Name : Rithwik Reddy Rokkam

Github Link - 1rithwik (github.com)

Index

Click on the them to go to respective dates.

Github Link - [1rithwik (github.com)]

The topics understood on (16-09-2024)

1. What is meant by lifecycle - it is the sequence of changes that occurs as on development continues from initial stage to final stage.

2. what is development - the process which grows as on moving with time in positive perspective.

3. what is a software - which is used to operate computers using specific tasks or instructions.

4. why do we use SDLC

- to bring the idea to existence to solve the real world problem

- helps in planning what to do before itself, helps in determining costs and decisions, helps in getting better quality of product.

5. Steps followed in SDLC

- Requirement collection: Gathering all the requirements from the clients to develop a product. Can be done by discussing with them, conducting interviews, surveys, conducting polls, etc

- Design: Here software architecture is derived, which is used for system implementation.

- Implementation: Here the developers do the required work

- Testing: Testing is done for the developed code, whether the application behaves as expected, and when errors found, would be mentioned.

- Deployment and Maintenance: Deploys to clients where they can use the product. If any issue occurred from now on the maintenance team will help in solving the problem.

Here they have given a scenario and for this the I came up with this life-cycle:

1.Requirements gathering: The retired accounting faculty will approach the author to investigate the purpose.

Eg- on service of data transmission, monitoring, lifeline features.

2. Design: formulate a design based on requirements how the software system should be operated, what data needs to be collected, how the technology should work.

3. Implementation: development begins

4. Testing: testing for functionality, performance

5. Deployment and maintenance: Deployed for the use in Home Health facility. Routine inspections performed, updates or improvements will be implemented.

Different types of models:

1. waterfall model: This is a linear flow, next step is started only after completing the before step.

Cannot used for large projects.

Simple to implement

2. iterative model: first done development in small scale, later additional features are designed and added to the software.

In every iteration new features are add which improves the functionality of product.

3. Prototype model: This model is the one in which the prototype is developed prior to the actual product.

This prototype is shown to client, for feedback.

Feedbacks are implemented and the customer again reviews the prototype for any change. This process goes on until the customer accepts the model.

4. Spiral model: this model combines of iterative and waterfall models.

Risk management is a primary focus, with each iteration including a risk assessment phase to address potential issues.

Feedback is gathered at each iteration, allowing for adjustments based on customer input and risk evaluation.

5. V model: In this model, the development and testing stages go in parallel. This model consists of the verification stage on one side, and the validation stage on the other side.

V –model is good for smaller projects wherein the requirements is given in the early stage.

Agile meaning Adaptive nature

Scrum meaning uniting together and working on the same goal. Scrum is the subset of Agile.

Agile development cycle -

Concept

Inception

Design, Development, Construction, Testing, and Integration

Implementation Deployment

Retirement

Scrum stages -

Initiate - form scrum team

Plan and Estimate - create and estimate tasks

Implement - implementing and conducting Daily Standup

Review and Retrospect - Demonstrate and Validate Sprint

Release - retrospect project

Agile Development-

Agile software development is an iterative approach that focuses on collaboration, customer feedback, and small, rapid releases.

Purpose: Enhance flexibility and responsiveness to change.

Principles of Agile

- **Customer Satisfaction**: Deliver valuable software early and continuously.
- **Embrace Change**: Welcome changing requirements, even late in development.
- **Frequent Delivery**: Deliver working software frequently, from a couple of weeks to a couple of months.
- **Collaboration**: Close cooperation between business stakeholders and developers.

Agile Methodologies

- **Scrum**: Framework for managing complex projects with defined roles (Scrum Master, Product Owner, Development Team).
- **Kanban**: Visual management tool for optimizing workflow and limiting work in progress.
- **Extreme Programming (XP)**: Focuses on technical practices and customer involvement to improve software quality.

Scrum methodology -

Scrum is an Agile framework for managing complex projects. It emphasizes iterative progress, collaboration, and adaptability.

Purpose: To deliver high-quality software in a flexible, efficient manner.

Scrum Framework

- **Roles**:
    - **Product Owner**:
        - Defines the product vision and manages the product backlog.

        - Prioritizes features based on business value and stakeholder feedback.
    - **Scrum Master**:
        - Facilitates Scrum processes and removes impediments.
        - Serves as a coach for the team, ensuring adherence to Agile principles.
    - **Development Team**:
        - A cross-functional group that delivers the product increment.
        - Typically consists of 3 to 9 members with diverse skills.

Key Artifacts

- **Product Backlog**:
    - A prioritized list of features, enhancements, and fixes required for the product.
    - Continuously updated based on stakeholder feedback and changing requirements.
- **Sprint Backlog**:
    - A subset of the product backlog selected for a specific sprint.
    - Represents the work the team commits to completing during the sprint.
- **Increment**:
    - The sum of all completed backlog items at the end of a sprint.
    - Must be in a usable state, regardless of whether the Product Owner decides to release it.

Scrum Events

- **Sprint**:
    - A time-boxed iteration, usually lasting 1-4 weeks, during which a potentially shippable product increment is created.
- **Sprint Planning**:
    - A meeting at the start of each sprint where the team decides what to work on and how to achieve it.
- **Daily Scrum (Stand-up)**:
    - A short, daily meeting (15 minutes) for team members to discuss progress, plans, and obstacles.
- **Sprint Review**:
    - A meeting at the end of the sprint to showcase the increment and gather feedback from stakeholders.
- **Sprint Retrospective**:
    - A reflection meeting to identify what went well, what didn't, and how the team can improve in the next sprint.

The topics and concepts understood on (17-09-2024)

All the git commands

1. git init -

It is used for initializing a new Git repository in a directory.
I used it by creating a directory and then initializing the repo.

mkdir my-project

cd my-project

git init

After this using ‘echo’ command I created readme file and then added that file and then committed and then pushed it to the origin.

git remote add origin https://github.com/1rithwik/17sep.git

git branch -M main

git push -u origin main

2. Cloning

This is done using-

git clone “we paste the url”

this will create a copy of the repo in our local machine.

3. Adding and Committing

git add - this is to add the changes to the file. We use either ‘.’ or even ‘file name’ in using add command.

git commit - This command is used to finalize the changes. We also mention the message while committing.

git commit -m “message”

4. Status

Shows the current state of the working directory and staging area. It also lists files that have been modified but not yet staged for commit. It shows files that are staged and ready to be committed.

It also indicates the current branch and if it’s ahead, behind, or diverged from the remote branch.

5. Pull

git pull

It retrieves updates from a remote repository to your local repository. **It** automatically merges the fetched changes into the current branch. It performs both git fetch (to download changes) and git merge (to apply those changes) in one step. This command is essential for keeping your local repository up-to-date with the latest changes from the remote repository.

6. Push

The git push command is used to upload local changes to a remote repository. Depending on the scenario, you might use different options or variations of this command. Here's an overview of various ways to use git push:

1. Push to Default Remote Repository
    - Command: git push
    - Description: Pushes changes from the current branch to the remote branch with the same name on the default remote repository (usually origin).
    - Usage: Used when you want to update the remote branch that your local branch is tracking.
2. **Push to a Specific Remote Repository**
    - Command: git push <remote> <branch>
    - Description: Pushes changes to a specified remote repository and branch.
    - Example: Pushes changes from the local branch feature branch to the remote repository origin on the branch feature-branch.
3. Push All Branches
    - Command: git push --all <remote>
    - Description: Pushes all local branches to the specified remote repository.
4. **Force Push**
    - Command: git push --force
    - Usage: Use with caution, as it can overwrite changes on the remote branch and affect other collaborators.

7. Fetch

The git fetch command is used to retrieve changes from a remote repository without merging them into your local branch. This command updates your local references to match the remote repository.

Fetch from Default Remote

- Command: git fetch

- Description: Retrieves updates from the default remote repository (usually named origin) and updates your local remote-tracking branches.

Fetch from a Specific Remote

- Command: git fetch <remote>
- Description: Retrieves updates from a specified remote repository and updates the corresponding remote-tracking branches.

Fetch All Branches

- Command: git fetch --all
- Description: Retrieves updates from all configured remote repositories and updates all remote-tracking branches.

8. Branch

The git branch command lists , creates , or deletes branches in your Git repository. It helps manage and navigate different lines of development.

1. List Branches:
    - Command: git branch
    - Usage: Displays a list of all local branches in the repository, highlighting the currently active branch.
2. Create a New Branch:
    - Command: git branch <branch-name>
    - Usage: Creates a new branch with the specified name, based on the current branch's state.

3. to delete branch:

git branch -d branch-name

9. Checkout

The git checkout command is used to switch branches or restore working tree files. It updates your working directory to match the specified branch or commit.

1. Switch to a Different Branch:
    - Command: git checkout <branch-name>
    - Usage: Switches to the specified branch, updating the working directory to match the branch’s state.
2. Create and Switch to a New Branch:
    - Command: git checkout -b <new-branch-name>
    - Usage: Creates a new branch and immediately switches to it.

10. Merge

The git merge command integrates changes from one branch into another. It combines the histories of two branches, creating a merge commit if there are no conflicts.

1. Merge Another Branch into Current Branch:
    - Command: git merge <branch-name>
    - Usage: Merges changes from the specified branch into the currently checked-out branch.
2. Merge with a Specific Commit:
    - Command: git merge <commit-hash>
    - Usage: Merges the changes from the specified commit into the current branch, useful for integrating specific changes.

11. Rebase

The git rebase command is used to apply commits from one branch onto another, rewriting the commit history. It is often used to integrate changes from one branch into another or to clean up commit history.

1. Rebase Current Branch onto Another Branch:
    - Command: git rebase <branch-name>
    - Usage: Reapplies commits from the current branch on top of the specified branch, often used to incorporate changes from the main branch.
2. Interactive Rebase for History Editing:
    - Command: git rebase -i <commit-hash>
    - Usage: Opens an interactive interface to modify, reorder, or squash commits up to the specified commit hash, useful for cleaning up commit history.

12. Log

The git log command displays a chronological list of commits in the repository. It provides detailed information about each commit, including the commit hash, author, date, and commit message.

1. View Commit History:
    - Command: git log
    - Usage: Shows a detailed log of all commits in the current branch, displaying commit messages, authors, and dates.
2. View Commit History with Graph and Abbreviated Commit Hashes:

- Command: git log --graph –oneline –all
- Usage: Provides a simplified, graphical view of the commit history with abbreviated commit hashes and branch structure.

13. Diff

The git diff command shows differences between various states of the repository, such as between commits, branches, or the working directory and the index. It highlights what has changed in the code.

1. View Changes in Working Directory:
    - Command: git diff
    - Usage: Displays changes between the working directory and the staging area (index), showing modifications not yet staged for commit.
2. View Differences Between Commits:
    - Command: git diff <commit1> <commit2>
    - Usage: Shows changes between two specific commits, helping to compare the differences between different points in history.

14. Stash

The git stash command temporarily saves changes in your working directory and staging area, allowing you to work on something else without committing the current changes. It helps keep your working directory clean.

1. Save Current Changes:
    - Command: git stash
    - Usage: Stashes uncommitted changes in your working directory and index, reverting your working directory to the last commit state.
2. Apply Stashed Changes:
    - Command: git stash apply
    - Usage: Applies the most recently stashed changes back to your working directory without removing them from the stash list.

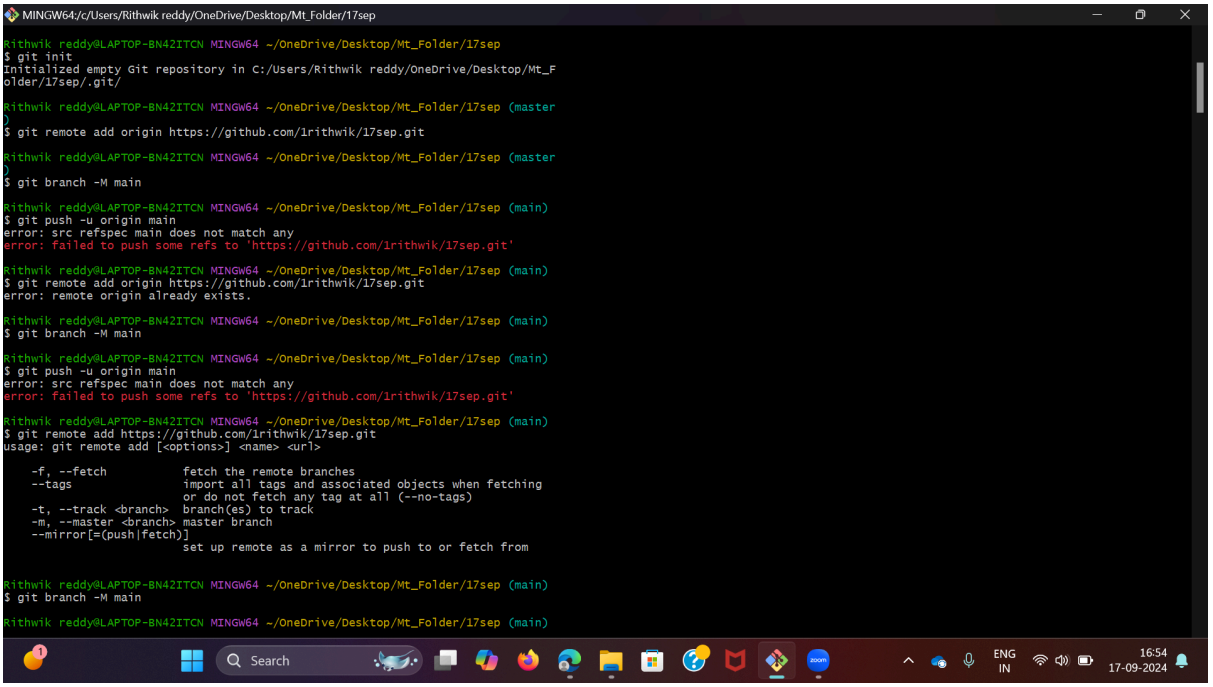
15. Reset

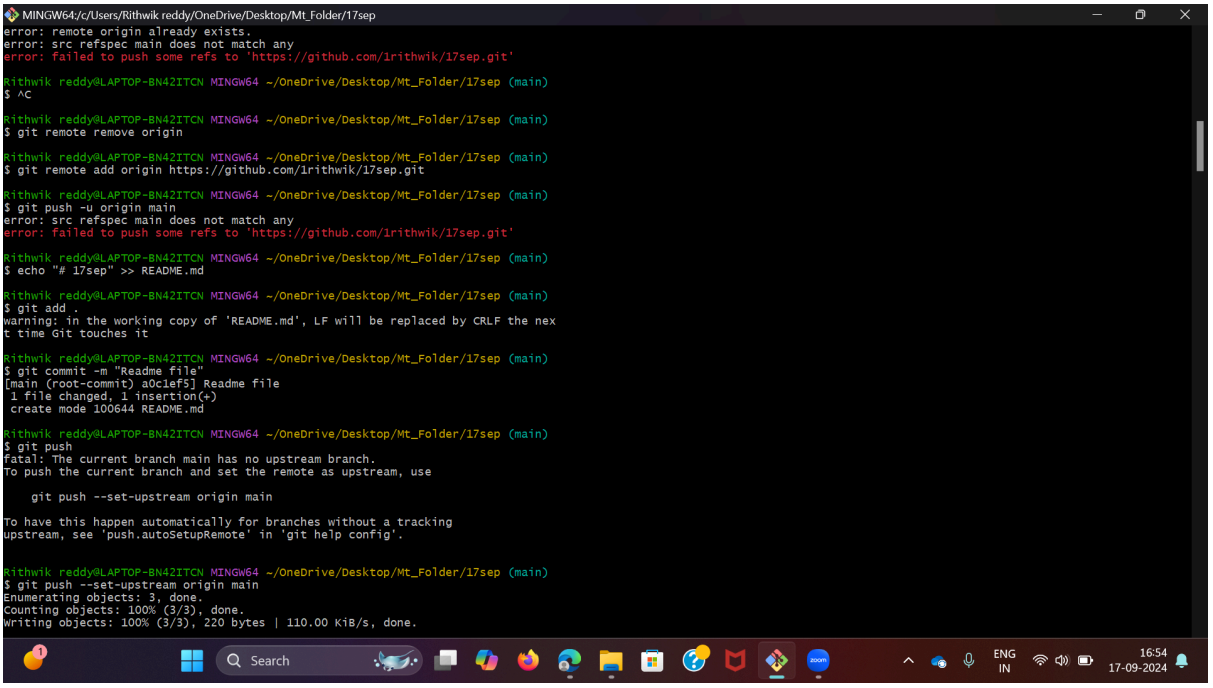
The git reset command changes the current branch’s commit history and working directory state. It can be used to unstage files, reset commits, or revert changes in the working directory.

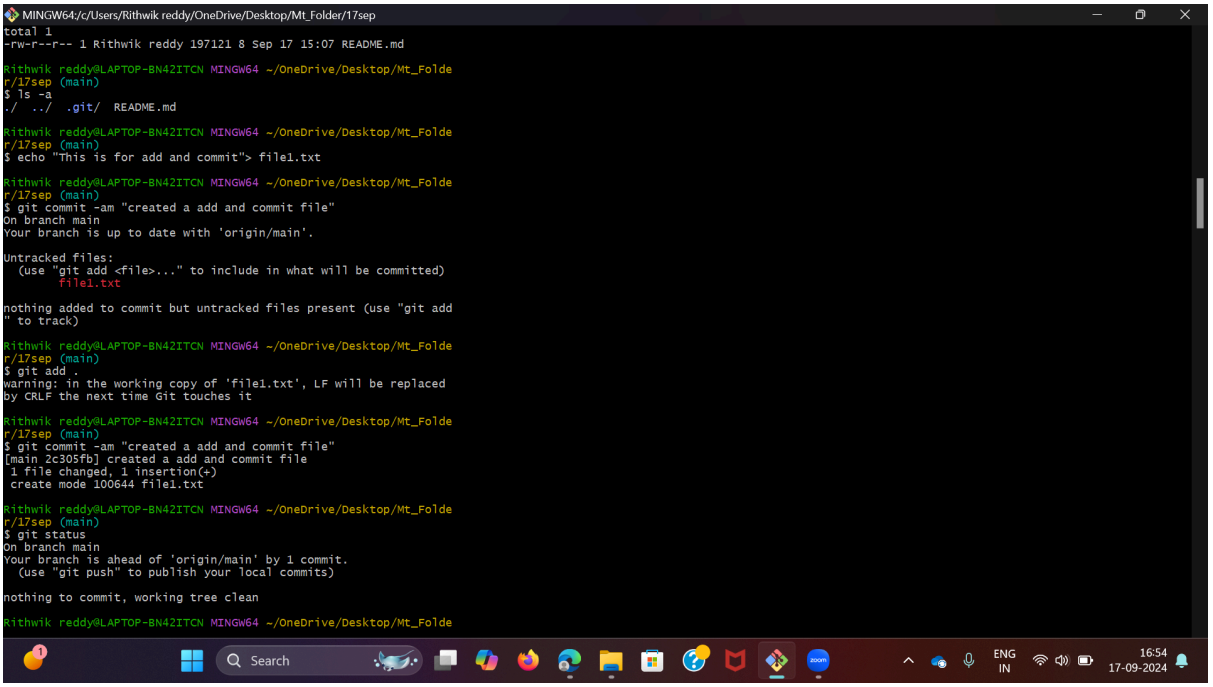
1. Unstaged Files:

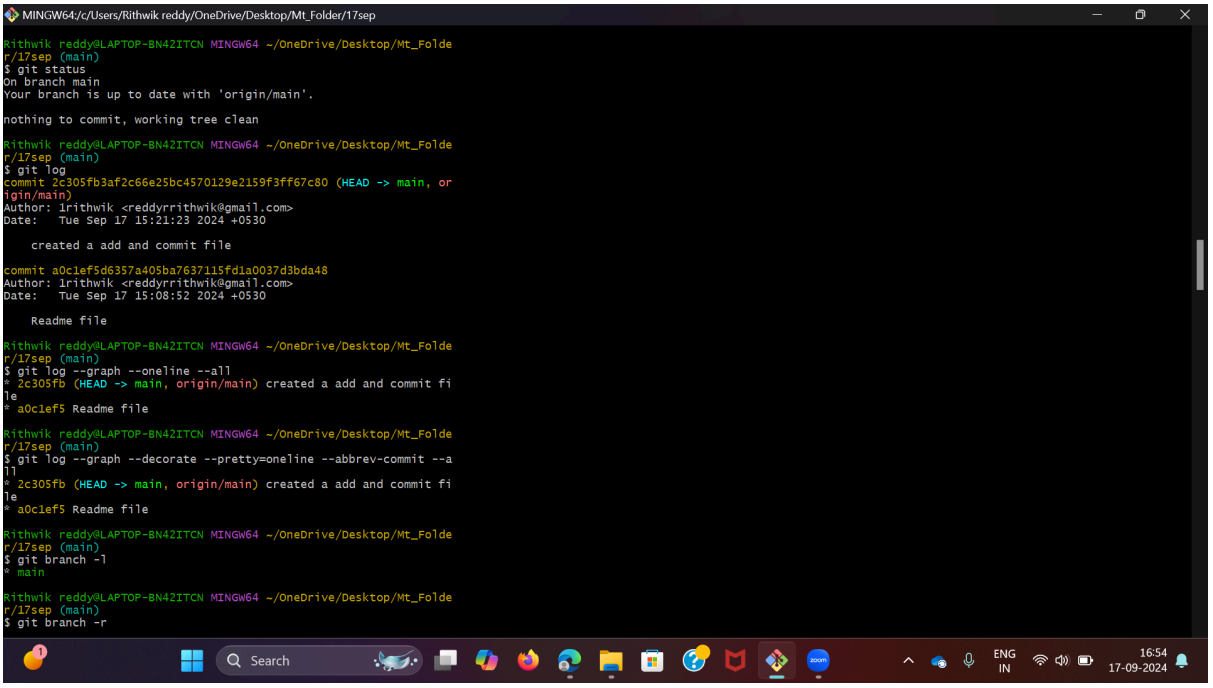
- Command: git reset <file>
- Removes the specified file(s) from the staging area, effectively undoing the git add command but keeping the changes in the working directory.

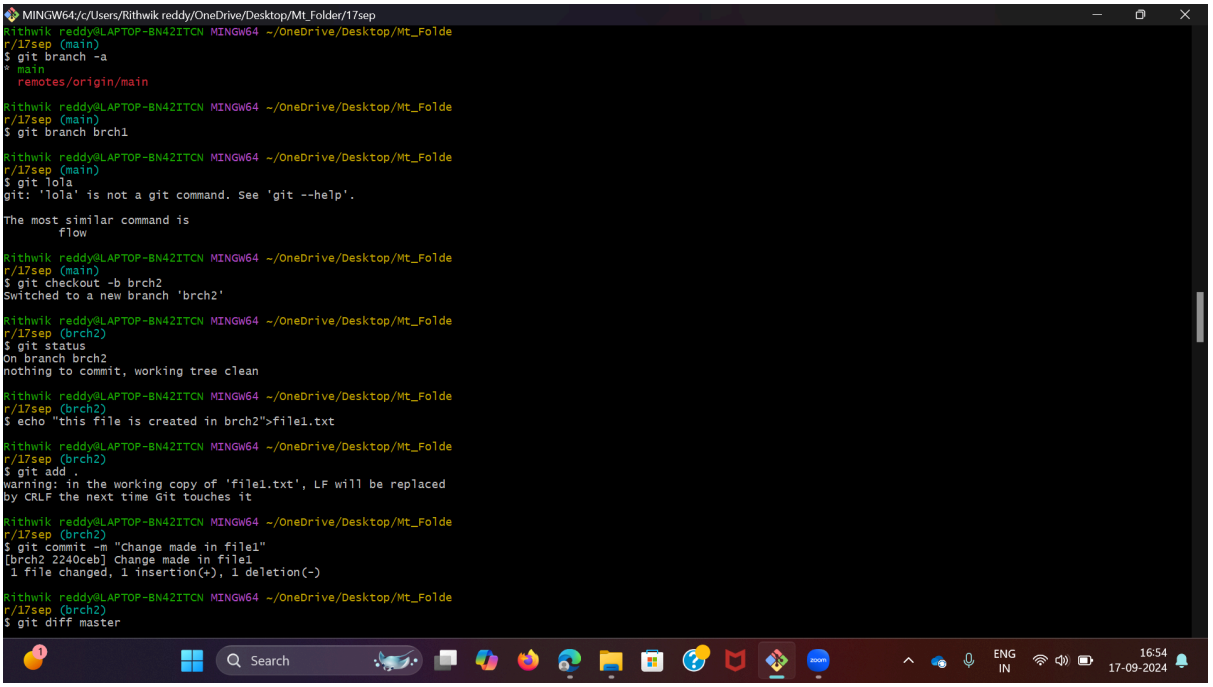
2. Reset to a Specific Commit (Soft Reset):
    - Command: git reset --soft <commit-hash>
    - Usage: Resets the branch to a specific commit but keeps all changes in the working directory and staging area, allowing you to amend or re-commit them.
    - The default is mixed, here it resets to the stage where the change is added but not committed
    - The other flag is hard. When using this tag, the entire changes done is erased.

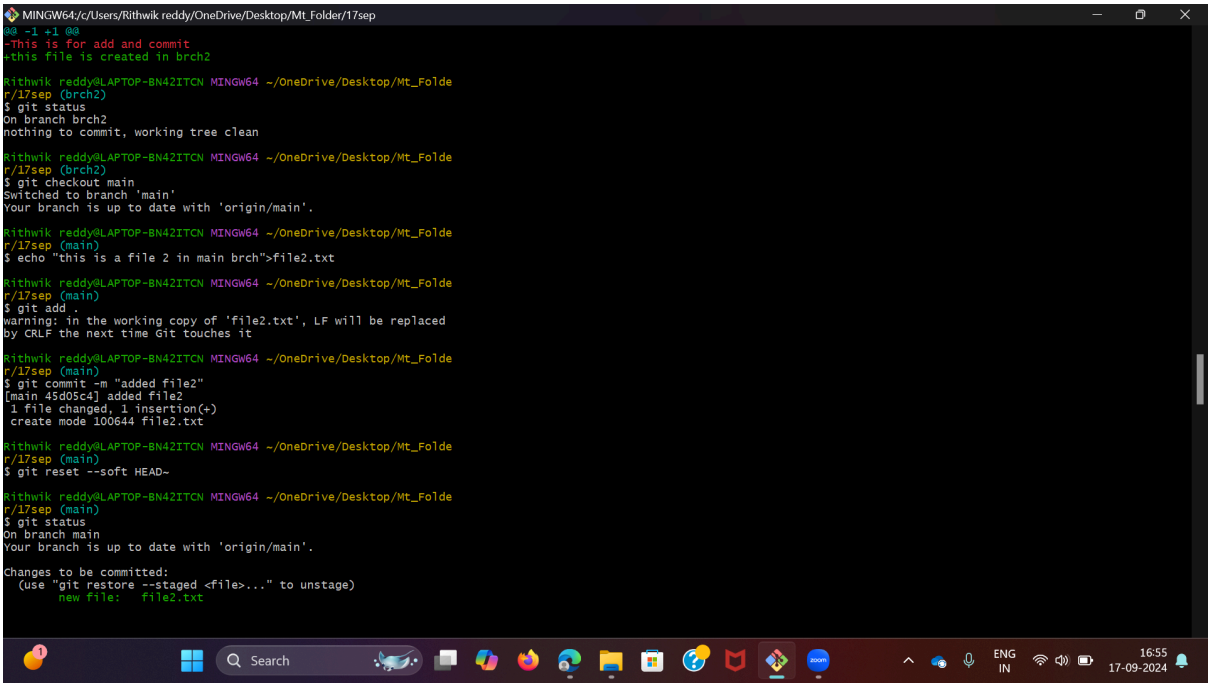

MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep
$ git init
Initialized empty Git repository in C:/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep/.git/
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (master)
$ git remote add origin https://github.com/1rithwik/17sep.git
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (master)
$ git branch -M main
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git push -u origin main
error: src refspec main does not match any
error: failed to push some refs to 'https://github.com/1rithwik/17sep.git'
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git remote add origin https://github.com/1rithwik/17sep.git
error: remote origin already exists.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git branch -M main
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git push -u origin main
error: src refspec main does not match any
error: failed to push some refs to 'https://github.com/1rithwik/17sep.git'
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git remote add https://github.com/1rithwik/17sep.git
usage: git remote add [<options>] <name> <url>
-f, --fetch fetch the remote branches
--tags import all tags and associated objects when fetching
or do not fetch any tag at all (--no-tags)
-t, --track <branch> branch(es) to track
-m, --master <branch> master branch
--mirror[=(push|fetch)]
set up remote as a mirror to push to or fetch from
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git branch -M main
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
Search
ENG
IN
16:54
17-09-2024

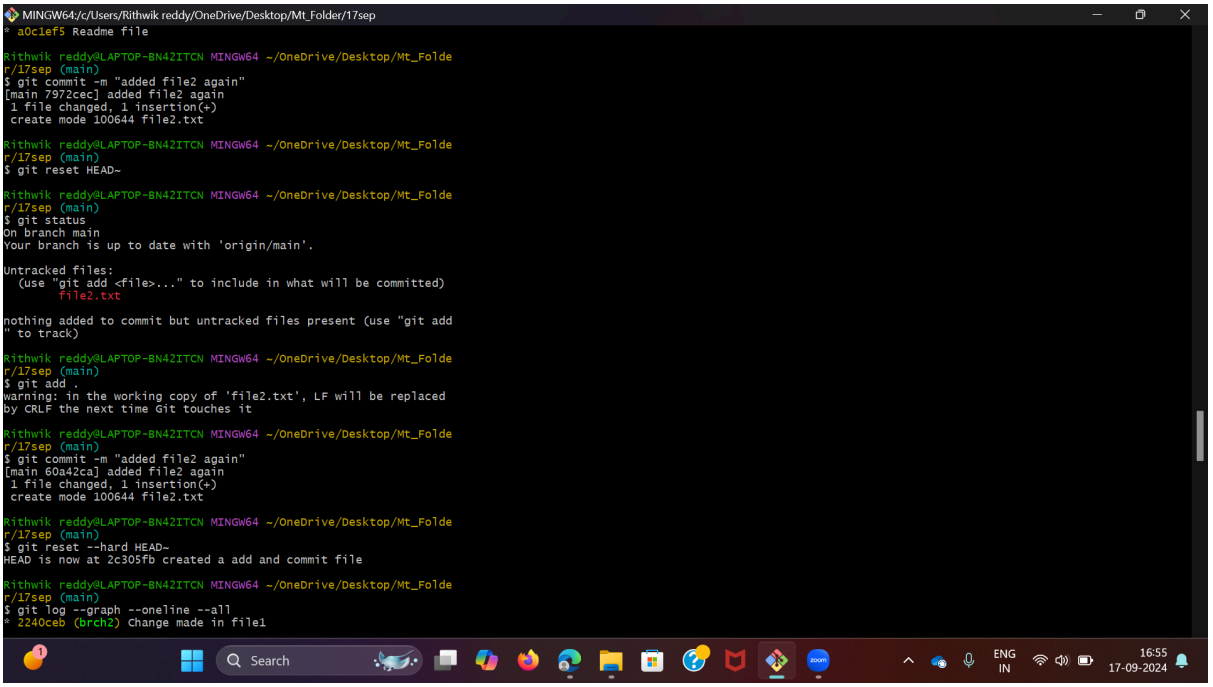

MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
error: remote origin already exists.
error: src refspec main does not match any
error: failed to push some refs to 'https://github.com/1rithwik/17sep.git'
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ ^C
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git remote remove origin
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git remote add origin https://github.com/1rithwik/17sep.git
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git push -u origin main
error: src refspec main does not match any
error: failed to push some refs to 'https://github.com/1rithwik/17sep.git'
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ echo "# 17sep" >> README.md
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git add .
warning: in the working copy of 'README.md', LF will be replaced by CRLF the next time Git touches it
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git commit -m "Readme file"
[main (root-commit) a0c1ef5] Readme file
1 file changed, 1 insertion(+)
create mode 100644 README.md
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git push
fatal: The current branch main has no upstream branch.
To push the current branch and set the remote as upstream, use
git push --set-upstream origin main
To have this happen automatically for branches without a tracking upstream, see 'push.autoSetupRemote' in 'git help config'.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folder/17sep (main)
$ git push --set-upstream origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 220 bytes | 110.00 KiB/s, done.
Search
ENG
IN
16:54
17-09-2024

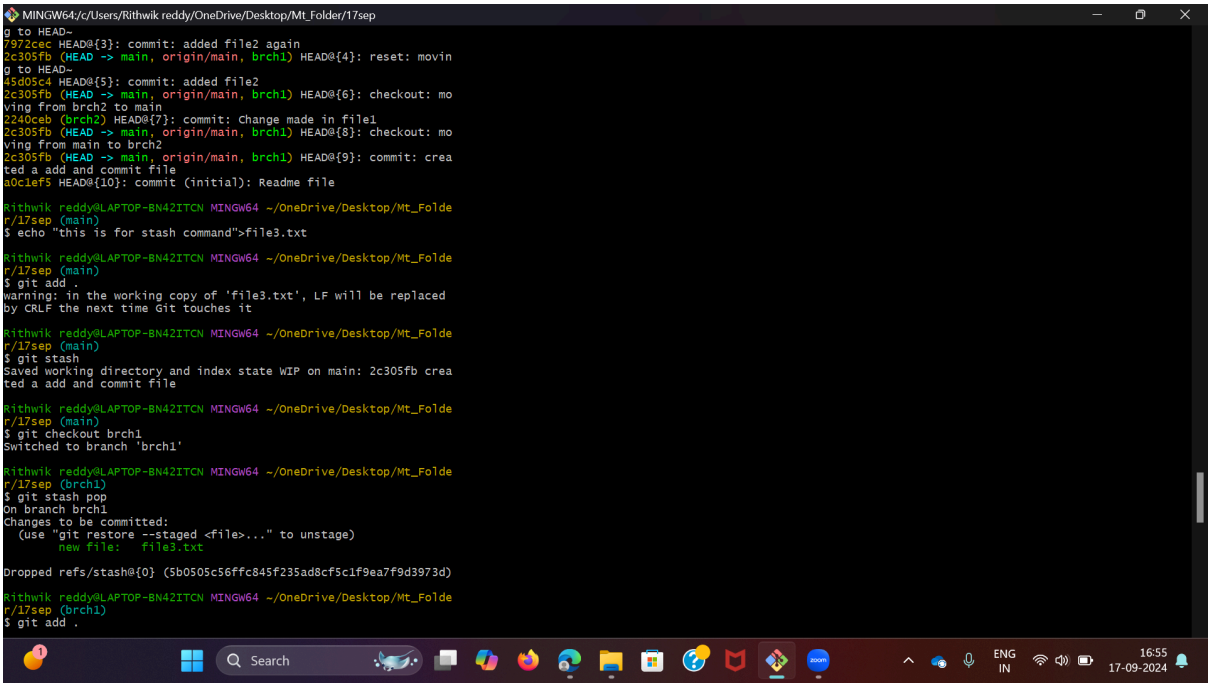

MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
total 1
-rw-r--r-- 1 Rithwik reddy 197121 8 Sep 17 15:07 README.md
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ ls -a
./ ../ .git/ README.md
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ echo "This is for add and commit"> file1.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git commit -am "created a add and commit file"
On branch main
Your branch is up to date with 'origin/main'.
Untracked files:
(use "git add <file>..." to include in what will be committed)
file1.txt
nothing added to commit but untracked files present (use "git add
" to track)
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git add .
warning: in the working copy of 'file1.txt', LF will be replaced
by CRLF the next time Git touches it
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git commit -am "created a add and commit file"
[main 2c305fb] created a add and commit file
1 file changed, 1 insertion(+)
create mode 100644 file1.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working tree clean
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
Search
ENG
IN
16:54
17-09-2024

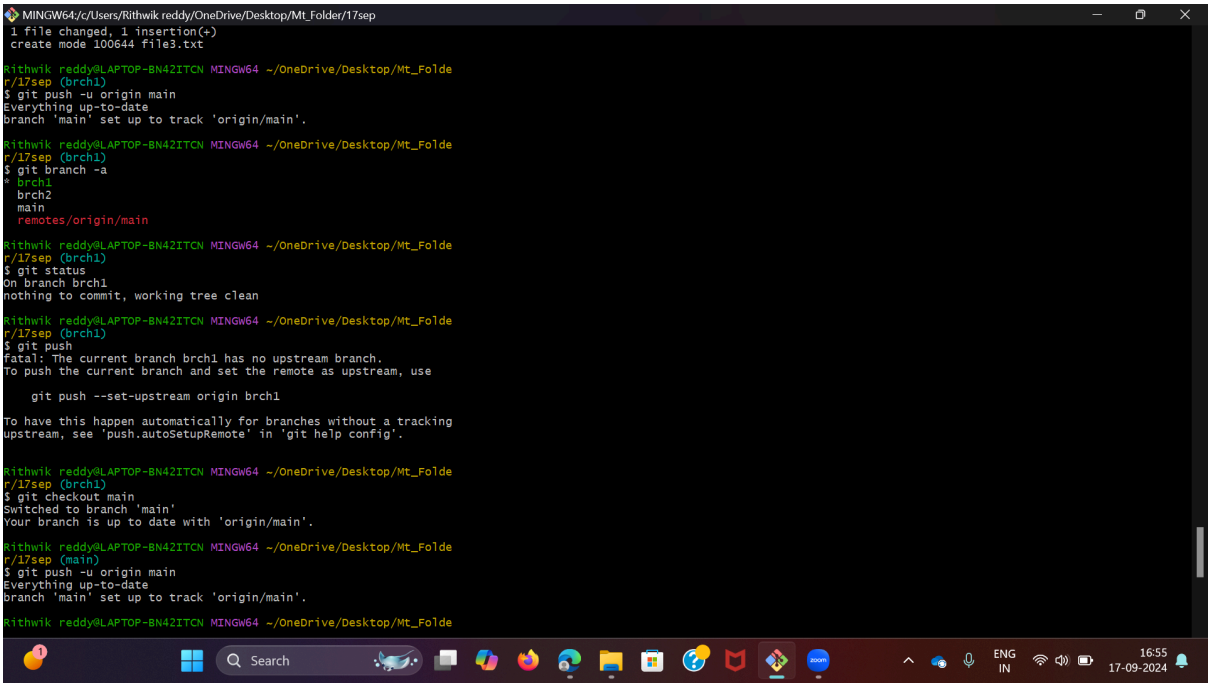

MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.
nothing to commit, working tree clean
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git log
commit 2c305fb3af2c66e25bc4570129e2159f3ff67c80 (HEAD -> main, or
igin/main)
Author: 1rithwik <reddyrrithwik@gmail.com>
Date: Tue Sep 17 15:21:23 2024 +0530
created a add and commit file
commit a0c1ef5d6357a405ba7637115fd1a0037d3bda48
Author: 1rithwik <reddyrrithwik@gmail.com>
Date: Tue Sep 17 15:08:52 2024 +0530
Readme file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git log --graph --oneline --all
* 2c305fb (HEAD -> main, origin/main) created a add and commit fi
le
* a0c1ef5 Readme file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git log --graph --decorate --pretty=oneline --abbrev-commit --a
ll
* 2c305fb (HEAD -> main, origin/main) created a add and commit fi
le
* a0c1ef5 Readme file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git branch -l
* main
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git branch -r
Search
ENG
IN
16:54
17-09-2024

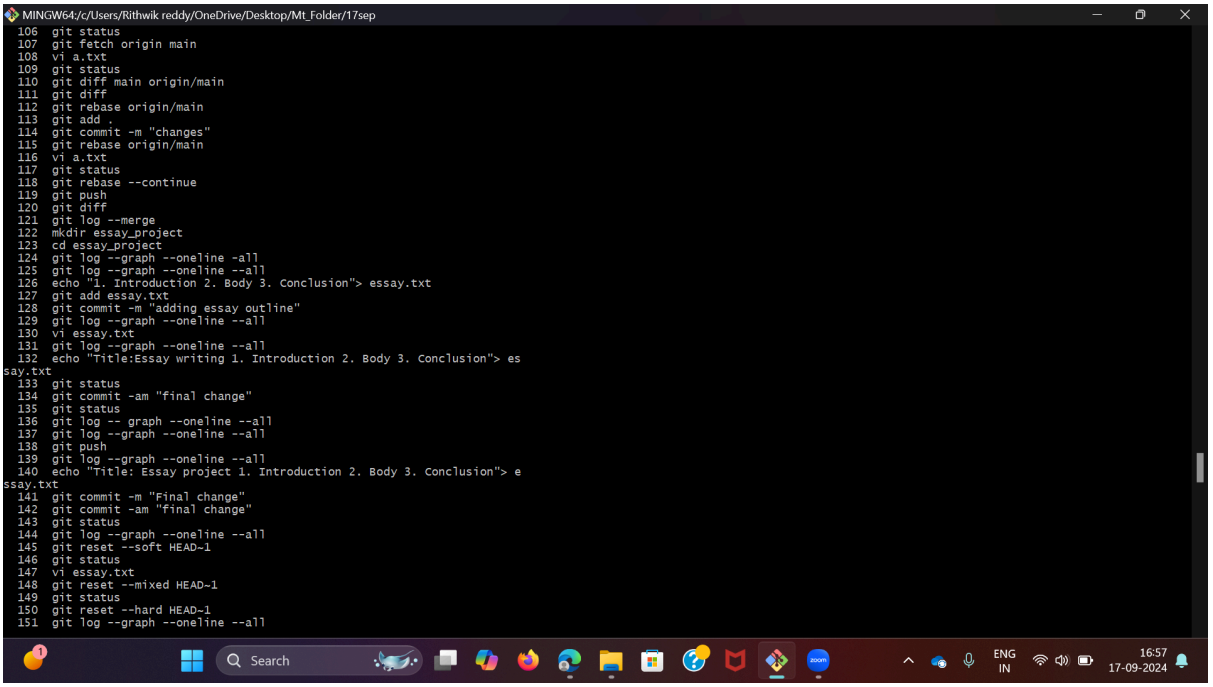

MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git branch -a
* main
remotes/origin/main
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git branch brch1
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git lola
git: 'lola' is not a git command. See 'git --help'.
The most similar command is
flow
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git checkout -b brch2
Switched to a new branch 'brch2'
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ git status
On branch brch2
nothing to commit, working tree clean
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ echo "this file is created in brch2">file1.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ git add .
warning: in the working copy of 'file1.txt', LF will be replaced
by CRLF the next time Git touches it
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ git commit -m "Change made in file1"
[brch2 2240ceb] Change made in file1
1 file changed, 1 insertion(+), 1 deletion(-)
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ git diff master
Search
ENG
IN
16:54
17-09-2024

