

WEEK 1: Review of Object-Oriented Programming Concepts

Objective: To refresh OOP concepts using Java

Theory:

Topics to be Covered

- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism
- Method Overloading and Overriding
- this and super keywords
- Constructors
- Abstraction

Note:

Object-Oriented Programming organizes software design around objects rather than functions. Java is a fully object-oriented language (except for primitive data types) and supports all major OOP principles.

- **Encapsulation:** Binding data and methods together using classes.
- **Inheritance:** Reusing properties of an existing class using extends.
- **Polymorphism:** Ability of a method to take multiple forms.
- **Abstraction:** Achieved using abstract classes and interfaces.

Sample exercises

1. Program demonstrating encapsulation using private data members and public getters/setters.

// Class demonstrating Encapsulation

```
class Student {
```

```
// Private data members
```

```
private int rollNumber;
```

```
private String name;
```

```
private double marks;
```

```
// Setter methods
```

```
public void setRollNumber(int rollNumber) {
```

```
    this.rollNumber = rollNumber;
```

```
}
```

```
public void setName(String name) {
```

```
    this.name = name;
```

```
}
```

```
public void setMarks(double marks) {
```

```
    if (marks >= 0 && marks <= 100) {
```

```
        this.marks = marks;
```

```
    } else {
```

```
        System.out.println("Invalid marks. Marks should be between 0 and 100.");
```

```
    }
```

```
}
```

```
// Getter methods
```

```
public int getRollNumber() {
```

```
    return rollNumber;
```

```
}
```

```
public String getName() {
```

```
    return name;
```

```
}
```

```
public double getMarks() {
```

```
    return marks;
```

```
}
```

```
}
```

```
// Main class
```

```
public class EncapsulationDemo {
```

```
    public static void main(String[] args) {
```

```
        // Creating object of Student class
```

```
        Student student = new Student();
```

```
        // Setting values using setter methods
```

```
        student.setRollNumber(1);
```

```
        student.setName("Ananya");
```

```
        student.setMarks(85.5);
```

```
        // Accessing values using getter methods
```

```
        System.out.println("Student Details");
```

```
        System.out.println("-----");
```

```
        System.out.println("Roll Number : " + student.getRollNumber());

        System.out.println("Name      : " + student.getName());

        System.out.println("Marks     : " + student.getMarks());

    }

}
```

Student Details

Roll Number : 1

Name : Ananya

Marks : 85.5

2. Program demonstrating method overloading.

// Class demonstrating Method Overloading

```
class Calculator {
```

```
    // Method to add two integers
```

```
    int add(int a, int b) {
```

```
        return a + b;
```

```
    }
```

```
    // Overloaded method to add three integers
```

```
    int add(int a, int b, int c) {
```

```
        return a + b + c;
```

```
    }
```

```
    // Overloaded method to add two double values
```

```
double add(double a, double b) {  
    return a + b;  
}  
}
```

// Main class

```
public class MethodOverloadingDemo {  
    public static void main(String[] args) {  
  
        Calculator calc = new Calculator();  
  
        // Calling overloaded methods  
        System.out.println("Addition of two integers : " + calc.add(10, 20));  
        System.out.println("Addition of three integers : " + calc.add(10, 20, 30));  
        System.out.println("Addition of two double values: " + calc.add(12.5, 7.5));  
    }  
}
```

3. Create a base class Employee and a derived class Manager. Override a method to calculate salary. Write a whole program in java

// Base class

```
class Employee {  
    protected int empId;  
    protected String name;  
    protected double basicSalary;
```

```
// Constructor
Employee(int empId, String name, double basicSalary) {
    this.empId = empId;
    this.name = name;
    this.basicSalary = basicSalary;
}

// Method to calculate salary
double calculateSalary() {
    // Basic salary calculation for an employee
    return basicSalary;
}

// Method to display employee details
void displayDetails() {
    System.out.println("Employee ID : " + empId);
    System.out.println("Employee Name : " + name);
}
}
```

```
// Derived class
class Manager extends Employee {
    private double hra;
    private double da;

    // Constructor
```

```
Manager(int empld, String name, double basicSalary, double hra, double da) {  
    super(empld, name, basicSalary);  
    this.hra = hra;  
    this.da = da;  
}  
  
// Overriding calculateSalary method  
@Override  
double calculateSalary() {  
    // Salary calculation for manager includes HRA and DA  
    return basicSalary + hra + da;  
}  
  
// Display manager details  
void displayManagerDetails() {  
    super.displayDetails();  
    System.out.println("Basic Salary : " + basicSalary);  
    System.out.println("HRA      : " + hra);  
    System.out.println("DA      : " + da);  
    System.out.println("Total Salary : " + calculateSalary());  
}  
}  
  
// Main class  
public class EmployeeDemo {  
    public static void main(String[] args) {
```

```
// Base class reference pointing to derived class object
Employee emp = new Manager(101, "Rahul Sharma", 40000, 8000, 6000);

System.out.println("---- Manager Salary Details ----");

// Runtime polymorphism
double totalSalary = emp.calculateSalary();

// Type casting to access Manager-specific method
if (emp instanceof Manager) {
    Manager mgr = (Manager) emp;
    mgr.displayManagerDetails();
}

System.out.println("-----");
System.out.println("Salary Calculated Using Polymorphism: " + totalSalary);
}
}

o/p
```

