Git

Source Control Management (SCM)

Definition:
Source Control Management (SCM) is the practice of **tracking and managing changes** to code, scripts, and configuration files in software development.

Why SCM Is Needed (Purpose)

- To **maintain version history** of code
- To **collaborate with multiple developers** efficiently
- To **track changes, revert mistakes, and manage releases**
- Essential for **DevOps and professional development workflows**

Key Concepts

Concept	Explanation
Repository	Storage for code (local or remote)
Commit	Save changes to repository
Branch	Independent line of development
Merge	Combine changes from different branches
Clone	Copy a repository locally
Pull	Fetch and integrate changes from remote
Push	Send local changes to remote
Conflict	When changes in branches clash

Popular SCM Tools

- **Git** – Most widely used distributed version control
- **SVN (Subversion)** – Centralized version control
- **Mercurial** – Another distributed version control

Basic Git Workflow

```
# Clone a repository
git clone <repo-url>

# Check status
git status

# Stage changes
git add file.txt

# Commit changes
git commit -m "Added new feature"

# Push to remote
git push origin main

# Pull updates
git pull origin main
```

Real-Life Example

- Team working on a project:

- Developer A adds a feature on branch feature-login
- Developer B fixes a bug on branch bugfix-auth
- Merge branches into main after testing

Technical Example (Production Support)

- Rollback a script to previous working version:

```
git log            # View commit history
git checkout <commit-id> script.sh
```

Interview Explanation (How to Say It)

"SCM tools like Git help track changes, collaborate, and manage versions of code, enabling teams to work efficiently and maintain code integrity."

Common Interview Questions

- Difference between Git and SVN?
- How do you resolve merge conflicts?
- What is a branch and why is it used?
- How to revert a commit?

Common Mistakes Freshers Make

- Not committing frequently
- Ignoring branching best practices
- Not resolving conflicts properly
- Forgetting to pull before pushing

One-Line Summary

SCM tracks, manages, and collaborates on code changes to maintain version control and project integrity.

WORKING LOCALLY WITH GIT

Definition:

Working locally with Git involves **creating, managing, and tracking code changes in your local repository** before sharing them with a remote repository.

Why Working Locally Is Needed (Purpose)

- To **develop and test code independently**
- To **maintain version history before pushing**
- To **work offline and commit changes safely**
- Essential for **collaboration and professional workflows**

Key Concepts

Concept	Explanation
Repository (repo)	Local storage of your project tracked by Git
Commit	Save changes in local repository with a message
Staging Area (Index)	Temporary area to prepare changes before commit
Branch	Independent line of development
HEAD	Pointer to the current commit

Basic Local Git Workflow

```
# Initialize a new local repository
git init

# Check status of files
git status

# Stage files for commit
git add file.txt
git add .       # Stage all changes

# Commit staged changes
git commit -m "Initial commit"

# View commit history
git log

# Create a new branch
git branch feature-login

# Switch to branch
git checkout feature-login
```

Real-Life Example

- You are developing a login module locally:
  - Create a branch feature-login
  - Make changes and commit them locally
  - Test everything before pushing to remote

Technical Example (Production Support)

- Edit a script and maintain history:

```
git status
git add update_script.sh
git commit -m "Fixed issue in backup script"
git log --oneline
```

Useful Local Git Commands

Command	Purpose
git diff	View changes not staged
git diff --staged	View staged changes
git reset HEAD file.txt	Unstage a file
git checkout -- file.txt	Revert local changes
git stash	Temporarily save changes
git stash pop	Apply stashed changes

Interview Explanation (How to Say It)

"Working locally with Git means managing commits, branches, and changes in a local repository before collaborating with a remote repository."

Common Interview Questions

- Difference between staging area and commit?
- How to revert changes locally?
- What is git stash used for?
- How to create and switch branches?

Common Mistakes Freshers Make

- Committing directly without staging
- Not writing meaningful commit messages
- Overwriting local changes without backup
- Ignoring branch management best practices

One-Line Summary

Local Git workflow allows developers to track, test, and commit code safely before sharing with remote repositories.

GIT FUNDAMENTALS

Definition:

Git is a **distributed version control system** used to track changes in code, enabling collaboration among developers and maintaining history of a project.

Why Git Is Needed (Purpose)

- Tracks **every change** in code
- Allows **collaboration across multiple developers**
- Enables **branching, merging, and rollback**
- Essential for **DevOps, software development, and production support**

Key Concepts

Concept	Explanation
Repository (Repo)	Storage for your project (local or remote)
Commit	Snapshot of changes in the repo
Branch	Independent line of development
Merge	Combine changes from different branches
Staging Area (Index)	Area where changes are prepared before committing
HEAD	Pointer to the current commit
Clone	Copy remote repo locally
Pull	Fetch and merge changes from remote
Push	Send local commits to remote

Git Architecture

1. **Working Directory** – Files you are editing
2. **Staging Area** – Files staged for commit
3. **Local Repository** – Committed changes stored locally
4. **Remote Repository** – Shared repository for collaboration

---

Basic Git Workflow

```
# Initialize a repository
git init

# Check status
git status

# Stage files
git add file.txt
```

```
# Commit staged files
git commit -m "Initial commit"

# View commit history
git log

# Create a new branch
git branch feature-branch

# Switch to branch
git checkout feature-branch

# Merge branch into main
git checkout main
git merge feature-branch
```

Real-Life Example

- Developer A creates a feature branch
- Commits changes locally
- Merges into main after testing
- Pushes to remote for team collaboration

Technical Example (Production Support)

- Rollback to previous commit:

```
git log --oneline
git checkout <commit-id> script.sh
```

Interview Explanation (How to Say It)

"Git is a distributed version control system that tracks changes, enables branching and merging, and facilitates collaboration among developers."

