1. Design a Java program to manage student records using a 'Student' class. The 'Student' class should include attributes like 'name', 'rollNumber', and an array 'marks' to store marks obtained in different subjects. Implement a constructor in the 'Student' class utilizing the 'this' keyword to initialize these attributes. Additionally, include methods for displaying student details, calculating the average marks, and updating marks for a specific subject. Utilize the 'Scanner' class for user input and ensure explicit error handling for invalid inputs, such as non- numeric marks or marks out of range (considering the maximum marks for each subject). Specify that the maximum marks for each subject are 100, and the program should consider a maximum of 5 subjects. Create a 'StudentManager' class with a main method to demonstrate the functionalities of the Student class. The program should enable users to create multiple Student objects, update their marks, and display their details.

```
import java.util.Scanner;
import java.util.InputMismatchException;

class Student {
    private String name;
    private int rollNumber;
    private int[] marks;

    public Student(String name, int rollNumber, int numSubjects) {
        this.name = name;
        this.rollNumber = rollNumber;
        this.marks = new int[numSubjects];
    }

    public void setMarks(int subjectIndex, int mark) {
        if (subjectIndex >= 0 && subjectIndex < marks.length && mark >= 0 && mark <= 100) {
            marks[subjectIndex] = mark;
        } else {
            System.out.println("Invalid subject index or mark value.");
        }
    }

    public double calculateAverage() {
        int sum = 0;
        for (int mark : marks) {
            sum += mark;
        }
        return (double) sum / marks.length;
    }

    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Roll Number: " + rollNumber);
        System.out.print("Marks: ");
        for (int mark : marks) {
            System.out.print(mark + " ");
        }
```

```
        System.out.println("\nAverage Marks: " + calculateAverage());
    }
}

public class StudentManager {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter the number of students: ");
            int numStudents = scanner.nextInt();
            scanner.nextLine(); // Consume the newline character

            Student[] students = new Student[numStudents];

            for (int i = 0; i < numStudents; i++) {
                System.out.println("\nEnter details for Student " + (i + 1) + ":");

                System.out.print("Name: ");
                String name = scanner.nextLine();

                System.out.print("Roll Number: ");
                int rollNumber = scanner.nextInt();
                scanner.nextLine(); // Consume the newline character

                students[i] = new Student(name, rollNumber, 5); // Assuming 5 subjects

                System.out.println("Enter marks for 5 subjects (0-100):");
                for (int j = 0; j < 5; j++) {
                    System.out.print("Subject " + (j + 1) + ": ");
                    int mark = scanner.nextInt();
                    students[i].setMarks(j, mark);
                }
                scanner.nextLine(); // Consume the newline character
            }

            System.out.println("\nStudent Details:");
            for (Student student : students) {
                student.displayDetails();
                System.out.println();
            }
        } catch (InputMismatchException e) {
            System.out.println("Invalid input. Please enter a valid integer value.");
        } finally {
            scanner.close();
        }
    }
}
```

Design a Java program for a banking system that incorporates inheritance, method overloading. [10] and method overriding. Implement a superclass 'Account' with attributes like 'accountNumber', 'accountHolderName', and 'balance'. Include methods for 'deposit'. 'withdrawal', and displaying 'account information'. In the 'withdrawal' method of the base class 'Account', it takes two arguments: the withdrawal amount (type double) and the withdrawal limit (type double). Ensure that the withdrawal limit parameter is used only for the 'Savings Account' subclass. Extend the 'Account' class to create subclasses 'Savings Account' and 'CurrentAccount', representing different types of bank accounts. In the 'SavingsAccount' subclass, overload the 'withdrawal method to include an additional parameter for withdrawal limit, set to Rs. 50,000/-. Override the 'withdrawal' method in the 'CurrentAccount' subclass to implement specific overdraft functionality, with an overdraft limit set to Rs. 80,000/-.