---- Manager Salary Details ----

Employee ID : 101

Employee Name : Rahul Sharma

Basic Salary : 40000.0

HRA : 8000.0

DA : 6000.0

Total Salary : 54000.0

Salary Calculated Using Polymorphism: 54000.0

Lab Exercises

1. Create a Book class with **private data members** including book ID, book title, author name, price, and availability status. Provide **public setter methods** to assign values to these data members and **public getter methods** to retrieve their values. Include validation in setter methods to ensure that the price is a positive value.
2. Create a base class named Room to represent general room details in a hotel. The class should contain data members such as room number, room type, and base price. Implement **multiple constructors (constructor overloading)** in the Room class to initialize room objects in different ways, such as:
 - i. Initializing only the room number and type
 - ii. Initializing room number, type, and base price
 - iii. Create a derived class named DeluxeRoom that **inherits** from the Room class using **single inheritance**. The derived class should include additional data members such as free Wi-Fi availability and complimentary breakfast. Implement appropriate constructors in the derived class that invoke the base class constructors using the super keyword.
 - iv. Create a main class to instantiate objects of both Room and DeluxeRoom using different constructors and display the room details. This application should clearly illustrate constructor overloading and inheritance.
3. Design and implement a Java application to simulate a **Hotel Room Booking System** that demonstrates the object-oriented concepts of **inheritance** and **runtime polymorphism**.
 - i. Create a base class named `Room` that represents a general hotel room. The class should contain data members such as room number and base tariff, and a method `calculateTariff()` to compute the room cost.
 - ii. Create derived classes such as `StandardRoom` and `LuxuryRoom` that **inherit** from the `Room` class. Each derived class should **override** the `calculateTariff()` method to compute the tariff based on room-specific features such as air conditioning, additional amenities, or premium services.
 - iii. In the main class, create a base class reference of type `Room` and assign it to objects of different derived classes (`StandardRoom`, `LuxuryRoom`). Invoke the `calculateTariff()` method using the base class reference to demonstrate **runtime polymorphism**, where the method call is resolved at runtime based on the actual object type.

4. Create an **abstract class** named `Room` that represents a generic hotel room. The abstract class should contain common data members such as room number and base price, and include an **abstract method** `calculateTariff()` that must be implemented by all subclasses. It may also include concrete methods such as `displayRoomDetails()`.
 - i. Create derived classes such as `StandardRoom` and `LuxuryRoom` that **extend** the abstract `Room` class and provide concrete implementations for the `calculateTariff()` method based on room-specific features.
 - ii. Create an **interface** named `Amenities` that declares methods such as `provideWifi()` and `provideBreakfast()`. The derived room classes should **implement** this interface to define the amenities offered for each room type.
 - iii. Create a main class to instantiate different room objects using a base class reference and invoke the implemented methods to demonstrate abstraction and interface-based design.

Week1 Course Outcome

Students will be able to design and implement Java programs using fundamental OOP principles.

WEEK 2: Java Library – Wrapper Classes, Enumeration, and Autoboxing

Objective: To understand Java wrapper classes, enumerations, autoboxing, and unboxing.

Topics to be covered

- Wrapper classes (Integer, Double, Character, etc.)
- Autoboxing and Unboxing
- Enumeration (enum)
- Enum methods and constructors

Sample exercises

1. Wrapper classes

```
public class WrapperClassDemo {  
    public static void main(String[] args) {  
  
        // Primitive data types  
        int a = 10;  
        double b = 25.5;  
        char c = 'A';  
        boolean flag = true;  
  
        // Converting primitives to wrapper objects (Boxing)
```

```

Integer intObj = Integer.valueOf(a);
Double doubleObj = Double.valueOf(b);
Character charObj = Character.valueOf(c);
Boolean boolObj = Boolean.valueOf(flag);

// Displaying wrapper objects
System.out.println("Wrapper Objects:");
System.out.println("Integer Object  : " + intObj);
System.out.println("Double Object   : " + doubleObj);
System.out.println("Character Object : " + charObj);
System.out.println("Boolean Object  : " + boolObj);

// Converting wrapper objects back to primitives (Unboxing)
int x = intObj.intValue();
double y = doubleObj.doubleValue();
char z = charObj.charValue();
boolean status = boolObj.booleanValue();

// Displaying primitive values after unboxing
System.out.println("\nPrimitive Values After Unboxing:");
System.out.println("int value   : " + x);
System.out.println("double value : " + y);
System.out.println("char value   : " + z);
System.out.println("boolean value : " + status);
}
}

```

o/p

Wrapper Objects:

Integer Object : 10

Double Object : 25.5

Character Object : A

Boolean Object : true

Primitive Values After Unboxing:

int value : 10

double value : 25.5

char value : A

boolean value : true

2. Autoboxing and unboxing

```
public class AutoBoxingUnboxingDemo {  
    public static void main(String[] args) {  
  
        // Autoboxing: primitive to wrapper object  
        int num = 50;  
        Integer intObj = num; // Autoboxing  
  
        System.out.println("Autoboxing Example:");  
        System.out.println("Primitive value : " + num);  
        System.out.println("Wrapper object : " + intObj);  
  
        // Unboxing: wrapper object to primitive  
        Integer obj = 100;  
        int value = obj;    // Unboxing  
  
        System.out.println("\nUnboxing Example:");  
        System.out.println("Wrapper object : " + obj);  
        System.out.println("Primitive value : " + value);  
    }  
}
```

o/p

Autoboxing Example:

Primitive value : 50

Wrapper object : 50

Unboxing Example:

Wrapper object : 100

Primitive value : 100

3. Enumeration(enum)

// Enum declaration

```
enum Day {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}
```

// Main class

```
public class EnumDemo {
```

```

public static void main(String[] args) {

    // Using enum
    Day today = Day.FRIDAY;

    System.out.println("Today is: " + today);

    // Using enum in switch statement
    switch (today) {
        case MONDAY:
            System.out.println("Start of the work week");
            break;
        case FRIDAY:
            System.out.println("Almost weekend!");
            break;
        case SATURDAY:
        case SUNDAY:
            System.out.println("Weekend");
            break;
        default:
            System.out.println("Midweek day");
    }

    // Iterating through enum values
    System.out.println("\nAll Days:");
    for (Day d : Day.values()) {
        System.out.println(d);
    }
}

```

o/p

Today is: FRIDAY
Almost weekend!

All Days:
MONDAY
TUESDAY
WEDNESDAY

THURSDAY

FRIDAY

SATURDAY

SUNDAY

4. Enum methods and Constructors

// Enum with constructor and methods

```
enum RoomType {
```

```
    // Enum constants with values
```

```
    STANDARD(2000),
```

```
    DELUXE(3500),
```

```
    SUITE(5000);
```

```
    // Instance variable
```

```
    private int pricePerNight;
```

```
    // Enum constructor
```

```
    RoomType(int pricePerNight) {
```

```
        this.pricePerNight = pricePerNight;
```

```
    }
```

```
    // Getter method
```

```
    public int getPricePerNight() {
```

```
        return pricePerNight;
```

```
    }
```

```
    // Method to calculate total cost
```

```
    public int calculateCost(int nights) {
```

```
        return pricePerNight * nights;
```

```
    }
```

```
}
```

```
// Main class
```

```
public class EnumConstructorMethodDemo {
```

```
    public static void main(String[] args) {
```

```
        RoomType room = RoomType.DELUXE;
```

```
        int nights = 3;
```

```

        System.out.println("Room Type    : " + room);
        System.out.println("Price per Night : " + room.getPricePerNight());
        System.out.println("Number of Nights: " + nights);
        System.out.println("Total Cost    : " + room.calculateCost(nights));
    }
}

```

o/p

```

Room Type    : DELUXE
Price per Night : 3500
Number of Nights: 3
Total Cost    : 10500

```

Lab Exercises

1. The Hotel Billing system should calculate the total bill amount for hotel guests based on room charges and additional service charges. Store numeric values such as room tariff, number of days stayed, and service charges using **wrapper class objects** (*Integer*, *Double*) instead of primitive data types.

