**WEEK-1**

**Software Development Life Cycle (SDLC)** is a cost-effective and time-efficient process that development teams use to design and build high quality software.

**SDLC PHASES**

**1. Requirement:** This first stage entails establishing the goals, resources, and scope of the project. To guarantee alignment with corporate objectives, it consists of risk assessments, project scheduling, and feasibility studies. Stakeholders provide detailed requirements. Analysts create a thorough requirements specification that directs the design phase by documenting both functional and non-functional needs.

**2. Design:** The criteria serve as the foundation for the architecture and design of the system. Creating design documentation, such as database schemas, data flow diagrams, and system models, is the task of this phase.

**3. Implementation:** At this point, actual coding starts. Developers use best practices and coding standards when writing code by following the design specifications.

**4. Verification:** In order to find and address flaws, the software is rigorously tested at this phase. To make sure the program satisfies the specified criteria, a variety of testing techniques are used, including unit testing, integration testing, and system testing. The program is deployed to the production environment after testing is finished and it has been validated. Installation, configuration, and user training are all included in this step.

**5. Maintenance:** The software goes through a maintenance period after deployment, during which it is patched and upgraded to address changes, resolve bugs, and gradually increase performance.

**SDLC Models:**

**1. Waterfall Model:** The Waterfall model is a sequential, linear method to software development where each step has to be finished before moving on to the next. Requirements, Design, Implementation, Testing, Deployment, and Maintenance are the phases. It is simple and easy to administer, but it is not flexible because it is hard to make modifications once a phase is finished. Projects with clearly specified requirements and little chance of changes during development are best suited for this strategy.

**2.Iterative Model:** With the iterative methodology, software development is approached by breaking the project up into small, manageable cycles or iterations. Phases of planning, design, development, and testing are included in every iteration. The product is reviewed and improved

upon depending on user feedback following each iteration. This paradigm works well for projects whose needs change over time or where regular feedback is required since it permits modifications and enhancements at any point during the development process. It improves adaptation and flexibility over linear models.

**3. Prototype Model:** In order to visualize and test functionality, a preliminary version of the software, or a prototype, is created. Before final development, this early model aids in gathering user feedback and streamlining needs. Up to the creation of the final edition, iterative prototypes are constructed and altered in response to user feedback. It's helpful for projects with ambiguous requirements, but if not handled appropriately, it can be time-consuming and result in scope

creep.

**4. Spiral Model:** The Spiral approach emphasizes risk assessment while including iterative development. It entails recurrent planning, risk analysis, engineering, and assessment cycles, or spirals. Every spiral aims to handle potential hazards and develop the project gradually by producing a revised prototype that takes user feedback into account. This paradigm offers flexibility and continuous risk management throughout the development process, making it appropriate for intricate projects with changing requirements.

**5. V- Model:** The Validation and Verification Model, or V-Model, is a software development methodology in which every development phase directly correlates to a testing phase. The steps in the procedure are as follows: System design follows requirements analysis, and detailed design follows system design and implementation. System testing, Integration testing, and Acceptance testing are the testing phases that correlate to each design step. This strategy places a strong emphasis on meticulous validation and verification, making sure that every phase of development is well-tested.

**Agile**

Agile methodology adopts an iterative, collaborative approach.

• Delivering the application's functional and comprehensive components as quickly as feasible is its main objective.

• In software development, "agile" refers to the ability to adjust to changes. • Iterative and progressive in its approach.

• The development team has direct client interaction.

• Every iteration lasts between one and three weeks.

• Offers many software upgrades.

• Engineering actions are carried out by cross-functional teams.

**Agile development life cycle:**

• Concept

• Inception

• Design, Development, Construction, Testing, and Integration

• Implementation Deployment

• Retirement

**Agile methodology has five phases**

1. Project initiation

2. Sprint planning

3. Development

4. Production

5. Retirement

**Scrum**

Scrum is an Agile subset. It is the most used process framework for agile development and is also the lightest.

Scrum is most frequently used to manage the incremental and iterative processes involved in complex software and product development.

Scrum techniques facilitate seamless adaptation of organizations to dynamic requirements and the creation of products that align with changing corporate objectives.

**Scrum process**

An Agile framework for managing and carrying out complex projects is the Scrum approach. Work is arranged into Sprints, which are iterative cycles that run between two and four weeks. The Product Owner establishes and prioritizes the product backlog, the Scrum Master stream lines the Scrum process and eliminates roadblocks, and the Development Team constructs the product are the three main players in the process. Important rituals include the Sprint Planning (establishing objectives and assignments for the Sprint), the Daily Standups (brief meetings every day to review progress), the Sprint Review (showcasing finished work), and the Sprint Retrospective (evaluating and refining the process).

**Scrum phases**

**Initiate**

1. **Create Project Vision:** Define the high-level objectives and goals of the project, ensuring alignment with business needs and stakeholder expectations.

2. **Identify Scrum Master and Stakeholder(s):** Appoint the Scrum Master to facilitate the process and engage relevant stakeholders who will provide input and feedback. 3. **Form Scrum Team:** Assemble a cross-functional team responsible for delivering the project increments. The team should have all necessary skills to complete the work. 4. **Develop Epic(s):** Create high-level, broad user stories (Epics) that outline major features or functionalities of the product.

5. **Create Prioritized Product Backlog:** List all desired features, enhancements, and fixes in the Product Backlog, ordered by priority.

6. **Conduct Release Planning:** Plan the release schedule, including target dates and milestones, based on the prioritized Product Backlog.

**Plan and Estimate**

7. **Create User Stories:** Break down Epics into smaller, detailed user stories that describe specific features or functionality from the user's perspective.

8. **Approve, Estimate, and Commit User Stories:** Validate user stories with stakeholders, estimate effort required, and commit to delivering selected stories in the upcoming Sprint.

9. **Create Tasks:** Decompose user stories into actionable tasks that can be completed during the Sprint.

10. **Estimate Tasks:** Assess the effort required for each task to ensure proper planning and workload distribution.

11. **Create Sprint Backlog:** Compile the user stories and tasks to be worked on during the Sprint into the Sprint Backlog.

**Implement**

12. **Create Deliverables:** Develop the product increment by completing tasks and user stories defined in the Sprint Backlog.

13. **Conduct Daily Standup:** Hold brief daily meetings to discuss progress, plan for the day, and address any impediments.

14. **Groom Prioritized Product Backlog:** Regularly review and adjust the Product Backlog to reflect new insights, changes in priorities, or feedback.

**Review and Retrospect**

15. **Convene Scrum of Scrums:** If needed, coordinate with other Scrum teams to ensure alignment and integration across multiple teams.

16. **Demonstrate and Validate Sprint:** Showcase the completed work to stakeholders for feedback and validation against the Sprint Goal.

17. **Retrospect Sprint:** Reflect on the Sprint process, discussing what worked well and what could be improved for future Sprints.

**Release**

18. **Ship Deliverables:** Deploy the completed product increment to the production environment or deliver it to the end-users.

19. **Retrospect Project:** Conduct a final review of the entire project, analyzing successes, challenges, and lessons learned for future projects.

—---------------------------------------------------------------------------------------------------------------

Git is a source control system, also known as a distributed version control system

(VCS). An application called a version control system (VCS) records all of the

modifications made to your code. It also enables us to work together and share our

code with others without erasing each other's modifications. We will walk you

through the basics of Git and show you how to save your code to a repository in

this course.

The different types of version control systems: local, centralized, and distributed.

1. Local Version Control

The most basic type of version control is called local version control. It entails

monitoring file modifications made on a single computer. Usually, this

technique makes use of a straightforward versioning system or database to

maintain file versions and enable users to roll back to earlier versions as

needed.

How it Works: A local database is used by the user to keep track of file

versions. Every modification is documented as a new version of the file.

-Revision Control System (RCS) One of the first systems for version control. It

keeps a repository of revisions to track changes made to specific files.

Limitations:

Only able to use one machine,

Making teamwork difficult,

No built-in systems to manage conflicts arising from multiple users, branch,

or merge..

2. Centralized Version Control (CVCS)

All users have access to a single central repository, which is managed using

centralized version control systems. Although each user has a functional copy

of the files, version control actions are carried out through interaction with the

central repository.

How it works: Users pull files from the central repository, edit them locally,

and then push their modifications back to the central repository. The project's

history is preserved and all changes are tracked by the central repository.

Advantages:

Simpler to administer in the case of a single central repository.

Ideal for settings where having reliable access to a single source of truth is

essential.

Limitations:

In order to communicate with the central repository, users must be online.

Network delay may have an effect on performance.

One potential single point of failure is the central repository.

3. Distributed Version Control (DVCS)

With distributed version control systems, every user can access a complete copy of

the repository, complete with all of its history. This method improves teamwork

and allows for more flexible workflows.

How it Works: Every user gets access to the whole local repository, which contains

the project's entire history. Users can work offline, push or pull changes to and

from other repositories (including a central repository, if one is utilized), and work

offline while committing changes locally. Advanced functionality like conflict

resolution, branching, and merging are supported by the system.

Advantages:

Users are able to access the entire history and feature set even when working

offline.

Branching and merging are made easier, which makes it ideal for teamwork

processes.

Limitations:

More difficult to configure and maintain than centralized systems.

Can be challenging to understand for users unfamiliar with distributed

workflows.

Some of the basic Git operations are:

1. Configuration

Set git username:

git config --global user.name “Your Name”

Set git email:

git config –global user.email “ Your email”

2. Repository Setup

Initializing a new git repository:

git init

Clone an existing repository:

git clone “repository url”

3. Add

What It Does: Stages changes for the next commit. When you modify files in

your working directory, you use git add to mark these changes as ready to be

included in the next commit.

Usage: git add <file> or git add . (to add all changes).

Purpose: To prepare changes in your working directory for committing.

4. Commit

What It Does: Records the staged changes into the local repository with a

descriptive message. Each commit represents a snapshot of your project at a

specific point in time.

Usage: git commit -m "Commit message".

Purpose: To save and document changes to your local repository.

5. Pull

What It Does: Fetches and integrates changes from a remote repository into

your current branch. This operation updates your local branch with changes

from the remote branch.

Usage: git pull.

Purpose: To synchronize your local branch with updates from the remote

repository.

6. Push

What It Does: Uploads your local commits to a remote repository. This

operation makes your changes available to other collaborators.

Usage: git push`.

Purpose: To share your local commits with others by updating the remote

repository.

7. Branching

What It Does: Creates a separate line of development. Branches allow you to

work on different features or fixes independently of the main project.

Usage:

- To create a branch: git branch <branch-name>.

- To switch to a branch: git checkout <branch-name>.

- To create and switch in one command: git checkout -b <branch-name>.

Purpose: To manage and isolate different lines of development, making it easier

to work on features or fixes independently.

8. Checkout:

Purpose: Switch branches or restore working tree files.

Usage:

- To switch branches: git checkout branch-name

- To restore a file: git checkout -- file-name

- Description: When used to switch branches, this command updates the

working directory to reflect the state of the branch. When used with a file, it

restores the file to its state from the last commit, discarding any local changes.

9. Status:

Purpose: Show the status of changes in the working directory and staging area.

Usage: git status

Description: This command provides a summary of the current state of the

working directory and staging area, including which files are modified, staged for

commit, or untracked.

10. Rebase:

Purpose: Reapply commits on top of another base tip.

Usage: git rebase branch-name

Description: This command takes the commits from your current branch and re-

applies them on top of the specified branch. It is often used to integrate changes

from one branch into another or to keep a feature branch up-to-date with the main

branch.

11. Push:

Purpose: Upload local repository changes to a remote repository.

Usage: git push origin branch-name

Description: This command sends your local commits to the remote repository,

updating the branch on the remote with your changes. The origin refers to the

remote repository name (often the default name for the primary remote repository).

12. Diff:

Purpose: Show differences between commits, working directory, and staging

area.

Usage:

- To see changes not staged for commit: git diff

- To see changes staged for commit: git diff --cached

- To compare commits: git diff commit1 commit2

Description: This command shows the differences between various states of the

repository, such as between the working directory and the staging area or between

commits.

13. Log:

Purpose: Show the commit history.

Usage: git log

Description: This command displays a list of commits in the current branch,

including commit hashes, author information, dates, and commit messages. It helps

you see the history of changes in the repository.

14. Remote Repository

To add remote repository:

git remote add origin “repository url”

To check remote repositories url:

git remote -v

15. Stash

Usage:

Basic Stash: git stash

Stash with Message: git stash push -m "Your message"

List Stashes: git stash list

Apply Stash: git stash apply

Apply Specific Stash: git stash apply stash@{index}

Drop Stash\*\*: git stash drop stash@{index}

Clear All Stashes: git stash clear

16. Undoing changes

To unstage a file:

git reset “file”

17. Deleting branches

To delete a local branch:

git branch –d “branch name”

—-----------------------------------------------------------------------------

Java is a versatile, object-oriented programming language known for its

portability and ease of use. It allows developers to write code once and run it

anywhere. Java is widely used for building web applications, mobile apps, and

large systems, making it a popular choice among programmers.

JDK (Java Development Kit)

Purpose: A complete package for developing Java applications.

Components: Includes the Java compiler (java c), libraries, tools (like debuggers),

and documentation.

Usage: Developers use the JDK to write, compile, and package Java applications.

JDK For development.

JVM (Java Virtual Machine)

Purpose: An abstract machine that executes Java byte code.

Function: It provides a runtime environment, handling memory management,

garbage collection, and system calls.

Portability: Allows Java programs to run on any platform that has a compatible

JVM, enabling the "write once, run anywhere" capability.

JVM For execution of byte code.

JRE (Java Runtime Environment)

Purpose: Provides the necessary environment to run Java applications.

Components: Includes the JVM and core libraries but lacks development tools like

compilers.

Usage: Users who want to run Java applications need the JRE, but not the full

JDK.

JRE For running applications.

PROGRAMS

1) To print sum of numbers and print Hello World

public class HelloWorld{

public static void main(String[] args){

int a = 10;

int b = 20;

int c = a + b;

System.out.println(“The sum of “ + a + “ and “ + b + “ is “ + c);

Systemout.println(“Hello World”);

}

}

Output:

The sum of a + b is 30

Hello World

2) Employees program

package main.java;

// Class representing an employee

public class Employee {

// Instance variables

public String fullName; // Employee's name

public int ageYears; // Employee's age in years

public String residenceCity; // City where the employee lives

public double annualSalary; // Employee's annual salary

public String employerName; // Name of the company the employee works

for

public String id; // Unique employee identifier

public String departmentName; // Department the employee belongs to

public String jobTitle; // Employee's job title

// Method to print employee details

public void printDetails() {

System.out.println("Name: " + fullName);

System.out.println("Age: " + ageYears);

System.out.println("City: " + residenceCity);

System.out.println("Salary: " + annualSalary);

System.out.println("Company Name: " + employerName);

System.out.println("Employee ID: " + id);

System.out.println("Department: " + departmentName);

System.out.println("Designation: " + jobTitle);

}

// Method to set employee details

public void setDetails(String fullName, int ageYears, String residenceCity,

double annualSalary,

String employerName, String id, String departmentName,

String jobTitle) {

this.fullName = fullName;

this.ageYears = ageYears;

this.residenceCity = residenceCity;

this.annualSalary = annualSalary;

this.employerName = employerName;

this.id = id;

this.departmentName = departmentName;

this.jobTitle = jobTitle;

}

// Default constructor

public Employee() {

fullName = "John Doe"; // Default name

ageYears = 30; // Default age

residenceCity = "New York"; // Default city

annualSalary = 100000; // Default salary

employerName = "ABC Company"; // Default company name

id = "1234567890"; // Default employee ID

departmentName = "IT"; // Default department

jobTitle = "Software Engineer"; // Default designation

printDetails(); // Print default details

}

// Parameterized constructor

public Employee(String fullName, int ageYears, String residenceCity, double

annualSalary,

String employerName, String id, String departmentName, String

jobTitle) {

this.fullName = fullName;

this.ageYears = ageYears;

this.residenceCity = residenceCity;

this.annualSalary = annualSalary;

this.employerName = employerName;

this.id = id;

this.departmentName = departmentName;

this.jobTitle = jobTitle;

printDetails(); // Print provided details

}

// Main method to run the program

public static void main(String[] args) {

Employee employee1 = new Employee(); // Create an instance with default

values

employee1.setDetails("Jasa", 30, "New York", 100000, "ABC Company",

"1234567890", "IT", "Software Engineer");

employee1.printDetails(); // Print updated details

Employee employee2 = new Employee("Jasa", 30, "New York", 100000,

"ABC Company", "1234567890", "IT", "Software Engineer");

employee2.printDetails(); // Print details of the second employee

}

}

Output:

Name: John Doe

Age: 30

City: New York

Salary: 100000.0

Company Name: ABC Company

Employee ID: 1234567890

Department: IT

Designation: Software Engineer

Name: Jasa

Age: 30

City: New York

Salary: 100000.0

Company Name: ABC Company

Employee ID: 1234567890

Department: IT

Designation: Software Engineer

Name: Jasa

Age: 30

City: New York

Salary: 100000.0

Company Name: ABC Company

Employee ID: 1234567890

Department: IT

Designation: Software Engineer

3) To demonstrate static keyword concept

package main.java;

// Code to demonstrate the static keyword concept in Java

public class StaticExample {

// Static variable to keep track of the number of instances created

private static int instanceCount = 0;

// Instance variable for each object's instance number

private int currentInstanceNumber;

// Constructor

public StaticExample() {

instanceCount++; // Increment the static count whenever a new object

is created

currentInstanceNumber = instanceCount; // Assign the current instance

number

}

// Method to display instance details

public void display() {

System.out.println("Instance Number: " + currentInstanceNumber +

"\n" + "Count: " + instanceCount);

}

// Main method to run the program

public static void main(String[] args) {

StaticExample firstObject = new StaticExample(); // Create the first

instance

firstObject.display(); // Display details of the first instance

StaticExample secondObject = new StaticExample(); // Create the

second instance

secondObject.display(); // Display details of the second instance

StaticExample thirdObject = new StaticExample(); // Create the third

instance

thirdObject.display(); // Display details of the third instance

}

}

Output:

Instance Number: 1

Count: 1

Instance Number: 2

Count: 2

Instance Number: 3

Count: 3

4) Program for typecasting demo

public class WideningTypeCastingDemo {

public static void main(String[] args) {

// Starting with a byte value

byte initialByteValue = 10; // Declare a byte variable

System.out.println("Original byte value: " + initialByteValue); // Print

the original byte value

// Widening from byte to short

short convertedShortValue = initialByteValue; // Implicit widening

from byte to short

System.out.println("byte to short: " + convertedShortValue); // Print the

converted short value

// Widening from short to char

// Note: Char is unsigned, so we use a positive short value

short positiveShortValue = 65; // ASCII value for 'A'

char convertedCharValue = (char) positiveShortValue; // Explicit cast

to char

System.out.println("short to char: " + convertedCharValue); // Print the

converted char value

// Widening from char to int

int intFromCharValue = convertedCharValue; // Implicit widening

from char to int

System.out.println("char to int: " + intFromCharValue); // Print the int

value from char

// Widening from int to long

long longValueFromInt = intFromCharValue; // Implicit widening from

int to long

System.out.println("int to long: " + longValueFromInt); // Print the long

value

// Widening from long to float

float floatValueFromLong = longValueFromInt; // Implicit widening

from long to float

System.out.println("long to float: " + floatValueFromLong); // Print the

float value

// Widening from float to double

double doubleValueFromFloat = floatValueFromLong; // Implicit

widening from float to double

System.out.println("float to double: " + doubleValueFromFloat);

// Print the double value

// Demonstrating multiple steps of widening in one assignment

int largeIntValue = 1234567; // Declare an integer

double doubleValueFromInt = largeIntValue; // Implicit widening from

int to double

System.out.println("int directly to double: " + doubleValueFromInt);

// Print the double value

// Demonstrating widening in expressions

byte smallByteValue = 10; // Declare a small byte

short mediumShortValue = 100; // Declare a medium short

int sumResult = smallByteValue + mediumShortValue; // Both are

promoted to int during addition

System.out.println("Result of byte + short (as int): " + sumResult); //

Print the result

// Widening with literals

int largeIntLiteral = 2\_000\_000\_000; // Declare a large integer literal (2

billion)

long veryLargeValue = largeIntLiteral \* 3L; // Multiplication results in

long due to the literal '3L'

System.out.println("Large calculation result (as long): " +

veryLargeValue); // Print the long result

// Demonstrating potential loss of precision

long veryPreciseValue = 123456789123456789L; // Declare a long

value

float lessPreciseValue = veryPreciseValue; // Implicit narrowing from

long to float (potential precision loss)

System.out.println("Long to float (potential precision loss): " +

lessPreciseValue); // Print the float value

}

}

Output:

Original byte value: 10

byte to short: 10

short to char: A

char to int: 65

int to long: 65

long to float: 65.0

float to double: 65.0

int directly to double: 1234567.0

Result of byte + short (as int): 110

Large calculation result (as long): 6000000000

Long to float (potential precision loss): 1.2345679E17

5) Programs for logical operators demo

public class ComprehensiveLogicalOperatorsDemo {

public static void main(String[] args) {

// Basic boolean variables

boolean isTrue = true; // Represents true condition

boolean isFalse = false; // Represents false condition

System.out.println("1. Basic Logical Operations:");

System.out.println(" AND: true && true = " + (isTrue && isTrue));

// AND operation

System.out.println(" AND: true && false = " + (isTrue && isFalse));

// AND operation

System.out.println(" AND: false && true = " + (isFalse && isTrue));

// AND operation

System.out.println(" AND: false && false = " + (isFalse && isFalse));

// AND operation

System.out.println(" OR: true || true = " + (isTrue || isTrue));

// OR operation

System.out.println(" OR: true || false = " + (isTrue || isFalse));

// OR operation

System.out.println(" OR: false || true = " + (isFalse || isTrue));

// OR operation

System.out.println(" OR: false || false = " + (isFalse || isFalse));

// OR operation

System.out.println(" NOT: !true = " + (!isTrue)); // NOT operation

System.out.println(" NOT: !false = " + (!isFalse)); // NOT operation

System.out.println("\n2. Short-circuit Evaluation:");

System.out.println(" false && (1/0 > 0) = " + (isFalse && (1 / 0 >

0))); // No exception

System.out.println(" true || (1/0 > 0) = " + (isTrue || (1 / 0 > 0))); // No

exception

System.out.println("\n3. Operator Precedence:");

System.out.println(" true || false && false = " + (isTrue || isFalse &&

isFalse)); // && has higher precedence

System.out.println(" (true || false) && false = " + ((isTrue || isFalse)

&& isFalse)); // Parentheses change precedence

System.out.println("\n4. Combining with Comparison Operators:");

int num1 = 5, num2 = 10; // Integer variables

System.out.println(" (num1 < num2) && (num2 > 0) = " + ((num1 <

num2) && (num2 > 0))); // Comparison

System.out.println(" (num1 > num2) || (num2 < 20) = " + ((num1 >

num2) || (num2 < 20))); // Comparison

System.out.println("\n5. Complex Conditions:");

boolean conditionA = true, conditionB = false, conditionC = true;

// Boolean variables

System.out.println(" (conditionA && conditionB) || (conditionA &&

conditionC) = " + ((conditionA && conditionB) || (conditionA &&

conditionC))); // Complex condition

System.out.println(" conditionA && (conditionB || conditionC) = " +

(conditionA && (conditionB || conditionC))); // Complex condition

System.out.println(" !conditionA || (conditionB && !conditionC) = " +

(!conditionA || (conditionB && !conditionC))); // Complex condition

System.out.println("\n6. Bitwise vs. Logical Operators:");

System.out.println(" true & false = " + (isTrue & isFalse));

// Bitwise AND

System.out.println(" true | false = " + (isTrue | isFalse));

// Bitwise OR

System.out.println(" true ^ false = " + (isTrue ^ isFalse));

// Bitwise XOR

System.out.println("\n7. Short-circuit vs. Non-short-circuit:");

int counter = 0; // Counter variable

boolean shortCircuitResult = (isFalse && (++counter > 0)); // i is not

incremented

boolean nonShortCircuitResult = (isFalse & (++counter > 0)); // i is

incremented

System.out.println(" Short-circuit AND result: " + shortCircuitResult

+ ", counter = " + counter); // Print results

System.out.println(" Non-short-circuit AND result: " +

nonShortCircuitResult + ", counter = " + counter); // Print results

System.out.println("\n8. Logical Operators with Non-boolean

Operands:");

System.out.println(" (1 < 2) && (3 < 4) = " + ((1 < 2) && (3 < 4)));

// Logical comparison

System.out.println(" ('a' < 'b') || ('c' > 'd') = " + (('a' < 'b') || ('c' > 'd')));

// Logical comparison

System.out.println("\n9. Logical Operators in Control Structures:");

if (isTrue && !isFalse) {

System.out.println(" This will be printed."); // Control structure

demonstration

}

int loopCounter = 0; // Loop counter

while (loopCounter < 3 && isTrue) { // Loop with condition

System.out.println(" loopCounter = " + loopCounter); // Print loop

counter

loopCounter++; // Increment loop counter

}

System.out.println("\n10. Logical Operators with Method Calls:");

System.out.println(" isPositive(5) && isEven(4) = " + (isPositive(5)

&& isEven(4))); // Method call with logical operator

System.out.println(" isPositive(-3) || isEven(7) = " + (isPositive(-3) ||

isEven(7))); // Method call with logical operator

System.out.println("\n11. Logical Operators with Null Checks:");

String testString = null; // String variable

System.out.println(" (testString != null) && (testString.length() > 0) =

" + ((testString != null) && (testString.length() > 0))); // Safe null check

// Uncommenting the next line would throw a NullPointerException

// System.out.println(" (testString.length() > 0) && (testString != null)

= " + ((testString.length() > 0) && (testString != null)));

System.out.println("\n12. Using Logical Operators for Conditional

Assignment:");

int assignedValue = isTrue ? 1 : 0; // Conditional assignment using

ternary operator

System.out.println(" assignedValue = " + assignedValue); // Print

assigned value

System.out.println("\n13. Logical Operators in Lambda Expressions:");

java.util.function.Predicate<Integer> isPositiveAndEven = n -> n > 0

&& n % 2 == 0; // Lambda expression

System.out.println(" Is 6 positive and even? " +

isPositiveAndEven.test(6)); // Test lambda expression

System.out.println(" Is 5 positive and even? " +

isPositiveAndEven.test(5)); // Test lambda expression

}

// Method to check if a number is positive

private static boolean isPositive(int n) {

return n > 0; // Return true if positive

}

// Method to check if a number is even

private static boolean isEven(int n) {

return n % 2 == 0; // Return true if even

}

}

Output:

1. Basic Logical Operations:

AND: true && true = true

AND: true && false = false

AND: false && true = false

AND: false && false = false

OR: true || true = true

OR: true || false = true

OR: false || true = true

OR: false || false = false

NOT: !true = false

NOT: !false = true

2. Short-circuit Evaluation:

false && (1/0 > 0) = false

true || (1/0 > 0) = true

3. Operator Precedence:

true || false && false = true

(true || false) && false = false

4. Combining with Comparison Operators:

(num1 < num2) && (num2 > 0) = true

(num1 > num2) || (num2 < 20) = true

5. Complex Conditions:

(conditionA && conditionB) || (conditionA && conditionC) = true

conditionA && (conditionB || conditionC) = true

!conditionA || (conditionB && !conditionC) = false

6. Bitwise vs. Logical Operators:

true & false = false

true | false = true

true ^ false = true

7. Short-circuit vs. Non-short-circuit:

Short-circuit AND result: false, counter = 0

Non-short-circuit AND result: false, counter = 1

8. Logical Operators with Non-boolean Operands:

(1 < 2) && (3 < 4) = true

('a' < 'b') || ('c' > 'd') = true

9. Logical Operators in Control Structures:

This will be printed.

6) Library Management System

import java.util.\*; // Importing necessary classes from the java.util package

// Book class to represent a book in the library

class Book {

String title; // Title of the book

String author; // Author of the book

boolean isIssued; // Indicates if the book is currently issued

// Constructor to initialize Book object

public Book(String title, String author) {

this.title = title; // Assigning title

this.author = author; // Assigning author

this.isIssued = false; // Initially, the book is not issued

}

// Getter method for title

public String getTitle() {

return title;

}

// Getter method for author

public String getAuthor() {

return author;

}

// Method to check if the book is issued

public boolean isIssued() {

return isIssued;

}

// Setter method to update the issued status of the book

public void setIssued(boolean issued) {

this.isIssued = issued;

}

// Overriding toString method for better representation

@Override

public String toString() {

return "Title: " + title + ", Author: " + author + ", Issued: " + isIssued;

}

}

// Member class to represent a library member

class Member {

int id; // Member ID

String name; // Member name

String contactNumber; // Member contact number

// Constructor to initialize Member object

public Member(int id, String name, String contactNumber) {

this.id = id; // Assigning ID

this.name = name; // Assigning name

this.contactNumber = contactNumber; // Assigning contact number

}

// Overriding toString method for better representation

@Override

public String toString() {

return "Member ID: " + id + ", Name: " + name + ", Contact: " +

contactNumber;

}

}

// Transaction class to record book issue and return transactions

class Transaction {

String bookTitle; // Title of the book involved in the transaction

String memberName; // Name of the member involved in the transaction

String type; // Type of transaction (Issued or Returned)

Date transactionDate; // Date of the transaction

Date issueDate; // Date of book issue

Date returnDate; // Date of book return

// Constructor to initialize Transaction object

public Transaction(String bookTitle, String memberName, String type,

Date issueDate, Date returnDate) {

this.bookTitle = bookTitle; // Assigning book title

this.memberName = memberName; // Assigning member name

this.type = type; // Assigning transaction type

this.transactionDate = new Date(); // Current date for transaction

this.issueDate = issueDate; // Assigning issue date

this.returnDate = returnDate; // Assigning return date

}

// Overriding toString method for better representation

@Override

public String toString() {

String issueDateStr = (issueDate != null) ? issueDate.toString() :

"N/A"; // Format issue date

String returnDateStr = (returnDate != null) ? returnDate.toString() :

"N/A"; // Format return date

return "Transaction: " + type +

" | Book: " + bookTitle +

" | Member: " + memberName +

" | Date: " + transactionDate +

" | Issue Date: " + issueDateStr +

" | Return Date: " + returnDateStr;

}

}

// Library class to manage books, members, and transactions

class Library {

List<Book> books = new ArrayList<>(); // List to store books

List<Member> members = new ArrayList<>(); // List to store members

List<Transaction> transactions = new ArrayList<>(); // List to store

transactions

// Method to add a book to the library

public void addBook(Book book) {

books.add(book); // Add book to the list

System.out.println("Book added: " + book); // Notify user

}

// Method to display all books in the library

public void displayBooks() {

System.out.println("\n--- List of Books ---");

for (Book book : books) {

System.out.println(book); // Print each book

}

}

// Method to issue a book to a member

public void issueBook(String title, String memberName) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title) && !book.isIssued()) {

book.setIssued(true); // Set book as issued

transactions.add(new Transaction(title, memberName, "Issued",

new Date(), null)); // Record transaction

System.out.println("Book issued: " + title + " to " +

memberName); // Notify user

return; // Exit method

}

}

System.out.println("Book not available or already issued."); // Notify if

book cannot be issued

}

// Method to return a book from a member

public void returnBook(String title, String memberName) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title) && book.isIssued()) {

book.setIssued(false); // Set book as not issued

transactions.add(new Transaction(title, memberName, "Returned",

null, new Date())); // Record transaction

System.out.println("Book returned: " + title + " by " +

memberName); // Notify user

return; // Exit method

}

}

System.out.println("Book not found or not issued."); // Notify if book

cannot be returned

}

// Method to add a member to the library

public void addMember(int id, String name, String contactNumber) {

members.add(new Member(id, name, contactNumber)); // Add member

to the list

System.out.println("Member added: " + name); // Notify user

}

// Method to display all members in the library

public void displayMembers() {

System.out.println("\n--- List of Members ---");

for (Member member : members) {

System.out.println(member); // Print each member

}

}

// Method to search for a book by title

public void searchByTitle(String title) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title)) {

System.out.println(book); // Print the found book

return; // Exit method

}

}

System.out.println("Book not found."); // Notify if book is not found

}

// Method to search for books by author

public void searchByAuthor(String author) {

boolean found = false; // Flag to track if any book is found

for (Book book : books) {

if (book.getAuthor().equalsIgnoreCase(author)) {

System.out.println(book); // Print each found book

found = true; // Set flag to true if found

}

}

if (!found) {

System.out.println("No books found by the author."); // Notify if no

books are found

}

}

// Method to display all transactions

public void displayTransactions() {

System.out.println("\n--- Transaction Details ---");

for (Transaction transaction : transactions) {

System.out.println(transaction); // Print each transaction

}

}

}

// Main class to run the Library Management System

public class LibraryManagementSystem {

static Scanner scanner = new Scanner(System.in); // Scanner for user

input

static Library library = new Library(); // Create a library instance

public static void main(String[] args) {

int choice; // Variable to hold user choice

do {

// Display menu options

System.out.println("\n--- Library Management System ---");

System.out.println("1. Add Book");

System.out.println("2. Display Books");

System.out.println("3. Issue Book");

System.out.println("4. Return Book");

System.out.println("5. Add Member");

System.out.println("6. Display Members");

System.out.println("7. Search by Title");

System.out.println("8. Search by Author");

System.out.println("9. Display Transactions");

System.out.println("10. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt(); // Read user choice

scanner.nextLine(); // Consume newline

switch (choice) {

case 1: // Add a book

System.out.print("Enter book title: ");

String title = scanner.nextLine(); // Read title

System.out.print("Enter book author: ");

String author = scanner.nextLine(); // Read author

library.addBook(new Book(title, author)); // Add book to library

break;

case 2: // Display all books

library.displayBooks();

break;

case 3: // Issue a book

System.out.print("Enter book title to issue: ");

title = scanner.nextLine(); // Read title

System.out.print("Enter member name: ");

String memberName = scanner.nextLine(); // Read member

name

library.issueBook(title, memberName); // Issue book

break;

case 4: // Return a book

System.out.print("Enter book title to return: ");

title = scanner.nextLine(); // Read title

System.out.print("Enter member name: ");

memberName = scanner.nextLine(); // Read member name

library.returnBook(title, memberName); // Return book

break;

case 5: // Add a member

System.out.print("Enter member ID: ");

int id = scanner.nextInt(); // Read member ID

scanner.nextLine(); // Consume newline

System.out.print("Enter member name: ");

String name = scanner.nextLine(); // Read member name

System.out.print("Enter contact number: ");

String contactNumber = scanner.nextLine(); // Read contact

number

library.addMember(id, name, contactNumber); // Add member

to library

break;

case 6: // Display all members

library.displayMembers();

break;

case 7: // Search by title

System.out.print("Enter book title to search: ");

title = scanner.nextLine(); // Read title

library.searchByTitle(title); // Search book by title

break;

case 8: // Search by author

System.out.print("Enter author name to search: ");

author = scanner.nextLine(); // Read author

library.searchByAuthor(author); // Search book by author

break;

case 9: // Display transactions

library.displayTransactions();

break;

case 10: // Exit the program

System.out.println("Exiting system...");

break;

default: // Handle invalid choices

System.out.println("Invalid choice. Please try again.");

}

} while (choice != 10); // Repeat until user chooses to exit

scanner.close(); // Close the scanner

}

}

Output:

--- Library Management System ---

1. Add Book

2. Display Books

3. Issue Book

4. Return Book

5. Add Member

6. Display Members

7. Search by Title

8. Search by Author

9. Display Transactions

10. Exit

Enter your choice: 1

Enter book title: The Great Gatsby

Enter book author: F. Scott Fitzgerald

Book added: Title: The Great Gatsby, Author: F. Scott Fitzgerald, Issued:

false

--- Library Management System ---

1. Add Book

2. Display Books

3. Issue Book

4. Return Book

5. Add Member

6. Display Members

7. Search by Title

8. Search by Author

9. Display Transactions

10. Exit

Enter your choice: 1

Enter book title: 1984

Enter book author: George Orwell

Book added: Title: 1984, Author: George Orwell, Issued: false

--- Library Management System ---

1. Add Book

2. Display Books

3. Issue Book

4. Return Book

5. Add Member

6. Display Members

7. Search by Title

8. Search by Author

9. Display Transactions

10. Exit

Enter your choice: 5

Enter member ID: 1

Enter member name: Alice

Enter contact number: 1234567890

Member added: Alice

--- Library Management System ---

1. Add Book

2. Display Books

3. Issue Book

4. Return Book

5. Add Member

6. Display Members

7. Search by Title

8. Search by Author

9. Display Transactions

10. Exit

Enter your choice: 3

Enter book title to issue: 1984

Enter member name: Alice

Book issued: 1984 to Alice

--- Library Management System ---

1. Add Book

2. Display Books

3. Issue Book

4. Return Book

5. Add Member

6. Display Members

7. Search by Title

8. Search by Author

9. Display Transactions

10. Exit

Enter your choice: 9

--- Transaction Details ---

Transaction: Issued | Book: 1984 | Member: Alice | Date: Wed Sep 20

15:30:45 PDT 2023 | Issue Date: N/A | Return Date: N/A

--- Library Management System ---

1. Add Book

2. Display Books

3. Issue Book

4. Return Book

5. Add Member

6. Display Members

7. Search by Title

8. Search by Author

9. Display Transactions

10. Exit

Enter your choice: 10

Exiting system...