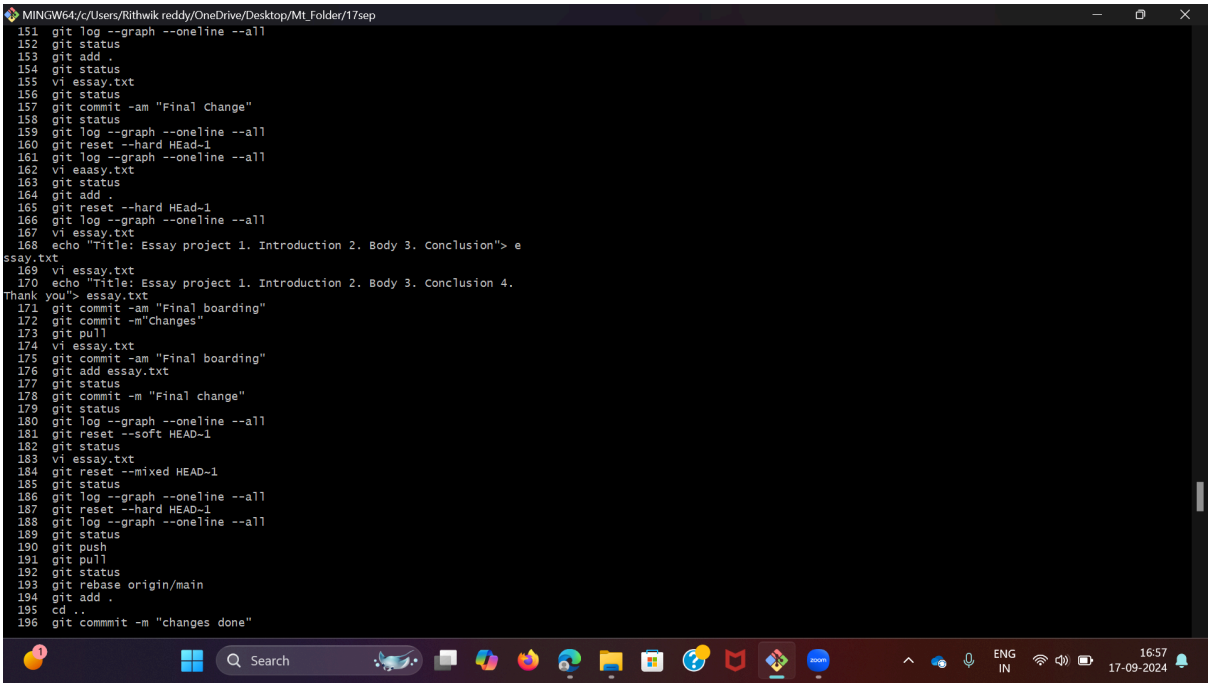

MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
@@ -1 +1 @@
-This is for add and commit
+this file is created in brch2
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ git status
On branch brch2
nothing to commit, working tree clean
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch2)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ echo "this is a file 2 in main brch">file2.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git add .
warning: in the working copy of 'file2.txt', LF will be replaced
by CRLF the next time Git touches it
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git commit -m "added file2"
[main 45d05c4] added file2
1 file changed, 1 insertion(+)
create mode 100644 file2.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git reset --soft HEAD~
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
new file: file2.txt
Search
ENG
IN
16:55
17-09-2024


MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
* a0c1ef5 Readme file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git commit -m "added file2 again"
[main 7972cec] added file2 again
1 file changed, 1 insertion(+)
create mode 100644 file2.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git reset HEAD~
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.
Untracked files:
(use "git add <file>..." to include in what will be committed)
file2.txt
nothing added to commit but untracked files present (use "git add
" to track)
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git add .
warning: in the working copy of 'file2.txt', LF will be replaced
by CRLF the next time Git touches it
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git commit -m "added file2 again"
[main 60a42ca] added file2 again
1 file changed, 1 insertion(+)
create mode 100644 file2.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git reset --hard HEAD~
HEAD is now at 2c305fb created a add and commit file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git log --graph --oneline --all
* 2240ceb (brch2) Change made in file1
Search
ENG
IN
16:55
17-09-2024


MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
g to HEAD~
7972cec HEAD@{3}: commit: added file2 again
2c305fb (HEAD -> main, origin/main, brch1) HEAD@{4}: reset: movin
g to HEAD~
45d05c4 HEAD@{5}: commit: added file2
2c305fb (HEAD -> main, origin/main, brch1) HEAD@{6}: checkout: mo
ving from brch2 to main
2240ceb (brch2) HEAD@{7}: commit: Change made in file1
2c305fb (HEAD -> main, origin/main, brch1) HEAD@{8}: checkout: mo
ving from main to brch2
2c305fb (HEAD -> main, origin/main, brch1) HEAD@{9}: commit: crea
ted a add and commit file
a0c1ef5 HEAD@{10}: commit (initial): Readme file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ echo "this is for stash command">file3.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git add .
warning: in the working copy of 'file3.txt', LF will be replaced
by CRLF the next time Git touches it
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git stash
Saved working directory and index state WIP on main: 2c305fb crea
ted a add and commit file
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git checkout brch1
Switched to branch 'brch1'
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git stash pop
On branch brch1
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
new file: file3.txt
Dropped refs/stash@{0} (5b0505c56ffc845f235ad8cf5c1f9ea7f9d3973d)
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git add .
Search
ENG
IN
16:55
17-09-2024


MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
1 file changed, 1 insertion(+)
create mode 100644 file3.txt
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git push -u origin main
Everything up-to-date
branch 'main' set up to track 'origin/main'.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git branch -a
* brch1
brch2
main
remotes/origin/main
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git status
On branch brch1
nothing to commit, working tree clean
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git push
fatal: The current branch brch1 has no upstream branch.
To push the current branch and set the remote as upstream, use
git push --set-upstream origin brch1
To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (brch1)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
r/17sep (main)
$ git push -u origin main
Everything up-to-date
branch 'main' set up to track 'origin/main'.
Rithwik reddy@LAPTOP-BN42ITCN MINGW64 ~/OneDrive/Desktop/Mt_Folde
Search
ENG
IN
16:55
17-09-2024


MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
106 git status
107 git fetch origin main
108 vi a.txt
109 git status
110 git diff main origin/main
111 git diff
112 git rebase origin/main
113 git add .
114 git commit -m "changes"
115 git rebase origin/main
116 vi a.txt
117 git status
118 git rebase --continue
119 git push
120 git diff
121 git log --merge
122 mkdir essay_project
123 cd essay_project
124 git log --graph --oneline -all
125 git log --graph --oneline --all
126 echo "1. Introduction 2. Body 3. Conclusion"> essay.txt
127 git add essay.txt
128 git commit -m "adding essay outline"
129 git log --graph --oneline --all
130 vi essay.txt
131 git log --graph --oneline --all
132 echo "Title:Essay writing 1. Introduction 2. Body 3. Conclusion"> es
say.txt
133 git status
134 git commit -am "final change"
135 git status
136 git log -- graph --oneline --all
137 git log --graph --oneline --all
138 git push
139 git log --graph --oneline --all
140 echo "Title: Essay project 1. Introduction 2. Body 3. Conclusion"> e
ssay.txt
141 git commit -m "Final change"
142 git commit -am "final change"
143 git status
144 git log --graph --oneline --all
145 git reset --soft HEAD~1
146 git status
147 vi essay.txt
148 git reset --mixed HEAD~1
149 git status
150 git reset --hard HEAD~1
151 git log --graph --oneline --all
Search
ENG
IN
16:57
17-09-2024


MINGW64:/c/Users/Rithwik reddy/OneDrive/Desktop/Mt_Folder/17sep
151 git log --graph --oneline --all
152 git status
153 git add .
154 git status
155 vi essay.txt
156 git status
157 git commit -am "Final Change"
158 git status
159 git log --graph --oneline --all
160 git reset --hard HEad~1
161 git log --graph --oneline --all
162 vi eaasy.txt
163 git status
164 git add .
165 git reset --hard HEad~1
166 git log --graph --oneline --all
167 vi essay.txt
168 echo "Title: Essay project 1. Introduction 2. Body 3. Conclusion"> e
ssay.txt
169 vi essay.txt
170 echo "Title: Essay project 1. Introduction 2. Body 3. Conclusion 4.
Thank you"> essay.txt
171 git commit -am "Final boarding"
172 git commit -m"Changes"
173 git pull
174 vi essay.txt
175 git commit -am "Final boarding"
176 git add essay.txt
177 git status
178 git commit -m "Final change"
179 git status
180 git log --graph --oneline --all
181 git reset --soft HEAD~1
182 git status
183 vi essay.txt
184 git reset --mixed HEAD~1
185 git status
186 git log --graph --oneline --all
187 git reset --hard HEAD~1
188 git log --graph --oneline --all
189 git status
190 git push
191 git pull
192 git status
193 git rebase origin/main
194 git add .
195 cd ..
196 git commmit -m "changes done"
Search
ENG
IN
16:57
17-09-2024

The concepts understood on (18-09-2024)

Java programming language:

Java is a high-level, object-oriented programming language designed to be platform-independent.

Steps of Java Execution:

1. Writing the Source Code:
    - Java programs are written in plain text files with a .java extension.
2. Compilation:
    - The Java compiler (javac) converts the source code into bytecode, which is an intermediate representation. This bytecode is stored in .class files. The command used is:

      javac MyProgram.java
3. Loading:
    - The Java Class Loader loads the compiled bytecode into the JVM. It locates the necessary classes and prepares them for execution.
4. Bytecode Verification:
    - Before execution, the bytecode is verified by the JVM to ensure it adheres to Java's security and integrity rules.
5. Execution:
    - The JVM interprets or compiles the bytecode into machine code specific to the operating system.
6. Running the Program:
    - The JVM executes the machine code, resulting in the desired output of the program.

JDK, JRE, JVM -

Java's architecture includes several components that work together to enable Java programming. The three main components are the **Java Development Kit (JDK)**, the **Java Runtime Environment (JRE)**, and the **Java Virtual Machine (JVM)**. Here's a breakdown of each:

1. Java Development Kit (JDK)

- The JDK is a comprehensive toolkit for Java developers. It includes everything needed to develop Java applications.
- **Components:**
    - Compiler (javac): Converts Java source code into bytecode.
    - **Debugger:** Helps find and fix bugs in the code.

    - **Libraries:** Pre-written classes and functions that developers can use to build applications.
    - **Development Tools:** Command-line tools for compiling, documenting, and packaging Java applications.
- **Use Case:** Required for writing and compiling Java programs.

2. Java Runtime Environment (JRE)

- The JRE provides the libraries, Java Virtual Machine, and other components necessary to run Java applications. It does not include development tools.
- **Components:**
    - **Java Class Libraries:** Standard libraries for Java that provide built-in functions for common tasks.
    - **JVM:** The engine that executes the bytecode.
- **Use Case:** Required for running Java applications but not for developing them.

3. Java Virtual Machine (JVM)

- The JVM is an abstract machine that enables a computer to run Java programs. It provides a runtime environment for Java bytecode.
- **Functions:**
    - **Bytecode Execution:** Converts bytecode into machine code that the host operating system can execute.
    - **Memory Management:** Handles memory allocation and garbage collection.
    - **Platform Independence:** Allows Java programs with a compatible JVM to run on any operating system.
- **Use Case:** Essential for executing Java applications, regardless of the underlying hardware or operating system.

We have multiple platforms on which to write the code. I will be using the curser.ai platform.

Java Data Types -

In Java, data types are categorized into two main groups: Primitive Data Types and Reference Data Types. Understanding these types is essential for effective programming in Java.

1. Primitive Data Types

Primitive data types are the most basic data types built into the Java language. They represent single values and are not objects.

- byte
    - Size: 8 bits
    - Range: -128 to 127
    - Usage: Useful for saving memory in large arrays, mainly in place of integers.
- short

    - Size: 16 bits
    - Range: -32,768 to 32,767
    - Usage: Also useful for saving memory, especially in large arrays.
- int
    - Size: 32 bits
    - Range: -2,147,483,648 to 2,147,483,647
    - Usage: Default data type for integer values.
- long
    - Size: 64 bits
    - Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
    - Usage: Used when a wider range than int is needed.
- float
    - Size: 32 bits
    - Range: Approximately ±3.40282347E+38 (6-7 significant decimal digits)
    - Usage: Used for single-precision floating-point numbers.
- double
    - Size: 64 bits
    - Range: Approximately ±1.79769313486231570E+308 (15 significant decimal digits)
    - Usage: Used for double-precision floating-point numbers.
- char
    - Size: 16 bits
    - Range: 0 to 65,535 (represents a single 16-bit Unicode character)
- boolean
    - Size: 1 bit (not precisely defined)
    - Values: true or false
    - Usage: Used for simple flags that track true/false conditions.

2. Reference Data Types(Non primitive data types):

Reference data types are used to refer to objects. They do not hold the value directly; instead, they hold a reference (or memory address) to the object.

- String
    - Represents a sequence of characters.

    - Example: String name = "Hello, World!";
- Arrays
    - A collection of similar types of data stored in a single variable.
    - Example: int[] numbers = {1, 2, 3, 4, 5};
- Objects
    - Instances of classes that can hold multiple fields and methods.
    - Example: MyClass obj = new MyClass();

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

        Hello_world obj1 = new Hello_world();

        obj1.display();

        int a = 10;

        int b = 20;

        int c = a + b;

        double d = 10.5;

        float f = 20.5f;

        char g = 'g';

        byte h = 100;

        short i = 200;

        long j = 10000000000000000l;

        boolean k = true;

        String l = "Hello World";

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

        // System.out.println("Hello World");

    }
```

Classes, Objects, Constructors, and Methods in Java

1. Class

- Definition: A class is a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from the class can have.

Syntax:

```
public class ClassName {
    dataType attributeName;

    // Constructor
    public ClassName() {
    }

    // Method
    public returnType methodName(parameters) {
    }
}
```

-

2. Object

- Definition: An object is an instance of a class. It represents a specific entity with attributes and behaviors defined by its class.
- Characteristics:
    - Objects can hold state through attributes.
    - They can perform actions via methods.

Example:
ClassName obj = new ClassName();

3. Constructor

- Definition: A constructor is a special method used to initialize objects. It has the same name as the class and does not have a return type.
- Types:
    - Default Constructor: No parameters. Initializes attributes with default values.
    - Parameterized Constructor: Takes arguments to initialize attributes with specific values.

Syntax:

```
public class ClassName {
    int x;
    public ClassName() {
```

```
        x = 0;
    }
    // Parameterized Constructor
    public ClassName(int value) {
        x = value; // Custom value
    }
}
```

4. Method

- A method is a block of code that performs a specific task. It can take parameters and may return a value.
- Characteristics:
    - Methods can be called on objects to perform actions.
    - They can modify object state or perform computations.

Example:

```
public int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}
```

Example for the topic:

```
public class Employee {

    String name;

    String city;

    String country;

    int salary;

    int age;

    String department;

    String designation;

    String company_name;

    static boolean is_employee_of_the_month = true;
```

```
    public void printDetails() {
        System.out.println("Name: " + name);
        System.out.println("City: " + city);
        System.out.println("Country: " + country);
        System.out.println("Salary: " + salary);
        System.out.println("Age: " + age);
        System.out.println("Department: " + department);
        System.out.println("Designation: " + designation);
        System.out.println("Company Name: " + company_name);
        System.out.println("Is Employee of the Month: " +
is_employee_of_the_month);
    }

    public void setDetails(String name, String city, String country,
int salary, int age, String department,
            String designation, String company_name) {
        this.name = name;
        this.city = city;
        this.country = country;
        this.salary = salary;
        this.age = age;
        this.department = department;
        this.designation = designation;
        this.company_name = company_name;
    }

    public void setIsEmployeeOfTheMonth(boolean
is_employee_of_the_month) {
        Employee.is_employee_of_the_month = is_employee_of_the_month;
```

```
    }


    public Employee(String name, String city, String country, int
salary, int age, String department,
            String designation, String company_name) {
        this.name = "John Doe";
        this.city = "New York";
        this.country = "USA";
        this.salary = 10000;
        this.age = 25;
        this.department = "IT";
        this.designation = "Software Engineer";
        this.company_name = "ABC Company";
    }


    public static void main(String[] args) {
        Employee emp1 = new Employee("John Doe", "New York", "USA",
10000, 25, "IT", "Software Engineer",
                "ABC Company");
        emp1.printDetails();
        emp1.setDetails("Diya", "London", "UK", 1000, 27, "Finance",
"Manager", "XYZ Company");
        emp1.setIsEmployeeOfTheMonth(false);
        emp1.printDetails();
        System.out.println(is_employee_of_the_month);
    }
}
```

Operators in Java -

Operators are special symbols in Java that perform operations on variables and values. They can be classified into several categories:

1. Arithmetic Operators

These operators perform basic mathematical operations.

- Addition (+): Adds two operands.
- Subtraction (-): Subtracts the second operand from the first.
- Multiplication (*): Multiplies two operands.
- Division (/): Divides the first operand by the second.
- Modulus (%): Returns the remainder of division.

Example:

```
int a = 10;

int b = 3;

int sum = a + b;        // 13

int difference = a - b; // 7

int product = a * b;    // 30

int quotient = a / b;   // 3

int remainder = a % b;  // 1
```

2. Relational Operators

These operators are used to compare two values.

- Equal to (==): Checks if two operands are equal.
- Not equal to (!=): Checks if two operands are not equal.
- Greater than (>): Checks if the left operand is greater than the right.
- Less than (<): Checks if the left operand is less than the right.
- Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right.
- Less than or equal to (<=): Checks if the left operand is less than or equal to the right.

Example:

```
boolean isEqual = (a == b); // false

boolean isGreater = (a > b); // true
```

3. Logical Operators

These operators are used to combine multiple boolean expressions.

- Logical AND (&&): Returns true if both operands are true.
- Logical OR (||): Returns true if at least one operand is true.
- Logical NOT (!): Reverses the logical state of its operand.

Example:

```
boolean x = true;

boolean y = false;

boolean resultAnd = x && y; // false

boolean resultOr = x || y;  // true

boolean resultNot = !x;     // false
```

4. Assignment Operators

These operators are used to assign values to variables.

- Simple assignment (=): Assigns the right operand's value to the left operand.
- Add and assign (+=): Adds the right operand to the left operand and assigns the result to the left operand.
- Subtract and assign (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.
- Multiply and assign (*=): Multiplies the left operand by the right operand and assigns the result.
- Divide and assign (/=): Divides the left operand by the right operand and assigns the result.
- Modulus and assign (%=): Takes the modulus using two operands and assigns the result.

Example:

```
int c = 5;

c += 3; // c = c + 3, now c is 8
```

5. Unary Operators

These operators operate on a single operand.

- Unary plus (+): Indicates a positive value (usually optional).
- Unary minus (-): Negates the value.
- Increment (++): Increases the value by 1.

- Decrement (--): Decreases the value by 1.

Example:

int d = 5;

d++; // d is now 6

d--; // d is now 5

6. Ternary Operator (Conditional operator)

The ternary operator is a shorthand for the if-else statement.

- Syntax: condition ? valueIfTrue : valueIfFalse;

Example:

int max = (a > b) ? a : b; // max will be 10

Example for topic:

```
public class Operations {


    private static void demonstrateBasicLogicalOperations() {
        boolean t = true;
        boolean f = false;


        System.out.println("1. Basic Logical Operations:");
        System.out.println("   AND: true && true = " + (t && t));
        System.out.println("   AND: true && false = " + (t && f));
        System.out.println("   AND: false && true = " + (f && t));
        System.out.println("   AND: false && false = " + (f && f));
        System.out.println("   OR: true || true = " + (t || t));
        System.out.println("   OR: true || false = " + (t || f));
        System.out.println("   OR: false || true = " + (f || t));
        System.out.println("   OR: false || false = " + (f || f));
        System.out.println("   NOT: !true = " + (!t));
        System.out.println("   NOT: !false = " + (!f));
    }
```

```
    private static void demonstrateShortCircuitEvaluation() {
        boolean f = false;
        boolean t = true;
        System.out.println("\n2. Short-circuit Evaluation:");
        System.out.println("   false && (1/0 > 0) = " + (f && (1 / 0 > 0))); // as the first condition is false then the

// second condition will not be evaluated
        System.out.println("   true || (1/0 > 0) = " + (t || (1 / 0 > 0))); // as the first condition is true then the

// second condition will not be evaluated
    }

    private static void demonstrateOperatorPrecedence() {
        boolean t = true;
        boolean f = false;

        System.out.println("\n3. Operator Precedence:");
        System.out.println("   true || false && false = " + (true || false && false)); // && has higher precedence
        System.out.println("   (true || false) && false = " + ((t || f) && f)); // Parentheses change precedence
    }

    private static void demonstrateCombiningWithComparisonOperators() {
        int x = 5, y = 10;
        System.out.println("\n4. Combining with Comparison Operators:");
        System.out.println("   (x < y) && (y > 0) = " + ((x < y) && (y > 0)));
```

```
        System.out.println("   (x > y) || (y < 20) = " + ((x > y) || (y
< 20)));
    }

    private static void demonstrateComplexConditions() {
        boolean a = true, b = false, c = true;
        System.out.println("\n5. Complex Conditions:");
        System.out.println("   (a && b) || (a && c) = " + ((a && b) ||
(a && c)));
        System.out.println("   a && (b || c) = " + (a && (b || c)));
        System.out.println("   !a || (b && !c) = " + (!a || (b &&
!c)));
    }

    private static void demonstrateBitwiseVsLogicalOperators() {
        boolean t = true;
        boolean f = false;

        System.out.println("\n6. Bitwise vs. Logical Operators:");
        System.out.println("   true & false = " + (t & f)); // Bitwise
AND
        System.out.println("   true | false = " + (t | f)); // Bitwise
OR
        System.out.println("   true ^ false = " + (t ^ f)); // Bitwise
XOR
    }

    private static void demonstrateShortCircuitVsNonShortCircuit() {
        boolean f = false;
        System.out.println("\n7. Short-circuit vs.
Non-short-circuit:");
```

```
        int i = 0;
        boolean result1 = (f && (++i > 0)); // i is not incremented
        boolean result2 = (f & (++i > 0)); // i is incremented
        System.out.println("   Short-circuit AND result: " + result1 +
", i = " + i);
        System.out.println("   Non-short-circuit AND result: " +
result2 + ", i = " + i);
    }

    private static void
demonstrateLogicalOperatorsWithNonBooleanOperands() {
        System.out.println("\n8. Logical Operators with Non-boolean
Operands:");
        System.out.println("   (1 < 2) && (3 < 4) = " + ((1 < 2) && (3
< 4)));
        System.out.println("   ('a' < 'b') || ('c' > 'd') = " + (('a' <
'b') || ('c' > 'd')));
    }

    private static void
demonstrateLogicalOperatorsInControlStructures() {
        boolean t = true;
        boolean f = false;

        System.out.println("\n9. Logical Operators in Control
Structures:");
        if (t && !f) {
            System.out.println("   This will be printed.");
        }

        int j = 0;
        while (j < 3 && t) {
```

```
            System.out.println("    j = " + j);
            j++;
        }
    }

    private static void demonstrateLogicalOperatorsWithNullChecks() {
        System.out.println("\n11. Logical Operators with Null
Checks:");
        String str = null;
        System.out.println("    (str != null) && (str.length() > 0) = "
+ ((str != null) && (str.length() > 0))); // Safe
    }

    private static void conditionalOperator() {
        System.out.println("\n12. Conditional Operator:");
        int a = 5, b = 10;
        int max = (a > b) ? a : b;
        System.out.println("    The maximum of " + a + " and " + b + "
is " + max);
    }

    private static void instanceofOperator() {
        System.out.println("\n13. Instanceof Operator:");
        String str = "Hello";
        System.out.println("    str instanceof String = " + (str
instanceof String));
    }

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

```
        demonstrateShortCircuitEvaluation();

        demonstrateOperatorPrecedence();

        demonstrateCombiningWithComparisonOperators();

        demonstrateComplexConditions();

        demonstrateBitwiseVsLogicalOperators();

        demonstrateShortCircuitVsNonShortCircuit();

        demonstrateLogicalOperatorsWithNonBooleanOperands();

        demonstrateLogicalOperatorsInControlStructures();

        demonstrateLogicalOperatorsWithNullChecks();

        conditionalOperator();

        instanceofOperator();

        Operations obj1 = new Operations();

        Operations obj2 = null;

        System.out.println("obj1 instanceof Operator: " + (obj1
instanceof Operations));

        System.out.println("obj2 instanceof Operator: " + (obj2
instanceof Operations));

    }

}
```

Scanner Class in Java -

The Scanner class in Java is part of the java.util package and is used to read input from various sources, including keyboard input, files, and strings. It's commonly used for obtaining user input in console applications.

Importing the Scanner Class

To use the Scanner class, you need to import it at the beginning of your Java program:

import java.util.Scanner;

Creating a Scanner Object

You can create a Scanner object to read input. The most common way is to read from the standard input (keyboard):

Scanner sc = new Scanner(System.in);

sc.nextInt();

Reading Different Types of Input

The Scanner class provides various methods to read different types of data:

- nextInt(): Reads an integer.
- nextDouble(): Reads a double.
- nextLine(): Reads a line of text (including spaces).

- next(): Reads the next token (word).
- nextBoolean(): Reads a boolean value.

Example for the topic:

```
public class Input_Scanner {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int a = sc.nextInt();

        if (a % 2 == 0) {

            System.out.println(" Number is Even");

        } else {

            System.out.println(" Number is Odd");

        }

    }

}
```

Switch Statement in Java -

The switch statement in Java is a control flow statement that allows you to execute one block of code among many choices based on the value of a variable. It is often used as an alternative to multiple if-else statements for cleaner and more readable code when dealing with discrete values.

Syntax of Switch Statement

switch (expression) {

  case value1:

```
    // Code to be executed if expression equals value1
    break; // Exit the switch
  case value2:
    // Code to be executed if expression equals value2
    break; // Exit the switch
  // ... more cases ...
  default:
    // Code to be executed if none of the cases match
}
```

Example for the topic:

```
System.out.println("Choose what you want to do:");
            System.out.println("1. Deposit money");
            System.out.println("2. Withdraw money");
            System.out.println("3. Check balance");
            System.out.println("4. Calculate interest");
            System.out.println("5. Account details");
            System.out.println("6. Close account");
            int choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter the amount to deposit:
");
                    int a = sc.nextInt();
                    per1.depositMoney(a);
                    break;
                case 2:
                    System.out.println("Enter the amount to withdraw:
");
                    int b = sc.nextInt();
                    per1.withdrawMoney(b);
```

```
                    break;
                case 3:
                    per1.checkBalance();
                    break;
                case 4:
                    System.out.println("Enter the time in years: ");
                    int c = sc.nextInt();
                    per1.calculateInterest(rateofinterest, c);
                    break;
                case 5:
                    per1.AccountDetails();
                    break;
                case 6:
                    per1.closeAccount();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice");
            }
```

The Static keyword in Java -

The static keyword in Java is used to indicate that a particular member (field or method) belongs to the class rather than to instances of the class. This means that static members are shared among all instances of the class.

Key Characteristics of static

1. Static Variables (Class Variables)
    - Declared with the static keyword.
    - Only one copy exists, shared by all instances of the class.
    - Useful for defining constants or for maintaining shared data.

Example:
public class Counter {

static int count = 0; // Static variable

```
    public Counter() {
        count++; // Increment count whenever a new instance is created
    }
}
```

2. Static Methods
    - Can be called without creating an instance of the class.
    - Can only access static variables and call other static methods directly.
    - Cannot access instance variables or instance methods unless they are referenced by an object.

Example:

```
public class MathUtils {
    static int add(int a, int b) {
        return a + b; // Static method
    }
}
// Usage
int sum = MathUtils.add(5, 10); // No need to create an instance
```

Practiced Using the learned concepts - making a banking application

```java
package main.sep_18;


import java.util.Scanner;


public class BankApplication {
    String name;
    String BankName;
    String Branch;
    int depositmoney;
```

```
    int withdrawmoney;
    int balance = 0;
    static int rateofinterest = 5;
    int time_in_years;
    int totalamount = 0;
    int totalinterest = 0;


    public void accountOpen() {
        System.out.println("Account opened successfully");
    }


    public void depositMoney(int depositmoney) {
        balance = balance + depositmoney;
        System.out.println("Deposited money: " + balance);
    }


    public void withdrawMoney(int withdrawmoney) {
        if (balance < withdrawmoney) {
            System.out.println("Insufficient balance");
        } else {
            balance = balance - withdrawmoney;
            System.out.println("money withdrawn successfully");
        }
    }


    public void checkBalance() {
        System.out.println("Balance: " + balance);
    }
```

```
    public void calculateInterest(int rateofinterest, int
time_in_years) {
        totalinterest = (balance * rateofinterest * time_in_years) /
100;
        System.out.println("Total interest: " + totalinterest);
    }

    public void AccountDetails() {
        System.out.println("Account details: " + name + " " + BankName
+ " " + Branch);
        System.out.println("Balance: " + balance);
    }

    public void closeAccount() {
        System.out.println("closed successfully");
    }

    public static void main(String[] args) {
        BankApplication per1 = new BankApplication();
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the name: ");
        per1.name = sc.nextLine();
        System.out.println("Enter the BankName: ");
        per1.BankName = sc.nextLine();
        System.out.println("Enter the Branch: ");
        per1.Branch = sc.nextLine();
        per1.accountOpen();
        while (true) {
            System.out.println("Choose what you want to do:");
            System.out.println("1. Deposit money");
```

```
            System.out.println("2. Withdraw money");
            System.out.println("3. Check balance");
            System.out.println("4. Calculate interest");
            System.out.println("5. Account details");
            System.out.println("6. Close account");
            int choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter the amount to deposit:
");
                    int a = sc.nextInt();
                    per1.depositMoney(a);
                    break;
                case 2:
                    System.out.println("Enter the amount to withdraw:
");
                    int b = sc.nextInt();
                    per1.withdrawMoney(b);
                    break;
                case 3:
                    per1.checkBalance();
                    break;
                case 4:
                    System.out.println("Enter the time in years: ");
                    int c = sc.nextInt();
                    per1.calculateInterest(rateofinterest, c);
                    break;
                case 5:
                    per1.AccountDetails();
                    break;
```

```
                case 6:
                    per1.closeAccount();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice");
            }
        }
    }
}
```

You can checkout my github for all the program files - [1rithwik (github.com)]

The concepts understood on (19-09-2024)

1. BufferReader -

BufferedReader is a class in Java that provides an efficient way to read text from an input stream, particularly when reading large amounts of data. It buffers characters to provide efficient reading of characters, arrays, and lines.

Key Features

- Buffering: It reads a larger block of data at once and reduces the number of I/O operations, making it more efficient than reading one character at a time.
- Reading Lines: It offers a convenient method to read an entire line of text at once.
- Character Encoding: Works well with different character encodings, allowing for versatile text input handling.

Methods -

- BufferedReader(Reader in): Constructs a BufferedReader that uses a default-sized input buffer.
- BufferedReader(Reader in, int sz): Constructs a BufferedReader with a specified buffer size.
- String readLine(): Reads a line of text. Returns null when the end of the stream is reached.
- int read(): Reads a single character. Returns -1 when the end of the stream is reached.
- int read(char[] cbuf, int off, int len): Reads characters into an array.

Example (not an example given in class):

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;



public class ConsoleInputExample {

    public static void main(String[] args) {

        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        System.out.println("Enter some text (type 'exit' to quit):");



        String input;

        try {
```

```
            while (!(input =
reader.readLine()).equalsIgnoreCase("exit")) {
                System.out.println("You entered: " + input);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

2. Access Modifiers -

Access modifiers in Java determine the visibility or accessibility of classes, methods, and variables. They help enforce encapsulation and control access to the components of a class.

Types of Access Modifiers

1. **Public**
    - Accessible from any other class.
    - No restrictions on visibility.
2. **Private**
    - Accessible only within the same class.
    - Not visible to subclasses or other classes in the same package.
3. **Protected**
    - Accessible within the same package and by subclasses, even if they are in different packages.
    - Provides a level of visibility between public and private.
4. **Default (Package-Private)**
    - If no access modifier is specified, the default modifier applies.
    - Accessible only within classes in the same package.
    - Not visible to classes in other packages.

Example : -(not an example given in class)

```
package sep_19;

public class AccessModifiers {

    public String name;

    protected String branch;

    private String secret;


    public AccessModifiers(String name, String branch, String secret) {

        this.name = name;

        this.branch = branch;

        this.secret = secret;

    }


    public void displayName() { // Public method

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

    }


    protected void displayBranch() { // Protected method

        System.out.println("Branch: " + branch);

    }


    void displayDefault() { // Default (package-private) method

        System.out.println("Default method: Accessed within the same
package.");

    }


    private void displaySecret() { // Private method

        System.out.println("Secret: " + secret);

    }
```

```
    public static void main(String[] args) {
        AccessModifiers a = new AccessModifiers("rithwik", "CSE",
"treasure");
        a.displaySecret(); // private method
        a.displayBranch(); // protected method
        a.displayDefault(); // default method
        a.displayName(); // public method
    }
}
```

OtherPackageExample.java file -

```
import sep_19.AccessModifiers;

public class OtherPackageExample extends AccessModifiers {
    public OtherPackageExample(String name, String branch, String
secret) {
        super(name, branch, secret);
    }

    public void displayAll() {
        displayName();
        displayBranch();
        // displayDefault(); // This line would cause a compile error
(default method)
        // displaySecret(); // This line would cause a compile error
(private method)
    }
```

```
    public static void main(String[] args) {
        OtherPackageExample obj = new OtherPackageExample("John",
"CSE", "treasure");
        obj.displayAll();
    }
}
```

3. Four pillars of OOPS -

Encapsulation

- Bundling data (attributes) and methods (functions) that operate on the data into a single unit or class.
- Restricts direct access to some components, which can prevent unintended interference and misuse.
- Achieved using access modifiers (public, private, protected).

Abstraction

- Hiding complex implementation details and exposing only the necessary features of an object.
- Simplifies interaction with the object by presenting a clear interface.
- Can be achieved through abstract classes and interfaces.

Inheritance

- Mechanism by which one class can inherit fields and methods from another class, promoting code reuse.
- Supports a hierarchical classification where a subclass can extend the functionality of a superclass.
- Facilitates polymorphism, allowing methods to be used in different contexts.

Polymorphism

- Ability for different classes to be treated as instances of the same class through a common interface.
- Allows methods to do different things based on the object it is acting upon.
- Achieved through method overloading (same method name, different parameters) and method overriding (subclass provides a specific implementation of a method declared in its superclass).

Example (not an example given in class)

```
// Abstraction: Abstract class to define a blueprint
abstract class Animal {
    String name;

    // Constructor
    Animal(String name) {
        this.name = name;
    }

    // Abstract method (no implementation)
    abstract void sound();

    // Concrete method
    void display() {
        System.out.println("Animal Name: " + name);
    }
}

// Inheritance: Dog class inherits from Animal
class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    // Polymorphism: Overriding the sound method
    @Override
    void sound() {
        System.out.println(name + " says: Woof!");
```

```
    }
}

// Inheritance: Cat class inherits from Animal
class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    // Polymorphism: Overriding the sound method
    @Override
    void sound() {
        System.out.println(name + " says: Meow!");
    }
}

public class OOPExample {
    public static void main(String[] args) {
        // Encapsulation: Creating objects
        Animal dog = new Dog("Buddy");
        Animal cat = new Cat("Whiskers");

        // Abstraction: Using methods defined in Animal class
        dog.display();
        dog.sound();

        cat.display();
        cat.sound();
    }
```

```
}
```

4. Input Output Operations :-

Java provides a rich set of classes for input and output (I/O) operations, primarily in the java.io package. These classes enable reading from and writing to data sources like files, memory, and network sockets.

Key Input and Output Classes

1. InputStream / OutputStream
    - InputStream: An abstract class representing an input stream of bytes. Subclasses include FileInputStream, BufferedInputStream, etc.
    - OutputStream: An abstract class representing an output stream of bytes. Subclasses include FileOutputStream, BufferedOutputStream, etc.
2. Reader / Writer
    - Reader: An abstract class for reading character streams. Subclasses include FileReader, BufferedReader, StringReader, etc.
    - Writer: An abstract class for writing character streams. Subclasses include FileWriter, BufferedWriter, PrintWriter, etc.
3. File Classes
    - File: Represents a file or directory path in the file system. Provides methods to create, delete, and manipulate files.
    - FileReader / FileWriter: Specialized classes for reading and writing files in character format.
4. Buffered Classes
    - BufferedReader: Buffers characters for efficient reading of text data. It provides methods like readLine() for reading lines of text.
    - BufferedWriter: Buffers characters for efficient writing of text data. It can write text in larger blocks, improving performance.
5. Print Classes
    - PrintWriter: Extends Writer and provides convenient methods for printing formatted representations of objects to a text-output stream.
    - PrintStream: Extends OutputStream and provides methods for printing formatted representations of data, including primitives.

Example (not an example given in class) -

```
import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;


public class FileIOExample {

    public static void main(String[] args) {

        String inputFile = "input.txt";

        String outputFile = "output.txt";


        // Writing to a file

        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(outputFile))) {

            writer.write("Hello, World!");

            writer.newLine();

            writer.write("Welcome to Java I/O.");

        } catch (IOException e) {

            e.printStackTrace();

        }


        // Reading from a file

        try (BufferedReader reader = new BufferedReader(new
FileReader(inputFile))) {

            String line;

            while ((line = reader.readLine()) != null) {

                System.out.println(line);

            }

        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

5. Custom exception -

A custom exception is a user-defined exception class that extends the existing exception classes in Java. Custom exceptions allow developers to create meaningful error messages and handle specific error conditions more effectively.

Creating a Custom Exception

1. Extending the Exception Class:
    - To create a custom checked exception, extend the Exception class.
    - To create a custom unchecked exception, extend the RuntimeException class.
2. Constructor:
    - Define one or more constructors that allow passing error messages or other parameters.

Example (apart from example taught in class) -

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
class UserRegistration {
    public void registerUser(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age is less than 18");
        } else {
            System.out.println("User registered successfully");
        }
    }
}
```

```java
public class CustomException {

    public static void main(String[] args) {

        UserRegistration obj = new UserRegistration();

        try {

            System.out.println("Enter the age:");

            Scanner sc = new Scanner(System.in);

            int n = sc.nextInt();

            obj.registerUser(n);

            sc.close();

        } catch (InvalidAgeException e) {

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

        }

    }

}
```

Checked Exceptions:

- These are exceptions that the compiler forces you to either handle with a try-catch block or declare using the throws clause in the method signature.
- Examples: IOException, SQLException, or custom exceptions that extend Exception (but not RuntimeException).

Unchecked Exceptions:

- These are exceptions that do not need to be declared in the method signature with throws, nor do they require you to handle them.
- Examples: NullPointerException, IllegalArgumentException, or any exceptions that extend RuntimeException.

All the programs are there in github - https://github.com/1rithwik

The concepts understood on(20-09-2024)

1. Enumeration

Enum, introduced in Java 5, is a special data type that consists of a set of pre-defined named values separated by commas. These named values are also known as elements or enumerators or enum instances.

- enum constants cannot be overridden
- enum doesn't support the creation of objects

Why we need enums is -

Sometimes inbuilt data types are not sufficient. Let's suppose that we have data of different data types required to be stored in a single variable. In such a situation, inbuilt data types won't fulfill the need. That's why there is a requirement for user-defined Data Types, and enum is one of them.

1. Enum constructors are always private.
2. We can't create instance of enum using new operator.
3. We can declare an abstract method in Java enum, then all the enum fields must implement the abstract method. In above example getDetail() is the abstract method and all the enum fields have implemented it.

Example -

```
public class EnumExample {


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

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the day of the week: ");
        String userInput = sc.nextLine().toUpperCase();
        Day today = Day.valueOf(userInput);

        switch (today) {
            case MONDAY:
                System.out.println("It's Monday, time to work!");
                break;
            case FRIDAY:
                System.out.println("It's Friday, the weekend is
near!");
                break;
            case SUNDAY:
                System.out.println("It's Sunday, time to relax!");
                break;
            default:
                System.out.println("It's just another day.");
                break;
        }
        sc.close();
    }
}
```

2. Cache in Java persistence

Definition: In JPA, caching is a way to store entities that have been retrieved from a database, so that subsequent requests for those entities can be served faster without hitting the database again.

Types of Caching:

- First-Level Cache (Session Cache):
    - This is the default cache that exists within the persistence context (e.g., EntityManager).
    - Each EntityManager instance has its own first-level cache.
    - Once an entity is retrieved, it stays in the cache until the EntityManager is closed.
    - This cache is not shared across different EntityManager instances.
- Second-Level Cache:
    - This is a shared cache across multiple EntityManager instances and is configured separately.
    - It can cache entities, collections, and query results.
    - Popular implementations include Hibernate's second-level cache with support for caching providers like Ehcache, Infinispan, or Hazelcast.

- You can use annotations like @Cacheable, @Cache in JPA or Hibernate to indicate which entities should be cached.

Benefits of Caching:

- Reduces database load by minimizing repeated queries for the same data.
- Improves application performance through faster access to cached data.
- Allows for better scalability by handling increased request loads.

Considerations:

- **Stale Data**: Caches can serve outdated data, so you need a strategy for cache invalidation or expiration.
- **Memory Usage**: Caching consumes memory, so you must manage cache size and eviction policies.

```
import java.io.*;

import java.util.*;
```

```
public class MovieRecommandCache {

    // L1 Cache Implementation

    private static final int MAX_ENTRIES = 100;

    private final Map<String, String> recentMoviewCache = new
LinkedHashMap<String, String>(100, 0.75f, true) {

        protected boolean removeEldestEntry(Map.Entry<String, String>
eldest) {

            return size() > MAX_ENTRIES;

        }

    };

    // L2 Cache Implementation

    private static final int MAX_ENTRIES_L2 = 1000;

    private final Map<String, String> popularMoviewCache = new
LinkedHashMap<String, String>(MAX_ENTRIES_L2, 0.75f,

            true) {

        protected boolean removeEldestEntry(Map.Entry<String, String>
eldest) {

            return size() > MAX_ENTRIES_L2;

        }

    };


}
```

Explaining of the above code:

The class MovieRecommandCache is designed to handle two levels of caching (L1 and L2) for movie recommendations.