Demonstrate **autoboxing** by automatically converting primitive values to wrapper class objects when assigning values or storing them in collections. Demonstrate **unboxing** by automatically converting wrapper class objects back to primitive types while performing arithmetic operations for bill calculation.

Create a main class to:

- i. Initialize room tariff and number of days using primitive data types and store them in wrapper objects.
 - ii. Perform total bill calculation using unboxed primitive values.
 - iii. Display the final hotel bill.
2. Design and implement a Java application to manage room tariff details in a **Hotel Management System** using **Java enumerations (enum)**. The application should demonstrate the use of **enum constants**, **enum constructors**, and **enum methods**.
 - i. Define an **enum** named `RoomType` to represent different types of hotel rooms such as `STANDARD`, `DELUXE`, and `SUITE`. Each enum constant should be associated with a base tariff value using an **enum constructor**. The enum should also include **methods** to return the base tariff and to calculate the total room cost based on the number of days stayed.
 - ii. Create a main class to select a room type, specify the number of days of stay, and compute the total room tariff by invoking the **enum methods**. The application should clearly illustrate how enum constructors are used to initialize constant-specific data and how enum methods operate on that data.

Week2 Course outcome

Students will be able to apply the wrapper class, auto boxing, unboxing,enum methods, constructors

WEEK 3: Multithreaded Programming – Basics

To understand the concept of multithreading and thread lifecycle in Java.

Topics Covered

- Thread creation using Thread class
- Thread creation using Runnable interface
- Thread lifecycle
- sleep(), join(), yield()

Multithreading allows concurrent execution of two or more threads for efficient CPU utilization.

Sample Exercises

1. Thread creation using Thread class

```
// Thread class by extending Thread
class RoomCleaningThread extends Thread {

    private String roomName;

    // Constructor
    RoomCleaningThread(String roomName) {
        this.roomName = roomName;
    }

    // Overriding run() method
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(roomName + " - Cleaning step " + i);
            try {
```



```

        Thread.sleep(500); // Pause thread execution
    } catch (InterruptedException e) {
        System.out.println("Thread interrupted");
    }
}
}
}
}

```

// Main class

```

public class ThreadUsingThreadClass {
    public static void main(String[] args) {

        // Creating thread objects
        RoomCleaningThread t1 = new RoomCleaningThread("Room 101");
        RoomCleaningThread t2 = new RoomCleaningThread("Room 102");

        // Starting threads
        t1.start();
        t2.start();
    }
}

```

o/p

```

Room 101 - Cleaning step 1
Room 102 - Cleaning step 1
Room 101 - Cleaning step 2
Room 102 - Cleaning step 2
Room 101 - Cleaning step 3
Room 102 - Cleaning step 3
...

```

2. Thread creation using Runnable interface

// Runnable implementation

```

class RoomCleaningTask implements Runnable {

```

```

    private String roomName;

```

// Constructor

```

RoomCleaningTask(String roomName) {
    this.roomName = roomName;
}

```

```

    }

    // Implementing run() method
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(roomName + " - Cleaning step " + i);
            try {
                Thread.sleep(500); // Pause execution
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

```

```

// Main class
public class ThreadUsingRunnableInterface {
    public static void main(String[] args) {

        // Creating Runnable objects
        RoomCleaningTask task1 = new RoomCleaningTask("Room 201");
        RoomCleaningTask task2 = new RoomCleaningTask("Room 202");

        // Creating Thread objects
        Thread t1 = new Thread(task1);
        Thread t2 = new Thread(task2);

        // Starting threads
        t1.start();
        t2.start();
    }
}

```

o/p

```

Room 201 - Cleaning step 1
Room 202 - Cleaning step 1
Room 201 - Cleaning step 2
Room 202 - Cleaning step 2

```

Room 201 - Cleaning step 3

Room 202 - Cleaning step 3

...

3. Demo of join yield etc methods

// Thread class demonstrating sleep(), join(), and yield()

```
class BookingThread extends Thread {

    private String taskName;

    BookingThread(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {

        for (int i = 1; i <= 5; i++) {
            System.out.println(taskName + " - Processing step " + i);

            // yield() gives chance to other threads
            Thread.yield();

            try {
                // sleep() pauses execution
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }

        System.out.println(taskName + " completed.");
    }
}

// Main class
public class ThreadMethodsDemo {
    public static void main(String[] args) {
```

```

BookingThread t1 = new BookingThread("Room Booking");
BookingThread t2 = new BookingThread("Payment Processing");

// Start first thread
t1.start();

try{
    // join() makes main thread wait until t1 completes
    t1.join();
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
}

// Start second thread after t1 finishes
t2.start();

System.out.println("Main thread completed.");
}
}

```

o/p

```

Room Booking - Processing step 1
Room Booking - Processing step 2
Room Booking - Processing step 3
Room Booking - Processing step 4
Room Booking - Processing step 5
Room Booking completed.
Payment Processing - Processing step 1
Payment Processing - Processing step 2
Payment Processing - Processing step 3
Payment Processing - Processing step 4
Payment Processing - Processing step 5
Payment Processing completed.
Main thread completed.