Ensure input validation to handle errors:

- For invalid withdrawals: Ensure that the withdrawal amount is greater than zero.
- For insufficient funds: Check if the withdrawal al amount less than or equal to the account balance.
- For incorrect account details: Validate that the account number and account holder's name are not empty strings, and the balance is not negative.

```
import java.util.Scanner;

// Base class 'Account'
class Account {
    protected int accountNumber;
    protected String accountHolderName;
    protected double balance;

    public Account(int accountNumber, String accountHolderName, double balance) {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.balance = balance;
        validateAccountDetails();
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposit of Rs. " + amount + " successful. Updated balance: Rs. " + balance);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

    public void withdraw(double amount) {
        if (amount > 0) {
            if (amount <= balance) {
                balance -= amount;
                System.out.println("Withdrawal of Rs. " + amount + " successful. Updated balance: Rs. " + balance);
            } else {
                System.out.println("Insufficient funds. Cannot withdraw Rs. " + amount);
            }
```

```
        } else {
            System.out.println("Invalid withdrawal amount.");
        }
    }

    public void displayAccountInfo() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder Name: " + accountHolderName);
        System.out.println("Account Balance: Rs. " + balance);
    }

    private void validateAccountDetails() {
        if (accountNumber <= 0 || accountHolderName.isEmpty() || balance < 0) {
            throw new IllegalArgumentException("Invalid account details.");
        }
    }
}

// SavingsAccount subclass
class SavingsAccount extends Account {
    private static final double WITHDRAWAL_LIMIT = 50000.0;

    public SavingsAccount(int accountNumber, String accountHolderName, double balance) {
        super(accountNumber, accountHolderName, balance);
    }

    // Overloaded withdrawal method with withdrawal limit
    public void withdraw(double amount, double withdrawalLimit) {
        if (amount > 0) {
            if (amount <= balance && amount <= withdrawalLimit) {
                balance -= amount;
                System.out.println("Withdrawal of Rs. " + amount + " successful. Updated balance: Rs. " + balance);
            } else if (amount > withdrawalLimit) {
                System.out.println("Withdrawal amount exceeds the limit of Rs. " + withdrawalLimit);
            } else {
                System.out.println("Insufficient funds. Cannot withdraw Rs. " + amount);
            }
        } else {
            System.out.println("Invalid withdrawal amount.");
        }
    }
}

// CurrentAccount subclass
class CurrentAccount extends Account {
    private static final double OVERDRAFT_LIMIT = 80000.0;

    public CurrentAccount(int accountNumber, String accountHolderName, double balance) {
```

```
        super(accountNumber, accountHolderName, balance);
    }

    // Overridden withdrawal method with overdraft functionality
    @Override
    public void withdraw(double amount) {
        if (amount > 0) {
            if (amount <= balance + OVERDRAFT_LIMIT) {
                balance -= amount;
                System.out.println("Withdrawal of Rs. " + amount + " successful. Updated balance: Rs. " + balance);
            } else {
                System.out.println("Withdrawal amount exceeds the overdraft limit of Rs. " + OVERDRAFT_LIMIT);
            }
        } else {
            System.out.println("Invalid withdrawal amount.");
        }
    }
}

public class BankingSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Welcome to the Banking System!");
        System.out.println("1. Savings Account");
        System.out.println("2. Current Account");
        System.out.print("Enter your choice (1 or 2): ");
        int choice = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character

        System.out.print("Enter account number: ");
        int accountNumber = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character

        System.out.print("Enter account holder name: ");
        String accountHolderName = scanner.nextLine();

        System.out.print("Enter initial balance: ");
        double balance = scanner.nextDouble();

        Account account;

        if (choice == 1) {
            account = new SavingsAccount(accountNumber, accountHolderName, balance);
        } else if (choice == 2) {
            account = new CurrentAccount(accountNumber, accountHolderName, balance);
        } else {
            System.out.println("Invalid choice.");
```

```
            scanner.close();
            return;
        }

        System.out.println("\nAccount created successfully!");
        account.displayAccountInfo();

        while (true) {
            System.out.println("\nChoose an option:");
            System.out.println("1. Deposit");
            System.out.println("2. Withdraw");
            System.out.println("3. Display Account Information");
            System.out.println("4. Exit");
            System.out.print("Enter your choice (1-4): ");
            int option = scanner.nextInt();

            switch (option) {
                case 1:
                    System.out.print("Enter the deposit amount: ");
                    double depositAmount = scanner.nextDouble();
                    account.deposit(depositAmount);
                    break;
                case 2:
                    System.out.print("Enter the withdrawal amount: ");
                    double withdrawalAmount = scanner.nextDouble();
                    if (account instanceof SavingsAccount) {
                        System.out.print("Enter the withdrawal limit: ");
                        double withdrawalLimit = scanner.nextDouble();
                        ((SavingsAccount) account).withdraw(withdrawalAmount, withdrawalLimit);
                    } else {
                        account.withdraw(withdrawalAmount);
                    }
                    break;
                case 3:
                    account.displayAccountInfo();
                    break;
                case 4:
                    System.out.println("Thank you for using the Banking System!");
                    scanner.close();
                    return;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

3.Write a Java program that generates various patterns using loops. The program should prompt the user to select a pattern type and then print the corresponding pattern. The available pattern types are as follows:

Pattern 1: A pattern with an increasing number of stars on each line.

Pattern 2: A pattern with decreasing number of stars on each line.

Pattern 3: A pyramid pattern with increasing number of stars from top to bottom.

Pattern 4: A pyramid pattern with decreasing number of stars from top to bottom.

Pattern 5: A pattern with a diamond shape.