Common Interview Questions

- Difference between Git and SVN?
- What is the purpose of staging area?
- Difference between git pull and git fetch?
- How to revert a commit?

Common Mistakes Freshers Make

- Committing without meaningful messages
- Working directly on main branch
- Ignoring branch merges and conflicts
- Not syncing local and remote repos frequently

One-Line Summary

Git fundamentals include tracking code changes, branching, merging, and collaborating using local and remote repositories.

GIT FUNDAMENTALS

Definition:

Git is a **distributed version control system** used to track changes in code, enabling collaboration among developers and maintaining history of a project.

Why Git Is Needed (Purpose)

- Tracks **every change** in code
- Allows **collaboration across multiple developers**
- Enables **branching, merging, and rollback**
- Essential for **DevOps, software development, and production support**

Key Concepts

Concept	Explanation
Repository (Repo)	Storage for your project (local or remote)
Commit	Snapshot of changes in the repo
Branch	Independent line of development
Merge	Combine changes from different branches
Staging Area (Index)	Area where changes are prepared before committing
HEAD	Pointer to the current commit
Clone	Copy remote repo locally
Pull	Fetch and merge changes from remote
Push	Send local commits to remote

Git Architecture

1. **Working Directory** – Files you are editing
2. **Staging Area** – Files staged for commit
3. **Local Repository** – Committed changes stored locally
4. **Remote Repository** – Shared repository for collaboration

---

Basic Git Workflow

```
# Initialize a repository
git init

# Check status
git status

# Stage files
git add file.txt

# Commit staged files
git commit -m "Initial commit"

# View commit history
git log

# Create a new branch
git branch feature-branch

# Switch to branch
git checkout feature-branch

# Merge branch into main
git checkout main
git merge feature-branch
```

Real-Life Example

- Developer A creates a feature branch

- Commits changes locally
- Merges into main after testing
- Pushes to remote for team collaboration

Technical Example (Production Support)

- Rollback to previous commit:

```
git log --oneline
git checkout <commit-id> script.sh
```

Interview Explanation (How to Say It)

"Git is a distributed version control system that tracks changes, enables branching and merging, and facilitates collaboration among developers."

Common Interview Questions

- Difference between Git and SVN?
- What is the purpose of staging area?
- Difference between git pull and git fetch?
- How to revert a commit?

Common Mistakes Freshers Make

- Committing without meaningful messages
- Working directly on main branch
- Ignoring branch merges and conflicts
- Not syncing local and remote repos frequently

One-Line Summary

Git fundamentals include tracking code changes, branching, merging, and collaborating using local and remote repositories.

Git Commit, Branch & Merge – Fresher & Interview Friendly

Git Commit

Definition:

A **commit** is a **snapshot of your changes** in the local repository with a descriptive message.

Purpose:

- Tracks changes in project files
- Enables version history
- Helps revert to previous state if needed

Basic Commands:

```
git add file.txt       # Stage file for commit
git commit -m "Added login feature"  # Commit staged files with message
git log                # View commit history
```

Real-Life Example:

- Saving changes in a script before testing:

```
git add backup.sh
git commit -m "Fixed backup script logic"
```

Git Branch

Definition:

A **branch** is an **independent line of development** in a repository.

Purpose:

- Enables **parallel development**
- Isolates **features, bug fixes, experiments**
- Helps avoid breaking the main code

Basic Commands:

```
git branch               # List branches
git branch feature-login # Create new branch
git checkout feature-login # Switch to branch
git checkout main        # Switch back to main
```

Real-Life Example:

- Working on a new feature without affecting the main code:

```
git branch feature-ui
git checkout feature-ui
```

Git Merge

Definition:

Merging combines changes from one branch into another.

Purpose:

- Integrates completed features into main branch
- Maintains code consistency across team

Basic Commands:

```
git checkout main           # Switch to main branch
git merge feature-login     # Merge feature-login into main
```

Handling Merge Conflicts:

- Git highlights conflicts in files
- Edit files to resolve conflicts
- Mark resolved and commit:

```
git add resolved_file.txt
git commit -m "Resolved merge conflict"
```

Real-Life Example:

- Feature branch development is complete, merge into main for release:

```
git checkout main
git merge feature-ui
```

Interview Explanation (How to Say It)

“Git commits save snapshots of code, branches allow parallel development, and merging integrates changes from one branch to another.”