—-----------------------------------------------------------------

**Object-Oriented Programming (OOP**) is a programming paradigm that uses "objects" to represent data and methods to manipulate that data. OOP helps in creating modular, reusable, and maintainable code, making it easier to manage complex software systems. It promotes a clear structure and logical organization of code through the use of objects.

OOP emphasizes the following key concepts:

**1. Encapsulation**: Bundling data and methods within classes, restricting access to internal states, and exposing only necessary interfaces.

**2. Abstraction**: Hiding complex implementation details and exposing only the essential features of an object, simplifying interaction.

**3. Inheritance**: Allowing one class (subclass) to inherit properties and behaviors from another class (super class), promoting code reuse and establishing relationships.

**4. Polymorphism**: Enabling objects of different classes to be treated as instances of a common super class, allowing for flexible method usage (e.g., method overriding and overloading).

**Programs related to OOP:**

**Codes discussed in class with detailed explanation of each line:-**

**1. Conditional statements**

import java.io.BufferedReader; // Importing BufferedReader for reading files import java.io.FileReader; // Importing FileReader for file operations import java.io.IOException; // Importing IOException for handling input/output exceptions

public class JavaControlStatements {

public static void main(String[] args) {

// Calling methods to demonstrate different control statements conditionalStatements();

loopingStatements();

jumpStatements();

exceptionHandling();

assertions();

}

// Method to demonstrate conditional statements

private static void conditionalStatements() {

System.out.println("\n--- Conditional Statements ---"); int userAge = 18; // Variable to store age

if (userAge >= 18) { // Check if age is 18 or older

System.out.println("You are eligible to vote."); // Print eligibility message

}

int testScore = 75; // Variable to store score

if (testScore >= 60) { // Check if score is 60 or more System.out.println("You passed!"); // Print pass message } else {

System.out.println("You failed."); // Print fail message }

int studentGrade = 85; // Variable to store grade

if (studentGrade >= 90) {

System.out.println("Grade: A"); // Print grade A } else if (studentGrade >= 80) {

System.out.println("Grade: B"); // Print grade B } else if (studentGrade >= 70) {

System.out.println("Grade: C"); // Print grade C } else if (studentGrade >= 60) {

System.out.println("Grade: D"); // Print grade D } else {

System.out.println("Grade: F"); // Print grade F

}

boolean hasDrivingLicense = true; // Variable to check license status if (userAge >= 18) { // Check if age is 18 or older

if (hasDrivingLicense) {

System.out.println("You can drive."); // Print driving permission } else {

System.out.println("You need to get a license first."); // Print license requirement

}

} else {

System.out.println("You are too young to drive."); // Print age restriction

}

int currentDay = 3; // Variable to store the current day number switch (currentDay) { // Switch based on the current day case 1:

System.out.println("Day: Monday"); // Print day for case 1 break;

case 2:

System.out.println("Day: Tuesday"); // Print day for case 2 break;

case 3:

System.out.println("Day: Wednesday"); // Print day for case 3 break;

case 4:

System.out.println("Day: Thursday"); // Print day for case 4 break;

case 5:

System.out.println("Day: Friday"); // Print day for case 5 break;

case 6:

case 7:

System.out.println("Day: Weekend"); // Print weekend for cases 6 and 7

break;

default:

System.out.println("Invalid day"); // Print error message for invalid input

}

}

// Method to demonstrate looping statements

private static void loopingStatements() {

System.out.println("\n--- Looping Statements ---");

System.out.println("For loop:");

for (int iteration = 1; iteration <= 5; iteration++) { // Loop from 1 to 5 System.out.println("Iteration: " + iteration); // Print current iteration number

}

System.out.println("\nEnhanced for loop:");

int[] numberArray = {1, 2, 3, 4, 5}; // Array of numbers for (int number : numberArray) { // Iterate over each number in the array

System.out.println("Number: " + number); // Print current number }

System.out.println("\nWhile loop:");

int count = 0; // Initialize count variable

while (count < 5) { // Continue loop while count is less than 5 System.out.println("Count: " + count); // Print current count count++; // Increment count

}

System.out.println("\nDo-while loop:");

int number = 1; // Initialize number variable

do {

System.out.println("Number: " + number); // Print current number number++; // Increment number

} while (number <= 5); // Continue while number is less than or equal to 5

}

private static void jumpStatements() {

System.out.println("\n--- Jump Statements ---");

System.out.println("Break statement:");

for (int i = 1; i <= 10; i++) { // Loop from 1 to 10

if (i == 6) { // Check if current number is 6

break; // Exit the loop when i is 6

}

System.out.println("Iteration: " + i); // Print current iteration number }

System.out.println("\nContinue statement:");

for (int i = 1; i <= 5; i++) { // Loop from 1 to 5

if (i == 3) { // Check if current number is 3

continue; // Skip the rest of the loop for i = 3

}

System.out.println("Iteration: " + i); // Print current iteration number }

System.out.println("\nReturn statement:");

System.out.println("Sum: " + sum(5, 3)); // Call sum method and print the result

}

// Method to calculate the sum of two integers

private static int sum(int firstNumber, int secondNumber) { return firstNumber + secondNumber; // Return the sum of the two numbers

}

// Method to demonstrate exception handling

private static void exceptionHandling() {

System.out.println("\n--- Exception Handling ---");

System.out.println("Try-catch:");

try {

int divisionResult = 10 / 0; // Attempt to divide by zero } catch (ArithmeticException e) { // Catch ArithmeticException System.out.println("Error: " + e.getMessage()); // Print error message }

System.out.println("\nTry-catch-finally:");

try {

int[] numberArray = {1, 2, 3}; // Create an array of numbers

System.out.println(numberArray[3]); // Attempt to access an out-of bounds index

} catch (ArrayIndexOutOfBoundsException e) { // Catch ArrayIndexOutOfBoundsException

System.out.println("Error: " + e.getMessage()); // Print error message } finally {

System.out.println("This block always executes."); // Execute this block regardless of exception

}

System.out.println("\nTry-with-resources:");

try (BufferedReader bufferedReader = new BufferedReader(new FileReader("test.txt"))) { // Automatically close resources

String line;

while ((line = bufferedReader.readLine()) != null) { // Read lines until end of file

System.out.println(line); // Print each line read from the file }

} catch (IOException e) { // Catch IOException

System.out.println("Error reading file: " + e.getMessage()); // Print error message

}

}

// Method to demonstrate assertions

private static void assertions() {

System.out.println("\n--- Assertions ---");

int userAge = -5; // Variable to store age

assert userAge >= 0 : "Age cannot be negative"; // Assert that age is non-negative

System.out.println("Age: " + userAge); // Print age if assertion passes }

}

**Output:**

--- Conditional Statements ---

You are eligible to vote. You passed!

Grade: B

You can drive.

--- Looping Statements --- For loop:

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

Iteration: 5

Enhanced for loop:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

While loop:

Count: 0

Count: 1

Count: 2

Count: 3

Count: 4

Do-while loop:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

--- Jump Statements ---

Break statement:

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

Iteration: 5

Continue statement:

Iteration: 1

Iteration: 2

Iteration: 4

Iteration: 5

Return statement:

Sum: 8

--- Exception Handling ---

Try-catch:

Error: / by zero

Try-catch-finally:

Error: Index 3 out of bounds for length 3

This block always executes.

Try-with-resources:

Error reading file: test.txt (or similar message if the file does not exist)

--- Assertions ---

Age cannot be negative

**2. ATM simulator**

import java.util.Scanner; // Importing Scanner for user input import java.util.Random; // Importing Random for simulating processing time

public class ATMSimulator {

private static final int CORRECT\_PIN = 1234; // Simulated correct PIN private static final int MAX\_PIN\_ATTEMPTS = 3; // Maximum number of allowed PIN attempts

private static double accountBalance = 1000.00; // Initial account balance

public static void main(String[] args) {

Scanner inputScanner = new Scanner(System.in); // Creating a Scanner object for input

Random processingTimeSimulator = new Random(); // Creating a Random object for simulating processing times

System.out.println("Welcome to the ATM Simulator"); // Welcome message

// PIN verification using a do-while loop

int pinAttempts = 0; // Initialize the number of attempts boolean isPinVerified = false; // Flag to track PIN verification status do {

System.out.print("Please enter your PIN: "); // Prompt for PIN entry int userEnteredPin = inputScanner.nextInt(); // Read the entered PIN

// Check if the entered PIN is correct

if (userEnteredPin == CORRECT\_PIN) {

isPinVerified = true; // Set flag to true if PIN is correct System.out.println("PIN accepted. Access granted."); // Access granted message

} else {

pinAttempts++; // Increment the attempt counter System.out.println("Incorrect PIN. Attempts remaining: " + (MAX\_PIN\_ATTEMPTS - pinAttempts)); // Show remaining attempts }

} while (!isPinVerified && pinAttempts < MAX\_PIN\_ATTEMPTS); // Continue until PIN is verified or max attempts reached

// If PIN is not verified after maximum attempts

if (!isPinVerified) {

System.out.println("Too many incorrect attempts. Your card has been blocked."); // Block message

return; // Exit the program

}

// Main ATM menu using a do-while loop

int userChoice; // Variable to store user's menu choice do {

// Display the ATM menu

System.out.println("\nATM Menu:");

System.out.println("1. Check Balance"); // Option to check balance System.out.println("2. Withdraw Money"); // Option to withdraw money

System.out.println("3. Deposit Money"); // Option to deposit money System.out.println("4. Exit"); // Option to exit

System.out.print("Enter your choice: "); // Prompt for choice userChoice = inputScanner.nextInt(); // Read the user's choice

// Handle the user's choice using a switch statement switch (userChoice) {

case 1: // Check balance

System.out.printf("Your current balance is: $%.2f\n", accountBalance); // Print balance

break;

case 2: // Withdraw money

System.out.print("Enter amount to withdraw: $"); // Prompt for withdrawal amount

double withdrawAmount = inputScanner.nextDouble(); // Read withdrawal amount

if (withdrawAmount > accountBalance) { // Check for insufficient funds

System.out.println("Insufficient funds."); // Insufficient funds message

} else {

accountBalance -= withdrawAmount; // Deduct withdrawal amount from balance

System.out.printf("$%.2f withdrawn. New balance: $%.2f\n", withdrawAmount, accountBalance); // Print new balance

}

break;

case 3: // Deposit money

System.out.print("Enter amount to deposit: $"); // Prompt for deposit amount

double depositAmount = inputScanner.nextDouble(); // Read deposit amount

accountBalance += depositAmount; // Add deposit amount to balance

System.out.printf("$%.2f deposited. New balance: $%.2f\n", depositAmount, accountBalance); // Print new balance

break;

case 4: // Exit

System.out.println("Thank you for using the ATM. Goodbye!"); // Exit message

break;

default: // Handle invalid choice

System.out.println("Invalid choice. Please try again."); // Invalid choice message

}

try {

Thread.sleep(processingTimeSimulator.nextInt(1000) + 500); // Sleep for a random time between 500ms and 1500ms

} catch (InterruptedException e) { // Handle interrupted exception e.printStackTrace(); // Print stack trace if interrupted }

} while (userChoice != 4); // Repeat until the user chooses to exit

inputScanner.close(); // Close the scanner resource }

}

**Output:**

Welcome to the ATM Simulator

Please enter your PIN: 1234

PIN accepted. Access granted.

ATM Menu:

1. Check Balance

2. Withdraw Money

3. Deposit Money

4. Exit

Enter your choice: 1

Your current balance is: $1000.00

ATM Menu:

1. Check Balance

2. Withdraw Money

3. Deposit Money

4. Exit

Enter your choice: 2

Enter amount to withdraw: $1500

Insufficient funds.

ATM Menu:

1. Check Balance

2. Withdraw Money

3. Deposit Money

4. Exit

Enter your choice: 2

Enter amount to withdraw: $200

$200.00 withdrawn. New balance: $800.00

ATM Menu:

1. Check Balance

2. Withdraw Money

3. Deposit Money

4. Exit

Enter your choice: 3

Enter amount to deposit: $500

$500.00 deposited. New balance: $1300.00

ATM Menu:

1. Check Balance

2. Withdraw Money

3. Deposit Money

4. Exit

Enter your choice: 4

Thank you for using the ATM. Goodbye!

**3. Program to explain OOP concepts:**

// 1. Class representing a Car

class Vehicle {

// Fields (attributes)

String make; // Brand of the car

String type; // Model of the car

int manufactureYear; // Year the car was manufactured

// Constructor: initializes a Vehicle object with brand, model, and year Vehicle(String make, String type, int manufactureYear) { this.make = make;

this.type = type;

this.manufactureYear = manufactureYear;

}

// Method to display vehicle information

void showInfo() {

System.out.println("Vehicle: " + manufactureYear + " " + make + " " + type);

}

}

class Account {

private String accountID; // Unique account number

private double accountBalance; // Current account balance

// Constructor to initialize an Account with ID and initial balance public Account(String accountID, double initialBalance) { this.accountID = accountID;

this.accountBalance = initialBalance;

}

// Getter method for account ID

public String getAccountID() {

return accountID;

}

// Getter method for account balance

public double getAccountBalance() {

return accountBalance;

}

// Method to deposit money into the account

public void deposit(double amount) {

if (amount > 0) {

accountBalance += amount; // Increase balance

}

}

// Method to withdraw money from the account

public void withdraw(double amount) {

if (amount > 0 && amount <= accountBalance) { accountBalance -= amount; // Decrease balance }

}

}

// Superclass representing a generic Animal

class Creature {

String creatureName; // Name of the creature

// Constructor for initializing the creature's name Creature(String creatureName) {

this.creatureName = creatureName;

}

// Method to simulate eating behavior

void consume() {

System.out.println(creatureName + " is eating."); }

}

// Subclass representing a Dog, which inherits from Creature class Puppy extends Creature {

// Constructor that passes the name to the superclass Puppy(String creatureName) {

super(creatureName); // Call the superclass constructor }

// Method specific to Dog to simulate barking void bark() {

System.out.println(creatureName + " is barking."); }

}

// Subclass representing a Cat, which inherits from Creature

class Kitten extends Creature {

// Constructor that passes the name to the superclass

Kitten(String creatureName) {

super(creatureName);

}

// Override the consume() method to provide specific behavior for cats @Override

void consume() {

System.out.println(creatureName + " is eating fish.");

}

// Method specific to Cat to simulate meowing

void meow() {

System.out.println(creatureName + " is meowing.");

}

}

// Abstract class representing a generic Shape

abstract class Figure {

// Abstract method to calculate the area of the shape

abstract double computeArea();

// Concrete method to display a generic message

void displayMessage() {

System.out.println("This is a figure.");

}

}

// Class representing a Circle, which extends Figure

class Round extends Figure {

double circleRadius; // Radius of the circle

// Constructor to initialize the circle's radius

Round(double circleRadius) {

this.circleRadius = circleRadius;

}

// Implement the abstract method to calculate area

@Override

double computeArea() {

return Math.PI \* circleRadius \* circleRadius; // Area formula for a circle

}

}

interface Paintable {

void paint(); // Abstract method to paint

}

// Class representing a Rectangle, which extends Figure and implements Paintable

class Rectangle extends Figure implements Paintable { double rectangleLength; // Length of the rectangle

double rectangleWidth; // Width of the rectangle

// Constructor to initialize length and width

Rectangle(double rectangleLength, double rectangleWidth) { this.rectangleLength = rectangleLength;

this.rectangleWidth = rectangleWidth;

}

// Implement the abstract method to calculate area

@Override

double computeArea() {

return rectangleLength \* rectangleWidth; // Area formula for a rectangle

}

// Implement the method from Paintable interface

@Override

public void paint() {

System.out.println("Painting a rectangle."); // Paint message }

}

// Main class to demonstrate all OOP concepts

public class OOPConceptsDemo {

public static void main(String[] args) {

// 1. Classes and Objects

System.out.println("1. Classes and Objects:");

Vehicle myVehicle = new Vehicle("Honda", "Civic", 2023); // Create a Vehicle object

myVehicle.showInfo(); // Display vehicle information

// 2. Encapsulation

System.out.println("\n2. Encapsulation:");

Account userAccount = new Account("987654", 2000); // Create an Account object

userAccount.deposit(800); // Deposit money

userAccount.withdraw(300); // Withdraw money

System.out.println("Account balance: $" +

userAccount.getAccountBalance()); // Show balance

// 3. Inheritance

System.out.println("\n3. Inheritance:");

Puppy myPuppy = new Puppy("Charlie"); // Create a Puppy object myPuppy.consume(); // Call eat method from Creature myPuppy.bark(); // Call bark method specific to Puppy

// 4. Polymorphism

System.out.println("\n4. Polymorphism:");

Creature myCreature = new Kitten("Luna"); // Create a Kitten object as a Creature

myCreature.consume(); // Call the overridden eat method in Kitten

// 5. Abstraction

System.out.println("\n5. Abstraction:");

Figure myCircle = new Round(7); // Create a Round object System.out.println("Circle area: " + myCircle.computeArea()); // Show area of circle

// 6. Interfaces

System.out.println("\n6. Interfaces:");

Rectangle myRectangle = new Rectangle(6, 4); // Create a Rectangle object

myRectangle.paint(); // Call the paint method

System.out.println("Rectangle area: " + myRectangle.computeArea()); // Show area of rectangle

}

}

**Output:**

1. Classes and Objects:

Vehicle: 2023 Honda Civic

2. Encapsulation:

Account balance: $2500.0

3. Inheritance:

Charlie is eating.

Charlie is barking.

4. Polymorphism:

Luna is eating fish.

5. Abstraction:

Circle area: 153.93804002589985

6. Interfaces:

Painting a rectangle.

Rectangle area: 24.0

**4. Computer System Composition Example** class Processor {

private String modelName; // Model name of the CPU private double speedGHz; // Clock speed in GHz

// Constructor to initialize the CPU model and clock speed public Processor(String modelName, double speedGHz) { this.modelName = modelName;

this.speedGHz = speedGHz;

}

// Method to simulate data processing

public void executeInstructions() {

System.out.println("Processor " + modelName + " executing instructions at " + speedGHz + " GHz");

}

}

// Class representing Random Access Memory (RAM) class Memory {

private String memoryType; // Type of RAM (e.g., DDR4) private int capacityGB; // Capacity in GB

// Constructor to initialize the RAM type and capacity public Memory(String memoryType, int capacityGB) { this.memoryType = memoryType;

this.capacityGB = capacityGB;

}

// Method to simulate loading data into RAM

public void load() {

System.out.println("Loading data into " + capacityGB + "GB " + memoryType + " memory");

}

}

// Class representing storage (e.g., Hard Drive or SSD) class DataStorage {

private String storageType; // Type of storage (e.g., SSD, HDD) private int capacityGB; // Capacity in GB

// Constructor to initialize the storage type and capacity public DataStorage(String storageType, int capacityGB) { this.storageType = storageType;

this.capacityGB = capacityGB;

}

// Method to simulate storing data

public void save() {

System.out.println("Saving data on " + capacityGB + "GB " + storageType);

}

}

// Class representing a Graphics Processing Unit (GPU) class GraphicsCard {

private String modelName; // Model name of the GPU private int memoryGB; // Memory size in GB

// Constructor to initialize the GPU model and memory size public GraphicsCard(String modelName, int memoryGB) { this.modelName = modelName;

this.memoryGB = memoryGB;

}

// Method to simulate rendering graphics

public void render() {

System.out.println("Rendering graphics using " + modelName + " with " + memoryGB + "GB memory");

}

}

// Main class that composes the components to form a complete Computer class PC {

private Processor processor; // The CPU component

private Memory memory; // The RAM component

private DataStorage storage; // The storage component

private GraphicsCard graphicsCard; // The GPU component

// Constructor to initialize the PC with its components

public PC(Processor processor, Memory memory, DataStorage storage, GraphicsCard graphicsCard) {

this.processor = processor;

this.memory = memory;

this.storage = storage;

this.graphicsCard = graphicsCard;

}

// Method to simulate booting up the computer

public void powerOn() {

System.out.println("Booting up the computer...");

processor.executeInstructions();

memory.load();

storage.save();

graphicsCard.render();

System.out.println("Computer is ready for use!");

}

// Method to simulate shutting down the computer

public void powerOff() {

System.out.println("Shutting down the computer..."); storage.save(); // Saving data before shutdown

System.out.println("Computer has been shut down."); }

// Getter methods for each component

public Processor getProcessor() { return processor; }

public Memory getMemory() { return memory; }

public DataStorage getStorage() { return storage; }

public GraphicsCard getGraphicsCard() { return graphicsCard; } }

// Example usage of the classes to demonstrate composition public class CompositionDemo {

public static void main(String[] args) {

// Creating instances of each component with specific attributes Processor myProcessor = new Processor("AMD Ryzen 5", 3.8); Memory myMemory = new Memory("DDR4", 32); DataStorage myStorage = new DataStorage("NVMe SSD", 1024); GraphicsCard myGraphicsCard = new GraphicsCard("AMD Radeon RX 6800", 16);

// Creating a PC object with all its components

PC myPC = new PC(myProcessor, myMemory, myStorage, myGraphicsCard);

// Booting up the computer

myPC.powerOn();

System.out.println("\nPerforming some operations..."); myPC.getProcessor().executeInstructions(); // Execute processor method

myPC.getGraphicsCard().render(); // Execute GPU render method

System.out.println();

// Shutting down the computer

myPC.powerOff();

}

}

**Output:**

Booting up the computer...

Processor AMD Ryzen 5 executing instructions at 3.8 GHz Loading data into 32GB DDR4 memory

Saving data on 1024GB NVMe SSD

Rendering graphics using AMD Radeon RX 6800 with 16GB memory Computer is ready for use!

Performing some operations...

Processor AMD Ryzen 5 executing instructions at 3.8 GHz Rendering graphics using AMD Radeon RX 6800 with 16GB memory

Shutting down the computer...

Saving data on 1024GB NVMe SSD

Computer has been shut down.

**5. Abstract and interface examples**

**1) Shape hierarchy**

abstract class GeometricShape {

protected String shapeColor; // Color of the shape

// Constructor to initialize the shape's color

public GeometricShape(String shapeColor) {

this.shapeColor = shapeColor;

}

// Abstract method to calculate area

public abstract double calculateArea();

// Concrete method to display the color of the shape public void showColor() {

System.out.println("Shape Color: " + shapeColor); }

}

// Class representing a Circle, extending GeometricShape class CircleShape extends GeometricShape {

private double circleRadius; // Radius of the circle

// Constructor to initialize color and radius

public CircleShape(String shapeColor, double circleRadius) {

super(shapeColor);

this.circleRadius = circleRadius;

}

// Implementing the abstract method to calculate area of the circle @Override

public double calculateArea() {

return Math.PI \* circleRadius \* circleRadius;

}

}

// Class representing a Rectangle, extending GeometricShape class RectangleShape extends GeometricShape {

private double rectLength; // Length of the rectangle private double rectWidth; // Width of the rectangle

// Constructor to initialize color, length, and width

public RectangleShape(String shapeColor, double rectLength, double rectWidth) {

super(shapeColor);

this.rectLength = rectLength;

this.rectWidth = rectWidth;

}

// Implementing the abstract method to calculate area of the rectangle @Override

public double calculateArea() {

return rectLength \* rectWidth;

}

}

System.out.println(“\nShape Hierarchy");

GeometricShape myCircle = new CircleShape("Red", 5); GeometricShape myRectangle = new RectangleShape("Blue", 4, 6);

System.out.println("Circle:");

myCircle.showColor(); // Display color of the circle

System.out.println("Area: " + myCircle.calculateArea()); // Calculate and display area

System.out.println("\nRectangle:");

myRectangle.showColor(); // Display color of the rectangle

System.out.println("Area: " + myRectangle.calculateArea()); // Calculate and display area

**Output:**

Shape Hierarchy Circle:

Shape Color: Red

Area: 78.53981633974483

Rectangle: Shape

Color: Blue Area: 24.0

**2) Vehicle interface and implementations** interface Automobile {

void ignite(); // Start the vehicle

void halt(); // Stop the vehicle

int checkFuelLevel(); // Check fuel level

}

// Abstract class implementing the Vehicle interface abstract class AbstractAutomobile implements Automobile { protected int currentFuelLevel; // Current fuel level

// Constructor to initialize fuel level

public AbstractAutomobile(int initialFuel) { this.currentFuelLevel = initialFuel;

}

// Implementing the method to get current fuel level @Override

public int checkFuelLevel() {

return currentFuelLevel;

}

// Abstract method specific to AbstractAutomobile public abstract void refuel(int amount);

}

// Class representing a Car, extending Abstract Automobile class CarVehicle extends AbstractAutomobile { // Constructor to initialize the car's fuel level public CarVehicle(int initialFuel) {

super(initialFuel);

}

// Implementing methods from the Automobile interface @Override

public void ignite() {

System.out.println("Car engine has started."); }

@Override

public void halt() {

System.out.println("Car engine has stopped.");

}

// Method to refuel the car

@Override

public void refuel(int amount) {

currentFuelLevel += amount;

System.out.println("Car refueled. New fuel level: " + currentFuelLevel); }

}

// Class representing a Bicycle, implementing Automobile interface class BicycleVehicle implements Automobile {

@Override

public void ignite() {

System.out.println("Bicycle is now moving.");

}

@Override

public void halt() {

System.out.println("Bicycle has stopped.");

}

// Bicycles don't use fuel, so always return 100%

@Override

public int checkFuelLevel() {

return 100;

}

}

System.out.println("\nVehicle Interface");

Automobile myCar = new CarVehicle(50);

Automobile myBicycle = new BicycleVehicle();

myCar.ignite(); // Start the car

System.out.println("Car fuel level: " + myCar.checkFuelLevel()); // Check fuel level

((CarVehicle) myCar).refuel(20); // Refuel the car

myCar.halt(); // Stop the car

myBicycle.ignite(); // Start the bicycle

System.out.println("Bicycle 'fuel' level: " + myBicycle.checkFuelLevel()); // Check fuel level

myBicycle.halt(); // Stop the bicycle

**Output:**

Vehicle Interface Car engine has started.

Car fuel level: 50

Car refueled. New fuel

level: 70

Car engine has stopped.

Bicycle is now moving.

Bicycle 'fuel' level: 100

Bicycle has stopped.

**3) Payment processing**

interface PaymentGateway {

boolean processPayment(double amount); // Process a payment void issueRefund(double amount); // Issue a refund }

// Abstract class for online payment processing

abstract class DigitalPaymentProcessor implements PaymentGateway { protected String merchantIdentifier; // Unique ID for the merchant

// Constructor to initialize the merchant ID

public DigitalPaymentProcessor(String merchantIdentifier) { this.merchantIdentifier = merchantIdentifier;

}

// Abstract method specific to DigitalPaymentProcessor public abstract boolean authenticateMerchant();

}

// Class for processing credit card payments

class CreditCardGateway extends DigitalPaymentProcessor { public CreditCardGateway(String merchantIdentifier) { super(merchantIdentifier);

}

@Override

public boolean processPayment(double amount) {

if (authenticateMerchant()) {

System.out.println("Processing credit card payment of $" + amount); return true;

}

return false;

}

@Override

public void issueRefund(double amount) {

System.out.println("Refunding credit card payment of $" + amount); }

@Override

public boolean authenticateMerchant() {

// Simulate merchant verification

return merchantIdentifier.startsWith("CC");

}

}

// Class for processing PayPal payments

class PayPalGateway extends DigitalPaymentProcessor { public PayPalGateway(String merchantIdentifier) {

super(merchantIdentifier);

}

@Override

public boolean processPayment(double amount) {

if (authenticateMerchant()) {

System.out.println("Processing PayPal payment of $" + amount);

return true;

}

return false;

}

@Override

public void issueRefund(double amount) {

System.out.println("Refunding PayPal payment of $" + amount); }

@Override

public boolean authenticateMerchant() {

// Simulate merchant verification

return merchantIdentifier.startsWith("PP");

}

}

// Main class to demonstrate the concepts

public class AbstractAndInterfaceExamples {

public static void main(String[] args) {

System.out.println("\n Payment Processing");

PaymentGateway creditCard = new CreditCardGateway("CC123"); PaymentGateway payPal = new PayPalGateway("PP456");

creditCard.processPayment(100.50); // Process credit card payment creditCard.issueRefund(20.00); // Refund credit card payment

payPal.processPayment(75.25); // Process PayPal payment payPal.issueRefund(75.25); // Refund PayPal payment }

}

**Output:**

Payment Processing

Processing credit card payment of $100.5

Refunding credit card payment of $20.0

Processing PayPal payment of $75.25

Refunding PayPal payment of $75.25

**6. Constructers examples**

1) // Class representing a Book

class LiteraryWork {

private String bookTitle; // Title of the book

private String bookAuthor; // Author of the book

private int totalPages; // Total number of pages in the book private String bookIsbn; // ISBN number of the book

private boolean isHardCover; // Indicates if the book is hardcover

// Parameterized constructor with all attributes

public LiteraryWork(String bookTitle, String bookAuthor, int totalPages, String bookIsbn, boolean isHardCover) {

this.bookTitle = bookTitle;

this.bookAuthor = bookAuthor;

this.totalPages = totalPages;

this.bookIsbn = bookIsbn;

this.isHardCover = isHardCover;

}

// Parameterized constructor with essential attributes

public LiteraryWork(String bookTitle, String bookAuthor, String bookIsbn) {

this(bookTitle, bookAuthor, 0, bookIsbn, false); // Calls the full constructor with default values

}

// Parameterized constructor for a book without known ISBN public LiteraryWork(String bookTitle, String bookAuthor, int totalPages) {

this(bookTitle, bookAuthor, totalPages, "Unknown", false); }

// Copy constructor

public LiteraryWork(LiteraryWork otherBook) {

this(otherBook.bookTitle, otherBook.bookAuthor,

otherBook.totalPages, otherBook.bookIsbn, otherBook.isHardCover); }

// Overriding the toString method for a readable representation of the book @Override

public String toString() {

return "LiteraryWork{" +

"bookTitle='" + bookTitle + '\'' +

", bookAuthor='" + bookAuthor + '\'' +

", totalPages=" + totalPages +

", bookIsbn='" + bookIsbn + '\'' +

", isHardCover=" + isHardCover +

'}';

}

}

**Output:**

Book Examples:

LiteraryWork{bookTitle='The Great Gatsby', bookAuthor='F. Scott Fitzgerald', totalPages=180, bookIsbn='9780743273565', isHardCover=true} LiteraryWork{bookTitle='To Kill a Mockingbird', bookAuthor='Harper Lee', totalPages=0, bookIsbn='9780446310789', isHardCover=false} LiteraryWork{bookTitle='1984', bookAuthor='George Orwell', totalPages=328, bookIsbn='Unknown', isHardCover=false}

LiteraryWork{bookTitle='The Great Gatsby', bookAuthor='F. Scott Fitzgerald', totalPages=180, bookIsbn='9780743273565', isHardCover=true}

2) // Class representing a Bank Account

class FinancialAccount {

private String accountId; // Unique account number

private String accountHolderName; // Name of the account holder private double accountBalance; // Current balance of the account private String accountCategory; // Type of the account

// Parameterized constructor for creating a new account

public FinancialAccount(String accountHolderName, String accountCategory, double initialDeposit) {

this.accountId = createAccountId(); // Generates a new account ID this.accountHolderName = accountHolderName;

this.accountCategory = accountCategory;

this.accountBalance = initialDeposit;

}

// Parameterized constructor for loading an existing account public FinancialAccount(String accountId, String accountHolderName, double accountBalance, String accountCategory) {

this.accountId = accountId;

this.accountHolderName = accountHolderName;

this.accountBalance = accountBalance;

this.accountCategory = accountCategory;

}

// Method to generate a unique account ID

private String createAccountId() {

// Simulating account ID generation

return "ACC" + Math.round(Math.random() \* 1000000); }

// Overriding the toString method for a readable representation of the account

@Override

public String toString() {

return "FinancialAccount{" +

"accountId='" + accountId + '\'' +

", accountHolderName='" + accountHolderName + '\'' + ", accountBalance=" + accountBalance +

", accountCategory='" + accountCategory + '\'' + '}';

}

}

// Main class to demonstrate parameterized constructors

public class ParameterizedConstructorDemo {

public static void main(String[] args) {

// Creating Book examples

LiteraryWork firstBook = new LiteraryWork("The Great Gatsby", "F. Scott Fitzgerald", 180, "9780743273565", true);

LiteraryWork secondBook = new LiteraryWork("To Kill a Mockingbird", "Harper Lee", "9780446310789");

LiteraryWork thirdBook = new LiteraryWork("1984", "George Orwell", 328);

LiteraryWork fourthBook = new LiteraryWork(firstBook); // Using copy constructor

// Displaying book information

System.out.println("Book Examples:");

System.out.println(firstBook);

System.out.println(secondBook);

System.out.println(thirdBook);

System.out.println(fourthBook);