```
import java.util.Scanner;

public class PatternGenerator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Prompt user to select a pattern type
        System.out.println("Select a pattern type:");
        System.out.println("1. Pattern with increasing number of stars on each line");
        System.out.println("2. Pattern with decreasing number of stars on each line");
        System.out.println("3. Pyramid pattern with increasing number of stars from top to bottom");
        System.out.println("4. Pyramid pattern with decreasing number of stars from top to bottom");
        System.out.println("5. Pattern with diamond shape");
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();

        // Prompt user for pattern size
        System.out.print("Enter pattern size: ");
        int size = scanner.nextInt();

        // Generate pattern based on user's choice
        switch (choice) {
            case 1:
                generateIncreasingPattern(size);
                break;
            case 2:
                generateDecreasingPattern(size);
                break;
            case 3:
                generateIncreasingPyramid(size);
                break;
            case 4:
                generateDecreasingPyramid(size);
                break;
            case 5:
                generateDiamondPattern(size);
                break;
            default:
```

```
            System.out.println("Invalid choice!");
        }

        scanner.close();
    }

    // Method to generate pattern with increasing number of stars on each line
    public static void generateIncreasingPattern(int size) {
        for (int i = 1; i <= size; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }

    // Method to generate pattern with decreasing number of stars on each line
    public static void generateDecreasingPattern(int size) {
        for (int i = size; i >= 1; i--) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }

    // Method to generate pyramid pattern with increasing number of stars from top to bottom
    public static void generateIncreasingPyramid(int size) {
        for (int i = 1; i <= size; i++) {
            for (int j = size; j > i; j--) {
                System.out.print(" ");
            }
            for (int k = 1; k <= i; k++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }

    // Method to generate pyramid pattern with decreasing number of stars from top to bottom
    public static void generateDecreasingPyramid(int size) {
        for (int i = size; i >= 1; i--) {
            for (int j = size; j > i; j--) {
                System.out.print(" ");
            }
            for (int k = 1; k <= i; k++) {
                System.out.print("* ");
            }
```

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

    // Method to generate diamond pattern
    public static void generateDiamondPattern(int size) {
        for (int i = 1; i <= size; i++) {
            for (int j = size; j > i; j--) {
                System.out.print(" ");
            }
            for (int k = 1; k <= 2 * i - 1; k++) {
                System.out.print("*");
            }
            System.out.println();
        }
        for (int i = size - 1; i >= 1; i--) {
            for (int j = size - 1; j >= i; j--) {
                System.out.print(" ");
            }
            for (int k = 1; k <= 2 * i - 1; k++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Custom Border Diamond only

