EXERCISE BOOK FOR GIT

Exercise 1: Basic Git Commands
Scenario: Imagine you are working on a software project with a team. Your task is to set up a Git repository and perform some basic Git commands.

1. Create a new directory called "MyProject" on your computer.
2. Initialize a Git repository in the "MyProject" directory.
3. Create a file named "index.html" inside the "MyProject" directory.
4. Add some content to "index.html."
5. Use the appropriate Git command to stage "index.html."
6. Commit your changes with a meaningful commit message.
7. Check the status of your Git repository.
8. Create a new branch named "feature-branch."
9. Switch to the "feature-branch."
10. Create a new file named "style.css."
11. Add and commit "style.css" to the "feature-branch."
12. Switch back to the main branch (usually "master" or "main").
13. Merge the "feature-branch" into the main branch.

Exercise 2: Collaborative Git Workflow
Scenario: You and a colleague are collaborating on a project using Git. Practice a collaborative workflow with branches and remote repositories.

1. Create a new Git repository on a platform like GitHub or GitLab.
2. Clone the repository to your local machine.
3. Create a branch named "collaborative-feature" and switch to it.
4. Add a new file named "feature.js" with some code.
5. Commit your changes and push the branch to the remote repository.
6. Ask your colleague to clone the repository to their machine.
7. Your colleague should create their own branch named "collaborative-fix" and make some changes to "feature.js."
8. They should commit and push their changes to the remote repository.
9. You should fetch their changes from the remote repository.
10. Merge your colleague's changes into your "collaborative-feature" branch.
11. Resolve any merge conflicts if they occur.
12. Push the updated "collaborative-feature" branch to the remote repository.
13. Create a pull request (PR) on the remote platform to merge "collaborative-feature" into the main branch.
14. Review and merge the PR.

Exercise 3: Git Branching Strategies

Scenario: You are part of a larger development team, and you need to implement a branching strategy to manage releases and features.

1. Review the existing project and identify the main development branch (e.g., "main" or "develop").
2. Create a new branch named "release-1.0" for an upcoming release.
3. Add some new features or changes to the "release-1.0" branch.
4. Create a new branch for a hotfix named "hotfix-1.0.1."
5. Make a critical bug fix in the "hotfix-1.0.1" branch.
6. Merge the hotfix into both "main" and "release-1.0" branches.
7. Tag the "main" branch with version "1.0."
8. Create a new branch for the next release (e.g., "release-2.0").
9. Repeat the process for the next release, including feature development and hotfixes.
10. Discuss the advantages and disadvantages of this branching strategy.

Exercise 4: Forking a Repository and Making Contributions
Scenario: You want to contribute to an open-source project hosted on GitHub by forking the repository and making changes.

1. Go to the GitHub repository of an open-source project you're interested in.
2. Fork the repository to your own GitHub account.
3. Clone your forked repository to your local machine.
4. Create a new branch called "feature-contribution."
5. Make some changes to a file in your branch.
6. Commit the changes and push them to your forked repository.
7. Create a pull request (PR) from your branch to the original repository.
8. Discuss the PR process and any feedback received from maintainers.

Exercise 5: Merging Conflict Resolution
Scenario: You and a colleague are working on the same project and encounter a merge conflict. Practice resolving the conflict.

1. Clone a shared Git repository to your local machine.
2. Create a new branch called "merge-conflict-demo."
3. Make changes to a file that your colleague also modified on their branch.
4. Commit your changes and attempt to merge your branch into the main branch.
5. Encounter and simulate a merge conflict.
6. Use Git tools to resolve the conflict manually.
7. Commit the resolved changes and complete the merge.
8. Push the updated main branch.

Exercise 6: Rebasing
Scenario: You have a feature branch with multiple commits, and you want to rebase it onto the latest changes from the main branch.

1. Clone a Git repository and create a new branch called "feature-branch."
2. Make several commits to the feature branch.
3. Meanwhile, changes have been made to the main branch.
4. Use the rebase command to rebase your "feature-branch" onto the main branch.
5. Resolve any conflicts that arise during the rebase.
6. Push the rebased "feature-branch" to the remote repository.
7. Discuss when and why you might use rebasing in a real project.

Exercise 7: Stash and Pop
Scenario: You're working on a feature, but you need to switch to another task temporarily. Use the stash to save your changes and continue later.

1. Create a new Git repository and initialize it.
2. Create a new file called "task1.txt" and make some changes.
3. Use the stash command to save your changes without committing them.
4. Create another file called "task2.txt" and make changes to it.
5. Complete task2 and commit the changes.
6. Use the stash pop command to apply the changes from "task1.txt" back to your working directory.
7. Commit the changes from "task1.txt."

Exercise 8: Git Time Machine
Scenario: Imagine you have the power to time travel through your Git commit history. Can you use Git commands to achieve this?

1. Clone a Git repository with a rich commit history.
2. Use the Git log command to view the commit history.
3. Challenge: Can you find a commit from three months ago?
4. Use Git commands (like checkout or reset) to travel back in time to that commit.
5. Make a change or add a file in this "past" state.
6. Return to the present by navigating to the latest commit.

Exercise 9: The Mysterious Branch
Scenario: You come across a Git repository with a mysterious branch that seems to contain hidden secrets. Can you uncover them?