// Creating Bank Account examples

FinancialAccount firstAccount = new FinancialAccount("John Doe", "Savings", 1000.0);

FinancialAccount secondAccount = new

FinancialAccount("ACC123456", "Jane Smith", 5000.0, "Checking");

// Displaying bank account information

System.out.println("\nBank Account Examples:");

System.out.println(firstAccount);

System.out.println(secondAccount);

}

}

**Output:**

Bank Account Examples:

FinancialAccount{accountId='ACC123456', accountHolderName='John Doe', accountBalance=1000.0, accountCategory='Savings'}

FinancialAccount{accountId='ACC123456', accountHolderName='Jane Smith', accountBalance=5000.0, accountCategory='Checking'}

**7. Access Specifiers Programs**

// The public class AccessModifiersExample can be accessed from any other class.

public class AccessModifiersExample {

// Public member variable

// Can be accessed from any other class

public int globalValue = 10;

// Protected member variable

// Can be accessed within the same package and by subclasses (even if in a different package)

protected int packageValue = 20;

// Default (package-private) member variable

// Can be accessed only within the same package

int localValue = 30;

// Private member variable

// Can be accessed only within this class

private int hiddenValue = 40;

// Public method

// Can be called from any other class

public void showPublicInfo() {

System.out.println("This is a public method");

// Public methods can access all members of the class

System.out.println("Accessing all variables:");

System.out.println("globalValue: " + globalValue);

System.out.println("packageValue: " + packageValue);

System.out.println("localValue: " + localValue);

System.out.println("hiddenValue: " + hiddenValue); }

// Protected method

// Can be called within the same package and by subclasses protected void showProtectedInfo() {

System.out.println("This is a protected method"); }

// Default (package-private) method

// Can be called only within the same package

void showLocalInfo() {

System.out.println("This is a default method");

}

// Private method

// Can be called only within this class

private void showHiddenInfo() {

System.out.println("This is a private method");

}

// Inner class to demonstrate access modifiers within the same class public class InnerExample {

public void accessOuterMembers() {

// Inner classes can access all members of the enclosing class, // including private members

System.out.println("InnerExample accessing hiddenValue: " + hiddenValue);

showHiddenInfo();

}

}

}

// This class is in the same file to demonstrate default access class SamePackageClass {

public void accessOuterMembers() {

AccessModifiersExample example = new AccessModifiersExample(); // Can access public, protected, and default members

System.out.println("SamePackageClass accessing globalValue: " + example.globalValue);

System.out.println("SamePackageClass accessing packageValue: " + example.packageValue);

System.out.println("SamePackageClass accessing localValue: " + example.localValue);

// Cannot access private members

example.showPublicInfo();

example.showProtectedInfo();

example.showLocalInfo();

}

}

// Subclass in the same package

class SubExample extends AccessModifiersExample {

public void accessOuterMembers() {

// Can access public, protected, and default members of the superclass System.out.println("SubExample accessing globalValue: " + globalValue);

System.out.println("SubExample accessing packageValue: " + packageValue);

System.out.println("SubExample accessing localValue: " + localValue); // Cannot access private members of the superclass

showPublicInfo();

showProtectedInfo();

showLocalInfo();

// showHiddenInfo(); // This would cause a compilation error }

}

// Class in a different package

public class ExternalClass {

public void accessOuterMembers() {

AccessModifiersExample example = new AccessModifiersExample(); // Can only access public members

System.out.println("ExternalClass accessing globalValue: " + example.globalValue);

example.showPublicInfo();

}

}

// Subclass in a different package

public class ExternalSubExample extends AccessModifiersExample { public void accessOuterMembers() {

// Can access public and protected members of the superclass System.out.println("ExternalSubExample accessing globalValue: " + globalValue);

System.out.println("ExternalSubExample accessing packageValue: " + packageValue);

// Cannot access default or private members of the superclass showPublicInfo();

showProtectedInfo();

}

}

**Output:**

SamePackageClass accessing globalValue: 10

SamePackageClass accessing packageValue: 20

SamePackageClass accessing localValue: 30

This is a public method

Accessing all variables:

globalValue: 10

packageValue: 20

localValue: 30

hiddenValue: 40

SubExample accessing globalValue: 10 SubExample accessing packageValue: 20 SubExample accessing localValue: 30 This is a public method

Accessing all variables:

globalValue: 10

packageValue: 20

localValue: 30

hiddenValue: 40

ExternalClass accessing globalValue: 10 This is a public method

Accessing all variables:

globalValue: 10

packageValue: 20

localValue: 30

hiddenValue: 40

ExternalSubExample accessing globalValue: 10 ExternalSubExample accessing packageValue: 20

This is a public method

Accessing all variables:

globalValue: 10

packageValue: 20

localValue: 30

hiddenValue: 40

**8. I/O Exception Error Program**

import java.io.\*;

import java.nio.file.\*;

import java.util.Scanner;

import java.util.stream.Stream;

public class FileIOExamples {

public static void main(String[] args) {

// File name to read and write

String exampleFileName = "example.txt";

// Content to be written to the file

String fileContent = "Hello, Java I/O!";

// Writing content to the file

writeToFile(exampleFileName, fileContent);

// Reading from the file using various methods

System.out.println("1. Using FileReader and BufferedReader:"); readUsingBufferedReader(exampleFileName);

System.out.println("\n2. Using FileInputStream:");

readUsingFileInputStream(exampleFileName);

System.out.println("\n3. Using Scanner:");

readUsingScanner(exampleFileName);

System.out.println("\n4. Using Files.readAllLines (Java NIO):"); readUsingFilesReadAllLines(exampleFileName);

System.out.println("\n5. Using Files.lines (Java NIO) with Stream API:");

readUsingFilesLines(exampleFileName);

// Demonstrating file operations such as checking file status and renaming

demonstrateFileOperations(exampleFileName);

}

// Method to write content to a specified file

public static void writeToFile(String fileName, String content) { try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {

writer.write(content); // Write the content to the file System.out.println("Content written to file successfully."); } catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage()); }

}

// Method to read file using BufferedReader

public static void readUsingBufferedReader(String fileName) { try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {

String line;

while ((line = reader.readLine()) != null) { // Read until the end of the file

System.out.println(line); // Print each line

}

} catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage()); }

}

// Method to read file using FileInputStream

public static void readUsingFileInputStream(String fileName) { try (FileInputStream fileInputStream = new

FileInputStream(fileName)) {

int charValue;

while ((charValue = fileInputStream.read()) != -1) { // Read byte by byte

System.out.print((char) charValue); // Print character representation

}

System.out.println(); // New line after reading the file } catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage()); }

}

// Method to read file using Scanner

public static void readUsingScanner(String fileName) { try (Scanner scanner = new Scanner(new File(fileName))) { while (scanner.hasNextLine()) { // Check for more lines System.out.println(scanner.nextLine()); // Print each line }

} catch (FileNotFoundException e) {

System.err.println("File not found: " + e.getMessage()); }

}

// Method to read file using Files.readAllLines (Java NIO) public static void readUsingFilesReadAllLines(String fileName) { try {

// Read all lines at once and print each line

Files.readAllLines(Paths.get(fileName)).forEach(System.out::println); } catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage()); }

}

// Method to read file using Files.lines (Java NIO) with Stream API public static void readUsingFilesLines(String fileName) { try (Stream<String> stream = Files.lines(Paths.get(fileName))) { stream.forEach(System.out::println); // Print each line using stream } catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage()); }

}

// Method to demonstrate various file operations

public static void demonstrateFileOperations(String fileName) { File file = new File(fileName); // Create File object for the file

System.out.println("\nFile Operations:");

System.out.println("Exists: " + file.exists()); // Check if the file exists System.out.println("Is File: " + file.isFile()); // Check if it's a file System.out.println("Is Directory: " + file.isDirectory()); // Check if it's a directory

System.out.println("Can Read: " + file.canRead()); // Check read permission

System.out.println("Can Write: " + file.canWrite()); // Check write permission

System.out.println("Absolute Path: " + file.getAbsolutePath()); // Print absolute path

// Renaming the file

File renamedFile = new File("new\_" + fileName);

if (file.renameTo(renamedFile)) { // Attempt to rename the file System.out.println("File renamed successfully."); } else {

System.out.println("Failed to rename the file.");

}

// Deleting the renamed file

if (renamedFile.delete()) { // Attempt to delete the renamed file System.out.println("File deleted successfully.");

} else {

System.out.println("Failed to delete the file.");

}

}

}

**Output:**

Content written to file successfully.

1. Using FileReader and BufferedReader:

Hello, Java I/O!

2. Using FileInputStream:

Hello, Java I/O!

3. Using Scanner:

Hello, Java I/O!

4. Using Files.readAllLines (Java NIO):

Hello, Java I/O!

5. Using Files.lines (Java NIO) with Stream API:

Hello, Java I/O!

File Operations:

Exists: true

Is File: true

Is Directory: false

Can Read: true

Can Write: true

Absolute Path: [absolute/path/to/example.txt]

File renamed successfully.

File deleted successfully.

**9. Checked Exception Error**

import java.util.Scanner;

// Custom checked exception for insufficient funds

class FundsInsufficientException extends Exception {

private double deficit; // Amount needed to complete the transaction

// Constructor that initializes the exception message and deficit amount public FundsInsufficientException(double deficit) {

super("Insufficient funds. You need " + deficit + " more to complete this transaction.");

this.deficit = deficit;

}

// Method to retrieve the deficit amount

public double getDeficit() {

return deficit;

}

}

// Custom unchecked exception for invalid account IDs

class AccountInvalidException extends RuntimeException { private String accountNumber; // Account ID that is invalid

// Constructor that initializes the exception message and account ID public AccountInvalidException(String accountNumber) { super("Invalid account ID: " + accountNumber);

this.accountNumber = accountNumber;

}

// Method to retrieve the invalid account ID

public String getAccountNumber() {

return accountNumber;

}

}

// Main class demonstrating the use of exceptions in banking operations public class BankAccountExceptionHandling {

public static void main(String[] args) {

// Create a new bank account with an initial balance BankAccount myAccount = new BankAccount("67890", 1500.0); Scanner inputScanner = new Scanner(System.in); // Scanner for user input

while (true) {

try {

// Prompt the user for account ID and withdrawal amount System.out.print("Enter your account ID: ");

String userAccountId = inputScanner.nextLine(); System.out.print("Enter amount to withdraw: "); double withdrawalAmount =

Double.parseDouble(inputScanner.nextLine());

// Attempt to withdraw funds from the account

myAccount.withdraw(userAccountId, withdrawalAmount);

System.out.println("Withdrawal successful. New balance: " + myAccount.getBalance());

break; // Exit loop on successful withdrawal

} catch (FundsInsufficientException e) {

// Handle the custom checked exception for insufficient funds System.out.println("Error: " + e.getMessage());

System.out.println("You need $" + e.getDeficit() + " more. Would you like to try again? (y/n)");

if (!inputScanner.nextLine().equalsIgnoreCase("y")) { break; // Exit loop if user does not want to retry }

} catch (AccountInvalidException e) {

// Handle the custom unchecked exception for invalid account IDs System.out.println("Error: " + e.getMessage());

System.out.println("Please try again with a valid account ID."); } catch (NumberFormatException e) {

// Handle case where the user input is not a valid number System.out.println("Error: Invalid amount entered. Please enter a valid number.");

} catch (Exception e) {

// Generic handler for any unexpected exceptions System.out.println("An unexpected error occurred: " + e.getMessage());

e.printStackTrace(); // Print stack trace for debugging break; // Exit loop on unexpected error

} finally {

// This block always executes, regardless of exceptions

System.out.println("Transaction attempt completed."); }

}

inputScanner.close(); // Close the scanner resource

}

}

// Bank account class with methods that may throw exceptions class BankAccount {

private String accountNumber; // Unique identifier for the account private double balance; // Current balance of the account

// Constructor to initialize account number and initial balance public BankAccount(String accountNumber, double initialBalance) { this.accountNumber = accountNumber;

this.balance = initialBalance;

}

// Method to retrieve the current balance

public double getBalance() {

return balance;

}

// Method to withdraw funds from the account

public void withdraw(String accountNumber, double amount) throws FundsInsufficientException {

// Validate the provided account ID (may throw unchecked exception) if (!this.accountNumber.equals(accountNumber)) {

throw new AccountInvalidException(accountNumber); }

// Check if sufficient funds are available (may throw checked exception)

if (amount > balance) {

double neededFunds = amount - balance; // Calculate how much more is needed

throw new FundsInsufficientException(neededFunds); // Throw exception

}

// Deduct the amount from balance if all checks pass

balance -= amount; // Perform the withdrawal

}

Output:

Enter your account ID: 67890

Enter amount to withdraw: 200

Withdrawal successful New balance: 1300.0

Transaction attempt completed.

**Project: Shopping Application**

**Code Overview**

This Java code simulates a shopping application that applies discounts based on several factors:

The number of products in the cart.

The membership status of the customer (e.g., Premium or Regular). The total bill amount.

It uses object-oriented programming principles, with classes representing different entities such as products, carts, and discounts.

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

// Product class representing each product in the store class Product {

String name;

double price;

public Product(String name, double price) { this.name = name;

this.price = price;

}

public double getPrice() {

return price;

}

}

// Cart class representing a shopping cart class Cart {

List<Product> products = new ArrayList<>(); String cartMember;

public Cart(String cartMember) { this.cartMember = cartMember; }

// Adds a product to the cart

public void addProduct(Product product) { products.add(product);

}

// Calculates the total price of the cart public double getTotalPrice() {

double totalPrice = 0;

for (Product product : products) { totalPrice += product.getPrice(); }

return totalPrice;

}

// Returns the number of products in the cart public int getProductCount() {

return products.size();

}

// Returns the cart member type

public String getCartMember() {

return cartMember;

}

}

// Discount class to apply various discounts

class Discount {

// Discount based on the number of products in the cart public double productCountDiscount(Cart cart) { if (cart.getProductCount() > 5) {

return 0.10; // 10% discount for more than 5 products } else if (cart.getProductCount() > 3) {

return 0.05; // 5% discount for more than 3 products }

return 0;

}

// Discount based on the cart member type

public double cartMemberDiscount(Cart cart) { if (cart.getCartMember().equalsIgnoreCase("Premium")) {

return 0.15; // 15% discount for premium members } else if (cart.getCartMember().equalsIgnoreCase("Regular")) { return 0.05; // 5% discount for regular members }

return 0;

}

// Discount based on the total bill amount

public double billDiscount(Cart cart) {

double total = cart.getTotalPrice();

if (total > 500) {

return 0.12; // 12% discount for bills over 500 } else if (total > 300) {

return 0.07; // 7% discount for bills over 300 }

return 0;

}

// Calculate the total discount

public double calculateTotalDiscount(Cart cart) { double totalDiscount = 0;

// Calculate discounts

double productDiscount = productCountDiscount(cart); double memberDiscount = cartMemberDiscount(cart);

double billDiscount = billDiscount(cart);

// Apply highest discount

totalDiscount = Math.max(productDiscount, Math.max(memberDiscount, billDiscount));

return totalDiscount;

}

// Calculate final price after applying the discount

public double applyDiscount(Cart cart) {

double discount = calculateTotalDiscount(cart);

double totalPrice = cart.getTotalPrice();

double finalPrice = totalPrice - (totalPrice \* discount);

return finalPrice;

}

}

// Main class to simulate the shopping application

public class ShoppingApplication {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Get the cart member type from user

System.out.print("Enter cart member type (Premium/Regular): "); String cartMember = scanner.nextLine();

Cart cart = new Cart(cartMember);

// Get the number of products from user

System.out.print("Enter the number of products you want to add to the cart: ");

int numberOfProducts = scanner.nextInt();

scanner.nextLine(); // Consume newline

// Adding products to cart based on user input

for (int i = 1; i <= numberOfProducts; i++) {

System.out.println("Enter details for product " + i);

System.out.print("Product Name: ");

String name = scanner.nextLine();

System.out.print("Product Price: ");

double price = scanner.nextDouble();

scanner.nextLine(); // Consume newline

Product product = new Product(name, price); cart.addProduct(product);

}

// Creating a discount object

Discount discount = new Discount();

// Applying discounts and calculating final price double finalPrice = discount.applyDiscount(cart);

// Display total price and final price after discount System.out.println("Total price: " + cart.getTotalPrice()); System.out.println("Final price after discount: " + finalPrice);

scanner.close();

}

}

**Output:** With simple inputs

Enter cart member type (Premium/Regular): Premium Enter the number of products you want to add to the cart: 3 Enter details for product 1

Product Name: Shampoo

Product Price: 120

Enter details for product 2

Product Name: Soap

Product Price: 50

Enter details for product 3

Product Name: Laptop

Product Price: 40000

Total price: 40170.0

Final price after discount: 34144.5

**Documentation for the shopping application:**

**Key Components:**

Product Class: Defines a product with a name and price.

Cart Class: Manages a list of products and calculates the total price and product count.

Discount Class: Applies discounts based on:

Number of products in the cart.

Membership type (e.g., "Premium" or "Regular").

Total bill amount.

Main Class (ShoppingApplication): Simulates the application with product creation, cart management, and discount calculation.

**Class: Product**

This class represents an individual product in the store.

**Attributes:**

name (String): The name of the product.

price (double): The price of the product.

**Constructor:**

Product(String name, double price): Initializes a new product with the provided name and price.

**Methods:**

getPrice(): Returns the price of the product.

**Class: Cart**

This class represents the shopping cart containing multiple products. **Attributes:**

products (List<Product>): A list of products in the cart.

cartMember (String): The membership type of the cart owner (Premium or Regular).

**Constructor:**

Cart(String cartMember): Initializes a new cart for a given member type (Premium or Regular).

**Methods:**

addProduct(Product product): Adds a product to the cart.

getTotalPrice(): Returns the total price of all products in the cart. getProductCount(): Returns the total number of products in the cart. getCartMember(): Returns the type of cart member.

**Class: Discount**

This class applies different types of discounts to the cart based on product count, membership type, and total bill amount.

**Methods:**

productCountDiscount(Cart cart): Returns a discount based on the number of products in the cart.

**Discounts:**

More than 5 products: 10% discount.

More than 3 products: 5% discount.

cartMemberDiscount(Cart cart): Returns a discount based on the type of cart member.

Premium members: 15% discount.

Regular members: 5% discount.

billDiscount(Cart cart): Returns a discount based on the total price of the cart. Bill over 500: 12% discount.

Bill over 300: 7% discount.

calculateTotalDiscount(Cart cart): Calculates the highest discount among the three available discounts and returns it.

applyDiscount(Cart cart): Applies the highest discount to the total price and returns the final price after the discount is applied.

This is the main class of the program, which simulates the shopping application. **Main Flow:**

The user is prompted to input the cart member type (Premium or Regular). The user is asked how many products they want to add to the cart.

For each product, the user inputs the product name and price, which are then added to the cart.

A Discount object is created to apply discounts to the cart.

The final price is calculated and displayed after the appropriate discount is applied. **Product Count Discount:**

If the cart contains more than 5 products, the user receives a 10% discount. If the cart contains more than 3 products, the user receives a 5% discount. **Cart Member Discount:**

If the user is a Premium member, they receive a 15% discount.

If the user is a Regular member, they receive a 5% discount.

**Bill Discount:**

If the total bill exceeds 500, the user receives a 12% discount.

If the total bill exceeds 300, the user receives a 7% discount.

Final Discount Calculation: The highest applicable discount (from product count, member type, and bill) is applied to the total price.

—----------------------------------------------------------------------

The **Enum** in Java is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly.

**Programs:**

**1. Coffee shop program using enum**

import java.util.Scanner; // Import the Scanner class for user input

public class CoffeeShop {

public enum CoffeeSize {

SMALL(3.5, 8), // Enum constant for small coffee with price and ounces

MEDIUM(4.0, 12), // Enum constant for medium coffee LARGE(4.5, 16), // Enum constant for large coffee

EXTRA\_LARGE(5.0, 20); // Enum constant for extra large coffee

private final double price; // Price of the coffee size

private final int ounces; // Ounces of the coffee size

// Constructor for the enum to initialize price and ounces CoffeeSize(double price, int ounces) {

this.price = price; // Assign the price

this.ounces = ounces; // Assign the ounces

}

// Getter method to retrieve the price

public double getPrice() {

return price; // Return the price

}

// Getter method to retrieve the ounces

public int getOunces() {

return ounces; // Return the ounces

}

}

// Main method to run the Coffee Shop program

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in); // Create a Scanner object for input

// Welcome message

System.out.println("Welcome to our Coffee Shop!");

System.out.println("Available sizes:");

// Display all available coffee sizes with their prices and ounces for (CoffeeSize size : CoffeeSize.values()) {

System.out.printf("%s (%d oz) - $%.2f%n",

size.name(), // Name of the size (e.g., SMALL) size.getOunces(), // Ounces of the size

size.getPrice()); // Price of the size

}

// Prompt the user to enter their choice of coffee size

System.out.print("Enter your choice (SMALL/MEDIUM/LARGE/EXTRA\_LARGE): ");

String userChoice = scanner.nextLine().toUpperCase(); // Get user input and convert to uppercase

// Try to select the coffee size based on user input

try {

CoffeeSize selectedSize = CoffeeSize.valueOf(userChoice); // Convert input to enum

// Display selected coffee size and its details

System.out.printf("You've selected a %s coffee (%d oz).%n", selectedSize.name(), // Name of the selected size selectedSize.getOunces()); // Ounces of the selected size

System.out.printf("Price: $%.2f%n", selectedSize.getPrice()); // Display price

} catch (IllegalArgumentException e) {

// Handle case where user input does not match any coffee size System.out.println("Invalid size selected. Please try again."); }

scanner.close(); // Close the scanner to prevent resource leak }

}

**Output:**

Welcome to our Coffee Shop!

Available sizes:

SMALL (8 oz) - $3.50

MEDIUM (12 oz) - $4.00

LARGE (16 oz) - $4.50

EXTRA\_LARGE (20 oz) - $5.00

Enter your choice (SMALL/MEDIUM/LARGE/EXTRA\_LARGE): MEDIUM

You've selected a MEDIUM coffee (12 oz).

Price: $4.00

**Caching Concept**

**Caching** is a performance optimization technique used to store frequently accessed data in a temporary storage area (cache) for faster retrieval. In Java, caching can significantly enhance application performance by reducing the time and resources spent on data fetching and processing.

Purpose: Caching reduces latency by storing results of expensive operations, like database queries or complex computations, so that subsequent requests can access the data quickly.

Types of Caching:

- In-Memory Caching: Stores data in the application’s memory (e.g., using collections like `HashMap`).

- Distributed Caching: Uses external caching systems (e.g., Redis, Memcached) to share cache data across multiple instances of an application.

**Program**

**1. Collections concept**

import java.util.\*; // Import all utility classes from the java.util package import java.util.concurrent.\*; // Import concurrent utility classes

import java.util.stream.Collectors; // Import stream classes for functional-style operations

public class CollectionsConcepts {

public static void main(String[] args) {

demonstrateList(); // Call method to demonstrate List interface demonstrateSet(); // Call method to demonstrate Set interface demonstrateQueue(); // Call method to demonstrate Queue interface demonstrateMap(); // Call method to demonstrate Map interface demonstrateUtilityClasses(); // Call method to demonstrate utility classes }

private static void demonstrateList() {

System.out.println("\n--- List Interfaces ---");

// Create an ArrayList to store strings (dynamic array)

List<String> fruitList = new ArrayList<>();

fruitList.add("Apple");

fruitList.add("Banana");

fruitList.add("Cherry");

System.out.println("ArrayList: " + fruitList); // Display the ArrayList

// Create a LinkedList initialized with the contents of fruitList List<String> linkedFruitList = new LinkedList<>(fruitList); linkedFruitList.add(1, "Blueberry"); // Insert at index 1

System.out.println("LinkedList: " + linkedFruitList); // Display the LinkedList

// Create a Vector initialized with fruitList (thread-safe dynamic array) Vector<String> fruitVector = new Vector<>(fruitList);

fruitVector.add("Date"); // Add a new element

System.out.println("Vector: " + fruitVector); // Display the Vector

// Create a Stack to demonstrate LIFO (Last In, First Out) Stack<String> fruitStack = new Stack<>();

fruitStack.push("Elderberry"); // Add to the top of the stack fruitStack.push("Fig");

System.out.println("Stack: " + fruitStack); // Display the Stack

System.out.println("Popped from stack: " + fruitStack.pop()); // Remove and display the top element

// Sort and display the ArrayList

Collections.sort(fruitList);

System.out.println("Sorted ArrayList: " + fruitList);

System.out.println("Binary search for 'Cherry': " +

Collections.binarySearch(fruitList, "Cherry")); // Search for an element }

private static void demonstrateSet() {

System.out.println("\n--- Set Interfaces ---");

// Create a HashSet (unordered collection with no duplicates) Set<String> colorSet = new HashSet<>();

colorSet.add("Red");

colorSet.add("Green");

colorSet.add("Blue");

colorSet.add("Red"); // Duplicate entry (will not be added) System.out.println("HashSet: " + colorSet); // Display the HashSet

// Create a LinkedHashSet to maintain insertion order

Set<String> orderedColorSet = new LinkedHashSet<>(); orderedColorSet.add("Dog");

orderedColorSet.add("Cat");

orderedColorSet.add("Bird");

System.out.println("LinkedHashSet: " + orderedColorSet); // Display the LinkedHashSet

// Create a TreeSet to maintain sorted order

Set<String> sortedAnimalSet = new TreeSet<>();

sortedAnimalSet.add("Zebra");

sortedAnimalSet.add("Elephant");

sortedAnimalSet.add("Lion");

System.out.println("TreeSet: " + sortedAnimalSet); // Display the TreeSet

// Demonstrate set operations: Intersection

Set<String> setA = new HashSet<>(Arrays.asList("A", "B", "C")); Set<String> setB = new HashSet<>(Arrays.asList("B", "C", "D")); setA.retainAll(setB); // Keep only common elements

System.out.println("Intersection: " + setA); // Display the intersection }

private static void demonstrateQueue() {

System.out.println("\n--- Queue Interfaces ---");

// Create a PriorityQueue (elements ordered by natural ordering) Queue<Integer> priorityQueue = new PriorityQueue<>(); priorityQueue.offer(5);

priorityQueue.offer(1);

priorityQueue.offer(3);

System.out.println("PriorityQueue: " + priorityQueue); // Display the PriorityQueue

System.out.println("Poll from PriorityQueue: " + priorityQueue.poll()); // Remove and display the highest priority element

// Create a LinkedList as a FIFO queue

Queue<String> stringQueue = new LinkedList<>();

stringQueue.offer("First");

stringQueue.offer("Second");

stringQueue.offer("Third");

System.out.println("Queue: " + stringQueue); // Display the Queue

System.out.println("Poll from Queue: " + stringQueue.poll()); // Remove and display the first element

// Create a Deque (double-ended queue)

Deque<String> stringDeque = new ArrayDeque<>();

stringDeque.addFirst("Front");

stringDeque.addLast("Back");

System.out.println("Deque: " + stringDeque); // Display the Deque

System.out.println("Poll first from Deque: " + stringDeque.pollFirst()); // Remove and display the first element

System.out.println("Poll last from Deque: " + stringDeque.pollLast()); // Remove and display the last element

// Create a Blocking Queue for thread-safe operations

BlockingQueue<String> taskQueue = new LinkedBlockingQueue<>(); taskQueue.offer("Task 1");

taskQueue.offer("Task 2");

System.out.println("BlockingQueue: " + taskQueue); // Display the Blocking Queue

}

private static void demonstrateMap() {

System.out.println("\n--- Map Interfaces ---");

// Create a HashMap to store key-value pairs

Map<String, Integer> numberMap = new HashMap<>(); numberMap.put("One", 1);

numberMap.put("Two", 2);

numberMap.put("Three", 3);

System.out.println("HashMap: " + numberMap); // Display the HashMap

// Create a LinkedHashMap to maintain insertion order Map<String, String> countryMap = new LinkedHashMap<>(); countryMap.put("USA", "Washington D.C.");

countryMap.put("UK", "London");

countryMap.put("Japan", "Tokyo");

System.out.println("LinkedHashMap: " + countryMap); // Display the LinkedHashMap

// Create a TreeMap for sorted key-value pairs

Map<String, Double> piMap = new TreeMap<>();

piMap.put("Pi", 3.14159);

piMap.put("Phi", 1.61803);

piMap.put("e", 2.71828);

System.out.println("TreeMap: " + piMap); // Display the TreeMap

// Create a Hashtable (legacy thread-safe hash table)

Hashtable<Integer, String> legacyTable = new Hashtable<>(); legacyTable.put(1, "First");

legacyTable.put(2, "Second");

System.out.println("Hashtable: " + legacyTable); // Display the Hashtable

// Create a ConcurrentHashMap for better performance in concurrent scenarios

ConcurrentMap<String, Integer> concurrentMap = new

ConcurrentHashMap<>();

concurrentMap.put("Concurrent", 1);

concurrentMap.put("Hash", 2);

concurrentMap.put("Map", 3);

System.out.println("ConcurrentHashMap: " + concurrentMap); // Display the ConcurrentHashMap

// Display keys, values, and entries of the HashMap

System.out.println("Keys in HashMap: " + numberMap.keySet()); System.out.println("Values in HashMap: " + numberMap.values()); System.out.println("Entries in HashMap: " + numberMap.entrySet()); }

private static void demonstrateUtilityClasses() {

System.out.println("\n--- Utility Classes ---");

// Use Arrays utility class

int[] numberArray = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

Arrays.sort(numberArray); // Sort the array

System.out.println("Sorted array: " + Arrays.toString(numberArray)); // Display sorted array

System.out.println("Binary search for 5: " +

Arrays.binarySearch(numberArray, 5)); // Search for an element

// Use Collections utility class

List<String> stringList = Arrays.asList("apple", "banana", "cherry"); Collections.reverse(stringList); // Reverse the list

System.out.println("Reversed list: " + stringList); // Display reversed list Collections.shuffle(stringList); // Shuffle the list

System.out.println("Shuffled list: " + stringList); // Display shuffled list

System.out.println("Max element: " + Collections.max(stringList)); // Display max element

System.out.println("Min element: " + Collections.min(stringList)); // Display min element

// Unmodifiable collections example

List<String> unmodifiableList = Collections.unmodifiableList(new ArrayList<>(stringList));

// Synchronized collections

List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>(stringList));

System.out.println("Synchronized list: " + synchronizedList); // Display synchronized list

List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); List<Integer> evenSquares = intList.stream()

.filter(n -> n % 2 == 0) // Filter even numbers

.map(n -> n \* n) // Square the even numbers

.collect(Collectors.toList()); // Collect results

System.out.println("Even squares: " + evenSquares); // Display even squares **Output:**

--- List Interfaces ---

ArrayList: [Apple, Banana, Cherry]

LinkedList: [Apple, Blueberry, Banana, Cherry]

Vector: [Apple, Banana, Cherry, Date]

Stack: [Elderberry, Fig]

Popped from stack: Fig

Sorted ArrayList: [Apple, Banana, Cherry]

Binary search for 'Cherry': 2

--- Set Interfaces ---

HashSet: [Red, Blue, Green]

LinkedHashSet: [Dog, Cat, Bird]

TreeSet: [Elephant, Lion, Zebra]

Intersection: [B, C]

--- Queue Interfaces ---

PriorityQueue: [1, 5, 3]

Poll from PriorityQueue: 1

Queue: [First, Second, Third]

Poll from Queue: First

Deque: [Front, Back]

Poll first from Deque: Front

Poll last from Deque: Back

BlockingQueue: [Task 1, Task 2]

--- Map Interfaces ---

HashMap: {One=1, Two=2, Three=3}

LinkedHashMap: {USA=Washington D.C., UK=London, Japan=Tokyo} TreeMap: {Pi=3.14159, Phi=1.61803, e=2.71828}

Hashtable: {1=First, 2=Second}

ConcurrentHashMap: {Concurrent=1, Hash=2, Map=3} Keys in HashMap: [One, Two, Three]

Values in HashMap: [1, 2, 3]

Entries in HashMap: [One=1, Two=2, Three=3]

--- Utility Classes ---

Sorted array: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

Binary search for 5: 6

Reversed list: [cherry, banana, apple]

Shuffled list: [banana, apple, cherry]

Max element: cherry

Min element: apple

Synchronized list: [banana, apple, cherry]

Even squares: [4, 16, 36, 64, 100]

**Threads Concept:**

In Java, the concept of threads allows for concurrent execution of code, enabling multitasking within a single program. A thread is essentially a lightweight process that can run independently, sharing the same memory space of the parent process. This parallel execution helps improve application performance, especially in programs that require handling multiple tasks simultaneously, such as web servers or user interfaces.

Java provides two main ways to create threads: by extending the `Thread` class or by implementing the `Runnable` interface. The `Thread` class has methods like `start()`, which initiates the thread, and `run()`, where the thread's code is defined. Implementing `Runnable` is often preferred because it allows for more flexibility, as a class can extend another class while still implementing the `Runnable` interface.

Thread management is crucial to avoid issues such as deadlocks (where two or more threads are waiting on each other indefinitely) and race conditions (where the output depends on the sequence of thread execution). Java provides synchronization mechanisms, such as the `synchronized` keyword, to control access to shared resources, ensuring thread safety.

Java also has high-level concurrency utilities in the `java.util.concurrent` package, which include thread pools (managed groups of threads) and synchronization constructs like `CountDownLatch`, `CyclicBarrier`, and `Semaphore`. These tools make it easier to develop scalable and efficient multithreaded applications, improving resource management and performance.