```

1. Design and implement a Java application to simulate a **Hotel Room Service Management System** where multiple service requests are handled concurrently using **multithreading**.
 - i. In a hotel, different room service tasks such as **room cleaning**, **food delivery**, and **maintenance** may occur at the same time. To efficiently manage these tasks, the application should create **separate threads** for each service request so that they can execute concurrently rather than sequentially.
 - ii. Create individual threads for different service operations using Java thread creation techniques (`Thread` class or `Runnable` interface). Each thread should simulate a service task by displaying status messages and pausing execution using the `sleep()` method to represent processing time.
 - iii. The main program should start multiple threads simultaneously and demonstrate concurrent execution of hotel service tasks.
2. Design and implement a Java application to simulate an **Online Order Processing System** where multiple customer orders are processed simultaneously using **multithreading**.
 - i. In an e-commerce platform, several operations such as **order validation**, **payment processing**, and **order shipment** must be handled concurrently for different customers. To improve system performance and responsiveness, each order processing task should be executed in a **separate thread**.
 - ii. Create individual threads for handling different customer orders or different stages of order processing. Each thread should simulate processing by displaying status messages and using the `sleep()` method to represent time-consuming operations.
 - iii. The main program should start multiple threads at the same time and demonstrate concurrent execution of order-related tasks.

WEEK 4: Multithreaded Programming – Synchronization

Objective: To study thread synchronization and inter-thread communication.

Topics Covered

- Synchronization methods
- Synchronization blocks
- Deadlock
- Inter-thread communication (`wait()`, `notify()`)

Synchronization prevents thread interference when multiple threads access shared resources.

Sample exercises

1. // Shared resource

```
class BankAccount {  
  
    private int balance = 1000;  
  
    // Synchronized method  
    synchronized void withdraw(int amount) {  
  
        System.out.println(Thread.currentThread().getName()  
            + " is trying to withdraw " + amount);  
  
        if (balance >= amount) {  
            System.out.println(Thread.currentThread().getName()  
                + " is processing withdrawal...");  
            try {  
                Thread.sleep(1000); // Simulate processing time  
            } catch (InterruptedException e) {  
                System.out.println("Thread interrupted");  
            }  
            balance -= amount;  
            System.out.println(Thread.currentThread().getName()  
                + " completed withdrawal. Remaining balance: " + balance);  
        } else {  
            System.out.println(Thread.currentThread().getName()  
                + " - Insufficient balance!");  
        }  
    }  
}
```

// Thread class

```
class Customer extends Thread {  
  
    private BankAccount account;  
    private int amount;  
  
    Customer(BankAccount account, int amount, String name) {
```

```

        super(name);
        this.account = account;
        this.amount = amount;
    }

```

```

    @Override
    public void run() {
        account.withdraw(amount);
    }
}

```

// Main class

```

public class SynchronizedMethodDemo {
    public static void main(String[] args) {

```

```

        BankAccount account = new BankAccount();

```

```

        // Creating multiple threads accessing same resource

```

```

        Customer c1 = new Customer(account, 700, "Customer-1");

```

```

        Customer c2 = new Customer(account, 500, "Customer-2");

```

```

        c1.start();

```

```

        c2.start();

```

```

    }

```

```

}

```

o/p

Customer-1 is trying to withdraw 700

Customer-1 is processing withdrawal...

Customer-1 completed withdrawal. Remaining balance: 300

Customer-2 is trying to withdraw 500

Customer-2 - Insufficient balance!

2. // Shared resource

```

class BankAccount {

```

```

    private int balance = 1000;

```

```

    void withdraw(int amount) {

```

```

System.out.println(Thread.currentThread().getName()
    + " is trying to withdraw " + amount);

// Synchronized block
synchronized (this) {

    if (balance >= amount) {
        System.out.println(Thread.currentThread().getName()
            + " is processing withdrawal...");
        try {
            Thread.sleep(1000); // Simulate processing time
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
        balance -= amount;
        System.out.println(Thread.currentThread().getName()
            + " completed withdrawal. Remaining balance: " + balance);
    } else {
        System.out.println(Thread.currentThread().getName()
            + " - Insufficient balance!");
    }
}
}
}

```

// Thread class

```
class Customer extends Thread {
```

```
    private BankAccount account;
```

```
    private int amount;
```

```
    Customer(BankAccount account, int amount, String name) {
```

```
        super(name);
```

```
        this.account = account;
```

```
        this.amount = amount;
```

```
    }
```

```
@Override
```



```

    public void run() {
        account.withdraw(amount);
    }
}

```

// Main class

```

public class SynchronizedBlockDemo {
    public static void main(String[] args) {

        BankAccount account = new BankAccount();

        // Creating multiple threads accessing same resource
        Customer c1 = new Customer(account, 700, "Customer-1");
        Customer c2 = new Customer(account, 500, "Customer-2");

        c1.start();
        c2.start();
    }
}

```

o/p

```

Customer-1 is trying to withdraw 700
Customer-1 is processing withdrawal...
Customer-1 completed withdrawal. Remaining balance: 300
Customer-2 is trying to withdraw 500
Customer-2 - Insufficient balance!

```

3. // Shared resource

```

class Buffer {

    private int data;
    private boolean available = false;

    // Method to produce data
    synchronized void produce(int value) {

        while (available) {
            try {
                wait(); // Wait if buffer is full
            } catch (InterruptedException e) {

```

```

        System.out.println("Producer interrupted");
    }
}

data = value;
available = true;
System.out.println("Produced: " + data);

notify(); // Notify consumer
}

// Method to consume data
synchronized int consume() {

    while (!available) {
        try {
            wait(); // Wait if buffer is empty
        } catch (InterruptedException e) {
            System.out.println("Consumer interrupted");
        }
    }

    available = false;
    System.out.println("Consumed: " + data);

    notify(); // Notify producer
    return data;
}
}

// Producer thread
class Producer extends Thread {

    private Buffer buffer;

    Producer(Buffer buffer) {
        this.buffer = buffer;
    }
}

```

```

@Override
public void run() {

    for (int i = 1; i <= 5; i++) {
        buffer.produce(i);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            System.out.println("Producer sleep interrupted");
        }
    }
}

// Consumer thread
class Consumer extends Thread {

    private Buffer buffer;

    Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {

        for (int i = 1; i <= 5; i++) {
            buffer.consume();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Consumer sleep interrupted");
            }
        }
    }
}