L1 Cache: recentMoviewCache

- Purpose: The L1 Cache stores recent movie recommendations and has a maximum capacity of 100 entries.
- Key Characteristics:
    - Initial Capacity: 100 – The initial size of the cache.

    - Load Factor: 0.75f – This means the cache is resized when it's 75% full. In this case, resizing won't really happen because the capacity is fixed.
    - Access Order: true – This ensures that the entries are kept in access order, meaning the most recently accessed items are at the end of the cache, and the least recently accessed are at the beginning.
- Eviction Policy: The method removeEldestEntry() defines the eviction policy. When the cache exceeds 100 entries, the oldest (least recently accessed) entry is automatically removed.

L2 Cache: popularMoviewCache

```
private static final int MAX_ENTRIES_L2 = 1000;

private final Map<String, String> popularMoviewCache = new LinkedHashMap<String, String>(MAX_ENTRIES_L2, 0.75f, true) {

   protected boolean removeEldestEntry(Map.Entry<String, String> eldest) {

      return size() > MAX_ENTRIES_L2;

   }

};
```

- Purpose: The L2 Cache stores popular movie recommendations and has a maximum capacity of 1000 entries.
- Key Characteristics: Similar to the L1 cache, it uses a LinkedHashMap with:
    - Initial Capacity: 1000
    - Load Factor: 0.75f
    - Access Order: true
- Eviction Policy: The eviction policy here is that the cache removes the least recently accessed entry when the number of entries exceeds 1000.

4. How the Cache Works

- L1 Cache (recentMoviewCache): This stores up to 100 movie recommendations that were recently accessed. When it hits the 101st entry, the least recently accessed entry will be evicted to make room for the new one.
- L2 Cache (popularMoviewCache): This stores up to 1000 movie recommendations based on access frequency. Once it reaches capacity, the least recently accessed entry will be evicted, just like in the L1 cache.

Both caches are implemented using LinkedHashMap with access-order iteration (true), ensuring that the most recently used (MRU) item is always at the end of the list, and the least recently used (LRU) item is at the beginning. This allows for efficient eviction of the oldest items.

3. Collections in Java -

a. List -

A list in Java is an ordered collection (also known as a sequence) that allows duplicate elements. It provides a way to store and manipulate a sequence of items.

Key Interfaces:

1. List: The main interface representing a list, part of the Java Collections Framework.
2. ArrayList: A resizable array implementation of the List interface, offering fast random access to elements.
3. LinkedList: A doubly-linked list implementation of the List interface, suitable for frequent insertions and deletions.
4. Vector: A legacy synchronized implementation of the List interface, similar to ArrayList but thread-safe. Vector Class in Java - GeeksforGeeks

Common Methods:

- add(E element): Adds an element to the end of the list.
- get(int index): Retrieves an element at a specified index.
- remove(int index): Removes the element at the specified index.
- size(): Returns the number of elements in the list.
- contains(Object o): Checks if the list contains a specific element.
- indexOf(Object o): Returns the index of the first occurrence of a specified element.

Features:

- Dynamic Sizing: Lists can grow or shrink in size dynamically as elements are added or removed.
- Order: Elements are maintained in the order they were added.
- Index-Based Access: Elements can be accessed using their index.

Use Cases:

- Storing a collection of items where order and duplicates matter, such as a list of users or products.
- Implementing stack or queue data structures.

```
List<String> arrayList = new ArrayList<>();

        arrayList.add("Apple");

        arrayList.add("Banana");
```

```
        arrayList.add("Cherry");
        System.out.println("ArrayList: " + arrayList);

        // LinkedList: Doubly-linked list implementation
        List<String> linkedList = new LinkedList<>(arrayList);
        linkedList.add(1, "Blueberry");
        System.out.println("LinkedList: " + linkedList);

        // Vector: Thread-safe dynamic array (legacy class)
        Vector<String> vector = new Vector<>(arrayList);
        vector.add("Date");
        System.out.println("Vector: " + vector);

        // Stack: LIFO stack (extends Vector, legacy class)
        Stack<String> stack = new Stack<>();
        stack.push("Elderberry");
        stack.push("Fig");
        System.out.println("Stack: " + stack);
        System.out.println("Popped from stack: " + stack.pop());

        // Operations
        Collections.sort(arrayList);
        System.out.println("Sorted ArrayList: " + arrayList);
        System.out.println("Binary search for 'Cherry': " +
Collections.binarySearch(arrayList, "Cherry"));
```

```
    }
```

b. Utility classes -

Utility classes in Java are classes that contain static methods and fields. They provide commonly used functionalities that can be reused across different parts of an application. These classes are not meant to be instantiated.

Key Characteristics:

- **Static Methods**: All methods in utility classes are usually static, meaning they can be called on the class itself without creating an instance.
- **No Constructor**: Utility classes often have a private constructor to prevent instantiation.
- **Grouping of Related Methods**: They organize related functionalities into a single class for easier access and management.

Common Utility Classes:

1. java.util.Collections:
    - Provides static methods for operating on collections, such as sorting and searching.
    - Includes methods like sort(), shuffle(), reverse(), and max().
2. java.util.Objects:
    - Contains static utility methods for operating on objects, including null-safe comparisons.
    - Useful methods include equals(), hashCode(), requireNonNull(), and toString().
3. java.util.Arrays:
    - Contains static methods for working with arrays, such as sorting and searching.
    - Provides methods like sort(), binarySearch(), and copyOf().
4. java.lang.Math:
    - Contains methods for performing basic numeric operations, including trigonometric, logarithmic, and exponential functions.
    - Key methods include abs(), max(), min(), pow(), and sqrt().
5. java.util.UUID:
    - Used to generate universally unique identifiers (UUIDs).
    - The randomUUID() method generates a new random UUID.
6. java.util.concurrent.TimeUnit:
    - Provides time-related constants and methods to convert between different time units (e.g., seconds, minutes, hours).

Use Cases:

- Performing common operations without the need for instantiation.
- Reducing code duplication by centralizing frequently used methods.
- Enhancing code readability by grouping related functionalities.

Example:

```
int[] numbers = { 5, 3, 8, 1, 9, 2 };



        // 1. Sorting the array

        Arrays.sort(numbers);

        System.out.println("Sorted array: " +
Arrays.toString(numbers));



        // 2. Filling an array with a specific value

        int[] filledArray = new int[5];

        Arrays.fill(filledArray, 7); // Fills the array with the value
7

        System.out.println("Array filled with 7: " +
Arrays.toString(filledArray));



        // Collections utility class

        List<String> list = Arrays.asList("apple", "banana", "cherry");

        Collections.reverse(list);

        System.out.println("Reversed list: " + list);

        Collections.shuffle(list);

        System.out.println("Shuffled list: " + list);

        System.out.println("Max element: " + Collections.max(list));

        System.out.println("Min element: " + Collections.min(list));
```

```
        List<Integer> li = Collections.EMPTY_LIST;
        System.out.println(li);

        // Synchronized collections
        List<String> normalList = new ArrayList<>();

        // Making it a synchronized list
        List<String> synchronizedList =
Collections.synchronizedList(normalList);

        // Now, synchronizedList is thread-safe
        synchronizedList.add("Movie 1");
        synchronizedList.add("Movie 2");

        // We can safely access it in a multi-threaded environment
        System.out.println(synchronizedList);
    }

    public static void map() {
        System.out.println("\n--- Map Interfaces ---");
        Map<Integer, Integer> mp = new HashMap<Integer, Integer>();
        mp.put(2, 65);
        mp.put(22, 82);
        mp.put(8, 78);
        mp.put(1, 78);
```

```
        mp.put(3, 90);

        System.out.println(mp);



        LinkedHashMap<Integer, Integer> lmp = new LinkedHashMap<>();

        lmp.put(2, 65);

        lmp.put(8, 78);

        lmp.put(1, 78);

        lmp.put(3, 90);

        System.out.println(lmp);



        TreeMap<Integer, Integer> tmap = new TreeMap<>(mp);

        System.out.println(tmap);



        Hashtable<Integer, Integer> hm = new Hashtable<>(mp);

        System.out.println(hm);



        ConcurrentHashMap<Integer, Integer> chm = new
ConcurrentHashMap<>(mp);

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

c. Map -

A Map in Java is a collection that maps keys to values, allowing each key to be associated with exactly one value. It is part of the Java Collections Framework.

Key Interfaces:

1. Map: The main interface that represents the mapping of keys to values.
2. HashMap: A hash table-based implementation of the Map interface, providing fast access, insertion, and deletion.

3. LinkedHashMap: Similar to HashMap but maintains the order of insertion.
4. TreeMap: A red-black tree-based implementation that sorts the keys in natural order or by a specified comparator.
5. Hashtable: A legacy synchronized implementation of the Map interface, similar to HashMap but thread-safe.

Common Methods:

- put(K key, V value): Adds a key-value pair to the map.
- get(Object key): Retrieves the value associated with a specific key.
- remove(Object key): Removes the key-value pair associated with the specified key.
- size(): Returns the number of key-value pairs in the map.
- containsKey(Object key): Checks if a specific key exists in the map.
- keySet(): Returns a set of all keys in the map.
- values(): Returns a collection of all values in the map.
- entrySet(): Returns a set of all key-value pairs in the map.

Features:

- Key-Value Association: Each key maps to a single value, enabling efficient data retrieval.
- Dynamic Sizing: Maps can grow or shrink dynamically as entries are added or removed.
- Non-Ordered: The order of entries in a HashMap is not guaranteed, while LinkedHashMap maintains insertion order, and TreeMap sorts keys.

Use Cases:

- Storing configuration settings, caching data, or representing relationships between objects where lookups by key are frequent.

```
Map<Integer, Integer> mp = new HashMap<Integer, Integer>();

        mp.put(2, 65);

        mp.put(22, 82);

        mp.put(8, 78);

        mp.put(1, 78);

        mp.put(3, 90);

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

```
        LinkedHashMap<Integer, Integer> lmp = new LinkedHashMap<>();

        lmp.put(2, 65);

        lmp.put(8, 78);

        lmp.put(1, 78);

        lmp.put(3, 90);

        System.out.println(lmp);



        TreeMap<Integer, Integer> tmap = new TreeMap<>(mp);

        System.out.println(tmap);



        Hashtable<Integer, Integer> hm = new Hashtable<>(mp);

        System.out.println(hm);



        ConcurrentHashMap<Integer, Integer> chm = new
ConcurrentHashMap<>(mp);

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

d. Set

A Set in Java is a collection that does not allow duplicate elements and is part of the Java Collections Framework. It models the mathematical set abstraction.

Key Interfaces:

1. Set: The main interface representing a collection of unique elements.
2. HashSet: A hash table-based implementation of the Set interface, providing fast access, insertion, and deletion without guaranteeing order.
3. LinkedHashSet: Similar to HashSet but maintains the order of insertion.
4. TreeSet: A red-black tree-based implementation that sorts the elements in natural order or by a specified comparator.

Common Methods:

- add(E e): Adds an element to the set; returns true if the set did not already contain the element.
- remove(Object o): Removes the specified element from the set.
- contains(Object o): Checks if a specific element is present in the set.
- size(): Returns the number of elements in the set.
- isEmpty(): Checks if the set is empty.
- iterator(): Returns an iterator over the elements in the set.

Features:

- Uniqueness: Sets automatically handle duplicates, ensuring each element is stored only once.
- Dynamic Sizing: Sets can grow or shrink in size dynamically.
- Non-Ordered (for HashSet): The order of elements is not guaranteed; however, LinkedHashSet maintains insertion order, and TreeSet sorts elements.

Use Cases:

- Storing unique items, such as a collection of user IDs or product SKUs.
- Implementing mathematical set operations (union, intersection, difference).

```
Set<String> hashSet = new HashSet<>();

        hashSet.add("Red");

        hashSet.add("Green");

        hashSet.add("Blue");

        hashSet.add("Red"); // Duplicate, won't be added

        System.out.println("HashSet: " + hashSet);



        // LinkedHashSet: Hash table and linked list implementation,
maintains insertion

        // order

        Set<String> linkedHashSet = new LinkedHashSet<>();

        linkedHashSet.add("Dog");

        linkedHashSet.add("Cat");
```

```
        linkedHashSet.add("Bird");
        System.out.println("LinkedHashSet: " + linkedHashSet);

        // TreeSet: Red-black tree implementation, sorted order
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("Zebra");
        treeSet.add("Elephant");
        treeSet.add("Lion");
        System.out.println("TreeSet: " + treeSet);

        // Operations
        Set<String> set1 = new HashSet<>(Arrays.asList("15", "23",
"43"));
        Set<String> set2 = new HashSet<>(Arrays.asList("23", "21",
"15"));
        set1.retainAll(set2); // Intersection
        System.out.println("Intersection: " + set1);
```

e. Queue -

A Queue in Java is a collection that represents a first-in-first-out (FIFO) data structure. It allows elements to be added at one end (the tail) and removed from the other end (the head).

Key Interfaces:

1. Queue: The main interface that represents the queue data structure.
2. LinkedList: Implements the Queue interface and provides a dynamic queue that can grow as needed.
3. ArrayDeque: A resizable array implementation of the Deque interface, also used for queue operations.
4. PriorityQueue: A queue that orders elements based on their natural ordering or a specified comparator.

Common Methods:

- add(E e): Inserts an element at the tail of the queue; throws an exception if the queue is full.
- offer(E e): Inserts an element at the tail of the queue; returns false if the queue is full (non-blocking).
- remove(): Removes and returns the head of the queue; throws an exception if the queue is empty.
- poll(): Removes and returns the head of the queue; returns null if the queue is empty (non-blocking).
- peek(): Retrieves the head of the queue without removing it; returns null if the queue is empty.

Features:

- FIFO Order: Elements are processed in the order they were added.
- Dynamic Sizing: Queues can grow or shrink as needed.
- Blocking Queues: Some queue implementations, like ArrayBlockingQueue, support blocking operations for multithreaded applications.

Use Cases:

- Managing tasks in scheduling systems, like printer jobs or thread pools.
- Implementing breadth-first search (BFS) algorithms.
- Storing messages in messaging systems or event-driven architectures.

Example:

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

        pq.add(1);

        pq.add(2);

        pq.add(3);

        System.out.println(pq);

        System.out.println(pq.poll());// it will remove the first
element

        Queue<String> queue = new LinkedList<>();

        queue.offer("First");

        queue.offer("Second");

        queue.offer("Third");
```

```
        System.out.println("Queue: " + queue);

        System.out.println("Element: " + queue.element());

        Deque<String> deque = new ArrayDeque<>();

        // Adding elements to the front (head) and back (tail) of the
deque

        deque.addFirst("Element 1 (Head)"); // Add to head

        deque.addLast("Element 2 (Tail)"); // Add to tail

        deque.addFirst("Element 3 (Head)"); // Add another to head

        // Display the Deque

        System.out.println("Deque: " + deque);

        // Access elements at the head and tail

        System.out.println("Head element: " + deque.getFirst());

        System.out.println("Tail element: " + deque.getLast());



        // Remove elements from the front (head) and back (tail)

        String removedHead = deque.removeFirst(); // Remove from head

        System.out.println("Removed from head: " + removedHead);



        String removedTail = deque.removeLast(); // Remove from tail

        System.out.println("Removed from tail: " + removedTail);

        // Display the Deque after removals

        System.out.println("Deque after removals: " + deque);
```

Threading concept in Java -

A thread is like a small task or unit of a larger program that can run independently and simultaneously with other tasks. It allows multiple parts of a program to run at the same time, making the program more efficient, especially when performing tasks like handling multiple requests or doing background work.

Simple Example:

Imagine a program is like a kitchen where different chefs (threads) can cook different dishes at the same time. One chef might be boiling water, while another is chopping vegetables. Both tasks happen independently, but they contribute to the overall goal of preparing a meal.

In programming, threads allow different tasks, like downloading a file or processing user input, to happen simultaneously without waiting for each other to finish.

There are two ways to create a thread in our program. Using

1. Runnable interface
2. Thread class

Lifecycle of a Thread in Java

The Life Cycle of a Thread in Java refers to the state transformations of a thread that begins with its birth and ends with its death. When a thread instance is generated and executed by calling the start() method of the Thread class, the thread enters the runnable state. When the sleep() or wait() methods of the Thread class are called, the thread enters a non-runnable mode.

Thread returns from non-runnable state to runnable state and starts statement execution. The thread dies when it exits the run() process. In Java, these thread state transformations are referred to as the Thread life cycle.

Below are the stages in the lifecycle of a thread :

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead
6. New State

As we use the Thread class to construct a thread entity, the thread is born and is defined as being in the New state. That is, when a thread is created, it enters a new state, but the start() method on the instance has not yet been invoked.

Runnable State

A thread in the runnable state is prepared to execute the code. When a new thread's start() function is called, it enters a runnable state.

In the runnable environment, the thread is ready for execution and is awaiting the processor's availability (CPU time). That is, the thread has entered the queue (line) of threads waiting for execution.

Running State

Running implies that the processor (CPU) has assigned a time slot to the thread for execution. When a thread from the runnable state is chosen for execution by the thread scheduler, it joins the running state.

In the running state, the processor allots time to the thread for execution and runs its run procedure. This is the state in which the thread directly executes its operations. Only from the runnable state will a thread enter the running state.

Blocked State

When the thread is alive, i.e., the thread class object persists, but it cannot be selected for execution by the scheduler. It is now inactive.

Dead State

When a thread's run() function ends the execution of sentences, it automatically dies or enters the dead state. That is, when a thread exits the run() process, it is terminated or killed. When the stop() function is invoked, a thread will also go dead.

Java Thread Priorities

The number of services assigned to a given thread is referred to as its priority. Any thread generated in the JVM is given a priority. The priority scale runs from 1 to 10.

1 is known as the lowest priority.

5 is known as standard priority.

10 represents the highest level of priority.

Thread Synchronization in Java

Thread synchronization in Java is a mechanism that ensures that two or more concurrent threads do not simultaneously access shared resources, which could lead to data inconsistency and unpredictable behavior.

1. Concurrency: Multiple threads executing simultaneously, potentially sharing data and resources.
2. Race Condition: A situation where the outcome of a program depends on the sequence or timing of uncontrollable events, leading to inconsistent data.
3. Critical Section: A part of the code where shared resources are accessed. Only one thread should execute within a critical section at a time to maintain data integrity.

Synchronization Mechanisms:

1. Synchronized Methods:
    - Declaring a method with the synchronized keyword ensures that only one thread can execute that method at a time for a given object.

Example:

```
public synchronized void methodName() {
  // critical section
}
```

2. Synchronized Blocks:
    - More flexible than synchronized methods, allowing synchronization of specific blocks of code.

Example:

```
public void methodName() {
  synchronized(this) {
    // critical section
  }
}
```

Thread methods in Java:

1. start():
    - Starts a new thread and invokes the run() method.
    - Allows the thread to begin execution in parallel with other threads.
2. run():
    - Contains the code that defines the task performed by the thread.
    - It's invoked automatically when start() is called, or can be run directly without creating a new thread (but not recommended).
3. sleep(long millis):
    - Causes the currently executing thread to pause for a specified number of milliseconds.
    - Allows other threads to execute while the current thread is sleeping.
4. join():
    - Causes the current thread to wait until the thread on which join() was called completes its execution.
    - Ensures sequential execution when needed in a multi-threaded environment.

5. interrupt():
    - Interrupts a sleeping or waiting thread, causing it to throw an InterruptedException.
    - Allows graceful termination of a thread or signals it to stop.
6. getName():
    - Returns the name of the thread.
    - Useful for identifying or debugging threads in a multi-threaded program.
7. setPriority(int newPriority):
    - Sets the priority of a thread (ranges from Thread.MIN_PRIORITY to Thread.MAX_PRIORITY).
    - Affects the thread's scheduling but doesn't guarantee the order of execution.
8. isAlive():
    - Checks if the thread has started and is still running.
    - Returns true if the thread is alive; false if it has completed.

Example(not same example as taught in class):

```
public class ThreadingExample {

    // volatile keyword ensures that the variable is always read from
main memory

    // private volatile boolean flag = false;



    // synchronized method to demonstrate thread synchronization

    public synchronized void synchronizedMethod() {

        System.out.println(Thread.currentThread().getName() + " entered
synchronized method");

        try {
```

```
            Thread.sleep(10000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        System.out.println(Thread.currentThread().getName() + " exiting
synchronized method");

    }



    final static Object lock = new Object();



    // Runnable interface implementation

    static class MyRunnable implements Runnable {

        public void run() {

            System.out.println(Thread.currentThread().getName() + " is
running");

            try {

                Thread.sleep(3000);

            } catch (Exception e) {

                e.printStackTrace();

            }
```

```
            System.out.println(Thread.currentThread().getName() +
"completed running");



            synchronized (lock) {

                System.out.println(Thread.currentThread().getName() + "
entered synchronized block");

                try {

                    Thread.sleep(5000); // Simulate work inside
synchronized block

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

                System.out.println(Thread.currentThread().getName() + "
exiting synchronized block");

            }

        }

    }



    // Thread class extension

    class MyThread extends Thread {

        public void run() {
```

```
            System.out.println(Thread.currentThread().getName() + " is
running");

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

                e.printStackTrace();

            }

            System.out.println(Thread.currentThread().getName() +
"completed running");



            synchronized (lock) {

                System.out.println(Thread.currentThread().getName() + "
entered synchronized block");

                try {

                    Thread.sleep(5000); // Simulate work inside
synchronized block

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

                System.out.println(Thread.currentThread().getName() + "
exiting synchronized block");

            }
```

```
        }

    }



    public void demonstrateThreading() {

        // Creating and starting a thread using Runnable

        Thread thread1 = new Thread(new MyRunnable(),
"RunnableThread");

        thread1.start();



        // MyRunnable runnable = new MyRunnable();

        // runnable.setName("RunnableThread"); we cannot call start()
on runnable

        // runnable.start();



        // Creating and starting a thread by extending Thread class

        MyThread thread2 = new MyThread();

        thread2.setName("ExtendedThread");

        thread2.start();



        // Creating a thread using lambda expression (Java 8+)
```

```
        Thread thread3 = new Thread(() ->
System.out.println(Thread.currentThread().getName() + " is running"),

                "LambdaThread");

        thread3.start();



        // Demonstrating thread states

        System.out.println("Thread 3 state: " + thread3.getState());



        // Thread priority

        thread3.setPriority(Thread.MAX_PRIORITY);

        System.out.println("Thread 3 priority: " +
thread3.getPriority());



        // Joining threads

        try {

            thread1.join();// main thread will wait for thread1 to
complete

            thread2.join();// main thread will wait for thread2 to
complete

            thread3.join();// main thread will wait for thread3 to
complete

        } catch (InterruptedException e) {
```

```
            e.printStackTrace();

        }



        // Demonstrating synchronized block

        synchronized (this) {

            System.out.println(Thread.currentThread().getName() + " in
synchronized block");

        }



        // Using wait() and notify()

        Thread waiter = new Thread(() -> {

            synchronized (lock) {

                try {

                    System.out.println(Thread.currentThread().getName()
+ " is in synchronized block");

                    System.out.println("Waiter is waiting");

                    lock.wait();

                    System.out.println("Waiter is notified");

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }
```

```
            }
        });
        Thread notifier = new Thread(() -> {
            synchronized (lock) {
                System.out.println("Notifier is notifying");
                lock.notify();
            }
        });
        waiter.start();
        try {
            System.out.println("Main thread is sleeping");
            Thread.sleep(2000);
            System.out.println("Main thread woke up");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        notifier.start();
        // Demonstrating interrupt
        Thread sleeper = new Thread(() -> {
```

```
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                System.out.println("Sleeper was interrupted");
            }
        });
        sleeper.start();
        sleeper.interrupt();
        // Using ThreadLocal
        ThreadLocal<Integer> threadLocal = new ThreadLocal<>();
        threadLocal.set(42);
        System.out.println("ThreadLocal value: " + threadLocal.get());
    }
    public static void main(String[] args) {
        ThreadingExample obj = new ThreadingExample();
        obj.demonstrateThreading();
        obj.synchronizedMethod();
    }
}
```

I would also explain this example in simple terms:

1. Main Method Invocation:
    - The execution begins in the main method, where an instance of ThreadingExample is created (obj), and the demonstrateThreading() method is called.
2. Creating and Starting Threads:
    - Inside demonstrateThreading():
        - Runnable Thread:
            - A new thread (thread1) is created using the MyRunnable class. This thread is started with thread1.start().
            - Execution in MyRunnable:
                - thread1 prints its name and simulates work by sleeping for 3 seconds.
                - After waking up, it prints a completion message and enters the synchronized block.
                - Inside the synchronized block, it prints its entry message, sleeps for 5 seconds to simulate further work, and then prints its exit message.
        - Extended Thread:
            - A second thread (thread2) is created by extending the Thread class using MyThread and started with thread2.start().
            - Execution in MyThread:
                - thread2 prints its name, sleeps for 1 second, and then completes the running message.
                - It enters the synchronized block, where it follows the same process as thread1.
        - Lambda Thread:
            - A third thread (thread3) is created using a lambda expression and started.
            - Execution in Lambda Thread:
                - thread3 prints its name and completes its execution.
3. Thread States and Priorities:
    - The main thread prints the state of thread3 and sets its priority to maximum.

4. Joining Threads:
    - The main thread calls join() on thread1, thread2, and thread3, ensuring it waits for all three threads to complete their execution before proceeding.
5. Entering a Synchronized Block in Main:
    - After all threads complete, the main thread enters its own synchronized block, printing a message.
6. Demonstrating Wait and Notify:
    - Two additional threads (waiter and notifier) are created:
        - Waiter Thread:
            - The waiter thread enters a synchronized block, prints its status, and calls wait(), pausing its execution until notified.
        - Notifier Thread:
            - After a short delay (main thread sleeps for 2 seconds), the notifier thread enters its synchronized block and calls notify(), waking up the waiter.
7. Demonstrating Interrupt:
    - A sleeper thread is created to sleep for 5 seconds.
    - Immediately after starting, the main thread interrupts sleeper, causing it to throw an InterruptedException and print a message.
8. Using ThreadLocal:
    - A ThreadLocal variable is created and set to 42. The main thread prints its value, demonstrating that each thread can maintain its own separate value.
9. Program Completion:
    - Once all threads have executed, the demonstrateThreading() method concludes, and control returns to the main method, where the synchronized method (synchronizedMethod()) is called.
10. Final Synchronized Method:
    - In synchronizedMethod(), the main thread enters the synchronized section, sleeps for 10 seconds, and then exits, completing the program.

The programs are also in the github - https://github.com/1rithwik

The concepts understood on(23-09-2024)

1. Synchronization -

Synchronization in Java is a mechanism that ensures that two or more concurrent threads do not simultaneously execute a particular piece of code. It is mainly used to prevent thread interference and to ensure memory consistency.

Example -

```
class BankAccount {
    private volatile int balance;

    BankAccount() {
        balance = 0;
    }

    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println("Deposited: " + amount + " New Balance: " +
balance);
    }

    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount + " New Balance:
" + balance);
        } else {
            System.out.println("Insufficient balance");
        }
    }
```

```
    public int getBalance() {
        return balance;
    }
}

public class BankDemo {
    public static void main(String[] args) throws InterruptedException
{
        System.out.println("Bank Demo");
        BankAccount account = new BankAccount();
        Thread depositThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.deposit(100);

            }
        }, "Deposit Thread");
        Thread withdrawThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.withdraw(80);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "Withdraw Thread");
        depositThread.start();
        Thread.sleep(10);
        withdrawThread.start();
```

```
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final Balance: " + account.getBalance());
    }
}
```

BankAccount Class:

- Private Variable:
    - private volatile int balance;: The balance variable holds the current balance of the bank account and is marked as volatile to ensure visibility across threads.
- Constructor:
    - BankAccount(): Initializes the balance to 0.
- Synchronized Methods:
    - public synchronized void deposit(int amount):
        - This method allows a thread to add money to the account.
        - The synchronized keyword ensures that only one thread can execute this method at a time, preventing race conditions when updating the balance.
        - It prints the deposited amount and the new balance.
    - public synchronized void withdraw(int amount):
        - This method allows a thread to withdraw money from the account.
        - It first checks if there are sufficient funds before proceeding.
        - Similar to deposit, this method is synchronized to prevent multiple threads from accessing it simultaneously, ensuring consistent balance updates.
- Get Balance Method:
    - public int getBalance(): This method returns the current balance, which is not synchronized, as it only reads the balance.

BankDemo Class:

- Main Method:

    - Creates a BankAccount instance.
    - Two threads are created: one for depositing and one for withdrawing.
- Deposit Thread:
    - This thread deposits 100 units into the account five times in a loop.
- Withdraw Thread:
    - This thread attempts to withdraw 80 units from the account five times, with a 100ms pause between each withdrawal to simulate time taken for operations.
- Thread Execution:
    - The deposit thread starts first, followed by a brief sleep to allow deposits to happen before the withdraw thread starts.
    - After both threads complete their execution, the final balance is printed.

We can use wait() and notify() methods to manage the synchronization between the deposit and withdraw operations. This approach will allow the withdraw thread to wait until there is enough balance available after a deposit occurs, instead of using Thread.sleep() to create an artificial delay.

The code will look like this:

```
public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println("Deposited: " + amount + " New Balance: " +
balance);
        notifyAll();
    }

    public synchronized void withdraw(int amount) {
        while (balance < amount) {
            try {
                wait();
            } catch (Exception e) {
                Thread.currentThread().interrupt();
            }
        }
        balance -= amount;
```

```
        System.out.println("Withdrawn: " + amount + " New Balance: " +
balance);

    }
```

2. Understanding implementation of Cache -

a. Simple cache -

It is the just the implementation of map (any mapping technique can be used)

```
public class SimpleCache<K, V> {

    private final Map<K, V> cache;


    public SimpleCache() {

        this.cache = new HashMap<>();

    }

    public void put(K key, V value) {

        cache.put(key, value);

    }

    public void remove(K key) {

        cache.remove(key);

    }

    public void clear() {

        cache.clear();

    }

    public int size() {

        return cache.size();

    }

    public static void main(String[] args) {

        SimpleCache<String, String> cache = new SimpleCache<>();

        cache.put("key1", "value1");

        cache.put("key2", "value2");
```

```
        cache.put("key3", "value3");

        System.out.println(cache.get("key1"));

        cache.remove("key2");

        System.out.println(cache.size());

    }

}
```

b. LRU Cache -

An LRU (Least Recently Used) cache is a type of data structure that stores a limited number of items. When the cache reaches its capacity, it evicts the least recently used item to make room for a new one. This ensures that frequently accessed items remain available, while less accessed ones are removed.

```
public class LruCache<K, V> extends LinkedHashMap<K, V> {

    private final int capacity;

    public LruCache(int capacity) {

        super(capacity, 0.75f, true);// in linkedhashmap class we have
constructor which takes 3 parameters

        this.capacity = capacity;

    }

    @Override

    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {

        return size() > capacity;

    }

    public static void main(String[] args) {

        LruCache<String, String> cache = new LruCache<>(3);

        cache.put("key1", "value1");

        cache.put("key2", "value2");

        cache.put("key3", "value3");

        System.out.println(cache);

        cache.get("key2");// this will make key2 as the most recently
used i.e last element
```

```
        cache.put("key4", "value4");

        cache.put("key5", "value5");

        System.out.println(cache);

    }

}
```

Class :

- public class LruCache<K, V> extends LinkedHashMap<K, V>: The LruCache class extends LinkedHashMap, which maintains insertion order and allows custom ordering based on access.

Constructor:

- public LruCache(int capacity): Initializes the cache with a specified capacity. The super call configures the map with:
    - capacity: Initial capacity.
    - 0.75f: Load factor (default).
    - true: Enables access order, which is crucial for LRU functionality.

Eviction Logic:

- protected boolean removeEldestEntry(Map.Entry<K, V> eldest): This method checks if the size of the cache exceeds its capacity. If it does, the eldest (least recently used) entry is removed.

Main Method:

- An instance of LruCache is created with a capacity of 3.
- Key-value pairs are added to the cache.
- The cache contents are printed to show current entries.
- Accessing key2 makes it the most recently used.
- Adding new keys (key4 and key5) will cause the cache to evict the least recently used entries (first key1, then key3). Here key3 is removed because key 2 is recently used.

c. Guava Cache -

Guava Cache is a powerful caching library provided by Google's Guava library. It offers a flexible and high-performance cache implementation, allowing developers to store objects in memory with various eviction policies, expiration strategies, and loading mechanisms.

1. Eviction type which we can implement automatic:
    - Size-based eviction: Evicts entries when a specified size limit is reached.
    - Time-based eviction: Evicts entries after a specified duration since their last access or insertion.
2. Loading Mechanism:
    - You can define a loading strategy using the CacheLoader class, which automatically loads entries when they are requested and not present in the cache.
3. Concurrency:
    - Guava Cache is designed to be thread-safe, allowing concurrent access from multiple threads without manual synchronization.
4. Custom Expiration Policies:
    - You can set expiration times for cached entries based on creation or last access times.

Example(other example not given in class) -

```
public class GuavaCacheExample {

    public static void main(String[] args) {

        // Create a cache that holds String keys and Integer values

        Cache<String, Integer> cache = CacheBuilder.newBuilder()

                .maximumSize(100) // Maximum number of entries in the
cache

                .expireAfterWrite(10, TimeUnit.MINUTES) // Expire
entries 10 minutes after write

                .build();


        // Adding values to the cache

        cache.put("key1", 1);

        cache.put("key2", 2);


        // Retrieving values from the cache

        Integer value1 = cache.getIfPresent("key1");

        System.out.println("Value for key1: " + value1); // Should
print 1
```

```
        // Attempting to retrieve a non-existent key
        Integer value3 = cache.getIfPresent("key3");
        System.out.println("Value for key3: " + value3); // Should
print null


        // Invalidating a specific key
        cache.invalidate("key1");


        // Checking the value after invalidation
        Integer value1AfterInvalidation = cache.getIfPresent("key1");
        System.out.println("Value for key1 after invalidation: " +
value1AfterInvalidation); // Should print null
    }
}
```

Creating the Cache:

- CacheBuilder.newBuilder(): This starts the construction of a cache.
- .maximumSize(100): Limits the cache to hold a maximum of 100 entries. When the limit is exceeded, the cache will evict entries using a least-recently-used (LRU) policy.
- .expireAfterWrite(10, TimeUnit.MINUTES): Specifies that entries should expire 10 minutes after they are written. After this time, the entries will be considered expired and eligible for removal.
- .build(): This method builds the cache instance.

Adding Values:

- cache.put("key1", 1): This adds an entry with key "key1" and value 1.
- cache.put("key2", 2): This adds another entry with key "key2" and value 2.

Retrieving Values:

- cache.getIfPresent("key1"): This retrieves the value associated with "key1". If the key does not exist, it returns null.
- The retrieved value is printed.

Handling Non-Existent Keys:

- Trying to retrieve "key3" (which hasn't been added) will return null.

Invalidating Keys:

- cache.invalidate("key1"): This explicitly removes the entry for "key1" from the cache.
- After invalidation, attempting to retrieve "key1" again confirms that it is no longer present.

d. Two level cache -

A two-level cache is a caching strategy that utilizes two layers of caches to improve performance and reduce latency.

Components

1. **Level 1 (L1) Cache**:
    - **Location**: Typically located closer to the CPU (e.g., on-chip).
    - **Speed**: Very fast, often made from static RAM (SRAM).
    - **Size**: Generally smaller (e.g., a few kilobytes to several megabytes).
    - **Purpose**: Holds frequently accessed data and instructions for immediate access, reducing the time it takes to fetch data from main memory.
2. **Level 2 (L2) Cache**:
    - **Location**: Can be on-chip (still close to the CPU) or off-chip (slightly further away).
    - **Speed**: Slower than L1 but faster than main memory (often dynamic RAM - DRAM).
    - **Size**: Larger than L1, typically ranging from a few megabytes to tens of megabytes.
    - **Purpose**: Acts as a buffer for L1 cache misses, storing data that is likely to be reused soon.

Benefits

- **Improved Performance**: By keeping frequently accessed data in the L1 cache, the system reduces access times significantly.
- **Reduced Latency**: Fetching data from L1 is faster than fetching from L2 or main memory, which enhances overall system performance.
- **Better Resource Utilization**: L2 cache allows for a larger data set to be cached without significantly impacting speed, effectively balancing speed and capacity.

Use Cases

- Commonly used in CPUs to optimize instruction and data access.
- Can also be implemented in software caching strategies to improve application performance by reducing database or network calls.

Example -

```
long startTime = System.nanoTime();

        for (int i = 0; i < 10000; i++) {

            int key = i % 200; // This will cause hits in both levels
and some misses

            getWithTwoLevelCache(key);

        }

        long endTime = System.nanoTime();

        System.out.println("Time taken: " + (endTime - startTime) /
1_000_000.0 + " ms");

        System.out.println("L1 Cache size: " + l1Cache.size() + ", L2
Cache size: " + l2Cache.size());

private static String getWithTwoLevelCache(int key) {

        // Check L1 Cache

        if (l1Cache.containsKey(key)) {

            return l1Cache.get(key);

        }

        // Check L2 Cache

        if (l2Cache.containsKey(key)) {

            String value = l2Cache.get(key);

            // Move to L1 cache

            if (l1Cache.size() >= L1_CAPACITY) {

                // If L1 is full, remove the first entry (simulating
LRU)

                int oldestKey = l1Cache.keySet().iterator().next();
```

```
                l1Cache.remove(oldestKey);
            }
            l1Cache.put(key, value);
            return value;
        }
        // Not in cache, get from database
        String value = database.get(key);
        // Add to L2 cache
        if (l2Cache.size() >= L2_CAPACITY) {
            // If L2 is full, remove the first entry (simulating LRU)
            int oldestKey = l2Cache.keySet().iterator().next();
            l2Cache.remove(oldestKey);
        }
        l2Cache.put(key, value);
        return value;
    }
```

1. When you request a key, **L1** is checked first. If the key is found, you get a cache hit.
2. If **L1** misses, **L2** is checked. If the key is found in **L2**, it is moved to **L1** for faster future access.
3. If both caches miss, the value is fetched from the database and inserted into **L2**.
4. **L1** has a much smaller size than **L2**, so it's meant to store the most frequently accessed items.
5. **LRU (Least Recently Used) Evictions**: Both **L1** and **L2** remove the oldest entry when they exceed their capacity.

e. Concurrent or Thread safe cache -

A concurrent cache, or thread-safe cache, is designed to handle multiple threads accessing and modifying the cache simultaneously without causing data corruption or inconsistency. Here are the key points:

1. **Thread Safety**: Ensures that operations on the cache (e.g., adding, retrieving, or invalidating entries) are safe to perform from multiple threads at the same time.
2. **Locking Mechanisms**: Utilizes various strategies, such as fine-grained locking, read-write locks, or lock-free algorithms, to manage concurrent access efficiently.
3. **Atomic Operations**: Provides atomic methods for adding, removing, or updating cache entries to avoid race conditions.

Benefits

- **Performance**: Allows multiple threads to access the cache concurrently, leading to better performance in multi-threaded applications.
- **Scalability**: Can handle a growing number of threads without significant degradation in performance.
- **Consistency**: Maintains data integrity across threads, ensuring that cached data remains accurate.

Implementation Examples

- **Java ConcurrentHashMap**: Often used for thread-safe caching implementations.
- **Guava Cache**: Provides thread-safe caching with configurable eviction policies.
- **Caffeine**: A high-performance caching library for Java that supports concurrent access.

Use Cases

- Web applications that need to cache frequently accessed data to reduce latency.
- Multi-threaded services that require shared access to cached resources, such as database query results or API responses.

Example -

```
System.out.println("\nTesting Concurrent Cache:");

        long startTime = System.nanoTime();

        // Simulate concurrent access with multiple threads

        Thread[] threads = new Thread[10];

        for (int t = 0; t < threads.length; t++) {

            threads[t] = new Thread(() -> {

                for (int i = 0; i < 1000; i++) {

                    int key = i % 200;

                    getWithConcurrentCache(key);

                }

            });
```

```
            threads[t].start();
        }
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        long endTime = System.nanoTime();
        System.out.println("Time taken: " + (endTime - startTime) /
1_000_000.0 + " ms");
        System.out.println("Cache size: " + concurrentCache.size());
private static String getWithConcurrentCache(int key) {
        return concurrentCache.computeIfAbsent(key, k ->
database.get(k));
    }
```

Explanation -

The concurrent cache uses a ConcurrentHashMap to allow multiple threads to safely access and update the cache concurrently without needing explicit synchronization.

computeIfAbsent(key, k -> database.get(k)) is used to either:

- Return the value if the key is already in the cache.
- Load the value from the database if the key is not present, and then cache the result.

1. Multiple threads are launched, each making 1,000 requests to the cache.
2. Each thread may check if the cache contains the requested key.
3. If the key is missing, the value is fetched from the database and stored in the cache using the computeIfAbsent method. This ensures that only one thread fetches the value for a particular key, while others will wait until the value is available.

The concepts understood on(24-09-2024)

1. Executor framework -

The Executor Framework provides a higher-level replacement for managing threads in Java, offering an easy and flexible way to manage concurrent tasks.

Key Components

1. **Executor Interface**: The core interface that defines methods for managing and controlling the execution of asynchronous tasks.
2. **ExecutorService Interface**: Extends the Executor interface and provides methods for managing the lifecycle of tasks (like shutdown) and returning results.
3. **ScheduledExecutorService Interface**: Extends ExecutorService to support scheduling tasks to run after a given delay or periodically.
4. **Thread Pool Executors**: These allow you to create a pool of threads to execute tasks concurrently. They manage thread creation, reuse, and lifecycle automatically.

Key Classes

- Executors: A factory class for creating different types of Executor and ExecutorService instances, including:
    - newFixedThreadPool(int n): Creates a thread pool with a fixed number of threads.
    - newCachedThreadPool(): Creates a thread pool that creates new threads as needed and reuses previously constructed threads.
    - newSingleThreadExecutor(): Creates a single-threaded executor that executes tasks sequentially.
- Future: Represents the result of an asynchronous computation. It provides methods to check if the task is complete, retrieve the result, and handle exceptions.

Advantages

- **Thread Management**: Simplifies thread management by handling thread creation, reuse, and destruction automatically.
- **Concurrency Control**: Provides features to control concurrency, such as specifying the number of threads and task scheduling.
- **Improved Resource Utilization**: Helps optimize resource usage by reusing threads instead of constantly creating new ones.
- **Task Lifecycle Management**: Offers methods to manage the lifecycle of tasks and control their execution.