Program

public class ThreadingExamples {

// Volatile keyword ensures that the variable is always read from main memory private volatile boolean isRunning = false;

// Synchronized method to demonstrate thread synchronization public synchronized void executeSynchronizedMethod() {

System.out.println(Thread.currentThread().getName() + " entered synchronized method");

try {

Thread.sleep(1000); // Simulating work with sleep

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " exiting synchronized method");

}

// Runnable interface implementation

class TaskRunnable implements Runnable {

public void run() {

System.out.println(Thread.currentThread().getName() + " is executing Runnable");

}

}

// Thread class extension

class TaskThread extends Thread {

public void run() {

System.out.println(Thread.currentThread().getName() + " is executing Thread");

}

}

public void demonstrateThreading() {

// Creating and starting a thread using Runnable

Thread runnableThread = new Thread(new TaskRunnable(), "RunnableThread");

runnableThread.start();

// Creating and starting a thread by extending Thread class TaskThread extendedThread = new TaskThread();

extendedThread.setName("ExtendedThread");

extendedThread.start();

// Creating a thread using lambda expression

Thread lambdaThread = new Thread(() ->

System.out.println(Thread.currentThread().getName() + " is executing Lambda"), "LambdaThread");

lambdaThread.start();

// Demonstrating thread states

System.out.println("LambdaThread state: " + lambdaThread.getState());

// Thread priority

lambdaThread.setPriority(Thread.MAX\_PRIORITY);

System.out.println("LambdaThread priority: " + lambdaThread.getPriority());

// Joining threads

try {

runnableThread.join(); // Wait for runnableThread to finish

extendedThread.join(); // Wait for extendedThread to finish lambdaThread.join(); // Wait for lambdaThread to finish } catch (InterruptedException e) {

e.printStackTrace();

}

// Demonstrating synchronized block

synchronized(this) {

System.out.println(Thread.currentThread().getName() + " is in synchronized block");

}

// Using wait() and notify()

final Object lockObject = new Object();

Thread waiterThread = new Thread(() -> {

synchronized(lockObject) {

try {

System.out.println("Waiter is waiting");

lockObject.wait(); // Wait for notification System.out.println("Waiter is notified");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

Thread notifierThread = new Thread(() -> { synchronized(lockObject) {

System.out.println("Notifier is notifying"); lockObject.notify(); // Notify waiting threads }

});

waiterThread.start();

try {

Thread.sleep(1000); // Ensure waiter starts waiting first } catch (InterruptedException e) {

e.printStackTrace();

}

notifierThread.start();

// Demonstrating interrupt

Thread sleeperThread = new Thread(() -> { try {

Thread.sleep(5000); // Sleep for 5 seconds } catch (InterruptedException e) {

System.out.println("Sleeper was interrupted");

}

});

sleeperThread.start();

sleeperThread.interrupt(); // Interrupt sleeperThread immediately

// Using ThreadLocal

ThreadLocal<Integer> threadLocalValue = new ThreadLocal<>(); threadLocalValue.set(42); // Set value for this thread

System.out.println("ThreadLocal value: " + threadLocalValue.get());

// Demonstrating deadlock (be cautious when running this) final Object resourceA = new Object();

final Object resourceB = new Object();

Thread deadlockThread1 = new Thread(() -> {

synchronized(resourceA) {

System.out.println("Thread 1: Holding resource A..."); try { Thread.sleep(100); } catch (InterruptedException e) {} System.out.println("Thread 1: Waiting for resource B..."); synchronized(resourceB) {

System.out.println("Thread 1: Holding resource A and resource B"); }

}

});

Thread deadlockThread2 = new Thread(() -> {

synchronized(resourceB) {

System.out.println("Thread 2: Holding resource B..."); try { Thread.sleep(100); } catch (InterruptedException e) {} System.out.println("Thread 2: Waiting for resource A..."); synchronized(resourceA) {

System.out.println("Thread 2: Holding resource B and resource A"); }

}

});

deadlockThread1.start();

deadlockThread2.start();

}

public static void main(String[] args) {

new ThreadingExamples().demonstrateThreading();

}

}

**Output:**

RunnableThread is executing Runnable ExtendedThread is executing Thread LambdaThread is executing Lambda LambdaThread state: RUNNABLE LambdaThread priority: 10

Thread 1: Holding resource A... Thread 2: Holding resource B... Thread 1: Waiting for resource B... Thread 2: Waiting for resource A... Waiter is waiting

Notifier is notifying

Waiter is notified

Thread 1 is in synchronized block ThreadLocal value: 42

Sleeper was interrupted

—------------------------------------------------------------------------

WEEK-2

**HashMap and Caching in Java**

**HashMap**

A **HashMap** in Java is a part of the Java Collections Framework. It implements the Map interface and stores key-value pairs. Here are some key features:

∙ **Key-Value Pairs**: Each key maps to exactly one value. Keys are unique, while values can be duplicated.

∙ **Null Values**: A HashMap allows one null key and multiple null values. ∙ **Performance**: It provides average time complexity of O(1) for basic operations like add, remove, and contains.

**Syntax Example:**

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

HashMap<String, Integer> map = new HashMap<>();

map.put("Apple", 1);

map.put("Banana", 2);

map.put("Orange", 3);

// Accessing a value

System.out.println("Apple: " + map.get("Apple")); // Output: Apple: 1

// Checking existence

System.out.println("Contains Banana? " + map.containsKey("Banana")); // Output: true

// Iterating over entries

for (var entry : map.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

**Caching**

**Caching** is a technique to store frequently accessed data for quick retrieval, enhancing performance and reducing load times. In Java, caching can be implemented using collections like HashMap.

**Example of Caching with HashMap**:

import java.util.HashMap;

public class SimpleCache {

private HashMap<String, String> cache = new HashMap<>();

public String getData(String key) {

// Check if data is in cache

if (cache.containsKey(key)) {

return cache.get(key); // Return cached data

} else {

// Simulate data retrieval (e.g., from a database)

String data = "Data for " + key;

cache.put(key, data); // Cache the new data

return data;

}

}

public static void main(String[] args) {

SimpleCache simpleCache = new SimpleCache();

System.out.println(simpleCache.getData("user1")); // Fetches and caches System.out.println(simpleCache.getData("user1")); // Retrieves from cache }

}

—--------------------------------------------------------------------------------

JAVA application that implements a weather data caching and prediction

system

Program

import java.io.\*;

import java.util.concurrent.\*;

import java.util.concurrent.atomic.AtomicInteger;

// Class representing the weather data

class WeatherData implements Serializable {

private static final long serialVersionUID = 1L;

private String location;

private String temperature;

public WeatherData(String location, String temperature) {

this.location = location;

this.temperature = temperature;

}

@Override

public String toString() {

return "WeatherData{" +

"location='" + location + '\'' +

", temperature='" + temperature + '\'' +

'}';

}

}

// Main class for caching and fetching weather data

public class WeatherDataCache {

private final ConcurrentHashMap<String, WeatherData> cache = new

ConcurrentHashMap<>();

private final AtomicInteger cacheHits = new AtomicInteger(0);

private final AtomicInteger cacheMisses = new AtomicInteger(0);

private final ExecutorService executorService = Executors.newFixedThreadPool(4);

private volatile boolean isInitialized = false; // For thread-safe initialization

// Method to fetch weather data

public Future<WeatherData> fetchWeatherData(String location) {

// Check if the data is in the cache

WeatherData cachedData = cache.get(location);

if (cachedData != null) {

cacheHits.incrementAndGet(); // Increment cache hit count

return CompletableFuture.completedFuture(cachedData); // Return cached data

} else {

cacheMisses.incrementAndGet(); // Increment cache miss count

return executorService.submit(() -> {

// Simulating an asynchronous fetch from a weather API

// In a real scenario, you would replace this with an actual API call

Thread.sleep(2000); // Simulating delay

WeatherData newData = new WeatherData(location, "20°C"); // Dummy data

cache.put(location, newData); // Cache the new data

return newData; // Return the new data

});

}

}

// Method to save cache to disk

public void saveCacheToDisk(String filePath) throws IOException {

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filePath)))

{

oos.writeObject(cache);

}

}

// Method to load cache from disk

public void loadCacheFromDisk(String filePath) throws IOException,

ClassNotFoundException {

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filePath))) {

ConcurrentHashMap<String, WeatherData> loadedCache =

(ConcurrentHashMap<String, WeatherData>) ois.readObject();

cache.putAll(loadedCache);

isInitialized = true; // Mark the cache as initialized

}

}

// Method to display cache hits and misses

public void displayStats() {

System.out.println("Cache Hits: " + cacheHits.get());

System.out.println("Cache Misses: " + cacheMisses.get());

}

// Main method to demonstrate functionality

public static void main(String[] args) {

WeatherDataCache weatherDataCache = new WeatherDataCache();

// Load cache from disk (if needed)

String cacheFilePath = "weather\_cache.ser";

try {

weatherDataCache.loadCacheFromDisk(cacheFilePath);

} catch (IOException | ClassNotFoundException e) {

System.out.println("No cache found. Starting fresh.");

}

// Fetch weather data for different locations

Future<WeatherData> futureData1 = weatherDataCache.fetchWeatherData("New York");

Future<WeatherData> futureData2 = weatherDataCache.fetchWeatherData("Los Angeles");

Future<WeatherData> futureData3 = weatherDataCache.fetchWeatherData("New York");

// This should hit the cache

try {

// Get the results from the futures

System.out.println(futureData1.get()); // Blocking call until data is available

System.out.println(futureData2.get()); // Blocking call

System.out.println(futureData3.get()); // Should return cached data

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

// Display cache statistics

weatherDataCache.displayStats();

// Save cache to disk

try {

weatherDataCache.saveCacheToDisk(cacheFilePath);

} catch (IOException e) {

e.printStackTrace();

}

// Shutdown the executor service

weatherDataCache.executorService.shutdown();

}

}

OUTPUT:

No cache found. Starting fresh.

WeatherData{location='New York', temperature='20°C'}

WeatherData{location='Los Angeles', temperature='20°C'}

WeatherData{location='New York', temperature='20°C'}

Cache Hits: 1

Cache Misses: 2

Documentation of the above code:

1. Imports: Necessary classes for concurrent programming, serialization, and file handling

are imported.

2. WeatherData Class: Represents the weather data, which is serializable for disk storage.

3. WeatherDataCache Class:

o Contains a ConcurrentHashMap for thread-safe caching of weather data.

o Uses AtomicInteger for tracking cache hits and misses.

o Utilizes ExecutorService for managing asynchronous tasks.

o A volatile boolean isInitialized ensures the latest data visibility across threads.

4. fetchWeatherData Method:

o Checks the cache for the requested location.

o If found, increments cache hits and returns the data.

o If not found, increments cache misses and fetches new data asynchronously.

5. Disk Operations:

o saveCacheToDisk method serializes the cache to a file.

o loadCacheFromDisk method deserializes the cache from a file.

6. displayStats Method: Displays the number of cache hits and misses.

7. main Method:

o Initializes the cache, loads from disk if available, fetches weather data, and

displays results.

o Saves the cache to disk before exiting

—----------------------------------------------------------------------------

DATA STRUCTURES

Sorting techniques are algorithms used to arrange elements in a specific order,

typically ascending or descending. Common methods include:

1. Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in

the wrong order. This process continues until the array is sorted.

Syntax:

public static void bubbleSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap arr[j] and arr[j+1]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

2. Selection Sort

Selection Sort divides the array into a sorted and unsorted region, repeatedly

selecting the smallest element from the unsorted part.

Syntax:

public static void selectionSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap arr[i] and arr[minIndex]

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

3. Insertion Sort

Insertion Sort builds a sorted array one element at a time by repeatedly taking an

element and inserting it into its correct position.

Syntax:

public static void insertionSort(int[] arr) {

int n = arr.length;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

4. Merge Sort

Merge Sort is a divide-and-conquer algorithm that splits the array into halves,

recursively sorts them, and merges the sorted halves.

Syntax:

public static void mergeSort(int[] arr) {

if (arr.length > 1) {

int mid = arr.length / 2;

int[] left = Arrays.copyOfRange(arr, 0, mid);

int[] right = Arrays.copyOfRange(arr, mid, arr.length);

mergeSort(left);

mergeSort(right);

merge(arr, left, right);

}

}

private static void merge(int[] arr, int[] left, int[] right) {

int i = 0, j = 0, k = 0;

while (i < left.length && j < right.length) {

if (left[i] <= right[j]) {

arr[k++] = left[i++];

} else {

arr[k++] = right[j++];

}

}

while (i < left.length) arr[k++] = left[i++];

while (j < right.length) arr[k++] = right[j++];

}

5. Quick Sort

Quick Sort selects a pivot element and partitions the array into sub-arrays that are

less than and greater than the pivot, then recursively sorts the sub-arrays.

Syntax:

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap arr[i + 1] and arr[high] (or pivot)

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

—----------------------------------------------------------------------------------

Arrays are a fundamental data structure that stores a fixed-size sequence of

elements of the same type. They allow for efficient access and modification of

elements using indices, making operations like searching and sorting

straightforward. Arrays are stored in contiguous memory locations, providing O(1)

time complexity for accessing elements. However, they have a fixed size, which

can limit flexibility. Dynamic arrays, like Array Lists in Java, can resize but may

incur performance overhead during resizing. Arrays are widely used in various

applications, from basic data storage to complex algorithms, due to their simplicity

and efficiency.

Program\_1: To remove duplicates from an array using a Set

import java.util.Arrays;

import java.util.LinkedHashSet;

public class RemoveDuplicates {

public static int[] removeDuplicates(int[] arr) {

// Using LinkedHashSet to maintain insertion order

LinkedHashSet<Integer> set = new LinkedHashSet<>();

for (int num : arr) {

set.add(num); // Add elements to the set

}

// Convert the set back to an array

int[] uniqueArray = new int[set.size()];

int index = 0;

for (int num : set) {

uniqueArray[index++] = num;

}

return uniqueArray;

}

public static void main(String[] args) {

int[] array = {1, 2, 2, 3, 4, 4, 5};

int[] result = removeDuplicates(array);

System.out.println(Arrays.toString(result)); // Output: [1, 2, 3, 4, 5]

}

}

Program\_2: To achieve the task of aligning unique elements at the beginning of

an array and placing duplicates at the end, we can use a combination of a Set to

track unique elements and a List to maintain the order.

import java.util.\*;

public class AlignUniqueElements {

public static int[] alignUniqueElements(int[] arr) {

Set<Integer> uniqueSet = new HashSet<>();

List<Integer> uniqueList = new ArrayList<>();

List<Integer> duplicateList = new ArrayList<>();

// Traverse the array to separate unique and duplicate elements

for (int num : arr) {

if (!uniqueSet.add(num)) { // If add returns false, it’s a duplicate

duplicateList.add(num);

} else {

uniqueList.add(num); // It's unique

}

}

// Combine unique and duplicate elements

uniqueList.addAll(duplicateList);

// Convert List back to int array

return uniqueList.stream().mapToInt(i -> i).toArray();

}

public static void main(String[] args) {

int[] array = {1, 1, 2, 3, 4, 5, 5};

int[] result = alignUniqueElements(array);

System.out.println(Arrays.toString(result)); // Output: [2, 3, 4, 1, 5]

}

}

Program\_3: To find the second largest number in an array

public class SecondLargest {

public static int findSecondLargest(int[] arr) {

if (arr.length < 2) {

throw new IllegalArgumentException("Array must have at least two

elements.");

}

int largest = Integer.MIN\_VALUE;

int secondLargest = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > largest) {

secondLargest = largest; // Update second largest

largest = num; // Update largest

} else if (num > secondLargest && num != largest) {

secondLargest = num; // Update second largest if it's not equal to

largest

}

}

if (secondLargest == Integer.MIN\_VALUE) {

throw new IllegalArgumentException("There is no second largest

element.");

}

return secondLargest;

}

public static void main(String[] args) {

int[] array = {12, 35, 1, 10, 34, 1};

int secondLargest = findSecondLargest(array);

System.out.println("Second largest number is: " + secondLargest);

// Output: 34

}

}

Program\_4: To left rotate an array by 1 position

public class LeftRotateArray {

public static void leftRotateByOne(int[] arr) {

if (arr.length == 0) return; // Handle empty array case

int firstElement = arr[0];

for (int i = 0; i < arr.length - 1; i++) {

arr[i] = arr[i + 1]; // Shift elements to the left

}

arr[arr.length - 1] = firstElement; // Place the first element at the end

}

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

System.out.println("Original array: ");

printArray(array);

leftRotateByOne(array);

System.out.println("Array after left rotation: ");

printArray(array);

}

public static void printArray(int[] arr) {

for (int num : arr) {

System.out.print(num + " ");

}

System.out.println();

}

}

Output:

For the input array {1, 2, 3, 4, 5}, the output will be:

Original array:

1 2 3 4 5

Array after left rotation:

2 3 4 5 1

Program\_5: To left rotate an array by 3 positions (by array slicing)

import java.util.Arrays;

public class LeftRotateArray {

public static void leftRotateByK(int[] arr, int k) {

int n = arr.length;

k = k % n; // Handle cases where k is greater than the array length

if (k == 0 || n == 0) return; // No rotation needed

// Create a temporary array to hold the rotated values

int[] temp = new int[n];

// Fill the temporary array with rotated values

for (int i = 0; i < n; i++) {

temp[i] = arr[(i + k) % n];

}

// Copy the temporary array back to the original array

for (int i = 0; i < n; i++) {

arr[i] = temp[i];

}

}

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

System.out.println("Original array: " + Arrays.toString(array));

leftRotateByK(array, 3);

System.out.println("Array after left rotation by 3: " + Arrays.toString(array));

}

}

Output:

For the input array {1, 2, 3, 4, 5}, the output will be:

Original array: [1, 2, 3, 4, 5]

Array after left rotation by 3: [4, 5, 1, 2, 3]

Program\_6: To move all zeros in an array to the end while maintaining the order

of non-zero elements

public class MoveZerosToEnd {

public static void moveZeros(int[] arr) {

int count = 0; // Count of non-zero elements

// Traverse the array and keep non-zero elements at the front

for (int num : arr) {

if (num != 0) {

arr[count++] = num; // Move non-zero element to the front

}

}

// Fill remaining positions with zeros

while (count < arr.length) {

arr[count++] = 0;

}

}

public static void main(String[] args) {

int[] array = {0, 1, 0, 3, 12};

System.out.println("Original array: ");

printArray(array);

moveZeros(array);

System.out.println("Array after moving zeros to the end: ");

printArray(array);

}

public static void printArray(int[] arr) {

for (int num : arr) {

System.out.print(num + " ");

}

System.out.println();

}

}

Output:

For the input array {0, 1, 0, 3, 12}, the output will be:

Original array:

0 1 0 3 12

Array after moving zeros to the end:

1 3 12 0 0

Program\_7: To find the union of two arrays (common elements combined with all

unique elements), removing duplicates and sorting the result, you can use a

combination of Set and List

import java.util.\*;

public class UnionOfArrays {

public static int[] findUnion(int[] arr1, int[] arr2) {

Set<Integer> unionSet = new HashSet<>();

// Add all elements from both arrays to the set

for (int num : arr1) {

unionSet.add(num);

}

for (int num : arr2) {

unionSet.add(num);

}

// Convert the set to a list and sort it

List<Integer> unionList = new ArrayList<>(unionSet);

Collections.sort(unionList);

// Convert the list back to an array

return unionList.stream().mapToInt(i -> i).toArray();

}

public static void main(String[] args) {

int[] array1 = {1, 2, 3, 4, 5};

int[] array2 = {2, 3, 4, 5, 5};

int[] result = findUnion(array1, array2);

System.out.println("Union of arrays (sorted and unique): " +

Arrays.toString(result));

}

}

Output:

For the input arrays {1, 2, 3, 4, 5} and {2, 3, 4, 5, 5}, the output will be:

Union of arrays (sorted and unique): [1, 2, 3, 4, 5]

—-----------------------------------------------------------------------------------------------------

PROJECT-BOOKSTORE

## 1. Project Overview

The **Bookstore Management System** is a web application developed to automate and simplify the process of managing books in a bookstore. This system allows the store manager to add, update, view, and delete books, while customers can browse books and check availability. The project incorporates simple caching to reduce database load for frequently requested data, improving the system's efficiency.

## 2. Project Objectives

* Develop a reliable and efficient system for managing bookstore operations.
* Implement caching to improve performance, minimizing repeated database calls.
* Use SQL for efficient data storage and retrieval.
* Design an intuitive UI with HTML and CSS for a smooth user experience.

## 3. Tools and Technologies Used

* **Backend**: Java (JDK 11+), Spring Boot (optional), JDBC (for database connectivity)
* **Database**: MySQL
* **Frontend**: HTML, CSS
* **Caching**: Simple in-memory caching with HashMap or Ehcache for temporary data storage

## 4. System Design

### 4.1 Use Case Diagram

* **Admin (Store Manager)**:
  + Add books to inventory.
  + Update or delete book details.
  + View current inventory.
* **Customer**:
  + Search for books by title, author, or genre.
  + View book details.
  + Check book availability.

### 4.2 Database Design

The database consists of a **Books** table with the following fields:

* **book\_id**: INT (Primary Key)
* **title**: VARCHAR(100)
* **author**: VARCHAR(100)
* **genre**: VARCHAR(50)
* **price**: DECIMAL(10,2)
* **stock**: INT (represents available copies)

### 4.3 Data Flow and Architecture

* **Frontend**: HTML and CSS provide a responsive user interface for accessing various functions.
* **Backend (Java)**: Provides the main application logic, interacts with the database, and serves data to the frontend.
* **Caching Layer**: Simple caching stores frequently accessed data to reduce direct database access.

## 5. Key Components and Code

### 5.1 Backend (Java)

#### Book Class (Model)

java

Copy code

public class Book {

private int bookId;

private String title;

private String author;

private String genre;

private double price;

private int stock;

// Constructors, getters, and setters

}

#### Database Operations (DAO Class)

java

Copy code

public class BookDAO {

private Connection connection;

public BookDAO() {

connection = DatabaseConnection.getConnection();

}

// Method to retrieve book by title (with cache check)

public Book getBookByTitle(String title) {

Book book = Cache.getFromCache(title);

if (book == null) {

String query = "SELECT \* FROM books WHERE title = ?";

// JDBC code to fetch from database

// If found, store in cache for future requests

Cache.addToCache(title, book);

}

return book;

}

}

#### Caching Class

java

Copy code

import java.util.HashMap;

public class Cache {

private static HashMap<String, Book> bookCache = new HashMap<>();

public static Book getFromCache(String title) {

return bookCache.get(title);

}

public static void addToCache(String title, Book book) {

bookCache.put(title, book);

}

public static void clearCache() {

bookCache.clear();

}

}

### 5.2 Database (SQL)

#### SQL Script to Create Database and Table

sql

Copy code

CREATE DATABASE bookstore\_db;

USE bookstore\_db;

CREATE TABLE books (

book\_id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(100),

author VARCHAR(100),

genre VARCHAR(50),

price DECIMAL(10, 2),

stock INT

);

#### Sample Data

sql

Copy code

INSERT INTO books (title, author, genre, price, stock) VALUES

('The Great Gatsby', 'F. Scott Fitzgerald', 'Fiction', 15.99, 10),

('To Kill a Mockingbird', 'Harper Lee', 'Classic', 12.99, 5),

('1984', 'George Orwell', 'Dystopian', 14.99, 8);

### 5.3 Frontend (HTML and CSS)

#### HTML Structure

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Bookstore</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<header>

<h1>Welcome to the Bookstore</h1>

</header>

<section id="book-list">

<!-- Book list goes here -->

</section>

</body>

</html>

#### CSS Styling (styles.css)

css

Copy code

body {

font-family: Arial, sans-serif;

background-color: #f5f5f5;

color: #333;

}

header {

background-color: #333;

color: white;

text-align: center;

padding: 1em;

}

#book-list {

margin: 20px;

padding: 10px;

background-color: white;

border: 1px solid #ddd;

}

## 6. Implementation of Caching

In this project, a simple cache is implemented using a HashMap in the Cache class. This cache stores books fetched by title to avoid redundant database calls. When a book is requested:

1. The system checks the cache first.
2. If the book is found in the cache, it’s returned directly.
3. If the book isn’t in the cache, it’s retrieved from the database and stored in the cache for future requests.

## 7. Testing and Validation

1. **Database Operations**: Test CRUD operations (Create, Read, Update, Delete) on book records to ensure data consistency.
2. **Cache Functionality**: Verify that frequently accessed books are served from the cache to minimize database queries.
3. **UI**: Test the HTML/CSS UI for responsiveness and compatibility across different devices and browsers.

## 8. Future Improvements

* **Advanced Caching**: Implement a distributed cache for larger systems or integrate with caching libraries (e.g., Redis or Ehcache).
* **Enhanced UI**: Use a JavaScript framework (like Angular or React) for a dynamic and interactive frontend.
* **User Authentication**: Add login functionality to manage different roles (e.g., admin, customer).

## 9. Conclusion

The Bookstore Management System successfully automates book management processes, enabling efficient operations. By implementing a simple caching mechanism, the application optimizes database performance, serving repeated data requests more efficiently.

—-----------------------------------------------------------------------  
WEEK-3

A **Trie** (or **prefix tree**) is a specialized data structure used to store a set of strings in an efficient way. Tries are particularly useful for tasks like **autocomplete**, **spell-checking**, and **prefix-based searching**. Each node in a trie represents a single character, and paths from the root to the nodes represent prefixes of strings in the trie.

## 1. Structure of a Trie Node

Each node in the Trie contains:

* **Children nodes**: These represent each letter (character) that could follow the current character.
* **End of Word marker**: A boolean flag that indicates whether a node represents the end of a valid word.

class TrieNode {

TrieNode[] children = new TrieNode[26]; // Array for 26 letters

boolean isEndOfWord = false; // True if this node represents the end of a word

}

public class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

// Insert a word into the trie

public void insert(String word) {

TrieNode currentNode = root;

for (char ch : word.toCharArray()) {

int index = ch - 'a'; // Map character to index (0 for 'a', 1 for 'b', etc.)

if (currentNode.children[index] == null) {

currentNode.children[index] = new TrieNode(); // Create node if it doesn't exist

}

currentNode = currentNode.children[index];

}

currentNode.isEndOfWord = true; // Mark the end of the word

}

// Search for a word in the trie

public boolean search(String word) {

TrieNode currentNode = root;

for (char ch : word.toCharArray()) {

int index = ch - 'a';

if (currentNode.children[index] == null) {

return false; // If node doesn't exist, word isn't in the trie

}

currentNode = currentNode.children[index];

}

return currentNode.isEndOfWord; // Return true if it's the end of a word

}

}

EXAMPLE

public class Main {

public static void main(String[] args) {

Trie trie = new Trie();

trie.insert("apple");

trie.insert("app");

System.out.println(trie.search("apple")); // Output: true

System.out.println(trie.search("app")); // Output: true

System.out.println(trie.search("ap")); // Output: false

}

}

### Explanation

* **Insert**: insert("apple") inserts each character in "apple" and marks the last character as the end of a word.
* **Search**: search("apple") and search("app") return true if they are stored in the trie. search("ap") returns false because "ap" was not marked as a complete word.

—------------------------------------------------------------------------------------------

Quick Sort

Quick Sort is a divide-and-conquer algorithm that sorts an array by choosing a

"pivot" element, partitioning the array around the pivot, and then recursively

sorting the left and right subarrays.

Key Characteristics:

1. Divide-and-Conquer: The array is divided into two parts around a pivot.

2. In-Place Sorting: It requires only a small, constant amount of extra storage

space.

3. Unstable Sort: The relative order of equal elements may not be preserved.

4. Time Complexity:

o Best case: O(n log n)

o Average case: O(n log n)

o Worst case: O(n2) (occurs when the pivot is the smallest or largest

element)

Quick Sort Algorithm:

1. Choose a Pivot: Select an element from the array.

2. Partition: Reorder the array so that elements smaller than the pivot are on

the left, and elements larger are on the right.

3. Recursive Sort: Recursively apply the same procedure to the subarrays on

either side of the pivot.

Quick Sort Example Code in Java:

import java.util.Arrays;

public class QuickSortExample {

// Main quicksort method

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1); // Recursively sort the left side

quickSort(arr, pivotIndex + 1, high); // Recursively sort the right side

}

}

// Partition method that rearranges elements

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high]; // Select the pivot (last element)

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) { // If current element is smaller or equal to pivot

i++;

swap(arr, i, j); // Swap it with the smaller element

}

}

swap(arr, i + 1, high); // Place pivot in the correct position

return i + 1; // Return the index of the pivot

}

// Utility method to swap two elements

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void main(String[] args) {

int[] arr = {10, 7, 8, 9, 1, 5};

System.out.println("Unsorted array: " + Arrays.toString(arr));

quickSort(arr, 0, arr.length - 1);

System.out.println("Sorted array: " + Arrays.toString(arr));

}

}

Output:

Unsorted array: [10, 7, 8, 9, 1, 5]

Sorted array: [1, 5, 7, 8, 9, 10]

Merge Sort

Merge Sort is another divide-and-conquer algorithm that recursively divides an

array into two halves, sorts each half, and then merges the two halves together in a

sorted manner.

Key Characteristics:

1. Divide-and-Conquer: The array is repeatedly divided into two halves.

2. Stable Sort: The relative order of equal elements is preserved.

3. Not In-Place: Requires extra space proportional to the size of the input array

for merging.

4. Time Complexity:

o Best case: O(n log n)

o Average case: O(n log n)

o Worst case: O(n log n) (same for all cases)

Merge Sort Algorithm:

1. Divide: Split the array into two halves.

2. Conquer: Recursively sort each half.

3. Merge: Merge the two sorted halves into a single sorted array.

Merge Sort Example Code in Java:

import java.util.Arrays;

public class MergeSortExample {

// Main merge sort method

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

// Recursively sort the left and right halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

// Merge method to combine two sorted subarrays

private static void merge(int[] arr, int left, int mid, int right) {

// Find the sizes of the two subarrays to be merged

int n1 = mid - left + 1;

int n2 = right - mid;

// Temporary arrays to hold the subarrays

int[] leftArr = new int[n1];

int[] rightArr = new int[n2];

// Copy the data into the temporary arrays

for (int i = 0; i < n1; ++i) {

leftArr[i] = arr[left + i];

}

for (int j = 0; j < n2; ++j) {

rightArr[j] = arr[mid + 1 + j];

}

// Initial indexes for left, right, and merged arrays

int i = 0, j = 0, k = left;

// Merge the temporary arrays back into the original array

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j];

j++;

}

k++;

}

// Copy the remaining elements from the left subarray

while (i < n1) {

arr[k] = leftArr[i];

i++;

k++;

}

// Copy the remaining elements from the right subarray

while (j < n2) {

arr[k] = rightArr[j];

j++;

k++;

}

}

public static void main(String[] args) {

int[] arr = {12, 11, 13, 5, 6, 7};

System.out.println("Unsorted array: " + Arrays.toString(arr));

mergeSort(arr, 0, arr.length - 1);

System.out.println("Sorted array: " + Arrays.toString(arr));

}

}

Output:

Unsorted array: [12, 11, 13, 5, 6, 7]

Sorted array: [5, 6, 7, 11, 12, 13]

Tree Traversals: Inorder, Preorder, and Postorder

Tree traversal is the process of visiting (checking or updating) each node in a tree

data structure, exactly once. There are three primary types of depth-first tree

traversals: Inorder, Preorder, and Postorder. Each method follows a different

strategy for visiting nodes.

1. Inorder Traversal

In Inorder traversal, the nodes are visited in this order:

1. Visit the left subtree.

2. Visit the root node.

3. Visit the right subtree.

For a binary search tree (BST), an Inorder traversal visits the nodes in ascending

order.

Example:

1

/ \

2 3

/ \

4 5

Inorder Traversal: 4 → 2 → 5 → 1 → 3

Inorder Traversal Java Code:

class Node {

int data;

Node left, right;

public Node(int data) {

this.data = data;

left = right = null;

}

}

public class InorderTraversal {

// Inorder traversal method

void inorder(Node node) {

if (node == null) {

return;

}

// Traverse the left subtree

inorder(node.left);

// Visit the root node

System.out.print(node.data + " ");

// Traverse the right subtree

inorder(node.right);

}

public static void main(String[] args) {

InorderTraversal tree = new InorderTraversal();

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

System.out.print("Inorder traversal: ");

tree.inorder(root); // Output: 4 2 5 1 3

}

}

2. Preorder Traversal

In Preorder traversal, the nodes are visited in this order:

1. Visit the root node.

2. Visit the left subtree.

3. Visit the right subtree.

Preorder traversal is used to create a copy of the tree and is also used in prefix

expression notation.

Example:

1

/ \

2 3

/ \

4 5

Preorder Traversal: 1 → 2 → 4 → 5 → 3

Preorder Traversal Java Code:

class PreorderTraversal {

// Preorder traversal method

void preorder(Node node) {

if (node == null) {

return;

}

// Visit the root node

System.out.print(node.data + " ");

// Traverse the left subtree

preorder(node.left);

// Traverse the right subtree

preorder(node.right);

}

public static void main(String[] args) {

PreorderTraversal tree = new PreorderTraversal();

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

System.out.print("Preorder traversal: ");

tree.preorder(root); // Output: 1 2 4 5 3

}

}

3. Postorder Traversal

In Postorder traversal, the nodes are visited in this order:

1. Visit the left subtree.

2. Visit the right subtree.

3. Visit the root node.

Postorder traversal is used to delete the tree and is also useful for evaluating

postfix expressions.

Example:

1

/ \

2 3

/ \

4 5

Postorder Traversal: 4 → 5 → 2 → 3 → 1

Postorder Traversal Java Code:

class PostorderTraversal {

// Postorder traversal method

void postorder(Node node) {

if (node == null) {

return;

}

// Traverse the left subtree

postorder(node.left);

// Traverse the right subtree

postorder(node.right);

// Visit the root node

System.out.print(node.data + " ");

}

public static void main(String[] args) {

PostorderTraversal tree = new PostorderTraversal();

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

System.out.print("Postorder traversal: ");

tree.postorder(root); // Output: 4 5 2 3 1

}

}

Binary Tree Overview

A binary tree is a tree data structure where each node has at most two children,

referred to as the left child and the right child. Binary trees are used in various

applications, including searching, sorting, and hierarchical data storage.

Types of Binary Trees:

1. Full Binary Tree: Every node has either 0 or 2 children.

2. Complete Binary Tree: All levels are completely filled except possibly the

last, which is filled from left to right.

3. Perfect Binary Tree: A binary tree where all the internal nodes have two

children, and all leaves are at the same level.

4. Balanced Binary Tree: The height of the tree is balanced so that it grows

logarithmically with the number of nodes.

Example of a Binary Tree:

50

/ \

30 70

/ \ / \

20 40 60 80

In this tree, the root node is 50, and it has left and right subtrees rooted at 30 and

70, respectively.

Code for Deleting a Node in a Binary Search Tree (BST)

class Node {

int data;

Node left, right;

public Node(int data) {

this.data = data;

left = right = null;

}

}

public class BinaryTree {

Node root;

// Function to delete a node in the binary search tree

Node deleteNode(Node root, int key) {

if (root == null) {

return root; // base case: the tree is empty

}

// Recur down the tree

if (key < root.data) {

root.left = deleteNode(root.left, key);

} else if (key > root.data) {

root.right = deleteNode(root.right, key);

} else {

// Node with only one child or no child

if (root.left == null) {

return root.right;

} else if (root.right == null) {

return root.left;

}

// Node with two children: Get the inorder successor (smallest in the right

subtree)

root.data = minValue(root.right);

// Delete the inorder successor

root.right = deleteNode(root.right, root.data);

}

return root;

}

// Utility function to find the minimum value node in the tree

int minValue(Node root) {

int minValue = root.data;

while (root.left != null) {

minValue = root.left.data;

root = root.left;

}

return minValue;

}

// Inorder traversal of the tree (used to print the tree)

void inorder(Node root) {

if (root != null) {

inorder(root.left);

System.out.print(root.data + " ");

inorder(root.right);

}

}

public static void main(String[] args) {

BinaryTree tree = new BinaryTree();

tree.root = new Node(50);

tree.root.left = new Node(30);

tree.root.right = new Node(70);

tree.root.left.left = new Node(20);

tree.root.left.right = new Node(40);

tree.root.right.left = new Node(60);

tree.root.right.right = new Node(80);

System.out.println("Inorder traversal of the tree:");

tree.inorder(tree.root);

System.out.println();

System.out.println("\nDelete node 20 (a leaf node):");

tree.root = tree.deleteNode(tree.root, 20);

tree.inorder(tree.root);

System.out.println();

System.out.println("\nDelete node 30 (a node with one child):");

tree.root = tree.deleteNode(tree.root, 30);

tree.inorder(tree.root);

System.out.println();

System.out.println("\nDelete node 50 (a node with two children):");

tree.root = tree.deleteNode(tree.root, 50);

tree.inorder(tree.root);

System.out.println();

}

}

Output

Inorder traversal of the tree:

20 30 40 50 60 70 80

Delete node 20 (a leaf node):

30 40 50 60 70 80

Delete node 30 (a node with one child):

40 50 60 70 80

Delete node 50 (a node with two children):

40 60 70 80

—----------------------------------------------------------------------

Dynamic programming (DP) is a method used in computer science and

mathematics to solve complex problems by breaking them down into simpler sub

problems. It is particularly useful for optimization problems where the solution

involves making decisions at various stages.