```
import java.util.Scanner;
public class PatternBorderGenerator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Select a pattern type:");
        System.out.println("1. Pattern with increasing number of stars on each line (border only)");
        System.out.println("2. Pattern with decreasing number of stars on each line (border only)");
        System.out.println("3. Pyramid pattern with increasing number of stars from top to bottom (border only)");
        System.out.println("4. Pyramid pattern with decreasing number of stars from top to bottom (border only)");
        System.out.println("5. Pattern with diamond shape (border only)");
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();
        System.out.print("Enter pattern size: ");
        int size = scanner.nextInt();
        switch (choice) {
            case 1:
                generateIncreasingPatternBorder(size);
                break;
```

```
            case 2:
                generateDecreasingPatternBorder(size);
                break;
            case 3:
                generateIncreasingPyramidBorder(size);
                break;
            case 4:
                generateDecreasingPyramidBorder(size);
                break;
            case 5:
                generateDiamondPatternBorder(size);
                break;
            default:
                System.out.println("Invalid choice!");
        }
        scanner.close();
    }
    // Method to generate border of diamond pattern
    public static void generateDiamondPatternBorder(int size) {
        for (int i = 1; i <= size; i++) {
            for (int j = size; j > i; j--) {
                System.out.print("  ");
            }
            for (int k = 1; k <= 2 * i - 1; k++) {
                if (k == 1 || k == 2 * i - 1) {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
        for (int i = size - 1; i >= 1; i--) {
            for (int j = size - 1; j >= i; j--) {
                System.out.print("  ");
            }
            for (int k = 1; k <= 2 * i - 1; k++) {
                if (k == 1 || k == 2 * i - 1) {
                    System.out.print("* ");
                } else {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
```

4. Design a Java program to cover different scenarios of exception handling. Create a User Defined exception class named 'InvalidAgeException' to handle situations where the user inputs an age less than 0 or greater than 100. Construct a class that receives user input for a number as a string and attempts to convert it to an integer, providing handling for 'NumberFormatException' if the input string is not a valid integer. Define a class 'Person' with attributes 'name' and 'age', create an object without assigning values to its members. Handle 'NullPointerException' when accessing the attributes of this object. Write a method to declare an array of size 5 and attempting to access the 6th element, with handling for ArrayIndexOutOfBoundsException.

```
import java.util.Scanner;

// User Defined Exception class for handling invalid age
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

// Class to handle NumberFormatException
class NumberConverter {
    public static int convertToInt(String str) {
        try {
            return Integer.parseInt(str);
        } catch (NumberFormatException e) {
            System.out.println("NumberFormatException: " + e.getMessage());
            return 0;
        }
    }
}

// Class to demonstrate NullPointerException
class Person {
    private String name;
    private int age;

    public void displayInfo() {
        try {
            System.out.println("Name: " + name);
            System.out.println("Age: " + age);
        } catch (NullPointerException e) {
            System.out.println("NullPointerException: " + e.getMessage());
        }
    }
}

public class ExceptionHandlingDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```
        // Scenario 1: Handling InvalidAgeException
        try {
            System.out.print("Enter age: ");
            int age = scanner.nextInt();
            if (age < 0 || age > 100) {
                throw new InvalidAgeException("Invalid age! Age should be between 0 and 100.");
            }
        } catch (InvalidAgeException e) {
            System.out.println("InvalidAgeException: " + e.getMessage());
        }

        // Scenario 2: Handling NumberFormatException
        System.out.print("Enter a number: ");
        String numStr = scanner.next();
        int num = NumberConverter.convertToInt(numStr);
        System.out.println("Converted number: " + num);

        // Scenario 3: Handling NullPointerException
        Person person = null;
        try {
            person.displayInfo();
        } catch (NullPointerException e) {
            System.out.println("NullPointerException: " + e.getMessage());
        }