Use Cases

- Running background tasks that do not require immediate results.
- Performing tasks in parallel to improve performance.
- Scheduling tasks to run at specific times or intervals.

Example (not shown in class) -

```
public class CacheMain {

    public static void main(String[] args) {

        // Demo task

        Runnable runnableTask = () -> {

            try {

                TimeUnit.MILLISECONDS.sleep(1000);

                System.out.println("Current time :: " +
LocalDateTime.now());

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        };

        // Executor service instance

        ExecutorService executor = Executors.newFixedThreadPool(10);

        // 1. execute task using execute() method

        executor.execute(runnableTask);

        // 2. execute task using submit() method

        Future<String> result = executor.submit(runnableTask, "DONE");


        while (result.isDone() == false) {

            try {

                System.out.println("The method return value : " +
result.get());

                break;

            } catch (InterruptedException | ExecutionException e) {

                e.printStackTrace();

            }

            // Sleep for 1 second
```

```
            try {
                Thread.sleep(1000L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Shut down the executor service
        executor.shutdownNow();
    }
}
```

Explanation:

Runnable Task:

- A Runnable task is created that simulates a delay of 1 second.
- After the delay, it prints the current time.
- It includes error handling for potential interruptions.

Executor Service Creation:

- An ExecutorService is instantiated with a fixed thread pool of 10 threads.
- This allows the execution of multiple tasks concurrently.

Task Execution:

- The task is submitted to the executor using the execute() method, which runs the task without returning a result.
- The task is also submitted using the submit() method, which allows capturing the result as a Future object.

Checking Task Completion:

- A loop is employed to check if the task is completed using the isDone() method of the Future object.
- If the task is complete, the result is retrieved using get() and printed.
- If the task is not complete, the loop pauses for 1 second before checking again.

Handling Exceptions:

- Both task interruptions and execution failures are managed with exception handling, printing the stack trace if any occur.

Executor Shutdown:

- The executor service is shut down immediately using shutdownNow(), stopping any ongoing tasks and freeing up resources.

2. Future Interface -

A Future interface provides methods to check if the computation is complete, to wait for its completion and to retrieve the results of the computation. The result is retrieved using Future's get() method when the computation has completed, and it blocks until it is completed.

Future and FutureTask both are available in java.util.concurrent package

Key Features

1. Asynchronous Result Retrieval:
    - A Future allows you to initiate a task that runs asynchronously. You can later retrieve the result once the task is complete.
2. Blocking Method:
    - The get() method is used to obtain the result of the computation. If the task is still running, get() will block the calling thread until the result is available.
3. Exception Handling:
    - If the task completes exceptionally (due to an error), calling get() will throw an ExecutionException. This allows you to handle any exceptions that occurred during the execution of the task.
4. Cancellation:
    - The Future interface provides methods like cancel(boolean mayInterruptIfRunning) to attempt to cancel the task. If the task has not started yet or has already completed, the cancellation will fail.
5. Task Status:
    - You can check whether the task is completed, canceled, or running using the methods:
        - isDone(): Returns true if the task has completed.
        - isCancelled(): Returns true if the task was canceled before it completed.

Common Use Cases

- Long-Running Tasks: When tasks take a significant amount of time, using Future allows the main thread to continue execution while waiting for the result.
- Parallel Processing: When executing multiple tasks concurrently, you can collect their results using Future instances.

- Handling Time-Consuming Operations: Suitable for operations such as I/O tasks, network calls, or computations that may block.

Example -

In a scenario where wesubmit a task to calculate a value, wecan:

1. Submit the task using an ExecutorService, which returns a Future.
2. Continue processing other tasks or actions.
3. Retrieve the result using future.get()

```
public class CacheMain {

    public static void main(String[] args) {

        // Create a fixed thread pool with 2 threads

        ExecutorService executorService =
Executors.newFixedThreadPool(2);

        // Submit a task to calculate the factorial of 5

        Future<Long> futureFactorial = executorService.submit(new
FactorialTask(5));



        // Continue processing other tasks or actions

        System.out.println("Main thread is doing other work...");



        // Simulate doing other work with sleep

        try {

            Thread.sleep(1000); // Simulating other processing time

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        // Now retrieve the result of the factorial calculation

        try {

            Long result = futureFactorial.get(); // This will block if
the computation is not done yet
```

```
            System.out.println("Factorial result: " + result);
        } catch (InterruptedException e) {
            System.err.println("Task was interrupted.");
        } catch (ExecutionException e) {
            System.err.println("An error occurred while executing the
task: " + e.getCause());
        } finally {
            // Shutdown the executor service
            executorService.shutdown();
        }
    }
}
class FactorialTask implements Callable<Long> {
    private final int number;

    public FactorialTask(int number) {
        this.number = number;
    }
    @Override
    public Long call() throws Exception {
        return calculateFactorial(number);
    }
    private Long calculateFactorial(int n) {
        long factorial = 1;
        for (int i = 1; i <= n; i++) {
            factorial *= i;
        }
        return factorial;
    }
```

```
}
```

3. Transient keyword -

The transient keyword in Java is used to indicate that a particular field should not be serialized when an object is converted into a byte stream. This is particularly useful when you want to control which parts of an object's state are saved and which are not.

Key Points about transient:

1. Serialization: When an object is serialized (for example, when writing to a file or sending over a network), the Java Serialization mechanism saves the state of all non-transient fields. Fields marked as transient are ignored.
2. Use Cases:
    - Sensitive Information: You might not want to serialize sensitive information (like passwords).
    - Non-Serializable Fields: If a field refers to an object that does not implement the Serializable interface, you can mark it as transient to avoid serialization errors.
    - Temporary or Derived Data: Fields that can be derived from other fields or that are only relevant during the object's lifecycle (like currentPage in your Book class) can be marked as transient.

Example -

```
private String isbn;

        private String title;

        private String author;

        // trainsent feild, it would not be serialized

        transient int currentPage = 0;
```

4. Atomic Integer -

AtomicInteger is a class in the java.util.concurrent.atomic package that provides an integer value that may be updated atomically. It is designed for use in concurrent programming and allows multiple threads to update an integer value without the need for synchronization.

Key Features

1. Atomic Operations:

- AtomicInteger supports various atomic operations like incrementing, decrementing, and comparing and setting values. These operations are performed as a single, indivisible step, which eliminates the risk of race conditions.

2. Thread-Safe:
    - The class is thread-safe, meaning that multiple threads can read and write to the AtomicInteger instance concurrently without any external synchronization.
3. Performance:
    - AtomicInteger typically offers better performance than using synchronized blocks or methods because it uses low-level atomic machine instructions (like Compare-And-Swap) to perform operations.

Common Use Cases

- Counters: Tracking counts (like the number of tasks completed) in a multi-threaded environment.
- Shared State: Managing shared state among multiple threads without explicit locking mechanisms.
- Concurrent Algorithms: Implementing non-blocking algorithms that rely on atomic updates.

```
import java.util.concurrent.atomic.AtomicInteger;


public class AtomicIntegerExample {
   public static void main(String[] args) {
      AtomicInteger counter = new AtomicInteger(0);


      // Create multiple threads to increment the counter
      Thread[] threads = new Thread[10];


      for (int i = 0; i < threads.length; i++) {
         threads[i] = new Thread(() -> {
            for (int j = 0; j < 1000; j++) {
               // Atomically increment the counter
               counter.incrementAndGet();
            }
         });
```

```
            threads[i].start();
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Print the final value of the counter
        System.out.println("Final counter value: " + counter.get());
    }
}
```

Explanation -

- An AtomicInteger instance named counter is created, initialized to 0.
- Ten threads are created, each incrementing the counter 1,000 times.
- Each thread calls counter.incrementAndGet(), which atomically increments the counter.
- The main thread waits for all child threads to complete using join().
- Finally, the main thread prints the value of the counter. Since each thread increments it 1,000 times, the expected final value is 10,000.

programs are there in github - 1rithwik (github.com)

Building an application by using all the topics learned till date

Still has to complete this application.

Link to it -Mthree_JavaSession/LibraryManagement at main · 1rithwik/Mthree_JavaSession (github.com)

The concepts on (25-09-2024)

Starting with Data Structures

1. Array

Arrays are fundamental structures in Java that allow us to store multiple values of the same type in a single variable. They are useful for managing collections of data efficiently.

Examples done in class -

a. To find the target two sum in the array -

i. The naive approach is using two for loops:

- First I will define the integer array and the target sum.
- Then I loop through the array using the first index (i).
- Inside the outer loop, looping through the array again using a second index (j), starting from i + 1.
- For each pair of indices (i, j), checking if the sum of the elements at these indices equals the target sum.
- If a pair is found, we print the indices or the pair itself.
- Then we will continue until all pairs have been checked.
- If no pairs are found after both loops, indicate that no pairs sum to the target.

ii. Using hashmap (effective way):

- Initializing an empty HashMap to store array elements and their indices.
- Then looping through each element of the array using an index (`i`).
- For each element, we compute the complement (target - currentElement).
- If the complement exists in the HashMap, a pair is found. You can retrieve and store the indices or the pair.
- If the complement does not exist, add the current element and its index to the HashMap.
- If no pairs are found by the end of the loop, we indicate that no pairs exist.

The function which is called in main function to solve is this

```
public static void twoSum(HashMap<Integer, Integer> map, int target) {
        for (int i : map.keySet()) {
            int complement = target - i;
            if (map.containsKey(complement)) {
                System.out.println("The indices are " + map.get(i) + "
and " + map.get(complement));
```

```
            }

        }

    }
```

b. To find maximum sum of sub array in given array (Kanade's algorithm) -

- First we set two variables: maxSoFar (to store the maximum sum found) and maxEndingHere (to store the maximum sum of the subarray ending at the current index).
- Then initializing both maxSoFar and maxEndingHere to the first element of the array.
- Looping through the array starting from the second element.
- For each element, we update maxEndingHere as the maximum of the current element or the sum of maxEndingHere and the current element.
- Then we update maxSoFar to be the maximum of itself and maxEndingHere.
- we repeat above two for all elements in the array.
- After finishing the loop, maxSoFar contains the maximum sum of any subarray.

The function which is called in main method is this

```
public static void twoSum(HashMap<Integer, Integer> map, int target) {
        for (int i : map.keySet()) {
            int complement = target - i;
            if (map.containsKey(complement)) {
                System.out.println("The indices are " + map.get(i) + "
and " + map.get(complement));
            }
        }
    }
```

c. Finding two lines, which together with x-axis forms a container, such that the container contains the most water. (Trapping rain water problem).

- I initialize two pointers: i at the beginning (0) and j at the end (length - 1) of the array.
- Next, I create two variables: Maxarea to keep track of the maximum container area found so far, and area to calculate the area of the current container.
- I then enter a while loop that continues as long as the i pointer is less than the j pointer.

- Inside the loop, I calculate the area of the current container using the formula area = Math.min(arr[i], arr[j]) * (j - i), which finds the minimum height and multiplies it by the width (distance between the two pointers).
- I update Maxarea by comparing it with the current area using Maxarea = Math.max(Maxarea, area), ensuring that it holds the largest area found.
- Next, I check if the height at arr[i] is less than the height at arr[j]. If it is, I increment the i pointer to explore a potentially taller container; otherwise, I decrement the j pointer.
- I continue this process until the i pointer meets the j pointer. After the loop, I print the maximum container area found with System.out.println("The maximum container area is" + Maxarea);

The function which is called in main method is this -

```
static void maxContainer(int[] arr) {
        int i = 0, j = arr.length - 1;
        int Maxarea = 0, area = 0;
        while (i < j) {
            area = Math.min(arr[i], arr[j]) * (j - i);
            Maxarea = Math.max(Maxarea, area);
            // Maxarea=Math.max(Maxarea,Math.min(arr[i],arr[j])*(j-i));
            if (arr[i] < arr[j])
                i++;
            else
                j--;
        }
        System.out.println("The maximum container area is" + Maxarea);
    }
```

d. Finding all the water stored through out the array

This problem is to just sum all the water stored in the array, where the above problem is to find maximum of it.

- I initialize a variable sum to 0, which will hold the total amount of water trapped.
- I then enter a for loop that iterates from the beginning of the array to the second-to-last element (i < arr.length - 1).

- Inside the loop, I calculate the minimum of the current height (arr[i]) and the next height (arr[i + 1]). This represents the height of the water that can be trapped between these two points.
- I add this minimum value to sum to accumulate the total water trapped so far.
- After the loop completes, I print the total water trapped with System.out.println("The water trapped is " + sum);.

The function which is called in main method is this -

```
static void waterTrapped(int[] arr) {

        int sum = 0;

        for (int i = 0; i < arr.length - 1; i++) {

            sum = sum + Math.min(arr[i], arr[i + 1]);

        }

        System.out.println("The water trapped is " + sum);

    }
```

2. Sorting methods -

1. Selection Sort

- It divides the array into a sorted and an unsorted section. It repeatedly selects the smallest (or largest) element from the unsorted section and swaps it with the first unsorted element.
- **Time Complexity**:
    - Worst-case: $O(n^2)$
    - Best-case: $O(n^2)$
    - Average-case: $O(n^2)$
- **Space Complexity**: O(1) — sorts in place with minimal additional space.
- **Stability**: Not stable; the relative order of equal elements can change.
- **Adaptability**: Does not adapt to existing order; always performs the same number of comparisons.

2. Insertion Sort

- It builds a sorted section one element at a time by taking each element from the unsorted section and inserting it into the correct position in the sorted section.
- **Time Complexity**:
    - Worst-case: $O(n^2)$ (e.g., when the array is sorted in reverse)
    - Best-case: O(n) (e.g., when the array is already sorted)
    - Average-case: $O(n^2)$
- **Space Complexity**: O(1) — also sorts in place.
- **Stability**: Stable; maintains the relative order of equal elements.

- **Adaptability**: Highly adaptive; performs better on nearly sorted data due to fewer movements.

3. Bubble Sort

- it repeatedly steps through the array, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no swaps are needed.
- **Time Complexity**:
    - Worst-case: $O(n^2)$
    - Best-case: O(n) (with optimizations to stop early)
    - Average-case: $O(n^2)$
- **Space Complexity**: O(1) — sorts in place.
- **Stability**: Stable; maintains the relative order of equal elements.
- **Adaptability**: Can be more efficient on nearly sorted data, especially with early exit optimizations.

Comparison in between these algorithms:

Efficiency: Insertion sort generally outperforms selection and bubble sorts, especially on small or partially sorted datasets. Bubble sort can also perform better with optimizations.

Stability: Insertion sort and bubble sort are stable, while selection sort is not.

Usage: All three are primarily educational tools and are not suitable for large datasets. Insertion sort is often preferred for small or nearly sorted datasets.

Mechanism: Selection sort focuses on selecting the minimum, insertion sort focuses on building a sorted list, and bubble sort focuses on swapping adjacent elements.

The implementation can be seen in github - [1rithwik (github.com)]

Time duration for the 3 algorithms for the array dataset {32, 41, 54, 6, 23, 18, 20, 12, 9, 10}

Time taken to sort the array using selection sort: 4900 nanoseconds

Time taken to sort the array using bubble sort: 2300 nanoseconds

Time taken to sort the array using insertion sort: 2100 nanoseconds

Time duration for the 3 algorithms for the array dataset {10,9,8,7,6,5,4,3,2,1}

Time taken to sort the array using selection sort: 4900 nanoseconds

Time taken to sort the array using bubble sort: 1600 nanoseconds

Time taken to sort the array using insertion sort: 1900 nanoseconds

Time duration for the 3 algorithms for the array dataset - 12,54,65,7,34,67,23

Time taken to sort the array using selection sort: 3200 nanoseconds

Time taken to sort the array using bubble sort: 1500 nanoseconds

Time taken to sort the array using insertion sort: 2400 nanoseconds

In many cases I saw that selection sort is taking more time compared to other two. Reason could be:

1. **Insertion Sort**: Generally efficient for small and nearly sorted datasets, which explains its better performance in some cases. However, it can become slower with completely reversed data.
2. **Bubble Sort**: With optimizations (like early exit), it performs better in cases where fewer swaps are needed, which may be reflected in the first and second datasets.
3. **Selection Sort**: Has a fixed number of comparisons and swaps, which tends to result in similar performance across different datasets. It is consistently slower due to its less adaptive nature.

Nature of the Input Data:

- **Partially Sorted Data:** For the first dataset {32, 41, 54, 6, 23, 18, 20, 12, 9, 10}, there are some elements in order, allowing insertion sort to perform better as it can quickly find where to insert elements.
- **Reverse Sorted Data**: The second dataset {10, 9, 8, 7, 6, 5, 4, 3, 2, 1} is completely reversed. In this case, insertion sort has to perform the maximum number of shifts, leading to longer execution times compared to bubble sort, which can exit early if optimized.
- **Random Data:** The third dataset {12, 54, 65, 7, 34, 67, 23} is a mix. The performance of each algorithm varies based on how many elements need to be moved around.

The concepts on (26-09-2024)

1. Finding second largest element -

Brute Force Approach:

1. I began with the array of numbers that I needed to analyze to find the second largest number.
2. Sorting the Array: My first idea was to sort the array in descending order. I thought, “If I sort it, the second largest number will just be the second element.”
3. Accessing the Result: After sorting, I simply accessed the second element of the array. It felt like a quick and straightforward solution.
4. However, the array took a lot of time, especially for larger arrays, with a time complexity of O(n log n). This could be done better

Code -

The function which is called from main method

```
public static Integer findSecondLargest(int[] arr) {
    // Sort the array in descending order
    Arrays.sort(arr);
    // Check for distinct elements from the end
    for (int i = arr.length - 2; i >= 0; i--) {
        if (arr[i] != arr[arr.length - 1]) {
            return arr[i];
        }
    }
    return null; // No second largest number found
```

Optimal Approach:

1. I initialized two variables, largest and secondLargest, both set to negative infinity. This would help me track the largest and second largest numbers as I went through the array.
2. Iterating Through the Array: As I looped through each element, I compared it to largest. If I found a number greater than largest, I knew I had to update both largest and secondLargest.
3. Updating the Variables: Whenever I found a new largest number, I updated secondLargest to be the previous value of largest, and then I set largest to the current number.

4. Handling Duplicates: I made sure to check for duplicates by skipping any number that was equal to largest when determining if I should update secondLargest.
5. After looping through the array, I checked if secondLargest was still negative infinity. If it was, that meant there wasn't a valid second largest number.
6. Finally, I returned the value of secondLargest as my result. This approach time complexity is O(n).

Code -

```
public static Integer findSecondLargest(int[] arr) {

    int largest = Integer.MIN_VALUE;

    int secondLargest = Integer.MIN_VALUE;



    for (int num : arr) {

        if (num > largest) {

            secondLargest = largest;

            largest = num;

        } else if (num > secondLargest && num != largest) {

            secondLargest = num;

        }

    }



    return secondLargest != Integer.MIN_VALUE ? secondLargest : null;
// Return null if no second largest found

}
```

2. In an array we have duplicate elements, our goal is to keep the distinct elements and then remaining elements after that.

Using HashSet

1. I decided to use a HashSet because it automatically handles duplicates for me. I initialized a HashSet to store unique elements from the array.
2. Adding Elements: I looped through the original array and added each element to the HashSet. Since a HashSet only keeps unique values, any duplicates were automatically ignored.

3. Storing Distinct Elements: After populating the HashSet, I determined the number of unique elements by checking its size. I then iterated over the HashSet to transfer the distinct elements back into the original array.
4. Finally, I returned the count of distinct elements, which was the size of the HashSet, so that it that count we popularize the distinct elements in array and then printing the array.

Code -

```
static int fun1(int[] arr) {
        HashSet<Integer> set = new HashSet<>();
        for (int i = 0; i < arr.length; i++) {
            set.add(arr[i]);
        }
        int k = set.size();
        int j = 0;
        for (int a : set) {
            arr[j++] = a;
        }
        return k;
    }
int k = fun1(arr);
        for (int i = 0; i < k; i++) {
            System.out.println(arr[i] + "");
        }
```

Using two pointer technique -

1. For the second approach, I opted for a two-pointer technique. I initialized a pointer i to track the position of unique elements in the array.
2. Iterating through the Array: I started with a loop that compared the current element (arr[j]) to the last unique element (arr[i]). If they were different, it indicated a new unique element.
3. Updating the Array: When I found a new unique element, I moved it to the next position after the last unique element, effectively shifting all unique elements to the front of the array.
4. Final Output: After completing the iteration, I printed the updated array, which now contained all the distinct elements in their original order.

Code -

```
static void fun2(int[] arr) {
        int i = 0;
        for (int j = 1; j < arr.length; j++) {
            if (arr[i] != arr[j]) {
                arr[i + 1] = arr[j];
                i++;
            }
        }
        for (int z = 0; z < arr.length; z++) {
            System.out.println(arr[z] + " ");
        }
    }
```

3. Rotation of array -

a. Rotation of element by one element (Brute Force) -

1. When we rotate the array to the right by one position, the last element becomes the first, and all other elements shift one position to the right.
2. I created a new array of the same size.
3. I placed the last element of the original array at the first position of the new array.
4. I then copied the rest of the elements from the original array to the new array, starting from the second position.
5. Finally, I copied the elements from the new array back to the original array.

Code -

```
public static void rotateByOneBruteForce(int[] arr) {
    int n = arr.length;
    if (n == 0) return;
    int[] temp = new int[n];

    // Place the last element at the start
```

```
    temp[0] = arr[n - 1];
    for (int i = 1; i < n; i++) {
        temp[i] = arr[i - 1];
    }
    System.arraycopy(temp, 0, arr, 0, n);
}
```

b. Rotation of element by one element (Optimal approach) -

1. Instead of creating a new array, I realized I could simply manipulate the original array using a temporary variable.
2. Storing Last Element: I stored the last element of the array in a temporary variable.
3. I then iterated from the second last element to the start of the array, shifting each element one position to the right.
4. Atlast, I placed the last element in the first position of the array.

Code -

```
public static void rotateByOneOptimal(int[] arr) {
    int n = arr.length;
    if (n == 0) return; // Handle empty array
    int temp = arr[n - 1];

    for (int i = n - 1; i > 0; i--) {
        arr[i] = arr[i - 1];
    }

    arr[0] = temp;
}
```

c. Rotation of element by k elements (Brute force) -

1. Setting Up the Function: I created a method named rotateArr that takes two parameters: the array arr and the number of positions d to rotate.

2. Looping for Each Rotation: To achieve the left rotation, I decided to repeat the rotation process d times. Each iteration effectively moves the first element to the end of the array.
3. Storing the First Element: In each iteration, I stored the first element of the array in a temporary variable called first. This is important because I need to place this element at the end after shifting the other elements.
4. I then used a loop to shift all elements one position to the left. Starting from the first element (index 0), I assigned each element the value of the next element (i.e., arr[j] = arr[j + 1]). This continues until I reach the second-to-last element.
5. Placing the First Element at the End: After completing the shifts, I placed the previously stored first element into the last position of the array (arr[n - 1]).
6. In the main method, I initialized an array { 1, 2, 3, 4, 5, 6 } and set d to 2. After calling the rotateArr function, the output shows the array after two left rotations: 3, 4, 5, 6, 1, 2.

Code -

```
static void rotateArr(int[] arr, int d) {
    int n = arr.length;
    for (int i = 0; i < d; i++) {

        // Left rotate the array by one position
        int first = arr[0];
        for (int j = 0; j < n - 1; j++) {
            arr[j] = arr[j + 1];
        }
        arr[n - 1] = first;
    }
}

public static void main(String[] args) {
    int[] arr = { 1, 2, 3, 4, 5, 6 };
    int d = 2;

    rotateArr(arr, d);
```

```
    for (int i = 0; i < arr.length; i++)

        System.out.print(arr[i] + " ");

}
```

d. Rotation of element by k elements (Optimal solution) -

(Using Reversal Algorithm).

1. Handling Large d Values: I started by taking care of cases where d is greater than the size of the array. I used the modulus operation (d %= n) to ensure that d is within the bounds of the array length. This means if d is equal to or greater than n, the effective rotations are just d % n.
2. Reversing the Entire Array: The first step in my approach is to reverse the entire array. This step flips the order of all elements, moving the last elements to the front and the first elements to the back.
3. Reversing the First d Elements: After reversing the entire array, I then reverse the first d elements. This step restores the original order of these elements, which are now in the front.
4. Reversing the Remaining Elements: Finally, I reverse the remaining elements of the array from index d to the end. This ensures that the order of the elements in the back portion of the array is correct.
5. I implemented a helper function reverse that takes the array and the starting and ending indices. This function uses a while loop to swap elements from the start and end until they meet in the middle.
6. Example Execution: In the main method, I initialized the array { 1, 2, 3, 4, 5, 6 } and set d to 2. After calling rotateArr, the output correctly shows the array as 3, 4, 5, 6, 1, 2, reflecting two left rotations.

Code -

```
static void rotateArr(int[] arr, int d) {

    int n = arr.length;



    // Handle the case where d > size of array

    d %= n;



    reverse(arr, 0, d - 1);

    reverse(arr, d, n - 1);

    reverse(arr, 0, n - 1);

}

static void reverse(int[] arr, int start, int end) {
```

```
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

public static void main(String[] args) {
    int[] arr = { 1, 2, 3, 4, 5, 6 };
    int d = 2;

    rotateArr(arr, d);

    for (int i = 0; i < arr.length; i++)
        System.out.print(arr[i] + " ");
}
```

4. Moving all the zero elements to the last of the array -

Optimal approach

1. I start by setting up a variable to track the index of the first zero I find in the array. I initialize it to -1, which means I haven't found any zeros yet.
2. I loop through the array to search for the first zero. As soon as I find one, I record its index and break out of the loop. This helps me know where the first zero is located.
3. After that, I check if I found any zeros at all. If I didn't find a zero (meaning the index is still -1), I can conclude that there's nothing to change, and I can stop the process here.
4. If I did find a zero, I start another loop from the position right after the first zero. In this loop, I look for non-zero elements. Whenever I find a non-zero element, I swap it with the element at the first zero's index. This effectively moves the zero forward in the array.
5. Every time I swap a non-zero with the zero, I also update my index for the first zero to point to the next position. This ensures that as I continue through the array, I keep moving the zeros to the back while preserving the order of the non-zero elements.

Code -

```
int firstZeroIndex = -1;

        for (int i = 0; i < nums.length; i++) {

            if (nums[i] == 0) {

                firstZeroIndex = i;

                break;

            }

        }

        if (firstZeroIndex == -1) {

            return;

        }

        for (int i = firstZeroIndex + 1; i < nums.length; i++) {

            if (nums[i] != 0) {

                int temp = nums[i];

                nums[i] = nums[firstZeroIndex];

                nums[firstZeroIndex] = temp;

                firstZeroIndex++;

            }

        }
```

Githhub link - [1rithwik (github.com)]

The concepts on (27-09-2024)

1. To check if given pair with target sum exists on array -

(It is two sum array problem)

1. First, I sorted the array. Sorting is crucial because it allows us to systematically explore possible pairs using two pointers, rather than checking every combination, which would be less efficient.
2. I initialized two pointers: one (`l`) at the start of the array and the other (`r`) at the end. This setup enables us to consider the smallest and largest values in the sorted array, facilitating the search for the target sum.
3. **Iterative Comparison:** I entered a loop that continues until the left pointer is less than the right pointer. Within the loop:
    - I calculated the sum of the values at the two pointers.
    - If the sum equaled the target, I had found my solution and noted the indices.
    - If the sum was greater than the target, it indicated that the right pointer's value was too large, so I moved the right pointer one step to the left.
    - Conversely, if the sum was less than the target, it suggested that the left pointer's value was too small, prompting me to move the left pointer one step to the right.
4. **Ending the Search:** This process continued until the pointers converged. If I did not find a pair that matched the target sum, it meant that no such indices existed in the array.

- **Time Complexity:** The most significant aspect of this approach is the sorting step, which gives it a time complexity of O(nlogn).
- **Space Complexity:** Since I used a two-pointer approach, the space complexity is O(1), assuming the sorting is done in place.

Code -

the function fun2() called in main function

```
static void fun2(int[] arr, int target) {

        Arrays.sort(arr);

        int l = 0, r = arr.length - 1;

        while (l < r) {

            if (arr[l] + arr[r] == target) {
```

```
                System.out.println("The indices are " + l + " and " +
r);

            } else if (arr[l] + arr[r] > target) {

                r--;

            } else {

                l++;

            }

        }

    }
```

2. Stock buy and sell problem

Brute force approach

The brute force method examines every possible pair of buy and sell days to find the maximum profit. For each day, we consider all subsequent days to check if selling on that day would yield a profit.

Implementation:

- We can use two nested loops: the outer loop to select a buying day and the inner loop to select a selling day.
- For each pair of days, we calculate the profit (difference between selling price and buying price).
- We keep track of the maximum profit encountered.

Time Complexity: This approach has a time complexity of O(n2) since it requires checking every combination of buy and sell days.

Optimal solution:

1. We start by initializing two variables:
2. currmin to keep track of the minimum stock price observed so far.
3. res to store the maximum profit.
4. We iterate through the array, starting from the second element.
5. For each price, we update currmin to be the minimum of itself and the current price.
6. We then calculate the profit if we were to sell on the current day by subtracting currmin from the current price. If this profit is greater than res, we update res.
7. Final Output: After iterating through the entire array, res will hold the maximum profit achievable.
8. Time Complexity: This optimal approach runs in O(n) time, as it requires just a single pass through the array.

Code -

```
public class StockBuySell {

    static void fun1(int[] arr) {

        int currmin = arr[0];

        int res = 0;

        for (int i = 1; i < arr.length; i++) {

            currmin = Math.min(currmin, arr[i]);

            res = Math.max(res, arr[i] - currmin);

        }

        System.out.println("The maximum profit is " + res);

    }

    public static void main(String[] args) {


        System.out.println("Enter the size of the array");

        int[] arr = { 7, 1, 2, 6, 4, 5 };

        fun1(arr);

    }

}
```

3. Return sub arrays of an array which are having the target sum

Brute Force approach

A subarray is defined as a contiguous portion of the array. By checking each subarray, we can determine if its sum matches the target.

Implementation:

- I used two nested loops: the outer loop iterates through the starting index of the subarray, while the inner loop extends the subarray to check all possible ending indices.
- For each subarray defined by these indices, I calculated the sum. If this sum equals the target, I recorded the subarray.

Time Complexity: This brute force approach has a time complexity of O(n3) in the worst case because:

- The outer loop runs nnn times.
- The inner loop also runs nnn times.
- The sum calculation for each subarray can take up to O(n)O(n)O(n) in the worst case.

Optimal Approach -

By using a hash map, we can check if there is a previous cumulative sum that, when subtracted from the current cumulative sum, equals the target. This allows us to identify valid subarrays more efficiently.

Implementation Steps:

- I initialized a hash map to store cumulative sums and their frequencies.
- As I iterated through the array, I maintained a variable for the cumulative sum.
- For each element, I updated the cumulative sum and checked if the difference between the cumulative sum and the target exists in the hash map. If it does, it indicates that a subarray summing to the target exists.
- I also stored the cumulative sum in the hash map after checking, so that subsequent iterations can utilize this information.

Time Complexity: This optimal approach runs in O(n) time because we make a single pass through the array and perform constant time operations (insert and lookup) with the hash map.

Code -

```
public class SubArraySum {

    static void findSubarrays(int[] arr, int target) {

        HashMap<Integer, ArrayList<Integer>> map = new HashMap<>();

        ArrayList<int[]> result = new ArrayList<>();

        int currSum = 0;


        for (int i = 0; i < arr.length; i++) {

            currSum += arr[i];


            if (currSum == target) {

                result.add(new int[] { 0, i });

            }
```

```
            if (map.containsKey(currSum - target)) {
                for (int startIndex : map.get(currSum - target)) {
                    result.add(new int[] { startIndex + 1, i });
                }
            }

            map.putIfAbsent(currSum, new ArrayList<>());
            map.get(currSum).add(i);
        }
        for (int[] subarray : result) {
            System.out.println("Subarray found from index " +
subarray[0] + " to " + subarray[1]);
        }
    }

    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 7, 5 };
        int target = 12;
        findSubarrays(arr, target);
    }
}
```

4. Spring Framework

a. Introduction

Spring is a powerful, feature-rich framework for building enterprise-level Java applications. It provides comprehensive infrastructure support, enabling developers to focus on their application's core functionality without worrying about the underlying complexities.

Key Features:

1. **Dependency Injection (DI):** Dependency Injection is a key aspect of IoC in Spring. It allows the framework to manage dependencies between objects, automatically injecting required dependencies at runtime. This leads to more modular and maintainable code, as classes do not need to manage their own dependencies.
2. **Inversion of Control (IoC):** At the heart of Spring is the IoC container, which manages the instantiation and lifecycle of application objects (beans). By inverting the control flow, Spring promotes loose coupling and easier testability.
3. **Aspect-Oriented Programming (AOP):** Spring supports AOP, allowing developers to define cross-cutting concerns (like logging, security, and transaction management) separately from the business logic. This modular approach enhances code maintainability.

Advantages of Using Spring:

- **Modularity:** Spring's modular architecture allows developers to use only the parts they need, making it lightweight and flexible.
- **Testability:** The IoC container supports dependency injection, which simplifies testing and promotes better code design.
- **Community Support:** Spring has a large and active community, providing extensive documentation, tutorials, and resources, which facilitates learning and problem-solving.

Overall, Spring is a comprehensive framework that offers a robust platform for building modern Java applications. Its focus on simplicity, testability, and integration makes it a popular choice among developers for creating scalable and maintainable software solutions.

Dependency injection -

Dependency Injection is a design pattern and a key feature of the Spring framework that allows for the decoupling of object creation and usage. In simpler terms, it enables an object to receive its dependencies from an external source rather than creating them itself. This promotes better code organization, easier testing, and improved maintainability.

1. **Types of Dependency Injection:**

- **Constructor Injection:** Dependencies are provided through the class constructor. This method ensures that the object is always created with its required dependencies.
- **Setter Injection:** Dependencies are provided through setter methods after the object is constructed. This allows for more flexibility, as dependencies can be changed at runtime.
- **Interface Injection:** Although less common, this involves an interface that defines a method for injecting the dependency.

2. **Benefits of Dependency Injection:**
   - **Easier Testing:** DI facilitates unit testing, as dependencies can be mocked or stubbed during tests. This isolates the unit of work, making it simpler to validate behavior.
   - **Improved Maintainability:** Changes to dependency implementations require minimal changes to the dependent classes, leading to better maintainability and scalability.
3. **Spring and Dependency Injection:** In Spring, DI is achieved through the use of the IoC container, which can be configured using XML, annotations, or Java-based configuration. The container manages the lifecycle and dependencies of beans (objects) defined in the application.

Example -

```
@Component
class Service {
   public void execute() {
      System.out.println("Service executed");
   }
}

@Component
class Client {
   private final Service service;

   @Autowired // Constructor Injection
   public Client(Service service) {
      this.service = service;
   }

   public void perform() {
      service.execute();
   }
}
```

In this example:

- The Service class is a dependency for the Client class.
- The @Autowired annotation tells Spring to inject an instance of Service into Client when it is created.

Inversion of control -

Inversion of Control (IoC) is a fundamental principle in software design that transfers the control of object creation and management from the application code to a container or framework.

Key Concepts:

1. **Decoupling:** The primary goal of IoC is to decouple the execution of a task from its implementation. This means that the application components do not need to know how their dependencies are created or managed, leading to a more modular architecture.
2. **Dependency Injection as IoC:** Dependency Injection (DI) is one of the most common forms of IoC. In DI, dependencies are provided to a class rather than the class creating them itself. This allows for better separation of concerns and facilitates easier testing and maintenance.
3. **Types of IoC:**
    - **Constructor Injection:** Dependencies are provided through a class constructor.
    - **Setter Injection:** Dependencies are provided through setter methods after the object is created.
    - **Interface Injection:** Dependencies are injected through an interface method (less common in practice).
4. **Benefits of IoC:**
    - **Improved Testability:** Since components are loosely coupled, it's easier to mock dependencies during unit testing, leading to more effective tests.
    - **Flexible and Maintainable Code:** Changes to one component do not require changes to others, making it easier to update or replace components.
    - **Centralized Configuration:** IoC allows for centralized management of dependencies, which simplifies the configuration of complex applications.

Example -

```
@Component
class GreetingService {
```

```
    public void greet() {
        System.out.println("Hello, World!");
    }
}

@Component
class GreetingApp {
    private final GreetingService greetingService;

    public GreetingApp(GreetingService greetingService) {
        this.greetingService = greetingService; // Dependency is injected
    }

    public void run() {
        greetingService.greet();
    }
}

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(Main.class);
        GreetingApp app = context.getBean(GreetingApp.class);
        app.run(); // Output: Hello, World!
    }
}
```

In this example:

- The GreetingService is a dependency for the GreetingApp class.
- The Spring container is responsible for creating instances of these classes and injecting the dependencies.

Aspect oriented programming -

Aspect-Oriented Programming (AOP) is a programming paradigm that complements object-oriented programming (OOP) by allowing the separation of cross-cutting concerns. In the context of the Spring framework, AOP provides a way to modularize concerns that affect multiple parts of an application, such as logging, security, and transaction management.

Key Concepts:

1. **Cross-Cutting Concerns:** Cross-cutting concerns are functionalities that affect multiple modules of an application but do not belong to the main business logic. Examples include logging, error handling, security checks, and transaction management. AOP allows these concerns to be handled separately from the core business logic.

2. **Aspects, Join Points, and Advice:**
    - **Aspect:** An aspect is a module that encapsulates a cross-cutting concern. It combines the definition of the concern and its behavior.
    - **Join Point:** A join point is a specific point in the execution of the program, such as a method execution, where an aspect can be applied.
    - **Advice:** Advice is the action taken by an aspect at a particular join point.

    - Types of advice include:
        - **Before:** Executed before the join point.
        - **After:** Executed after the join point, regardless of the outcome.
        - **After Returning:** Executed after the join point only if the method returns successfully.
        - **After Throwing:** Executed after the join point if an exception is thrown.
        - **Around:** Wraps the join point, allowing for custom behavior before and after the execution.
3. **Benefits of AOP:**
    - **Modularity:** AOP enhances modularity by separating concerns, leading to cleaner and more maintainable code.
    - **Reusability:** Aspects can be reused across different parts of the application, reducing code duplication.
    - **Easier Maintenance:** Changes to cross-cutting concerns can be made in one place without affecting the core business logic.

Example -

```
@Aspect
@Component
class LoggingAspect {
   @Before("execution(* com.example.service.*.*(..))")
   public void logBeforeMethod() {
      System.out.println("Method is about to be executed");
   }
}

@Component
class UserService {
   public void createUser() {
      System.out.println("User created");
   }
}

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

```
        ApplicationContext context = new AnnotationConfigApplicationContext(Main.class);
        UserService userService = context.getBean(UserService.class);
        userService.createUser(); // Logs: Method is about to be executed
                                  // Output: User created
    }
}
```

In this example:

- The LoggingAspect defines a logging advice that is executed before any method in the UserService class.
- The @Before annotation indicates that the logBeforeMethod advice should run before the execution of methods matching the specified pointcut expression.

Group project of building an application using -

Core java, Database(MySQL), OOPs concepts, MultiThreading and cache application

Documentation - Group Project Agile SDLC Document on 27-09-2024 - Google Docs

Project link - itsgits01/FinancialNewsAggregator (github.com)

Project done - only added 2 use cases

The concepts on (30-09-2024)

Advanced cache - [Advance cache - Google Docs]

Binary Search -

Binary search is an efficient algorithm for finding a specific element in a sorted list. The key idea is to repeatedly divide the search interval in half.

1. **Start with the entire list**: Identify the lowest and highest indices of the list.
2. **Find the middle element**: Calculate the index of the middle element.
3. **Compare**:
    - If the middle element is equal to the target value, you've found the element.
    - If the target value is less than the middle element, narrow your search to the left half of the list.
    - If the target value is greater, narrow your search to the right half.
4. **Repeat**: Continue this process until the target value is found or the interval is empty.

The algorithm has a time complexity of O(log n), making it much faster than a linear search for large lists. However, it requires the list to be sorted beforehand.

Code -

Both in iterative and recursive approach

```
package sep_30;


// Java implementation of iterative Binary Search
class BinrySrch {
    // Returns index of x if it is present in arr[l....r], else return
-1
    int fun1(int arr[], int l, int r, int x) {
        while (l <= r) {
            int mid = (l + r) / 2;
            if (arr[mid] == x) {
                return mid;
            } else if (arr[mid] > x) {
                r = mid - 1;
```

```
            } else {
                l = mid + 1;
            }
        }
        return -1;
    }

    int fun2(int arr[], int l, int r, int x) {
        if (l <= r) {
            int mid = l + (r - l) / 2;
            if (arr[mid] == x) {
                return mid;
            } else if (arr[mid] > x) {
                return fun2(arr, l, mid - 1, x);
            } else {
                return fun2(arr, mid + 1, r, x);
            }
        }
        return -1;
    }

    public static void main(String args[]) {
        BinrySrch ob = new BinrySrch();

        int arr[] = { 2, 3, 4, 8, 20 };
        int n = arr.length;
        int x = 8;
        // int result = ob.fun1(arr, 0, n - 1, x);
        int result2 = ob.fun2(arr, 0, n - 1, x);
```

```
        if (result2 == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "
                    + result2);
    }
}
```

The concepts on (1-10-2024)

1. Merge sort -

Merge Sort is a divide-and-conquer sorting algorithm that divides a list into smaller sublists, sorts those sublists, and then merges them back together in sorted order. It is particularly efficient for large datasets and is stable, meaning that it preserves the order of equal elements.

Steps of Merge Sort

1. **Divide**: Split the unsorted list into n sublists, each containing one element (or empty).
2. **Conquer**: Repeatedly merge sublists to produce new sorted sublists:
    - Compare the first elements of each sublist.
    - Select the smaller element and append it to a new list.
    - Repeat until all elements from the sublists are merged.
3. **Combine**: Continue merging until there is only one sublist remaining, which is the sorted list.

Complexity:

- Time: O(n log n) for all cases (best, average, worst).
- Space: O(n) due to the temporary arrays used for merging.

Use Cases: Efficient for large datasets, external sorting, linked lists.

Code -

```java
package oct_1;


import java.util.Arrays;


public class MergeSrt {

    static void merge(int arr[], int l, int m, int r) {

        // Find sizes of two subarrays to be merged

        int n1 = m - l + 1;

        int n2 = r - m;


        int L[] = Arrays.copyOfRange(arr, l, m + 1);

        int R[] = Arrays.copyOfRange(arr, m + 1, r + 1);
```

```
        int i = 0, j = 0;


        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }


        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }


    static void sort(int arr[], int l, int r) {
        if (l < r) {
```

```
            int m = l + (r - l) / 2;


            sort(arr, l, m);

            sort(arr, m + 1, r);


            merge(arr, l, m, r);

        }

    }


    public static void main(String args[]) {

        int arr[] = { 21, 34, 1, 8, 11, 3, 56 };

        sort(arr, 0, arr.length - 1);


        System.out.println("\nSorted array is");

        for (int i = 0; i < arr.length; i++) {

            System.out.print(arr[i] + " ");

        }

    }

}
```

2. Quick sort -

Quick Sort is a highly efficient sorting algorithm that follows the divide-and-conquer principle. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays: those less than the pivot and those greater than the pivot. The sub-arrays are then sorted recursively.

Steps

1. Select an element from the array as the pivot (commonly the first, last, or a random element).

2. Partition: Rearrange the array so that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right.
3. Recursively Sort: Recursively apply the above steps to the left and right sub-arrays.
4. Once the sub-arrays are sorted, combine them with the pivot to form a sorted array.

Code -

```
package oct_1;


public class Quicksrt {

    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);

        }

    }


    public static int partition(int[] arr, int low, int high) {

        int pivot = arr[low];

        int i = low, j = high;

        while (i < j) {

            while (arr[i] <= pivot && i < high) {

                i++;

            }

            while (arr[j] >= pivot && j > low) {

                j--;

            }

            if (i < j) {

                int temp = arr[i];

                arr[i] = arr[j];
```

```
                arr[j] = temp;

            }

        }

        int temp = arr[low];

        arr[low] = arr[j];

        arr[j] = temp;

        return j;

    }


    public static void main(String[] args) {

        int[] arr = { 10, 7, 8, 9, 1, 5 };

        quicksort(arr, 0, arr.length - 1);

        System.out.println("Sorted array: ");

        for (int i = 0; i < arr.length; i++) {

            System.err.println(arr[i] + " ");

        }

    }


}
```

3. Binary Tree -

A binary tree is a data structure in which each node has at most two children, referred to as the left child and the right child. This structure allows for efficient data organization, searching, and traversal.

Key Characteristics

- **Nodes**: Each node contains a value and pointers to its left and right children.
- **Height**: The height of a binary tree is the length of the longest path from the root to a leaf node.
- **Types**:

- **Full Binary Tree**: Every node other than the leaves has two children.
- **Complete Binary Tree**: All levels are fully filled except possibly for the last level.
- **Perfect Binary Tree**: All internal nodes have two children and all leaf nodes are at the same level.
- **Binary Search Tree (BST)**: A binary tree where the left child is less than the parent node, and the right child is greater.

Tree traversals - inorder, pre-order, post-order

Implemetation of node in tree -

we have the value field and other two fields for linking of left and right sub-tree.