```

```
// Main class
public class ProducerConsumerDemo {
    public static void main(String[] args) {

        Buffer buffer = new Buffer();

        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

        producer.start();
        consumer.start();
    }
}
```

o/p

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

4. Deadlock Demonstration code

```
// Deadlock demonstration
class DeadlockDemo {

    // Two shared resources
    static final Object resource1 = new Object();
    static final Object resource2 = new Object();

    // Thread 1
    static class ThreadA extends Thread {
        @Override
        public void run() {
            synchronized (resource1) {
                System.out.println("Thread A locked Resource 1");
            }
        }
    }
}
```

```

        try{
            Thread.sleep(500);
        } catch (InterruptedException e) {
            System.out.println("Thread A interrupted");
        }

        System.out.println("Thread A waiting for Resource 2");
        synchronized (resource2) {
            System.out.println("Thread A locked Resource 2");
        }
    }
}

// Thread 2
static class ThreadB extends Thread {
    @Override
    public void run() {
        synchronized (resource2) {
            System.out.println("Thread B locked Resource 2");

            try{
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Thread B interrupted");
            }

            System.out.println("Thread B waiting for Resource 1");
            synchronized (resource1) {
                System.out.println("Thread B locked Resource 1");
            }
        }
    }
}

// Main method
public static void main(String[] args) {

```

```

        Thread t1 = new ThreadA();
        Thread t2 = new ThreadB();

        t1.start();
        t2.start();
    }
}
o/p
Thread A locked Resource 1
Thread B locked Resource 2
Thread A waiting for Resource 2
Thread B waiting for Resource 1
Hangs''''''''

```

Lab Exercises

1. Design and implement a Java-based hotel room management application that simulates **concurrent room booking and room release operations** using **multiple threads**. The system must ensure **data consistency** when multiple customers attempt to book or release rooms simultaneously. A hotel has a limited number of rooms. Multiple customer threads attempt to book rooms at the same time. If no rooms are available, the booking thread must **wait**. When a room is released by another thread, the waiting booking thread must be **notified** and allowed to proceed.

WEEK 5: Input/Output Streams – Byte Streams , Character Streams

Objective: To perform file operations using byte-oriented I/O streams and character oriented I/O Streams

Topics Covered

- Difference between byte and character stream
- FileInputStream
- FileOutputStream
- Reading and writing binary files
- FileReader
- FileWriter

Byte streams handle raw binary data and are suitable for images, audio, and other non-text files. Character streams are used for text files.

Sample Exercises

2. FileInputStream demo

```
import java.io.FileInputStream;

import java.io.IOException;

public class FileInputStreamDemo {

    public static void main(String[] args) {

        FileInputStream fis = null;

        try {

            // Open the file

            fis = new FileInputStream("input.txt");

            int data;

            System.out.println("File contents:");

            // Read file byte by byte

            while ((data = fis.read()) != -1) {

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

            }

        }
```

```

    } catch (IOException e) {

        System.out.println("File error: " + e.getMessage());

    } finally {

        try {

            if (fis != null) {

                fis.close(); // Close the file

            }

        } catch (IOException e) {

            System.out.println("Error closing file");

        }

    }

}

```

Suppose input.text has

Welcome to Java FileInputStream.

This is a sample file.
o/p will be

File contents:

Welcome to Java FileInputStream.

This is a sample file.

3. FileOutputStream demo

```

import java.io.FileOutputStream;
import java.io.IOException;

```

```

public class FileOutputStreamDemo {

```



```

public static void main(String[] args) {

    FileOutputStream fos = null;

    try {
        // Create or open the file
        fos = new FileOutputStream("output.txt");

        String message = "Welcome to Java FileOutputStream.\nFile writing example.";

        // Convert string to bytes
        byte[] data = message.getBytes();

        // Write data to file
        fos.write(data);

        System.out.println("Data written to file successfully.");

    } catch (IOException e) {
        System.out.println("File error: " + e.getMessage());
    } finally {
        try {
            if (fos != null) {
                fos.close(); // Close the file
            }
        } catch (IOException e) {
            System.out.println("Error closing file");
        }
    }
}

```

Output.txt will contain
Welcome to Java FileOutputStream.
File writing example.
o/p: Data written to file successfully.

4. File Reader demo

```
import java.io.FileReader;
```

```
import java.io.IOException;
```

```
public class FileReaderDemo {
```

```
    public static void main(String[] args) {
```

```
        try {
```

```
            FileReader fr = new FileReader("input.txt");
```

```
            int ch;
```

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

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

```
            }
```

```
            fr.close();
```

```
        } catch (IOException e) {
```

```
            System.out.println("File error: " + e.getMessage());
```

```
        }
```

```
    }
```

```
}
```

Input.txt

Hello Java

FileReader Demo

o/p

Hello Java

FileReader Demo

Try with resources example

```
import java.io.FileReader;

import java.io.IOException;


public class FileReaderDemo {

    public static void main(String[] args) {

        try (FileReader fr = new FileReader("input.txt")) {

            int ch;

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

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

            }

        } catch (IOException e) {

            System.out.println("Error: " + e.getMessage());

        }

    }

}
```

5. File Writer demo

```
import java.io.FileWriter;

import java.io.IOException;


public class FileWriterDemo {

    public static void main(String[] args) {
```

```

try {
    FileWriter fw = new FileWriter("output.txt");

    fw.write("Hello Java\n");
    fw.write("FileWriter Demo");

    fw.close();
    System.out.println("Data written successfully.");
} catch (IOException e) {
    System.out.println("File error: " + e.getMessage());
}
}

```

Output.txt

Hello Java

FileWriter Demo

Appending mode

```
import java.io.FileWriter;
```

```
import java.io.IOException;
```

```

public class FileWriterDemo {
    public static void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("output.txt", true);

            fw.write("\nAppending new data...");
            fw.close();

            System.out.println("Data appended successfully.");
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

Try with resources

```
import java.io.FileWriter;
```

```
import java.io.IOException;
```

```

public class FileWriterDemo {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            fw.write("Using try-with-resources");
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

Lab exercises

1. Design and implement a Java application that copies the contents of one file to another using **byte streams**. The program must use `FileInputStream` to read data from a source file and `FileOutputStream` to write the same data to a destination file.
2. Design and implement a Java application that reads textual data from an existing text file using `FileReader` and writes the same content into another text file using `FileWriter`.