The key idea of dynamic programming is to store the solutions of sub problems to

avoid recomputation, allowing the algorithm to work more efficiently. DP typically

follows two main principles:

1. Optimal Sub structure: This property means that the solution to a problem can

be constructed from the solutions of its sub problems. If a problem has this

property, it is possible to recursively break it down into smaller problems.

2. Overlapping Sub problems: DP takes advantage of solving the same sub

problems multiple times. By storing the results of sub problems (memoization or

tabulation), DP avoids redundant computations.

Steps in Dynamic Programming:

1. Define the sub problems: Determine the smaller sub problems that will

contribute to solving the larger problem.

2. Recurrence relation: Formulate a recursive relationship between the solutions of

the subproblems.

3. Memoization or tabulation:

-Memoization: Store the results of sub problems in memory (often in a hash map

or array) and retrieve them when needed.

-Tabulation: Build a table (usually a 2D array) in a bottom-up fashion to store

solutions for all sub problems.

4. Solve the base cases: Start from the simplest sub problems (base cases) and

build up to the solution of the original problem.

Applications:

-Algorithm design: Shortest path problems (e.g., Bellman-Ford algorithm), edit

distance, and matrix chain multiplication.

- Resource management: In operations research for resource allocation and

optimization.

-Artificial intelligence: Used in game theory, reinforcement learning, and decision-

making processes.

Program to Generate Fibonacci Series:

import java.util.Scanner;

public class FibonacciSeries {

// Function to print Fibonacci series up to n terms

public static void printFibonacci(int n) {

// Initialize the first two terms of the Fibonacci series

int num1 = 0, num2 = 1;

// Print the first term

System.out.print(num1 + " ");

// Print the rest of the terms

for (int i = 1; i < n; i++) {

System.out.print(num2 + " ");

// Calculate the next term by adding the previous two terms

int nextTerm = num1 + num2;

// Update num1 and num2 for the next iteration

num1 = num2;

num2 = nextTerm;

}

}

public static void main(String[] args) {

// Create a Scanner object for user input

Scanner scanner = new Scanner(System.in);

// Ask the user for the number of terms

System.out.print("Enter the number of terms for Fibonacci series: ");

int n = scanner.nextInt();

// Call the printFibonacci function to print the series

System.out.println("Fibonacci Series up to " + n + " terms:");

printFibonacci(n);

}

}

Sample Output:

Enter the number of terms for Fibonacci series: 8

Fibonacci Series up to 8 terms:

0 1 1 2 3 5 8 13

Coin Change Problem:

import java.util.Arrays;

public class CoinChange {

// Function to find the minimum number of coins required to make a given

amount

public static int coinChange(int[] coins, int amount) {

// Create an array to store the minimum number of coins needed for each

amount from 0 to 'amount'

int[] dp = new int[amount + 1];

// Initialize dp array with a large value (amount + 1), which represents an

unreachable state

Arrays.fill(dp, amount + 1);

// Base case: To make an amount of 0, we need 0 coins

dp[0] = 0;

// Iterate over all amounts from 1 to the target amount

for (int i = 1; i <= amount; i++) {

// Check each coin denomination

for (int coin : coins) {

// If the coin is less than or equal to the current amount

if (i - coin >= 0) {

// Update dp[i] to be the minimum of its current value or 1 + dp[i -

coin]

dp[i] = Math.min(dp[i], dp[i - coin] + 1);

}

}

}

// If dp[amount] is still amount + 1, it means the amount can't be made with

the given coins

return dp[amount] > amount ? -1 : dp[amount];

}

public static void main(String[] args) {

// Example array of coin denominations

int[] coins = {1, 2, 5};

// Example target amount

int amount = 11;

// Call the coinChange function

int result = coinChange(coins, amount);

// Print the result

if (result == -1) {

System.out.println("It's not possible to make the amount " + amount + "

with the given coins.");

} else {

System.out.println("The minimum number of coins required to make the

amount " + amount + " is: " + result);

}

}

}

Sample Output:

The minimum number of coins required to make the amount 11 is: 3

Explanation: For amount = 11 and coins {1, 2, 5}, the minimum number of coins

required is 3. This can be achieved with three coins: 5 + 5 + 1.

Lambda Expressions in Java

Lambda expressions are a key feature introduced in Java 8, which allow for writing

concise, functional-style code. A lambda expression is a way to define

anonymous functions that can be passed around or used directly in the code

without defining a new method.

General Syntax of a Lambda Expression:

(parameters) -> {expression or code block}

Example Using Lambda Expression:

import java.util.Arrays;

import java.util.List;

import java.util.Collections;

public class LambdaExample {

public static void main(String[] args) {

// Example 1: Sorting a list using lambda expression

List<String> names = Arrays.asList("John", "Anna", "Mark", "Sophia");

// Sorting using lambda expression

Collections.sort(names, (String a, String b) -> b.compareTo(a));

// Output the sorted list

System.out.println("Sorted names in reverse order: " + names);

// Example 2: Using lambda expression for Runnable

Runnable runnable = () -> System.out.println("Thread is running using

Lambda!");

// Create a thread and pass the lambda Runnable

Thread thread = new Thread(runnable);

thread.start();

// Example 3: Using lambda to perform an action on each element of a list

names.forEach(name -> System.out.println("Name: " + name));

// Example 4: Lambda expression for a simple mathematical operation (addition)

MathOperation addition = (a, b) -> a + b;

System.out.println("10 + 5 = " + addition.operate(10, 5));

}

// Functional interface for mathematical operations

interface MathOperation {

int operate(int a, int b);

}

}

Sample Output for Above Program:

Sorted names in reverse order: [Sophia, Mark, John, Anna]

Thread is running using Lambda!

Name: Sophia

Name: Mark

Name: John

Name: Anna

10 + 5 = 15

Programs using Strings:

program to reverse the words in a string

public class ReverseWords {

// Function to reverse the words in a given string

public static String reverseWords(String input) {

// Split the string into words using space as the delimiter

String[] words = input.split(" ");

// StringBuilder to store the reversed string

StringBuilder reversedString = new StringBuilder();

// Traverse the array of words in reverse order

for (int i = words.length - 1; i >= 0; i--) {

// Append each word to the reversedString

reversedString.append(words[i]);

// Add a space after each word except the last one

if (i > 0) {

reversedString.append(" ");

}

}

// Convert StringBuilder to String and return the result

return reversedString.toString();

}

public static void main(String[] args) {

// Example input string

String input = "Hello World from Java";

// Call the reverseWords function

String result = reverseWords(input);

// Print the original and reversed strings

System.out.println("Original String: " + input);

System.out.println("Reversed Words String: " + result);

}

}

Sample Output:

Original String: Hello World from Java

Reversed Words String: Java from World Hello

Program to check if two strings are anagrams

import java.util.Arrays;

public class AnagramChecker {

// Function to check if two strings are anagrams

public static boolean areAnagrams(String str1, String str2) {

// Remove any whitespace and convert strings to lowercase for case-

insensitive comparison

str1 = str1.replaceAll("\\s", "").toLowerCase();

str2 = str2.replaceAll("\\s", "").toLowerCase();

// If the lengths of the strings are not equal, they cannot be anagrams

if (str1.length() != str2.length()) {

return false;

}

// Convert both strings to character arrays

char[] array1 = str1.toCharArray();

char[] array2 = str2.toCharArray();

// Sort the character arrays

Arrays.sort(array1);

Arrays.sort(array2);

// Compare the sorted arrays; if they are equal, the strings are anagrams

return Arrays.equals(array1, array2);

}

public static void main(String[] args) {

// Example input strings

String str1 = "Listen";

String str2 = "Silent";

// Call the areAnagrams function

boolean result = areAnagrams(str1, str2);

// Print the results

if (result) {

System.out.println(str1 + " and " + str2 + " are anagrams.");

} else {

System.out.println(str1 + " and " + str2 + " are not anagrams.");

}

}

}

Sample Output:

Listen and Silent are anagrams

—--------------------------------------------------------------------------------------------------

Program to find the longest palindrome substring

public class LongestPalindromicSubstring {

// Function to find the longest palindromic substring

public static String longestPalindromicSubstring(String s) {

int n = s.length();

if (n == 0) return "";

// dp[i][j] will be 'true' if substring s[i..j] is a palindrome

boolean[][] dp = new boolean[n][n];

int maxLength = 1; // Stores the length of the longest palindrome

int start = 0; // Stores the starting index of the longest palindrome

// All substrings of length 1 are palindromes

for (int i = 0; i < n; i++) {

dp[i][i] = true;

}

// Check for palindromes of length 2

for (int i = 0; i < n - 1; i++) {

if (s.charAt(i) == s.charAt(i + 1)) {

dp[i][i + 1] = true;

start = i;

maxLength = 2;

}

}

// Check for palindromes of length greater than 2

for (int length = 3; length <= n; length++) {

for (int i = 0; i <= n - length; i++) {

int j = i + length - 1; // Ending index of current substring

// If the current substring is palindrome

if (dp[i + 1][j - 1] && s.charAt(i) == s.charAt(j)) {

dp[i][j] = true;

if (length > maxLength) {

start = i;

maxLength = length;

}

}

}

}

// Return the longest palindromic substring

return s.substring(start, start + maxLength);

}

public static void main(String[] args) {

// Example string

String str = "babad";

// Find and print the longest palindromic substring

System.out.println("The longest palindromic substring is: " + longestPalindromicSubstring(str));

}

}

Sample Output:

The longest palindromic substring is: bab

Hashing and Hash Table Collision Techniques

Hashing is a process of mapping data to a fixed-size value, called a hash code or hash value,

typically for faster retrieval. A hash function takes an input (or "key") and returns an integer,

which is then used to index into a hash table.

Hashing is commonly used in data structures like hash tables, where data (often key-value pairs)

is stored and retrieved in constant time O(1), on average.

A hash table (or hash map) is a data structure that stores key-value pairs. The keys are processed

using a hash function to generate a unique index (called the hash index) where the

corresponding value is stored. Hash tables are known for their fast insertion, deletion, and lookup

operations.

Example of Hash Table Operations:

Insert: Adds a new key-value pair to the table.

Delete: Removes a key-value pair based on the key.

Search/Lookup: Finds the value corresponding to a given key.

A hash function is used to compute the index at which the key-value pair will be stored. A good

hash function minimizes collisions and distributes the keys uniformly across the table.

Properties of a good hash function:

Deterministic: Same input should always produce the same output.

Fast: Should be computationally efficient.

Uniform Distribution: Should spread keys uniformly across the hash table.

Minimize Collisions: Should minimize the chances of two different keys producing the

same hash index.

Hash Collisions

A collision occurs when two different keys produce the same hash index. Since a hash table

needs to store data at unique indices, collisions need to be handled to ensure correct data

retrieval.

Collision Resolution Techniques

There are two primary categories for resolving hash collisions:

Open Addressing: All data is stored in the hash table itself, and collisions are resolved

by finding another available spot within the table.

Chaining: Each index of the hash table points to a linked list or another data structure

that stores multiple key-value pairs in case of collisions.

Program that demonstrates the basics of a hash table with chaining to handle collisions:

import java.util.LinkedList;

class HashTable {

private LinkedList<Entry>[] table;

private int size;

// Hash table constructor

public HashTable(int size) {

this.size = size;

table = new LinkedList[size];

for (int i = 0; i < size; i++) {

table[i] = new LinkedList<>();

}

}

// Hash function

private int getHashIndex(String key) {

return key.hashCode() % size;

}

// Insert key-value pair

public void insert(String key, String value) {

int index = getHashIndex(key);

table[index].add(new Entry(key, value));

}

// Search for a value by key

public String search(String key) {

int index = getHashIndex(key);

for (Entry entry : table[index]) {

if (entry.key.equals(key)) {

return entry.value;

}

}

return null; // Key not found

}

// Entry class to store key-value pairs

private static class Entry {

String key;

String value;

Entry(String key, String value) {

this.key = key;

this.value = value;

}

}

// Main method to demonstrate usage

public static void main(String[] args) {

HashTable hashTable = new HashTable(10);

// Inserting values

hashTable.insert("apple", "fruit");

hashTable.insert("carrot", "vegetable");

// Searching for values

System.out.println("Search for 'apple': " + hashTable.search("apple")); // Output: fruit

System.out.println("Search for 'banana': " + hashTable.search("banana")); // Output: null

}

}

Program to perform binary search in a 2D matrix

To perform a binary search in a matrix where each row is sorted but the columns are not necessarily

sorted, we can follow a modified search approach. Since only rows are sorted, we will perform a linear

search for each row instead of using binary search across the columns.

public class MatrixBinarySearch {

// Method to perform binary search on a specific row

public static boolean binarySearchRow(int[] row, int target) {

int left = 0, right = row.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (row[mid] == target) {

return true; // Target found

} else if (row[mid] < target) {

left = mid + 1; // Search right half

} else {

right = mid - 1; // Search left half

}

}

return false; // Target not found in this row

}

// Method to search the target in the entire matrix

public static boolean searchInMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0) {

return false;

}

for (int i = 0; i < matrix.length; i++) {

if (binarySearchRow(matrix[i], target)) {

return true; // Target found in the current row

}

}

return false; // Target not found in any row

}

public static void main(String[] args) {

int[][] matrix = {

{1, 3, 5, 7},

{21, 25, 29, 30},

{10, 19, 24, 29},

{17, 32, 43, 50}

};

// Test with various target values

int[] targets = {5, 29, 10, 32, 100};

for (int target : targets) {

if (searchInMatrix(matrix, target)) {

System.out.println("Target " + target + " found in the matrix.");

} else {

System.out.println("Target " + target + " not found in the matrix.");

}

}

}

}

Output:

Target 5 found in the matrix.

Target 29 found in the matrix.

Target 10 found in the matrix.

Target 32 found in the matrix.

Target 100 not found in the matrix.

Problem Statement: Given an array, print all the elements which are leaders. A Leader is an element

that is greater than all of the elements on its right side in the array.

To solve the problem of finding all the leaders in an array, we can follow these steps:

1. Traverse the array from right to left.

2. Keep track of the maximum element encountered so far (starting with the last element as

it is always a leader).

3. If the current element is greater than the maximum element, it is a leader. Update the

maximum element accordingly.

4. Store the leaders in a list and print them.

Code:

import java.util.ArrayList;

import java.util.List;

public class LeadersInArray {

// Function to find and print all leaders in the array

public static void printLeaders(int[] arr) {

int n = arr.length;

List<Integer> leaders = new ArrayList<>();

// Start from the last element

int maxFromRight = arr[n - 1];

leaders.add(maxFromRight); // The last element is always a leader

// Traverse the array from right to left

for (int i = n - 2; i >= 0; i--) {

if (arr[i] > maxFromRight) {

leaders.add(arr[i]); // Found a leader

maxFromRight = arr[i]; // Update the maximum

}

}

// Print leaders in the order they appeared in the array

System.out.println("Leaders in the array: ");

for (int i = leaders.size() - 1; i >= 0; i--) {

System.out.print(leaders.get(i) + " ");

}

System.out.println(); // For a new line

}

public static void main(String[] args) {

// Example array

int[] arr = {16, 17, 4, 3, 5, 2};

// Find and print leaders

printLeaders(arr);

}

}

Output:

For the example input array {16, 17, 4, 3, 5, 2}, the output will be:

Leaders in the array:

17 5 2

—-------------------------------------------------------------------------------------

WEEK-4

Introduction to SQL

SQL (Structured Query Language) is a standardized programming language

used to manage and manipulate relational databases. It allows users to perform

various operations such as querying data, updating records, creating and modifying

schemas, and controlling access to data.

SQL Categories

SQL can be divided into several categories based on its functionality. The main

categories are:

1. DDL (Data Definition Language)

2. DML (Data Manipulation Language)

3. DQL (Data Query Language)

1. DDL (Data Definition Language)

DDL is used to define and manage all database objects. Common commands

include:

CREATE: Creates a new table or database.

ALTER: Modifies an existing database object.

DROP: Deletes an entire table or database.

TRUNCATE: Removes all records from a table but keeps the structure.

2. DML (Data Manipulation Language)

DML is used to manipulate data stored in the database. Common commands

include:

INSERT: Adds new records to a table.

UPDATE: Modifies existing records in a table.

DELETE: Removes records from a table.

3. DQL (Data Query Language)

DQL is used to query data from a database. The primary command is:

SELECT: Retrieves data from one or more tables.

SQL Joins

Joins allow you to query data from multiple tables and are fundamental for

working with relational databases. The main types of joins are:

1. INNER JOIN

Description: Returns only the rows that have matching values in both tables.

Syntax:

SELECT columns

FROM table1

INNER JOIN table2 ON table1.common\_column = table2.common\_column;

2. LEFT JOIN (or LEFT OUTER JOIN)

Description: Returns all rows from the left table and the matched rows from

the right table. If no match is found, NULL values are returned for the right

table’s columns.

Syntax:

SELECT columns

FROM table1

LEFT JOIN table2 ON table1.common\_column = table2.common\_column;

3. RIGHT JOIN (or RIGHT OUTER JOIN)

Description: Returns all rows from the right table and the matched rows

from the left table. If no match is found, NULL values are returned for the

left table’s columns.

Syntax:

SELECT columns

FROM table1

RIGHT JOIN table2 ON table1.common\_column = table2.common\_column;

4. FULL JOIN (or FULL OUTER JOIN)

Description: Returns all rows when there is a match in either the left or right

table. If there is no match, NULL values are returned for missing matches in

either table.

Syntax:

SELECT columns

FROM table1

FULL JOIN table2 ON table1.common\_column = table2.common\_column;

5. CROSS JOIN

Description: Produces a Cartesian product of two tables, returning all

possible combinations of rows from both tables.

Syntax:

SELECT columns

FROM table1

CROSS JOIN TABLE1

—------------------------------------------------------------------------------------------

Employees Table

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

department VARCHAR(50),

salary DECIMAL(10, 2),

hire\_date DATE

);

INSERT INTO employees (first\_name, last\_name, department, salary, hire\_date)

VALUES

('Alice', 'Smith', 'Sales', 70000, '2021-05-01'),

('Bob', 'Johnson', 'Sales', 60000, '2019-03-15'),

('Charlie', 'Williams', 'Engineering', 90000, '2022-02-10'),

('David', 'Jones', 'Engineering', 95000, '2023-01-20'),

('Eva', 'Brown', 'HR', 50000, '2020-06-01'),

('Frank', 'Garcia', 'HR', 52000, '2022-03-20'),

('Grace', 'Martinez', 'Sales', 58000, '2023-04-01');

Projects Table

CREATE TABLE projects (

project\_id SERIAL PRIMARY KEY,

project\_name VARCHAR(100),

start\_date DATE,

end\_date DATE

);

INSERT INTO projects (project\_name, start\_date, end\_date) VALUES

('Project A', '2021-01-01', '2022-01-01'),

('Project B', '2020-05-01', '2021-12-01'),

('Project C', '2023-01-01', '2024-01-01');

Employee projects Table

CREATE TABLE employee\_projects (

employee\_id INT REFERENCES employees(employee\_id),

project\_id INT REFERENCES projects(project\_id),

role VARCHAR(50)

);

INSERT INTO employee\_projects (employee\_id, project\_id, role) VALUES

(1, 1, 'Manager'),

(2, 1, 'Contributor'),

(3, 2, 'Manager'),

(4, 3, 'Contributor'),

(5, 1, 'Contributor'),

(6, 3, 'Manager'),

(7, 2, 'Contributor');

1. Calculate the average salary for each department, but only include

employees hired in the last 3 years.

SELECT

department,

AVG(salary) as avg\_salary

FROM

employees

WHERE

hire\_date > CURRENT\_DATE - INTERVAL '3 years'

GROUP BY

department

ORDER BY

avg\_salary DESC;

Output:

| department | avg\_salary |

|--------------|------------|

| Engineering | 95000.00 |

| Sales | 63999.99 |

| HR | 52000.00 |

2. Create a pivot table showing the count of employees in each department,

with columns for different salary ranges.

SELECT

department,

COUNT(CASE WHEN salary < 65000 THEN 1 END) as "< 65000",

COUNT(CASE WHEN salary BETWEEN 65000 AND 75000 THEN 1 END)

as "65000-75000",

COUNT(CASE WHEN salary > 75000 THEN 1 END) as "> 75000"

FROM

employees

GROUP BY

department

ORDER BY

department;

Output:

| department | < 65000 | 65000-75000 | > 75000 |

|--------------|---------|--------------|---------|

| Engineering | 0 | 0 | 2 |

| HR | 0 | 1 | 1 |

| Sales | 1 | 1 | 1 |

3. Find the employee(s) with the highest salary in their respective

departments, who are also working on the longest-running project.

WITH dept\_max\_salaries AS (

SELECT

department,

MAX(salary) as max\_salary

FROM

employees

GROUP BY

department

),

longest\_project AS (

SELECT

project\_id,

end\_date - start\_date as duration

FROM

projects

ORDER BY

duration DESC

LIMIT 1

)

SELECT

e.employee\_id,

e.first\_name,

e.last\_name,

e.department,

e.salary,

p.project\_name

FROM

employees e

JOIN

dept\_max\_salaries dms ON e.department = dms.department AND e.salary =

dms.max\_salary

JOIN

employee\_projects ep ON e.employee\_id = ep.employee\_id

JOIN

projects p ON ep.project\_id = p.project\_id

JOIN

longest\_project lp ON p.project\_id = lp.project\_id;

Output:

| employee\_id | first\_name | last\_name | department | salary | project\_name |

|-------------|------------|-----------|--------------|--------|---------------|

| 4 | David | Jones | Engineering | 95000 | Project C |

4. Calculate the percentage of each department's salary compared to the total

salary of the company.

WITH total\_salary AS (

SELECT SUM(salary) as company\_total

FROM employees

)

SELECT

department,

SUM(salary) as dept\_total,

(SUM(salary) \* 100.0 / (SELECT company\_total FROM total\_salary)) as

percentage

FROM

employees

GROUP BY

department

ORDER BY

percentage DESC;

Output:

| department | dept\_total | percentage |

|--------------|------------|------------|

| Engineering | 185000.00 | 54.35 |

| Sales | 185000.00 | 40.35 |

| HR | 102000.00 | 5.30 |

5. Identify employees who have a higher salary than their department's

average, and show by what percentage their salary exceeds the average.

WITH dept\_avg\_salaries AS (

SELECT

department,

AVG(salary) as avg\_salary

FROM

employees

GROUP BY

department

)

SELECT

e.employee\_id,

e.first\_name,

e.last\_name,

e.department,

e.salary,

das.avg\_salary,

((e.salary - das.avg\_salary) \* 100.0 / das.avg\_salary) as percent\_above\_avg

FROM

employees e

JOIN

dept\_avg\_salaries das ON e.department = das.department

WHERE

e.salary > das.avg\_salary

ORDER BY

percent\_above\_avg DESC;

Output:

| employee\_id | first\_name | last\_name | department | salary | avg\_salary |

percent\_above\_avg |

|-------------|------------|-----------|--------------|--------|------------|--------------------|

| 4 | David | Jones | Engineering | 95000 | 92500.00 | 5.41 |

| 1 | Alice | Smith | Sales | 70000 | 64000.00 | 9.37 |

6. Create a query that shows a hierarchical view of employees and their

projects, with multiple levels of projects if an employee is in more than one.

WITH RECURSIVE employee\_projects\_hierarchy AS (

SELECT

e.employee\_id,

e.first\_name,

e.last\_name,

p.project\_id,

p.project\_name,

ep.role,

1 as level,

CAST(p.project\_name AS VARCHAR(1000)) as project\_path

FROM

employees e

JOIN

employee\_projects ep ON e.employee\_id = ep.employee\_id

JOIN

projects p ON ep.project\_id = p.project\_id

UNION ALL

SELECT

e.employee\_id,

e.first\_name,

e.last\_name,

p.project\_id,

p.project\_name,

ep.role,

eph.level + 1,

CAST(eph.project\_path || ' > ' || p.project\_name AS VARCHAR(1000))

FROM

employees e

JOIN

employee\_projects ep ON e.employee\_id = ep.employee\_id

JOIN

projects p ON ep.project\_id = p.project\_id

JOIN

employee\_projects\_hierarchy eph ON e.employee\_id = eph.employee\_id

WHERE

p.project\_id > eph.project\_id

)

SELECT

employee\_id,

first\_name,

last\_name,

project\_id,

project\_name,

role,

level,

project\_path

FROM

employee\_projects\_hierarchy

ORDER BY

employee\_id, level, project\_id;

Output:

| employee\_id | first\_name | last\_name | project\_id | project\_name | role | level |

project\_path |

|-------------|------------|-----------|------------|---------------|-------------|-------|------------

---|

| 1 | Alice | Smith | 1 | Project A | Manager | 1 | Project A

|

| 2 | Bob | Johnson | 1 | Project A | Contributor | 1 | Project A

|

| 3 | Charlie | Williams | 2 | Project B | Manager | 1 | Project B

|

| 4 | David | Jones | 3 | Project C | Contributor | 1 | Project C

|

| 5 | Eva | Brown | 1 | Project A | Contributor | 1 | Project A

|

| 6 | Frank | Garcia | 3 | Project C | Manager | 1 | Project C

|

| 7 | Grace | Martinez | 2 | Project B | Contributor | 1 | Project B

|

7. Implement a query to find the "Kevin Bacon Number" equivalent for

projects.

WITH RECURSIVE project\_connections AS (

-- Base case: direct connections

SELECT

ep1.employee\_id as employee1,

ep2.employee\_id as employee2,

1 as connection\_level

FROM

employee\_projects ep1

JOIN

employee\_projects ep2 ON ep1.project\_id = ep2.project\_id AND

ep1.employee\_id < ep2.employee\_id

UNION

-- Recursive case: indirect connections

SELECT

pc.employee1,

ep.employee\_id as employee2,

pc.connection\_level + 1

FROM

project\_connections pc

JOIN

employee\_projects ep ON pc.employee2 = ep.employee\_id

WHERE

pc.employee1 < ep.employee\_id AND pc.connection\_level < 6

)

SELECT

e1.first\_name || ' ' || e1.last\_name as employee1,

e2.first\_name || ' ' || e2.last\_name as employee2,

MIN(connection\_level) as shortest\_connection

FROM

project\_connections pc

JOIN

employees e1 ON pc.employee1 = e1.employee\_id

JOIN

employees e2 ON pc.employee2 = e2.employee\_id

GROUP BY

e1.employee\_id, e1.first\_name, e1.last\_name,

e2.employee\_id, e2.first\_name, e2.last\_name

ORDER BY

e1.last\_name, e1.first\_name, shortest\_connection;

Output:

| employee1 | employee2 | shortest\_connection |

|--------------------|-------------------|---------------------|

| Alice Smith | Bob Johnson | 1 |

| Eva Brown | Frank Garcia | 1 |

—-----------------------------------------------------------------------------------------------------------------

Primary Key

Definition: A primary key is a unique identifier for a record in a table. No

two records can have the same primary key value.

Characteristics:

o Must contain unique values.

o Cannot contain NULL values.

o Each table should have a primary key.

Foreign Key

Definition: A foreign key is a field (or collection of fields) in one table that

refers to the primary key in another table.

Purpose: It establishes a relationship between the two tables, ensuring

referential integrity.

Characteristics:

o Can contain duplicate values.

o Can have NULL values (unless explicitly stated otherwise).

Normal Forms

Normalization is a process to organize data to reduce redundancy and improve data

integrity. The most common normal forms are:

1. First Normal Form (1NF):

o Ensure that each column contains atomic values (no repeating groups

or arrays).

o Each entry in a column must be of the same data type.

2. Second Normal Form (2NF):

o Must be in 1NF.

o All non-key attributes must be fully functionally dependent on the

primary key (no partial dependency).

3. Third Normal Form (3NF):

o Must be in 2NF.

o No transitive dependency exists (non-key attributes must not depend

on other non-key attributes).

Entity-Relationship (E-R) Diagram

An E-R diagram visually represents the relationships between entities in a

database.

Entities: Objects or concepts (e.g., Customer, Order).

Attributes: Properties of entities (e.g., CustomerID, OrderDate).

Relationships:

o One-to-One (1:1): A single entity in Table A relates to a single entity

in Table B.

o One-to-Many (1

): A single entity in Table A can relate to multiple entities in Table B.

o Many-to-Many (M

): Entities in Table A can relate to multiple entities in Table B and

vice versa.

Example E-R Diagram Components

1. Entities:

o Customer (CustomerID, Name, Email)

o Order (OrderID, OrderDate, CustomerID)

2. Relationships:

o Customer (1) — (N) Order

LAG and LEAD Functions

These functions are used to access data from a previous or subsequent row in the

result set without the need for self-joins.

LAG: Returns the value from a previous row in the result set.

LEAD: Returns the value from a subsequent row in the result set.

Syntax

LAG(column\_name, offset, default\_value) OVER (PARTITION BY

partition\_column ORDER BY order\_column)

LEAD(column\_name, offset, default\_value) OVER (PARTITION BY

partition\_column ORDER BY order\_column)

PARTITION BY

The PARTITION BY clause is used to divide the result set into smaller groups

(partitions) where the window functions can be applied independently.

ORDER BY

The ORDER BY clause is used to sort the result set by one or more columns. It can

be used both in standard queries and within the OVER() clause of window

functions.

Window Functions

Window functions perform calculations across a set of table rows that are related to

the current row. Unlike aggregate functions, they do not group rows into a single

output row, allowing access to both grouped and individual row data.

Syntax

function\_name(column\_name) OVER (

[PARTITION BY partition\_column]

[ORDER BY order\_column]

)

Common Window Functions

1. ROW\_NUMBER(): Assigns a unique sequential integer to rows within a

partition.

2. RANK(): Similar to ROW\_NUMBER(), but assigns the same rank to tied

rows, skipping the subsequent rank(s).

3. DENSE\_RANK(): Like RANK(), but does not skip ranks after ties.

4. SUM(), AVG(), etc.: Aggregate functions can be used as window functions.

Common Table Expressions (CTEs)

A CTE provides a way to write auxiliary statements for use within a larger SQL

query. It improves readability and can be referenced multiple times in a query.

Syntax

WITH cte\_name AS (

SELECT columns

FROM table

WHERE conditions

)

SELECT columns

FROM cte\_name

WHERE conditions;

Types of CTEs

1. Non-recursive CTE: A simple CTE used to simplify complex queries.

2. Recursive CTE: A CTE that references itself to generate hierarchical data.

Example of Non-recursive CTE

WITH DepartmentSalaries AS (

SELECT

DepartmentID,

AVG(Salary) AS AvgSalary

FROM

Employees

GROUP BY

DepartmentID

)

SELECT

e.EmployeeID,

e.Salary,

d.AvgSalary

FROM

Employees e

JOIN

DepartmentSalaries d ON e.DepartmentID = d.DepartmentID;

Example of Recursive CTE

WITH RECURSIVE EmployeeHierarchy AS (

SELECT

EmployeeID,

ManagerID,

Name

FROM

Employees

WHERE

ManagerID IS NULL -- Start with top-level managers

UNION ALL

SELECT

e.EmployeeID,

e.ManagerID,

e.Name

FROM

Employees e

INNER JOIN

EmployeeHierarchy eh ON e.ManagerID = eh.EmployeeID

)

SELECT

\*

FROM

EmployeeHierarchy;

Some SQL Queries

-- Create Tables

CREATE TABLE customers (

customer\_id INT PRIMARY KEY AUTO\_INCREMENT,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100) UNIQUE,

registration\_date DATE

);

CREATE TABLE products (

product\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_name VARCHAR(100),

category VARCHAR(50),

price DECIMAL(10, 2),

stock\_quantity INT

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY AUTO\_INCREMENT,

customer\_id INT,

order\_date DATETIME,

total\_amount DECIMAL(10, 2),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

CREATE TABLE order\_items (

order\_item\_id INT PRIMARY KEY AUTO\_INCREMENT,

order\_id INT,

product\_id INT,

quantity INT,

price\_per\_unit DECIMAL(10, 2),

FOREIGN KEY (order\_id) REFERENCES orders(order\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

CREATE TABLE product\_reviews (

review\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_id INT,

customer\_id INT,

rating INT CHECK (rating BETWEEN 1 AND 5),

review\_text TEXT,

review\_date DATE,

FOREIGN KEY (product\_id) REFERENCES products(product\_id),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

Question 1: Find the top 3 customers who have spent the most money, along

with their total spend and the number of orders they've made.

SELECT

c.customer\_id,

CONCAT(c.first\_name, ' ', c.last\_name) AS customer\_name,

SUM(o.total\_amount) AS total\_spent,

COUNT(o.order\_id) AS order\_count

FROM

customers c

JOIN

orders o ON c.customer\_id = o.customer\_id

GROUP BY

c.customer\_id

ORDER BY

total\_spent DESC

LIMIT 3;

Question 2: Calculate the average rating for each product category, but only

include categories with at least 2 reviews.

SELECT

p.category,

AVG(pr.rating) AS average\_rating

FROM

products p

JOIN

product\_reviews pr ON p.product\_id = pr.product\_id

GROUP BY

p.category

HAVING

COUNT(pr.review\_id) >= 2;

Question 3: Find products that have never been ordered.

SELECT

p.product\_id,

p.product\_name

FROM

products p

LEFT JOIN

order\_items oi ON p.product\_id = oi.product\_id

WHERE

oi.order\_item\_id IS NULL;

Question 4: For each customer, find the time difference between their

registration date and their first order date.

SELECT

c.customer\_id,

CONCAT(c.first\_name, ' ', c.last\_name) AS customer\_name,

DATEDIFF(MIN(o.order\_date), c.registration\_date) AS days\_to\_first\_order

FROM

customers c

LEFT JOIN

orders o ON c.customer\_id = o.customer\_id

GROUP BY

c.customer\_id;

Question 5: Create a report showing the total revenue for each month, along

with a running total of revenue throughout the year.

SELECT

DATE\_FORMAT(order\_date, '%Y-%m') AS month,

SUM(total\_amount) AS total\_revenue,

SUM(SUM(total\_amount)) OVER (ORDER BY DATE\_FORMAT(order\_date,

'%Y-%m')) AS running\_total

FROM

orders

GROUP BY

month

ORDER BY

month;

Question 6: Identify customers who have made a purchase but have never left

a product review.

SELECT

c.customer\_id,

CONCAT(c.first\_name, ' ', c.last\_name) AS customer\_name

FROM

customers c

JOIN

orders o ON c.customer\_id = o.customer\_id

LEFT JOIN

product\_reviews pr ON c.customer\_id = pr.customer\_id

WHERE

pr.review\_id IS NULL

GROUP BY

c.customer\_id;

Question 7: Find the product that has been ordered the most times (by

quantity).