```
class Node {

    int key;

    Node left, right;


    public Node(int key) {

        this.key = key;

        left = right = null;

    }

}
```

Insertion for Binary tree -

Create a new node with the desired value.

- If the tree is empty, the new node becomes the root.
- If the tree is not empty, use a level-order traversal (typically using a queue) to find the first available position (null child) for the new node.
- Once the appropriate position is found, insert the new node as either the left or right child of the parent node.

Code -

```
public void insert(int key) {
```

```
        root = insertRec(root, key);
    }

    public Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }
        if (key < root.key)
            root.left = insertRec(root.left, key);
        else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        return root;
    }
```

Searching in Binary tree -

We Begin the search at the root node of the BST.

Compare Values:

- If the target value is equal to the current node's value, the search is successful.
- If the target value is less than the current node's value, move to the left child.
- If the target value is greater than the current node's value, move to the right child.

Repeat: Continue the comparison and traversal until:

- The target value is found.
- A null reference is reached, indicating that the value is not in the tree.

Code -

```
public boolean search(int key) {
```

```
        return searchRec(root, key);
    }

    public boolean searchRec(Node root, int key) {
        if (root == null) {
            return false;
        }
        if (root.key == key) {
            return true;
        }
        if (key > root.key) {
            return searchRec(root.right, key);
        } else {
            return searchRec(root.left, key);
        }
    }
```

DFS search -

Depth-First Search (DFS) is a method for exploring a tree or graph by going as deep as possible along each branch before backtracking. In a tree, there are three common DFS traversal methods:

1. Inorder Traversal - (Left, Root, Right)
2. Preorder Traversal - (Root, Left, Right)
3. Postorder Traversal - (Left, Right, Root)

Code -

```
public static void dfs(Node root) {
        if (root == null) {
            return;
```

```
        }

        System.out.println(root.key);

        dfs(root.left);

        dfs(root.right);

    }
```

Deleting of a node -

Deleting a node from a BST involves three main scenarios depending on the node's children:

1. **Node with No Children (Leaf Node)**:
    - Simply remove the node from the tree.
2. **Node with One Child**:
    - Remove the node and connect its parent directly to its child.
3. **Node with Two Children**:
    - Find the node's in-order predecessor (the maximum value in the left subtree) or in-order successor (the minimum value in the right subtree).
    - Replace the value of the node to be deleted with the value of the predecessor or successor.
    - Delete the predecessor or successor node, which will be a simpler case (either a leaf or with one child).

Steps for Deleting:

1. We wegin by locating the node to be deleted using the search method.
2. Handling the Deletion:
    - If it’s a leaf node, simply remove it.
    - If it has one child, link its parent directly to its child.
    - If it has two children:
        - Find the in-order predecessor or successor.
        - Replace the node’s value with that of the predecessor or successor.
        - Delete the predecessor or successor node.

Code -

```
public void delete(int key) {

        root = deleteRec(root, key);

        System.out.println("Node deleted");
```

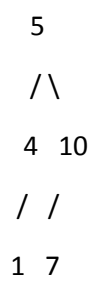
```
    }


    public Node deleteRec(Node root, int key) {
        if (root == null) {
            return root;
        }
        if (key < root.key) {
            root.left = deleteRec(root.left, key);
        } else if (key > root.key) {
            root.right = deleteRec(root.right, key);
        } else {
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }
            root.key = maxValue(root.left);
            root.left = deleteRec(root.left, root.key);
        }
        return root;
    }
```

Here we find the maximum value of the left subtree and then replace it with the deleting node value.

Explanation:

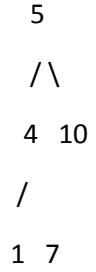
- First, I started at the root of the tree, which has a value of 5.
- Since 3 is less than 5, I moved to the left child, where I found 3. This was the node I wanted to delete.

- I noticed that the node 3 has two children: 1 on the left and 4 on the right. Since it has two children, I knew I needed to take a specific approach to delete it while keeping the tree balanced.
- To maintain the binary search tree properties, I needed to find the in-order successor of 3. This is the smallest value in its right subtree.
- The right child of 3 is 4, which doesn't have a left child, making 4 the in-order successor.
- I replaced the value of node 3 with the in-order successor 4. This step keeps the structure intact while allowing me to remove 3.
- After this replacement, the tree looked like this:



- Now that 3 has been replaced by 4, I needed to delete the original 4 node. Since it had no left child, I simply removed it.
- This adjustment ensured that the left child of 5 now points directly to 1.

Final Tree Structure



The concepts on (3-10-2024)

1. Problem to print the string in the reverse order.

Approach 1: Using array

- We start by converting the input string into a character array using s.toCharArray(). This allows us to easily access each character in the string.
- We create two strings: ans, which will hold the final reversed output, and currString, which will temporarily store characters of the current word being processed.
- We loop through the character array from the last character to the first. This helps us to build the output in reverse order.
- Inside the loop, we check if the current character is a space (' '):
    - If it is a space, we append the currString (which contains the last processed word) to ans, followed by a space, and then reset currString to start collecting the next word.
    - If it's not a space, we prepend the character to currString to build the current word.
- After the loop, there might still be one last word in currString (in case there's no trailing space), so we append it to ans.
- Finally, we print the result stored in ans, which now contains the words in reverse order.

Code -

```
static void fun1(String s) {

        char[] arr = s.toCharArray();

        String ans = "", currString = "";

        for (int i = arr.length - 1; i >= 0; i--) {

            if (arr[i] == ' ') {

                ans = ans + currString + " ";

                currString = "";

            } else {

                currString = arr[i] + currString;

            }

        }

        ans = ans + currString;

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

```
    }
```

Approach 2: Using stack

- I started by splitting the input string "before words" into an array of words using s.split(" "). This gave me an array: ["before", "words"].
- I created a stack (Stack<String> st) to hold the words. The stack will allow me to reverse the order because it follows the Last In, First Out (LIFO) principle.
- I iterated over each word in the array s1 using a for-each loop. For each word (str1), I pushed it onto the stack using st.push(str1).
- After pushing all the words onto the stack, I initialized an empty string ans to hold the final reversed output.
- I then entered a while loop that continues as long as the stack is not empty. Inside the loop, I popped the top word from the stack using st.pop() and appended it to ans, followed by a space.
- Finally, I printed the result stored in ans, which contains the words in reverse order.

Code -

```
static void fun2(String s) {

        Stack<String> st = new Stack<>();

        String[] s1 = s.split(" ");

        for (String str1 : s1) {

            st.push(str1);

        }

        String ans = "";

        while (!st.isEmpty()) {

            ans = ans + st.pop() + " ";

        }

        System.out.println(ans);

    }
```

2. Dynamic programming

Dynamic programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations.

Key Characteristics:

1. **Overlapping Subproblems**: DP is used when the problem can be divided into subproblems that are reused multiple times. For example, calculating Fibonacci numbers involves recalculating values for the same inputs.
2. **Optimal Substructure**: A problem exhibits optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. This means that solving smaller instances of the problem helps in solving the overall problem.

Problem - Coin Change Problem

Approach: Different from the approach explained in class

Given an array of coin denominations and a total amount, I needed to find out how many different combinations of these coins could sum up to that amount.

I created an array arr of size n + 1, where n is the target amount. This array is used to store the number of ways to achieve each amount from 0 to n. I initialized arr[0] to 1, representing one way to make the amount 0 (by using no coins).
I implemented a nested loop:

- The outer loop iterates through each coin denomination.
- The inner loop iterates through all amounts from 0 to n. For each amount j, if the current coin denomination is less than or equal to j, I updated the number of ways to make the amount j by adding the number of ways to make the amount j - coins[i]. This captures the essence of using the current coin denomination in forming the total.

After filling the DP table, the value at arr[n] provides the total number of ways to make the amount n using the given coins.
I tested my implementation with the coin denominations {2, 3, 5} and a target amount of 12.
Running the function successfully computed the result.

Code -

```
static void fun1(int[] coins, int n) {
        int[] arr = new int[n + 1];
        arr[0] = 1;
        for (int i = 0; i < coins.length; i++) {
            for (int j = 0; j < arr.length; j++) {
```

```
                if (coins[i] <= j)
                    arr[j] += arr[j - coins[i]];
            }
        }
        System.out.println(arr[n]);
    }

    public static void main(String[] args) {
        int[] coins = { 2, 3, 5 };
        int n = 12;
        fun1(coins, n);
    }
```

3. To solve if the given two string are anagrams to each other.

An anagram consists of the same characters arranged in a different order. For example, "listen" and "silent" are anagrams, while "hello" and "bye" are not.

Approach:

1. Given two strings, I needed to check if they contain the same characters in the same frequencies, regardless of their order.
2. To start, I compared the lengths of the two strings. If they were not equal, I could immediately conclude that they were not anagrams, as different lengths imply different character counts.
3. I created an array arr of size 26 (for each letter of the English alphabet). This array would be used to count the occurrences of each character in the first string.
    - I iterated through the characters of s1, incrementing the corresponding index in arr based on the character's ASCII value.
4. After counting the characters from the first string, I iterated through the characters of s2, decrementing the corresponding indices in arr. This allowed me to check how many characters from s1 matched those in s2.
5. I then checked the contents of the arr. If any index did not equal zero, it indicated that the strings were not anagrams because there was a mismatch in character counts.
6. If all indices were zero, I concluded that the two strings were anagrams and printed the result.

Code -

```
static void fun1(String s1, String s2) {
        char[] arr = new char[26];
        if (s1.length() != s2.length()) {
            System.out.println("Not anagram");
            return;
        }
        for (int i = 0; i < s1.length(); i++) {
            arr[s1.charAt(i) - 'a']++;
        }
        for (int i = 0; i < s1.length(); i++) {
            arr[s1.charAt(i) - 'a']--;
        }
        for (int i = 0; i < s1.length(); i++) {
            if (arr[i] != 0)
                System.out.println("Not anagram");
        }
        System.out.println("it is anagram");
    }

    public static void main(String[] args) {
        String s1 = "bye";
        String s2 = "hello";
        fun1(s1, s2);
    }
```

4. Lambda functions -

Lambda functions, introduced in Java 8, allow developers to implement functional programming principles in Java. They enable the creation of anonymous functions (or expressions) that can be treated as first-class citizens, leading to more concise and flexible code.

Key Features

1. Syntax:
   Lambda expressions provide a concise way to represent single-method interfaces. The basic structure includes parameters, an arrow (->), and an expression or block of code.
2. Functional Interfaces:
   A lambda expression can be used where a functional interface is expected. A functional interface contains only one abstract method, making it a target for lambda expressions. Common examples include Runnable and Comparator.
3. Benefits:
   - Conciseness: Reduces boilerplate code by eliminating the need to create entire classes for simple implementations.
   - Improved Readability: Makes the code clearer and easier to understand.
   - Supports Functional Programming: Facilitates treating actions as first-class citizens, allowing functions to be passed as arguments or returned from other functions.
4. Use Cases:
   - Lambda expressions are often used with the Java Collections Framework, particularly for operations like iteration, filtering, and mapping.
   - They simplify event handling and callback mechanisms, enhancing the readability of the code.

Used in application of calculator. Code given below after the concept of Streams

5. Streams

Streams in Java, introduced in Java 8, provide a powerful way to process sequences of elements (like collections) in a functional style. They enable developers to perform operations such as filtering, mapping, and reducing in a more declarative manner.

Key Features -

1. A stream is not a data structure but a sequence of elements that can be processed. It allows for functional-style operations on collections and can be generated from various data sources, including collections, arrays, or I/O channels.
2. Pipeline:
   Stream operations are typically executed in a pipeline, consisting of:
   - Source: The initial data source (e.g., a collection).

- Intermediate Operations: These are operations that transform a stream into another stream (e.g., filter, map, sorted). They are lazy, meaning they do not process the elements until a terminal operation is invoked.
- Terminal Operations: These operations produce a result or a side effect (e.g., collect, forEach, count). They trigger the processing of the stream.

3. Common Operations:
   - Filtering: Selecting elements based on certain criteria.
   - Mapping: Transforming each element in the stream to another form.
   - Sorting: Arranging elements in a specific order.
   - Reduction: Combining elements to produce a single result (e.g., summing values).
4. Collecting Results:
   The collect method is used to gather the results of stream processing into a collection, like a List or Set. The Collectors utility class provides various static methods to facilitate this.
5. Performance Considerations:
   While parallel streams can significantly improve performance for large datasets, they also introduce overhead. It's important to analyze the workload and environment to determine when to use parallel processing.

Code for both lambda and Streams -

```
public class LambdaAndStream {

    static void fun1() {

        int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8 };

        Arrays.stream(arr).forEach(System.out::println);

        IntStream a1 = Arrays.stream(arr);

        a1.filter(x -> x % 2 == 0).map(x -> x *
x).forEach(System.out::println);



        int[] a2 = { 4, 6, 1, 9, 3, 10 };

        List<Integer> l1 =
Arrays.stream(a2).boxed().sorted().collect(Collectors.toList());

        for (int i : l1) {

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

```
        }


    }


    public static void main(String[] args) {
        Runnable t1 = () -> System.out.println("Welcome");
        t1.run();


        BiFunction<Double, Double, Double> add = (a, b) -> a + b;
        BiFunction<Double, Double, Double> sub = (a, b) -> a - b;
        BiFunction<Double, Double, Double> mul = (a, b) -> a * b;
        BiFunction<Double, Double, Double> div = (a, b) -> a / b;


        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter first number: ");
        double n1 = scanner.nextDouble();


        System.out.println("Enter second number: ");
        double n2 = scanner.nextDouble();


        System.out.println("Choose operation: +, -, *, /");
        char operator = scanner.next().charAt(0);


        scanner.close();
        double res = 0;
```

```
        switch (operator) {
            case '+':
                add.apply(n1, n2);
                break;
            case '-':
                res = sub.apply(n1, n2);
                break;
            case '*':
                res = mul.apply(n1, n2);
                break;
            case '/':
                res = div.apply(n1, n2);
                break;
            default:
                System.out.println("Invalid operator!");
                return;
        }
        System.out.println(res);
        fun1();
    }
}
```

Explanation:

1. Functional Interface Definition:
   I began by defining a functional interface, which is an interface with a single abstract method. This serves as the foundation for using lambda expressions.

2. Lambda Expression Creation:
   I created a lambda expression to provide an implementation for the functional interface. This involved using the syntax (parameters) -> expression:
   - The parameters are defined within parentheses.
   - The arrow (->) separates the parameters from the body of the expression, which contains the implementation logic.
3. Usage in Collections:
   I applied the lambda expression in the context of Java Collections, demonstrating its effectiveness:
   - I used a lambda expression to iterate over a collection and perform actions on each element, showcasing how it simplifies the code compared to traditional loops.

Explanation:

Creating a Stream:
I initiated a stream from a collection or array, which allows for a sequence of operations to be performed on the data. This serves as the entry point for stream processing.

I demonstrated several intermediate operations, such as:

- Filter: Used to select elements that meet specific criteria. This operation produces a new stream containing only the filtered elements.
- Map: Transformed each element in the stream by applying a function, which modifies the data as required.
- Sorted: Ordered the elements in the stream, enhancing the organization of data for subsequent operations.

After performing the intermediate operations, I applied terminal operations to produce a result or side effect:

- Collect: Gathered the processed data into a collection, such as a List or Set, allowing for further use or display.
- ForEach: Executed an action on each element of the stream, demonstrating how to apply functions directly to the elements.

Github link - https://github.com/1rithwik

The concepts on (4-10-2024)

Problems done in class:

1. Finding the longest palidrome substring in a given string

2. Implementing Dictionary

3. To find the unique elements in the array

4. To search element in the row-wise sorted matrix.

5. finding all the leader elements in an array.

Problems done in break-out room:

1. Climbing stairs problem

2. Minimum sum path

3. Jump Game

1. Climbing stairs problem -

The "Climbing Stairs" problem requires to determine the number of distinct ways to reach the top of a staircase with n steps, where at each step you can either climb 1 step or 2 steps.

Step 1:

- For each step i, the number of ways to reach that step can be derived from the previous two steps:
    - From step i-1 (1 step up)
    - From step i-2 (2 steps up)

Thus, I can establish the relation:

$dp[i]=dp[i-1]+dp[i-2]$

Step 2: Base Cases

- dp[0] = 1: There's one way to stay at the ground (doing nothing).
- dp[1] = 1: There's one way to reach the first step (one single step).

Step 3: Dynamic Programming Array

- Creating an array dp of size n + 1 to store the number of ways to reach each step from 0 to n.

Complexity Analysis

- Time Complexity: O(n) because we calculate the number of ways for each step once.

- Space Complexity: O(n) due to the dp array. However, we could optimize it to O(1) by storing only the last two computed values.

Optimized Solution:

If we observe the output we can clearly identify that the answers are in fibonacci numbers

for 1 it is 1

for 2 it is 2

for 3 it is 3

for 4 it is 5

for 5 it is 8 so on….

so we can just print the fibonacci number the nth value in the series.

Code-

```
public static int climbStairsOptimized(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;

    int a = 1; // dp[0]
    int b = 1; // dp[1]

    for (int i = 2; i <= n; i++) {
        int c = a + b; // Current number of ways
        a = b; // Move to the next step
        b = c; // Update for the next iteration
    }

    return b; // Number of ways to reach the nth step
}
```

2. Minimum sum path

Problem Overview

The "Minimum Path Sum" problem requires finding a path from the top-left corner to the bottom-right corner of a grid filled with non-negative integers in a matrix. The path can only move either down or right, and the goal is to minimize the sum of the values along the path.

Code -

```
class Solution {

    public int minPathSum(int[][] grid) {

        for(int j=1;j<grid[0].length;j++)

        {

            grid[0][j]+=grid[0][j-1];

        }

        for(int i=1;i<grid.length;i++)

        {

            grid[i][0]+=grid[i-1][0];

        }

        for(int i=1;i<grid.length;i++)

        {

            for(int j=1;j<grid[0].length;j++)

            {


grid[i][j]=Math.min(grid[i-1][j]+grid[i][j],grid[i][j-1]+grid[i][j]);

            }

        }

        return grid[grid.length-1][grid[0].length-1];

    }

}
```

Explanation -

Step 1:

- To reach any cell (i, j), you can either come from the left cell (i, j-1) or from the top cell (i-1, j).
- The value of each cell in the grid represents the cost to enter that cell.

Step 2: Dynamic Programming Approach

1. Use the original grid itself to store the minimum path sums, modifying it in place.
2. Base Cases:
    - Initialize the first row by accumulating values from the left.
    - Initialize the first column by accumulating values from above.
3. Recurrence Relation: For each cell (i, j), compute the minimum path sum as: grid[i][j]=min(grid[i−1][j],grid[i][j−1])+grid[i][j]

Complexity Analysis

- **Time Complexity**: O(m×n), where mmm is the number of rows and nnn is the number of columns in the grid. We traverse each cell once.
- **Space Complexity**: O(1) if we modify the grid in place, as no additional data structures are required.

3. Jump Game (Leetcode) -

Problem Overview

The "Jump Game" problem involves determining whether it is possible to reach the last index of an array starting from the first index. Each element in the array represents the maximum jump length at that position. The challenge is to efficiently assess whether a valid path exists to the end of the array.

Code -

```
class Solution {

    public boolean canJump(int[] nums) {

        boolean dp[] =new boolean[nums.length];

        dp[nums.length-1]=true;

        int last=nums.length-1;

        for(int i=nums.length-2;i>=0;i--)
```

```
        {
            if(nums[i]>=last-i)
            {
                dp[i]=true;
                last=i;
            }
            else
            {
                dp[i]=false;
            }
        }
        return dp[0];
    }
}
```

Explanation -

1. I started by analyzing the mechanics of the jumps. The value at each index indicates how far I can jump forward from that position. To solve this, I needed to determine if there's a way to progress from the start to the end of the array using the provided jump lengths.
2. I implemented a dynamic programming (DP) approach to track which indices can lead to the last index. I created a boolean array dp where dp[i] indicates whether it's possible to reach the last index starting from index i.
3. Base Case Initialization: I initialized the last index of the dp array to true since if I'm already at the last index, I can trivially "reach" the end.
4. Iterating Backwards: I iterated through the array backwards, starting from the second-to-last index. For each index i, I checked if the jump length at that index (nums[i]) was sufficient to reach or surpass the position of the last reachable index (last). If it was, I set dp[i] to true and updated last to the current index i, indicating that this index can also reach the end.
5. Determining the Result: After processing the entire array, I simply returned the value of dp[0]. If it's true, it indicates that starting from the first index, it is possible to reach the last index; otherwise, it's not.
6. Complexity: I noted that the time complexity of this approach is O(n), where nnn is the length of the array, since I traverse the array only once.

1. Finding the longest palidromic substring for the given string

The "Longest Palindromic Substring" problem involves finding the longest substring within a given string that reads the same backward as forward. For example, in the string "babad", the longest palindromic substrings are "bab" and "aba".

Brute Force Approach:

1. Iterate Through the String: We will loop through each character of the string and consider it as a potential center for a palindrome.
2. Expand Around Center: For each character (and for each pair of characters), we will attempt to expand outward to check for palindromic substrings.
    - We will use two indices (low and high) to represent the current substring and expand outward as long as the characters at those indices are the same.
3. Track Maximum Length: If we find a longer palindrome during the expansion, we will update our starting index and maximum length accordingly.

Code -

```
static int fun1(String s) {
        int n = s.length();
        if (n == 0)
            return 0;
        int start = 0, maxLen = 1;

        for (int i = 0; i < n; i++) {
            for (int j = 0; j <= 1; j++) {
                int low = i;
                int high = i + j;
                while (low >= 0 && high < n && s.charAt(low) ==
s.charAt(high)) {
                    int currLen = high - low + 1;
                    if (currLen > maxLen) {
```

```
                    start = low;
                    maxLen = currLen;
                }
                low--;
                high++;
            }
        }
    }

    return maxLen;
}
```

Time Complexity: O(n^3), where n is the length of the string. This is due to the nested loops for the center expansion and the inner while loop checking for palindromes.

Space Complexity: O(1) since we are using only a constant amount of extra space.

Better way improving the time complexity(not same approach said in class):

Explanation -

1. To efficiently check for palindromic substrings, I used a 2D boolean array dp where dp[i][j] indicates whether the substring from index i to index j is a palindrome. This table allows me to keep track of previously calculated results, which avoids redundant checks.

2. Base Cases: I initialized the table by marking single-character substrings as palindromes because any single character is inherently a palindrome. Then, I examined two-character substrings, marking them as palindromic if both characters were the same. This step laid the foundation for checking longer substrings.

3. Filling the DP Table: For substrings longer than two characters, I implemented a nested loop to check each substring of length k. For each substring, I evaluated the characters at both ends. If they matched and the substring enclosed between them was also a palindrome (as indicated by the dp table), I marked the current substring as a palindrome. This recursive relationship was key to building up from smaller substrings to larger ones.

4. Tracking the Longest Palindrome: As I filled the dp table, I kept track of the maximum length of any palindromic substring found and its starting index. This way, I could efficiently return the longest palindrome without needing to store all palindromic substrings.

5. Complexity Considerations: Throughout this process, I noted that the time complexity remained O(n2), which is manageable for typical input sizes. However, I also recognized that this approach requires O(n2) space for the dp table, which could be a consideration for very large strings.

Code -

```
static int fun2(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        int maxLen = 1;
        int start = 0;

        for (int i = 0; i < n; ++i)
            dp[i][i] = true;

        for (int i = 0; i < n - 1; ++i) {
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = true;
                start = i;
                maxLen = 2;
            }
        }

        for (int k = 3; k <= n; ++k) {
            for (int i = 0; i < n - k + 1; ++i) {
                int j = i + k - 1;
                if (dp[i + 1][j - 1] && s.charAt(i) == s.charAt(j)) {
```

```
                    dp[i][j] = true;


                    if (k > maxLen) {
                        start = i;
                        maxLen = k;
                    }
                }
            }
        }

        return maxLen;
    }
```

3. To find unique elements in array -

1. I recognized that a straightforward way to determine uniqueness is to count how many times each element appears in the array. If an element appears only once, it is unique.
2. I opted for a hashmap because it allows for efficient insertion and lookup operations. The average time complexity for these operations is O(1), making it suitable for this task where I need to count occurrences.
3. **Counting Occurrences**: I initiated the process by creating a hashmap where the keys would be the elements of the array, and the values would be their respective counts. I iterated through the array, and for each element, I either added it to the hashmap with a count of 1 or incremented its count if it was already present. This gave me a complete tally of how many times each element appeared.
4. After populating the hashmap, I performed a second pass through the hashmap to collect the keys (elements) that had a value of 1. These keys represented the unique elements in the original array.
5. **Complexity:** I noted that the time complexity of this approach is O(n), where n is the number of elements in the array. The space complexity is also O(n) due to the storage requirements of the hashmap, but this is a trade-off I was willing to make for the performance benefits.

4. To search element in row-wise sorted matrix -

Explanation:

1. I started by recognizing that since each row is sorted, I could utilize binary search for optimal searching. This is more efficient than a linear search, especially in larger matrices, as binary search operates in O(logm) time complexity, where m is the number of columns.

2. Iterating Through Rows: I implemented a loop to go through each row of the matrix. For each row, I applied the binary search algorithm to check if the key exists within that row. The Arrays.binarySearch method provided a straightforward way to perform this check.

3. Handling the Search Result: The result of the binary search returns the index of the found element, or a negative value if the element is not present. I ensured to check if the returned index was positive, indicating that the element was found

4. Tracking the Row Number: To communicate where the element was found, I kept a counter (row_no) to track which row I was currently examining. If the element was found, I printed out its index and the corresponding row number.

5. Complexity Considerations: I noted that the overall time complexity for searching the entire matrix is O(nlogm), where n is the number of rows and mmm is the number of columns. This is efficient for searching in sorted matrices compared to a full traversal, which would be O(n×m).

Code -

```
public class MatrixSearch {

    static void search(int[][] arr, int n, int key) {

        int res = 0, row_no = 0;

        for (int[] row : arr) {

            res = Arrays.binarySearch(row, key);

            if (res > 0) {

                break;

            }

            row_no++;

        }

        System.out.println("Found at:" + res + "in the row " + row_no);
```

```
    }

    public static void main(String[] args) {

        Scanner s = new Scanner(System.in);

        int n = 3, key = 16;

        int[][] arr = new int[n][n];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                arr[i][j] = s.nextInt();

            }

        }

        s.close();

        search(arr, n, key);

    }

}
```

The concepts on (7-10-2024)

Practicing SQL

1. Creating few tables - Employees, Departments, Customers, Orders, Products, Locations

```
create table departments(
dept_id int primary key,
dept_name varchar(50),
loc_id int(10)
);
CREATE TABLE locations (
    location_id INT PRIMARY KEY,
    city VARCHAR(50)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100),
    phone_number VARCHAR(20),
    hire_date DATE,
    job_id VARCHAR(10),
    salary DECIMAL(10, 2),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(dept_id)
);

CREATE TABLE customers (
```

```
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    email VARCHAR(100)
);
CREATE TABLE IF NOT EXISTS products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10, 2)
);

CREATE TABLE IF NOT EXISTS orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

2. Inserting records into tables

```
INSERT INTO departments (dept_id, dept_name, loc_id) VALUES
(10, 'Administration', 1),
(20, 'Marketing', 2),
(50, 'Shipping', 3);

INSERT INTO locations (location_id, city) VALUES
(1, 'New York'),
(2, 'Los Angeles'),
(3, 'Chicago'),
(5, 'Phoenix');
```

```
-- Insert multiple rows into employees
INSERT INTO employees (employee_id, first_name, last_name, email, phone_number, hire_date, job_id, salary, department_id) VALUES
(1, 'John', 'Doe', 'john.doe@example.com', '515.123.4567', '2019-06-17', 'AD_PRES', 24000.00, 10),
(2, 'Jane', 'Smith', 'jane.smith@example.com', '515.123.4568', '2020-02-20', 'AD_VP', 17000.00, 10),
(5, 'Charlie', 'Davis', 'charlie.davis@example.com', '515.123.4561', '2021-11-30', 'SH_CLERK', 3000.00, 50);

-- Insert multiple rows into customers
INSERT INTO customers (customer_id, customer_name, email) VALUES
(1, 'Acme Corp', 'contact@acmecorp.com'),
(4, 'Big Industries', 'sales@bigindustries.com'),
(5, 'Small Startup', 'hello@smallstartup.com');

INSERT INTO orders (order_id,product_id, customer_id, order_date) VALUES
(1,1, 1, '2023-01-15'),
(4,3, 3, '2023-03-01'),
(5,3, 2, '2023-03-15');

INSERT INTO products (product_id, product_name, category, price) VALUES
(1, 'Laptop', 'Electronics', 999.99),
(2, 'Smartphone', 'Electronics', 699.99),
(5, 'Wireless Mouse', 'Electronics', 29.99);
```

3. Query to give output of employees along with their working department

```
select e.first_name, e.last_name, d.dept_name
from employees e
inner join departments d
on e.department_id= d.dept_id;
```

As department name is is present in department table we do join taking reference of the common column which is department id.

4. Left Join and Right Join

```
select c.customer_name , o.order_id

from customers c

left join orders o on c.customer_id =o.customer_id;
```

```
select c.customer_name , o.order_id

from customers c

right join orders o on c.customer_id =o.customer_id;
```

For query 1 we get output of all records of customer name irrespective of them having orders in Orders table

For query 2 it vice versa.

5. Query to find the employees whose salary is greater than the average salary

```
select first_name,last_name, salary

from employees

where salary >(select avg(salary) from employees);
```

This is a concept of subquery. Query inside a query

Subquery:

- The subquery (SELECT AVG(salary) FROM employees) calculates the average salary across all employees in the employees table.

Main Query:

- The main query selects first_name, last_name, and salary from the employees table.
- It filters the results using the WHERE clause to include only those employees whose salary exceeds the average calculated by the subquery.

6. Retrieving the department name, first name, last name, salary of employees, and their department's average salary for those employees whose salary is greater than the overall average salary of all employees.

```
SELECT
   d.dept_name,
   e.first_name,
   e.last_name,
   e.salary,
   (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id) AS
dept_avg_salary
FROM     employees e
INNER JOIN departments d ON e.department_id = d.dept_id
WHERE e.salary > (SELECT AVG(salary) FROM employees)
ORDER BY     d.dept_name, e.salary DESC;
```

- The query performs an inner join between the employees table (aliased as e) and the departments table (aliased as d) using department_id and dept_id. This allows access to both employee and department data.
- The subquery (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id) calculates the average salary for the specific department of each employee.
- The main query selects the department name, employee's first name, last name, salary, and the calculated department average salary.
- The WHERE clause filters employees whose salary is greater than the overall average salary of all employees.
- The results are ordered first by department name and then by employee salary in descending order.

7. Retrieving the names of customers, the count of their orders, and the total value of those orders. It specifically focuses on customers who have placed at least one order.

```
select c.customer_name,
count(o.order_id) as order_count,
```

```
coalesce(sum(p.price),0) as total_order_value
from
customers c
left join  orders o on c.customer_id =o.customer_id
left join products p on p.product_id = o.product_id
group by c.customer_id,c.customer_name
having count(o.order_id)> 0
order by total_order_value desc;
```

- The query starts with the customers table and performs a left join with the orders table to include all customers, even those without orders. It then left joins the products table to access the product prices associated with each order.
- The COUNT(o.order_id) function counts the number of orders for each customer.
- The COALESCE(SUM(p.price), 0) function calculates the total order value. If there are no orders, it returns 0 instead of NULL.
- The results are grouped by customer_id and customer_name, allowing aggregate functions to compute values for each customer.
- The HAVING clause filters the results to include only those customers who have placed at least one order (COUNT(o.order_id) > 0).
- The final results are ordered by total_order_value in descending order, showing customers with the highest total order values first.

8. Retrieving pairs of employees who share the same job title, displaying their first and last names along with their job ID. It ensures that each pair is listed only once.

```
select e1.first_name as employee1_first_name,
e1.last_name as employee1_last_name,
e2.first_name as employee2_first_name,
e2.last_name as employee2_last_name,
e1.job_id
from
employees e1
inner join
```

employees e2

on e1.job_id =e2.job_id and e1.employee_id<e2.employee_id;

- The query performs a self-join on the employees table, allowing it to compare rows within the same table.
- The join condition e1.job_id = e2.job_id ensures that only employees with the same job title are paired.
- The condition e1.employee_id < e2.employee_id ensures that each pair is unique.
- The query selects the first and last names of both employees (aliased as employee1 and employee2) and the job ID of the first employee in each pair.

9. Retrieving department names along with employee details and computes the average salary for each department, as well as the difference between each employee's salary and their department's average.

select d.dept_name,

e.first_name,

e.last_name,

e.salary,

avg(e.salary) over (partition by d.dept_id) as dept_avg_salary,

e.salary-avg(e.salary) over (partition by d.dept_id) as sal_diff

from departments d

left outer join

employees e on

d.dept_id =e.department_id

order by d.dept_name,e.salary desc;

- The query starts with the departments table and performs a left outer join with the employees table to ensure all departments are included, even those without employees.
- The AVG(e.salary) OVER (PARTITION BY d.dept_id) calculates the average salary for employees within each department. This average is repeated for each employee in that department.
- The expression e.salary - AVG(e.salary) OVER (PARTITION BY d.dept_id) computes the difference between each employee's salary and the department's average salary, providing insights into how each employee's salary compares to their peers.

10. Retrieving customer names, order dates, product details, and classifies orders based on the day of the week and product price category. It filters for orders placed within the last two years.

```
select c.customer_name, o.order_date,
case
 when dayofweek(o.order_date) in (1,7) then 'weekend'
 else 'weekday'
end as order_day_type,
p.product_name,p.price,
case
 when p.price<100 then 'budget'
 when p.price between 100 and 500 then 'mid range'
 else 'premium'
end as price_category
from customers c
inner join
orders o on
o.customer_id=c.customer_id
inner join
products p on p.product_id=o.product_id
where o.order_date>=date_sub(curdate(),interval 2 year)
order by o.order_id desc;
```

- The query uses inner joins to link the customers, orders, and products tables, ensuring that only customers with orders and associated products are included.
- The CASE statement WHEN DAYOFWEEK(o.order_date) IN (1, 7) checks if the order date falls on a weekend (1 for Sunday, 7 for Saturday). It labels these as 'weekend' or 'weekday' accordingly.
- Another CASE statement categorizes product prices into 'budget' (less than $100), 'mid range' ($100 to $500), and 'premium' (above $500).
- The WHERE clause filters for orders placed within the last two years using DATE_SUB(CURDATE(), INTERVAL 2 YEAR).
- The results are ordered by o.order_id in descending order, which allows the most recent orders to appear first.

For 6th problem we can even write this query -

```
WITH AvgSalaries AS (
   SELECT
      d.dept_id,
      AVG(e.salary) AS dept_avg_salary
   FROM departments d
   LEFT JOIN  employees e ON d.dept_id = e.department_id
   GROUP BY d.dept_id
)
SELECT
   d.dept_name,
   e.first_name,
   e.last_name,
   e.salary,
   a.dept_avg_salary
FROM  departments d
LEFT JOIN  employees e ON d.dept_id = e.department_id
JOIN  AvgSalaries a ON d.dept_id = a.dept_id
WHERE  e.salary > a.dept_avg_salary
ORDER BY  d.dept_name, e.salary DESC;
```

For 8th problem we can even write this query -

```
SELECT e1.first_name, e1.last_name
FROM employees e1
JOIN (
   SELECT job_id
   FROM employees
   GROUP BY job_id
   HAVING COUNT(*) > 1
) e2 ON e1.job_id = e2.job_id
ORDER BY e1.job_id;
```

The concepts on (8-10-2024)

Practicing few more topics and queries

CREATE TABLE products (product_id INT PRIMARY KEY, product_name VARCHAR(100), category VARCHAR(50), price DECIMAL(10, 2)

);

CREATE TABLE product_recommendations (

product_id INT,recommended_product_id INT,strength DECIMAL(3, 2),PRIMARY KEY (product_id, recommended_product_id),FOREIGN KEY (product_id) REFERENCES products(product_id), FOREIGN KEY (recommended_product_id) REFERENCES products(product_id)

);

Inserting few values into both the tables

1. Retrieve the names of products recommended for the laptop (product_id = 1) along with the strength of each recommendation, ordered by the strength in descending order.

select p1.product_name,

p2.product_name,

pr.strength

from product_recommendations pr

join products p1 on p1.product_id = pr.product_id

join products p2 on pr.recommended_product_id = p2.product_id

where p1.product_id =1

order by pr.strength desc;

Explanation of the Query

I started by identifying that we need to get recommendations for the laptop (product_id = 1).

I used the product_recommendations table to understand which products are recommended together. Since we want the names of the products, I needed to join this table with the products table. I did this twice: once to get the name of the laptop and again to get the names of the recommended products.

Next, I added a WHERE clause to filter the results specifically for the laptop. Finally, I used ORDER BY to sort the recommendations by strength in descending order, so the strongest ones appear first.

2. Retrieve flight details for a journey starting from John F. Kennedy International Airport (JFK) with a layover at an intermediate airport before arriving at Narita International Airport (NRT). Include information about the departure and arrival cities, as well as the relevant flight times, ensuring that the layover flight departs after the initial flight arrives.

```
select f1.flight_id,a1.city as departure_city,a2.city as layover_city,a3.city as arrival_city,

f1.departure_time, f1.arrival_time as layover_arrival, f2.departure_time as
layover_departure,f2.arrival_time

from flights f1

join flights f2 on f1.arrival_airport =f2.departure_airport

join airports a1 on f1.departure_airport=a1.airport_code

join airports a2 on f1.arrival_airport=a2.airport_code

join airports a3 on f2.arrival_airport=a3.airport_code

where f1.departure_airport ='JFK'

and f2.arrival_airport ='NRT'

and f2.departure_time>f1.arrival_time;
```

Approach to Building the Query

1. I recognized that I need to get two flights:
    - The first flight (f1) departs from JFK and arrives at a layover airport.
    - The second flight (f2) departs from that layover airport and arrives at NRT.
2. I used a JOIN between the flights table to connect the two flights based on the arrival airport of the first flight matching the departure airport of the second flight.
3. To get the cities associated with the airport codes, I joined the airports table three times:
    - First to get the departure city from JFK.
    - Second to get the layover city from the arrival airport of the first flight.
    - Third to get the arrival city at NRT from the second flight.
4. I added a WHERE clause to ensure that the departure airport is JFK, the arrival airport for the second flight is NRT, and that the second flight departs after the first flight arrives.

3. Retrieve a hierarchical list of employees starting from the top-level managers down to their subordinates. Display each employee's ID, full name, and their level in the hierarchy, ordering the results by level and employee ID.

```
with recursive employee_hierarchy as (

select employee_id, first_name,last_name , manager_id,0 as level
```

```
from employees

where manager_id is NULL

union all

select e.employee_id, e.first_name,e.last_name,e.manager_id, eh.level+1

from employees e join employee_hierarchy eh on e.manager_id =eh.employee_id

)

select employee_id,

concat(first_name, ' ', last_name) as employee_name,level

from employee_hierarchy

order by level , employee_id;
```

Explanation

1. I recognized that the employees table contains a self-referencing structure where each employee may have a manager, represented by the manager_id.
2. Define the Recursive Common Table Expression (CTE):
    - In the initial part of the CTE, I selected employees with no managers (i.e., top-level employees) by checking for manager_id IS NULL and set their level to 0.
    - In the recursive part, I joined the employees table back to the CTE to find employees who report to those in the previous level, incrementing their level by 1.
3. Select from the CTE: After building the hierarchy, I selected the employee_id, concatenated the first_name and last_name to create a full name, and included the hierarchy level.
4. I used an ORDER BY clause to ensure the results are sorted first by level and then by employee ID.