Common Interview Questions

- Difference between git merge and git rebase?
- How to resolve merge conflicts?
- Difference between branch and checkout?
- Can you merge without committing?

Common Mistakes Freshers Make

- Working directly on main branch
- Forgetting to pull before merging
- Not handling merge conflicts properly
- Using unclear commit messages

One-Line Summary

Commits track changes, branches isolate work, and merges combine code to maintain a clean version history.

Working Remotely with Git – Fresher & Interview Friendly

Definition (Simple)
Working remotely with Git involves **interacting with a repository hosted on a remote server** (like GitHub, GitLab, or Bitbucket) to **push, pull, and collaborate** on code.

Why Remote Git Is Needed (Purpose)

- Enables **team collaboration** on the same project
- Keeps a **centralized backup** of code
- Facilitates **continuous integration and deployment**
- Essential for **professional development workflows**

Key Concepts

Concept	Explanation
Remote Repository	Repository hosted on a server like GitHub
Push	Send local commits to remote repository
Pull	Fetch and merge remote changes into local
Fetch	Download remote changes without merging
Clone	Copy remote repository to local machine
Origin	Default name of the remote repository
Tracking Branch	Local branch linked to remote branch

Basic Commands

4.1 Cloning a Remote Repository

```
git clone <repo-url>
```

- Copies entire repository locally
- Sets origin as default remote

4.2 Viewing Remote Repositories

```
git remote -v
```

4.3 Pushing Local Changes

```
git add file.txt
git commit -m "Added new feature"
git push origin main
```

4.4 Pulling Remote Changes

```
git pull origin main
```

4.5 Fetching Remote Changes Without Merge

```
git fetch origin
```

Real-Life Example

- Developer clones a project from GitHub:

```
git clone https://github.com/username/project.git
cd project
git checkout -b feature-login
# make changes
git add .
git commit -m "Implemented login"
git push origin feature-login
```

- Team members pull changes to stay updated

Technical Example (Production Support)

- Sync local branch with remote main:

```
git checkout main
git pull origin main
```

- Push bug fix to remote:

```
git add fix.sh
git commit -m "Bug fix in backup script"
git push origin main
```

Interview Explanation (How to Say It)
"Working remotely with Git allows developers to collaborate by pushing, pulling, and syncing code with a centralized repository on platforms like GitHub."

Common Interview Questions

- Difference between git pull and git fetch?
- What is the default remote name?
- How to resolve conflicts after git pull?
- How to push a new branch to remote?

Common Mistakes Freshers Make

- Forgetting to pull before pushing
- Conflicts due to multiple commits on the same branch
- Pushing directly to main without review
- Misunderstanding tracking branches

One-Line Summary
Remote Git allows collaboration and version control by syncing local changes with a centralized repository.

Git Branching, Merging & Rebasing – Fresher & Interview Friendly

Git Branching
Definition:
A **branch** is an **independent line of development** in a Git repository.
Purpose:

- Develop features or fixes **isolated from main code**
- Enables **parallel development** without conflicts
- Supports **experimentation** safely

Commands:

```
git branch               # List branches
git branch feature-login # Create new branch
git checkout feature-login # Switch to branch
git checkout main        # Switch back to main
git branch -d feature-login # Delete branch after merge
```

Real-Life Example:

- Create a branch for login feature development:

```
git checkout -b feature-login
```

Git Merging
Definition:
Merging integrates changes from one branch into another.
Purpose:

- Combine completed features into main branch
- Maintain code consistency
- Track integrated changes in version history

Commands:

```
git checkout main
git merge feature-login
```

Handling Conflicts:

- Git marks conflicting sections in files
- Edit manually to resolve conflicts
- Stage and commit after resolution:

```
git add resolved_file.txt
git commit -m "Resolved merge conflict"
```

Real-Life Example:

- Merge a tested feature branch into main for release:

```
git checkout main
git merge feature-ui
```

Git Rebasing

Definition:

Rebasing moves or reapplies commits from one branch onto another **base commit**, creating a **linear history**.

Purpose:

- Maintain **clean, readable history**
- Avoid unnecessary merge commits
- Apply latest main changes to feature branch

Commands:

```
git checkout feature-login
git rebase main   # Apply feature-login changes on top of main
```

Handling Conflicts During Rebase:

- Resolve conflicts as prompted
- Continue rebase:

```
git add resolved_file.txt
git rebase --continue
```

Real-Life Example:

- Keep feature branch updated with main before final merge:

```
git checkout feature-login
git fetch origin
git rebase origin/main
```

Interview Explanation (How to Say It)

“Branching allows parallel development, merging integrates changes, and rebasing reapplies commits on top of another branch to maintain a clean history.”

Common Interview Questions

- Difference between merge and rebase?
- When should you use rebase?
- How to resolve conflicts during merge or rebase?
- Difference between fast-forward merge and regular merge?

Common Mistakes Freshers Make

- Rebasing shared branches (can rewrite history)
- Merging without pulling latest changes
- Ignoring conflict resolution
- Deleting branches before merging

One-Line Summary

Branching isolates work, merging integrates it, and rebasing reapplies commits to maintain a clean Git history.