1- Recursion in Practice:

1-1- Topic Introduction: A recursion is a programming concept that calls itself to solve problems. It's great for breaking down complicated tasks into smaller ones. Recursion has two parts:

- **The base case:** In the base case, the problem can be solved directly without using recursion. Basically, it prevents infinite loops by stopping recursion. As soon as the function reaches the base case, it stops calling itself and starts returning values.
- **The recursive case:** A recursive case is one in which a smaller or simpler version of the problem can be used to solve the problem. As a result, the problem is broken down into smaller instances, which are eventually reduced to the base case(s). It is the recursive case that drives the problem-solving process deeper into the recursion.

1-2- Assignment Goals:

- Use recursion in the linked lists and create a meaningful program.
- Discover how to traverse hierarchies and linked lists using recursion.
- Learn how recursion works.

1-3- Assignment Description:

Let's suppose we have a movie store where we would like to create a linked list of movies and perform several tasks, such as adding and removing.

1-4- key:

The code is provided under the name 'Recursion in Practice' in the final project folder. This code was created using a link list. It has four three class linkList class and Movie class and Enum MovieFenres class.

- linkList class: There is a private class node in the link list class (I could define another class, but I decided to limit the number of public classes). The nodes of the linked list are represented by this class. Each node contains a Movie object and a reference to the next node in the list. Next, I declare two object nodes for the first and last node of the link list, plus addLast, addFirst, indexOF, contain methods, and at the end, I create two special recursive methods (size and print methods).

- **sizeHelper(Node current):** A helper method for calculating the size of the linked list. recursively.
- **printHelper(Node current):** A helper method for printing the linked list recursively.

- Movie class: Movie class has properties like release date, rating, title, and genre. It provides methods to create and manipulate movie objects and retrieve information about movies.

- MovieGenres: The MovieGenres enum is used to store the genre of the movie.

2- Abstract Classes and Interfaces:

2-1- Topic Introduction: Abstract classes cannot be instantiated directly, meaning that they cannot be created as objects. The abstract class can be inherited by other classes and implemented by them. Methods can be abstract or non-abstract in abstract classes. An interface consists of abstract methods that any class can implement. Interfaces define a contract that implementing classes must adhere to, ensuring that related classes have the same methods.

2-2- Assignment Goals:

- Learn about Java's abstract classes and interfaces.
- Create and implement abstract classes and interfaces.

2-3- Assignment Description:

Consider a movie store where we would like to create a list of movies with different genres and detect the best movie based on user feedback.

2-4- key:

The code is provided under the name ‘Abstract Classes and Interfaces' in the final project folder.

This code was created using a abstract class and interface. It has these classes:

- **class Action:** This class represents an action movie. Action inherits from Movie and implements the abstract methods (isGood, goodFeedBack) in the IBest interface.
- **class Drama:** This class represents an drama movie. Drama inherits from Movie and implements the abstract methods (isGood, goodFeedBack) in the IBest interface.
- **interface IBest:** The interface provides two abstract methods, and the abstract class Movie implements them. I'm defining this abstract method in the interface instead of the abstract class so if I need it later, I can add another abstract class that implements it.
- **abstract class Movie:** this abstract class has properties like release date, rating, title, and genre. It provides methods to create and manipulate movie objects and retrieve information about movies and it also implements IBest interface.
- **Class MovieCategory:** This class creates a list of movies and has special methods for getAverageRating() and getHighestRated().enum MovieGenres.
- **class ScienceFiction:** This class represents an Science fiction movie. Science fiction inherits from Movie and implements the abstract methods (isGood, goodFeedBack) in the IBest interface.
- **MovieGenres:** The MovieGenres enum is used to store the genre of the movie.

3- Abstracted Linked Lists:

3-1- Topic Introduction: In this part, we'll explore abstracted linked lists, a fundamental data structure in computer science. Abstracted linked lists are linear collections of elements that point to each other. Structures like these are useful because they let you insert and delete elements efficiently, as well as resize them dynamically. We'll use recursion to implement various operations on the linked list, showcasing its versatility.