Tables employees, projects, employee_projects -

```
CREATE TABLE employees (

   employee_id INT PRIMARY KEY,first_name VARCHAR(50),last_name VARCHAR(50),department VARCHAR(50),salary DECIMAL(10, 2),hire_date DATE

);
```

```
INSERT INTO employees VALUES
(1, 'John', 'Doe', 'IT', 75000, '2020-01-15'),
(2, 'Jane', 'Smith', 'HR', 65000, '2019-05-11'),
(3, 'Bob', 'Johnson', 'IT', 80000, '2018-03-23'),
(9, 'Henry', 'Anderson', 'Finance', 71000, '2021-01-05'),
(10, 'Ivy', 'Thomas', 'HR', 63000, '2022-06-30');

CREATE TABLE projects (
    project_id INT PRIMARY KEY,project_name VARCHAR(100),start_date DATE,end_date DATE
);

INSERT INTO projects VALUES
(1, 'Database Migration', '2023-01-01', '2023-06-30'),
(2, 'New HR System', '2023-03-15', '2023-12-31'),
(3, 'Financial Reporting Tool', '2023-02-01', '2023-11-30'),
(4, 'IT Infrastructure Upgrade', '2023-05-01', '2024-04-30');

CREATE TABLE employee_projects (
    employee_id INT,project_id INT,role VARCHAR(50),FOREIGN KEY (employee_id) REFERENCES employees(employee_id),FOREIGN KEY (project_id) REFERENCES projects(project_id)
);

INSERT INTO employee_projects VALUES
(1, 1, 'Project Lead'),
(2, 2, 'Project Manager'),
(3, 1, 'Database Admin');
```

1. Retrieve the top 3 highest-paid employees in each department, along with their employee IDs, names, department names, and salaries. The results should be ordered by department and salary in descending order.

```
WITH RankedEmployees AS (
    SELECT employee_id,first_name,department,salary,
```

```
        RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS ranks
    FROM employees
)
SELECT employee_id,first_name,department,salary,ranks
FROM RankedEmployees
WHERE ranks <= 3
ORDER BY department, salary DESC;
```

Explanation

1. The objective is to find the top 3 highest-paid employees in each department, which requires sorting salaries within each department.
2. Define a Common Table Expression (CTE):
    - I created a CTE called RankedEmployees to rank employees based on their salary within each department. This uses the RANK() window function, which allows us to partition the data by department and order it by salary in descending order.
    - Each employee's rank is determined relative to others in the same department.
3. Select from the CTE: After ranking the employees, I selected the relevant fields: employee_id, first_name, department, salary, and their rank.
4. Filter the Results: I used a WHERE clause to limit the results to only those employees who ranked in the top 3 within their respective departments.
5. Order the Output: Finally, I sorted the results by department and salary in descending order to present the data clearly.

2. Calculate the running total of salaries for employees in each department, ordered by their hire date. The result should include the department, employee names, salary, hire date, and the running total of salaries.

```
select department,first_name,last_name,salary,hire_date,
sum(salary) over( partition by department order by hire_date rows between unbounded preceding and current row) as running_total
from employees
order by department, hire_date;
```

Explanation -

1. I began by selecting the necessary fields: department, first_name, last_name, salary, and hire_date from the employees table.
2. Use the SUM() Window Function: I applied the SUM() window function to calculate the running total:
    - PARTITION BY department: This clause ensures that the running total is calculated separately for each department.
    - ORDER BY hire_date: This clause determines the order in which the salaries are summed, based on the hire date.
    - ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW: This specifies that the running total should include all rows from the start of the partition up to the current row.
3. Finally, I used an ORDER BY clause to sort the results by department and hire date, ensuring the output is presented clearly.

3. Identify employees who are involved in more than one project, along with the count of projects they are working on. The results should include employee ID, name, department, salary, hire date, and the number of projects.

```
select e.employee_id,e.first_name,e.department,e.salary,e.hire_date,count(ep.project_id) as project_count

from employees e join employee_projects ep

on e.employee_id=ep.employee_id

group by ep.employee_id

having count(ep.project_id)>1;
```

Approach -

1. The aim is to find employees who are participating in multiple projects and to count how many projects each of those employees is involved in.
2. Join the Tables: I joined the employees table with the employee_projects table on the employee ID, as this allows us to link each employee to their corresponding projects.
3. In the SELECT clause, I included employee ID, first name, department, salary, hire date, and the count of project IDs.
4. Group the Results: I used the GROUP BY clause to aggregate the results based on each employee's details. This is necessary to perform the count operation on the number of projects.

5. The HAVING clause was added to restrict the results to those employees who are involved in more than one project, using the condition COUNT(ep.project_id) > 1.

4. Identify projects that have team members from all departments. The results should include the project ID and project name.

```
SELECT p.project_id, p.project_name

FROM projects p

JOIN employee_projects ep ON p.project_id = ep.project_id

JOIN employees e ON ep.employee_id = e.employee_id

GROUP BY p.project_id, p.project_name

HAVING COUNT(DISTINCT e.department) = (SELECT COUNT(DISTINCT department) FROM
employees);
```

Approach -

1. The objective is to find projects that involve employees from every department within the organization.
2. I joined the projects table with the employee_projects table to connect each project with its team members. I then joined the employees table to access the department information for each team member.
3. Select Relevant Columns: In the SELECT clause, I included the project ID and project name from the projects table.
4. Group the Results: I used the GROUP BY clause to aggregate results based on the project ID and project name, allowing us to analyze the team composition for each project.
5. Filter with HAVING Clause: The HAVING clause checks that the count of distinct departments involved in each project matches the total number of distinct departments in the organization. This is done by comparing COUNT(DISTINCT e.department) for each project against a subquery that counts the total number of distinct departments in the employees table.

5. Calculate the average salary for each department, considering only employees who were hired in the last 3 years. The result should include the department name and the corresponding average salary.

```
select department,avg(salary) from employees

where hire_date>=date_sub(current_date(),interval 3 year)

group by department;
```

Approach -

1. The aim is to find the average salary for employees in each department, but only for those hired within the last 3 years.
2. Select Relevant Columns: I included the department column and calculated the average salary using AVG(salary).
3. Filter Employees by Hire Date: I used the WHERE clause to filter employees based on their hire date. The condition checks if the hire date is within the last 3 years using DATE_SUB(CURRENT_DATE(), INTERVAL 3 YEAR).
4. Group the Results: The GROUP BY clause groups the results by department, allowing the average salary to be calculated for each department.

7. Find the employee(s) with the highest salary in their respective departments who are also working on the longest-running project. The result should include employee details and the project name.

```
with temp_table 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 temp_table t
join employees e on e.department=t.department
join employee_projects ep on ep.employee_id=e.employee_id
join projects p on p.project_id=ep.project_id
join longest_project lp on lp.project_id=p.project_id
where e.salary=t.max_salary;
```

Approach -

1. I needed to find the employees earning the highest salary in each department and ensure they are working on the longest-running project.

2. Create a Temporary Table for Maximum Salaries:
    - I defined a Common Table Expression (CTE) named temp_table to calculate the maximum salary for each department using MAX(salary) and grouped the results by department.
3. Identify the Longest-Running Project:
    - I defined another CTE called longest_project to find the project with the greatest duration by calculating the difference between end_date and start_date. I sorted the projects by duration in descending order and limited the results to the top one.
4.
    - In the main query, I joined the temp_table with the employees table to get employee details, ensuring I match the department.
    - I also joined with the employee_projects and projects tables to connect employees with the projects they are working on.
    - Finally, I used a WHERE clause to filter the results, ensuring that only employees whose salary matches the maximum salary for their department are included.

8. Calculate the total salary for each department and determine what percentage that total salary represents compared to the overall salary of the company. The result should include the department name, total salary, and the percentage.

```
select department,sum(salary),sum(salary)/(select sum(salary) from employees) *100 as percentage

from employees

group by department;
```

Approach -

1. The aim is to find the total salary for each department and calculate how that total compares to the company's overall salary.
2. Select Relevant Columns: In the SELECT clause, I included the department and used SUM(salary) to calculate the total salary for each department.
3. Calculate the Total Company Salary:
    - I used a subquery to calculate the total salary of the company with SELECT SUM(salary) FROM employees. This total will serve as the denominator in our percentage calculation.
4. Calculate the Percentage: I divided the total department salary by the total company salary and multiplied by 100 to get the percentage.
5. Group the Results: I used the GROUP BY clause to ensure that the results are aggregated by department.

9. Identify employees whose salaries exceed the average salary of their respective departments. Additionally, calculate the percentage by which their salary exceeds the average. The result should include employee ID, their salary, the department's average salary, and the percentage difference.

```
with temp_table as(

select department,avg(salary) as avg_salary

from employees

group by department)

select e.employee_id,e.salary,t.avg_salary,(e.salary-t.avg_salary)/t.avg_salary *100 as Percentage_by

from employees e join temp_table t

on e.department=t.department

where e.salary>t.avg_salary;
```

Approach -

1. The aim is to find employees who earn more than the average salary in their departments and to calculate how much more they earn in percentage terms.
    - I defined a Common Table Expression (CTE) called temp_table to calculate the average salary for each department using AVG(salary) and grouped the results by department.
    - In the main query, I selected the employee ID, their salary, and the average salary from the temp_table.
    - I calculated the percentage by which the employee's salary exceeds the department average using the formula (e.salary - t.avg_salary) / t.avg_salary * 100.
    - I used a WHERE clause to filter the results to include only those employees whose salary is greater than their department's average salary.

Explain keyword -

When we run an ‘explain’ statement in SQL, it provides insight into how the database engine plans to execute the query. Each column in the output represents different aspects of the execution plan.

Query -

```
explain select e.employee_id,e.first_name,e.department,e.salary,e.hire_date,count(ep.project_id) as project_count

from employees e join employee_projects ep

on e.employee_id=ep.employee_id
```

group by ep.employee_id

having count(ep.project_id)>1;

Output -

1	SIMPLE	ep	ALL	employee_id				10	100.00	Using where; Using temporary
1	SIMPLE	e	eq_ref	PRIMARY	PRIMARY	4	projects.ep.employee_id	1	100.00	

id: This column represents the identifier for the query. In our output, both rows have the same id, indicating that they belong to the same query block.

select_type: This indicates the type of query being executed. Here, SIMPLE means that the query does not contain any subqueries or unions.

table: This column specifies the table that the database is accessing at this step in the query. In our output, the first row is for the employee_projects table (ep), and the second row is for the employees table (e).

type: This shows the join type being used.

- ALL indicates a full table scan, which means the database is scanning every row in the employee_projects table to find matches.
- eq_ref indicates a more efficient access method where the database uses an index on the employees table to quickly find matching rows based on a primary key.

possible_keys: This column lists any indexes that could be used for the operation. In our output, it's blank for the ep table, suggesting no suitable index was found for that table. For e, it shows PRIMARY, indicating that the primary key can be used.

key: This shows which key (index) the database actually decided to use for the query. For ep, there's no key used since it's performing a full scan, while e uses the PRIMARY key.

key_len: This represents the length of the key used. For the PRIMARY key in the employees table, the length is 4 bytes, which is typical for integer primary keys.

ref: This column indicates which columns or constants are compared to the key. For e, it shows projects.ep.employee_id, meaning the join condition is based on the employee_id column from the employee_projects table.

rows: This estimates the number of rows the database expects to examine for this step of the query. For ep, it estimates 10 rows, while for e, it expects to check 1 row.

Extra: This column provides additional information about the query execution.

- Using where indicates that a filtering condition is applied.
- Using temporary suggests that a temporary table is used to hold intermediate results, which can indicate that the query involves grouping or sorting.

The concepts on (9-10-2024)

Normalization -

Normalization is used to reduce data redundancy. It provides a method to remove the following anomalies from the database and bring it to a more consistent state:

A database anomaly is a flaw in the database that occurs because of poor planning and redundancy.

1. Insertion anomalies: This occurs when we are not able to insert data into a database because some attributes may be missing at the time of insertion.
2. Updation anomalies: This occurs when the same data items are repeated with the same values and are not linked to each other.
3. Deletion anomalies: This occurs when deleting one part of the data deletes the other necessary information from the database.

Types of Normal Forms -

1. 1st Normal Form -

A relation is in 1NF if all its attributes have an atomic value.

Eg - Table contains columns -

employee_id, name, mobile no
As there can be multiple mobile no's, we split the table

2. 2nd Normal Form -

A relation is in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the candidate key in DBMS.

Eg - Table contains -

employee_id, project_id, employee_name, project_name

the prime attributes of the table are Employee Code and Project ID. We have partial dependencies in this table because Employee Name can be determined by Employee Code and Project Name can be determined by Project ID. Thus, the above relational table violates the rule of 2NF.

We then convert <EmployeeProjectDetail> table into 2NF by decomposing it into <EmployeeDetail>, <ProjectDetail> and <EmployeeProject> tables. As you can see, the above tables satisfy the following two rules of 2NF as they are in 1NF and every non-prime attribute is fully dependent on the primary key.

3. 3rd Normal Form -

A relation is in 3NF if it is in 2NF and there is no transitive dependency.

Eg - Employee_id, employee_name, employee_zipcode, city

The above table is not in 3NF because it has Employee Code -> Employee City transitive dependency because:

- Employee Code -> Employee Zipcode
- Employee Zipcode -> Employee City

Also, Employee Zipcode is not a super key and Employee City is not a prime attribute.

Thus, we've converted the <EmployeeDetail> table into 3NF by decomposing it into <EmployeeDetail> and <EmployeeLocation> tables as they are in 2NF and they don't have any transitive dependency.

The 2NF and 3NF impose some extra conditions on dependencies on candidate keys and remove redundancy caused by that. However, there may still exist some dependencies that cause redundancy in the database. These redundancies are removed by a more strict normal form known as BCNF.

4. BCNF Normal Form -

A relation is in BCNF if it is in 3NF and for every Functional Dependency, LHS is the super key.

Specialization -

In specialization, an entity is divided into sub-entities based on its characteristics. It is a top-down approach where the higher-level entity is specialized into two or more lower-level entities. For Example, an EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER, etc.

Generalization -

Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher-level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher-level entity called PERSON. In this case, common attributes like P_NAME, and P_ADD become part of a higher entity (PERSON), and specialized attributes like S_FEE become part of a specialized entity (STUDENT).

Activity -

Normalizing the tables and then constructing ER diagram for the given set of tables

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)
);
```

Normalizing into 1NF -

1. Customers table -

In the real world, customers may have multiple email addresses or phone numbers, but this table restricts the customer to one email.

If we need to support multiple emails or phone numbers, we'd split them into a new table to maintain 1NF:

```
CREATE TABLE customer_emails (
    email_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    email VARCHAR(100),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
```

);

TABLE customers (

customer_id INT PRIMARY KEY AUTO_INCREMENT,

first_name VARCHAR(50),

last_name VARCHAR(50),

registration_date DATE

);

2. products Table -

A product might have multiple categories (e.g., a smartphone could belong to both "Electronics" and "Gadgets"). Storing multiple categories in one column would violate 1NF.

To fix this, we could introduce a separate table for categories:

CREATE TABLE product_categories (

product_id INT,

category VARCHAR(50),

FOREIGN KEY (product_id) REFERENCES products(product_id)

);

TABLE products (

product_id INT PRIMARY KEY AUTO_INCREMENT,

product_name VARCHAR(100),

price DECIMAL(10, 2),

stock_quantity INT

);

3. orders Table -

The total_amount could be derived from order_items. In 1NF, there is no issue with this. However, in real-world scenarios, it's better to calculate the total_amount dynamically to avoid inconsistency.

Storing it as a calculated value in the orders table is acceptable for now as per 1NF, but for 3NF (later), this needs to be reconsidered.

4. order_items Table -

This table seems well-normalized for 1NF. Each column is atomic, and there are no repeating groups.

No further changes are needed.

5. product_reviews Table -

In the real world, a product could have multiple reviews from the same customer. However, each review is atomic because each row represents one review by one customer.

If customers could have multiple ratings or review texts, we would need to store each review in its own row, which is already happening here.

Normalizing to 2NF -

1. customers Table (2NF):

- The customers table has a single-column primary key (customer_id), so it already satisfies 2NF, as all non-key attributes (first name, last name, email, registration date) depend directly on customer_id.

2. products Table (2NF):

- Similarly, the products table uses a single-column primary key (product_id). All non-key attributes (product_name, category, price, stock_quantity) depend directly on product_id.
- Therefore, it satisfies 2NF.

3. orders Table (2NF):

- The orders table also has a single-column primary key (order_id), and all attributes are directly dependent on the order_id. Therefore, this table is in 2NF.

4. order_items Table (2NF):

- Primary Key: order_item_id (single column).
- Each column in this table depends fully on order_item_id. There are no partial dependencies here.
- It satisfies 2NF.

5. product_reviews Table (2NF):

- Primary Key: review_id (single column).
- All other columns (product_id, customer_id, rating, review_text, review_date) depend fully on review_id. Hence, it satisfies 2NF.

Normalizing to 3NF -

Only in orders table there will be an issue of transitive dependency

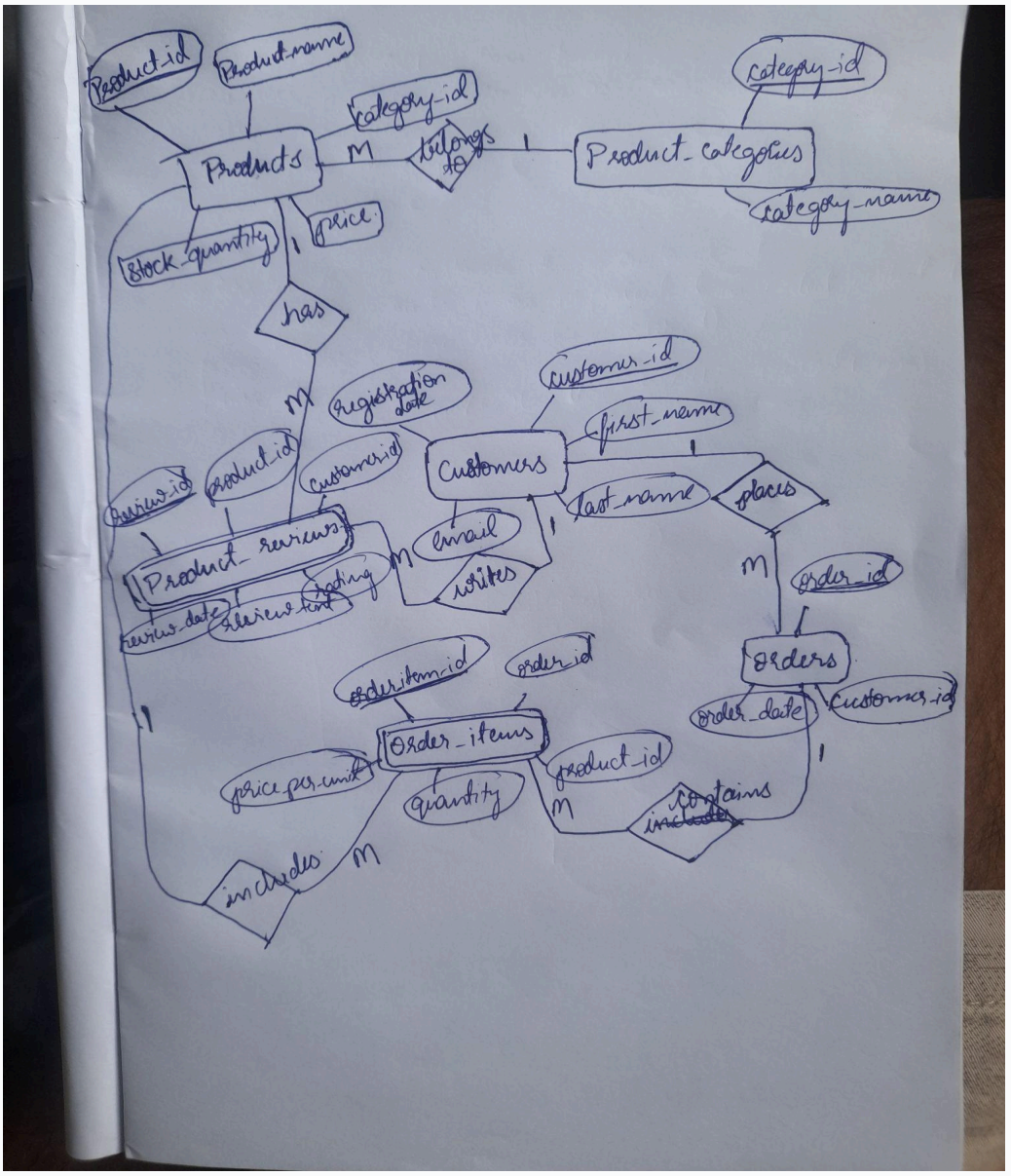
In orders table total_amount can be derived from the order_items table by summing up the price and quantity of the products in an order. Therefore, total_amount is redundant and introduces a transitive dependency because it depends on the order_id, but it can also be calculated from the values in the order_items table.

1. To convert into 3NF form we should remove the the total_amount column

2. The total amount for each order should be calculated dynamically by summing the price from the related entries in the order_items table.

```
TABLE orders (
  order_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  order_date DATETIME,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

ER diagram -



Activity -

CREATE TABLE products (product_id INT PRIMARY KEY, product_name VARCHAR(100), category VARCHAR(50), unit_price DECIMAL(10, 2));

CREATE TABLE sales (sale_id INT PRIMARY KEY, product_id INT, sale_date DATE, quantity INT, FOREIGN KEY (product_id) REFERENCES products(product_id));

Insert sample data INSERT INTO products (product_id, product_name, category, unit_price) VALUES (1, 'Laptop', 'Electronics', 999.99), (2, 'Smartphone', 'Electronics', 599.99), (3, 'Tablet', 'Electronics', 399.99), (4, 'Desk Chair', 'Furniture', 149.99), (5, 'Coffee Table', 'Furniture', 199.99), (6, 'Bookshelf', 'Furniture', 89.99), (7, 'Running Shoes', 'Apparel', 79.99), (8, 'T-shirt', 'Apparel', 19.99), (9, 'Jeans', 'Apparel', 59.99);

INSERT INTO sales (sale_id, product_id, sale_date, quantity) VALUES (1, 1, '2023-01-15', 2), (2, 2, '2023-01-16', 3), (3, 4, '2023-01-17', 1), (4, 7, '2023-01-18', 4), (5, 3, '2023-02-01', 2), (6, 5, '2023-02-02', 1), (7, 8, '2023-02-03', 5), (8, 1, '2023-02-15', 1), (9, 6, '2023-02-16', 2), (10, 2, '2023-02-17', 2), (11, 9, '2023-03-01', 3), (12, 4, '2023-03-02', 2), (13, 7, '2023-03-03', 3), (14, 3, '2023-03-15', 1), (15, 5, '2023-03-16', 1);

Problem Statement:

-- Analyze the sales data to find the top-performing product category for each month of 2023.

-- The performance should be based on total revenue (quantity sold * unit price).

-- Your query should return the month, the top-performing category, and its total revenue.

-- In case of a tie, include all categories with the same top performance.

Query -

```
with monthly_rev as (
   select date_format(s.sale_date, '%Y-%m') as sale_month, p.category,
          sum(s.quantity * p.unit_price) as total_revenue
   from sales s join products p on s.product_id = p.product_id
   where year(s.sale_date) = 2023
   group by sale_month, p.category
),
ranked_categories as (
   select sale_month,category,total_revenue,
          rank() over (partition by sale_month order by total_revenue desc) as revenue_rank
   from monthly_rev
```

```
)

select sale_month,category,total_revenue

from ranked_categories

where revenue_rank = 1;
```

Approach -

Calculate Monthly Revenue: First, calculate the total revenue (quantity sold * unit price) for each product sale grouped by both product category and month of the sale date.

Rank Categories by Revenue: Use a window function, like RANK(), to rank product categories within each month based on their total revenue, sorted in descending order.

Select Top Categories: Filter the results to only include the top-ranked categories for each month. In case of ties, include all categories with the same rank.

Aggregate Results: Finally, return the month, category name, and total revenue for the top-performing categories.

The concepts on (10-10-2024)

MySQL Execution Engine

The MySQL execution engine is a crucial component of the MySQL database management system. It handles the execution of SQL queries and manages how data is retrieved and manipulated. Here's a breakdown of its key aspects:

1. Query Processing

- **Parsing**: When a SQL query is received, the execution engine first parses it to check for syntax errors and validates the query structure.
- **Optimization**: The engine then optimizes the query using the query optimizer. It decides the most efficient way to execute the query by considering different execution plans.

2. Execution Phases

- **Execution Plan**: The execution engine generates an execution plan based on the optimization phase. This plan outlines the steps to retrieve or manipulate the required data.
- **Execution**: The engine executes the plan, interacting with the storage engine to retrieve or modify data as necessary.

3. Storage Engines

- MySQL supports multiple storage engines (e.g., InnoDB, MyISAM). Each engine has different characteristics for handling data storage and retrieval.
- The choice of storage engine can affect performance, transactions, and data integrity.

4. Data Retrieval

- **Access Methods**: The execution engine uses various access methods to fetch data, such as table scans or index lookups, depending on the execution plan.
- **Join Operations**: For queries involving multiple tables, the execution engine performs join operations (e.g., INNER JOIN, LEFT JOIN) to combine rows from different tables.

5. Result Set

- After executing the query, the engine returns the result set to the client application. This includes the requested data in a structured format.

6. Error Handling

- The execution engine manages errors that may occur during execution, providing feedback to the user and maintaining data integrity.

Query Optimization -

1. Do Statistical Analysis

Statistical analysis involves gathering and analyzing data about the database's structure and contents. This information helps the query optimizer make informed decisions. Key aspects include:

- **Data Distribution**: Analyzing how data is distributed across tables and columns (e.g., value frequencies).
- **Histograms**: Creating histograms to provide insights into the distribution of values in a column, which can influence index selection.
- **Cardinality**: Understanding the uniqueness of data in a column (high cardinality means many unique values; low cardinality means few unique values).

2. Cost Estimation

Once statistical data is gathered, the optimizer uses it to estimate the "cost" of different execution plans. Cost estimation typically involves:

- **Resource Usage**: Evaluating CPU, memory, and I/O operations required for different strategies (e.g., using indexes vs. full table scans).
- **Execution Time**: Estimating how long it will take to execute each potential plan based on the resource usage.
- **Trade-offs**: Balancing between faster execution and resource consumption, helping to choose the most efficient plan.

3. Plan Generation

In this phase, the optimizer generates various execution plans based on the SQL query and the statistical data. Key points include:

- **Possible Strategies**: Considering different strategies for data access, such as:
    - Using indexes
    - Choosing join algorithms (e.g., nested loop, hash join)
    - Determining whether to filter data early (pushdown predicates)
- **Plan Structure**: Each generated plan has a specific structure outlining how the query will be executed.

4. Plan Selection

After generating multiple execution plans, the optimizer selects the most efficient one based on the estimated costs. This process involves:

- **Comparing Costs**: Evaluating the estimated costs of all generated plans and selecting the one with the lowest cost.
- **Adaptive Optimization**: In some cases, the optimizer might adapt the plan based on runtime statistics or conditions encountered during execution.
- **Caching**: Storing chosen execution plans for reuse in future queries to improve performance.

Delimiters -

In MySQL, a delimiter is a character or string that indicates the end of a statement. By default, the delimiter is a semicolon (;). However, when creating stored procedures, triggers, or functions, you often need to use multiple statements. This is where changing the delimiter becomes necessary.

Need of Changing the Delimiter?

1. **Multiple Statements**: When defining a stored procedure, you might have multiple SQL statements separated by semicolons. If you don't change the delimiter, MySQL will interpret the first semicolon it encounters as the end of the entire procedure.
2. **Avoiding Errors**: Changing the delimiter allows you to encapsulate your entire procedure (or function, etc.) in a single statement.

Example -

```
DELIMITER //

CREATE PROCEDURE COUNT_RECORDS(IN tableName VARCHAR(64))

BEGIN

   SET @sql = CONCAT('SELECT COUNT(*) INTO @count FROM ', tableName);

   PREPARE stmt FROM @sql;

   EXECUTE stmt;

   DEALLOCATE PREPARE stmt;

   SELECT @count AS total_count;

END //

DELIMITER ;
```

DELIMITER // changes the delimiter from ; to //. This allows the entire procedure definition to be treated as one statement.

Stored Procedures -

Stored procedures are precompiled collections of SQL statements stored in the database. They can be executed on demand and are particularly useful for encapsulating complex logic, improving performance, and ensuring consistency.

Key Features of Stored Procedures

1. **Encapsulation**: Procedures allow you to group related SQL statements, making it easier to manage and reuse code.
2. **Parameters**: You can define input (IN), output (OUT), or both (INOUT) parameters, allowing flexibility in how procedures are executed.
3. **Performance**: Since stored procedures are precompiled, they can improve performance by reducing the amount of SQL parsing and execution planning needed.
4. **Security**: Procedures can help enforce security by restricting direct access to tables while allowing access through the procedure.
5. **Transaction Control**: Stored procedures can manage transactions, enabling multiple operations to be treated as a single unit of work.

Syntax -

```
CREATE PROCEDURE procedure_name (parameters)
BEGIN
   -- SQL statements
END;
```

Example -

```
DELIMITER //
CREATE PROCEDURE GET_EMPLOYEE_DETAILS(IN emp_id INT)
BEGIN
   SELECT name, position, salary FROM employees WHERE id = emp_id;
END //
DELIMITER ;
```

Indexing -

Indexing is a database optimization technique used to speed up the retrieval of rows from a table. An index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional space and slower write operations (like INSERT, UPDATE, and DELETE).

Key Benefits of Indexing

1. **Faster Query Performance**: Indexes significantly reduce the amount of data the database engine needs to scan when executing a query, leading to quicker response times.
2. **Efficient Sorting**: Indexes help with sorting data efficiently, especially for ORDER BY clauses.
3. **Unique Constraints**: Unique indexes ensure that no two rows have the same value in specified columns, helping maintain data integrity.
4. **Improved JOIN Performance**: Indexes on foreign keys can improve the performance of JOIN operations between tables.

Types of Indexes

1. **Single-Column Index**: An index based on a single column of a table.
2. **Composite Index**: An index that uses multiple columns to create a more complex structure, useful for queries that filter on multiple columns.
3. **Unique Index**: Ensures that all values in the index column are unique.
4. **Full-Text Index**: Used for full-text searches in text-based columns.
5. **Spatial Index**: Designed for spatial data types to speed up spatial queries.

Example -

There are tables customers and orders table and we had created index to the ‘value’ column

create index idx_customer_reg_Date on customers(registration);

Then used select query to retrieve information from table

select o.orderId,o.orderDate,o.TotalAmount,c.Name

from orders o join customers c use index(idx_customer_reg_Date) on o.customer_id =c.customer_id

where c.registration>=DATE_SUB(CURDATE(),INTERVAL 30 day);

When we see the execution plan of we see time improvement in retrieving the rows.

Force Index and Ignore Index in MySQL

In MySQL, indexes are crucial for optimizing query performance. However, there are scenarios where the optimizer might not choose the best index for a given query. In such cases, you can use **FORCE INDEX** and **IGNORE INDEX** to explicitly control index usage.

FORCE INDEX
The FORCE INDEX hint instructs the MySQL optimizer to use a specific index, even if it thinks another index (or no index) might be more efficient. This can be useful in scenarios where you know the data distribution and how the index will improve performance better than the optimizer does.

Syntax -

SELECT * FROM table_name FORCE INDEX (index_name) WHERE condition;

Example:

Suppose you have a products table with two indexes: idx_category and idx_price. You want to query for products in a specific category but know that the idx_price index will yield better performance due to the data distribution.

SELECT * FROM products FORCE INDEX (idx_price) WHERE category = 'Electronics';

IGNORE INDEX
The IGNORE INDEX hint tells the optimizer to ignore one or more specified indexes when executing a query. This is useful if you believe that using a particular index is slowing down the query, or if the optimizer is choosing a suboptimal index.

Syntax -

SELECT * FROM table_name IGNORE INDEX (index_name) WHERE condition;

Example:

Continuing with the products table, you might want to ignore the idx_price index because you have determined that it performs poorly for a specific query:

SELECT * FROM products IGNORE INDEX (idx_price) WHERE category = 'Furniture';

When to Use

- **FORCE INDEX**: Use when you know that a specific index will significantly improve query performance, especially in cases where the optimizer's choice seems inefficient.
- **IGNORE INDEX**: Use when the optimizer is choosing an index that you know is not optimal for your query. This can help reduce query execution time.

The concepts on (11-10-2024)

Database design -

Modelling database -

One way is Normalization the other way is by dimensional modelling

Dimensional Modelling -

Dimensional Data Modeling is one of the data modeling techniques used in data warehouse design. The concept of Dimensional Modeling was developed by Ralph Kimball which is comprised of facts and dimension tables. Since the main goal of this modeling is to improve the data retrieval so it is optimized for SELECT OPERATION. The advantage of using this model is that we can store data in such a way that it is easier to store and retrieve the data once stored in a data warehouse.

1. Star Schema

A **star schema** is a type of dimensional modeling that organizes data in a way that makes it easy to retrieve for reporting and analysis. It is called a "star schema" because the diagram of the database looks like a star: the central fact table is at the hub, and the dimension tables radiate outward like the points of a star.

Example of Star Schema:

Let's take the example of a Sales Data Warehouse. The main business process we are modeling is sales transactions.

1. Fact Table:

The Fact Table contains numeric measurements or facts, such as sales_amount, quantity_sold, etc. It also contains foreign keys that link to the associated dimension tables.

- Fact Table Name: Sales_Fact
- Columns:

    sale_id (Primary Key),customer_id (Foreign Key),product_id (Foreign Key),store_id (Foreign Key),date_id (Foreign Key),sales_amount,quantity_sold

2. Dimension Tables:

The Dimension Tables provide context to the facts. They store descriptive information that allows you to slice and dice the data in various ways.

- Customer Dimension (Customer_Dim):

    customer_id (Primary Key),customer_name,email,city,country

- Product Dimension (Product_Dim):

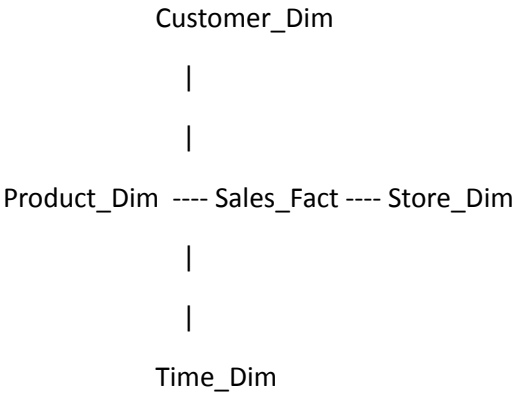
product_id (Primary Key),product_name,category,brand,price

- Store Dimension (Store_Dim):

  store_id (Primary Key),store_name,city,country

- Time Dimension (Time_Dim):

  date_id (Primary Key),date,month,quarter,year



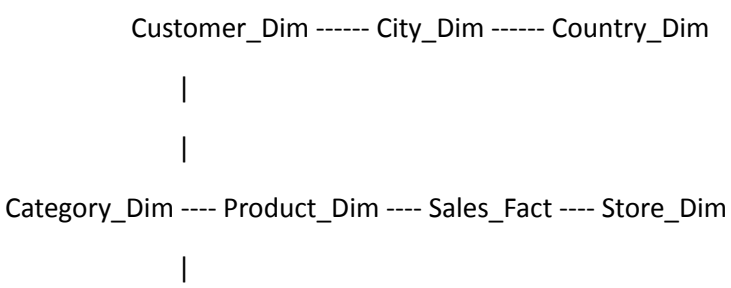
2. Snowflake schema -

A snowflake schema is equivalent to the star schema. "A schema is known as a snowflake if one or more dimension tables do not connect directly to the fact table but must join through other dimension tables."

The snowflake schema is an expansion of the star schema where each point of the star explodes into more points. It is called snowflake schema because the diagram of snowflake schema resembles a snowflake. Snowflaking is a method of normalizing the dimension tables in a STAR schemas. When we normalize all the dimension tables entirely, the resultant structure resembles a snowflake with the fact table in the middle.

Snowflaking is used to develop the performance of specific queries. The schema is diagramed with each fact surrounded by its associated dimensions, and those dimensions are related to other dimensions, branching out into a snowflake pattern.

The snowflake schema consists of one fact table which is linked to many dimension tables, which can be linked to other dimension tables through a many-to-one relationship. Tables in a snowflake schema are generally normalized to the third normal form. Each dimension table performs exactly one level in a hierarchy.



|

Time_Dim

3. Fact Constellation -

A Fact constellation means two or more fact tables sharing one or more dimensions. It is also called Galaxy schema.

Fact Constellation Schema describes a logical structure of data warehouse or data mart. Fact Constellation Schema can design with a collection of de-normalized FACT, Shared, and Conformed Dimension tables.

Fact Constellation Schema is a sophisticated database design that is difficult to summarize information. Fact Constellation Schema can implement between aggregate Fact tables or decompose a complex Fact table into independent simplex Fact tables.

This schema defines two fact tables, sales, and shipping. Sales are treated along four dimensions, namely, time, item, branch, and location. The schema contains a fact table for sales that includes keys to each of the four dimensions, along with two measures: Rupee_sold and units_sold. The shipping table has five dimensions, or keys: item_key, time_key, shipper_key, from_location, and to_location, and two measures: Rupee_cost and units_shipped.

4. Slowly Changing Dimensions -

Slowly changing dimensions refer to how data in your data warehouse changes over time. Slowly changing dimensions have the same natural key but other data columns that may or may not change over time depending on the type of dimensions that it is.

Slowly changing dimensions are important in data analytics to track how a record is changing over time. The way the database is designed directly reflects whether historical attributes can be tracked or not, determining different metrics available for the business to use.

For example, if data is constantly being overwritten for one natural key, the business will never be able to see how changes in that row's attributes affect key performance indicators. If a company continually iterates on a product and its different features, but doesn't track how those features have changed, it will have no idea how customer retention, revenue, customer acquisition cost, or other marketing analytics were directly impacted by those changes.

Type 0 -

Type 0 refers to dimensions that never change. You can think of these as mapping tables in your data warehouse that will always remain the same, such as states, zipcodes, and county codes. Date_dim tables that you may use to simplify joins are also considers type 0 dimensions. In addition to mapping tables, other pieces of data like social security number and date of birth are considered type 0 dimensions.

Type 1 -

Type 1 refers to data that is overwritten by new data without keeping a historical record of that old piece of data. With this type, there is no way to keep track of changes over time. I've seen many companies use this type of dimension accidentally, not realizing that they can never get the old values back. When implementing this dimension, make sure you do not need to track the trends in that data column over time.

A good example of this is customer addresses. You don't need to keep track of how a customer's address has changed over time, you just need to know you are sending an order to the right place.

Type 2 -

Type 2 dimensions are always created as a new record. If a detail in the data changes, a new row will be added to the table with a new primary key. However, the natural key would remain the same in order to map a record change to one another. Type 2 dimensions are the most common approach to tracking historical records.

There are a few different ways you can handle type 2 dimensions from an analytics perspective. The first is by adding a flag column to show which record is currently active. This is the approach Fivetran takes with data tables that have CDC implemented. Instead of deleting any historic records, they will add a new one with the _FIVETRAN_DELETED column set to FALSE. The old record will then be set to TRUE for this _FIVETRAN_DELETED column. Now, when querying this data, you can use this column to filter for records that are active while still being able to get historical records if needed.

You can also handle type 2 dimensions by adding a timestamp column or two to show when a new record was created or made active and when it was made ineffective. Instead of checking for whether a record is active or not, you can find the most recent timestamp and assume that is the active data row. You can then piece together the timestamps to get a full picture of how a row has changed over time.

5. Junk Dimension -

When dealing with data warehousing and business intelligence, a frequently encountered concept is the Junk Dimension. It is a dimension table in a star schema of a data warehouse, which combines several low cardinality flags and indicators. A junk dimension is essentially a collection of random transactional codes, flags, and/or text attributes which are unrelated to any particular dimension. These dimensions are designed to help in improving the efficiency of queries against fact tables.

Functionality and Features

Junk Dimension's primary function is to group rarely used or low-cardinality attributes, such as flags or indicators, in a separate dimension table. This role ensures these attributes don't clutter the primary dimension tables which can have more meaningful, often used attributes. Furthermore, by placing these attributes in the junk dimension, the data model becomes easier to navigate, and data analysis is more streamlined.

Benefits and Use Cases

The introduction of the Junk Dimension in a data warehouse setup can bring several benefits to businesses. The most significant advantage being the reduction in query complexity. By consolidating the random, less frequently used fields into a single table, it helps reduce the size and complexity of the fact table. Moreover, the use of a junk dimension can improve query performance and simplifies data warehouse design. It is widely used in scenarios where the business needs to analyze data regarding flags or indicators that aren't frequently needed.

Challenges and Limitations

Despite its benefits, the implementation of the Junk Dimension is not free from challenges. The most notable limitation is the requirement of proper maintenance and management. As the volume of data grows, it may become challenging to manage the low cardinality flags and indicators, affecting overall query performance. Also, if these rarely used indicators suddenly become critical business indicators, moving them out of the junk dimension can be a difficult process.

6. Degenerate Dimension -

Degenerate Dimension, often termed as a junk dimension, is a type of dimension table in a star schema of a data warehouse. It does not have its own dimension table, but resides in the fact table as a primary key. Predominantly, it is used for storing transaction control values, invoice numbers, or other numeric values that are useful for tracking, but do not link to other dimensions.

Functionality and Features

The primary function of a degenerate dimension is to assist in transaction tracking and streamline data analytics. The key features of a degenerate dimension are:

- It helps in categorizing facts in the fact table.
- It provides a way to track the sequence of events in a transaction.
- It facilitates quicker data retrieval due to residing in fact table.

Benefits and Use Cases

- Reduced Complexity: The dimension being part of the fact table cuts down the need for joining tables which simplifies queries.
- Effective Tracking: With the help of Degenerate Dimension, tracking the sequence of events or transactions becomes easier.
- Speed: As it resides in the fact table, data retrieval is quicker which is crucial for time-sensitive analytics.

One primary use case of Degenerate Dimension is in retail sales, where invoice numbers act as a degenerate dimension to track each unique transaction.

