Final Synthesis

CS5004

Kevin Chen

4/25/2023

Index:

Section 1: Recursion In Practice

Topic Introduction:

The topic of this lab is recursion, which is a programming technique where a function calls itself to solve a problem. Recursion can be used to solve many problems that involve repeating patterns, such as searching through a tree data structure or computing a mathematical sequence. In this lab, we will explore a creative problem that can be solved using recursion.

Assignment Goals:

- Understand the concept of recursion and how it can be used to solve problems
- Design and implement a recursive function to solve a creative problem
- Analyze the time and space complexity of a recursive solution

Assignment Description:

Imagine you are given a list of strings, where each string represents a possible combination of parentheses of varying length. For example, you might have a list like this:

["()", "(())", "()()", "((()))", "(()())", "()(())", "((()))", "()()()"]

Your task is to write a function that returns the length of the longest valid sequence of parentheses in the list. A valid sequence of parentheses is one where each open parenthesis has a corresponding closing parenthesis, and no closing parenthesis is used without a corresponding open parenthesis. For example, "(())" and "()()" are valid sequences, but ")()(" and "((()))" are not.

Starting code:

```
public class Parentheses {
    public static boolean isValidParentheses(String s, int numOpen) {
        // Your code here
    }

    public static int longestValidSequence(String[] strings) {
        // Your code here
        return 0;
    }

    public static void main(String[] args) {
        String[] testStrings = {"()", "(())", "()()", "((()))", "(()())",
"()(())", "((()))", "()()()"};
```

```
        System.out.println(longestValidSequence(testStrings)); // Expected
output: 4
    }
}
```

Key:

```
public class Parentheses {
    public static boolean isValidParentheses(String s, int numOpen) {
        if (s.equals("") && numOpen == 0) {
            return true;
        }
        if (s.equals("") || numOpen < 0) {
            return false;
        }
        if (s.charAt(0) == '(') {
            return isValidParentheses(s.substring(1), numOpen + 1);
        } else if (s.charAt(0) == ')' && numOpen > 0) {
            return isValidParentheses(s.substring(1), numOpen - 1);
        } else {
            return isValidParentheses(s.substring(1), numOpen);
        }
    }

    public static int longestValidSequence(String[] strings) {
        int longest = 0;
        for (String s : strings) {
            int length = 0;
            for (int i = 0; i < s.length(); i++) {
                if (isValidParentheses(s.substring(i), 0)) {
                    length++;
```

```
                }
            }
            if (length > longest) {
                longest = length;
            }
        }
        return longest/2;
    }

    public static void main(String[] args) {
        String[] testStrings = {"()", "(())", "()()", "((()))", "(()())",
"()(())", "(((())))", "()()()"};
        System.out.println(longestValidSequence(testStrings)); // Expected
output: 4
    }
}
```

Section 2: Abstract Classes and Interfaces

Introduction:

This assignment aims to enhance your understanding of Java programming concepts such as abstract classes and interfaces. These concepts will be used to develop a payment processing system that supports different payment methods such as credit card, debit card, and PayPal.

Assignment Goals:

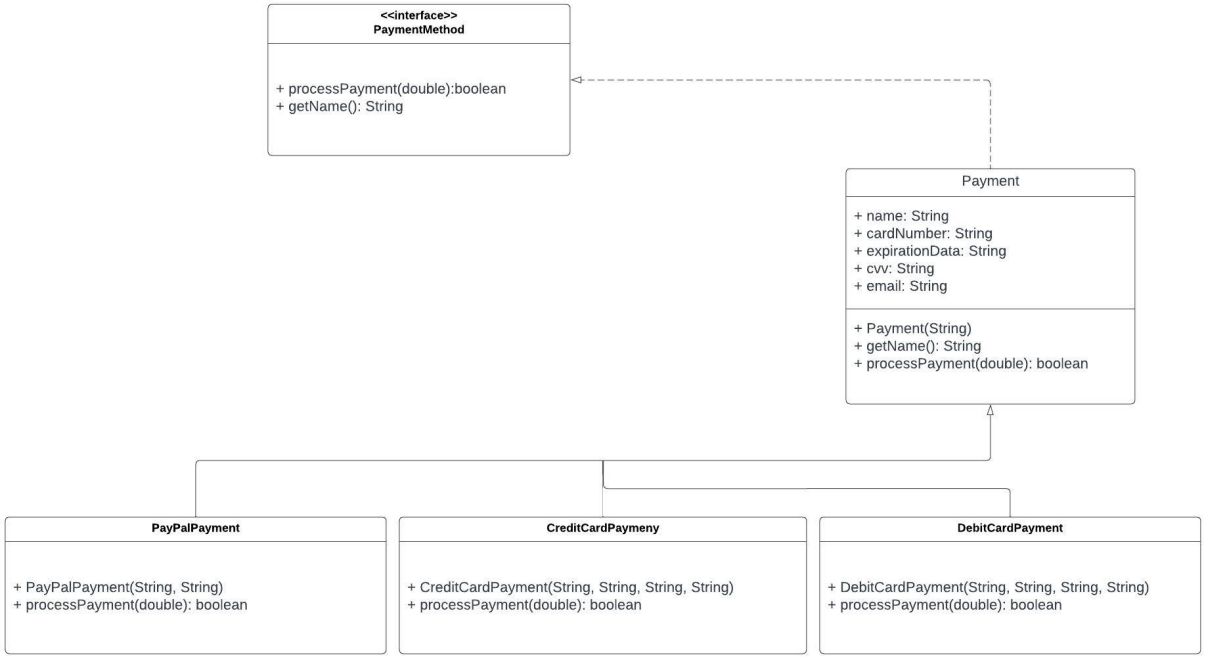
- Develop an understanding of abstract classes and interfaces in Java
- Learn to use abstract classes and interfaces appropriately and for their intended use
- Develop problem-solving and programming skills
- Get a knowledge how to use a UML Diagram for software development.