        // Scenario 4: Handling ArrayIndexOutOfBoundsException
        try {
            int[] arr = new int[5];
            arr[5] = 10;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException: " + e.getMessage());
        }

        scanner.close();
    }
}
```

4. Create a JavaFX program for a simple 'calculator' app. The calculator should include a graphical user interface (GUI) with buttons for the numbers 0-9, operators (+, -, ", 1), and functions (clear, equal). It should utilize the layout GridPane to properly align these elements.Create code that does basic arithmetic operations (addition, subtraction, multiplication, and division) when the respective operator buttons are clicked. Make that the calculator displays the entered numbers as well as the operation's result. Create a code for the clear button to reset the calculator and an equal button to compute the results.

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

public class Calculator extends Application {

    private Label display;
    private double firstOperand;
    private String operation;

    @Override
    public void start(Stage primaryStage) {
        // Create the display label
        display = new Label("0");
        display.setStyle("-fx-font-size: 30; -fx-font-weight: bold;");
        display.setAlignment(Pos.CENTER_RIGHT);

        // Create the buttons
        Button button0 = new Button("0");
        Button button1 = new Button("1");
        Button button2 = new Button("2");
        Button button3 = new Button("3");
        Button button4 = new Button("4");
        Button button5 = new Button("5");
        Button button6 = new Button("6");
        Button button7 = new Button("7");
        Button button8 = new Button("8");
        Button button9 = new Button("9");
        Button buttonPlus = new Button("+");
        Button buttonMinus = new Button("-");
        Button buttonMultiply = new Button("x");
        Button buttonDivide = new Button("÷");
        Button buttonClear = new Button("C");
        Button buttonEqual = new Button("=");

        // Handle button clicks
        button0.setOnAction(e -> appendDigit(0));
        button1.setOnAction(e -> appendDigit(1));
        button2.setOnAction(e -> appendDigit(2));
        button3.setOnAction(e -> appendDigit(3));
        button4.setOnAction(e -> appendDigit(4));
        button5.setOnAction(e -> appendDigit(5));
```

```
    button6.setOnAction(e -> appendDigit(6));
    button7.setOnAction(e -> appendDigit(7));
    button8.setOnAction(e -> appendDigit(8));
    button9.setOnAction(e -> appendDigit(9));
    buttonPlus.setOnAction(e -> performOperation("+"));
    buttonMinus.setOnAction(e -> performOperation("-"));
    buttonMultiply.setOnAction(e -> performOperation("x"));
    buttonDivide.setOnAction(e -> performOperation("÷"));
    buttonClear.setOnAction(e -> clear());
    buttonEqual.setOnAction(e -> calculate());

    // Create the layout
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.setVgap(5);
    gridPane.setPadding(new Insets(10));

    // Add the buttons to the grid
    gridPane.add(button7, 0, 0);
    gridPane.add(button8, 1, 0);
    gridPane.add(button9, 2, 0);
    gridPane.add(buttonPlus, 3, 0);
    gridPane.add(button4, 0, 1);
    gridPane.add(button5, 1, 1);
    gridPane.add(button6, 2, 1);
    gridPane.add(buttonMinus, 3, 1);
    gridPane.add(button1, 0, 2);
    gridPane.add(button2, 1, 2);
    gridPane.add(button3, 2, 2);
    gridPane.add(buttonMultiply, 3, 2);
    gridPane.add(button0, 0, 3);
    gridPane.add(buttonClear, 1, 3);
    gridPane.add(buttonEqual, 2, 3);
    gridPane.add(buttonDivide, 3, 3);

    // Create the root layout
    VBox root = new VBox(10);
    root.setAlignment(Pos.CENTER);
    root.getChildren().addAll(display, gridPane);

    // Show the stage
    primaryStage.setTitle("Calculator");
    primaryStage.setScene(new Scene(root, 300, 300));
    primaryStage.show();
}

private void appendDigit(int digit) {
    String currentValue = display.getText();
```

```
        if (currentValue.equals("0")) {
            display.setText(String.valueOf(digit));
        } else {
            display.setText(currentValue + digit);
        }
    }

    private void performOperation(String operation) {
        double currentValue = Double.parseDouble(display.getText());
        firstOperand = currentValue;
        this.operation = operation;
        display.setText("0");
    }

    private void clear() {
        display.setText("0");
        firstOperand = 0;
        operation = null;
    }

    private void calculate() {
        double currentValue = Double.parseDouble(display.getText());
        double result = 0;
        if (operation != null) {
            switch (operation) {
                case "+":
                    result = firstOperand + currentValue;
                    break;
                case "-":
                    result = firstOperand - currentValue;
                    break;
                case "x":
                    result = firstOperand * currentValue;
                    break;
                case "÷":
                    result = firstOperand / currentValue;
                    break;
            }
        }
        display.setText(String.valueOf(result));
        operation = null;
    }

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

6. Develop a JavaServer Pages (JSP) application for an online bookstore. Create an index.jsp page that greets users with the bookstore's name and presents a list of available books with details like title, author, and price. Include a search form that enables the user to search for books by title or author, with dynamic handling to display matching results. Ensure the page is well-structured, responsive, and visually appealing using CSS styling. Specifically, use inline styling to set the background color of the form to light grey.

index.jsp