WEEK 6: Random Access File, Serialization and Deserialization

Topics to be covered

RandomAccessFile

Serialization

Deserialization

Sample Programs

1. Random Accessfile usage

```

import java.io.RandomAccessFile;
import java.io.IOException;

```

```

public class RandomAccessFileDemo {
    public static void main(String[] args) {
        try {
            // Open file in read-write mode
            RandomAccessFile raf = new RandomAccessFile("data.txt", "rw");

            // Writing data to file
            raf.writeInt(101);
            raf.writeUTF("Java");
            raf.writeDouble(99.5);
        }
    }
}

```

```

// Move file pointer to beginning
raf.seek(0);

// Reading data sequentially
int id = raf.readInt();
String name = raf.readUTF();
double marks = raf.readDouble();

System.out.println("ID: " + id);
System.out.println("Name: " + name);
System.out.println("Marks: " + marks);

// Random access: move pointer and read again
raf.seek(4); // Skips integer (4 bytes)
System.out.println("Name (Random Access): " + raf.readUTF());

raf.close();
} catch (IOException e) {
    System.out.println("File error: " + e.getMessage());
}
}

```

ID: 101

Name: Java

Marks: 99.5

Name (Random Access): Java

2. Serialization and deserialization

```
import java.io.Serializable;
```

```
public class Student implements Serializable {
    int id;
    String name;
    double marks;
```

```

    Student(int id, String name, double marks) {
        this.id = id;
        this.name = name;
    }
}

```

```

        this.marks = marks;
    }
}
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class SerializeDemo {
    public static void main(String[] args) {
        try {
            Student s1 = new Student(101, "Ravi", 85.5);

            FileOutputStream fos = new FileOutputStream("student.dat");
            ObjectOutputStream oos = new ObjectOutputStream(fos);

            oos.writeObject(s1);

            oos.close();
            fos.close();

            System.out.println("Object serialized successfully.");
        } catch (IOException e) {
            System.out.println("Serialization error: " + e.getMessage());
        }
    }
}

//Deserialization
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.lang.ClassNotFoundException;

public class DeserializeDemo {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("student.dat");
            ObjectInputStream ois = new ObjectInputStream(fis);

```

```

Student s = (Student) ois.readObject();

System.out.println("ID: " + s.id);
System.out.println("Name: " + s.name);
System.out.println("Marks: " + s.marks);

ois.close();
fis.close();
} catch (IOException | ClassNotFoundException e) {
    System.out.println("Deserialization error: " + e.getMessage());
}
}
}

```

Object serialized successfully.

ID: 101

Name: Ravi

Marks: 85.5

Lab exercises

1. Design and implement a Java application to manage hotel room bookings where room records are stored in a file and accessed using **RandomAccessFile**. Each room record should be of **fixed length**, enabling direct (random) access to any room's booking information without reading the file sequentially.

The system must support operations such as **adding rooms**, **viewing room details**, and **updating booking status** by directly navigating to the required record position in the file.

- i. hotel room details in a file using RandomAccessFile.
- ii. Each room record must contain:
- iii. Room Number (int)
- iv. Room Type (fixed-length String, e.g., 20 characters)
- v. Price per Night (double)
- vi. Booking Status (boolean)
- vii. Provide an option to:
- viii. Add new room records
- ix. Display details of a specific room using its room number
- x. Update booking status (book / vacate a room)
- xi. Use the seek() method to jump directly to the position of a room record.
- xii. Ensure data is read and written in the same sequence and format.
- xiii. Close the file after each operation.

2. Sign and implement a Java application for managing hotel room bookings where room booking details are stored as **serialized objects** in a file. The application should use **serialization** to save hotel room booking objects permanently and **deserialization** to retrieve them when required.

This approach should simulate real-world object persistence without using a database.

- i. Create a Room class that implements Serializable.
- ii. Each room object must store:
 - a. Room Number
 - b. Room Type
 - c. Price per Night
 - d. Booking Status
 - e. Guest Name
- iii. Serialize room booking objects and store them in a file.
- iv. Deserialize objects from the file to:
 - a. Display all room details
 - b. Search for a room using room number
- v. Allow updating booking status by:
 - a. Deserializing the objects
 - b. Modifying the required room object
 - c. Re-serializing the updated objects back to the file
- vi. Handle file and class-related exceptions properly.

WEEK 7: Generics

Topics to be covered

- Generic classes with two type parameters
- Bounded types

Generic methods

Sample Programs

1. Create a generic class with two type parameters.

```
class Pair<T, U> {  
    private T first;  
    private U second;  
  
    public Pair(T first, U second) {  
        this.first = first;  
    }  
}
```

```

        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }

    public void display() {
        System.out.println("First Value: " + first);
        System.out.println("Second Value: " + second);
    }
}

```

```

public class GenericDemo {
    public static void main(String[] args) {
        Pair<Integer, String> p1 = new Pair<>(101, "Deluxe Room");
        p1.display();

        Pair<String, Double> p2 = new Pair<>("Room Price", 3500.50);
        p2.display();
    }
}

```

o/p

First Value: 101

Second Value: Deluxe Room

First Value: Room Price

Second Value: 3500.5

1. bounded types

```

class NumberOperations<T extends Number> {
    private T num1;
    private T num2;

```

```

    public NumberOperations(T num1, T num2) {
        this.num1 = num1;

```

```

        this.num2 = num2;
    }

    public double add() {
        return num1.doubleValue() + num2.doubleValue();
    }

    public double multiply() {
        return num1.doubleValue() * num2.doubleValue();
    }
}

public class BoundedTypeDemo {
    public static void main(String[] args) {

        NumberOperations<Integer> obj1 =
            new NumberOperations<>(10, 20);
        System.out.println("Addition (Integer): " + obj1.add());
        System.out.println("Multiplication (Integer): " + obj1.multiply());

        NumberOperations<Double> obj2 =
            new NumberOperations<>(5.5, 2.0);
        System.out.println("Addition (Double): " + obj2.add());
        System.out.println("Multiplication (Double): " + obj2.multiply());

        // Compilation error if non-number type is used
        // NumberOperations<String> obj3 = new NumberOperations<>("A", "B");
    }
}

Addition (Integer): 30.0
Multiplication (Integer): 200.0
Addition (Double): 7.5
Multiplication (Double): 11.0

```

2. Generic methods

```

class GenericMethodDemo {

    // Generic method
    public static <T> void display(T value) {

```

```

        System.out.println("Value: " + value);
    }

```

```

public static void main(String[] args) {

```

```

    display(100);    // Integer
    display("Java"); // String
    display(45.6);   // Double
    display(true);   // Boolean
}
}

