`
   - `float marks`
2. Write a program to:
   - Take input from the user for `id`, `name`, and `marks`.
   - Create a `Student` object.
   - Serialize the object and save it to a file named `student.dat`.
3. Ensure proper exception handling.

Deliverables:

- `Student` class with `Serializable` interface
- Object saved to file successfully
- Handle `IOException`

✅ Lab Question 7: Deserializing a Student Object

Objective: Demonstrate object deserialization using `ObjectInputStream`.

Task:

1. Using the file `student.dat` created in Lab 1, write a program to:
    - Read the `Student` object from the file.
    - Display the details (`id`, `name`, `marks`) on the console.
2. Ensure proper exception handling.

Deliverables:

- Read object from file successfully
- Display object data correctly
- Handle `IOException` and `ClassNotFoundException`

✔ Evaluation Focus

- Proper usage of file classes (`File`, `FileReader`, `FileWriter`, `RandomAccessFile`)
- Exception handling (`IOException`, `FileNotFoundException`)
- Correct reading/writing operations
- Clear console output and user interaction

Multi-Threading

✅ Lab Question 1: Creating a Thread by Extending Thread Class

Objective: Demonstrate basic thread creation using `Thread` class.

Task:

1. Create a class `MyThread` that extends `Thread`.
2. Override the `run()` method to print numbers from 1 to 10 with a delay of 500ms between prints.
3. Create an object of `MyThread` in `main()` and start the thread.

Deliverables:

- Use `start()` to run the thread
- Proper use of `Thread.sleep()`
- Output shows numbers printed by the thread

✅ Lab Question 2: Creating a Thread by Implementing Runnable Interface

Objective: Demonstrate thread creation using `Runnable`.

Task:

1. Create a class `MyRunnable` that implements `Runnable`.
2. Override `run()` to print a message 5 times.
3. Use `Thread` object to start the thread.
4. Create multiple threads of `MyRunnable` and run them concurrently.

Deliverables:

- Use `Runnable` interface
- Demonstrate concurrent execution

- Proper exception handling (`InterruptedException`)

✅ Lab Question 3: Thread Synchronization

Objective: Demonstrate shared resource handling using `synchronized`.

Task:

1. Create a class `BankAccount` with a `balance` variable and method `deposit(amount)` and `withdraw(amount)`.
2. Create multiple threads to withdraw money simultaneously.
3. Use `synchronized` to prevent race conditions.

Deliverables:

- Shared resource access is thread-safe
- Use of `synchronized` keyword
- Proper console output showing correct balance updates

✅ Lab Question 4: Thread Priority and Lifecycle

Objective: Demonstrate thread priorities and lifecycle states.

Task:

1. Create three threads: `ThreadA`, `ThreadB`, `ThreadC`.
2. Assign different priorities: `MAX_PRIORITY`, `NORM_PRIORITY`, `MIN_PRIORITY`.
3. Print thread name, priority, and current state in the console.
4. Observe the effect of priorities on thread execution order.

Deliverables:

- Use `setPriority()` and `getPriority()`
- Use `getState()` to print lifecycle state
- Output shows thread information clearly

✅ Lab Question 5: Inter-Thread Communication Using wait() and notify()

Objective: Demonstrate thread coordination.

Task:

1. Create a `Producer` class and a `Consumer` class sharing a common `Buffer` object.
2. Use `wait()` and `notify()` to synchronize producer and consumer.
3. Producer should add items to the buffer; consumer should remove items.

Deliverables:

- Correct use of `wait()` and `notify()`
- No deadlock occurs
- Proper console output showing producer and consumer actions

---

✅ Lab Question 5: Thread Synchronization with Shared Resource

Objective: Demonstrate thread-safe access to a shared resource using `synchronized`.

Task:

1. Create a class `Counter` with:
    - Private integer variable `count` initialized to 0
    - Method `increment()` that increases `count` by 1
    - Method `getCount()` to return the current value
2. Create two threads that call the `increment()` method **1000 times each** on the same `Counter` object.
3. Use `synchronized` keyword to ensure that increments are thread-safe.
4. After both threads finish, display the final value of `count`.

Deliverables:

- Proper use of `synchronized` to prevent race conditions
- Console output showing the correct final value (`2000`)
- Comments explaining why synchronization is necessary

✅ Lab Question 6: Demonstrating Deadlock in Java

Objective: Understand and observe deadlock caused by improper resource locking between threads.

Task:

1. Create a class `Resource` with a `String name` and a method `use()` that prints a message indicating the thread using the resource.
2. Create two `Resource` objects: `resource1` and `resource2`.
3. Create two threads (`ThreadA` and `ThreadB`) that attempt to acquire locks on both resources in **opposite order**:
    - `ThreadA` locks `resource1` first, then `resource2`.
    - `ThreadB` locks `resource2` first, then `resource1`.
4. Use `synchronized` blocks on the resources to simulate resource locking.
5. Print messages to the console showing when a thread:
    - Acquires a lock
    - Waits for a lock
    - Completes its execution
6. Observe that both threads get blocked indefinitely due to deadlock.

Deliverables:

- Proper use of `synchronized` to lock resources
- Console output demonstrating deadlock occurrence
- Comments explaining why the deadlock happens

✔ Evaluation Focus

- Proper creation of threads (`Thread` vs `Runnable`)
- Understanding thread lifecycle and priorities
- Synchronization for shared resources
- Inter-thread communication using `wait()`/`notify()`
- Exception handling (`InterruptedException`)
- Clean and readable console output

Arrays & Jagged Arrays

✅ Lab Question 1: Basic Array Operations

Objective: Demonstrate basic array creation and traversal.

Task:

1. Create an integer array of size `5`.
2. Take input from the user to populate the array.
3. Display the array elements, their sum, and average.

Deliverables:

- Use `for` loop for input and traversal
- Correct calculation of sum and average
- Proper console output

✅ Lab Question 2: 2D Array Operations

Objective: Demonstrate handling of 2D arrays.

Task:

1. Create a 2D integer array of size `3x3`.
2. Take input from the user to fill the matrix.
3. Display the matrix in row-column format.
4. Calculate and display the sum of each row and each column.

Deliverables:

- Correct input and output of 2D array
- Accurate sum calculations

- Well-formatted console display

✅ Lab Question 3: Jagged Array (Ragged Array) Implementation

Objective: Demonstrate creation and use of jagged arrays.

Task:

1. Create a jagged array with 3 rows:
    - Row 1 → 2 elements
    - Row 2 → 3 elements
    - Row 3 → 4 elements
2. Take input from the user to populate the jagged array.
3. Display the jagged array in proper format.
4. Calculate and display the sum of elements in each row.

Deliverables:

- Proper creation of jagged array using `new int[row][]`
- Correct traversal using nested loops
- Accurate sum for each row
- Clear console output showing row-wise elements

✔ Evaluation Focus

- Correct creation of arrays and jagged arrays
- Accurate input and output handling
- Proper use of nested loops for 2D and jagged arrays
- Summation and average calculations for array elements
- Clean and readable console output