3-2- Assignment Goals:

- Learn about abstract linked list.
- Create and implement a abstract linked list.

3-3- Assignment Description:

Assume we are creating a movie and theater store where we would like to create a list of movies and theaters based on genre and detect the number of movies and theaters.

3-4- key:

The code is provided under the name ‘Abstracted Linked Lists ' in the final project folder. This code was created using a abstracted linked lists. Every node is read in the linked list (if the first node is a movie or theater, it goes after being read, and if there's no movie or theater node in the next node, the empty node is read). It has these classes:

- **class EmptyNode:** this class indicates the last node in the link list. This class implements interface MovieStoreList<T>.
- **class MovieNode:** this class indicates a node in the link list. This class implements interface MovieStoreList<T>.
- **class TheaterNode:** this class indicates a node in the link list. This class implements interface MovieStoreList<T>.
- **interface MovieStoreList<T>:** This class provides abstract methods in link lists that every node must implement. Since these cod have two enum theater and movie classes, I use generic.
- **class MovieListImpl:** This class implements the abstract link list. It has various methods (addMovie, addTheater, getNumberOfMovies, getNumberOfTHeater and getGenreByFilter)
- **enum TheaterGenres:** Represents a list of theater genres.
- **enum MovieGenres:** Represents a list of theater genres.

4- Higher order functions map, filter, and fold:

4-1- Topic Introduction: Functional programming uses higher-order functions like map, filter, and fold. Data can be transformed and manipulated in a concise and expressive way with these functions.

- **map:** Map transforms each element in a collection by applying a function.
- **filter:** The filter operation selects elements from a collection that satisfy a condition (predicate).
- **fold:** Folding or reducing elements into a single value is done by applying a function repeatedly.

4-2- Assignment Goals:

- Learn how to use map, filter, and fold functions.
- Understand the concept of higher-order functions

4-3- Assignment Description:

Imagine a movie store where we'd like to create a list of movies with special features (finding max rate movie or sort movies by release date) using higher-order functions.

4-4- key:

The code is provided under the name ‘Higher order functions map, filter, and fold ' in the final project folder. This code was created using Higher order functions map, filter, and fold. It has these classes:

- class HigherOrderFunctions: We create a list of movies and stream them using the stream function. After that, we can implement Higher order functions map, filter, and fold in this list. We create some methods:

- **Count():** This method gives us the number of movies in the list. I use the reduce function in this method.
- **printIsGoodMovie():** By using this method, we can identify good movies. In this method, I use the filter and forEach.
- **sortReleaseDate():** This method sorts movies based on their release date. In this method, I use the Comparator and forEach.
- **maxRating():** This method identifies the movie with the highest rating. In this method, I use the Max and Comparator.
- **toList:** This method converts the stream into the list. In this function, I use the collect.
- **getSizeByGenres:** This method gives us the number of movies in the list based on their genres. I use reduce and map in this method.

- Movie class: Movie class has properties like release date, rating, title, and genre. It provides methods to create and manipulate movie objects and retrieve information about movies.

- MovieGenres: The MovieGenres enum is used to store the genre of the movie.

5- Hierarchical Data Representation:

5-1- Topic Introduction: Tree-like hierarchical data representations organize and structure data in a way that lets each item (node) have one parent and multiple children. Using this type of data representation makes sense when there are natural relationships between elements in the data set, and those relationships form a hierarchy. This allows for efficient searching of the data, as well as easy navigation of the data set. It also helps to make the data more understandable, as the relationships between elements are easily visible. An element at the top of a hierarchy is called the root, and it has no parents. A node with children is called an internal node or a branch node. A node without children is called a leaf node. The depth of a node is the distance from the root to that node, and the height of a tree is its maximum depth. All of the leaf nodes together form the set of leaves of the tree. The ancestors of a node are the nodes along the path from the node to the root. The siblings of a node are the nodes that have the same parent.

