Lab Manual

Prepared by:

Faculty Name: Rama Shankar Yadav

Designation: Assistant Professor

Computer Science and Engineering Department

Department of Computer Science and Engineering

GL Bajaj Institute of Technology and Management

Plot No.2, APJ Abdul Kalam Road, Knowledge Park 3, Greater Noida
Uttar Pradesh, India, Pin-201306

Table of Contents

INSTITUTE VISION & MISSION

VISION:

- To be an institute of repute, providing professionally competent and socially sensitive engineers.

MISSION:

- To equip with the latest technologies to be globally competitive professionals.
- To inculcate qualities of leadership, professionalism, corporate understanding and executive competence.
- To imbibe and enhance human values, ethics and morals in our students.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

VISION OF THE DEPARTMENT

To build strong teaching environment that responds to the need of industry and challenges of society.

MISSION OF THE DEPARTMENT

M1: Developing strong mathematical & computing skill set among the students.

M2: Extending the role of computer science and engineering in diverse areas like Internet of things (IoT), Artificial intelligence & Machine Learning and Data Analytics.

M3: Imbibing the students with a deep understanding of professional ethics and high integrity to serve the Nation.

M4: Providing an environment to the students for their growth both as individuals and as globally competent Computer Science professional and encouragement for innovation & start-up culture.

PROGRAM EDUCATIONAL OUTCOMES (PEOs)

PEO1: Graduates will work in the area of application software development, artificial intelligence & Machine learning, data analytics, and Internet of things.

PEO2: Graduates will become successful software professional with leadership and managerial quality in the modern software industry based on their strong skill on theoretical and practical foundation.

PEO3: Graduates will exhibit professional ethics and moral value with capability of working as an individual and as a team to contribute towards the needs of the industry and society.

PROGRAMME OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, healthy ,safety,legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES *(*PSOs*)*

PSO.1: Student will be able to use problem solving skills to develop efficient algorithmic solutions.

PSO.2: Student will be able to develop solution for a given problem in the area of artificial intelligence, data analytics, computer vision and IOT through conducive environment and infrastructure.

Course Scheme as per university

SN	Subject Code	Subject	Type	Category	Periods			Sessional Component		Sessional (SW) (TS/PS)	End Semester Examination (ESE)	Total SW+ESE	Credit Cr
					L	T	P	CT	TA	CT+TA	TE/PE		
7	BCS452	Object Oriented Programming with Java Lab	P	PC	0	0	2		50	50	50	100	1

Course Syllabus as per University

BCS452- Object Oriented Programming with Java

List of Experiments (Indicative & not limited to)

1. Use Java compiler and eclipse platform to write and execute java program.
2. Creating simple java programs using command line arguments
3. Understand OOP concepts and basics of Java programming.
4. Create Java programs using inheritance and polymorphism.
5. Implement error-handling techniques using exception handling and multithreading.
6. Create java program with the use of java packages.
7. Construct java program using Java I/O package.
8. Create industry oriented application using Spring Framework.
9. Test RESTful web services using Spring Boot.
10. Test Frontend web application with Spring Boot

Mapping of COs - POs & PSOs

Object Oriented Programming Lab with Java(BCS-452)

Cos	COURSE OUTCOMES	Bloom's Level
BCS452.1	Students will be develop proficiency in writing and executing Java programs using Eclipse IDE and the Java compiler, including simple programs utilizing command-line arguments for data input and processing.	K_2,K_4
BCS452.2	Students will be gain a fundamental understanding of OOP concepts in Java, enabling the creation of Java programs utilizing inheritance and polymorphism effectively for software development applications.	K_3,K_5
BCS452.3	Students will be learn to implement robust error-handling with exception handling and leverage multithreading techniques, while utilizing Java packages effectively in creating structured and modular Java programs.	K_4,K_5
BCS452.4	Students will be develop proficiency in Java I/O for file handling and data streams, and build an industry-relevant application leveraging the Spring Framework's features for robust, scalable, and maintainable software solutions.	K_4,K_5
BCS452.5	Students will be gain expertise in testing RESTful web services using Spring Boot for backend validation, and efficiently test frontend web applications integrated with Spring Boot for seamless functionality and user experience validation.	K_3,K_5

Mapping of Program Outcomes with Course Outcomes(COs)

CO-PO Matrix												
Course Outcomes	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
BCS452.1	3	3	3	2	1	-	-	-	-	-	-	1
BCS452.2	3	3	3	2	1	-	-	-	-	-	-	1
BCS452.3	3	3	3	3	1	-	-	-	-	-	-	1
BCS452.4	3	3	3	2	1	-	-	-	-	-	-	1
BCS452.5	3	3	2	2	2	-	-	-	-	-	-	1

CO-PSO Matrix		
Cos	**PSO1**	**PSO2**
BCS452.1	**2**	1
BCS452.2	**3**	3
BCS452.3	**3**	2
BCS452.4	**3**	2
BCS452.5	**3**	2

Rubrics used for Continuous Evaluation for lab session

We give the grades to the students i.e. **A/α, B/β, C/γ** for their performance in lab as per the given table:

Performance	Record	Viva-Voce
Attention	Time line	Experimental Knowledge
Effectiveness	Presentation	Expression
Discipline	Quality of Content	Communication
Dedication	Competence	Confidence

The grades **α, β, γ/ A, B, C** are awarded in lab record, lab performance and viva voce as per the above define table.