```
<%@ page import="java.util.ArrayList, com.example.Book" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
    <title>Online Bookstore</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 20px;
        }
        h1 {
            color: #333;
        }
        .book {
            border: 1px solid #ccc;
            padding: 10px;
            margin-bottom: 10px;
        }
        .search-form {
            background-color: #f2f2f2;
            padding: 20px;
            display: inline-block;
        }
    </style>
</head>
<body>
    <h1>Welcome to Online Bookstore</h1>

    <div class="search-form">
        <h3>Search Books</h3>
        <form action="index.jsp" method="get">
            <input type="text" name="searchQuery" placeholder="Enter title or author" />
            <input type="submit" value="Search" />
        </form>
    </div>

    <%
        // Sample book data
        ArrayList<Book> books = new ArrayList<>();
```

```
        books.add(new Book("Book 1", "Author 1", 19.99));
        books.add(new Book("Book 2", "Author 2", 24.99));
        books.add(new Book("Book 3", "Author 3", 14.99));
        books.add(new Book("Book 4", "Author 1", 29.99));

        // Search functionality
        String searchQuery = request.getParameter("searchQuery");
        ArrayList<Book> searchResults = new ArrayList<>();

        if (searchQuery != null && !searchQuery.isEmpty()) {
            for (Book book : books) {
                if (book.getTitle().toLowerCase().contains(searchQuery.toLowerCase()) ||
                    book.getAuthor().toLowerCase().contains(searchQuery.toLowerCase())) {
                    searchResults.add(book);
                }
            }
        } else {
            searchResults = books;
        }
    %>

    <h2>Book List</h2>
    <c:if test="${empty searchResults}">
        <p>No books found.</p>
    </c:if>
    <c:if test="${not empty searchResults}">
        <c:forEach var="book" items="${searchResults}">
            <div class="book">
                <h3>${book.title}</h3>
                <p>Author: ${book.author}</p>
                <p>Price: $${book.price}</p>
            </div>
        </c:forEach>
    </c:if>
</body>
</html>
```

Book.java

```java
public class Book {
    private String title;
    private String author;
    private double price;

    public Book(String title, String author, double price) {
        this.title = title;
        this.author = author;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public double getPrice() {
        return price;
    }
}
```

7.Explain the MVC design pattern and its importance in software development. Create a basic JavaFX application for managing a list of contacts using the Model-View-Controller (MVC) architecture.

The Model-View-Controller (MVC) design pattern is a software architectural pattern commonly used in developing user interfaces. It divides an application into three interconnected components:

1. **Model**: Represents the data and the business logic of the application. It encapsulates the data and provides methods to manipulate it. The model notifies the view of any changes in the data.
2. **View**: Represents the presentation layer of the application. It displays the data from the model to the user and captures user input. The view sends user actions to the controller for processing.
3. **Controller**: Acts as an intermediary between the model and the view. It receives user input from the view, interacts with the model to perform any necessary operations, and updates the view with the results.

The importance of the MVC design pattern in software development includes:

- **Separation of Concerns**: MVC separates the different aspects of an application (data, presentation, and user interaction), making the codebase more modular and easier to maintain.
- **Code Reusability**: Since each component (model, view, and controller) is separate, it promotes code reusability. For example, multiple views can interact with the same model without duplicating code.
- **Scalability**: MVC makes it easier to scale an application by allowing developers to make changes to one component without affecting the others. This modular approach simplifies the addition of new features or modifications to existing ones.
- **Testability**: MVC facilitates unit testing and debugging since each component can be tested independently. This ensures better code quality and reduces the risk of introducing bugs.

The MVC pattern promotes loose coupling between the components, making it easier to modify or extend parts of the application without affecting other components. This separation of concerns also facilitates code reuse, testability, and parallel development.

Contact.java