5-2- Assignment Goals:

- Learn how to Hierarchical Data structure(tree).
- Understand the concept of Hierarchical Data structure(tree).

5-3- Assignment Description:

Imagine a family with a long history. Their goal is to create a family tree program that lets them easily find out how many people are in the family and who's the oldest or youngest.

5-4- key:

The code is provided under the name ‘Hierarchical Data Representation' in the final project folder. This code was created using tree structure. It has these classes:

- class AbstractTreeNode<T>: AbstractTreeNode is an abstract class that implements tree nodes. AbstractTreeNode provides a foundation for implementing concrete tree node classes, like GroupNode and LeafNode. These classes inherit the data field and constructor from AbstractTreeNode, while implementing the specific behavior required by their respective types.

- class GroupNode<T>: Represents a node that can have children within a tree. GroupNodes contain data of type T and a list of children, either GroupNodes or LeafNodes.

- class LeafNode<T>: This is a leaf node in the tree that doesn't have any children. LeafNodes contain only T-type data.

- interface TreeNode <T>: This interface defines a blueprint for tree node classes.

- class Person: It's a class that represents a person with their name, age, and gender.

- interface PersonFamilyInterface: A PersonFamilyImpl class implements this interface.

- class PersonFamilyImpl: This class represents the family tree of a Person object. It implements the PersonFamilyInterface. This class has special methods:

- **getOldestPerson():** Returns the oldest person in the family tree. I use the lambada expression in this method.
- **getYoungestPerson():** Returns the youngest person in the family tree. I use the lambada expression in this method.
- **getAverageAge():** Calculates the average age of all persons. I use the lambada expression in this method.

6- MVC Design:

6-1- Topic Introduction: A Model-View-Controller (MVC) application consists of three main components: Model, View, and Controller. As a result of this separation, code can be better organized, maintained, and reused. Each component is described briefly below:

- **Model:** In the Model, the business logic and data of the application are represented. Data is retrieved and stored, as well as processed as needed. The Model does not specify how the data should be displayed or presented and is independent of the user interface.
- **View:** The View displays data from the Model to the user. User interface changes are reflected in the Model after updates are received.
- **Controller:** Controllers serve as intermediaries between Models and Views. Upon receiving user input from the View, it processes it and updates the Model accordingly. The Controller ensures synchronization between the Model and View by managing the flow of data.

6-2- Assignment Goals:

- Learn how to MVC.
- Understand the concept of MVC,

6-3- Assignment Description:

We can imagine a movie store where we can create a list of movies using the MVC design, and if we want to add or remove a movie, we type its name and genre.

6-4- key:

The code is provided under the name ‘MVC Design' in the final project folder. This code was created using MVC design. It has these classes:

- class Movie: Movie class has properties like release title, and genre. It provides methods to create and manipulate movie objects and retrieve information about movies.

- class MovieStore: A class representing a movie store that maintains a list of movies. It provides methods to add, remove, and search for movies based on their titles and genres. this class has special methods:

- **searchMoviesByTitle(String title):** Searches for movies in the store by their title and prints them.
- **searchMoviesByGenre(String genres):** Searches for movies in the store by their genre and prints them

- class MovieStoreView: this class is responsible for viewing. The view displays data from the Model to the user. This class handles user interactions, such as reading input, displaying movies, and presenting menu options.

- class MovieStoreViewController: this class representing the controller for a movie store application. This class connects the MovieStore and the MovieStoreView, allowing them to interact with each other. we have some private methodsin this classs because I want to handle better user input, and the reason they are private is because they do not need to be accessed outside of the class.

7- SOLID Principles:

7-1- Topic Introduction:

SOLID is an acronym that stands for five design principles in object-oriented programming that aim to make software systems more maintainable and scalable.