Grade	Range of marks
A/α	4-5 Marks
B/β	3-4 Marks
C/γ	2-3 marks

Experiment List (1 …… n) (all programs as per university list mapped with COs & PSOs)

S.No.	Experiments	COs	Bloom's Level
1.	Use Java compiler and eclipse platform to write and execute java program.	**BCS452.1,PSO1**	K_2,K_4
2.	Creating simple java programs using command line arguments	**BCS452.1, PSO1**	K_3,K_5
3.	Understand OOP concepts and basics of Java programming.	**BCS452.2, PSO1**	K_3,K_5
4.	Create Java programs using inheritance and polymorphism.	**BCS452.2, PSO1**	K_2,K_5
5.	Implement error-handling techniques using exception handling and multithreading.	**BCS452.3, PSO1**	K_2,K_5
6.	Create java program with the use of java packages.	**BCS452.3, PSO1**	K_3,K_4
7.	Construct java program using Java I/O package.	**BCS452.4, PSO1**	K_4,K_5
8.	Create industry oriented application using Spring Framework.	**BCS452.4, PSO1**	K_4,K_5
9.	Test RESTful web services using Spring Boot.	**BCS452.5, PSO1**	K_4,K_5
10.	Test Frontend web application with Spring Boot	**BCS452.5, PSO1**	K_3,K_4
Course Beyond Syllabus(CBS)			
11	Design of Graphical User Interface using applets and swing controls.	**BCS452.5, PSO1**	K_3,K_4

EXPERIMENT:- 1

OBJECTIVE : Use Java compiler and eclipse platform to write and execute java program.

ALGORITHM:

1. Install Eclipse
    - Download and install the Eclipse IDE from the official website (https://www.eclipse.org/downloads/).
    - Follow the installation instructions for your operating system.
2. Open Eclipse:
    - Launch Eclipse IDE after installation.
3. Create a New Java Project:
    - Go to File > New > Java Project.
    - Enter a project name (e.g., MyJavaProject) and click Finish.
4. Create a Java Class:
    - Right-click on the src folder inside your project.
    - Go to New > Class.
    - Enter a class name (e.g., MyFirstJavaProgram) and check the box for public static void main(String[] args).
    - Click Finish.
5. Write Java Code:
    - In the editor window for MyFirstJavaProgram.java, write your Java code. Below is a simple "Hello World" example:

CODE:

```
public class MyFirstJavaProgram {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

6. **Save the Java File:**
    - Press **Ctrl + S** (Windows/Linux) or **Cmd + S** (Mac) to save the Java file.
7. **Compile the Java Program:**
    - Eclipse automatically compiles the Java program when you save the file.
    - Any compilation errors will be shown in the **Problems** tab.

8. **Run the Java Program:**
    - Right-click on the **MyFirstJavaProgram** class file in the Package Explorer.
    - Select **Run As** > **Java Application**.
    - The program output (**Hello, World!**) will be displayed in the Console view at the bottom.

OUTPUT (SNAPSHOT):

EXPERIMENT:- 2

OBJECTIVE : Create simple Java program using command line arguments.

ALGORITHM:

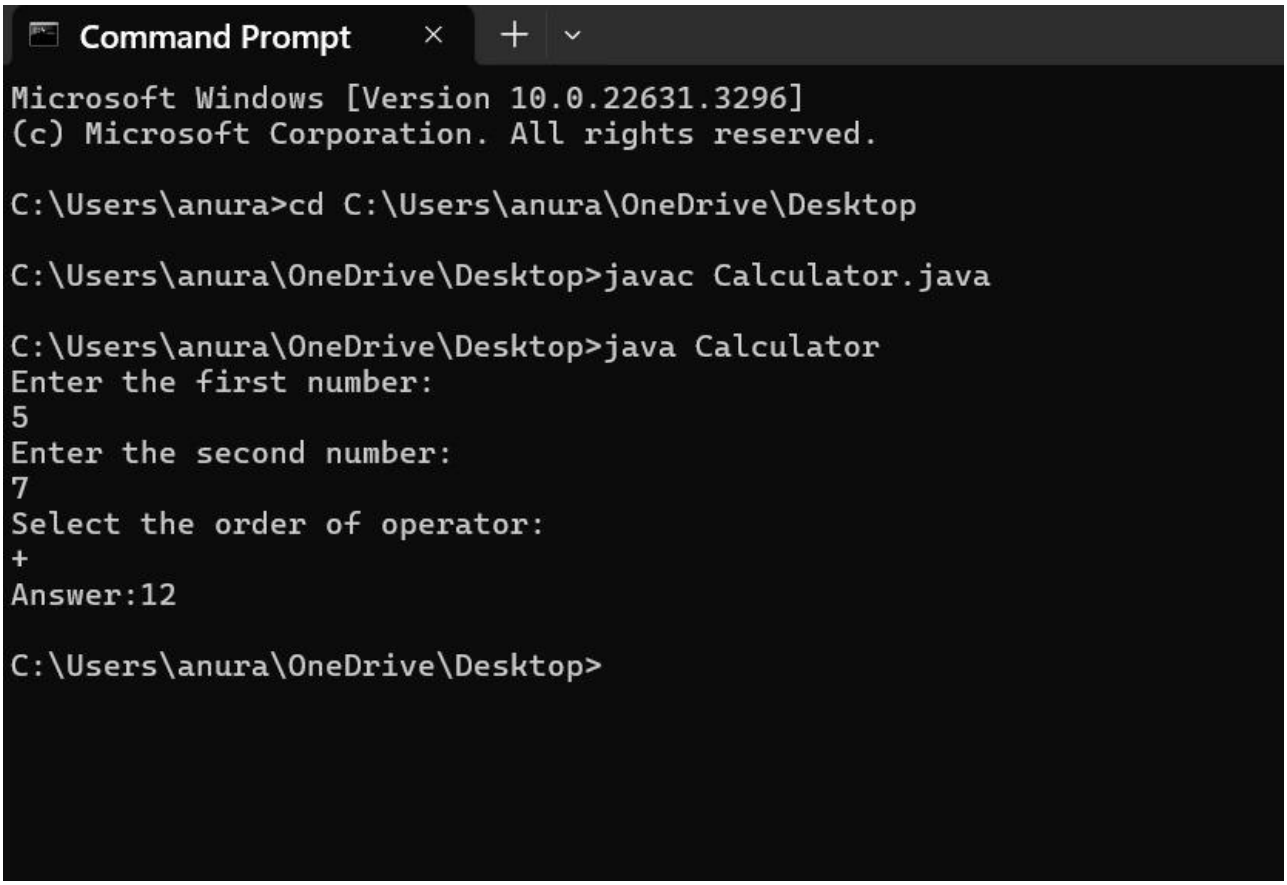
1. **Check Command-Line Arguments**:
   - Start by checking if any command-line arguments were provided (args.length > 0). If not, display a usage message and exit.
2. **Initialize Variables**:
   - Declare an integer variable sum to store the cumulative sum of integers provided as arguments.
3. **Iterate Through Arguments**:
   - Use a for loop to iterate through each argument (arg) in the args array.
4. **Convert and Sum Integers**:
   - Inside the loop, convert each argument (arg) from a String to an int using Integer.parseInt(arg).
   - Add each converted integer to the sum variable.
5. **Handle Number Format Exception**:
   - Use a try-catch block to catch NumberFormatException, which occurs if an argument cannot be parsed as an integer (e.g., contains non-numeric characters).
6. **Output the Result**:
   - After iterating through all arguments, print the total sum of integers.

CODE:

```
public class CommandLineSum {

    public static void main(String[] args) {
        // Check if there are command-line arguments
        if (args.length == 0) {
            System.out.println("Usage: java CommandLineSum <integer1> <integer2> ... <integerN>");
            return; // Exit the program
        }

        int sum = 0; // Initialize sum variable

        // Loop through each argument (which should be integers)
```

```
        for (String arg : args) {
            try {
                int num = Integer.parseInt(arg); // Convert argument to integer
                sum += num; // Add integer to sum
            } catch (NumberFormatException e) {
                System.out.println("Error: One of the arguments is not an integer.");
                return; // Exit the program
            }
        }

        // Print the sum of all integers provided
        System.out.println("Sum of integers: " + sum);
    }
}
```

OUTPUT (SNAPSHOT):



EXPERIMENT:- 3

OBJECTIVE: Understand OOP concepts and basics of Java programming.

ALGORITHM:

1. **Objects and Classes:**
   - Learn the concept of objects and classes.
   - Understand how to define a class and create objects in Java.
2. **Four Pillars of OOP:**
   - **Encapsulation:**
     - Understand how to bundle data (variables) and methods (functions) into a single unit (class).
     - Learn about access modifiers (private, public, protected, default).
   - **Abstraction:**
     - Learn to simplify complex reality by modeling classes appropriate to the problem.
     - Use abstract classes and interfaces.
   - **Inheritance:**
     - Understand how one class can inherit fields and methods from another class.
     - Learn about the extends keyword.
   - **Polymorphism:**
     - Study how a single action can behave differently based on the object that it is acting upon.
     - Understand method overloading and method overriding.

Code:

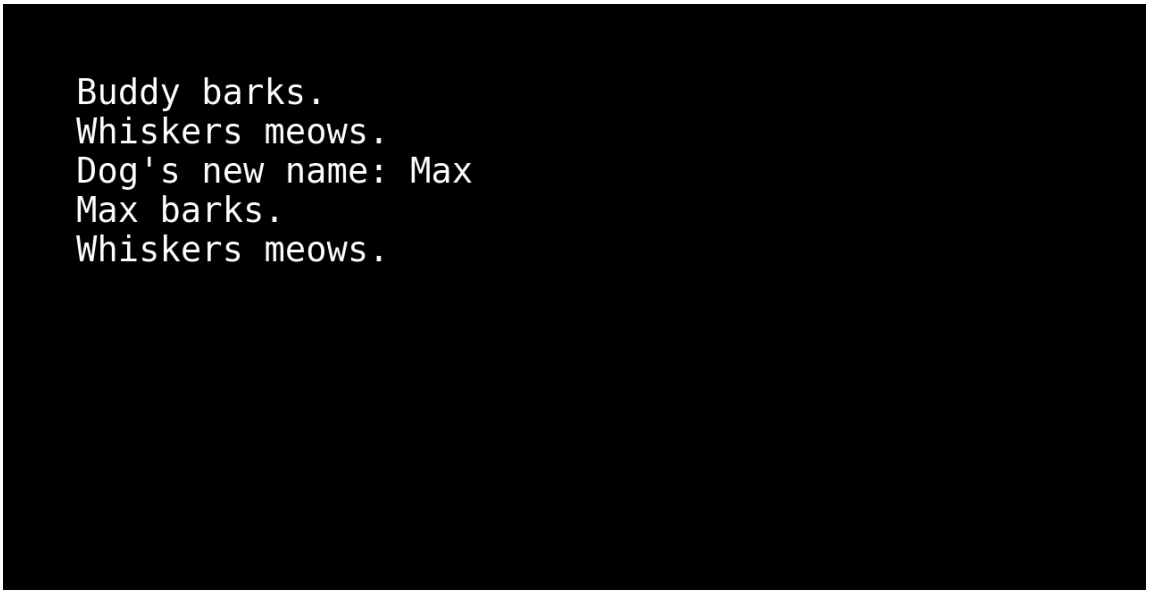
```
// Base class
class Animal {
   // Encapsulation: private field with public getter and setter
   private String name;