```

o/p

Value: 100

Value: Java

Value: 45.6

Value: true

3. Generic method with array parameter

```

class GenericArrayDemo {

```

```

    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}

```

```

public static void main(String[] args) {

```

```

    Integer[] nums = {1, 2, 3, 4};
    String[] names = {"Amit", "Ravi", "Neha"};

```

```

    printArray(nums);
    printArray(names);
}

```

```

}

```

1 2 3 4

Amit Ravi Neha

4. Bounded Generic method

```

class GenericBoundedMethodDemo {

    public static <T extends Number> double sum(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }

    public static void main(String[] args) {

        System.out.println(sum(10, 20));    // Integer
        System.out.println(sum(5.5, 2.5));  // Double

        // Compilation error
        // System.out.println(sum("A", "B"));
    }
}

```

Lab exercises

1. Develop a Java application that uses a **generic class with two type parameters** to store hotel room information. The generic class should be capable of holding different data types for room identifiers and room attributes.
 - Create a generic class `Room<T, U>`
 - `T` represents Room Number or Room ID
 - `U` represents Room Type or Price
 - Demonstrate usage with different data types (e.g., `Integer`, `String`, `Double`)
 - Display stored room details
2. Create a Java program for a hotel room management system that uses a **generic method** to display room-related data of different types such as room numbers, room types, prices, and booking status.
 - Implement a generic method `<T> void display(T data)`
 - Call the method with:
 1. Room number (`Integer`)
 2. Room type (`String`)
 3. Price per night (`Double`)
 4. Booking status (`Boolean`)
 - Ensure type safety without explicit casting
3. Develop a Java application that uses **bounded type parameters** to calculate room charges. The application should accept only numeric values for pricing and discount calculations.

- Create a generic class or method using `<T extends Number>`
 - Accept room price and discount as generic parameters
 - Perform calculations such as:
 - Total price
 - Discounted price
 - Prevent non-numeric data types at compile time
4. Design a Java program that uses **generic methods** to manage an array of hotel rooms. The program should be capable of storing and displaying arrays of different room attributes.
- Create a generic method to print arrays
 - Use it for:
 - Room numbers array
 - Room types array
 - Room prices array
 - Do not use collections framework
5. Develop a hotel room booking module that uses a **generic pair class** to associate room numbers with guest details.
- Create a generic class `Pair<T, U>`
 - Store:
 - Room Number (`Integer`)
 - Guest Name (`String`)
 - Display booking records
 - Ensure no type casting is required

WEEK 8: Collection Framework – List Interface

Topics to be covered

- Collection interfaces
- List interface
- ArrayList
- Iterator

Collections provide dynamic data structures for storing and manipulating groups of objects.

Sample programs

1. Program demonstrating ArrayList operations.

```
import java.util.ArrayList;
```

```
import java.util.Collections;

import java.util.Iterator;


public class ArrayListDemo {


    public static void main(String[] args) {


        // 1. Creating an ArrayList

        ArrayList<String> students = new ArrayList<>();


        // 2. Adding elements

        students.add("Amit");

        students.add("Riya");

        students.add("Neha");

        students.add("Karan");


        System.out.println("Initial ArrayList: " + students);


        // 3. Adding element at a specific index

        students.add(2, "Sonal");

        System.out.println("After insertion at index 2: " + students);


        // 4. Accessing elements
```

```
System.out.println("Element at index 1: " + students.get(1));
```

```
// 5. Updating elements
```

```
students.set(3, "Rahul");
```

```
System.out.println("After updating index 3: " + students);
```

```
// 6. Removing elements
```

```
students.remove("Neha"); // remove by value
```

```
students.remove(0); // remove by index
```

```
System.out.println("After removals: " + students);
```

```
// 7. Checking size
```

```
System.out.println("Size of ArrayList: " + students.size());
```

```
// 8. Searching elements
```

```
System.out.println("Contains 'Riya'? " + students.contains("Riya"));
```

```
// 9. Iterating using for-each loop
```

```
System.out.println("Iterating using for-each:");
```

```
for (String name : students) {
```

```
    System.out.println(name);
```

```
}
```



```
// 10. Iterating using Iterator

System.out.println("Iterating using Iterator:");

Iterator<String> it = students.iterator();

while (it.hasNext()) {

    System.out.println(it.next());

}
```

```
// 11. Sorting the ArrayList

Collections.sort(students);

System.out.println("After sorting: " + students);
```

```
// 12. Clearing the ArrayList

students.clear();

System.out.println("After clearing: " + students);

}

}
```

o/p:

2. Program accessing collection elements using Iterator.

```
import java.util.ArrayList;

import java.util.Iterator;

public class IteratorDemo {
```

```
public static void main(String[] args) {

    // Creating a collection

    ArrayList<Integer> numbers = new ArrayList<>();

    // Adding elements to the collection

    numbers.add(10);

    numbers.add(20);

    numbers.add(30);

    numbers.add(40);

    // Obtaining an Iterator

    Iterator<Integer> itr = numbers.iterator();

    // Accessing elements using Iterator

    System.out.println("Elements in the ArrayList:");

    while (itr.hasNext()) {

        Integer value = itr.next();

        System.out.println(value);

    }

}
```

3. Sorting a list

```

import java.util.ArrayList;
import java.util.Collections;

public class ListSortingDemo {

    public static void main(String[] args) {

        // Creating a list
        ArrayList<Integer> numbers = new ArrayList<>();

        // Adding elements to the list
        numbers.add(45);
        numbers.add(10);
        numbers.add(30);
        numbers.add(25);
        numbers.add(5);

        // Displaying original list
        System.out.println("Original List: " + numbers);

        // Sorting the list in ascending order
        Collections.sort(numbers);

        // Displaying sorted list
        System.out.println("Sorted List (Ascending): " + numbers);

        // Sorting the list in descending order
        Collections.sort(numbers, Collections.reverseOrder());

        // Displaying descending order list
        System.out.println("Sorted List (Descending): " + numbers);
    }
}

```

Lab exercises

1. Design and implement a **Hotel Management System** in Java using the **Collection Framework** to manage hotel operations efficiently. The system should store, retrieve, update, and process hotel-related data dynamically using appropriate collection classes.

System Requirements

Room Management

Store room details such as:

Room Number

Room Type (Single, Double, Deluxe, Suite)

Room Price per Day

Availability Status

Allow adding new rooms and displaying all available rooms.