```
public class Contact {
    private String name;
    private String email;

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

    // Getters and setters
    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;
    }
}
```

ContactController.java

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class ContactController {
    private ObservableList<String> contacts;
    private ContactView view;

    public ContactController(ContactView view) {
        this.view = view;
        contacts = FXCollections.observableArrayList();
    }

    public void addContact(String name, String email) {
        Contact contact = new Contact(name, email);
        contacts.add(contact.getName() + " - " + contact.getEmail());
        view.displayContacts(contacts);
    }

    // Other methods for managing contacts (delete, update, etc.)
}
```

ContactView.java

```
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;

public class ContactView extends VBox {
    private ListView<String> contactListView;

    public ContactView() {
        contactListView = new ListView<>();
        getChildren().add(contactListView);
    }

    public void displayContacts(ObservableList<String> contacts) {
        contactListView.setItems(contacts);
    }
}
```

Main.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        ContactView view = new ContactView();
        ContactController controller = new ContactController(view);

        // Simulate adding contacts
        controller.addContact("John Doe", "john@example.com");
        controller.addContact("Jane Smith", "jane@example.com");

        Scene scene = new Scene(view, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Contact Manager");
        primaryStage.show();
    }

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

8. You are tasked with developing a simple distributed application using Remote Method Invocation (RMI) in Java. The application consists of a client-server architecture where the client sends a request to the server to perform a basic mathematical operation, and the server responds with the result. Create an interface named MathOperation containing the method signatures for basic mathematical operations such as addition, subtraction, multiplication, and division. Create a server class that implements the MathOperation interface. Create a client class that connects to the server and invokes the remote methods to perform mathematical operations.

//Interface MathOperation.java

Create an interface named MathOperation:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MathOperation extends Remote {
    int add(int a, int b) throws RemoteException;
    int subtract(int a, int b) throws RemoteException;
    int multiply(int a, int b) throws RemoteException;
    int divide(int a, int b) throws RemoteException;
}
```

Create a server class that implements the MathOperation interface:

MathOperationImpl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MathOperationImpl extends UnicastRemoteObject implements MathOperation {
    protected MathOperationImpl() throws RemoteException {
        super();
    }

    @Override
    public int add(int a, int b) throws RemoteException {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) throws RemoteException {
        return a - b;
    }

    @Override
    public int multiply(int a, int b) throws RemoteException {
        return a * b;
```

```
    }

    @Override
    public int divide(int a, int b) throws RemoteException {
        if (b == 0) {
            throw new RemoteException("Cannot divide by zero");
        }
        return a / b;
    }
}
```

Create a server application to start the RMI registry and bind the MathOperationImpl object:

Server.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;

public class Server {
    public static void main(String[] args) {
        try {
            MathOperation mathOperation = new MathOperationImpl();
            LocateRegistry.createRegistry(1099); // Default RMI registry port
            Naming.rebind("MathOperation", mathOperation);
            System.out.println("Server started");
        } catch (RemoteException e) {
            System.err.println("RemoteException: " + e.getMessage());
        } catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
        }
    }
}
```

Create a client class that connects to the server and invokes remote methods:

Client.java

```
import java.rmi.Naming;
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        try {
            MathOperation mathOperation = (MathOperation) Naming.lookup("//localhost/MathOperation");
            Scanner scanner = new Scanner(System.in);
            System.out.print("Enter first number: ");
            int a = scanner.nextInt();
            System.out.print("Enter second number: ");
            int b = scanner.nextInt();

            System.out.println("Addition result: " + mathOperation.add(a, b));
            System.out.println("Subtraction result: " + mathOperation.subtract(a, b));
            System.out.println("Multiplication result: " + mathOperation.multiply(a, b));
            System.out.println("Division result: " + mathOperation.divide(a, b));
        } catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
        }
    }
}
```

Design a Java program for managing geometric shapes using interfaces and polymorphism. Define an interface named 'Shape' with methods 'calculateArea()' and 'calculate Perimeter()'. Declare two classes, 'Circle' and 'Rectangle', each implementing the 'Shape' interface'. The Circle class should have a constructor that takes the radius as a parameter, while the Rectangle class should take the length and width. Provide appropriate implementations for 'calculate Area() and 'calculate Perimeter()' in both classes based on their respective shapes.