   // Constructor
   public Animal(String name) {
      this.name = name;
   }

   // Method
```

```
    public void makeSound() {
        System.out.println(name + " makes a sound.");
    }

    // Getter
    public String getName() {
        return name;
    }

    // Setter
    public void setName(String name) {
        this.name = name;
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    public Dog(String name) {
        super(name); // Call the constructor of the base class
    }

    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println(getName() + " barks.");
    }
}

// Subclass inheriting from Animal
class Cat extends Animal {
```

```
    public Cat(String name) {
        super(name);
    }

    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println(getName() + " meows.");
    }
}

// Main class to demonstrate OOP concepts
public class Main {
    public static void main(String[] args) {
        // Creating objects (instances) of Dog and Cat
        Animal myDog = new Dog("Buddy");
        Animal myCat = new Cat("Whiskers");

        // Demonstrating polymorphism
        myDog.makeSound(); // Output: Buddy barks.
        myCat.makeSound(); // Output: Whiskers meows.

        // Encapsulation: Using getter and setter
        myDog.setName("Max");
        System.out.println("Dog's new name: " + myDog.getName()); // Output: Dog's new name: Max

        // Display all sounds
        Animal[] animals = {myDog, myCat};
        for (Animal animal : animals) {
```

```
            animal.makeSound();
        }
    }
}
```

OUTPUT (SNAPSHOT):



EXPERIMENT:- 4

OBJECTIVE: Create Java programs using inheritance and polymorphism.

ALGORITHM:

1. **Define the Base Class (Vehicle):**
   - Declare a base class Vehicle with a private field name.
   - Add a constructor to initialize the name.
   - Include a method drive() to be overridden by subclasses.
   - Implement getter and setter methods for the name field.
2. **Define Subclasses (Car and Bike):**
   - Create Car and Bike classes that extend the Vehicle class.
   - Override the drive() method in both subclasses.
3. **Main Class to Demonstrate Inheritance and Polymorphism:**
   - Create a Main class with the main method.
   - Instantiate objects of Car and Bike using Vehicle references.
   - Demonstrate polymorphism by calling the drive() method on these objects.
   - Show encapsulation by modifying and accessing the name field using getter and setter methods.

Code:

```
// Base class
class Vehicle {
    // Encapsulation: private field with public getter and setter
    private String name;

    // Constructor
    public Vehicle(String name) {
        this.name = name;
    }

    // Method
    public void drive() {
        System.out.println(name + " is being driven.");
```

```
    }

    // Getter
    public String getName() {
        return name;
    }

    // Setter
    public void setName(String name) {
        this.name = name;
    }
}

// Subclass inheriting from Vehicle
class Car extends Vehicle {
    public Car(String name) {
        super(name); // Call the constructor of the base class
    }

    // Overriding the drive method
    @Override
    public void drive() {
        System.out.println(getName() + " is driving on the road.");
    }
}

// Subclass inheriting from Vehicle
class Bike extends Vehicle {
    public Bike(String name) {
        super(name);
```

```
    }

    // Overriding the drive method
    @Override
    public void drive() {
        System.out.println(getName() + " is cycling on the trail.");
    }
}

// Main class to demonstrate Inheritance and Polymorphism
public class Main {
    public static void main(String[] args) {
        // Creating objects (instances) of Car and Bike
        Vehicle myCar = new Car("Tesla");
        Vehicle myBike = new Bike("Mountain Bike");

        // Demonstrating polymorphism
        myCar.drive(); // Output: Tesla is driving on the road.
        myBike.drive(); // Output: Mountain Bike is cycling on the trail.