SELECT

p.product\_id,

p.product\_name,

SUM(oi.quantity) AS total\_quantity\_ordered

FROM

products p

JOIN

order\_items oi ON p.product\_id = oi.product\_id

GROUP BY

p.product\_id

ORDER BY

total\_quantity\_ordered DESC

LIMIT 1;

Question 8: Calculate the percentage of total revenue that each product

category contributes.

SELECT

p.category,

SUM(oi.price\_per\_unit \* oi.quantity) AS category\_revenue,

(SUM(oi.price\_per\_unit \* oi.quantity) / (SELECT SUM(total\_amount) FROM

orders)) \* 100 AS percentage\_of\_total\_revenue

FROM

products p

JOIN

order\_items oi ON p.product\_id = oi.product\_id

JOIN

orders o ON oi.order\_id = o.order\_id

GROUP BY

p.category;

Question 9: For each customer, find their most frequently purchased product

category.

SELECT

c.customer\_id,

CONCAT(c.first\_name, ' ', c.last\_name) AS customer\_name,

p.category,

COUNT(oi.product\_id) AS purchase\_count

FROM

customers c

JOIN

orders o ON c.customer\_id = o.customer\_id

JOIN

order\_items oi ON o.order\_id = oi.order\_id

JOIN

products p ON oi.product\_id = p.product\_id

GROUP BY

c.customer\_id, p.category

ORDER BY

purchase\_count DESC;

Question 10: Create a query to show the distribution of ratings (count of 1-

star, 2-star, etc.) for each product.

SELECT

p.product\_id,

p.product\_name,

pr.rating,

COUNT(pr.review\_id) AS rating\_count

FROM

products p

LEFT JOIN

product\_reviews pr ON p.product\_id = pr.product\_id

GROUP BY

p.product\_id, pr.rating

ORDER BY

p.product\_id, pr.rating;

Explanation of Output

1. Top 3 Customers: Displays customer IDs, names, total spent, and order

count for the top three customers.

2. Average Rating per Category: Lists product categories with their average

ratings (only categories with at least 2 reviews).

3. Never Ordered Products: Shows products that have not been included in

any order.

4. Time to First Order: Displays the number of days between customer

registration and their first order.

5. Monthly Revenue Report: Provides total revenue per month with a running

total throughout the year.

6. Customers Without Reviews: Identifies customers who made purchases

but have no reviews.

7. Most Ordered Product: Shows the product ordered the most times by

quantity.

8. Revenue by Category: Calculates and displays each product category's

contribution to total revenue as a percentage.

9. Frequent Product Category per Customer: Lists the most frequently

purchased product category for each customer.

10.Rating Distribution: Displays the count of each rating level for each

product.

—---------------------------------------------------------------------------------------------------

Query Implementation

Definition: The process of translating a high-level SQL query into an

execution plan that the database engine can execute.

Steps:

1. Parsing: The SQL query is checked for syntax and semantic

correctness.

2. Optimization: The query optimizer creates a more efficient execution

plan.

3. Execution: The execution plan is executed to retrieve or manipulate

the data.

Query Hinting

Definition: Query hints are directives provided by the user to influence the

optimizer's choices for execution plans.

Purpose: To improve performance by specifying certain behaviors (e.g.,

which indexes to use).

Examples:

o USE INDEX: Forces the use of a specific index.

o FORCE INDEX: Forces the optimizer to use a particular index.

o NO INDEX: Instructs the optimizer not to use specific indexes.

Query Optimizer

Definition: A component of the database engine that determines the most

efficient way to execute a given SQL query.

Functions:

o Cost Estimation: Evaluates the cost of different execution plans

based on factors like I/O, CPU usage, and memory.

o Plan Selection: Chooses the plan with the lowest estimated cost.

o Adaptive Query Optimization: Some systems can adjust execution

plans on-the-fly based on real-time statistics.

Storage Engines

Definition: The underlying technology that handles how data is stored,

retrieved, and managed in a database.

Common Types:

o InnoDB: Supports transactions, foreign keys, and row-level locking.

Default engine in MySQL.

o MyISAM: Does not support transactions but is faster for read-heavy

operations. Uses table-level locking.

o SQLite: Lightweight, serverless, and file-based; good for smaller

applications.

o Columnar Stores: Optimized for read-heavy analytics (e.g., Google

BigQuery, Amazon Redshift).

Query Execution Methods

1. Sequential Scan (Table Scan)

o Definition: The database engine reads every row in a table to find

matching records.

o Use Case: Typically used for small tables or when no suitable indexes

exist.

o Performance: Can be slow for large tables due to the need to read all

rows.

2. Index Scan

o Definition: The engine uses an index to quickly locate the rows that

satisfy the query.

o Types:

Index Range Scan: Retrieves rows within a specified range.

Full Index Scan: Reads all entries in the index.

o Performance: Generally faster than sequential scans, especially for

larger datasets.

3. Index Seek

o Definition: The engine directly accesses rows using an index,

resulting in more efficient retrieval.

o Use Case: Best when queries include specific conditions that match

the index.

o Performance: The fastest method for retrieving data when

appropriate indexes are available.

4. Join Methods

o Nested Loop Join: Iterates through one table for each row in the

other; best for small datasets.

o Merge Join: Sorts both datasets and merges them; efficient for large,

sorted datasets.

o Hash Join: Uses a hash table to find matching rows; effective for

larger, unsorted datasets.

5. Materialized Views

o Definition: A pre-computed result set stored as a table; can be queried

like a regular table.

o Performance: Speeds up complex queries by storing the results of

expensive joins or aggregations.

Query Cache

Definition: A mechanism to store the results of previously executed queries,

allowing the database to return results faster without re-executing the query.

How It Works:

o When a query is executed, the result is stored in memory.

o If the same query is executed again, the database can return the

cached result instead of reprocessing the query.

Benefits:

o Performance Improvement: Reduces response time for frequently

executed queries.

o Reduced Load: Lowers the CPU and I/O usage by avoiding

redundant computations.

Considerations:

o Cache Invalidation: Cached results can become stale if the

underlying data changes. Policies need to be in place to manage this.

o Memory Usage: Query cache can consume significant memory, so

it’s essential to balance caching with available resources.

Show Profiles

Definition: A feature that provides detailed information about the execution

of SQL queries, including resource usage and execution time.

Usage:

o Useful for performance tuning and optimization.

o Allows you to analyze where time is spent during query execution.

Information Provided:

Duration: Time taken for execution.

CPU time: CPU cycles consumed.

I/O time: Time spent on disk operations.

Other resource usage metrics.

Auto Increment

Definition: A property that allows a column (usually a primary key) to

automatically generate a unique sequential number when a new row is

inserted.

Usage:

o Simplifies the process of inserting new records without needing to

manually specify unique values.

o Commonly used in tables where a unique identifier is required.

Behavior:

o The auto-increment value starts at 1 (or a specified start value) and

increments by 1 (or a specified increment value) for each new row.

Parse

Definition: The initial step in processing an SQL query where the query is

analyzed for syntax and semantic correctness.

Process:

o The SQL statement is checked against the database schema to ensure

all referenced objects (tables, columns) exist and are correctly

referenced.

o Any syntax errors or semantic issues are identified before further

processing.

Importance:

o Ensures that only valid queries are executed, helping to prevent

runtime errors.

SSD (Solid State Drive)

Definition: A type of data storage device that uses flash memory to store

data, as opposed to traditional hard drives (HDDs) that use spinning disks.

Benefits:

o Speed: Much faster read/write speeds compared to HDDs, leading to

improved database performance.

o Reliability: Fewer mechanical parts reduce the risk of failure.

o Reduced Latency: Low access times enhance the responsiveness of

database operations.

Usage in Databases:

o SSDs are commonly used for database storage to enhance

performance, particularly for applications requiring high-speed data

access.

—----------------------------------------------------------------------------------------

**Database design in SQL** involves creating a structured framework for storing data efficiently and effectively. A well-designed database ensures data integrity, minimizes redundancy, and provides a clear and intuitive way to access and manipulate data. Here are the key steps and considerations in database design:

**1. Requirements Gathering**

∙ Understand the business requirements and objectives.

∙ Identify the types of data to be stored, the relationships between different data entities, and user requirements.

**2. Conceptual Design**

∙ Create an **Entity-Relationship (ER) Diagram** to visualize entities (tables) and their relationships.

o **Entities**: Represent objects or things (e.g., Customers, Orders). o **Attributes**: Characteristics of entities (e.g., CustomerName, OrderDate).

o **Relationships**: Connections between entities (e.g., a Customer places an Order).

**3. Logical Design**

∙ Convert the ER diagram into a logical structure.

∙ Define tables, primary keys, foreign keys, and constraints:

o **Tables**: Each entity becomes a table.

o **Primary Keys**: Unique identifier for each record (e.g., CustomerID). o **Foreign Keys**: Establish relationships between tables (e.g., Order. CustomerID references Customer.CustomerID).

**4. Normalization**

∙ Normalize the database to reduce redundancy and improve data integrity, typically through 1NF, 2NF, 3NF, and possibly BCNF.

∙ Identify and resolve any anomalies in data relationships.

**5. Physical Design**

∙ Define the actual implementation details:

o **Data Types**: Choose appropriate data types for each column (e.g., INT, VARCHAR, DATE).

o **Indexes**: Create indexes to improve query performance on frequently accessed columns.

o **Storage**: Consider partitioning and storage requirements based on anticipated load.

**6. Schema Definition**

∙ Write SQL statements to create the database schema, including tables, constraints, and relationships.

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Email VARCHAR(100) UNIQUE

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

OrderDate DATE,

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES

Customers(CustomerID)

);

**7. Data Entry and Management**

∙ Define how data will be inserted, updated, and deleted.

∙ Implement stored procedures, triggers, or user-defined functions as needed for business logic.

**8. Testing and Validation**

∙ Test the database design with sample data to ensure it meets requirements. ∙ Validate queries to check performance and accuracy.

**9. Documentation**

∙ Document the database schema, relationships, and any business rules for future reference and maintenance.

**10. Maintenance and Evolution**

∙ Monitor the database for performance issues and adjust as necessary. ∙ Be prepared to update the design to accommodate new business needs or changes in data requirements.

**Example of a Simple Database Design**

**Scenario**: Designing a database for a small online bookstore.

1. **Entities**:

o Books

o Authors

o Customers

o Orders

2. **Relationships**:

o A book can have one or more authors (many-to-many).

o A customer can place one or more orders (one-to-many).

o An order can contain multiple books (many-to-many).

3. **Logical Design**:

CREATE TABLE Authors (

AuthorID INT PRIMARY KEY,

AuthorName VARCHAR(100)

);

CREATE TABLE Books (

BookID INT PRIMARY KEY,

Title VARCHAR(100),

Price DECIMAL(10, 2)

);

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Email VARCHAR(100) UNIQUE

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

OrderDate DATE,

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE BookAuthors (

BookID INT,

AuthorID INT,

PRIMARY KEY (BookID, AuthorID),

FOREIGN KEY (BookID) REFERENCES Books(BookID),

FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID) );

CREATE TABLE OrderDetails (

OrderID INT,

BookID INT,

Quantity INT,

PRIMARY KEY (OrderID, BookID),

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID), FOREIGN KEY (BookID) REFERENCES Books(BookID)

);

**Normalization in SQL** is a process used to organize a database to reduce data redundancy and improve data integrity. It involves structuring a relational database in a way that ensures dependencies are properly enforced through the use of tables and relationships.

**Snow Flake Design**

The Snowflake Schema is a type of database schema that organizes data into a multi-dimensional structure, which is particularly useful for data warehousing and analytical queries. It is an extension of the Star Schema, with additional normalization applied to the dimension tables. This structure minimizes redundancy and improves data integrity.

**Key Characteristics of Snowflake Schema**

1. **Normalized Dimension Tables**: Unlike the Star Schema, where dimension tables are typically denormalized, Snowflake Schema normalizes dimension tables into multiple related tables.

2. **Fact Table**: Central to the schema, it contains quantitative data (facts) for analysis, with foreign keys referencing the related dimension tables. 3. **Hierarchy**: The normalized dimension tables allow for hierarchical relationships, making it easier to manage complex data.

**Structure of a Snowflake Schema**

1. **Fact Table**: Contains metrics or measures, such as sales amounts, quantities sold, etc.

2. **Dimension Tables**: Provide context for the facts. These are further divided into related tables to normalize data.

**Example Scenario**

**Scenario**: A retail store database for analyzing sales data.

**Tables**

1. **Fact Table**: Sales

2. **Dimension Tables**:

o Products

o Categories

o Stores

o Regions

o Customers

**Snowflake Schema Design**

-- Fact Table

CREATE TABLE Sales (

SaleID INT PRIMARY KEY,

SaleDate DATE,

ProductID INT,

StoreID INT,

CustomerID INT,

Quantity INT,

TotalAmount DECIMAL(10, 2),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID), FOREIGN KEY (StoreID) REFERENCES Stores(StoreID), FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) );

-- Dimension Tables

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

CategoryID INT,

FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID) );

CREATE TABLE Categories (

CategoryID INT PRIMARY KEY,

CategoryName VARCHAR(100),

SubcategoryID INT,

FOREIGN KEY (SubcategoryID) REFERENCES

Subcategories(SubcategoryID)

);

CREATE TABLE Subcategories (

SubcategoryID INT PRIMARY KEY,

SubcategoryName VARCHAR(100)

);

CREATE TABLE Stores (

StoreID INT PRIMARY KEY,

StoreName VARCHAR(100),

LocationID INT,

FOREIGN KEY (LocationID) REFERENCES Locations(LocationID) );

CREATE TABLE Locations (

LocationID INT PRIMARY KEY,

City VARCHAR(100),

RegionID INT,

FOREIGN KEY (RegionID) REFERENCES Regions(RegionID) );

CREATE TABLE Regions (

RegionID INT PRIMARY KEY,

RegionName VARCHAR(100)

);

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Email VARCHAR(100),

Phone VARCHAR(15)

);

**Benefits of Snowflake Schema**

∙ **Reduced Data Redundancy**: Normalization reduces duplicate data, improving storage efficiency.

∙ **Improved Data Integrity**: Changes to data need to be made in only one place, reducing the risk of anomalies.

∙ **Enhanced Query Performance for Certain Queries**: While querying can be more complex due to joins, certain types of analytical queries benefit from the structure.

**Drawbacks of Snowflake Schema**

∙ **Complex Queries**: More joins can lead to complicated SQL queries, which may affect performance.

∙ **Potential Performance Issues**: Depending on the database and size of data, the need for multiple joins can slow down query performance compared to a denormalized structure.

**Use Cases**

Snowflake schemas are commonly used in data warehouses and business intelligence applications where data is analyzed for trends, forecasts, and business metrics.

**Fact Constellation** refers to a data model used in data warehousing. It is a type of schema that combines multiple star schemas, allowing for complex queries and analysis. Here’s a brief overview:

**Components of a Fact Constellation**

1. **Facts**: These are quantitative data points (measurable metrics) stored in fact tables. For example, sales revenue, quantities sold, etc.

2. **Dimensions**: These provide context to the facts, stored in dimension tables. Common dimensions include time, geography, product, and customer. 3. **Fact Tables**: Central tables that contain foreign keys to the dimension tables along with the measurable facts.

4. **Dimension Tables**: Tables that contain attributes related to the dimensions of the facts. Each dimension table is usually denormalized.

**Example**

Consider a retail data warehouse:

∙ **Fact Table**: Sales

o Columns: SalesID, DateKey, ProductKey, StoreKey, SalesAmount, QuantitySold

∙ **Dimension Tables**:

o Date: Contains details about dates.

o Product: Contains product details.

o Store: Contains store location information.

**Benefits**

∙ **Complex Analysis**: Allows for querying across multiple fact tables. ∙ **Flexibility**: Supports different analytical needs by combining various fact tables.

**SQL Implementation**

CREATE TABLE DimDate (

DateKey INT PRIMARY KEY,

Date DATE,

Month INT,

Year INT

);

CREATE TABLE DimProduct ( ProductKey INT PRIMARY KEY, ProductName VARCHAR(100), Category VARCHAR(100) );

CREATE TABLE DimStore ( StoreKey INT PRIMARY KEY, StoreName VARCHAR(100), Location VARCHAR(100) );

CREATE TABLE FactSales ( SalesID INT PRIMARY KEY, DateKey INT,

ProductKey INT,

StoreKey INT,

SalesAmount DECIMAL(10, 2),

QuantitySold INT,

FOREIGN KEY (DateKey) REFERENCES DimDate(DateKey), FOREIGN KEY (ProductKey) REFERENCES DimProduct(ProductKey), FOREIGN KEY (StoreKey) REFERENCES DimStore(StoreKey) );

**Slowly Changing Dimensions (SCD**) are a critical concept in data warehousing, particularly for managing and tracking changes in dimension data over time. There are several types of SCDs, with the most common being Type 1, Type 2, and Type 3. Here's a breakdown of each type along with SQL examples.

**Types of Slowly Changing Dimensions**

**Type 1: Overwrite**

This type simply overwrites the existing data. No historical data is kept. **Example SQL:**

UPDATE DimCustomer

SET CustomerName = 'New Name'

WHERE CustomerID = 1;

**Type 2: Add New Row**

This type creates a new record whenever a change occurs, preserving historical data. You typically include effective date columns to indicate the time period the record was valid.

**Example SQL:**

-- Assuming you have effective date columns

INSERT INTO DimCustomer (CustomerID, CustomerName, StartDate, EndDate, IsActive)

VALUES (1, 'New Name', CURRENT\_DATE, NULL, 1);

-- Mark the old record as inactive

UPDATE DimCustomer

SET EndDate = CURRENT\_DATE, IsActive = 0

WHERE CustomerID = 1 AND IsActive = 1;

**Type 3: Add New Attribute**

This type adds new columns to the existing table to store previous values, allowing for limited historical tracking (e.g., current and previous values).

**Example SQL:**

UPDATE DimCustomer

SET PreviousName = CustomerName, CustomerName = 'New Name' WHERE CustomerID = 1;

**Fact Table**

A **fact table** is the central table in a star schema of a data warehouse. It contains measurable, quantitative data (facts) and foreign keys to dimension tables.

***Characteristics:***

∙ **Contains Facts**: Stores numeric values that can be aggregated (e.g., sales amount, quantity sold).

∙ **Foreign Keys**: Includes foreign keys that link to dimension tables, providing context for the facts.

∙ **Granularity**: Defines the level of detail. For example, a sales fact table might have a record for each transaction.

**Example:**

Let's create a simple sales fact table:

CREATE TABLE FactSales (

SalesID INT PRIMARY KEY,

DateKey INT,

ProductKey INT,

StoreKey INT,

SalesAmount DECIMAL(10, 2),

QuantitySold INT,

FOREIGN KEY (DateKey) REFERENCES DimDate(DateKey), FOREIGN KEY (ProductKey) REFERENCES DimProduct(ProductKey), FOREIGN KEY (StoreKey) REFERENCES DimStore(StoreKey) );

**Factless Table**

A **factless table** is a fact table that does not contain any measurable facts or numeric data. Instead, it records the relationships between dimensions. It is often used to track events or coverage.

***Characteristics:***

∙ **No Numeric Facts**: Contains only foreign keys to dimension tables.

∙ **Event Tracking**: Useful for tracking events, such as attendance, inventory transactions, or many-to-many relationships.

***Example:***

An example of a factless table could be an attendance tracking table: CREATE TABLE FactAttendance (

StudentID INT,

CourseID INT,

AttendanceDate DATE,

FOREIGN KEY (StudentID) REFERENCES DimStudent(StudentID), FOREIGN KEY (CourseID) REFERENCES DimCourse(CourseID), PRIMARY KEY (StudentID, CourseID, AttendanceDate)

);

A **junk dimension** is a type of dimension in data warehousing that combines several low-cardinality attributes into a single dimension table. This is often done to simplify the schema and improve query performance. Junk dimensions typically consist of flags or miscellaneous attributes that don’t warrant their own dimension tables.

**Characteristics of Junk Dimensions**

∙ **Low Cardinality**: Attributes in a junk dimension have a limited number of distinct values (e.g., Yes/No flags, status indicators).

∙ **Simplification**: Helps avoid the proliferation of multiple small dimension tables.

∙ **Performance Improvement**: Reduces the complexity of joins in queries by consolidating related attributes.

**Example Scenario**

Imagine a scenario where you have several attributes related to customer orders, such as:

∙ Gift wrap (Yes/No)

∙ Discount applied (Yes/No)

∙ Order status (Pending, Completed, Cancelled)

**Creating a Junk Dimension**

You can create a junk dimension that consolidates these attributes into one table. Here’s how you might implement it in SQL:

**Step 1: Create the Junk Dimension Table**

CREATE TABLE DimJunk (

JunkID INT PRIMARY KEY,

GiftWrap BIT,

DiscountApplied BIT,

OrderStatus VARCHAR(20)

);

**Step 2: Populate the Junk Dimension Table**

INSERT INTO DimJunk (JunkID, GiftWrap, DiscountApplied, OrderStatus) VALUES

(1, 1, 0, 'Pending'),

(2, 0, 1, 'Completed'),

(3, 1, 1, 'Cancelled');

**Step 3: Fact Table Referencing the Junk Dimension**

You can then reference this junk dimension from your fact table. For example, an order fact table could look like this:

CREATE TABLE FactOrder (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

JunkID INT,

TotalAmount DECIMAL(10, 2),

FOREIGN KEY (JunkID) REFERENCES DimJunk(JunkID) );

**Benefits of Using Junk Dimensions**

1. **Reduced Complexity**: Minimizes the number of tables in the schema. 2. **Improved Query Performance**: Simplifies joins, which can enhance query performance.

3. **Better Organization**: Groups related attributes logically, making the schema easier to understand.

A **degenerate dimension** is a type of dimension in a data warehouse that does not have its own dimension table. Instead, it exists as an attribute within a fact table. This typically occurs when the dimension's values are unique and directly associated with a single transaction, such as order numbers, invoice numbers, or other identifiers.

**Characteristics of Degenerate Dimensions**

∙ **No Separate Dimension Table**: Degenerate dimensions are stored directly in the fact table instead of being linked to a separate dimension table. ∙ **Unique Values**: They usually represent attributes that have unique values and are not suitable for aggregation or analysis like traditional dimensions. ∙ **Simplicity**: Helps simplify the schema by reducing the number of dimension tables.

**Example Scenario**

Consider a retail sales environment where each sale is associated with a unique invoice number. The invoice number would serve as a degenerate dimension.

**Creating a Degenerate Dimension**

**Fact Table Example**

Defining a fact table that includes a degenerate dimension:

CREATE TABLE FactSales (

SaleID INT PRIMARY KEY,

InvoiceNumber VARCHAR(20), -- Degenerate dimension

DateKey INT,

ProductKey INT,

StoreKey INT,

SalesAmount DECIMAL(10, 2),

QuantitySold INT,

FOREIGN KEY (DateKey) REFERENCES DimDate(DateKey), FOREIGN KEY (ProductKey) REFERENCES DimProduct(ProductKey), FOREIGN KEY (StoreKey) REFERENCES DimStore(StoreKey) );

**Usage**

When we query the sales data, we can include the InvoiceNumber directly from the FactSales table without needing to join to a separate dimension table.

**Example Query**

SELECT

InvoiceNumber,

SalesAmount,

QuantitySold

FROM

FactSales

WHERE

DateKey = '20240101';

**Dimension tables** are essential components of a data warehouse, providing context to the facts stored in fact tables. They contain descriptive attributes (or fields) that help categorize, filter, and understand the measures in the fact tables.

**Characteristics of Dimension Tables**

∙ **Descriptive Attributes**: Dimension tables store attributes that provide context to the data, such as names, dates, categories, and locations. ∙ **Denormalization**: Often denormalized for easier querying, which means that data may be duplicated across rows to improve read performance. ∙ **Primary Keys**: Each dimension table has a primary key that uniquely identifies each record. This key is referenced by fact tables as a foreign key.

**Example Dimension Tables**

Here’s a breakdown of some common dimension tables in a retail data warehouse: **1. Date Dimension**

CREATE TABLE DimDate (

DateKey INT PRIMARY KEY, -- Usually in YYYYMMDD format

Date DATE,

Day INT,

Month INT,

Year INT,

Quarter INT,

DayOfWeek VARCHAR(10) );

**2.Product Dimension**

CREATE TABLE DimProduct ( ProductKey INT PRIMARY KEY, ProductName VARCHAR(100), Category VARCHAR(100), Brand VARCHAR(100),

Price DECIMAL(10, 2)

);

**3.Customer Dimension**

CREATE TABLE DimCustomer ( CustomerKey INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50), Email VARCHAR(100),

Phone VARCHAR(15),

Address VARCHAR(255),

City VARCHAR(100),

State VARCHAR(50),

ZipCode VARCHAR(10)

);

**4. Store Dimension**

CREATE TABLE DimStore ( StoreKey INT PRIMARY KEY, StoreName VARCHAR(100), Location VARCHAR(100), ManagerName VARCHAR(100),

OpeningDate DATE

);

**Connecting Dimension Tables to Fact Tables**

Dimension tables are linked to fact tables through foreign keys. For example, a fact sales table might look like this:

CREATE TABLE FactSales (

SalesID INT PRIMARY KEY,

DateKey INT,

ProductKey INT,

CustomerKey INT,

StoreKey INT,

SalesAmount DECIMAL(10, 2),

QuantitySold INT,

FOREIGN KEY (DateKey) REFERENCES DimDate(DateKey), FOREIGN KEY (ProductKey) REFERENCES DimProduct(ProductKey), FOREIGN KEY (CustomerKey) REFERENCES DimCustomer(CustomerKey), FOREIGN KEY (StoreKey) REFERENCES DimStore(StoreKey) );

**Benefits of Dimension Tables**

1. **Contextual Information**: Provide important context for interpreting the data in fact tables.

2. **Facilitating Analysis**: Enable easier filtering, grouping, and reporting in queries.

3. **Improved Performance**: Denormalization helps improve read performance, especially in analytical queries.

**Triggers** are special types of stored procedures that automatically execute (or "fire") in response to specific events on a table or view. They can help enforce business rules, maintain audit trails, and synchronize tables, among other uses.

**Types of Triggers**

1. **BEFORE Trigger**: Executes before the triggering action (INSERT, UPDATE, DELETE).

2. **AFTER Trigger**: Executes after the triggering action.

3. **INSTEAD OF Trigger**: Replaces the action (typically used for views).

**Creating Triggers**

Here are examples of how to create different types of triggers in SQL. **Example 1: BEFORE Trigger**

This trigger checks for a specific condition before inserting a record into a tabl CREATE TRIGGER trg\_before\_insert

BEFORE INSERT ON Employees

FOR EACH ROW

BEGIN

IF NEW.salary < 0 THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Salary cannot be negative';

END IF;

END;

**Example 2: AFTER Trigger**

This trigger logs changes to a separate audit table after an update occurs. CREATE TRIGGER trg\_after\_update

AFTER UPDATE ON Products

FOR EACH ROW

BEGIN

INSERT INTO ProductAudit (ProductID, OldPrice, NewPrice, ChangeDate) VALUES (OLD.ProductID, OLD.Price, NEW.Price, NOW()); END;

**Example 3: INSTEAD OF Trigger**

This trigger allows you to perform a specific action when trying to insert a record into a view.

CREATE TRIGGER trg\_instead\_of\_insert

INSTEAD OF INSERT ON ProductView

FOR EACH ROW

BEGIN

INSERT INTO Products (Name, Price) VALUES (NEW.Name, NEW.Price); -- Additional logic if needed

END;

**Sharding** is a database architecture pattern that involves splitting a large database into smaller, more manageable pieces, called **shards**. Each shard is a separate database that holds a portion of the overall data. This approach can help improve performance, scalability, and manageability for large datasets and high-traffic applications.

**Sharding Strategies**

1. **Range-based Sharding**: Data is divided based on ranges of the sharding key. For example, user IDs from 1 to 1000 might go to one shard, while 1001 to 2000 go to another.

2. **Hash-based Sharding**: A hash function is applied to the sharding key, and data is distributed based on the hash values. This method helps achieve a more uniform distribution of data.

3. **Directory-based Sharding**: A lookup table (directory) keeps track of where each piece of data resides. This method allows for more flexibility in shard allocation.

A **materialized queue** in SQL is a concept that combines aspects of both materialized views and traditional queue mechanisms. It is used to efficiently store, manage, and retrieve messages or data items in a way that allows for high performance and easy querying, often in systems that require handling large volumes of data.

**Implementing a Materialized Queue**

**Example Schema**

Here’s a simple example of how you might create a materialized queue in SQL: CREATE TABLE MaterializedQueue (

MessageID INT PRIMARY KEY AUTO\_INCREMENT,

MessageContent TEXT,

Status VARCHAR(20) DEFAULT 'pending', -- Status can be 'pending', 'processed', etc.

CreatedAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, ProcessedAt TIMESTAMP NULL

);

**Basic Operations**

1. **Enqueue (Insert a Message)**: To add a message to the queue:

INSERT INTO MaterializedQueue (MessageContent) VALUES ('Your message here');

2. **Dequeue (Process a Message)**: To retrieve and mark a message as processed:

UPDATE MaterializedQueue

SET Status = 'processed', ProcessedAt = CURRENT\_TIMESTAMP WHERE MessageID = (

SELECT MessageID FROM MaterializedQueue

WHERE Status = 'pending'

ORDER BY CreatedAt ASC

LIMIT 1

);

3. **Querying Messages**: To fetch pending messages:

SELECT \* FROM MaterializedQueue WHERE Status = 'pending'; 4. **Monitoring the Queue**: To see the count of messages in different statuses:

SELECT Status, COUNT(\*) AS Count

FROM MaterializedQueue

GROUP BY Status;

—--------------------------------------------------------------------------------------------------

WEEK-5

Dependency Injection (DI)

Dependency Injection is a design pattern used to implement IoC, allowing the

creation of dependent objects outside of a class and providing those objects to a

class in different ways. It promotes loose coupling and enhances testability.

Types of Dependency Injection

1. Constructor Injection:

Dependencies are provided through a class constructor.

Ensures that the required dependencies are available when the object is

created.

Example:

@Component

public class MyService {

private final MyRepository myRepository;

@Autowired

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

}

Setter Injection:

Dependencies are provided through setter methods after the object is

constructed.

Allows for optional dependencies and can change the dependency after

object creation.

Example:

@Component

public class MyService {

private MyRepository myRepository;

@Autowired

public void setMyRepository(MyRepository myRepository) {

this.myRepository = myRepository;

}

}

Field Injection:

Dependencies are injected directly into the fields of a class.

It's less preferred due to difficulties in testing and maintaining immutability.

Example:

@Component

public class MyService {

@Autowired

private MyRepository myRepository;

}

Autowiring

Autowiring allows Spring to automatically resolve and inject dependencies

based on type or by matching bean names.

Types of autowiring:

o By Type (@Autowired): Injects the bean that matches the type.

o By Name: Matches the bean name.

o Constructor: Uses constructor injection.

o Required: Can be set to true or false to make a dependency optional.

Inversion of Control (IoC)

Inversion of Control is a broader principle where the control of object creation and

management is transferred from the object itself to a container or framework.

Dependency Injection is a specific implementation of IoC.

Spring IoC Container Types

1. Bean Factory:

o The simplest container providing basic support for DI.

o Lazy initialization of beans, meaning beans are created only when

requested.

o Suitable for lightweight applications.

2. Application Context:

o A more advanced container with additional features like event

propagation, internationalization, and application-layer support.

o Eager initialization, meaning beans are created at startup unless

specified otherwise.

Bean Definition and Configuration

Beans in Spring are defined in configuration classes (using @Configuration)

or XML files.

You can define scopes, lifecycle methods, and custom bean initialization and

destruction methods.

—--------------------------------------------------------------------------------------------------

@RestController

It is a specialized version of the @Controller annotation in Spring, combining it

with the @ResponseBody annotation.

@GetMapping

Definition:

o It is a composed annotation that acts as a shortcut for

@RequestMapping(method = RequestMethod.GET).

BASIC PROGRAMS USING SPRING

Hello World Program

Main Application Class: HelloworldApplication.java

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })

public class HelloWorldApplication {

public static void main(String[] args) {

SpringApplication.run(HelloWorldApplication.class, args);

System.out.println("Hello world from main");

// SpringApplication app = new SpringApplication(HelloWorldApp.class);

// System.out.println("Hello World from main class");

// app.setAdditionalProfiles(args.length > 0 ? args[0] : "default");

// app.run(args);

}

}

Create a Controller: new Java class HelloController in the same package

package com.example.demo;

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

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

@RestController

public class HelloWorldController {

@GetMapping("/api/hello")

public String hello() {

return "Hello, World!";

}

}

Whether Controller Program

Main Application Class: WeatherAppApplication.java

package com.example.weatherapp;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class WeatherAppApplication {

public static void main(String[] args) {

SpringApplication.run(WeatherAppApplication.class, args);

}

}

Creating the Weather Controller: new class called WeatherController in the same

package (com.example.weatherapp).

package com.example.weatherapp;

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

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

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

@RestController

public class WeatherController {

@GetMapping("/weather/{location}")

public String getWeather(@PathVariable String location) {

// Simulating weather data for simplicity

return "The weather in " + location + " is sunny with a temperature of 25°C.";

}

}

Whether Service Program

Main Application Class: WeatherServiceApplication.java

package com.example.weather;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class WeatherServiceApplication {

public static void main(String[] args) {

SpringApplication.run(WeatherServiceApplication.class, args);

}

}

Creating a Weather Service Class: Create a new class called WeatherService in the

same package.

package com.example.weather;

import org.springframework.stereotype.Service;

@Service

public class WeatherService {

public String getWeather(String location) {

// Simulating weather data for simplicity

return "The weather in " + location + " is sunny with a temperature of 25°C.";

}

}

Creating the Weather Controller: new class called WeatherController in the

same package.

package com.example.weather;

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

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

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

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

@RestController

public class WeatherController {

private final WeatherService weatherService;

@Autowired

public WeatherController(WeatherService weatherService) {

this.weatherService = weatherService;

}

@GetMapping("/weather/{location}")

public String getWeather(@PathVariable String location) {

return weatherService.getWeather(location);

}

}

—----------------------------------------------------------------------------------------------------

Views are virtual tables that represent the result of a SQL query. They don't store

the data themselves but rather show data from one or more underlying tables.

Views provide an abstracted way to present the data, often simplifying complex

queries or restricting access to certain data for security purposes.

Example query based on the concepts “views”

1. Creating a Database and Table

CREATE DATABASE view\_test; creates a new database called view\_test.

USE view\_test; selects the database for current session use.

CREATE TABLE mv\_it\_employees (...); defines a table with columns like

id, first\_name, last\_name, department, and salary. Each employee has a

unique id, and salary is stored as a DECIMAL type.

Note: You must use employees as the base table in later parts of your code for

creating views and inserting data.

2. Inserting Data

The INSERT INTO mv\_it\_employees VALUES (...); statement inserts rows

into the mv\_it\_employees table. Each row represents an employee with

specific details, such as their name, department, and salary.