```
interface Shape {
    double calculateArea();
    double calculatePerimeter();
}

class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
```

```
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
}

class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * (length + width);
    }
}

public class ShapeManager {
    public static void main(String[] args) {
        // Creating a Circle object
        Circle circle = new Circle(5);
        System.out.println("Circle Area: " + circle.calculateArea());
        System.out.println("Circle Perimeter: " + circle.calculatePerimeter());

        // Creating a Rectangle object
        Rectangle rectangle = new Rectangle(4, 6);
        System.out.println("Rectangle Area: " + rectangle.calculateArea());
        System.out.println("Rectangle Perimeter: " + rectangle.calculatePerimeter());
    }
}
```

Discuss the fundamentals of the Spring Framework in Java, including major features such as dependency injection (DI), inversion of control (IoC), and aspect-oriented programming.

Develop a basic Spring application in Java to manage tasks. Design a Task class to represent tasks and a TaskService class to interact with tasks. Utilize Spring to configure the application context for managing beans and enable dependency injection. Provide functionality to add tasks and list all tasks.

The Spring Framework is a comprehensive and lightweight framework for building enterprise-level Java applications. It provides various features and modules to simplify the development process and increase productivity. Here are the fundamentals of the Spring Framework:

1. **Dependency Injection (DI)**:
    - Dependency Injection is a design pattern used to remove hard-coded dependencies between objects.
    - In Spring, DI is achieved by providing objects with their dependencies rather than creating them within the object itself.
    - It allows for loose coupling between components, making the code more modular, testable, and maintainable.
    - Spring provides various ways to perform dependency injection, including constructor injection, setter injection, and field injection.
2. **Inversion of Control (IoC)**:
    - Inversion of Control is a principle where the control of object creation and management is shifted from the application to a container or framework.
    - In Spring, IoC is achieved by allowing the Spring container to manage the lifecycle of application objects (beans).
    - The Spring container is responsible for instantiating, configuring, and assembling objects based on the configuration metadata provided (e.g., XML, Java annotations, Java code).
    - IoC simplifies the development process by reducing the amount of boilerplate code needed to instantiate and wire objects together.
3. **Aspect-Oriented Programming (AOP)**:
    - Aspect-Oriented Programming is a programming paradigm that allows developers to modularize cross-cutting concerns such as logging, security, and transaction management.
    - In Spring, AOP is achieved by defining aspects, which are reusable components that encapsulate cross-cutting concerns.
    - Aspects are applied to target objects at runtime to add additional behavior without modifying the code of the target objects.
    - Spring AOP supports both XML-based configuration and annotation-based configuration for defining aspects and pointcuts.

Define the Task class:

```
public class Task {
    private String description;

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

    public String getDescription() {
        return description;
    }
}
```

Define the TaskService interface:

```
import java.util.List;

public interface TaskService {
    void addTask(String description);
    List<Task> getAllTasks();
}
```

Implement the TaskServiceImpl class:

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

public class TaskServiceImpl implements TaskService {
    private List<Task> tasks = new ArrayList<>();

    @Override
    public void addTask(String description) {
        Task task = new Task(description);
        tasks.add(task);
    }

    @Override
    public List<Task> getAllTasks() {
        return tasks;
    }
}
```

Configure Spring for managing beans and enabling dependency injection:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public TaskService taskService() {
        return new TaskServiceImpl();
    }
}
```

Create the main application class:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

        TaskService taskService = context.getBean(TaskService.class);
        taskService.addTask("Task 1");
        taskService.addTask("Task 2");

        System.out.println("All tasks:");
        for (Task task : taskService.getAllTasks()) {
            System.out.println(task.getDescription());
        }

        context.close();
    }
}
```