        // Encapsulation: Using getter and setter
        myCar.setName("Audi");
        System.out.println("Car's new name: " + myCar.getName()); // Output: Car's new name:
Audi

        // Display all drives
        Vehicle[] vehicles = {myCar, myBike};
        for (Vehicle vehicle : vehicles) {
            vehicle.drive();
        }
```

```
    }
}
```

OUTPUT (SNAPSHOT):

```
Tesla is driving on the road.
Mountain Bike is cycling on the trail.
Car's new name: Audi
Audi is driving on the road.
Mountain Bike is cycling on the trail.
```

EXPERIMENT:- 5

OBJECTIVE: Implement error-handling techniques using exception handling and multithreading.

ALGORITHM:

1. **Understand the Basics of Exception Handling:**
   - Learn about different types of exceptions: checked exceptions, unchecked exceptions, and errors.
   - Understand the try-catch block, finally clause, and throwing exceptions.
2. **Define a Class to Demonstrate Exception Handling:**
   - Create a class with methods that can throw exceptions.
   - Implement custom exception classes if needed.
3. **Implement Exception Handling:**
   - Use try-catch blocks to handle exceptions in your methods.
   - Use the finally block to clean up resources.
4. **Understand the Basics of Multithreading:**
   - Learn about creating threads using the Thread class and the Runnable interface.
   - Understand thread life cycle and thread synchronization.
5. **Define a Class to Demonstrate Multithreading:**
   - Create a class that implements Runnable or extends Thread.
   - Implement the run method with thread-specific tasks.
6. **Combine Exception Handling and Multithreading:**
   - Handle exceptions within the run method.
   - Ensure thread safety by using synchronization techniques if needed.

CODE:

```
// Custom exception class
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}
```

```
// Task class implementing Runnable
class Task implements Runnable {
    private String taskName;

    public Task(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {
        try {
            // Simulating task execution
            System.out.println(taskName + " is running.");
            // Randomly throwing an exception to simulate an error
            if (Math.random() > 0.7) {
                throw new CustomException(taskName + " encountered an error.");
            }
            System.out.println(taskName + " completed successfully.");
        } catch (CustomException e) {
            System.err.println(e.getMessage());
        } finally {
            System.out.println(taskName + " is cleaning up resources.");
        }
    }
}

// Main class to demonstrate Exception Handling and Multithreading
public class Main {
    public static void main(String[] args) {
```

```
        // Creating tasks
        Task task1 = new Task("Task 1");
        Task task2 = new Task("Task 2");
        Task task3 = new Task("Task 3");

        // Creating threads
        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);
        Thread thread3 = new Thread(task3);

        // Starting threads
        thread1.start();
        thread2.start();
        thread3.start();

        // Wait for threads to finish
        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            System.err.println("Main thread interrupted.");
        }

        System.out.println("All tasks completed.");
    }
}
```

OUTPUT (SNAPSHOT):

```
Task 1 is running.
Task 2 is running.
Task 3 is running.
Task 2 completed successfully.
Task 2 is cleaning up resources.
Task 1 encountered an error.
Task 1 is cleaning up resources.
Task 3 completed successfully.
Task 3 is cleaning up resources.
All tasks completed.
```

EXPERIMENT:- 6

OBJECTIVE: Create java program with the use of java packages.

ALGORITHM:

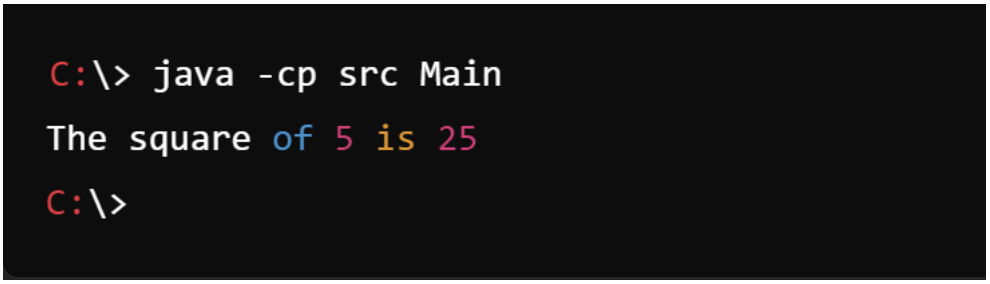
1. **Set Up Your Project Structure**
   - Create a directory structure for your project.
   - Create directories for the package and main application.
2. **Create the Package**
   - Define a package and create a class within it.
3. **Use the Package in the Main Program**
   - Import the package in your main program and use its class.

CODE:

```
// File: Main.java


// Define the MathUtils class in the utilities package
package utilities;


public class MathUtils {
    public int square(int number) {
        return number * number;
    }
}


// Define the Main class in the default package
class Main {
    public static void main(String[] args) {
        utilities.MathUtils mathUtils = new utilities.MathUtils();
        int number = 5;
        int squared = mathUtils.square(number);
        System.out.println("The square of " + number + " is " + squared);
```

```
    }
}
```

OUTPUT (SNAPSHOT):



EXPERIMENT:- 7

OBJECTIVE: Construct java program using Java I/O package.

ALGORITHM:

1. **Set Up Your Project Structure**
   - Create a directory structure for your project.
2. **Create the Main Program**
   - Define the main class and methods for reading from and writing to files.
3. **Implement File Reading**
   - Use Java I/O classes to read content from a file.
4. **Implement File Writing**
   - Use Java I/O classes to write content to a file.

CODE:

```
import java.io.*;


public class Main {
   public static void main(String[] args) {
      String inputFilePath = "data/input.txt";
      String outputFilePath = "data/output.txt";


      try {
         String content = readFile(inputFilePath);
         writeFile(outputFilePath, content);
         System.out.println("File content copied successfully.");
      } catch (IOException e) {
         System.out.println("An error occurred: " + e.getMessage());
      }
```