- **Single Responsibility Principle (SRP):** Classes should have only one reason to change, so they should only have one responsibility or function.
- **Open/Closed Principle (OCP):** Classes should be extensible but closed to modification, so we can add new functionality without changing the existing code.
- **Liskov Substitution Principle (LSP):** we should be able to replace objects of a superclass with objects of its subclasses without affecting the program's correctness.
- **Interface Segregation Principle (ISP):** Clients shouldn't be forced to use methods they don't use, so interfaces should be tailored to their needs.
- **Dependency Inversion Principle (DIP):** A high-level module shouldn't depend on a low-level module; both should be abstract. Details shouldn't depend on abstractions; abstractions should depend on details.

7-2- Assignment Goals:

- Learn how to SOLID principles.
- Understand the concept of SOLID principles,

7-3- Assignment Description:

Explain Solid principles and give examples for each part.

7-4- key:

The code is provided under the name ‘SOLID Principles' in the final project folder.

1- Single Responsibility Principle (SRP): SRP says a class should only have one reason to change, which means it should only be responsible for one thing. This example is about rectangles and circle. Each class calculates its own area. To change the area calculation for a specific shape, we only need to change the class corresponding to that shape. As a result, the code is more maintainable and scalable.

2- Open/Closed Principle (OCP):

- **Open for extension:** Classes can be extended to incorporate new functionality or adapt to changing needs.
- **Closed for modification:** Once a class is developed and tested, the source code shouldn't be changed. Changing an existing class can introduce new bugs and break existing functionality.

In this example, we created the Shape interface for the rectangle and circle classes, as well as another class (Calculator class) to take advantage of the Shape interface as composition. By doing this, if we want to add a new shape to calculate the area, we can just create a new class that implements Shape.

3- Liskov Substitution Principle (LSP):

Objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program. We create an abstract shape class with an abstract area() method and we create a new class Circle that extends the Shape class, implementing the area() method. Then we create the Rectangle class to extend the Shape class directly, implementing the area() method. when we substitute a Rectangle object and a Circle object for a Shape object in the main class, the program's correctness is maintained, and This adheres to the Liskov Substitution Principle, making our code more robust and maintainable.

4- Dependency Inversion Principle (DIP):

Suppose we use one of the classes (class and rectangle) instead of the shape interface in the Calculator class so it directly depends on the low-level Rectangle and Circle classes. In order to add more shapes, we have to modify the Calculator class, which is not ideal.

In this example, we introduced the Shape interface, which separates the Calculator class from the concrete shapes. Rectangle and Circle implement this interface, and Calculator relies on the abstraction instead of the concrete classes. By doing this, the code becomes more modular and easy to extend with new shapes.

8- Any Design Pattern:

8-1- Topic Introduction:

Observer Patterns define a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated. Often used when a state change in one object needs to be reflected in other objects without knowing exactly how many or what types they are.

- **Subject:** The main object that holds the state and notifies observers when it changes. It keeps track of observers and lets you add, remove, and notify them.
- **Observer:** This is the interface for the update method, which all observers have to implement.
- **Concrete Observer:** These objects implement the Observer interface. When they're notified of a state change, they update the reference to the subject.

8-2- Assignment Goals:

- Learn how to Observer Pattern.
- Understand the concept of Observer Pattern.

8-3- Assignment Description:

Imagine that we want to create a program for a movie shop that would alert users when a new movie is released.

8-4- key:

The code is provided under the name ‘Any Design Pattern' in the final project folder. This code has these classes:

- Class Movie: This is the main object that holds the state and notifies observers when it changes. A list of observers is maintained, and methods are provided for adding, removing, and notifying them.

-Interface MovieStoreSubject: This interface defines the addUserObserver, removeUserObserver, and notifyUserObservers methods, which all concrete subject need to implement.

- Class User: A User class that represents a user who is also an observer in the Observer design pattern. Implements the UserObserver interface and receives notifications about new movie releases from the MovieStore class.

- Interface UserObserver: This interface defines the update method, which all concrete observers need to implement.