Customer Management

Maintain customer details including:

Customer ID

Name

Contact Number

Room Number Allocated

Support operations to add, view, and remove customer records.

Booking Management

Enable room booking and checkout functionality.

Update room availability automatically after booking or checkout.

Prevent booking of already occupied rooms.

Collections Usage

Use suitable collection classes such as:

ArrayList to store room and customer objects

HashMap to map room numbers to customer details

Iterator to traverse and manage records

Apply sorting (e.g., by room price or room number) using Collections.sort().

Menu-Driven Interface

Provide a console-based menu with options such as:

Add Room

Display Available Rooms

Add Customer

Book Room

Checkout Customer

Display All Customers

Exit

Constraints

Do not use arrays for data storage.

All data must be handled using Java Collections.

Ensure proper validation and exception handling.

WEEK 9: Basic JavaFX GUI Programming

Objective: To design simple graphical user interfaces using JavaFX.

Topics to be covered:

- JavaFX architecture
- Stage and Scene

- Controls: Button, TextField, Label
- Event handling

Sample Programs

1. JavaFX program displaying a button.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ButtonDemo extends Application {

    @Override
    public void start(Stage stage) {

        // Creating a Button
        Button btn = new Button("Click Me");

        // Adding button to layout
        StackPane root = new StackPane();
        root.getChildren().add(btn);

        // Creating a Scene
        Scene scene = new Scene(root, 300, 200);

        // Setting stage properties
        stage.setTitle("JavaFX Button Example");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

2. Program using TextField and Label.

```
import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TextFieldLabelDemo extends Application {

    @Override
    public void start(Stage stage) {

        // Creating controls
        Label lblMessage = new Label("Enter your name:");
        TextField txtName = new TextField();
        Button btnShow = new Button("Display");

        Label lblResult = new Label();

        // Button action
        btnShow.setOnAction(e -> {
            String name = txtName.getText();
            lblResult.setText("Hello, " + name);
        });

        // Layout
        VBox root = new VBox(10);
        root.getChildren().addAll(lblMessage, txtName, btnShow, lblResult);

        // Scene and Stage
        Scene scene = new Scene(root, 300, 200);
        stage.setTitle("TextField and Label Example");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

```
}  
}
```

3. Event handling for button click.

```
import javafx.application.Application;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.layout.VBox;  
import javafx.stage.Stage;
```

```
public class ButtonEventDemo extends Application {
```

```
    @Override
```

```
    public void start(Stage stage) {
```

```
        // Creating controls
```

```
        Button btnClick = new Button("Click Here");
```

```
        Label lblMessage = new Label("Waiting for button click...");
```

```
        // Event handling for button click
```

```
        btnClick.setOnAction(e -> {
```

```
            lblMessage.setText("Button Clicked!");
```

```
        });
```

```
        // Layout
```

```
        VBox root = new VBox(15);
```

```
        root.getChildren().addAll(btnClick, lblMessage);
```

```
        // Scene and Stage
```

```
        Scene scene = new Scene(root, 300, 200);
```

```
        stage.setTitle("Button Click Event Handling");
```

```
        stage.setScene(scene);
```

```
        stage.show();
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        launch(args);
```

```
    }
```

```
}
```

4. Simple form-based application.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
```

```
public class SimpleFormDemo extends Application {
```

```
    @Override
```

```
    public void start(Stage stage) {
```

```
        // Creating labels
```

```
        Label lblName = new Label("Name:");
```

```
        Label lblEmail = new Label("Email:");
```

```
        Label lblResult = new Label();
```

```
        // Creating text fields
```

```
        TextField txtName = new TextField();
```

```
        TextField txtEmail = new TextField();
```

```
        // Creating button
```

```
        Button btnSubmit = new Button("Submit");
```

```
        // Button event handling
```

```
        btnSubmit.setOnAction(e -> {
```

```
            String name = txtName.getText();
```

```
            String email = txtEmail.getText();
```

```
            lblResult.setText("Submitted:\nName: " + name + "\nEmail: " + email);
```

```
        });
```

```
        // Layout using GridPane
```

```
        GridPane grid = new GridPane();
```

```
        grid.setHgap(10);
```

```
        grid.setVgap(10);
```



```

        grid.add(lblName, 0, 0);
        grid.add(txtName, 1, 0);
        grid.add(lblEmail, 0, 1);
        grid.add(txtEmail, 1, 1);
        grid.add(btnSubmit, 1, 2);
        grid.add(lblResult, 1, 3);

        // Scene and Stage
        Scene scene = new Scene(grid, 350, 250);
        stage.setTitle("Simple Form Application");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

Lab exercise

1. Design and implement a **Hotel Management System** with a **Graphical User Interface (GUI)** using **JavaFX** to automate and simplify basic hotel operations. The application should provide an interactive, user-friendly interface for managing rooms, customers, and bookings.

Room Management

- Display room details such as:
 - Room Number
 - Room Type (Single, Double, Deluxe)
 - Price per Day
 - Availability Status
- Provide GUI options to:
 - Add new rooms
 - View all rooms
 - Show available rooms only

Customer Management

- Capture customer information through forms:
 - Customer Name
 - Contact Number

- Selected Room Number
- Display customer booking details in the GUI.
- ? **Booking and Checkout**
- Allow booking of rooms using a button click.
- Prevent booking of already occupied rooms.
- Provide a checkout option to release rooms and update availability.
- ? **GUI Requirements**
- Use JavaFX controls such as:
 - Label, TextField, Button, ComboBox, TableView
- Use suitable layouts:
 - GridPane for forms
 - VBox / HBox for buttons and navigation
- Implement **event handling** for all user actions.
- ? **User Interaction**
- Display confirmation or error messages using labels or alerts.
- Clear input fields after successful operations.

Week 10 : Concluding with complete Hotel management application

Guidelines: ? The application should be menu-driven or tab-based.

? Code should be modular and well-structured.

? No database is required; in-memory/file data handling is sufficient.

? Application must run as a standalone JavaFX desktop application.

? Add and view room details

? Book and checkout rooms

Marking scheme:

1. Basic System => 5M
2. GUI design, other additional features,=>5M

After WEEK5 midterm exam for 20M

Record :5+5=10M

Evaluation before midterm 10M

Evaluation After midterm 10M

After WEEK10 Final exam for 40 M