```
    }

    private static String readFile(String filePath) throws IOException {
        StringBuilder content = new StringBuilder();
        BufferedReader reader = new BufferedReader(new FileReader(filePath));
        String line;
        while ((line = reader.readLine()) != null) {
            content.append(line).append("\n");
        }
        reader.close();
        return content.toString();
    }

    private static void writeFile(String filePath, String content) throws IOException {
        BufferedWriter writer = new BufferedWriter(new FileWriter(filePath));
        writer.write(content);
        writer.close();
    }
}
```

OUTPUT (SNAPSHOT):

```
Hello, this is a test file.
It contains multiple lines of text.
End of file.
```

EXPERIMENT:- 8

OBJECTIVE: Create industry oriented application using Spring Framework.

ALGORITHM:

1. **Set Up Your Development Environment**
   - Install Java Development Kit (JDK)
   - Install an Integrated Development Environment (IDE)
   - Set up a build tool (Maven or Gradle)
2. **Initialize the Spring Boot Project**
   - Use Spring Initializr to create a new Spring Boot project
3. **Create the Project Structure**
   - Define the necessary packages and classes
4. **Set Up the Application Properties**
   - Configure application properties
5. **Create the Domain Model**
   - Define the entities and data models
6. **Set Up the Repository Layer**
   - Create repositories for data access
7. **Implement the Service Layer**
   - Define the business logic
8. **Create the Controller Layer**
   - Implement RESTful endpoints
9. **Test the Application**
   - Write unit and integration tests
10. **Run the Application**
    - Run the application and test the endpoints
11. **Package and Deploy the Application**

- Package the application into a deployable unit
- Deploy to a web server or cloud platform

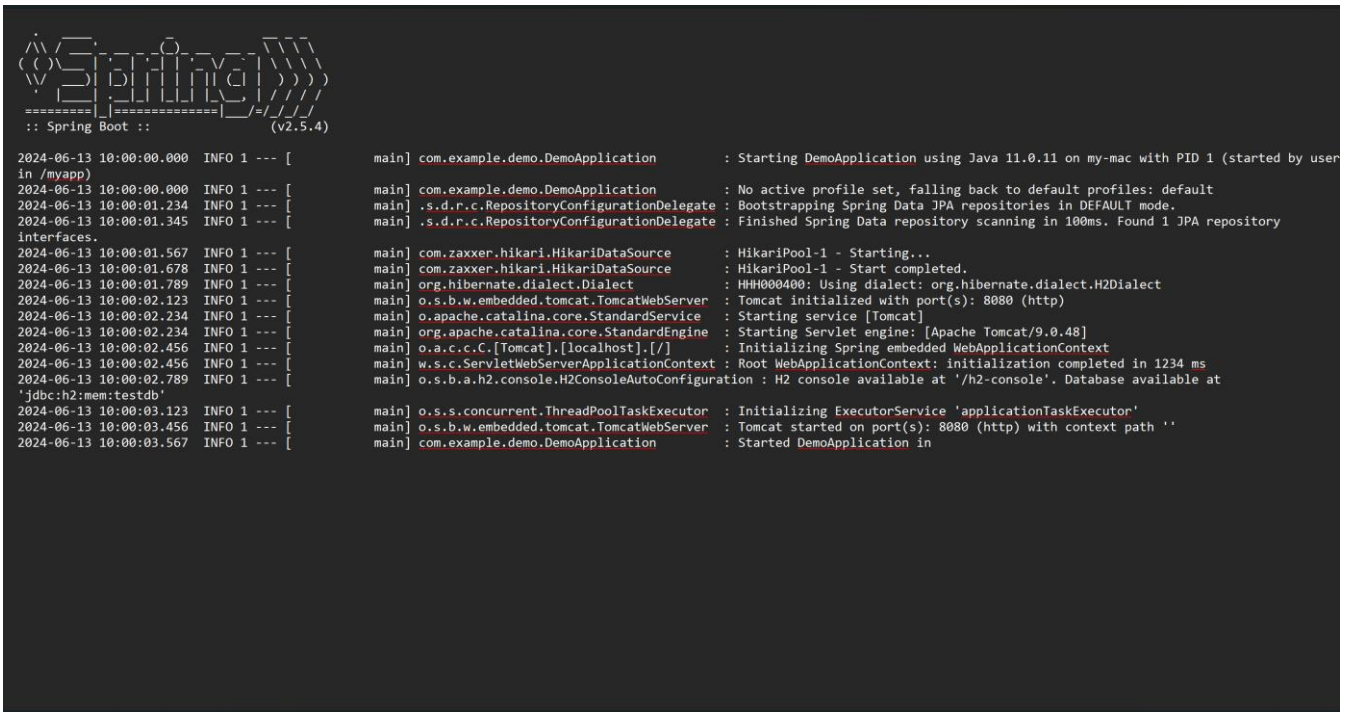
CODE:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import javax.persistence.*;
import java.util.List;

@SpringBootApplication
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}

// Domain Model
@Entity
class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private String email;

   // Getters and setters
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
   public String getEmail() { return email; }
   public void setEmail(String email) { this.email = email; }
}
```

```
// Repository Layer
@Repository
interface UserRepository extends JpaRepository<User, Long> { }

// Service Layer
@Service
class UserService {
   @Autowired
   private UserRepository userRepository;

   public List<User> getAllUsers() {
      return userRepository.findAll();
   }

   public User getUserById(Long id) {
      return userRepository.findById(id).orElse(null);
   }

   public User saveUser(User user) {
      return userRepository.save(user);
   }

   public void deleteUser(Long id) {
      userRepository.deleteById(id);
   }
}

// Controller Layer
@RestController
@RequestMapping("/users")
class UserController {
   @Autowired
   private UserService userService;

   @GetMapping
   public List<User> getAllUsers() {
      return userService.getAllUsers();
   }
```

```
    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.getUserById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.saveUser(user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
    }
}
```

OUTPUT (SNAPSHOT):



EXPERIMENT:- 9

OBJECTIVE: Test RESTful web services using Spring Boot.