Assignment Description:

The Payment Processing System is a Java program that allows a user to select a payment method (credit card, debit card, or PayPal), enter payment details, and process the payment. The program uses interfaces, abstract classes, and inheritance to implement the different payment methods and

process the payment. It also includes exception handling to ensure that any errors during payment processing are handled gracefully.

UML Diagram:



Key:

The key code has many files so it is provided Separately in the folder “KeyCodeSection2”.

Section 3: Abstracted Linked Lists

Topic Introduction:

The goal of this assignment is to implement a task management system using an abstracted linked list data structure in Java. Linked lists are a fundamental data structure in computer science that enable the efficient storage and manipulation of collections of items. In this lab, you will learn how to create and manipulate a linked list to manage tasks in a task management system.

Assignment Goals:

- Gain an understanding of linked lists as a data structure.
- Learn how to create and manipulate a linked list in Java.
- Practice object-oriented programming principles by abstracting the linked list into a separate class.
- Implement a task management system using the abstracted linked list.

Assignment Description:

- Create a Task class that has the following attributes
    - String title: the title of the task
    - String description: a brief description of the task
    - boolean completed: a flag indicating whether or not the task has been completed
- Create an Abstracted Linked List that will be used to store Task objects. This class should have the following methods:
    - add(Task t): add a new task to the linked list
    - remove(Task t): remove a task from the linked list
    - get(int index): retrieve a task at a specific index in the linked list
    - Size(): return the number of tasks in the linked list
- Create a TaskManager class that will act as the interface for the task management system. This class should have the following methods:
    - addTask(String title, String description): create a new task with the given title and description, add it to the linked list, and return the task
    - removeTask(Task t): remove the given task from the linked list
    - completeTask(Task t): mark the given task as completed
    - printTasks(): return a list of all tasks in the linked list
    - printCompletedTasks(): return a list of all completed tasks in the linked list
- Test the TaskManager class by creating a main method that adds tasks, completes tasks, and retrieves tasks from the linked list.

Key:

The key code has many files so it is provided Separately in the folder “KeyCodeSection3”.

Section 4: Higher Order Functions – Map, Filter, and Fold

Topic Introduction:

The concept of higher-order functions - specifically map, filter, and fold - is an important tool in functional programming. These functions allow us to write concise and efficient code for transforming and processing data, and are commonly used in a variety of applications. In this coding assignment, you will implement a program that uses all three of these functions to process a dataset of student records and generate a report on their grades.

Assignment Goals:

- Understand the concepts of higher-order functions and how they can be used to process data.
- Practice using map, filter, and fold in a real-world application.
- Develop skills in functional programming and stream processing.

Assignment Description:

You are given a CSV file containing records of students in a school, including their names, grades, and courses taken. Your task is to write a program that reads this file, processes the data using map, filter, and fold, and generates a report on the students' grades.

1. Use map to transform the original list of Student objects into a new list of GradeInfo objects, where each GradeInfo object contains the course name, the average grade for that course, and the number of students who took the course. You can assume that each student takes only one course.
2. Use filter to remove any courses with less than 5 students.
3. Use fold to calculate the total number of students in each grade category (A, B, C, D, F).
4. Generate a report that displays the average grade and number of students for each course (only for courses with at least 5 students), as well as the total number of students in each grade category.

Your output should look something like this.
Course Report: // Only Display result of courses with more than 4 students ---------- English: avg=83.60, count=5 Math: avg=87.20, count=5 Grade Distribution: // show data from all students ---------- A: 4 B: 5 C: 3 D: 1 F: 0

Key:

The key code has many files so it is provided Separately in the folder “KeyCodeSection4”.