Triggers -

Triggers in SQL are special types of stored procedures that automatically execute (or "fire") in response to certain events on a specified table or view. They are commonly used for enforcing business rules, validating data, and keeping track of changes.

Types of Triggers

1. **DML Triggers**: Fired by Data Manipulation Language (INSERT, UPDATE, DELETE) events.
    - **AFTER Trigger**: Executes after the triggering event.
    - **BEFORE Trigger**: Executes before the triggering event.
2. **INSTEAD OF Trigger**: Used mainly with views, this trigger executes instead of the triggering event.
3. **DDL Triggers**: Fired by Data Definition Language (CREATE, ALTER, DROP) events.
4. **LOGON and LOGOFF Triggers**: Used for actions on user sessions.

Syntax's -

1. Syntax for DML Triggers

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER }
{ INSERT | UPDATE | DELETE }
ON table_name
FOR EACH ROW
BEGIN
   -- trigger logic here
END;
```

2. Syntax for INSTEAD OF Trigger

```
CREATE TRIGGER trigger_name
INSTEAD OF
{ INSERT | UPDATE | DELETE }
ON view_name
FOR EACH ROW
BEGIN
   -- trigger logic here
END;
```

3. Syntax for DDL Triggers

CREATE TRIGGER trigger_name

AFTER | BEFORE

{ CREATE | ALTER | DROP }

ON schema

FOR EACH STATEMENT

BEGIN

-- trigger logic here

END;

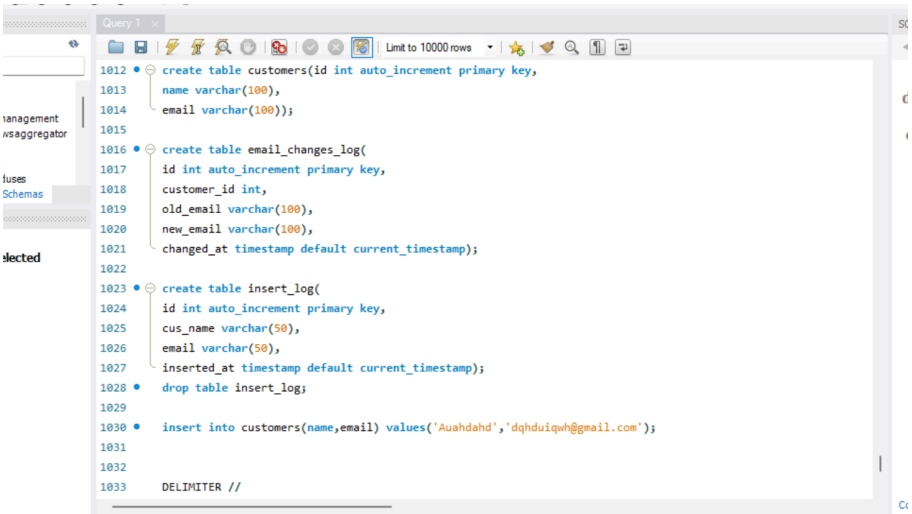
Need of triggers -

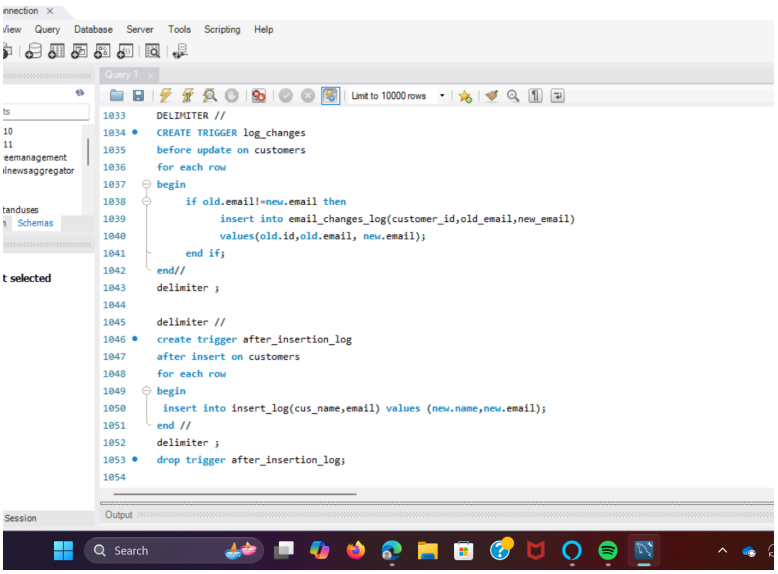
When we need to carry out some actions automatically in certain desirable scenarios, triggers will be useful. For instance, we need to be aware of the frequency and timing of changes to a table that is constantly changing. In such cases, we could create a trigger to insert the required data into a different table if the primary table underwent any changes.

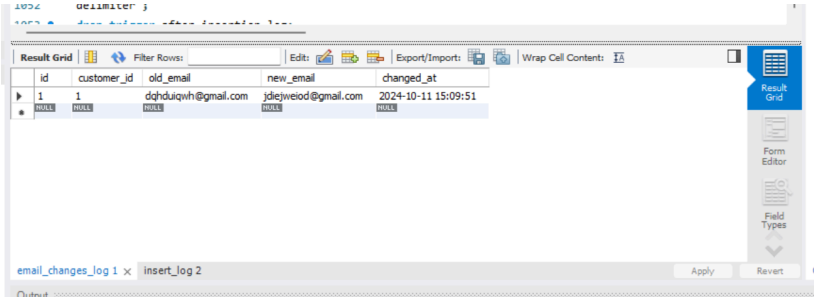
Example query -

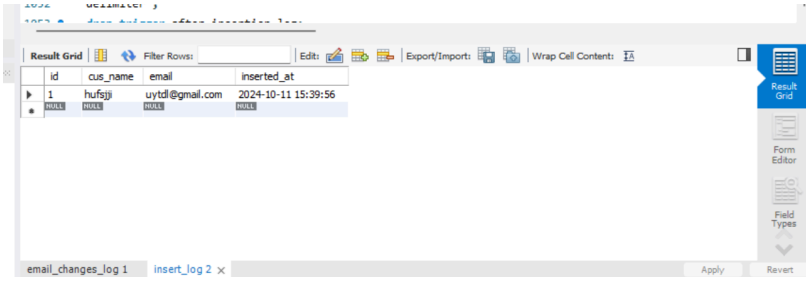
To log in table if there is any update on columns and also to log data of the new record inserted.

Using after and before key words.




Query 1
Limit to 10000 rows
DELIMITER //
CREATE TRIGGER log_changes
before update on customers
for each row
begin
if old.email!=new.email then
insert into email_changes_log(customer_id,old_email,new_email)
values(old.id,old.email, new.email);
end if;
end//
delimiter ;
delimiter //
create trigger after_insertion_log
after insert on customers
for each row
begin
insert into insert_log(cus_name,email) values (new.name,new.email);
end //
delimiter ;
drop trigger after_insertion_log;
Output
Search


Result Grid
Filter Rows:
Edit:
Export/Import:
Wrap Cell Content:
id
customer_id
old_email
new_email
changed_at
1
1
dqhduiqwh@gmail.com
jdiejweiod@gmail.com
2024-10-11 15:09:51
email_changes_log 1
insert_log 2
Apply
Revert


Result Grid
Filter Rows:
Edit:
Export/Import:
Wrap Cell Content:
id
cus_name
email
inserted_at
1
hufsjjj
uytdl@gmail.com
2024-10-11 15:39:56
email_changes_log 1
insert_log 2
Apply
Revert

The concepts on (14-10-2024) and (15-10-2024)

Spring Introduction -

1. Inversion of Control

Inversion of Control is a design principle used in software development, where the control of object creation and management is transferred from the application code to a framework or container. This promotes loose coupling and enhances modularity.

Key Concepts:

1. **Dependency Injection (DI):** A common form of IoC, where dependencies (objects that a class requires) are provided to the class rather than the class creating them itself. This can be done through:
    - Constructor Injection
    - Setter Injection
    - Field Injection

Benefits:

- **Decoupling:** Classes become less dependent on concrete implementations, making them easier to test and maintain.
- **Configuration Management:** Dependencies can be configured externally (e.g., in XML files or through annotations), allowing for easier changes.
- **Enhanced Testability:** Classes can be tested in isolation by mocking their dependencies.

Spring Framework and IoC:

- Spring utilizes IoC through its **ApplicationContext**. It manages the lifecycle and configuration of application objects (beans).
- Developers define beans and their dependencies, and Spring automatically wires them together based on the configuration.

Example -

```
// Using Constructor Injection in Spring

@Component

public class UserService {

   private final UserRepository userRepository;

   @Autowired

   public UserService(UserRepository userRepository) {
```

```
    this.userRepository = userRepository;
}}
```

2. Dependency Injection

Dependency Injection is a design pattern and a specific implementation of Inversion of Control (IoC) that allows a program to remove hard-coded dependencies and make the system more modular and testable. In DI, an object (often referred to as a client) receives its dependencies from an external source rather than creating them itself.

Ways of Dependency Injection -

1. Constructor Injection:

- Dependencies are provided through the class constructor.
- This approach ensures that the object is always in a valid state, as dependencies must be provided upon instantiation.

```
@Component
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

2. Setter Injection:

- Dependencies are provided through setter methods after the object is constructed.
- This allows for more flexible configurations, as dependencies can be changed or set after instantiation.

```
@Component
public class UserService {
    private UserRepository userRepository;

    @Autowired
```

```
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

3. Field Injection (not recommended):

- Dependencies are injected directly into the fields of the class using annotations like @Autowired.
- While convenient, it can lead to issues with immutability and makes testing more challenging since the dependencies cannot be changed without modifying the class itself.

```
@Component
public class UserService {
    @Autowired
    private UserRepository userRepository;
}
```

3. @Service -

This annotation is used to indicate that a class is a service component in the service layer of an application. It typically contains business logic.

Functionality:

- Component Scanning: When Spring scans the classpath for components, it picks up classes annotated with @Service and registers them as beans in the application context.
- Business Logic: Classes marked with @Service are typically used to encapsulate business logic and interact with the data layer (like repositories).
- Transactional Behavior: Often, service classes manage transactions (using @Transactional) to ensure that a series of operations either complete successfully or not at all.

Example -

```
@Service
public class WeatherService {
   // Business logic methods here
}
```

4. @RestController -

This is a specialized version of the @Controller annotation, which is used to define RESTful web services.

Functionality:

- Response Body: It combines @Controller and @ResponseBody, meaning that the methods in this class return data directly as JSON or XML (rather than a view).
- Endpoint Handling: It's commonly used in Spring MVC to create REST APIs, handling HTTP requests and returning data to the client.

Example -

```
@RestController
@RequestMapping("/api")
public class HelloWorldController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

5. @Configuration -

This annotation indicates that a class declares one or more @Bean methods, which Spring will use to instantiate beans in the application context.

Functionality:

- Bean Definition: Classes annotated with @Configuration are treated as a source of bean definitions. Each method annotated with @Bean will be called by the Spring container, and its return value will be registered as a bean.
- Dependency Injection: It helps in defining beans that can be injected into other components (like services or controllers) via Spring's dependency injection mechanism.

Example -

```
@Configuration
public class WeatherConfig {
    @Bean
```

```
    public WeatherService weatherService() {
        return new WeatherService();
    }
}
```

6. @RequestMapping -

This annotation is used to map web requests to specific handler methods in a controller. It can be applied at the class level or the method level.

Functionality:

- **Mapping HTTP Requests**: You can specify the URL path, HTTP method (GET, POST, etc.), and other parameters to define how requests should be handled.
- **Flexible Configuration**: You can use attributes like `produces`, `consumes`, `params`, and `headers` to fine-tune the request mapping.
- **Class-Level vs. Method-Level**: When used at the class level, it defines a base path for all methods in the controller. When used at the method level, it specifies the path for that particular method.

Example -

```
@RestController
@RequestMapping("/api") // Base path for all methods in this controller
public class HelloWorldController {

    @GetMapping("/hello") // Maps GET requests to /api/hello
    public String hello() {
        return "Hello, World!";
    }

    @PostMapping("/greet") // Maps POST requests to /api/greet
    public String greet(@RequestParam String name) {
        return "Hello, " + name + "!";
    }
}
```

7. @Validated -

This annotation is used to indicate that a class or method parameters should be validated against certain constraints. It integrates with the Java Bean Validation framework (like Hibernate Validator).

Functionality:

- Input Validation: When applied to a method or a class, it ensures that the parameters or fields are validated before the method is executed.
- Constraint Annotations: You can combine @Validated with constraint annotations (like @NotNull, @NotBlank, etc.) to enforce validation rules.
- Method-Level Validation: When used on a method, it validates the method parameters.

Example -

```
@RestController
@RequestMapping("/api")
@Validated // Enables validation for this controller
public class WeatherController {

    @GetMapping("/weather/{city}")
    public String getWeather(@PathVariable @NotBlank String city) {
        return "Weather data for " + city;
    }

    @PostMapping("/weather")
    public String updateWeather(@RequestParam @NotBlank String city,
                                @RequestParam @NotBlank String
condition) {
        // Update weather logic
        return "Weather updated for " + city;
    }
}
```

The concepts on (16-10-2024)

Views -

In MySQL, a **view** is a virtual table that is based on the result of a SELECT query. It allows you to simplify complex queries, encapsulate logic, and present data in a specific format without modifying the underlying tables. Here's an overview of views in MySQL:

Key Features of Views

1. **Virtual Table**: A view does not store data physically; it provides a way to present data from one or more tables.
2. **Simplification**: Complex queries can be encapsulated within a view, making it easier to query the data without repeating complex logic.
3. **Security**: Views can restrict access to specific rows and columns of a table, enhancing data security.
4. **Data Abstraction**: They provide an abstraction layer, allowing changes to the underlying tables without affecting the applications using the views.

Creating a View

To create a view, you use the CREATE VIEW statement. Here's the syntax:

Example:

```
CREATE VIEW active_users AS

SELECT id, username, email

FROM users

WHERE status = 'active';
```

Using Views

Once created, you can query a view like a regular table:

Query - SELECT * FROM active_users;

Updating Views

While you can use views to SELECT data, updates can be limited. MySQL allows updates to views under certain conditions, such as:

- The view must map to a single table.

- All columns in the view must be updatable.

Dropping a View

To remove a view, use the DROP VIEW statement:

DROP VIEW view_name;

Limitations of Views

- **Performance**: Views may impact performance, especially if they involve complex queries or multiple joins.
- **Non-Updatable Views**: Some views cannot be updated if they do not meet certain criteria.

Views Enhancing security -

Views enhance data security in several ways by restricting access to specific rows and columns of underlying tables. Here's how this works:

1. Row-Level Security

When creating a view, you can include a WHERE clause to filter data, allowing users to see only the rows that meet certain criteria. This means users can be granted access to a view without being able to access the entire table.

Example:

CREATE VIEW sales_rep AS

SELECT * FROM sales

WHERE sales_rep_id = CURRENT_USER_ID;

In this case, a sales representative can only see their own sales data through the sales_rep view.

2. Column-Level Security

You can define a view to include only specific columns from a table. This prevents users from seeing sensitive information that they don't need.

Example:

CREATE VIEW public_employee_info AS

SELECT employee_id, name, department

FROM employees;

In this case, users granted access to the public_employee_info view cannot see salary information or personal details, which might be in the original employees table.

3. Granular Access Control

By using views, we can define multiple views for different user roles or permissions. For instance, we might have:

- A view for general users that excludes sensitive data.
- A view for managers that includes additional columns.
- A view for administrators that provides full access.

4. Limited Permissions

You can grant users permissions on a view without granting them permissions on the underlying tables. This means they can perform SELECT operations on the view without being able to directly access or modify the base tables.

Example:

GRANT SELECT ON public_employee_info TO user_role;

Materialized Views -

Materialized views (MVs) are database objects that store the results of a query physically, rather than calculating them each time the view is accessed. Unlike a regular view, which fetches data dynamically from the underlying tables when queried, a materialized view saves the data as a snapshot, allowing for faster query performance, especially for complex or resource-intensive queries.

Key Characteristics of Materialized Views:

1. **Physical Storage**:
    - MVs store the result of the query on disk. This makes data retrieval faster, as the view doesn't need to recompute the results every time.
2. **Periodic Refresh**:
    - MVs can be **refreshed periodically** to reflect changes in the underlying data. This refresh can be scheduled at specific intervals or triggered manually.
3. **Improved Performance**:
    - Since the data is stored and not recalculated every time, querying a materialized view can be **much faster** for large datasets or complex queries, especially in analytical or reporting workloads.

Example use case -

Suppose we have a table with millions of rows, and we frequently run a query to calculate total sales by region. Instead of running the same aggregation query each time, you can create a materialized

view that stores the result of this query. This way, subsequent queries can directly access the precomputed results from the materialized view, significantly improving performance.

Query -

```
CREATE MATERIALIZED VIEW total_sales_by_region AS

SELECT region, SUM(sales) AS total_sales

FROM sales_data

GROUP BY region;


SELECT * FROM total_sales_by_region;
```

Refreshing a Materialized View:

To ensure the materialized view reflects the latest data, you can refresh it:

```
REFRESH MATERIALIZED VIEW total_sales_by_region;
```

Application using @Service, @Restcontroller, @Configuration, etc (Spring MVC)

This Spring Boot application provides a simple REST API to manage weather data for different cities. The main components include a controller for handling HTTP requests, a service for business logic, and a configuration class for weather settings.

Application Components

1. WeatherController

- Responsible for handling HTTP requests related to weather data.
- Annotated with @RestController to make it a Spring-managed controller that returns JSON responses.
- Mapped to the base URL /api/weather.

Key Endpoints:

- GET /api/weather/{city}: Fetches the weather for a specific city.
- POST /api/weather/{city}: Updates the weather for a specific city.
- GET /api/weather: Retrieves weather data for all cities.

```
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));
    }
```

2. WeatherService

- Contains the business logic for managing weather data.
- Uses a Map to store weather records for each city.

Key Methods:

- getWeather(String city): Retrieves weather data for a specified city, returning a default condition if not found.
- updateWeather(String city, int temperature): Updates the weather for a city. This method currently has an issue with using a single instance of WeatherRecord.

- getAllWeatherData(): Returns weather data for all cities in a formatted way.

3. WeatherRecord

- A simple Java class representing the weather data for a city, including properties like city name, temperature, and condition.
- Contains an initialize method to set the city and temperature.

Properties:

- String city
- int temperature
- String condition

```
public void initialize(String city, int temperature) {
        this.city = city;
        this.temperature = temperature;
    }
```

4. WeatherConfig

- A configuration class that holds default weather conditions and temperature ranges for different cities.
- Uses @ConfigurationProperties to map properties from application configuration files.

```
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;
        }
    }
}
```

The Issue with WeatherRecord

The current implementation of the updateWeather method in WeatherService has a critical issue. The use of a single WeatherRecord instance means that when the weather for one city is updated, it may overwrite data for another city.

Explanation of the Problem:

The method uses context.getBean(WeatherRecord.class) to obtain a WeatherRecord instance for a new city. However, if the same WeatherRecord instance is reused, updating one city's weather will affect others.

Solution:

To fix this issue, you should create a new WeatherRecord instance for each city instead of reusing the same instance. You could modify the updateWeather method to ensure each city gets its own record:

```
public void updateWeather(String city, int temperature) {
    WeatherRecord record = new WeatherRecord();
    record.initialize(city, temperature);
    record.setTemperature(temperature);
    record.setCondition(determineCondition(city, temperature));
    weatherData.put(city, record);
}
```

The concepts on (17-10-2024)

Breakout Room Activity

Practice of SQL Queries -

1. Leetcode (577 - Employee Bonus)

I needed to extract information from two related tables: Employee and Bonus. The goal was to report the names and bonus amounts of employees who received a bonus of less than 1000.

Step-by-Step Solution -

- The Employee table contains employee details, including their unique empId, name, supervisor, and salary.
- The Bonus table contains the empId of employees and their respective bonus amounts.
- I need to join these two tables based on the empId to link each employee with their bonus.
- The final output should only include employees whose bonuses are below 1000.
- To achieve this, I used an INNER JOIN to combine the two tables on the empId.
- I then filtered the results using a WHERE clause to ensure only bonuses less than 1000 were included.

Query -

```
SELECT e.name, b.bonus FROM Employee e JOIN Bonus b ON e.empId = b.empId
WHERE b.bonus < 1000;
```

2. Leetcode problem - 1280

I was required to determine the number of times each student attended exams for each subject. This involved working with three tables: Students, Subjects, and Examinations.

Step-by-Step Solution

- The Students table contains unique entries for each student, identified by student_id and their student_name.
- The Subjects table lists unique subjects by their subject_name.
- The Examinations table logs each instance where a student attended an exam for a particular subject, potentially including duplicate entries for multiple attempts.
- I needed to count how many times each student attended an exam for each subject. The result should include the student_id, subject_name, and the count of attendances, ordered by student_id and subject_name.

- I utilized the GROUP BY clause to group the records by student_id and subject_name, allowing me to count occurrences of each combination.
- I used the COUNT(*) function to tally the number of exam attendances for each group.

Query -

```
SELECT e.student_id, e.subject_name, COUNT(*) AS attendance_count FROM
Examinations e GROUP BY e.student_id, e.subject_name ORDER BY e.student_id,
e.subject_name;
```

3. Leetcode - 1934

the goal was to calculate the confirmation rate for users based on their signup and confirmation actions. The relevant tables were Signups and Confirmations.

Step-by-Step Solution -

- The Signups table contains each user's signup information, including their unique user_id and the time_stamp of their signup.
- The Confirmations table records confirmation requests for each user, including their user_id, the time_stamp of the request, and the action (either 'confirmed' or 'timeout').
- The confirmation rate is defined as the ratio of confirmed messages to the total number of requested confirmation messages. If a user has not requested any confirmations, their rate should be recorded as 0.
- The result should display each user and their corresponding confirmation rate, rounded to two decimal places.
- I used a LEFT JOIN to combine the Signups and Confirmations tables based on user_id. This allowed me to include users who might not have any confirmation records.
- I then utilized a GROUP BY clause on user_id to calculate the number of 'confirmed' actions and the total number of requests.
- Finally, I computed the confirmation rate using a CASE statement to handle users without any confirmation requests and used ROUND to ensure the rate is formatted to two decimal places.

Query -

```
SELECT s.user_id,

ROUND(COALESCE(SUM(CASE WHEN c.action = 'confirmed' THEN 1 ELSE 0 END), 0) *
1.0 /NULLIF(COUNT(c.action), 0), 2) AS confirmation_rate

FROM Signups s LEFT JOIN Confirmations c ON s.user_id = c.user_id

GROUP BY s.user_id;
```

4. Leetcode - 1075

I needed to compute the average years of experience for employees working on each project. This involved working with the Project and Employee tables.

Step-by-Step Solution -

- The Project table contains records linking employees to projects through project_id and employee_id.
- The Employee table holds details about each employee, including their employee_id, name, and experience_years.
- The goal was to calculate the average experience years for employees assigned to each project and round the result to two decimal places.
- I used an INNER JOIN to combine the Project and Employee tables based on employee_id.
- Then, I applied the AVG() function to calculate the average experience for employees grouped by project_id.
- Finally, I utilized the ROUND() function to round the average experience to two decimal places.

Query -

```
SELECT p.project_id, ROUND(AVG(e.experience_years), 2) AS
average_experience_years

FROM Project p JOIN Employee e ON p.employee_id = e.employee_id

GROUP BY p.project_id;
```

5. Leetcode - 1193

I needed to summarize transaction data from a table that records incoming transactions. The goal was to find, for each month and country, the total number of transactions and their total amount, as well as the count and amount of approved transactions.

Step-by-Step Solution -

- The Transactions table contains details of each transaction, including id, country, state, amount, and trans_date. The state column indicates whether a transaction is "approved" or "declined".
- The task required calculating:
  - The total number of transactions and their total amount for each month and country.
  - The number of approved transactions and their total amount for the same grouping.

- I used the DATE_FORMAT function to extract the year and month from the trans_date for grouping purposes.
- I used GROUP BY to aggregate the data based on year, month, and country.
- The SUM() and COUNT() functions were utilized to compute the required totals for transactions and approved transactions.

Query -

```
SELECT

DATE_FORMAT(trans_date, '%Y-%m') AS month, country, COUNT(*) AS
total_transactions,

SUM(amount) AS total_amount, COUNT(CASE WHEN state = 'approved' THEN 1 END)
AS approved_transactions,

SUM(CASE WHEN state = 'approved' THEN amount ELSE 0 END) AS approved_amount

FROM Transactions

GROUP BY month, country;
```

6. Leetcode - 2356

I needed to determine how many unique subjects each teacher teaches within a university. The relevant table for this data was the Teacher table.

Step-by-Step Solution -

- The Teacher table contains columns for teacher_id, subject_id, and dept_id. The combination of subject_id and dept_id serves as the primary key, indicating that each teacher can teach multiple subjects across different departments.
- The objective was to calculate the number of unique subjects for each teacher. Since the subject_id may appear multiple times for different departments, we need to ensure that each subject is counted only once per teacher.
- I used the COUNT(DISTINCT subject_id) function to count unique subjects taught by each teacher.
- The GROUP BY clause was used to aggregate the results by teacher_id.

Query -

```
SELECT

teacher_id,

COUNT(DISTINCT subject_id) AS unique_subject_count

FROM Teacher

GROUP BY teacher_id;
```

The concepts on (18-10-2024)

SQL Practice -

```
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)
);
```

1. Ranks Employees Based on Projects Completed

I wanted to determine how many projects each employee has completed within their department. To do this, I joined the Employees, Projects, and Departments tables using the appropriate foreign keys. I counted the number of projects per employee with COUNT(p.ProjectID).

Then, to rank the employees within each department based on this count, I used the RANK() function with a PARTITION BY clause for DepartmentID. This way, each department's ranking is calculated separately. The result gives me a clear view of employee performance in terms of project completion.

Query -

```
SELECT e.EmployeeName, d.DepartmentName, COUNT(p.ProjectID) AS ProjectCount, RANK() OVER
(PARTITION BY e.DepartmentID ORDER BY COUNT(p.ProjectID) DESC) AS Rank

FROM Employees e JOIN Projects p ON e.EmployeeID = p.EmployeeID JOIN Departments d ON
e.DepartmentID = d.DepartmentID

GROUP BY e.EmployeeID, e.EmployeeName, d.DepartmentName;
```

2. Calculates Average Project Duration for Each Department

For this query, I aimed to find out the average project duration for each department. I calculated the duration of each project by subtracting the StartDate from the EndDate using the DATEDIFF() function.

I then joined the Projects table with the Departments table to get the corresponding department names. Finally, I grouped the results by DepartmentName and used the AVG() function to calculate the average duration for all projects within each department. This gives insights into how long projects typically take in different departments.

Query -

```
SELECT d.DepartmentName, AVG(DATEDIFF(p.EndDate, p.StartDate)) AS AvgProjectDuration

FROM Projects p

JOIN Departments d ON p.DepartmentID = d.DepartmentID

GROUP BY d.DepartmentName;
```

3. Identifies Projects Longer Than Department's Average

Here, I wanted to find projects that took longer than the average duration for their respective departments. To achieve this, I first created a Common Table Expression (CTE) named AvgDuration, where I calculated the average project duration for each department.

Then, I joined this CTE with the Projects and Departments tables to filter out only those projects whose duration exceeded the average duration for their department. This way, I could identify projects that might need further attention or evaluation based on their length.

Query -

```
WITH AvgDuration AS (
   SELECT DepartmentID, AVG(DATEDIFF(EndDate, StartDate)) AS AvgProjectDuration
   FROM Projects
   GROUP BY DepartmentID
)
SELECT p.ProjectName, d.DepartmentName, DATEDIFF(p.EndDate, p.StartDate) AS ProjectDuration

FROM Projects p

JOIN Departments d ON p.DepartmentID = d.DepartmentID

JOIN AvgDuration a ON p.DepartmentID = a.DepartmentID

WHERE DATEDIFF(p.EndDate, p.StartDate) > a.AvgProjectDuration;
```

4. Finds Top 3 Most Efficient Employees by Average Project Duration

In this query, my goal was to find the top 3 most efficient employees in terms of project duration within each department. I calculated the average project duration for each employee by joining the necessary tables and using AVG(DATEDIFF(EndDate, StartDate)).

I then ranked the employees within each department using the RANK() function and filtered the results to show only those with a rank of 3 or lower. This query highlights the employees who manage to complete their projects in the shortest time frames, showcasing efficiency.

Query -

```
SELECT EmployeeName, DepartmentName, AVG(DATEDIFF(EndDate, StartDate)) AS
AvgProjectDuration,
    RANK() OVER (PARTITION BY e.DepartmentID ORDER BY AVG(DATEDIFF(EndDate, StartDate)) ASC)
AS Rank
FROM Employees e
JOIN Projects p ON e.EmployeeID = p.EmployeeID
JOIN Departments d ON e.DepartmentID = d.DepartmentID
GROUP BY e.EmployeeID, e.EmployeeName, d.DepartmentName
HAVING Rank <= 3;
```

5. Compares Each Project's Duration to the Previous Project in the Same Department

For this query, I wanted to compare the duration of each project with the previous project within the same department. I started by creating a CTE called ProjectDurations, where I calculated each project's duration and assigned a row number based on their start date within their department.

Then, I performed a self-join on this CTE to access both the current project and the previous project by aligning their row numbers. This allowed me to compare the durations side by side, providing insights into project timelines and any variations between them.

Query -

```
WITH ProjectDurations AS (
  SELECT ProjectID, DepartmentID, ProjectName,
      DATEDIFF(EndDate, StartDate) AS ProjectDuration,
      ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY StartDate) AS rn
  FROM Projects
)
SELECT curr.ProjectName AS CurrentProject, curr.ProjectDuration AS CurrentDuration,
    prev.ProjectName AS PreviousProject, prev.ProjectDuration AS PreviousDuration
```

```
FROM ProjectDurations curr
LEFT JOIN ProjectDurations prev ON curr.DepartmentID = prev.DepartmentID AND curr.rn = prev.rn + 1;
```

Tables -

```
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)
);
```

1. Top 5 Customers by Total Spending

Query:

```
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;
```

Explanation: In this query, I wanted to identify the top 5 customers based on their total spending. I started by joining the Customers table with the Orders table using CustomerID. Then, I summed the TotalAmount from the Orders table for each customer using SUM(o.TotalAmount).

To get the top spenders, I grouped the results by CustomerID and Name, ordered them in descending order of TotalSpending, and limited the results to the top 5 customers. This gives a clear picture of who the highest-value customers are.

2. Monthly Sales Trend for the Past 6 Months

Query:

```
SELECT DATE_FORMAT(OrderDate, '%Y-%m') AS Month, SUM(TotalAmount) AS TotalSales
FROM Orders
WHERE OrderDate >= DATE_SUB(CURDATE(), INTERVAL 6 MONTH)
GROUP BY Month
ORDER BY Month;
```

Here, I aimed to analyze the sales trend over the past 6 months. I selected the OrderDate from the Orders table and formatted it to show only the year and month using DATE_FORMAT.

I filtered the results to include only those orders placed in the last 6 months using the WHERE clause. After that, I summed the TotalAmount for each month and grouped the results accordingly. This gives a monthly overview of sales, helping to identify trends over time.

3. Best-Selling Product Category in Each Month

Query:

```
SELECT DATE_FORMAT(o.OrderDate, '%Y-%m') AS Month, p.Category, SUM(od.Quantity) AS TotalSold
FROM OrderDetails od
JOIN Orders o ON od.OrderID = o.OrderID
JOIN Products p ON od.ProductID = p.ProductID
GROUP BY Month, p.Category
ORDER BY Month, TotalSold DESC;
```

In this query, I wanted to find out which product category sold the most each month. I joined the OrderDetails, Orders, and Products tables to access the necessary data.

I formatted the OrderDate to capture the year and month, then summed the Quantity sold for each category using SUM(od.Quantity). By grouping the results by month and product category, I was able to see the total units sold, ordered by month and total sold in descending order. This highlights the best-selling categories for each month.

4. Customers Who Haven't Made a Purchase in the Last 3 Months

Query:

```
SELECT c.CustomerID, c.Name
FROM Customers c
LEFT JOIN Orders o ON c.CustomerID = o.CustomerID AND o.OrderDate >= DATE_SUB(CURDATE(), INTERVAL 3 MONTH)
WHERE o.OrderID IS NULL;
```

For this query, I aimed to identify customers who haven’t made any purchases in the last 3 months. I used a LEFT JOIN to connect the Customers table with the Orders table, filtering to only include orders from the last 3 months.

The key part of the query is the WHERE clause, which checks for NULL in o.OrderID. This indicates that there were no matching orders within that timeframe, thus identifying customers who haven't purchased recently. This information can be useful for targeted marketing efforts.

5. Average Order Value and Number of Orders for Each Customer Segment

Query:

```
SELECT c.Segment, COUNT(o.OrderID) AS NumberOfOrders, AVG(o.TotalAmount) AS
AverageOrderValue

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

GROUP BY c.Segment;
```

In this query, I wanted to analyze the average order value and the number of orders for each customer segment. By joining the Customers and Orders tables on CustomerID, I could aggregate the necessary data.

I counted the OrderID to get the total number of orders per segment using COUNT(o.OrderID) and calculated the average order value with AVG(o.TotalAmount). Grouping by Segment allows me to see how different customer segments behave in terms of order frequency and value, providing insights for better targeting.

Calculate the total sales,daily 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.

CREATE TABLE employee_sales (employee_id INT PRIMARY KEY, employee_name VARCHAR(50), department VARCHAR(50), sales_amount DECIMAL(10, 2), sales_date DATE);

Query -

```
SELECT

   department,SUM(sales_amount) AS TotalSales,

   SUM(sales_amount) / COUNT(DISTINCT sales_date) AS DailyAverageSales,

   COUNT(DISTINCT employee_id) AS NumberOfEmployees,

   RANK() OVER (ORDER BY SUM(sales_amount) DESC) AS SalesRank

FROM employee_sales

GROUP BY department

ORDER BY TotalSales DESC;
```

My objective was to calculate and display total sales, daily average sales, and the number of employees for each department, while also ranking the departments based on their total sales.

- I selected the department to group the results accordingly.
- I calculated the total sales for each department using SUM(sales_amount) and labeled it as TotalSales.
- To find the daily average sales, I divided the total sales by the number of distinct sales dates with SUM(sales_amount) / COUNT(DISTINCT sales_date). This gives me the average sales per day and is aliased as DailyAverageSales.
- I counted the number of distinct employees in each department using COUNT(DISTINCT employee_id) to avoid duplicates, naming it NumberOfEmployees.
- I applied the RANK() function to rank the departments based on their total sales in descending order. This allows us to see which departments are performing best in terms of sales.
- The results are grouped by department to ensure that calculations are done for each department individually.
- Finally, I ordered the entire result set by TotalSales in descending order to display the departments from highest to lowest sales.

The concepts on (21-10-2024)

1. Primary key(@Id annotation)

The @Id annotation is used to define the primary key of an entity. The primary key uniquely identifies each record in the table.

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Optional, auto-generates the value
    private Long id;


    private String name;
    private double price;
}
```

@Id: Marks the id field as the primary key of the entity.

@GeneratedValue: Specifies the generation strategy for the primary key (e.g., AUTO, IDENTITY, SEQUENCE, TABLE). It can be omitted if you want to manually set the primary key values.

2. Foreign Key (@ManyToOne, @JoinColumn)

To establish relationships between tables, like a foreign key, you can use annotations like @ManyToOne, @OneToMany, and @JoinColumn. These annotations help define how entities are related, and a foreign key in one entity can point to the primary key of another entity.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderId;
```

```
    @ManyToOne // Many orders can belong to one customer
    @JoinColumn(name = "customer_id", referencedColumnName = "id") //
Specifies the foreign key column
    private Customer customer;


    private LocalDate orderDate;
}
```

- @ManyToOne: Defines a many-to-one relationship. In this case, multiple Order entities can be associated with one Customer.
- @JoinColumn(name = "customer_id"): Specifies the name of the foreign key column in the Order table. The referencedColumnName attribute indicates which column in the Customer table this foreign key references (typically the primary key of the referenced entity).

For one-to-many or many-to-many relationships, you can use @OneToMany or @ManyToMany annotations similarly, but the core idea of foreign keys usually involves the @ManyToOne side of the relationship.

JoinColumn annotation -

1. Basic Usage of @JoinColumn

When you establish a relationship between two entities (like ManyToOne, OneToOne, etc.), the @JoinColumn annotation is used to define the foreign key column in the entity table that stores the foreign key.

For example, consider an Order entity that references a Customer entity through a foreign key:

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderId;


    @ManyToOne
    @JoinColumn(name = "customer_id", referencedColumnName = "id") //
Defines the foreign key column
    private Customer customer;
```

```
    private LocalDate orderDate;
}
```

In this case:

- @JoinColumn(name = "customer_id"): This specifies that the customer_id column in the Order table will be used as the foreign key to the Customer table.
- referencedColumnName = "id": This specifies that the foreign key column customer_id references the id column in the Customer table.

2. Attributes of @JoinColumn

name

- Description: The name of the foreign key column in the child entity's table (the owning entity).
- Example: @JoinColumn(name = "customer_id")

referencedColumnName

- Description: The name of the primary key (or other unique key) column in the parent entity that the foreign key references.
- Example: @JoinColumn(name = "customer_id", referencedColumnName = "id")
    - Here, customer_id in the Order table refers to the id column in the Customer table.
    - If referencedColumnName is not provided, JPA assumes that the referenced column is the primary key of the referenced entity.

nullable

- Description: Indicates whether the foreign key column can accept NULL values.
- Default: true (the foreign key column allows NULL values by default).
- Example: @JoinColumn(name = "customer_id", nullable = false)
    - This will make the foreign key customer_id non-nullable (i.e., a customer is mandatory for every order).

unique

- Description: Indicates whether the foreign key column should have a unique constraint. This is typically used in one-to-one relationships.
- Default: false
- Example: @JoinColumn(name = "customer_id", unique = true)
    - This ensures that each customer_id appears only once in the Order table.

insertable and updatable

- Description: Determines whether the foreign key column should be included in SQL INSERT and UPDATE statements.
- Default: Both are true.
- Example: @JoinColumn(name = "customer_id", insertable = false, updatable = false)
    - This would mean that the customer_id foreign key column is read-only and cannot be inserted or updated.

3. Transactional annotation -

The @Transactional annotation in Spring is a powerful feature that manages transaction boundaries declaratively. It simplifies handling transactions in a Spring application by marking a method or class with the annotation to control the behavior of the transaction.

i) Transaction

A transaction is a sequence of operations performed as a single unit of work. It ensures the ACID (Atomicity, Consistency, Isolation, Durability) properties of a database:

- Atomicity: All or nothing execution of operations.
- Consistency: The transaction moves the system from one consistent state to another.
- Isolation: Transactions operate in isolation, avoiding conflicts with other transactions.
- Durability: Once a transaction is committed, its effects persist even in the event of a failure.

In Spring, @Transactional helps manage transactions at the method or class level by automatically starting and committing or rolling back the transaction based on the method execution.

ii) @Transactional Uses and Features

The @Transactional annotation can be applied in different scopes, typically at the service layer. Here's a breakdown of its uses:

a. Transaction Propagation

@Transactional allows controlling how transactions behave when calling a method that is already in a transaction.

Propagation types:

- **REQUIRED** (default): If a transaction exists, join it; if not, create a new one.
- **REQUIRES_NEW**: Suspends any existing transaction and creates a new one.
- **NESTED**: Creates a nested transaction if a current one exists, allowing partial rollbacks.
- **MANDATORY**: Requires an existing transaction, throws an exception if there isn't one.
- **SUPPORTS**: If a transaction exists, join it; otherwise, execute non-transactionally.
- **NOT_SUPPORTED**: Always execute non-transactionally, suspending any existing transaction.
- **NEVER**: Throws an exception if a transaction exists, enforcing non-transactional behavior.

b. Transaction Isolation

Isolation level controls the visibility of changes made by one transaction to other concurrent transactions. Spring supports several isolation levels:

- **READ_UNCOMMITTED**: Allows reading uncommitted changes from other transactions.
- **READ_COMMITTED**: Can only read committed changes, preventing dirty reads.
- **REPEATABLE_READ**: Ensures that if data is read twice within a transaction, the result will be the same.
- **SERIALIZABLE**: Ensures complete isolation by serializing transactions, preventing all possible conflicts.

c. Transaction Timeout

Specifies the maximum time a transaction is allowed to run before it is automatically rolled back.

```
@Transactional(timeout = 30) // Transaction will timeout after 30 seconds
public void someMethod() {
   // Business logic
}
```

d. Read-Only Transactions

Transactions marked as read-only can optimize performance when no modifications are needed.

```
@Transactional(readOnly = true) // No database modification allowed
public List<Customer> getCustomers() {
   return customerRepository.findAll();
}
```

e. Rollback Conditions

You can specify which exceptions trigger a rollback. By default, any runtime exception (unchecked) triggers a rollback, while checked exceptions do not. However, this behavior can be customized.

```
@Transactional(rollbackFor = Exception.class) // Rollback for any exception
public void updateInventory() {
   // Business logic
```

}

f. Commit or Rollback

- Commit: If a method annotated with @Transactional completes successfully, the transaction is committed.
- Rollback: If an exception occurs within the method, the transaction is rolled back to avoid inconsistent data.

Example -

We are managing a bank transfer operation between two accounts. This operation should ensure that both accounts are updated as a single transaction—either both accounts are updated, or neither is.

```
@Service
public class BankService {
    @Autowired
    private AccountRepository accountRepository;

    @Transactional
    public void transferMoney(Long fromAccountId, Long toAccountId,
Double amount) {
        Account fromAccount =
accountRepository.findById(fromAccountId).orElseThrow(() -> new
AccountNotFoundException());
        Account toAccount =
accountRepository.findById(toAccountId).orElseThrow(() -> new
AccountNotFoundException());

        if (fromAccount.getBalance() < amount) {
            throw new InsufficientFundsException("Not enough balance");
        }
        fromAccount.setBalance(fromAccount.getBalance() - amount);
        toAccount.setBalance(toAccount.getBalance() + amount);
        accountRepository.save(fromAccount); // First update
        accountRepository.save(toAccount);   // Second update
```

```
    }

}
```

1. The @Transactional annotation ensures that if an exception occurs during the execution of the transferMoney method (e.g., InsufficientFundsException), both account updates are rolled back. No changes are committed to the database.
2. If everything is successful, the transaction will be committed, meaning both account balances will be updated.
3. The default propagation behavior (Propagation.REQUIRED) is applied, meaning if there's an existing transaction, it will be joined; otherwise, a new one is started.

Rollback Example

We can also specify which exceptions trigger a rollback and which do not.