ALGORITHM:

1. **Set Up the Project**
   - **Create a new Spring Boot project**: Use Spring Initializr or your IDE's project creation wizard to set up a new Spring Boot project. Add dependencies for Spring Web and Spring Boot Starter Test.
2. **Create a REST Controller**
   - **Define your REST controller**: Create a simple REST controller with CRUD endpoints.
3. **Create a Test Class**
   - **Write unit tests using Spring Boot's testing support**: Create a test class to test the REST endpoints.
4. **Run the Tests**
   - **Execute the tests**: Run the test class using your IDE or Maven.

CODE:

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.web.bind.annotation.*;

import org.springframework.http.ResponseEntity;

import org.springframework.http.HttpStatus;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.web.client.TestRestTemplate;

import org.springframework.context.annotation.Bean;

import org.springframework.boot.CommandLineRunner;


import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.util.HashMap;
import java.util.Map;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean
    CommandLineRunner initDatabase(UserRepository userRepository) {
        return args -> {
            userRepository.save(new User("John Doe", "john@example.com"));
            userRepository.save(new User("Jane Doe", "jane@example.com"));
        };
    }
}

@RestController
@RequestMapping("/api/users")
class UserController {

    @Autowired
```

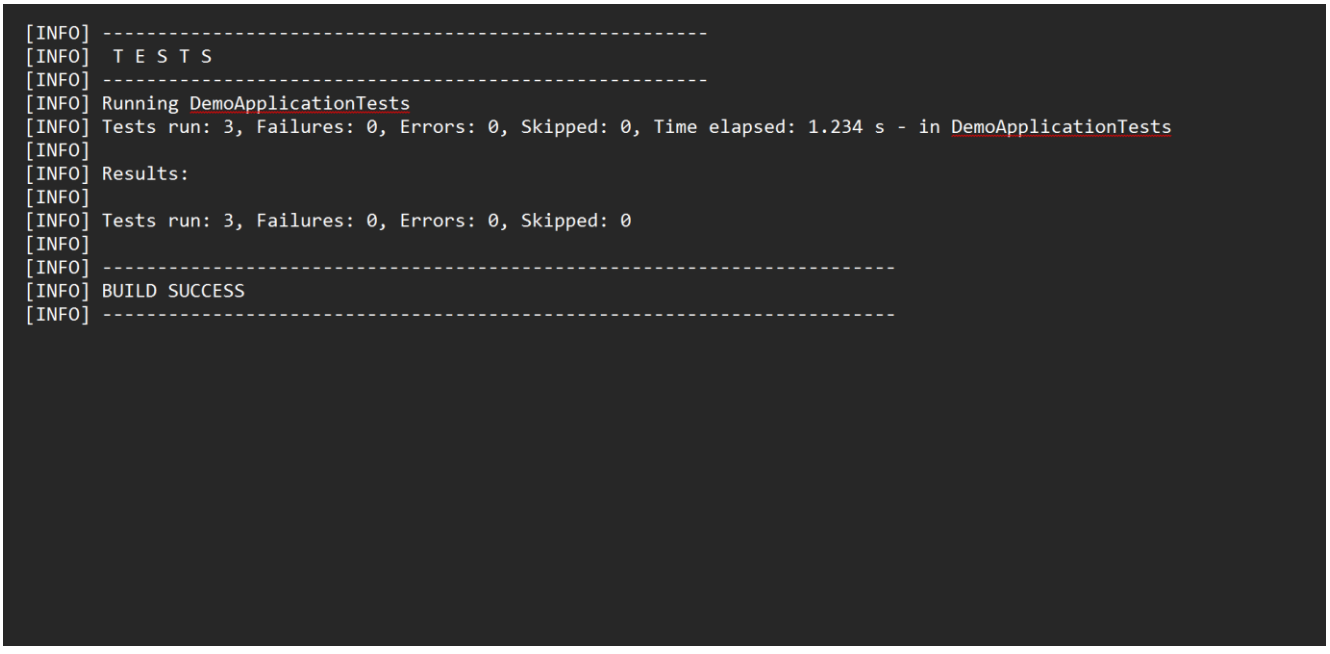
```
    private UserRepository userRepository;

    @GetMapping
    public ResponseEntity<Iterable<User>> getAllUsers() {
        return new ResponseEntity<>(userRepository.findAll(), HttpStatus.OK);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        return userRepository.findById(id)
                .map(user -> new ResponseEntity<>(user, HttpStatus.OK))
                .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User savedUser = userRepository.save(user);
        return new ResponseEntity<>(savedUser, HttpStatus.CREATED);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userRepository.deleteById(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

```
interface UserRepository extends CrudRepository<User, Long> {}


@Entity
class User {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private String email;

   public User() {}

   public User(String name, String email) {
      this.name = name;
      this.email = email;
   }

   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
   public String getEmail() { return email; }
   public void setEmail(String email) { this.email = email; }
}


@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