Section 5: Hierarchical Data Representation

Topic Introduction:

The topic of this assignment is hierarchical data representation in Java. Hierarchical data structures are commonly used in software development to represent data in a tree-like structure, where each node has one parent and zero or more children. In the context of a company, for example, a hierarchical data structure could be used to represent the relationships between the CEO, managers, supervisors, and employees.

Assignment Goals:

- Understand the concept of hierarchical data representation
- Learn how to implement hierarchical data structures in Java
- Learn how to read data from a CSV file in Java
- Learn how to use inheritance and polymorphism in Java to create a hierarchy of employee classes
- Learn how to traverse a hierarchical data structure in Java and print its contents in a hierarchical manner

Assignment Description:

In this assignment, you will write a Java program to read data from a CSV file and create a hierarchical data structure to represent a company. The CSV file will contain information about the employees in the company, including their names, IDs, roles, and the IDs of their managers.

- Define a set of classes to represent the employees in the company. You should have at least four classes: Employee, Supervisor, Manager, and CEO. Each class should have instance variables to store the employee's name and ID, and any additional information that is specific to that role (e.g., the number of direct reports for a supervisor).
- Use inheritance to create a hierarchy of employee classes, where each class represents a different level in the company hierarchy. For example, the Supervisor class should extend the Employee class, the Manager class should extend the Supervisor class, and the CEO class should extend the Manager class.
- Implement a method in each class to print the employee's name and ID in a formatted string. For example, the Employee class might have a print() method that prints "Employee: {name}, ID: {id}".
- Use polymorphism to create a print() method in the Employee class that can be called on any employee object, regardless of its type. This method should traverse the hierarchical data structure and print each employee's information in a hierarchical manner.
- Implement a method to read data from a CSV file and create the corresponding employee objects.
- Create a main method that uses the methods you have implemented to read the data from the CSV file, create the hierarchical data structure, and print the contents of the data structure in a hierarchical manner.

- By completing this assignment, you will gain hands-on experience with hierarchical data structures, inheritance, polymorphism, and file I/O in Java. You will also learn how to represent and visualize complex data relationships in a structured and hierarchical manner.

Expected Output:

- John (CEO)
  - Jane (Manager)
  - Steve (Supervisor)
  - Lisa (Employee)
  - Mark (Employee)
  - Karen (Supervisor)
  - Ann (Employee)
  - Bob (Manager)
  - Jack (Supervisor)
  - Sam (Employee)
  - Mary (Employee)

Key Code:

The key code has many files so it is provided Separately in the folder “KeyCodeSection5”.

Section 6: MVC Design

Topic Introduction:

The Model-View-Controller (MVC) is a software design pattern commonly used in building user interfaces. It aims to separate an application into three interconnected parts, namely, the Model, the View, and the Controller. The Model represents the data and business logic of the application, the View is responsible for displaying the data to the user and receiving user input, and the Controller connects the Model and View, implementing the application logic and processing user input.

Assignment Goals:

- Understand the concept of the MVC design pattern.

- Learn how to implement the MVC design pattern in software development.
- Build a simple console-based task manager using the MVC design pattern in Java.

Assignment Description:

In this assignment, you will learn how to implement the Model-View-Controller (MVC) design pattern in software development using Java. You will build a simple console-based task manager, where users can view, add, and complete tasks.

To achieve the assignment goals, you will need to follow these steps:

- Model: Implement the Task class that represents the data and business logic of the task manager. It should have attributes such as name, description, and completed.
- View: Implement the TaskView class that displays the list of tasks to the user and receives user input. It should have methods such as showTasks, getTaskName, getTaskDescription, and getTaskIndex.
- Controller: Implement the TaskController class that connects the Model and View, implementing the application logic and processing user input. It should have methods such as showTasks, addTask, completeTask, and run.
- Main: Implement the TaskModel class that stores and manages a list of tasks. Instantiate the Model, View, and Controller classes, and run the application.

By the end of this assignment, you will have a basic understanding of the MVC design pattern and how to implement it in software development using Java. Additionally, you will have gained experience in building a simple console-based application using Java.

Key:

The key code has many files so it is provided Separately in the folder “KeyCodeSection6”.

Section 7: SOLID Principles

S. Single Responsibility Principle (SRP):

This principle states that a class should have only one reason to change. In other words, a class should have only one responsibility or job to do. If a class has multiple responsibilities, changes to one responsibility may unintentionally affect other responsibilities.