```
@Transactional(rollbackFor = { InsufficientFundsException.class },
noRollbackFor = { AccountNotFoundException.class })
public void transferMoney(Long fromAccountId, Long toAccountId, Double
amount) {
}
```

In this case:

- Rollback will be triggered if an InsufficientFundsException is thrown.
- Rollback will not happen for an AccountNotFoundException.

The concepts on (22-10-2024) and (23-10-2024)

Validation Annotations -

Spring Boot provides a robust validation framework that simplifies the process of validating user inputs in web applications. Validation annotations are part of the Java Bean Validation (JSR 380) specification and are commonly used to enforce constraints on the properties of Java objects.

Common Validation Annotations -

1. @NotNull

Ensures that the annotated element is not null.

```
public class User {
    @NotNull
    private String name;
}
```

2. @Size

Specifies the size constraints for character sequences (e.g., String).

Can define minimum and maximum lengths.

```
public class User {
    @Size(min = 2, max = 30)
    private String username;
}
```

3. @Min and @Max

Used for numerical values to specify minimum and maximum values.

```
public class Product {
    @Min(0)
    @Max(100)
    private int quantity;
```

}

4. @Email

Validates that the annotated string is a valid email format.
public class User {

```
@Email
private String email;
```

}

5. @Pattern

Validates that the annotated string matches a specified regular expression.

public class User {

```
@Pattern(regexp = "^[A-Za-z0-9]{5,12}$")
private String username;
```

}

6. @Future and @Past

Validates that a date is in the future or in the past, respectively.
public class Event {

```
@Future
private LocalDate eventDate;
```

}

7. @Valid

Used for cascading validation of nested objects. When applied to a property, it triggers validation on the related object.

public class User {

```
@Valid
private Address address;
```

}

Using Validation Annotations in Controllers

To leverage these annotations in a Spring Boot application, you typically use them in conjunction with Spring MVC controller methods. Here's a simple example:

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<User> createUser(@Valid @RequestBody User user) {
        // Handle user creation
        return ResponseEntity.ok(user);
    }
}
```

In this example, the @Valid annotation triggers validation before the createUser method is executed. If validation fails, a MethodArgumentNotValidException is thrown, which can be handled globally or within the controller.

Handling Validation Errors

Spring Boot provides an easy way to handle validation errors by using the `@ControllerAdvice` annotation to define a global exception handler.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Object> handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error -> {
            errors.put(error.getField(), error.getDefaultMessage());
```

```
        });

        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);

    }

}
```

This handler captures validation errors and returns a structured response with the error messages.

Scope in Spring -

In Spring, a "scope" defines the lifecycle and visibility of a bean in the application context. Understanding the various scopes is crucial for managing bean instances effectively. Here are the main scopes provided by Spring, along with explanations and examples for each.

1. Singleton Scope

- **Description**: This is the default scope. A single instance of the bean is created and shared across the entire Spring container. The same instance is returned each time the bean is requested.

```
@Component

public class SingletonBean {

    public SingletonBean() {

        System.out.println("SingletonBean instance created");

    }



    public void display() {

        System.out.println("SingletonBean method called");

    }

}
```

2. Prototype Scope

- **Description**: A new instance of the bean is created each time it is requested from the Spring container. This is useful when you want a fresh instance of a bean for each use.

```java
@Component

@Scope("prototype")

public class PrototypeBean {

    public PrototypeBean() {

        System.out.println("PrototypeBean instance created");

    }



    public void display() {

        System.out.println("PrototypeBean method called");

    }

}
```

3. Request Scope

- **Description**: A new instance of the bean is created for each HTTP request. This scope is available only in a web-aware Spring ApplicationContext.

```java
@Component

@Scope(value = WebApplicationContext.SCOPE_REQUEST)

public class RequestBean {

    public RequestBean() {

        System.out.println("RequestBean instance created");

    }



    public void display() {

        System.out.println("RequestBean method called");

    }
```

```
}
```

4. Session Scope

- **Description**: A new instance of the bean is created for each HTTP session. This scope is also available only in a web-aware Spring ApplicationContext.

```
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION)
public class SessionBean {
    public SessionBean() {
        System.out.println("SessionBean instance created");
    }

    public void display() {
        System.out.println("SessionBean method called");
    }
}
```

For Example -

```
WeatherRecord.java
package com.example.weather.model;

public class WeatherRecord {
    private String city;
    private double temperature;
    private String condition;

    public WeatherRecord(String city, double temperature, String
condition) {
        this.city = city;
        this.temperature = temperature;
```

```
        this.condition = condition;
    }

    // Getters and Setters
    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
    public double getTemperature() { return temperature; }
    public void setTemperature(double temperature) { this.temperature =
temperature; }
    public String getCondition() { return condition; }
    public void setCondition(String condition) { this.condition =
condition; }
}
// WeatherService.java
package com.example.weather.service;

import org.springframework.stereotype.Service;
import java.util.HashMap;
import java.util.Map;

@Service  // Singleton by default
public class WeatherService {
    private Map<String, WeatherRecord> weatherData = new HashMap<>();
    private int totalRequests = 0;  // Shared counter for all users

    public void updateWeather(String city, double temperature, String
condition) {
        weatherData.put(city, new WeatherRecord(city, temperature,
condition));
        totalRequests++;
```

```
    }

    public WeatherRecord getWeather(String city) {
        totalRequests++;
        return weatherData.get(city);
    }

    public int getTotalRequests() {
        return totalRequests;
    }
}
// WeatherQuery.java
package com.example.weather.service;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import java.time.LocalDateTime;

@Component
@Scope("prototype")
public class WeatherQuery {
    private final String queryId =
java.util.UUID.randomUUID().toString();
    private final LocalDateTime queryTime = LocalDateTime.now();

    public String getQueryId() { return queryId; }
    public LocalDateTime getQueryTime() { return queryTime; }
}
// WeatherQuery.java
```

```java
package com.example.weather.service;


import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import java.time.LocalDateTime;


@Component

@Scope("prototype")

public class WeatherQuery {

    private final String queryId =
java.util.UUID.randomUUID().toString();

    private final LocalDateTime queryTime = LocalDateTime.now();


    public String getQueryId() { return queryId; }

    public LocalDateTime getQueryTime() { return queryTime; }

}

// WeatherRequestLogger.java

package com.example.weather.service;


import org.springframework.web.context.annotation.RequestScope;

import org.springframework.stereotype.Component;

import java.time.LocalDateTime;


@Component

@RequestScope

public class WeatherRequestLogger {

    private final String requestId =
java.util.UUID.randomUUID().toString();

    private final LocalDateTime requestTime = LocalDateTime.now();

    private String cityRequested;
```

```
    public void logRequest(String city) {
        this.cityRequested = city;
    }

    public String getRequestInfo() {
        return String.format("Request ID: %s, Time: %s, City: %s",
            requestId, requestTime, cityRequested);
    }
}
// UserWeatherPreferences.java
package com.example.weather.service;

import org.springframework.web.context.annotation.SessionScope;
import org.springframework.stereotype.Component;
import java.util.*;

@Component
@SessionScope
public class UserWeatherPreferences {
    private final String sessionId =
java.util.UUID.randomUUID().toString();
    private final List<String> recentSearches = new ArrayList<>();
    private String temperatureUnit = "Celsius";

    public void addSearch(String city) {
        if (recentSearches.size() >= 5) {
            recentSearches.remove(0);
        }
```

```
        recentSearches.add(city);
    }

    public List<String> getRecentSearches() {
        return new ArrayList<>(recentSearches);
    }

    public String getSessionId() {
package com.example.weather.controller;

import org.springframework.web.bind.annotation.*;
import org.springframework.context.ApplicationContext;
import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/weather")
public class WeatherController {
    private final WeatherService weatherService;              //
Singleton
    private final WeatherRequestLogger requestLogger;         //
Request Scope
    private final UserWeatherPreferences userPreferences;     //
Session Scope
    private final ApplicationContext applicationContext;      // For
Prototype

    public WeatherController(
            WeatherService weatherService,
            WeatherRequestLogger requestLogger,
```

```
            UserWeatherPreferences userPreferences,
            ApplicationContext applicationContext) {
        this.weatherService = weatherService;
        this.requestLogger = requestLogger;
        this.userPreferences = userPreferences;
        this.applicationContext = applicationContext;
    }

    @GetMapping("/{city}")
    public Map<String, Object> getWeather(@PathVariable String city) {
        // Request scope - log the request
        requestLogger.logRequest(city);

        // Session scope - add to recent searches
        userPreferences.addSearch(city);

        // Prototype scope - create new query
        WeatherQuery query =
applicationContext.getBean(WeatherQuery.class);

        // Singleton - get weather data
        WeatherRecord weather = weatherService.getWeather(city);

        Map<String, Object> response = new HashMap<>();
        response.put("weather", weather);
        response.put("requestInfo", requestLogger.getRequestInfo());
        response.put("recentSearches",
userPreferences.getRecentSearches());
        response.put("queryInfo", Map.of(
            "queryId", query.getQueryId(),
```

```
            "queryTime", query.getQueryTime()
        ));
        response.put("totalRequests",
weatherService.getTotalRequests());

        return response;
    }

    @PostMapping("/{city}")
    public Map<String, Object> updateWeather(
            @PathVariable String city,
            @RequestParam double temperature,
            @RequestParam String condition) {
        weatherService.updateWeather(city, temperature, condition);
        return getWeather(city);
    }

    @PostMapping("/preferences/unit")
    public Map<String, String> setUnit(@RequestParam String unit) {
        userPreferences.setTemperatureUnit(unit);
        return Map.of(
            "sessionId", userPreferences.getSessionId(),
            "unit", userPreferences.getTemperatureUnit()
        );
    }
}
```

1. The WeatherService class is marked with the @Service annotation, which makes it a singleton by default. This means that only one instance of this bean is created and shared across the entire application context.

Usage: Since it maintains a map of weather records and a shared counter for total requests, a singleton scope is appropriate. This ensures that all parts of the application reference the same weather data and total request count, providing a consistent state.

2. The WeatherQuery class is annotated with @Scope("prototype"), meaning that a new instance is created each time the bean is requested from the application context.

Usage: This is beneficial for the WeatherQuery because it needs to generate unique identifiers (queryId) and timestamps (queryTime) for each weather query made. By having a prototype scope, each request for a WeatherQuery results in a fresh object, ensuring that no state is shared between different queries.

3. The WeatherRequestLogger class uses @RequestScope, which means a new instance is created for each HTTP request. This instance lives only for the duration of the request.

Usage: This is ideal for logging request-specific information, such as the requestId, requestTime, and the city requested. Since each request may involve different cities, having a separate instance allows the application to track details specific to that request without interference from others.

4. The UserWeatherPreferences class is annotated with @SessionScope, meaning that a single instance is created for each user session. This instance will persist for the duration of that session.

Usage: This scope is used to track user-specific preferences, such as recent searches and the preferred temperature unit. By keeping this data tied to the session, it allows for a personalized experience as users interact with the application, ensuring that their preferences are maintained throughout their session.

Spring Security -

Spring Security is a powerful and customizable authentication and access control framework for Java applications, primarily built on the Spring Framework. It provides comprehensive security services for Java EE-based enterprise software applications.

Core Concepts

1. **Authentication**:
    - The process of verifying the identity of a user or system. In Spring Security, this involves checking user credentials (like username and password) against stored values.
2. **Authorization**:
    - This determines what an authenticated user is allowed to do (e.g., accessing certain URLs or performing specific actions).
3. **Filters**:
    - Spring Security uses a filter-based architecture where various filters can be applied to incoming requests. These filters handle tasks such as authentication and authorization checks.

4. **Security Context**:
    - This holds the security-related information for the current user. After authentication, user details are stored in the `SecurityContext`, which can be accessed throughout the application.
5. **UserDetailsService**:
    - This interface is used to retrieve user-related data. It typically loads user details from a database.

Key Features

1. **Authentication**:
    - Supports various authentication methods, including form-based login, basic authentication, OAuth2, LDAP, and more.
2. **Authorization**:
    - Provides fine-grained access control with role-based and method-level security. You can specify permissions for different user roles.
3. **CSRF Protection**:
    - Built-in protection against Cross-Site Request Forgery attacks, which can compromise user sessions.
4. **Session Management**:
    - Manages user sessions, including session fixation protection and concurrent session control.
5. **Password Storage**:
    - Supports secure password storage practices using bcrypt, PBKDF2, and other algorithms.
6. **Integration**:
    - Easily integrates with Spring Boot, making it simpler to configure security in applications.

Example -

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

import
org.springframework.security.config.annotation.authentication.builders.
AuthenticationManagerBuilder;

import
org.springframework.security.config.annotation.web.builders.HttpSecurit
y;

import
org.springframework.security.config.annotation.web.configuration.Enable
WebSecurity;
```

```
import
org.springframework.security.config.annotation.web.configuration.WebSec
urityConfigurerAdapter;


@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {


    @Override

    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {

        auth.inMemoryAuthentication()

            .withUser("user").password("{noop}password").roles("USER")

            .and()

            .withUser("admin").password("{noop}admin").roles("ADMIN");

    }


    @Override

    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()

            .antMatchers("/", "/public/**").permitAll()

            .anyRequest().authenticated()

            .and()

            .formLogin()

            .loginPage("/login")

            .permitAll()

            .and()

            .logout().permitAll();

    }

}
```

In-Memory Authentication: This example uses in-memory authentication with two users: "user" and "admin". Passwords are specified with the `{noop}` prefix to indicate no encoding.

Authorization: It configures access to URLs. The root (/) and /public/** paths are accessible without authentication, while all other requests require authentication.

Form Login: A custom login page is specified, and logout is permitted for all users.

The concepts on (24-10-2024)

Start of Angular -

In an Angular project, we'll see a number of files and folders after running the ng new <project-name> command.

Project Structure:

1. **src/**: This folder contains the source code for your Angular app. Most of your work will happen here.
    - **app/**: This is the core folder where the main components, services, and modules of your app live.
        - **app.module.ts**: The root module that bootstraps your app. You'll declare components and import other modules here.
        - **app.component.ts**: The main component of your app. It contains the logic for the root component.
        - **app.component.html**: The HTML file for the root component.
        - **app.component.css**: The CSS file for styling the root component.
        - **app.component.spec.ts**: The unit test file for the root component.
2. **angular.json**: Configuration file for your Angular project. It defines how the Angular CLI builds and serves your application.
3. **package.json**: This file keeps track of your project's dependencies, such as Angular packages, and includes scripts for starting, building, and testing your app.
4. **node_modules/**: A folder where npm (Node Package Manager) installs the project's dependencies.
5. **tsconfig.json**: Configuration for TypeScript. Angular is written in TypeScript, and this file controls how TypeScript compiles your code.

Key Commands:

- ng serve: Starts your application in development mode and opens it in your browser.
- ng build: Builds your application for production.

Angular Component Introduction and it's structure -

A component in Angular is a building block of the application. It controls a portion of the user interface (UI) and encapsulates its logic.

Anatomy of a Component:

1. Decorator:

Every component is decorated with the @Component decorator, which provides metadata about the component.

Example:

```
@Component({
  selector: 'app-example', // HTML tag to use the component
  templateUrl: './example.component.html', // Path to the HTML template
  styleUrls: ['./example.component.css'] // Path to the component-specific styles
})
```

2. Class:

The class contains the business logic for the component.

It can define properties and methods that manage data and behavior.

Example:

```
export class ExampleComponent {
  title: string = 'Hello, Angular!';

  greet() {
    console.log(this.title);
  }
}
```

3. **Template:**

The template defines the HTML structure that the component renders.

It can include Angular directives and bindings to display dynamic data.

Example:

```
<h1>{{ title }}</h1>
<button (click)="greet()">Greet</button>
```

4. Styles:

Component styles can be defined in a separate CSS file or inline within the component.

This encapsulates the styles, preventing them from affecting other components.

One-Way Data Binding in Angular -

One-way data binding allows data to flow in one direction: from the component to the view or vice versa. In Angular, there are two main types of one-way binding: property binding and event binding.

1. Property Binding

Definition: Property binding allows you to set a property of a DOM element or a component based on a component's property. It is used to pass data from the component class to the view.

Syntax: Use square brackets `[ ]` to bind a property.

Example:

typescript

```
// In the component class
export class ExampleComponent {
  title: string = 'Welcome to Angular!';
}
```

html

```
<!-- In the template -->
<h1 [innerText]="title"></h1>
```

In this example, the title property from the component is bound to the innerText property of the <h1> element. Whenever the title changes in the component, the displayed text updates automatically.

---

2. Event Binding

Definition: Event binding allows you to listen to events (like clicks, key presses, etc.) and execute methods defined in your component class. This is how you can react to user interactions.

Syntax: Use parentheses `( )` to bind to an event.

Example:

typescript

// In the component class

export class ExampleComponent {

  greet() {

    alert('Hello, Angular!');

  }

}

html

<!-- In the template -->

<button (click)="greet()">Click Me</button>

In this example, when the button is clicked, the greet() method is called. This method can perform actions or change data, allowing for interactivity within the application.

NgModel Two way binding -

ngModel is a directive in Angular that creates a two-way data binding between an input field and a property in your component's class. This means that changes in the input field update the component's property, and changes in the property update the input field.

```
// app.component.ts

import { Component } from '@angular/core';


@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  name: string = '';  // Declare a property

}
```

```
<!-- app.component.html -->

<div>

  <h2>Two-Way Data Binding Example</h2>


  <!-- Bind input field to "name" property using ngModel -->

  <input [(ngModel)]="name" placeholder="Enter your name">


  <p>Hello, {{ name }}!</p>

</div>
```

[(ngModel)]="name": This binds the input field to the name property in your component class. The [( )] syntax is called a banana-in-a-box and represents two-way data binding.

- When you type something in the input field, it updates the name property.
- If the name property changes in the component, the input field gets updated automatically.

If you type "John" in the input field, the component's name property becomes "John", and the <p> element dynamically updates to say "Hello, John!".

Angular Structural Directives -

In Angular, structural directives are used to change the DOM layout by adding or removing elements based on certain conditions. The examples provided demonstrate two commonly used structural directives: *ngIf and *ngFor.

1. Using *ngIf

Concept: The *ngIf directive conditionally includes or excludes a block of HTML based on a boolean expression. It's similar to an instruction that says, "If the condition is true, show this part of the UI."

```
<div *ngIf="isHeavy">

  Please get assistance!

</div>
```

Explanation:

- In this case, isHeavy is a boolean property in the component.
- If isHeavy is true, the message "Please get assistance!" will be displayed; if it's false, this div will not be rendered in the DOM at all.
- This is useful for displaying messages or elements based on certain conditions, enhancing user experience by providing relevant information.

2. Using *ngFor

Concept: The *ngFor directive is used to iterate over a collection (like an array) and repeat a block of HTML for each item in that collection. It's akin to an instruction that says, "Repeat this for each item in the list."

```
<div *ngFor="let item of parts">
  Install {{item}}
</div>
```

Explanation:

- Here, parts is an array of items defined in the component.
- For each item in the parts array, the div will be rendered, displaying "Install" followed by the name of the item.
- This is very useful for dynamically generating lists in the UI based on data, making your application flexible and responsive to changes in the underlying data model.

Conditional Styling with ngStyle -

In Angular, you can dynamically change the styles of an element based on component properties using the ngStyle directive. This allows you to apply conditional styling directly in your templates.

```
<div [ngStyle]="{'color': isImportant ? 'red' : 'black'}"> Important Note </div>
```

Explanation:

1. Dynamic Property Binding:
    - The square brackets [] indicate that you are binding to a property. In this case, you're binding to the ngStyle directive.
2. Using an Object Literal:
    - Inside the brackets, you provide an object that defines the styles to be applied.
    - Here, the object specifies that the color property should be either 'red' or 'black'.

The concepts on (25-10-2024)

Understanding Angular Modules -

In Angular, modules are a fundamental building block that help organize an application. Each Angular application has at least one module, the root module, which is typically called AppModule. Modules encapsulate components, services, directives, and pipes, providing a way to group related code together.

1. NgModule Decorator:
    - The NgModule decorator is used to define a module. It takes an object that specifies the components, directives, pipes, and services that belong to the module.
2. Declarations:
    - This array lists all components, directives, and pipes that are part of the module.
3. Imports:
    - This array imports other modules that are needed for the module to function.
4. Exports:
    - This array allows components, directives, and pipes to be used in other modules.
5. Providers:
    - This array is used for services that can be injected into components.
6. Bootstrap:
    - This array specifies the root component that Angular should bootstrap when the application starts.

```
// app.module.ts

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';



@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [
```

```
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent] })
export class AppModule { }
```

Explanation:

1. Imports:
    - NgModule, BrowserModule, and AppComponent are imported.
2. NgModule Definition:
    - declarations: Lists AppComponent as part of this module.
    - imports: Includes BrowserModule, which is necessary for any web application.
    - providers: Currently empty, but you can add services here.
    - bootstrap: Indicates that AppComponent is the starting point of the application.

Components and Templates -

Components are the core building blocks of Angular applications. Each component consists of three main parts: the TypeScript class, the HTML template, and the CSS styles. Components allow you to create reusable UI elements and encapsulate their behavior and presentation.

Key Concepts:

1. **Component Definition**:
    - Each component is defined using the @Component decorator, which specifies its selector, template, styles, and other metadata.
2. **Template**:
    - The HTML view associated with the component, defining how the UI should look.
3. **Styles**:
    - CSS styles specific to the component to ensure that styling is scoped to that component only.
4. **Data Binding**:
    - Angular provides two-way data binding, allowing synchronization between the model and the view.

```
// app.component.ts

import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <h1>{{ title }}</h1>
    <input [(ngModel)]="title" placeholder="Enter title">
  `,
  styles: ['h1 { font-family: Lato; }'] // Styles
})
export class AppComponent {
  title = 'Hello Angular';
}
```

Explanation:

1. Imports:
    - Component is imported from @angular/core.
2. Component Decorator:
    - selector: Defines the HTML tag <app-root> to use this component.
    - template: Contains HTML with data binding. The {{ title }} syntax binds the component's title property to the template. The input uses two-way binding ([(ngModel)]) to allow changes in the input field to update the title.
3. Styles:
    - A simple CSS rule to style the <h1> element.
4. Class Definition:
    - The AppComponent class contains a property title initialized with a string.

Directives -

Directives in Angular are special markers on a DOM element that tell Angular to attach a specific behavior to that element or even transform the DOM structure. They can be categorized into three types: **Components**, **Structural Directives**, and **Attribute Directives**.

Key Concepts:

1. Components:
    - These are directives with a template. They create a new view.
2. Structural Directives:
    - These change the structure of the DOM by adding or removing elements. Common structural directives include:
        - *ngIf: Conditionally includes a template based on a boolean expression.
        - *ngFor: Iterates over a collection and instantiates a template for each item.
        - *ngSwitch: Conditionally displays one of a set of templates based on a switch expression.
3. Attribute Directives:
    - These change the appearance or behavior of an existing element. Examples include:
        - ngClass: Adds or removes CSS classes based on a condition.
        - ngStyle: Allows setting styles dynamically.

```
// app.component.ts
import { Component } from '@angular/core';


@Component({
  selector: 'app-root',
  template: `
    <h1>My Favorite Fruits</h1>
    <ul>
      <li *ngFor="let fruit of fruits">{{ fruit }}</li>
    </ul>
    <div *ngIf="showMessage">
      <p>You have selected some fruits!</p>
    </div>
```

```
    <button (click)="toggleMessage()">Toggle Message</button>
  `
})
export class AppComponent {
  fruits = ['Apple', 'Banana', 'Cherry', 'Date'];
  showMessage = true;

  toggleMessage() {
    this.showMessage = !this.showMessage;
  }
}
```

Explanation:

1. Imports:
    - The Component decorator is imported from @angular/core.
2. Template:
    - The template includes:
        - A heading for the list of fruits.
        - An unordered list where *ngFor iterates over the fruits array and creates a list item for each fruit.
        - A div that uses *ngIf to conditionally display a message if showMessage is true.
        - A button that calls the toggleMessage method to toggle the visibility of the message.
3. Class Definition:
    - The AppComponent class contains an array of fruits and a boolean showMessage.
    - The toggleMessage method changes the value of showMessage.

Services and Dependency Injection -

In Angular, services are classes that provide specific functionality and can be reused throughout the application. They are typically used for tasks such as data retrieval, business logic, and shared state. Dependency Injection (DI) is a design pattern that allows you to inject services into components, promoting modularity and testability.

Key Concepts:

1. Creating a Service:
    - Services are typically created using the Angular CLI and are defined as classes annotated with the @Injectable() decorator.
2. Injecting a Service:
    - Angular's DI system allows you to inject services into components or other services. This is done by declaring the service in the constructor of the class.
3. Providing a Service:
    - Services can be provided at different levels: globally in the root module or locally in specific components or modules.
4. Singleton Services:
    - By default, services provided in the root module are singletons, meaning a single instance is shared across the application.

Example -

```
// fruit.service.ts

import { Injectable } from '@angular/core';



@Injectable({

  providedIn: 'root', // Makes the service available throughout the
application

})

export class FruitService {

  private fruits = ['Apple', 'Banana', 'Cherry'];



  getFruits() {

    return this.fruits; // Method to retrieve fruits

  }

}
```

```
// app.component.ts

import { Component, OnInit } from '@angular/core';

import { FruitService } from './fruit.service';


@Component({

  selector: 'app-root',

  template: `

    <h1>Fruits List</h1>

    <ul>

      <li *ngFor="let fruit of fruits">{{ fruit }}</li>

    </ul>

  `

})

export class AppComponent implements OnInit {

  fruits: string[] = [];


  constructor(private fruitService: FruitService) {} // Injecting the
service


  ngOnInit() {

    this.fruits = this.fruitService.getFruits(); // Using the service
to get fruits

  }

}
```

Explanation:

1. Service Definition:
    - FruitService is defined with the @Injectable() decorator, which makes it eligible for DI.
    - The getFruits method returns an array of fruits.

2. Component Using the Service:
    - In AppComponent, the FruitService is injected via the constructor.
    - The ngOnInit lifecycle hook is used to call getFruits and populate the fruits array.

Routing and Navigation -

Routing in Angular enables you to build single-page applications (SPAs) by navigating between different views or components without reloading the page. The Angular Router helps manage navigation, allowing you to define routes and link them to components.

Key Concepts:

1. **RouterModule**:
    - This module provides the necessary services and directives to enable routing in your application.
2. **Routes**:
    - An array of route objects that define paths and the components to load when those paths are accessed.
3. **RouterOutlet**:
    - A directive that acts as a placeholder for the routed component. It tells Angular where to display the component associated with the current route.
4. **Navigation**:
    - Angular provides methods to programmatically navigate between routes or use routerLink in templates for declarative navigation.
5. **Route Parameters**:
    - You can pass parameters through the route to the components, enabling dynamic views based on user input.

```
// app-routing.module.ts

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';


const routes: Routes = [

  { path: '', component: HomeComponent }, // Default route

  { path: 'about', component: AboutComponent } // About route

];
```

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

```
// home.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  template: '<h2>Welcome to the Home Page</h2>'
})
export class HomeComponent { }

// about.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-about',
  template: '<h2>About Us</h2>'
})
export class AboutComponent { }
```

```
// app.component.ts
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <nav>
      <a routerLink="/">Home</a>
      <a routerLink="/about">About</a>
    </nav>
    <router-outlet></router-outlet> <!-- Placeholder for routed
components -->
  `
})
export class AppComponent { }
```

Explanation:

1. Routing Module:
    - AppRoutingModule defines routes using an array of route objects.
    - Each route specifies a path and the component to render when that path is accessed.
    - The RouterModule.forRoot(routes) method configures the router at the application's root level.
2. Components:
    - HomeComponent and AboutComponent are simple components that display different content.
3. RouterOutlet:
    - In the main AppComponent, router-outlet acts as the placeholder for displaying the routed components.
    - routerLink directives are used to navigate between the Home and About pages without reloading the app.

The concepts on (28-10-2024)

1. Building a small Todo Application using angular

The application allows users to manage a list of tasks, including adding new tasks, marking them as completed, and deleting them.

Imports

```
import { Component } from '@angular/core';

import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
```

- Component: Decorator to define an Angular component.
- CommonModule: Provides common directives such as ngFor and ngIf.
- FormsModule: Enables two-way data binding for form elements.

Interface Definition

```
interface Todo{

  id:number;

  text:string;

  completed:boolean;

}
```

Defines a Todo interface to structure the data for each task, ensuring type safety.

Component Metadata

@Component({

 selector: 'app-root',

 standalone: true,

 imports: [CommonModule, FormsModule],

```
  template: `...`,
  styles: `...`
})
```

- **selector**: The component's HTML tag.
- **standalone**: Indicates that the component can operate independently.
- **imports**: Modules used by this component.
- **template**: Inline HTML for the component's view.
- **styles**: Inline CSS for styling the component.

The component's template consists of several sections:

1. **Add Todo Section**:
    - An input field for entering a new TODO task.
    - A button to add the task.
2. **Statistics Section**:
    - Displays the total number of TODOs and the count of completed TODOs.
3. **Todo List Section**:
    - Renders a list of TODO items using ngFor.
    - Each item has buttons to mark it as complete or delete it.

Properties

```
todos: Todo[] = [...];  // Array of TODO items
newTodoText: string = '';  // Input text for new TODO
nextId: number = 4;  // ID for the next TODO item
```

- todos: An array holding the list of TODO items.
- newTodoText: Tracks the input text for the new TODO.
- nextId: Automatically increments to ensure unique IDs.

Methods

- updateNewTodo(event: Event): Updates newTodoText with the input field's value.
- addTodo():
    - Checks if the input is not empty.
    - Adds a new TODO item to the todos array with a unique ID.
    - Resets the input field.

- getCompletedTodosCount():
    - Returns the number of completed TODOs by filtering the todos array.
- completeTodo(id: number):
    - Finds a TODO by its ID and marks it as completed.
- deleteTodo(id: number):
    - Filters out the TODO with the specified ID, effectively deleting it from the list.

User Interaction Flow

1. **Adding a TODO**:
    - The user types in the input field and clicks the "Add Todo" button. If the input is valid, a new TODO is created and added to the list.
2. **Completing a TODO**:
    - The user clicks the "Complete" button next to a TODO item, which updates its status to completed.
3. **Deleting a TODO**:
    - The user clicks the "Delete" button next to a TODO item, removing it from the list.

```
updateNewTodo(event:Event):void{
    const input =event.target as HTMLInputElement;
    this.newTodoText = input.value;
  }

  addTodo():void{
    if(this.newTodoText.trim() === ''){
      return;
    }

this.todos.push({id:this.nextId++,text:this.newTodoText,completed:false
});
    this.newTodoText = '';
  }
```

```
getCompletedTodosCount():number{
  return this.todos.filter(todo => todo.completed).length;
}

completeTodo(id: number): void {
  const todo = this.todos.find(todo => todo.id === id);
  if (todo) {
    todo.completed = true;
  }
}

deleteTodo(id:number):void{
  this.todos=this.todos.filter(todo =>  todo.id !== id);
}
```

In JavaScript, the filter method creates a new array containing elements that pass a specific condition defined by a callback function. It does not modify the original array.

1. Counting Completed TODOs

In your getCompletedTodosCount method, you use filter to count the completed tasks:

- this.todos.filter(todo => todo.completed): This line filters the todos array to include only those TODO items where the completed property is true.
    - todo => todo.completed: This is an arrow function that acts as the condition. For each todo in the array, it checks if todo.completed is true.
- .length: Finally, it returns the number of items in the filtered array, giving the total count of completed TODOs.

2. Deleting a TODO

In the deleteTodo method, you use filter to remove a TODO by its ID:

- this.todos.filter(todo => todo.id !== id): This line creates a new array containing all TODOs except the one with the specified ID.
    - todo => todo.id !== id: This arrow function checks each todo. If the todo.id is not equal to the id passed to the deleteTodo method, it includes that todo in the new array.

2. Building a simple shopping application-

The main functionality is encapsulated in the AppComponent. This component manages the shopping list's state and user interactions.

Imports

```
import { Component } from '@angular/core';

import { CommonModule } from '@angular/common';

import { FormsModule } from '@angular/forms';
```

Component: Used to define an Angular component.

CommonModule: Provides common directives like ngFor and ngIf.

FormsModule: Facilitates two-way data binding with forms.

Interface Definition

```
interface ShoppingItem {

    id: number;

    name: string;

    quantity: number;

    category: string;

    purchased: boolean;

  }
```

This interface defines the structure of each shopping item, ensuring type consistency throughout the application.

Component Metadata

```
@Component({

    selector: 'app-root',

    standalone: true,
```

```
    imports: [CommonModule, FormsModule],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
  })
```

- **selector**: The custom HTML tag for the component.
- **standalone**: Indicates that this component operates independently.
- **imports**: Specifies the modules that this component will use.
- **templateUrl** and **styleUrls**: Reference external HTML and CSS files for the component's layout and styling.

Template Structure

The template includes several sections:

1. **Add Item Section**:
    - Input fields for item name, quantity, and category.
    - A button to add the item.
2. **Filter Section**:
    - A dropdown to filter items by category.
3. **Statistics Section**:
    - Displays total, purchased, and pending items.
4. **Item List Section**:
    - Renders a list of shopping items with options to mark them as purchased or delete them.

Component Logic

Properties

```
items: ShoppingItem[] = [];  // Array of shopping items
newItemName: string = '';  // Input for item name
newItemQuantity: number = 1;  // Input for item quantity
newItemCategory: string = '';  // Input for item category
nextId: number = 1;  // ID for the next item
filterCategory: string = 'All';  // Current filter category
```

- items: Holds the list of shopping items.

- newItemName, newItemQuantity, newItemCategory: Track user input for adding new items.
- nextId: Automatically increments for unique item IDs.
- filterCategory: Manages the selected category filter.

Methods

- addItem():
    - Validates input and adds a new shopping item to the items array, resetting the input fields.
- markAsPurchased(id: number):
    - Marks an item as purchased by finding it by ID and updating its purchased property.
- deleteItem(id: number):
    - Uses filter to remove the item with the specified ID from the items array.
- getStatistics():
    - Calculates and returns total items, purchased items, and pending items.
- getUniqueCategories():
    - Extracts unique categories from the items array using a Set to remove duplicates.
- filterItems():
    - Filters the list of items based on the selected category. If "All" is selected, it returns all items; otherwise, it returns only items that match the selected category.

Filtering Items by Category

In the filterItems method, you use the filter method to display items based on the selected category:

```
filterItems() {
  if (this.filterCategory === 'All') {
    return this.items;
  }
  return this.items.filter(item => item.category === this.filterCategory);
}
```

- Condition Check: If filterCategory is set to "All," the method returns the complete list of items.

- Filtering Logic: If a specific category is selected, filter is applied:
    - this.items.filter(item => item.category === this.filterCategory): This creates a new array that includes only those items whose category matches the filterCategory. The condition checks each item in items.

Code -

```
<div class="shopping-list-container">
  <h1>Shopping List</h1>

  <div class="add-item">
    <input type="text" [(ngModel)]="newItemName" placeholder="Item Name" />
    <input type="number" [(ngModel)]="newItemQuantity" placeholder="Quantity" min="1" />
    <input type="text" [(ngModel)]="newItemCategory" placeholder="Category" />
    <button (click)="addItem()">Add Item</button>
  </div>

  <div class="filter">
    <label for="filter">Filter by Category: </label>
    <select [(ngModel)]="filterCategory" id="filter">
      <option value="All">All</option>
      <option *ngFor="let category of getUniqueCategories()" [value]="category">{{ category }}</option>
    </select>
  </div>

  <div class="statistics">
    <p>Total Items: {{ getStatistics().total }}</p>
    <p>Purchased Items: {{ getStatistics().purchased }}</p>
    <p>Pending Items: {{ getStatistics().pending }}</p>
```

```
  </div>


  <ul>
    <li *ngFor="let item of filterItems()">
      {{ item.name }} ({{ item.quantity }}) - {{ item.category }}
      <button (click)="markAsPurchased(item.id)">Purchased</button>
      <button (click)="deleteItem(item.id)">Delete</button>
    </li>
  </ul>
</div>
```

```
import { RouterOutlet } from '@angular/router';
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';


interface ShoppingItem {
  id: number;
  name: string;
  quantity: number;
  category: string;
  purchased: boolean;
}


@Component({
  selector: 'app-root',
  standalone:true,
```

```
  imports:[CommonModule,FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  items: ShoppingItem[] = [];
  newItemName: string = '';
  newItemQuantity: number = 1;
  newItemCategory: string = '';
  nextId: number = 1;
  filterCategory: string = 'All';

  addItem(): void {
    if (this.newItemName.trim() === '' || this.newItemCategory.trim()
=== '') {
      return;
    }
    this.items.push({
      id: this.nextId++,
      name: this.newItemName,
      quantity: this.newItemQuantity,
      category: this.newItemCategory,
      purchased: false
    });
    this.newItemName = '';
    this.newItemQuantity = 1;
    this.newItemCategory = '';
  }
```

```
  markAsPurchased(id: number): void {
    const item = this.items.find(i => i.id === id);
    if (item) {
      item.purchased = true;
    }
  }

  deleteItem(id: number): void {
    this.items = this.items.filter(item => item.id !== id);
  }

  getStatistics() {
    const total = this.items.length;
    const purchased = this.items.filter(item => item.purchased).length;
    const pending = total - purchased;
    return { total, purchased, pending };
  }

  getUniqueCategories(): string[] {
    const categories = this.items.map(item => item.category);
    return [...new Set(categories)];// using set to remove duplicates
  }

  filterItems() {
    if (this.filterCategory === 'All') {
      return this.items;
    }
    return this.items.filter(item => item.category ===
this.filterCategory);
```

```
  }

}
```

Flow when we add an item -

1. User Input:
    - The user fills in the input fields: item name, quantity, and category.
    - This is done through the HTML template using two-way data binding with [(ngModel)]:
2.
    - As the user types in these fields, the values of newItemName, newItemQuantity, and newItemCategory in the component are automatically updated.
3. Clicking the "Add Item" Button:
    - When the user clicks the "Add Item" button, the addItem() method is triggered:
4. Executing the addItem() Method:
    - The addItem() method is defined in the AppComponent class:
5.
    - **Input Validation**:
        - The method first checks if newItemName or newItemCategory are empty (after trimming any whitespace). If either is empty, the method exits early, and no item is added.
    - **Creating a New Item**:
        - If validation passes, a new shopping item object is created:
            - The id is assigned from this.nextId, which is then incremented for the next item.
            - The name, quantity, and category are taken from the user input.
            - The purchased property is set to false by default.
    - **Updating the Item List**:

The new item object is pushed into the items array:

this.items.push({...});

6. **Updating the User Interface**:
    - Because of Angular's change detection, when the items array is updated, the UI automatically reflects this change.
    - The shopping items are displayed in the list through *ngFor in the template:

The concepts on (30-10-2024)

Validation in spring where you define the values and messages in properties file-

```
@RestController

@RequestMapping("/api/users")

public class UserController {



    @PostMapping

    public ResponseEntity<String> createUser(@Valid @RequestBody
UserDTO userDTO) {

        // Process valid user

        return ResponseEntity.ok("User created successfully");

    }

}
```

UserController:

- Package: com.example.validation.controller
- Annotations:
    - @RestController: Marks this class as a controller where every method returns a domain object instead of a view.
    - @RequestMapping("/api/users"): Maps HTTP requests to /api/users to this controller.
- Method:
    - createUser: This method handles POST requests. It takes a UserDTO object as input, validated by the @Valid annotation. If the input is valid, it returns a success response

```
@Data

public class UserDTO {



    @NotNull(message = "{user.name.required}")
```

```
    @Size(min = 2, max = 10, message = "{user.name.length}")
    private String name;

    @NotNull(message = "{user.email.required}")
    private String email;

    @Size(min = 5, message = "{user.password.length}")
    @NotNull(message = "{user.password.required}")
    private String password;
}
```

UserDTO:

- Package: com.example.validation.dto
- Annotations:
    - @Data: A Lombok annotation that generates getters, setters, and other utility methods.
    - @NotNull, @Size: Validation annotations that ensure the fields are not null and that they meet size requirements.
- Fields:
    - name: Must be between 2 and 10 characters.
    - email: Must not be null.
    - password: Must not be null and must be at least 5 characters long.

```
@Data
@Configuration
@ConfigurationProperties(prefix = "validation.user")
public class ValidationProperties {
    private NameValidation name = new NameValidation();
    private EmailValidation email = new EmailValidation();
    private PasswordValidation password = new PasswordValidation();
```

```
    @Data
    public static class NameValidation {
        private int min;
        private int max;
    }

    @Data
    public static class EmailValidation {
        private String pattern;
    }

    @Data
    public static class PasswordValidation {
        private int min;
        private String pattern;
    }
}
```

ValidationProperties:

- Package: com.example.validation.config
- Purpose: This class is used to configure validation rules through external properties.
- Annotations:
    - @Configuration: Indicates that this class contains configuration methods.
    - @ConfigurationProperties(prefix = "validation.user"): Binds the properties prefixed with validation.user in application configuration files to this class.
- Inner Classes:
    - NameValidation, EmailValidation, PasswordValidation: Nested classes that hold validation properties for names, emails, and passwords, respectively.

```java
@Configuration
public class ValidationConfig {


    @Bean
    public MessageSource messageSource() {
        ReloadableResourceBundleMessageSource messageSource = new
ReloadableResourceBundleMessageSource();
        messageSource.setBasename("classpath:messages");
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
    }


    @Bean
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean bean = new
LocalValidatorFactoryBean();
        bean.setValidationMessageSource(messageSource());
        return bean;
    }
}
```

ValidationConfig:

- Package: com.example.validation.config
- Purpose: Configures message sources for validation messages and sets up the validator.
- Methods:
    - messageSource(): Configures a MessageSource to load validation messages from a properties file (messages).
    - validator(): Configures the validator to use the custom message source.

In Spring, the conversion of string values defined in properties files into corresponding Java types can be challenging, particularly when it comes to custom validations or complex data types. However, there are ways to achieve this through configuration.

Project

Project management - Service Center Web Application Project | Trello

Github Repo - 1rithwik/WebApplication_Project at rithwik

Project doc - MthreeProject document - Google Docs