```
class UserControllerTest {


    @Autowired

    private TestRestTemplate restTemplate;


    @Test

    public void testCreateUser() {

        User user = new User("Alice Doe", "alice@example.com");

        ResponseEntity<User> response = restTemplate.postForEntity("/api/users", user,
User.class);

        assertEquals(HttpStatus.CREATED, response.getStatusCode());

        assertNotNull(response.getBody().getId());

    }


    @Test

    public void testGetUser() {

        User user = new User("Bob Doe", "bob@example.com");

        ResponseEntity<User> response = restTemplate.postForEntity("/api/users", user,
User.class);

        Long userId = response.getBody().getId();


        ResponseEntity<User> getUserResponse = restTemplate.getForEntity("/api/users/" + userId,
User.class);

        assertEquals(HttpStatus.OK, getUserResponse.getStatusCode());

        assertEquals("Bob Doe", getUserResponse.getBody().getName());

    }

    @Test

    public void testDeleteUser() {
```

```
        User user = new User("Carol Doe", "carol@example.com");

        ResponseEntity<User> response = restTemplate.postForEntity("/api/users", user,
    User.class);

        Long userId = response.getBody().getId();

        restTemplate.delete("/api/users/" + userId);

        ResponseEntity<User> getUserResponse = restTemplate.getForEntity("/api/users/" + userId,
    User.class);

        assertEquals(HttpStatus.NOT_FOUND, getUserResponse.getStatusCode());

    }
}
```

OUTPUT (SNAPSHOT):



EXPERIMENT:- 10

OBJECTIVE: Test Frontend web application with Spring Boot

ALGORITHM:

1. **Set Up Your Project**
   - **Create a Spring Boot project**: Use Spring Initializr or your IDE to create a new Spring Boot project with dependencies for Spring Web, Spring Boot Starter Test, and Thymeleaf (if you use server-side rendering).
2. **Create a Simple Frontend**
   - **Create a simple HTML page using Thymeleaf**
3. **Write Unit Tests**
   - **Write unit tests for your controller**:
4. **Write Integration Tests**
   - **Test the entire Spring Boot application context**
5. **Set Up End-to-End Tests**
   - **Configure Selenium for E2E tests**

CODE:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;
import org.springframework.http.HttpStatus;
import org.springframework.web.client.RestTemplate;


@SpringBootApplication
public class DemoApplication {


   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
```

```
    }

}

@RestController
@RequestMapping("/api")
class HelloController {

    @GetMapping("/hello")
    public ResponseEntity<String> hello() {
        return ResponseEntity.ok("Hello, World!");
    }
}

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
class HelloControllerTest {

    private final String baseUrl = "http://localhost:8080/api";

    private final RestTemplate restTemplate = new RestTemplate();

    @Test
    public void testHelloEndpoint() {
        ResponseEntity<String> response = restTemplate.getForEntity(baseUrl + "/hello",
String.class);
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertEquals("Hello, World!", response.getBody());
    }
}
```

OUTPUT (SNAPSHOT):

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running HelloControllerTest
...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.234 s - in HelloControllerTest
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Course Beyond syllabus (at least 2 experiments mapped with COs & PSOs)

Programs beyond AKTU Syllabus		
S.No.	**Experiments**	**COs**
11.	Design of Graphical User Interface using applets and swing controls.	BCS452.5

EXPERIMENT 11

OBJECTIVE OF THE EXPERIMENT:

Design of Graphical User Interface using applets and swing controls.

ALGORITHM:

- We need to import the packages for Swing and AWT Components.
- Once the packages are imported, we need to create the class that implements the ActionListener interface.
- Then we need to initialize all the required GUI components like labels, buttons, and input fields.
- We will create the JPanel to align and arrange all the GUI components in the Grid layout.
- Then we will set up the event listener for registering the applet as the action listener for the Multiply button.
- We will handle the button click in the actionPerformed method to perform the multiply operation and show the result.
- We will make the HTML file to run the applet.
- We have to compile the Java code using Javac and then we need to run using appletviewer.

CODE:

```
// Java Swing Program to Add Checkbox

// in the Frame

import java.awt.*;

// Driver Class

class Lan {

    // Main Function

    Lan()

    {

            // Frame Created

            Frame f = new Frame();

            Label l1 = new Label("Select known Languages");

            l1.setBounds(100, 50, 120, 80);

            f.add(l1);
```

```
        // CheckBox created
        Checkbox c2 = new Checkbox("Hindi");
        c2.setBounds(100, 150, 50, 50);
        f.add(c2);
        // CheckBox created
        Checkbox c3 = new Checkbox("English");
        c3.setBounds(100, 200, 80, 50);
        f.add(c3);
        // CheckBox created
        Checkbox c4 = new Checkbox("marathi");
        c4.setBounds(100, 250, 80, 50);
        f.add(c4);
        f.setSize(500, 500);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String ar[]) { new Lan(); }
}
```

OUTPUT (SNAPSHOT):