This class's sole function is to process orders. It has three private methods, each of which handles a different aspect of the order processing logic. The class is easier to maintain and alter because the many responsibilities are separated into their own methods.

```
public class OrderProcessor {
    public void processOrder(Order order) {
```

```
        validateOrder(order);
        saveOrder(order);
        sendConfirmationEmail(order);
    }

    private void validateOrder(Order order) {
        // validation logic
    }

    private void saveOrder(Order order) {
        // save to database logic
    }

    private void sendConfirmationEmail(Order order) {
        // email sending logic
    }
}
```

O. Open/Closed Principle (OCP):

This principle states that a class should be open for extension but closed for modification. This means that new functionality should be added to a class by creating new classes that inherit from the original class, rather than modifying the original class itself.

In this case, we have a Shape interface that offers a method for determining the area of a shape. Then we have two classes, Circle and Rectangle, that implement the Shape interface and implement the area() method. We might add a new shape, like as a triangle, by creating a new class that implements the Shape interface rather than modifying the current Circle or Rectangle classes.

```
public interface Shape {
```

```
    double area();
}

public class Circle implements Shape {
    private double radius;

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

    public double area() {
        return Math.PI * radius * radius;
    }
}

public class Rectangle implements Shape {
    private double width;
    private double height;

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

    public double area() {
        return width * height;
    }
}
```

L. Liskov Substitution Principle (LSP):

This principle states that any instance of a parent class should be able to be replaced by an instance of a child class without affecting the correctness of the program. In other words, child classes should be able to be used in place of their parent class without causing unexpected behavior.

In this case, We have a Bird class with a fly() function in this example. Because penguins cannot fly, we have a Penguin class that extends Bird but overrides the fly() method to throw an error. The watchBird() method of the BirdWatcher class takes a Bird object and calls its fly() method. This method can be called with either a Bird or a Penguin object, and the behaviour is the same for both.

```
public class Bird {
    public void fly() {
        System.out.println("I'm flying!");
    }
}

public class Penguin extends Bird {
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly!");
    }
}

public class BirdWatcher {
    public void watchBird(Bird bird) {
        bird.fly();
    }
}
```

I. Interface Segregation Principle (ISP)

The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use. No code should be forced to depend on methods it does not use. This means that interfaces should be designed in such a way that they are specific to the needs of their clients.

In this case, each class only implements the methods it needs to, and the interface is designed specifically for the needs of its clients.

```
interface EditableDocument {
    void edit();
}

interface PrintableDocument {
    void print();
}

interface ShareableDocument {
    void share();
}

class WordDocument implements EditableDocument, PrintableDocument,
ShareableDocument {
    void edit() {
        // ...
    }

    void print() {
        // ...
    }
```

```
    void share() {
        // ...
    }
}

class PdfDocument implements PrintableDocument, ShareableDocument {
    void print() {
        // ...
    }

    void share() {
        // ...
    }
}
```

D. Dependency Inversion Principle (DIP)

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. Abstractions should not depend on details, but details should depend on abstractions.

Consider a scenario where we have a user class that needs to send an email when a new account is created. This code allows us to easily switch to a different email sending implementation in the future, without having to modify the User class.

```
interface EmailService {
    void sendEmail(String message);
}
```

```
class User {
    private EmailService emailService;

    User(EmailService emailService) {
        this.emailService = emailService;
    }

    void createAccount() {
// Create account logic
        emailService.sendEmail("New account created.");
    }
}

class EmailSender implements EmailService {
    void sendEmail(String message) {
// ...
    }
```

Section 8: Factory Design Pattern

Topic Introduction:

The Factory Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This pattern is useful when there is a need to create different objects based on a specific set of criteria. The Factory pattern separates the object creation logic from the client code, making the code more modular and easier to maintain.

Assignment Goals:

By completing this assignment, students should be able to:

- Understand the concept of the Factory Design Pattern and its purpose.

- Identify situations where the Factory Design Pattern can be used.
- Implement the Factory Design Pattern in Java.
- Understand the advantages and disadvantages of using the Factory Design Pattern.

Assignment Description:

In this assignment, you will implement the Factory Design Pattern in Java. You will be provided with an interface and some concrete implementations. Your task is to create a factory that will create different types of zombies based on a specific set of criteria. The types of zombies are taken from the game "The Last of Us" from the Naughty Dog developers. The types are Clicker, Runner and Bloater but you can add or remove any zombie type if you want. You will also create a client class that will use the factory to create objects.

Specifically, your tasks are:

- Create the Zombie interface and its concrete implementations of each zombie type.
- Implement the SimpleZombieFactory class that creates different types of zombies based on the type passed to it.
- Create a client class, ZombieStore, that uses the factory to create zombies.
- Test your implementation by creating different types of zombies using the ZombieStore class.

After completing this assignment, you should be able to understand the Factory Design Pattern and how it can be used to create objects based on specific criteria. You should also be able to implement the Factory Design Pattern in Java and recognize its advantages and disadvantages.

Key:

The key code has many files so it is provided Separately in the folder "KeyCodeSection8".