3. Creating a View

Views provide a virtual table based on a SELECT statement. This view only

shows employees from the IT department.

Example:

CREATE VIEW it\_emp AS

SELECT id, first\_name, last\_name, department, salary

FROM mv\_it\_employees

WHERE department = 'IT';

4. Querying a View

SELECT \* FROM it\_emp; retrieves all rows from the it\_emp view.

SHOW CREATE VIEW it\_emp; displays the SQL definition used to create

the view.

EXPLAIN SELECT \* FROM it\_emp; provides the execution plan, showing

how MySQL will retrieve data when querying the view, which helps

optimize performance.

5. Viewing the View Definition

The query on information\_schema.views retrieves the definition of the view

it\_emp from the view\_test database.

Example:

SELECT view\_definition

FROM information\_schema.views

WHERE table\_schema = 'view\_test'

AND table\_name = 'it\_emp';

6. Stored Procedures

CREATE PROCEDURE refresh\_mv\_it\_employees() defines a stored

procedure that updates the mv\_it\_employees table. This procedure inserts all

employees from the employees table where the department is IT into the

mv\_it\_employees table.

Example:

DELIMITER //

CREATE PROCEDURE refresh\_mv\_it\_employees()

BEGIN

INSERT INTO mv\_it\_employees

SELECT id, first\_name, last\_name, department, salary

FROM employees

WHERE department = 'IT';

END //

DELIMITER ;

CALL refresh\_mv\_it\_employees(); is used to execute the stored procedure.

7. Events

CREATE EVENT refresh\_mv\_it\_employees\_daily creates a scheduled

event that runs the refresh\_mv\_it\_employees stored procedure once every

day, starting from a specified time.

Example:

CREATE EVENT refresh\_mv\_it\_employees\_daily

ON SCHEDULE EVERY 1 DAY

STARTS CURRENT\_DATE + INTERVAL 1 DAY

DO CALL refresh\_mv\_it\_employees();

This ensures the materialized view (mv\_it\_employees) is updated daily.

8. Viewing Processes

SHOW PROCESSLIST; shows all currently running MySQL processes,

which is helpful for monitoring query performance and identifying issues

like long-running queries.

SELECT \* FROM information\_schema.processlist; queries the processlist

for more detailed information about the active sessions.

SELECT \* FROM performance\_schema.threads; retrieves thread-level

information, which can help diagnose database performance bottlenecks.

Program based on spring

// File: WeatherConfig.java

package com.example.weather;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.Configuration;

import java.util.Map;

import java.util.HashMap;

@Configuration

@ConfigurationProperties(prefix = "weather")

public class WeatherConfig {

private String defaultCondition;

private Map<String, TemperatureRange> cityTemperatures = new

HashMap<>();

// Getters and setters

public String getDefaultCondition() { return defaultCondition; }

public void setDefaultCondition(String defaultCondition) {

this.defaultCondition = defaultCondition; }

public Map<String, TemperatureRange> getCityTemperatures() { return

cityTemperatures; }

public void setCityTemperatures(Map<String, TemperatureRange>

cityTemperatures) { this.cityTemperatures = cityTemperatures; }

public static class TemperatureRange {

private int min;

private int max;

// Getters and setters

public int getMin() { return min; }

public void setMin(int min) { this.min = min; }

public int getMax() { return max; }

public void setMax(int max) { this.max = max; }

}

}

// File: WeatherRecord.java

package com.example.weather;

import org.springframework.stereotype.Component;

@Component

public class WeatherRecord {

private String city;

private int temperature;

private String condition;

public void initialize(String city, int temperature) {

this.city = city;

this.temperature = temperature;

}

// Getters and setters

public String getCity() { return city; }

public void setCity(String city) { this.city = city; }

public int getTemperature() { return temperature; }

public void setTemperature(int temperature) { this.temperature = temperature; }

public String getCondition() { return condition; }

public void setCondition(String condition) { this.condition = condition; }

}

// File: WeatherService.java

package com.example.weather;

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

import org.springframework.context.ApplicationContext;

import org.springframework.stereotype.Service;

import java.util.HashMap;

import java.util.Map;

@Service

public class WeatherService {

private final WeatherConfig weatherConfig;

private final ApplicationContext context;

private final Map<String, WeatherRecord> weatherData = new HashMap<>();

@Autowired

public WeatherService(WeatherConfig weatherConfig, ApplicationContext

context) {

this.weatherConfig = weatherConfig;

this.context = context;

}

public String getWeather(String city) {

WeatherRecord record = weatherData.get(city);

if (record == null) {

return weatherConfig.getDefaultCondition();

}

return String.format("The temperature in %s is %d°C. Condition: %s",

city, record.getTemperature(), record.getCondition());

}

public void updateWeather(String city, int temperature) {

WeatherRecord record = weatherData.computeIfAbsent(city, k -> {

WeatherRecord newRecord = context.getBean(WeatherRecord.class);

newRecord.initialize(city, temperature);

return newRecord;

});

record.setTemperature(temperature);

record.setCondition(determineCondition(city, temperature));

}

private String determineCondition(String city, int temperature) {

WeatherConfig.TemperatureRange range =

weatherConfig.getCityTemperatures().get(city);

if (range == null) {

return weatherConfig.getDefaultCondition();

}

if (temperature < range.getMin()) {

return "Cold";

} else if (temperature > range.getMax()) {

return "Hot";

} else {

return "Pleasant";

}

}

public Map<String, String> getAllWeatherData() {

Map<String, String> result = new HashMap<>();

weatherData.forEach((city, record) ->

result.put(city, String.format("%d°C, %s", record.getTemperature(),

record.getCondition()))

);

return result;

}

}

// File: WeatherController.java

package com.example.weather;

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

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

import java.util.Map;

@RestController

@RequestMapping("/api/weather")

public class WeatherController {

private final WeatherService weatherService;

@Autowired

public WeatherController(WeatherService weatherService) {

this.weatherService = weatherService;

}

@GetMapping("/{city}")

public String getWeather(@PathVariable String city) {

return weatherService.getWeather(city);

}

@PostMapping("/{city}")

public String updateWeather(@PathVariable String city, @RequestParam int

temperature) {

weatherService.updateWeather(city, temperature);

return "Weather updated for " + city;

}

@GetMapping

public Map<String, String> getAllWeather() {

return weatherService.getAllWeatherData();

}

}

// File: application.properties

weather.default-condition=Moderate

weather.city-temperatures.new-york.min=10

weather.city-temperatures.new-york.max=25

weather.city-temperatures.london.min=5

weather.city-temperatures.london.max=20

weather.city-temperatures.tokyo.min=15

weather.city-temperatures.tokyo.max=30

//File: xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.3.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.example</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>21</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

<version>2.0.0.Final</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.33</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<mainClass>com.example.HelloWorldApplication</mainClass>

</configuration>

</plugin>

</plugins>

</build>

</project>

—------------------------------------------------------------------------------------------------

Spring Boot Data JPA is a powerful and easy-to-use framework that simplifies the

development of database-driven applications in Java. It builds on the JPA (Java

Persistence API) specification and provides a set of features to interact with

relational databases using spring's programming model.

1. Dependencies

To use Spring Data JPA, you need to include the necessary dependencies in your

pom.xml (for Maven) or build.gradle (for Gradle):

//File: XML

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId> <!-- Or any other database -->

<artifactId>h2</artifactId> <!-- Embedded database for testing -->

</dependency>

2. Entity Class

An entity represents a table in the database. You annotate your Java class with

@Entity, and its fields represent the columns.

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class Employee {

@Id

private Long id;

private String name;

private String role;

// Getters and Setters

}

3. Repository Interface

You create a repository interface by extending one of Spring Data JPA’s interfaces,

typically JpaRepository, which provides CRUD operations for your entity.

import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Custom query methods (optional)

List<Employee> findByRole(String role);

}

4. Service Layer

The service layer contains business logic and interacts with the repository.

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

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> getEmployeesByRole(String role) {

return employeeRepository.findByRole(role);

}

public Employee saveEmployee(Employee employee) {

return employeeRepository.save(employee);

}

}

5. Controller

The controller handles HTTP requests and returns responses. You can use

@RestController to build a REST API.

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

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

import java.util.List;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping("/role/{role}")

public List<Employee> getEmployeesByRole(@PathVariable String role) {

return employeeService.getEmployeesByRole(role);

}

@PostMapping("/save")

public Employee saveEmployee(@RequestBody Employee employee) {

return employeeService.saveEmployee(employee);

}

}

6. Application Properties

You configure your database connection in application.properties or

application.yml. For example, using H2 (an in-memory database):

application.properties:

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.hibernate.ddl-auto=update

7. Running the Application

You can run your Spring Boot application by using:

Maven: mvn spring-boot:run

Gradle: ./gradlew bootRun

Spring Boot will automatically configure everything and connect to the database

—---------------------------------------------------------------------------------------------------

QUERY-1:

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(50)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(50),

DepartmentID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

CREATE TABLE Projects (

ProjectID INT PRIMARY KEY,

ProjectName VARCHAR(100),

DepartmentID INT,

StartDate DATE,

EndDate DATE,

EmployeeID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID),

FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)

);

Problem Statement: You're working with a company that manages projects across

different departments. They want to analyze project performance, employee

contributions, and department efficiency. Your task is to write a SQL query that:

1. Rank Employees Within Each Department Based on Number of Projects

Completed

WITH ProjectCounts AS (

SELECT

EmployeeID,

DepartmentID,

COUNT(ProjectID) AS ProjectCount

FROM Projects

GROUP BY EmployeeID, DepartmentID

)

SELECT

e.EmployeeID,

e.EmployeeName,

d.DepartmentName,

pc.ProjectCount,

ROW\_NUMBER() OVER (PARTITION BY e.DepartmentID ORDER BY

pc.ProjectCount DESC) AS RankInDepartment

FROM Employees e

JOIN ProjectCounts pc ON e.EmployeeID = pc.EmployeeID

JOIN Departments d ON e.DepartmentID = d.DepartmentID

ORDER BY d.DepartmentName, RankInDepartment;

2. Calculate the Average Project Duration for Each Department

WITH ProjectDurations AS (

SELECT

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration

FROM Projects

)

SELECT

d.DepartmentID,

d.DepartmentName,

AVG(pd.Duration) AS AvgProjectDuration

FROM ProjectDurations pd

JOIN Departments d ON pd.DepartmentID = d.DepartmentID

GROUP BY d.DepartmentID, d.DepartmentName;

3. Identify Projects That Took Longer Than the Department's Average

WITH ProjectDurations AS (

SELECT

ProjectID,

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration

FROM Projects

),

DepartmentAverages AS (

SELECT

DepartmentID,

AVG(Duration) AS AvgDuration

FROM ProjectDurations

GROUP BY DepartmentID

)

SELECT

p.ProjectID,

p.ProjectName,

d.DepartmentName,

pd.Duration,

da.AvgDuration

FROM Projects p

JOIN ProjectDurations pd ON p.ProjectID = pd.ProjectID

JOIN DepartmentAverages da ON pd.DepartmentID = da.DepartmentID

JOIN Departments d ON p.DepartmentID = d.DepartmentID

WHERE pd.Duration > da.AvgDuration

ORDER BY pd.Duration DESC;

4. Find the Top 3 Most Efficient Employees in Each Department (Based on

Average Project Duration)

WITH ProjectDurations AS (

SELECT

EmployeeID,

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration

FROM Projects

),

EmployeeAvgDurations AS (

SELECT

EmployeeID,

DepartmentID,

AVG(Duration) AS AvgDuration

FROM ProjectDurations

GROUP BY EmployeeID, DepartmentID

)

SELECT

e.EmployeeID,

e.EmployeeName,

d.DepartmentName,

ed.AvgDuration

FROM EmployeeAvgDurations ed

JOIN Employees e ON ed.EmployeeID = e.EmployeeID

JOIN Departments d ON ed.DepartmentID = d.DepartmentID

WHERE (

SELECT COUNT(\*)

FROM EmployeeAvgDurations ed2

WHERE ed2.DepartmentID = ed.DepartmentID

AND ed2.AvgDuration < ed.AvgDuration

) < 3

ORDER BY d.DepartmentName, ed.AvgDuration ASC;

5. Compare Each Project's Duration to the Previous Project in the Same

Department

WITH ProjectDurations AS (

SELECT

ProjectID,

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration,

ROW\_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY

StartDate) AS RowNum

FROM Projects

)

SELECT

p.ProjectID,

p.ProjectName,

d.DepartmentName,

pd.Duration AS CurrentDuration,

LAG(pd.Duration) OVER (PARTITION BY pd.DepartmentID ORDER BY

pd.RowNum) AS PreviousDuration

FROM Projects p

JOIN ProjectDurations pd ON p.ProjectID = pd.ProjectID

JOIN Departments d ON p.DepartmentID = d.DepartmentID

ORDER BY d.DepartmentName, pd.RowNum;

QUERY- 2:

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100),

RegistrationDate DATE,

Segment VARCHAR(20)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

Name VARCHAR(100),

Category VARCHAR(50),

Price DECIMAL(10, 2)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

TotalAmount DECIMAL(10, 2),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE OrderDetails (

OrderID INT,

ProductID INT,

Quantity INT,

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

Insert sample data into Customers

INSERT INTO Customers (CustomerID, Name, Email, RegistrationDate,

Segment)

VALUES

(1, 'John Doe', 'john@example.com', '2022-01-15', 'Regular'),

(2, 'Jane Smith', 'jane@example.com', '2022-02-20', 'Premium'),

(3, 'Bob Johnson', 'bob@example.com', '2022-03-10', 'Regular'),

(4, 'Alice Brown', 'alice@example.com', '2022-04-05', 'Premium'),

(5, 'Charlie Davis', 'charlie@example.com', '2022-05-01', 'Regular');

-- Insert sample data into Products

INSERT INTO Products (ProductID, Name, Category, Price)

VALUES

(1, 'Laptop', 'Electronics', 999.99),

(2, 'Smartphone', 'Electronics', 599.99),

(3, 'T-shirt', 'Clothing', 19.99),

(4, 'Jeans', 'Clothing', 49.99),

(5, 'Book', 'Books', 14.99);

-- Insert sample data into Orders

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount)

VALUES

(1, 1, '2023-01-10', 1019.98),

(2, 2, '2023-02-15', 649.98),

(3, 3, '2023-03-20', 34.98),

(4, 4, '2023-04-25', 1049.98),

(5, 5, '2023-05-30', 64.98),

(6, 1, '2023-06-05', 614.98),

(7, 2, '2023-07-10', 1014.98),

(8, 3, '2023-08-15', 599.99),

(9, 4, '2023-09-20', 69.98),

(10, 5, '2023-10-25', 999.99);

-- Insert sample data into OrderDetails

INSERT INTO OrderDetails (OrderID, ProductID, Quantity)

VALUES

(1, 1, 1), (1, 3, 1),

(2, 2, 1), (2, 4, 1),

(3, 3, 1), (3, 5, 1),

(4, 1, 1), (4, 4, 1),

(5, 3, 1), (5, 5, 1),

(6, 2, 1), (6, 3, 1),

(7, 1, 1), (7, 5, 1),

(8, 2, 1),

(9, 3, 1), (9, 4, 1),

(10, 1, 1);

[Input Tables]

|

|-- Customers

|-- Products

|-- Orders

|-- OrderDetails

|

v

[CTEs]

|

|-- CustomerSpending

| |-- JOIN: Customers & Orders

| |-- GROUP BY: CustomerID, Name

| |-- Calculate: TotalSpending

|

|-- MonthlySales

| |-- Filter: Last 6 months

| |-- GROUP BY: Month

| |-- Calculate: MonthlySales

|

|-- BestSellingCategory

| |-- JOIN: Orders, OrderDetails & Products

| |-- GROUP BY: Month, Category

| |-- Calculate: TotalQuantity

| |-- Rank categories within each month

|

|-- InactiveCustomers

| |-- LEFT JOIN: Customers & Orders

| |-- GROUP BY: CustomerID, Name, Email

| |-- Filter: No orders in last 3 months

|

|-- CustomerSegmentAnalysis

| |-- LEFT JOIN: Customers & Orders

| |-- GROUP BY: Segment

| |-- Calculate: AvgOrderValue, NumberOfOrders

|

v

[Main Query]

|

|-- Combine results from all CTEs using UNION ALL

|-- Order results by Insight and Value

|

v

[Output]

Insight, Detail, Value

Problem Statement:

You're working for an e-commerce company that wants to analyze its sales data.

The company has three main tables: Customers, Orders, and Products. Your task is

to write a SQL query that provides the following insights:

Top 5 customers by total spending

Monthly sales trend for the past 6 months

Best-selling product category in each month

Customers who haven't made a purchase in the last 3 months

Average order value and number of orders for each customer segment

-- Top 5 customers by total spending

WITH CustomerSpending AS (

SELECT

c.CustomerID,

c.Name,

SUM(o.TotalAmount) AS TotalSpending

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerID, c.Name

ORDER BY TotalSpending DESC

LIMIT 5

),

-- Monthly sales trend for the past 6 months

MonthlySales AS (

SELECT

DATE\_FORMAT(OrderDate, '%Y-%m') AS Month,

SUM(TotalAmount) AS MonthlySales

FROM Orders

WHERE OrderDate >= DATE\_SUB(CURDATE(), INTERVAL 6 MONTH)

GROUP BY Month

ORDER BY Month

),

-- Best-selling product category in each month

BestSellingCategory AS (

SELECT

DATE\_FORMAT(o.OrderDate, '%Y-%m') AS Month,

p.Category,

SUM(od.Quantity) AS TotalQuantity,

RANK() OVER (PARTITION BY DATE\_FORMAT(o.OrderDate, '%Y-

%m') ORDER BY SUM(od.Quantity) DESC) AS CategoryRank

FROM OrderDetails od

JOIN Orders o ON od.OrderID = o.OrderID

JOIN Products p ON od.ProductID = p.ProductID

GROUP BY Month, p.Category

HAVING CategoryRank = 1

),

-- Customers who haven't made a purchase in the last 3 months

InactiveCustomers AS (

SELECT

c.CustomerID,

c.Name,

c.Email,

MAX(o.OrderDate) AS LastPurchaseDate

FROM Customers c

LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerID, c.Name, c.Email

HAVING MAX(o.OrderDate) < DATE\_SUB(CURDATE(), INTERVAL 3

MONTH)

OR MAX(o.OrderDate) IS NULL

),

-- Average order value and number of orders for each customer segment

CustomerSegmentAnalysis AS (

SELECT

c.Segment,

COUNT(o.OrderID) AS NumberOfOrders,

AVG(o.TotalAmount) AS AvgOrderValue

FROM Customers c

LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.Segment

)

-- Combine the results

SELECT 'Top 5 Customers by Spending' AS Insight, cs.Name AS Detail,

cs.TotalSpending AS Value

FROM CustomerSpending cs

UNION ALL

SELECT 'Monthly Sales Trend', ms.Month, ms.MonthlySales

FROM MonthlySales ms

UNION ALL

SELECT 'Best-Selling Product Category', bc.Month, bc.Category

FROM BestSellingCategory bc

UNION ALL

SELECT 'Inactive Customers (No Purchase in 3 Months)', ic.Name, ic.Email

FROM InactiveCustomers ic

UNION ALL

SELECT 'Customer Segment Analysis', csa.Segment, CONCAT('Avg Order

Value: ', csa.AvgOrderValue, ', Orders: ', csa.NumberOfOrders)

FROM CustomerSegmentAnalysis csa

ORDER BY Insight, Value DESC;

Sample Output

Insight Detail Value

Top 5 Customers by Spending John Doe 1634.96

Top 5 Customers by Spending Jane Smith 1664.96

Monthly Sales Trend 2023-05 1049.98

Best-Selling Product Category 2023-06 Clothing

Inactive Customers (No

Purchase)

Alice

Brown

alice@example.com

Insight Detail Value

Customer Segment Analysis Premium Avg Order Value: 1400.33, Orders:

2

QUERY-3:

CREATE TABLE employee\_sales (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(50),

department VARCHAR(50),

sales\_amount DECIMAL(10, 2),

sales\_date DATE

);

INSERT INTO employee\_sales (employee\_id, employee\_name, department,

sales\_amount, sales\_date) VALUES

(1, 'John Doe', 'Electronics', 1500.00, '2023-01-15'),

(2, 'Jane Smith', 'Clothing', 2000.00, '2023-01-16'),

(3, 'Mike Johnson', 'Electronics', 1800.00, '2023-01-17'),

(4, 'Emily Brown', 'Home Goods', 1200.00, '2023-01-18'),

(5, 'David Lee', 'Clothing', 2200.00, '2023-01-19'),

(6, 'Sarah Wilson', 'Electronics', 1600.00, '2023-01-20'),

(7, 'Tom Harris', 'Home Goods', 1300.00, '2023-01-21'),

(8, 'Lisa Chen', 'Clothing', 1900.00, '2023-01-22');

Problem Statement:

You are working with a retail company that wants to analyze its sales data across

different departments. They have provided you with a table containing employee

sales information. Your task is to write a SQL query that accomplishes the

following:

Calculate the total sales, average sales, and number of employees for each

department.

Rank the departments based on their total sales.

Display this information in a single result set, ordered by total sales descending.

WITH DepartmentSales AS (

SELECT

department,

SUM(sales\_amount) AS total\_sales,

AVG(sales\_amount) AS avg\_sales,

COUNT(employee\_id) AS employee\_count

FROM employee\_sales

GROUP BY department

),

RankedDepartments AS (

SELECT

department,

total\_sales,

avg\_sales,

employee\_count,

RANK() OVER (ORDER BY total\_sales DESC) AS department\_rank

FROM DepartmentSales

)

SELECT

department,

total\_sales,

avg\_sales,

employee\_count,

department\_rank

FROM RankedDepartments

ORDER BY total\_sales DESC;

Sample Output:

department total\_sales avg\_sales employee\_count department\_rank

Clothing 6100.00 2033.33 3 1

Electronics 4900.00 1633.33 3 2

Home Goods 2500.00

—----------------------------------------------------------------------------------------------------

WEEK-6

QUERY-1:

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(50)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(50),

DepartmentID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

CREATE TABLE Projects (

ProjectID INT PRIMARY KEY,

ProjectName VARCHAR(100),

DepartmentID INT,

StartDate DATE,

EndDate DATE,

EmployeeID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID),

FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)

);

Problem Statement: You're working with a company that manages projects across

different departments. They want to analyze project performance, employee

contributions, and department efficiency. Your task is to write a SQL query that:

1. Rank Employees Within Each Department Based on Number of Projects

Completed

WITH ProjectCounts AS (

SELECT

EmployeeID,

DepartmentID,

COUNT(ProjectID) AS ProjectCount

FROM Projects

GROUP BY EmployeeID, DepartmentID

)

SELECT

e.EmployeeID,

e.EmployeeName,

d.DepartmentName,

pc.ProjectCount,

ROW\_NUMBER() OVER (PARTITION BY e.DepartmentID ORDER BY

pc.ProjectCount DESC) AS RankInDepartment

FROM Employees e

JOIN ProjectCounts pc ON e.EmployeeID = pc.EmployeeID

JOIN Departments d ON e.DepartmentID = d.DepartmentID

ORDER BY d.DepartmentName, RankInDepartment;

2. Calculate the Average Project Duration for Each Department

WITH ProjectDurations AS (

SELECT

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration

FROM Projects

)

SELECT

d.DepartmentID,

d.DepartmentName,

AVG(pd.Duration) AS AvgProjectDuration

FROM ProjectDurations pd

JOIN Departments d ON pd.DepartmentID = d.DepartmentID

GROUP BY d.DepartmentID, d.DepartmentName;

3. Identify Projects That Took Longer Than the Department's Average

WITH ProjectDurations AS (

SELECT

ProjectID,

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration

FROM Projects

),

DepartmentAverages AS (

SELECT

DepartmentID,

AVG(Duration) AS AvgDuration

FROM ProjectDurations

GROUP BY DepartmentID

)

SELECT

p.ProjectID,

p.ProjectName,

d.DepartmentName,

pd.Duration,

da.AvgDuration

FROM Projects p

JOIN ProjectDurations pd ON p.ProjectID = pd.ProjectID

JOIN DepartmentAverages da ON pd.DepartmentID = da.DepartmentID

JOIN Departments d ON p.DepartmentID = d.DepartmentID

WHERE pd.Duration > da.AvgDuration

ORDER BY pd.Duration DESC;

4. Find the Top 3 Most Efficient Employees in Each Department (Based on

Average Project Duration)

WITH ProjectDurations AS (

SELECT

EmployeeID,

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration

FROM Projects

),

EmployeeAvgDurations AS (

SELECT

EmployeeID,

DepartmentID,

AVG(Duration) AS AvgDuration

FROM ProjectDurations

GROUP BY EmployeeID, DepartmentID

)

SELECT

e.EmployeeID,

e.EmployeeName,

d.DepartmentName,

ed.AvgDuration

FROM EmployeeAvgDurations ed

JOIN Employees e ON ed.EmployeeID = e.EmployeeID

JOIN Departments d ON ed.DepartmentID = d.DepartmentID

WHERE (

SELECT COUNT(\*)

FROM EmployeeAvgDurations ed2

WHERE ed2.DepartmentID = ed.DepartmentID

AND ed2.AvgDuration < ed.AvgDuration

) < 3

ORDER BY d.DepartmentName, ed.AvgDuration ASC;

5. Compare Each Project's Duration to the Previous Project in the Same

Department

WITH ProjectDurations AS (

SELECT

ProjectID,

DepartmentID,

DATEDIFF(EndDate, StartDate) AS Duration,

ROW\_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY

StartDate) AS RowNum

FROM Projects

)

SELECT

p.ProjectID,

p.ProjectName,

d.DepartmentName,

pd.Duration AS CurrentDuration,

LAG(pd.Duration) OVER (PARTITION BY pd.DepartmentID ORDER BY

pd.RowNum) AS PreviousDuration

FROM Projects p

JOIN ProjectDurations pd ON p.ProjectID = pd.ProjectID

JOIN Departments d ON p.DepartmentID = d.DepartmentID

ORDER BY d.DepartmentName, pd.RowNum;

QUERY- 2:

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100),

RegistrationDate DATE,

Segment VARCHAR(20)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

Name VARCHAR(100),

Category VARCHAR(50),

Price DECIMAL(10, 2)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

TotalAmount DECIMAL(10, 2),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE OrderDetails (

OrderID INT,

ProductID INT,

Quantity INT,

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

Insert sample data into Customers

INSERT INTO Customers (CustomerID, Name, Email, RegistrationDate,

Segment)

VALUES

(1, 'John Doe', 'john@example.com', '2022-01-15', 'Regular'),

(2, 'Jane Smith', 'jane@example.com', '2022-02-20', 'Premium'),

(3, 'Bob Johnson', 'bob@example.com', '2022-03-10', 'Regular'),

(4, 'Alice Brown', 'alice@example.com', '2022-04-05', 'Premium'),

(5, 'Charlie Davis', 'charlie@example.com', '2022-05-01', 'Regular');

-- Insert sample data into Products

INSERT INTO Products (ProductID, Name, Category, Price)

VALUES

(1, 'Laptop', 'Electronics', 999.99),

(2, 'Smartphone', 'Electronics', 599.99),

(3, 'T-shirt', 'Clothing', 19.99),

(4, 'Jeans', 'Clothing', 49.99),

(5, 'Book', 'Books', 14.99);

-- Insert sample data into Orders

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount)

VALUES

(1, 1, '2023-01-10', 1019.98),

(2, 2, '2023-02-15', 649.98),

(3, 3, '2023-03-20', 34.98),

(4, 4, '2023-04-25', 1049.98),

(5, 5, '2023-05-30', 64.98),

(6, 1, '2023-06-05', 614.98),

(7, 2, '2023-07-10', 1014.98),

(8, 3, '2023-08-15', 599.99),

(9, 4, '2023-09-20', 69.98),

(10, 5, '2023-10-25', 999.99);

-- Insert sample data into OrderDetails

INSERT INTO OrderDetails (OrderID, ProductID, Quantity)

VALUES

(1, 1, 1), (1, 3, 1),

(2, 2, 1), (2, 4, 1),

(3, 3, 1), (3, 5, 1),

(4, 1, 1), (4, 4, 1),

(5, 3, 1), (5, 5, 1),

(6, 2, 1), (6, 3, 1),

(7, 1, 1), (7, 5, 1),

(8, 2, 1),

(9, 3, 1), (9, 4, 1),

(10, 1, 1);

[Input Tables]

|

|-- Customers

|-- Products

|-- Orders

|-- OrderDetails

|

v

[CTEs]

|

|-- CustomerSpending

| |-- JOIN: Customers & Orders

| |-- GROUP BY: CustomerID, Name

| |-- Calculate: TotalSpending

|

|-- MonthlySales

| |-- Filter: Last 6 months

| |-- GROUP BY: Month

| |-- Calculate: MonthlySales

|

|-- BestSellingCategory

| |-- JOIN: Orders, OrderDetails & Products

| |-- GROUP BY: Month, Category

| |-- Calculate: TotalQuantity

| |-- Rank categories within each month

|

|-- InactiveCustomers

| |-- LEFT JOIN: Customers & Orders

| |-- GROUP BY: CustomerID, Name, Email

| |-- Filter: No orders in last 3 months

|

|-- CustomerSegmentAnalysis

| |-- LEFT JOIN: Customers & Orders

| |-- GROUP BY: Segment

| |-- Calculate: AvgOrderValue, NumberOfOrders

|

v

[Main Query]

|

|-- Combine results from all CTEs using UNION ALL

|-- Order results by Insight and Value

|

v

[Output]

Insight, Detail, Value

Problem Statement:

You're working for an e-commerce company that wants to analyze its sales data.

The company has three main tables: Customers, Orders, and Products. Your task is

to write a SQL query that provides the following insights:

Top 5 customers by total spending

Monthly sales trend for the past 6 months

Best-selling product category in each month

Customers who haven't made a purchase in the last 3 months

Average order value and number of orders for each customer segment

-- Top 5 customers by total spending

WITH CustomerSpending AS (

SELECT

c.CustomerID,

c.Name,

SUM(o.TotalAmount) AS TotalSpending

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerID, c.Name

ORDER BY TotalSpending DESC

LIMIT 5

),

-- Monthly sales trend for the past 6 months

MonthlySales AS (

SELECT

DATE\_FORMAT(OrderDate, '%Y-%m') AS Month,

SUM(TotalAmount) AS MonthlySales

FROM Orders

WHERE OrderDate >= DATE\_SUB(CURDATE(), INTERVAL 6 MONTH)

GROUP BY Month

ORDER BY Month

),

-- Best-selling product category in each month

BestSellingCategory AS (

SELECT

DATE\_FORMAT(o.OrderDate, '%Y-%m') AS Month,

p.Category,

SUM(od.Quantity) AS TotalQuantity,

RANK() OVER (PARTITION BY DATE\_FORMAT(o.OrderDate, '%Y-

%m') ORDER BY SUM(od.Quantity) DESC) AS CategoryRank

FROM OrderDetails od

JOIN Orders o ON od.OrderID = o.OrderID

JOIN Products p ON od.ProductID = p.ProductID

GROUP BY Month, p.Category

HAVING CategoryRank = 1

),

-- Customers who haven't made a purchase in the last 3 months

InactiveCustomers AS (

SELECT

c.CustomerID,

c.Name,

c.Email,

MAX(o.OrderDate) AS LastPurchaseDate

FROM Customers c

LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.CustomerID, c.Name, c.Email

HAVING MAX(o.OrderDate) < DATE\_SUB(CURDATE(), INTERVAL 3

MONTH)

OR MAX(o.OrderDate) IS NULL

),

-- Average order value and number of orders for each customer segment

CustomerSegmentAnalysis AS (

SELECT

c.Segment,

COUNT(o.OrderID) AS NumberOfOrders,

AVG(o.TotalAmount) AS AvgOrderValue

FROM Customers c

LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.Segment

)

-- Combine the results

SELECT 'Top 5 Customers by Spending' AS Insight, cs.Name AS Detail,

cs.TotalSpending AS Value

FROM CustomerSpending cs

UNION ALL

SELECT 'Monthly Sales Trend', ms.Month, ms.MonthlySales

FROM MonthlySales ms

UNION ALL

SELECT 'Best-Selling Product Category', bc.Month, bc.Category

FROM BestSellingCategory bc

UNION ALL

SELECT 'Inactive Customers (No Purchase in 3 Months)', ic.Name, ic.Email

FROM InactiveCustomers ic

UNION ALL

SELECT 'Customer Segment Analysis', csa.Segment, CONCAT('Avg Order

Value: ', csa.AvgOrderValue, ', Orders: ', csa.NumberOfOrders)

FROM CustomerSegmentAnalysis csa

ORDER BY Insight, Value DESC;

Sample Output

Insight Detail Value

Top 5 Customers by Spending John Doe 1634.96

Top 5 Customers by Spending Jane Smith 1664.96

Monthly Sales Trend 2023-05 1049.98

Best-Selling Product Category 2023-06 Clothing

Inactive Customers (No

Purchase)

Alice

Brown

alice@example.com

Insight Detail Value

Customer Segment Analysis Premium Avg Order Value: 1400.33, Orders:

2

QUERY-3:

CREATE TABLE employee\_sales (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(50),

department VARCHAR(50),

sales\_amount DECIMAL(10, 2),

sales\_date DATE

);

INSERT INTO employee\_sales (employee\_id, employee\_name, department,

sales\_amount, sales\_date) VALUES

(1, 'John Doe', 'Electronics', 1500.00, '2023-01-15'),

(2, 'Jane Smith', 'Clothing', 2000.00, '2023-01-16'),

(3, 'Mike Johnson', 'Electronics', 1800.00, '2023-01-17'),

(4, 'Emily Brown', 'Home Goods', 1200.00, '2023-01-18'),

(5, 'David Lee', 'Clothing', 2200.00, '2023-01-19'),

(6, 'Sarah Wilson', 'Electronics', 1600.00, '2023-01-20'),

(7, 'Tom Harris', 'Home Goods', 1300.00, '2023-01-21'),

(8, 'Lisa Chen', 'Clothing', 1900.00, '2023-01-22');

Problem Statement:

You are working with a retail company that wants to analyze its sales data across

different departments. They have provided you with a table containing employee

sales information. Your task is to write a SQL query that accomplishes the

following:

Calculate the total sales, average sales, and number of employees for each

department.

Rank the departments based on their total sales.

Display this information in a single result set, ordered by total sales descending.

WITH DepartmentSales AS (

SELECT

department,

SUM(sales\_amount) AS total\_sales,

AVG(sales\_amount) AS avg\_sales,

COUNT(employee\_id) AS employee\_count

FROM employee\_sales

GROUP BY department

),

RankedDepartments AS (

SELECT

department,

total\_sales,

avg\_sales,

employee\_count,

RANK() OVER (ORDER BY total\_sales DESC) AS department\_rank

FROM DepartmentSales

)

SELECT

department,

total\_sales,

avg\_sales,

employee\_count,

department\_rank

FROM RankedDepartments

ORDER BY total\_sales DESC;

Sample Output:

department total\_sales avg\_sales employee\_count department\_rank

Clothing 6100.00 2033.33 3 1

Electronics 4900.00 1633.33 3 2

Home Goods 2500.00

—------------------------------------------------------------------------------------

In Spring Boot, scopes define the lifecycle and visibility of beans, which are

objects managed by the Spring container. Here are the most common Spring bean

scopes and their uses:

1. Singleton Scope (Default Scope)

Description: A single instance of the bean is created for the entire Spring

container. This instance is shared by all clients requesting that bean.

Use Case: Stateless services, where the state is not tied to a particular

request or session (e.g., service classes in the business layer).

Example:

@Service

public class MyService {

// Singleton by default

}

2. Prototype Scope

Description: A new bean instance is created every time it is requested.

Use Case: When you need separate instances of the same bean (e.g., domain

objects with different states).

Example:

@Component

@Scope("prototype")

public class MyPrototypeBean {

// Prototype scoped bean

}

3. Request Scope (Web Applications)

Description: A new bean instance is created for each HTTP request. The

instance is available only for the duration of that request.

Use Case: For beans that hold request-specific state, like request parameters

or attributes.

Example:

@Component

@Scope(value = WebApplicationContext.SCOPE\_REQUEST)

public class MyRequestScopedBean {

// Bean for request-specific data

}

4. Session Scope (Web Applications)

Description: A new bean instance is created per HTTP session. The instance

persists for the entire session and is shared across multiple requests in the

same session.

Use Case: For session-specific data, such as user information during login.

Example:

@Component

@Scope(value = WebApplicationContext.SCOPE\_SESSION)

public class MySessionScopedBean {

// Bean for session-specific data

}

5. Application Scope (Web Applications)

Description: A single instance of the bean is created for the entire

ServletContext (application-wide scope).

Use Case: For application-wide objects, like configuration settings shared

across all sessions and requests.

Example:

@Component

@Scope(value = WebApplicationContext.SCOPE\_APPLICATION)

public class MyApplicationScopedBean {

// Bean for application-wide data

}

Usage of Annotations:

@Scope annotation is used to define a scope on a bean. Some web-specific

scopes (request, session, application) can also be defined with dedicated

annotations for better readability:

o @RequestScope

o @SessionScope

o @ApplicationScope

Weather-related spring boot application

1. WeatherConfig

The configuration class for the application, defining necessary beans and settings

(e.g., external API keys, base URLs, etc.).

@Configuration

public class WeatherConfig {

@Value("${weather.api.key}")

private String apiKey;

@Value("${weather.api.url}")

private String apiUrl;

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

public String getApiKey() {

return apiKey;

}

public String getApiUrl() {

return apiUrl;

}

}

Weather Record

A domain model or entity representing weather data. This could be used to store

weather data retrieved from a weather API.

@Entity

public class WeatherRecord {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String city;

private double temperature;

private double humidity;

private LocalDateTime timestamp;

// Getters and Setters

}

Weather Controller

The controller that handles HTTP requests related to weather data. This class will

interact with the service layer to fetch weather information.

@RestController

@RequestMapping("/api/weather")

public class WeatherController {

@Autowired

private WeatherService weatherService;

@GetMapping("/{city}")

public ResponseEntity<WeatherRecord> getWeather(@PathVariable String

city) {

WeatherRecord weather = weatherService.getWeatherByCity(city);

return ResponseEntity.ok(weather);

}

@PostMapping

public ResponseEntity<WeatherRecord> saveWeather(@RequestBody

WeatherRequestDTO request) {

WeatherRecord weather = weatherService.saveWeatherData(request);

return ResponseEntity.status(HttpStatus.CREATED).body(weather);

}

}

Weather Repository

A Spring Data repository interface for performing CRUD operations on the

WeatherRecord entity.

@Repository

public interface WeatherRepository extends JpaRepository<WeatherRecord,

Long> {

Optional<WeatherRecord> findByCity(String city);

}

Weather Request

A Data Transfer Object (DTO) class used to capture weather data input from a

client.

public class WeatherRequestDTO {

@NotBlank

private String city;

@Min(0)

private double temperature;

private double humidity;

// Getters and Setters

}

Weather Service

This is the service layer that handles the business logic for fetching and storing

weather data.

@Service

public class WeatherService {

@Autowired

private WeatherRepository weatherRepository;

@Autowired

private WeatherAPIClient weatherAPIClient; // Client for external API

public WeatherRecord getWeatherByCity(String city) {

// Call external API if no local record exists

return weatherRepository.findByCity(city)

.orElseGet(() -> weatherAPIClient.fetchWeather(city));

}

public WeatherRecord saveWeatherData(WeatherRequestDTO request) {

WeatherRecord record = new WeatherRecord();

record.setCity(request.getCity());

record.setTemperature(request.getTemperature());

record.setHumidity(request.getHumidity());

record.setTimestamp(LocalDateTime.now());

return weatherRepository.save(record);

}

}

Weather Exception

Custom exceptions for handling errors in the weather service (e.g., when city not

found or external API failures).

public class WeatherException extends RuntimeException {

public WeatherException(String message) {

super(message);

}

}

Weather Validator

A utility class or component that validates weather requests (e.g., validating city

names, temperature ranges).

@Component

public class WeatherValidator {

public void validate(WeatherRequestDTO request) {

if (request.getTemperature() < -100 || request.getTemperature() > 60) {

throw new WeatherException("Invalid temperature range.");

}

if (request.getCity().isBlank()) {

throw new WeatherException("City name cannot be empty.");

}

}

}

WeatherQuery

A query object for fetching specific weather data, for example, filtering by date or

location.

public class WeatherQuery {

private String city;

private LocalDate startDate;

private LocalDate endDate;

// Getters and Setters

}

Weather Request DTO

This is the Data Transfer Object (DTO) for receiving weather data from the client

in POST requests.

public class WeatherRequestDTO {

private String city;

private double temperature;

private double humidity;

// Constructor, Getters, Setters

}

Weather Request Logger

A service or aspect that logs all weather data requests, including input validation

and API calls, for auditing or debugging purposes.

@Aspect

@Component

public class WeatherRequestLogger {

private static final Logger logger =

LoggerFactory.getLogger(WeatherRequestLogger.class);

@Before("execution(\* com.example.weather.WeatherController.\*(..))")

public void logRequest(JoinPoint joinPoint) {

logger.info("Weather request: " + Arrays.toString(joinPoint.getArgs()));

}

}

Weather Validation Constants

Constants used for validation (e.g., max/min temperature limits, regex patterns for

city names).

public class WeatherValidationConstants {

public static final double MIN\_TEMPERATURE = -100.0;

public static final double MAX\_TEMPERATURE = 60.0;

public static final String CITY\_NAME\_REGEX = "^[a-zA-Z\\s]+$";

}

User Weather Performance

public class UserWeatherPerformanceCodes {

public static final String RESPONSE\_TIME\_METRIC =

"weather.response.time";

public static final String API\_CALL\_SUCCESS = "weather.api.call.success";

public static final String API\_CALL\_FAILURE = "weather.api.call.failure";

—-------------------------------------------------------------------------------------------------

Spring Security with JWT (JSON Web Tokens) in Spring Boot

When implementing JWT authentication in Spring Boot with Spring Security, it

involves several components, including authentication filters, token generation and

validation, and security configurations. Here’s an outline of the key concepts and

potential conflicts you may encounter.

1. JWT Filter Overview

A JWT filter intercepts HTTP requests, extracts the token, validates it, and sets the

authentication in the Spring Security context. This filter works in tandem with the

Spring Security configuration to handle authorization and authentication logic.

Steps to Implement JWT in Spring Security:

User Authentication: The user sends a login request, which contains

credentials (username, password).

Token Generation: If the credentials are valid, the server generates a JWT

and returns it to the user.

Token Validation: On subsequent requests, the JWT is sent by the user

(typically in the Authorization header). The JWT filter intercepts these

requests to validate the token.

Security Context: If the token is valid, the user is authenticated, and the

SecurityContext is updated with the authenticated user.

2. JWT Filter Implementation

Step-by-Step Implementation:

2.1 Create a JwtUtil class for token generation and validation

This utility class will handle token creation and validation.

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.stereotype.Component;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

@Component

public class JwtUtil {

private String SECRET\_KEY = "your\_secret\_key";

// Extract username from token

public String extractUsername(String token) {

return extractClaim(token, Claims::getSubject);

}

// Extract expiration date from token

public Date extractExpiration(String token) {

return extractClaim(token, Claims::getExpiration);

}

public <T> T extractClaim(String token, Function<Claims, T> claimsResolver)

{

final Claims claims = extractAllClaims(token);

return claimsResolver.apply(claims);

}

private Claims extractAllClaims(String token) {

return

Jwts.parser().setSigningKey(SECRET\_KEY).parseClaimsJws(token).getBody();

}

// Check if token has expired

private Boolean isTokenExpired(String token) {

return extractExpiration(token).before(new Date());

}

// Generate token

public String generateToken(String username) {

Map<String, Object> claims = new HashMap<>();

return createToken(claims, username);

}

private String createToken(Map<String, Object> claims, String subject) {

return Jwts.builder()

.setClaims(claims)

.setSubject(subject)

.setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \*

10))

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

}

// Validate token

public Boolean validateToken(String token, String username) {

final String extractedUsername = extractUsername(token);

return (extractedUsername.equals(username) && !isTokenExpired(token));

}

}

2.2 Create a JwtRequestFilter to Intercept Requests

The filter intercepts incoming requests, checks for the JWT in the Authorization

header, and validates it.

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

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import

org.springframework.security.web.authentication.WebAuthenticationDetailsSource

;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;

import io.jsonwebtoken.ExpiredJwtException;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@Component

public class JwtRequestFilter extends OncePerRequestFilter {

@Autowired

private UserDetailsService userDetailsService;

@Autowired

private JwtUtil jwtUtil;

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response, FilterChain chain)

throws ServletException, IOException {

final String authorizationHeader = request.getHeader("Authorization");

String username = null;

String jwt = null;

// Extract JWT token from Authorization header

if (authorizationHeader != null && authorizationHeader.startsWith("Bearer

")) {

jwt = authorizationHeader.substring(7);

try {

username = jwtUtil.extractUsername(jwt);

} catch (ExpiredJwtException e) {

// Handle expired JWT exception

System.out.println("JWT Token has expired");

}

}

// If the username is found and the user is not authenticated yet

if (username != null &&

SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails =

this.userDetailsService.loadUserByUsername(username);

// Validate the token and set authentication

if (jwtUtil.validateToken(jwt, userDetails.getUsername())) {

UsernamePasswordAuthenticationToken

usernamePasswordAuthenticationToken = new

UsernamePasswordAuthenticationToken(

userDetails, null, userDetails.getAuthorities());

usernamePasswordAuthenticationToken

.setDetails(new

WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenti

cationToken);

}

}

chain.doFilter(request, response);

}

}

2.3 Configure Spring Security

You need to add the JwtRequestFilter to the security filter chain and disable default

form-based authentication.

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

import org.springframework.context.annotation.Bean;

import org.springframework.security.authentication.AuthenticationManager;

import

org.springframework.security.config.annotation.authentication.builders.Authentica

tionManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import

org.springframework.security.config.annotation.web.configuration.EnableWebSec

urity;

import

org.springframework.security.config.annotation.web.configuration.WebSecurityCo

nfigurerAdapter;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.password.NoOpPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import

org.springframework.security.web.authentication.UsernamePasswordAuthenticatio

nFilter;

@EnableWebSecurity

public class SecurityConfigurer extends WebSecurityConfigurerAdapter {

@Autowired

private JwtRequestFilter jwtRequestFilter;

@Autowired

private MyUserDetailsService myUserDetailsService;

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception

{

auth.userDetailsService(myUserDetailsService);

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests()

.antMatchers("/authenticate").permitAll() // Allow unauthenticated access

to login route

.anyRequest().authenticated() // All other routes require authentication

.and().sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS); // No

session management (stateless)

http.addFilterBefore(jwtRequestFilter,

UsernamePasswordAuthenticationFilter.class); // Add JWT filter

}

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception

{

return super.authenticationManagerBean();

}

@Bean

public PasswordEncoder passwordEncoder() {

return NoOpPasswordEncoder.getInstance(); // Simple password encoder for

testing (replace with BCrypt)

}

}

—------------------------------------------------------------------------------------------------

HTML Basics

HTML (HyperText Markup Language) is used to structure content on the

web.

It uses tags enclosed in angle brackets, e.g., <tagname>.

HTML documents consist of nested elements:

<!DOCTYPE html>

<html>

<head> <!-- Meta info (title, styles, etc.) -->

<title>Page Title</title>

</head>

<body> <!-- Page content -->

<h1>This is a Heading</h1>

<p>This is a paragraph.</p>

</body>

</html>

Common HTML Elements

Headings: <h1> to <h6>, where <h1> is the largest.

Paragraph: <p>, used for text content.

Links: <a href="URL">Link text</a>, where href defines the link's

destination.

Images: <img src="URL" alt="description">, where src is the image source.

Lists:

o Ordered: <ol><li>Item</li></ol>

o Unordered: <ul><li>Item</li></ul>

Tables: <table><tr><td>Data</td></tr></table>

Attributes

Attributes provide additional information about elements.

Example: <a href="https://example.com" target="\_blank">Visit

Example</a>.

o href: Link destination.

o target="\_blank": Opens in a new tab.

Forms

Used to collect user input:

<form action="/submit" method="post">

<label for="name">Name:</label>

<input type="text" id="name" name="name">

<input type="submit" value="Submit">

</form>

Types of Inputs: text, password, radio, checkbox, submit, etc.

Form Methods: GET (visible in URL) and POST (secure).

Div and Span

<div>: Block-level element, used for grouping content.

<span>: Inline element, used for styling parts of text or small parts of a

document.

Semantic HTML

Semantic tags describe their meaning in a more human- and machine-

readable way:

o <header>, <nav>, <main>, <section>, <article>, <footer>, etc.

Comments

HTML comments are not displayed in the browser and help in organizing

code:

<!-- This is a comment -->

HTML5 New Features

Media: <audio>, <video> for multimedia embedding.

<video controls>

<source src="movie.mp4" type="video/mp4">

</video>

CSS

CSS (Cascading Style Sheets) is used to style and layout web pages.

It controls the visual presentation of HTML elements.

CSS Syntax

CSS is composed of selectors and declarations.

selector {

property: value;

}

Selector targets the HTML element(s), and the declaration defines what

style to apply (e.g., color, size).

p {

color: blue;

font-size: 16px;

}

Ways to Add CSS

Inline CSS: Within the HTML tag using the style attribute.

<p style="color: blue;">This is a blue paragraph.</p>

Internal CSS: Inside the <style> tag in the <head> section of HTML.

<style>

p { color: blue; }

</style>

External CSS: In a separate .css file linked to HTML.

<link rel="stylesheet" href="styles.css">

Selectors

Element Selector: Targets all elements of a specific type.

h1 { color: red; }

Class Selector: Targets elements with a specific class. Use . for class.

.myClass { color: green; }

html

Copy code

<p class="myClass">This text is green.</p>

ID Selector: Targets a single element with a specific ID. Use # for ID.

#myID { color: orange; }

html

Copy code

<p id="myID">This text is orange.</p>

Group Selector: Target multiple elements at once.

h1, p { color: blue; }

Box Model

Every HTML element is treated as a box, consisting of:

o Content: The actual content.

o Padding: Space between the content and the border.

o Border: Surrounds the padding.

o Margin: Space outside the border.

p {

padding: 10px;

border: 1px solid black;

margin: 20px;

}

Colors

You can define colors using:

o Name: color: red;

o HEX: color: #ff0000;

o RGB: color: rgb(255, 0, 0);

o RGBA (with transparency): color: rgba(255, 0, 0, 0.5);

Font and Text Properties

Font family: Specifies the font.

p { font-family: Arial, sans-serif; }

Font size: Controls the size of the text.

h1 { font-size: 24px; }

Font weight: Sets the boldness (e.g., normal, bold, 100 to 900).

Text alignment: Aligns text (left, center, right, justify).

Text decoration: Underline, overline, or strikethrough (none, underline).

Layout and Positioning

Display: Defines how an element is displayed (block, inline, inline-block,

none).

div { display: block; }

span { display: inline; }

Position: Controls how elements are positioned in the document flow.

o Static (default): Element flows normally.

o Relative: Positioned relative to its normal position.

o Absolute: Positioned relative to its nearest positioned ancestor.

o Fixed: Positioned relative to the viewport, stays in the same place on

scroll.

div { position: absolute; top: 20px; left: 50px; }

Flexbox: Layout model for flexible and responsive designs.

.container {

display: flex;

justify-content: space-between;

}

Pseudo-Classes and Pseudo-Elements

Pseudo-Classes: Apply styles to elements in a specific state.

a:hover { color: red; } /\* Link turns red on hover \*/

Pseudo-Elements: Style specific parts of an element.

p::first-letter { font-size: 2em; } /\* Style the first letter \*/

Media Queries

Media queries make designs responsive based on the screen size or device

type.

@media (max-width: 600px) {

body { background-color: lightblue; }

}

ANGULAR

Component Anatomy in Angular

An Angular component is the core building block of the application. It consists of

several parts:

Component Class: Contains the logic of the component. It is a TypeScript

class decorated with the @Component decorator.

Template: Defines the HTML structure (view) of the component.

Styles: CSS/SCSS for styling the component.

Metadata: Information about the component, such as the selector, template,

and styles.

Example:

import { Component } from '@angular/core';

@Component({

selector: 'app-example', // Component's tag name

templateUrl: './example.component.html', // HTML template

styleUrls: ['./example.component.css'] // CSS styles

})

export class ExampleComponent {

title: string = 'Hello, Angular!'; // Component logic

}

Breakdown:

Selector: Defines the HTML tag used to include this component (<app-

example></app-example>).

Template: Can be defined using an external HTML file (templateUrl) or

inline with backticks (template).

Styles: CSS/SCSS for the component, either external (styleUrls) or inline.

Data Binding Types in Angular

Angular provides various types of data binding between the component’s class and

the view (template):

a) Interpolation (One-way binding: Component → View)

Used to bind data from the component class to the template (view).

Syntax: {{ expression }}.

Example:

<h1>{{ title }}</h1> <!-- title value comes from the component -->

b) Property Binding (One-way binding: Component → View)

Binds a component class property to an HTML element property or directive

input.

Syntax: [property]="expression".

Example:

html

Copy code

<input [value]="name">

c) Event Binding (One-way binding: View → Component)

Binds a DOM event to a method in the component class.

Syntax: (event)="expression".

Example:

<button (click)="onClick()">Click me</button>

typescript

Copy code

onClick() {

console.log('Button clicked!');

}

d) Two-way Data Binding

Combines both property binding and event binding, where changes in the

view are reflected in the component class, and vice versa.

Syntax: [(ngModel)]="property".

Example:

<input [(ngModel)]="name">

<p>Your name is: {{ name }}</p>

typescript

Copy code

name: string = '';

ngModel in Angular

The ngModel directive is used for two-way data binding in Angular. It binds the

form inputs (like input, textarea, select) to the component's model (data).

Usage of ngModel:

Two-way data binding allows automatic synchronization of data between

the input field and the component property.

Example:

<input [(ngModel)]="username">

<p>Username: {{ username }}</p>

typescript

Copy code

export class ExampleComponent {

username: string = '';

}

—-------------------------------------------------------------------------------

Modules, state management, and component anatomy in Angular:

1. Modules

Purpose: Organize an Angular app into cohesive blocks.

Structure: An Angular module (like AppModule) groups related

components, directives, services, and pipes.

Feature Modules: These are specific to features (e.g., task management)

and are imported/exported as needed.

2. State Management

Component State: Managed within components, suitable for isolated or

single-use cases.

Application State: Libraries like NgRx or services help handle shared state

across components, providing a single source of truth.

3. Component Anatomy

Template: Defines the component’s HTML structure.

Class (.ts): Manages component logic, properties, and methods.

Metadata (@Component): Configures the component’s selector, template,

styles, etc.

Styles: Contains component-specific CSS or SCSS.

Building a task management application in Angular with example code

snippets:

Setting Up the Project and Task Module

ng new task-manager

cd task-manager

ng generate module tasks --routing

This creates a new Angular project and a dedicated TasksModule for managing

tasks.

Creating Components

Generate components for task features.

ng generate component tasks/task-list

ng generate component tasks/task-detail

ng generate component tasks/task-form

Task List Component Example

The task-list.component.ts will display a list of tasks.

// task-list.component.ts

import { Component, OnInit } from '@angular/core';

import { TaskService } from '../task.service';

import { Task } from '../task.model';

@Component({

selector: 'app-task-list',

templateUrl: './task-list.component.html',

})

export class TaskListComponent implements OnInit {

tasks: Task[] = [];

constructor(private taskService: TaskService) {}

ngOnInit() {

this.tasks = this.taskService.getTasks();

}

}

And the HTML (task-list.component.html) might look like this:

<ul>

<li \*ngFor="let task of tasks">

{{ task.title }} - {{ task.status }}

</li>

</ul>

Task Service for State Management

A service can handle task data and manage state across components.

// task.service.ts

import { Injectable } from '@angular/core';

import { Task } from './task.model';

@Injectable({

providedIn: 'root',

})

export class TaskService {

private tasks: Task[] = [

{ id: 1, title: 'Buy groceries', status: 'pending' },

{ id: 2, title: 'Check emails', status: 'completed' },

];

getTasks(): Task[] {

return [...this.tasks];

}

addTask(task: Task) {

this.tasks.push(task);

}

}

Task Model

Define a model to structure task data.

// task.model.ts

export interface Task {

id: number;

title: string;

status: 'pending' | 'completed';

}

Task Form Component Example

A component to add new tasks.

// task-form.component.ts

import { Component } from '@angular/core';

import { TaskService } from '../task.service';

import { Task } from '../task.model';

@Component({

selector: 'app-task-form',

templateUrl: './task-form.component.html',

})

export class TaskFormComponent {

title = '';

constructor(private taskService: TaskService) {}

addTask() {

const newTask: Task = { id: Date.now(), title: this.title, status: 'pending' };

this.taskService.addTask(newTask);

this.title = '';

}

}

The task-form.component.html template:

<input [(ngModel)]="title" placeholder="New Task" />

<button (click)="addTask()">Add Task</button>

Registering Components in Task Module

Finally, ensure all components are declared in the TasksModule.

// tasks.module.ts

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

import { TaskListComponent } from './task-list/task-list.component';

import { TaskFormComponent } from './task-form/task-form.component';

import { TaskDetailComponent } from './task-detail/task-detail.component';

@NgModule({

declarations: [TaskListComponent, TaskFormComponent,

TaskDetailComponent],

imports: [CommonModule],

})

export class TasksModule {}

—----------------------------------------------------------------------------

**Validation in a Spring Boot application**, focusing on validating user data in a UserController, using a UserDTO with validation annotations, creating validation properties, and configuring a validation setup.

Spring provides robust support for validating user input and ensuring data integrity through:

∙ **Bean Validation (JSR 380)**: A Java specification that defines a metadata model and API for validating Java beans.

∙ **Hibernate Validator**: The reference implementation of Bean Validation, integrated seamlessly with Spring.

∙ **Spring MVC Integration**: Allows validation annotations to be applied directly to controller methods.

**Key Annotations:**

∙ @Valid: Triggers validation on an object.

∙ @Validated: Used at the class level to indicate that the class should participate in validation.

**2. Setting Up Validation Dependencies**

To use validation in your Spring application, ensure that the necessary dependencies are included in your project.

**Using Maven**

Add the following dependencies to your pom.xml:

<dependencies>

<!-- Spring Boot Starter Web -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Spring Boot Starter Validation -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

<!-- (Optional) Lombok for reducing boilerplate code -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

**User Controller (UserController)**

The UserController handles HTTP requests for user-related operations, such as creating or updating a user. This is where validation errors from the UserDTO will be captured and processed.

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.annotation.Validated;

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

import javax.validation.Valid;

@RestController

@RequestMapping("/api/users")

@Validated

public class UserController {

private final UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

@PostMapping("/register")

public ResponseEntity<String> registerUser(@Valid @RequestBody UserDTO userDTO) {

userService.registerUser(userDTO);

return new ResponseEntity<>("User registered successfully", HttpStatus.CREATED);

}

// Other endpoints like update, delete, get user

}

∙ **@RestController**: Handles HTTP requests and returns responses directly. ∙ **@RequestMapping**: Sets a base URL for user-related endpoints. ∙ **@PostMapping**: Maps to the POST /api/users/register endpoint. ∙ **@Valid**: Triggers validation on UserDTO before executing the method. ∙ **@Validated**: At the controller level, to ensure validations are enabled.

**User DTO (UserDTO)**

The UserDTO is a Data Transfer Object that represents user data coming from client requests. We add validation annotations to ensure that the incoming data meets specified criteria.

import javax.validation.constraints.Email;

import javax.validation.constraints.NotBlank;

import javax.validation.constraints.Size;

public class UserDTO {

@NotBlank(message = "{user.name.notblank}")

@Size(min = 3, max = 50, message = "{user.name.size}") private String name;

@NotBlank(message = "{user.email.notblank}")

@Email(message = "{user.email.valid}")

private String email;

@NotBlank(message = "{user.password.notblank}")

@Size(min = 6, message = "{user.password.size}")

private String password;

// Getters and Setters

}

∙ **@NotBlank**: Ensures the field is not null or empty.

∙ **@Size**: Sets minimum and/or maximum size for the field. ∙ **@Email**: Validates that the field is a valid email format. ∙ **message**: Points to the validation messages in Validation Properties for custom error messages.

**Validation Properties (Validation Messages)**

Define validation messages in a properties file (e.g., messages.properties) to provide more readable or localized error messages.

**src/main/resources/messages.properties**

user.name.notblank=Name cannot be blank

user.name.size=Name must be between 3 and 50 characters user.email.notblank=Email cannot be blank

user.email.valid=Invalid email format

user.password.notblank=Password cannot be blank

user.password.size=Password must be at least 6 characters long

To set this as the message source in Spring, configure it in the application.properties file:

**src/main/resources/application.properties**

spring.messages.basename=messages

This tells Spring to use messages.properties for validation messages.

**Validation Configuration (ValidationConfig)**

Create a configuration class to customize validation messages or add additional settings.

import org.springframework.context.MessageSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration; import org.springframework.context.support.ResourceBundleMessageSource;

import

org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration

public class ValidationConfig {

@Bean

public MessageSource messageSource() {

ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();

messageSource.setBasename("messages");

messageSource.setDefaultEncoding("UTF-8");

return messageSource;

}

@Bean

public LocalValidatorFactoryBean validator() {

LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean(); bean.setValidationMessageSource(messageSource());

return bean;

}

}

∙ **MessageSource**: Defines the base name (messages) and encoding for validation messages.

∙ **LocalValidatorFactoryBean**: Configures Spring to use messageSource as the validation message source.

**Handling Validation Errors**

When validation fails, Spring Boot returns a 400 Bad Request with validation error details. To customize this behavior, we can use an @ExceptionHandler in the controller or a @ControllerAdvice.

**Example of Handling Errors with @ControllerAdvice**

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.FieldError;

import org.springframework.web.bind.MethodArgumentNotValidException; import org.springframework.web.bind.annotation.ControllerAdvice; import org.springframework.web.bind.annotation.ExceptionHandler;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class ValidationExceptionHandler {

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResponseEntity<Map<String, String>>

handleValidationExceptions(MethodArgumentNotValidException ex) { Map<String, String> errors = new HashMap<>();

ex.getBindingResult().getAllErrors().forEach((error) -> {

String fieldName = ((FieldError) error).getField();

String errorMessage = error.getDefaultMessage();

errors.put(fieldName, errorMessage);

});

return new ResponseEntity<>(errors, HttpStatus.BAD\_REQUEST); }

}

∙ **@ControllerAdvice**: Applies globally to handle exceptions across the whole application.

∙ **@ExceptionHandler**: Catches MethodArgumentNotValidException, which is thrown for validation failures.

∙ **Error Mapping**: Maps validation errors for each field to make them readable in the response.

—-----------------------------------------------------------------------------------------------------

Document for creating a simple **To-Do Application** using Angular. **Objective**

Build a basic to-do application where users can:

∙ Add new tasks.

∙ Mark tasks as complete or incomplete.

∙ Delete tasks.

**Set up Models**

Define a model for Task in a new file task.model.ts to represent each to-do item. src/app/task.model.ts:

export interface Task {

id: number;

title: string;

completed: boolean;

}

**Implement To-Do Component Logic**

In the TodoComponent, define properties and methods to manage tasks. src/app/todo/todo.component.ts:

import { Component } from '@angular/core';

import { Task } from '../task.model';

@Component({

selector: 'app-todo',

templateUrl: './todo.component.html',

styleUrls: ['./todo.component.css']

})

export class TodoComponent {

tasks: Task[] = [];

newTask: string = '';

addTask() {

if (this.newTask.trim()) {

const newTask: Task = {

id: Date.now(),

title: this.newTask,

completed: false

};

this.tasks.push(newTask);

this.newTask = ''; // Reset input field

}

}

deleteTask(taskId: number) {

this.tasks = this.tasks.filter(task => task.id !== taskId); }

toggleTaskCompletion(task: Task) {

task.completed = !task.completed;

}

}

**Create the To-Do Component Template**

Defining the HTML structure for the todo component. src/app/todo/todo.component.html:

<div class="todo-container">

<h1>Simple To-Do Application</h1>

<input

type="text"

[(ngModel)]="newTask"

placeholder="Enter a new task"

(keyup.enter)="addTask()"

/>

<button (click)="addTask()">Add Task</button>

<ul>

<li \*ngFor="let task of tasks">

<input

type="checkbox"

[(ngModel)]="task.completed"

(change)="toggleTaskCompletion(task)"

/>

<span [class.completed]="task.completed">{{ task.title }}</span> <button (click)="deleteTask(task.id)">Delete</button> </li>

</ul>

</div>

**Add Basic CSS Styles**

src/app/todo/todo.component.css:

.todo-container {

max-width: 400px;

margin: 20px auto;

padding: 20px;

border: 1px solid #ccc;

border-radius: 8px;

text-align: center;

}

input[type="text"] {

padding: 8px;

width: 70%;

margin-right: 8px;

}

button {

padding: 8px;

}

ul {

list-style-type: none;

padding: 0;

}

li {

display: flex;

align-items: center;

margin-top: 10px;

}

li .completed {

text-decoration: line-through;

color: gray;

}

**Add FormsModule**

To use [(ngModel)] for two-way binding, we need to import FormsModule in the main module.

src/app/app.module.ts:

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser'; import { FormsModule } from '@angular/forms'; // Import FormsModule

import { AppComponent } from './app.component';

import { TodoComponent } from './todo/todo.component';

@NgModule({

declarations: [

AppComponent,

TodoComponent

],

imports: [

BrowserModule,

FormsModule // Add FormsModule here

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Display To-Do Component in App Component**

Update the app.component.html to display the todo component.

src/app/app.component.html:

<app-todo></app-todo>

**Run the Application**

Open browser and go to http://localhost:4200 to see the application in action.

—-----------------------------------------------------------------------------------------------------------------

**Document for Simple Shopping Application** using Angular. **Objective**

Create a basic shopping application with the following features:

∙ Display a list of products.

∙ Allow users to add products to a shopping cart.

∙ Display the shopping cart and manage quantities.

**Create Product Model**

Define a model for Product in a new file called product.model.ts. This will represent each product.

Create src/app/product.model.ts:

export interface Product {

id: number;

name: string;

price: number;

description: string;

}

**Implement Product List Component Logic**

src/app/product-list/product-list.component.ts:

import the Product model. Then, define a list of products for display.

import { Component } from '@angular/core';

import { Product } from '../product.model';

@Component({

selector: 'app-product-list',

templateUrl: './product-list.component.html',

styleUrls: ['./product-list.component.css']

})

export class ProductListComponent {

products: Product[] = [

{ id: 1, name: 'Laptop', price: 900, description: 'A high-performance laptop' }, { id: 2, name: 'Smartphone', price: 500, description: 'A smartphone with excellent features' },

{ id: 3, name: 'Headphones', price: 150, description: 'Noise-cancelling headphones' },

];

// Event to notify cart about the selected product

addToCart(product: Product) {

// Custom event logic will be added in the app component later }

}

**Design the Product List Template**

src/app/product-list/product-list.component.html:

<div class="product-list">

<h2>Product List</h2>

<div \*ngFor="let product of products" class="product-item">

<h3>{{ product.name }}</h3>

<p>{{ product.description }}</p>

<p>Price: ${{ product.price }}</p>

<button (click)="addToCart(product)">Add to Cart</button> </div>

</div>

**Implement Cart Component Logic**

Define an empty array to hold cart items, and implement methods to add products, manage quantities, and calculate the total price.

src/app/cart/cart.component.ts.

import { Component } from '@angular/core';

import { Product } from '../product.model';

interface CartItem {

product: Product;

quantity: number;

}

@Component({

selector: 'app-cart',

templateUrl: './cart.component.html',

styleUrls: ['./cart.component.css']

})

export class CartComponent {

cartItems: CartItem[] = [];

addToCart(product: Product) {

const item = this.cartItems.find(cartItem => cartItem.product.id === product.id); if (item) {

item.quantity++;

} else {

this.cartItems.push({ product, quantity: 1 });

}

}

removeFromCart(productId: number) {

this.cartItems = this.cartItems.filter(item => item.product.id !== productId); }

getTotalPrice(): number {

return this.cartItems.reduce((total, item) => total + item.product.price \* item.quantity, 0);

}

}

**Design the Cart Component Template**

src/app/cart/cart.component.html:

<div class="cart">

<h2>Shopping Cart</h2>

<ul>

<li \*ngFor="let item of cartItems">

<span>{{ item.product.name }}</span>

<span>Quantity: {{ item.quantity }}</span>

<span>Price: ${{ item.product.price \* item.quantity }}</span> <button (click)="removeFromCart(item.product.id)">Remove</button> </li>

</ul>

<p \*ngIf="cartItems.length === 0">Cart is empty</p>

<h3>Total: ${{ getTotalPrice() }}</h3>

</div>

**Integrate Components in App Component**

Pass product data between components by binding ProductListComponent and CartComponent in AppComponent.

src/app/app.component.ts:

import { Component } from '@angular/core';

import { Product } from './product.model';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

cart: Product[] = [];

onAddToCart(product: Product) {

// Logic to add product to cart

}

}

src/app/app.component.html:

<div class="app-container">

<app-product-list (addToCart)="onAddToCart($event)"></app-product-list> <app-cart [cartItems]="cartItems"></app-cart>

</div>

**Add Styles**

src/app/app.component.css:

.app-container {

display: flex;

justify-content: space-between;

padding: 20px;

}

.product-list, .cart {

width: 45%;

border: 1px solid #ddd;

padding: 20px;

border-radius: 8px;

}

.product-item {

margin-bottom: 15px;

}

button {

padding: 5px 10px;

margin-top: 10px;

}

**Run the Application**

Run the Angular development server:

ng serve

Open browser and go to http://localhost:4200 to see the application in action.

—--------------------------------------------------------------------------------------------

PROJECT

project—<https://trello.com/b/NBwdGgHw/service-center-web-application-project>

GitHub Repo-<https://github.com/1rithwik/WebApplication_Project/tree/rithwik>