1. Clone a Git repository with multiple branches.
2. Check out a branch named "mystery-branch."
3. Explore the branch's contents and try to find hidden files or messages.
4. Challenge: Decrypt any encoded messages or solve puzzles hidden in the branch.
5. Share your findings with others in the workshop.

Exercise 10: The Git Escape Room
Scenario: You're locked in a virtual "Git Escape Room" and must solve Git-related puzzles to escape.

1. Create a series of Git-related puzzles, each involving a specific Git command or concept.
2. Participants start in a "locked" state (e.g., a branch with no progress).
3. To unlock and advance, they must solve each puzzle correctly.
4. Example puzzles: "To move to the next room, stash your changes and pop them later," or "Merge the 'knowledge' branch to gain a hint."
5. Provide clues or hints for participants who get stuck.
6. The goal is to reach the "exit" (e.g., the main branch) and complete all challenges.

Exercise 11: Git Aliases and Emoji Commits
Scenario: Make Git more fun and expressive by using Git aliases and emoji commits.

1. Introduce participants to Git aliases, which allow custom shorthand for Git commands.
2. Help them set up aliases for common Git commands like commit, status, and log.
3. Encourage the use of emoji in commit messages to express emotions or context.
4. Challenge: Ask participants to create a Git commit using emoji-only messages that convey specific actions (e.g., 🚀 for a feature launch).
5. Discuss the benefits of expressive commit messages in team collaboration.

Exercise 12: Git Murder Mystery
Scenario: Create a Git-themed murder mystery where participants use their Git skills to unravel a fictional crime.

1. Develop a story with characters, motives, and clues related to Git repositories.
2. Provide participants with a Git repository that contains staged clues in the form of branches, commits, and files.
3. Participants must use Git commands to investigate the crime, find evidence, and solve the mystery.
4. Encourage collaboration among participants, as they may need to work together to piece together the story.
5. The first team or individual to solve the mystery wins.

Exercise 13: Git Disaster Recovery
Scenario: Simulate a Git disaster scenario where participants must recover a corrupted Git repository.

1. Provide participants with a Git repository that has been intentionally corrupted (e.g., files deleted, commits altered).
2. Challenge them to use Git's recovery and repair tools to restore the repository to a functional state.
3. Encourage participants to document the steps they take to recover the repository.

4. Discuss best practices for disaster recovery in real-world Git projects.

Exercise 14: Git Security Breach
Scenario: Participants must identify and remediate a security breach in a Git repository.

1. Create a Git repository with vulnerabilities, such as exposed credentials or sensitive data.
2. Challenge participants to perform a security audit on the repository to identify the vulnerabilities.
3. Instruct them to remediate the issues by implementing security best practices (e.g., removing sensitive data, rotating credentials).
4. Discuss the importance of security in Git repositories and ways to prevent breaches.

Exercise 15: The Git Olympics
Scenario: Host a Git Olympics competition with various Git challenges and obstacles.

1. Design a series of Git-related challenges, such as resolving complex merge conflicts, rebasing under time pressure, or creating intricate branching strategies.
2. Participants compete individually or in teams to complete each challenge within a set time limit.
3. Assign points for successful completion of challenges and declare winners at the end.
4. This activity tests participants' Git skills under pressure and adds an element of friendly competition.

Exercise 16: Git Code Review Tournament
Scenario: Participants engage in a code review tournament where they must review and critique each other's Git commits.

1. Divide participants into pairs or small groups.
2. Assign each group a set of Git commits with code changes.
3. Participants must perform code reviews, providing constructive feedback and identifying potential issues.
4. Encourage discussions and debates about code quality and best practices.
5. The group with the most insightful and constructive reviews wins the tournament.

Exercise 17: Git Repository Surgery
Scenario: Participants must perform advanced surgery on a Git repository to restructure its commit history and branches.

1. Provide a Git repository with a tangled commit history, including unnecessary merges, tangled branches, and inconsistent commits.
2. Challenge participants to reorganize and simplify the commit history while preserving important changes.
3. Tasks may include interactive rebasing, squashing commits, splitting commits, and restructuring branches.

4. Emphasize the importance of preserving a clean and logical commit history in real-world projects.

Exercise 18: Git Performance Optimization
Scenario: Participants are tasked with optimizing the performance of a large Git repository.

1. Provide a Git repository with a large number of commits and files.
2. Challenge participants to identify performance bottlenecks in their Git operations (e.g., cloning, pulling, pushing).
3. Instruct them to implement strategies to optimize the repository's performance, such as shallow clones, Git LFS, and Git's sparse-checkout feature.
4. Discuss best practices for managing large repositories in enterprise-level projects.

Exercise 19: Git Hooks and Automation
Scenario: Participants must create custom Git hooks and automation scripts to streamline their workflow.

1. Introduce participants to Git hooks and automation using scripts (e.g., Bash, Python).
2. Challenge them to create custom pre-commit and post-commit hooks that enforce specific project standards and actions.
3. Instruct them to automate routine tasks, such as generating documentation, running tests, or deploying code, using Git hooks and scripts.
4. Discuss the benefits of automation in ensuring code quality and consistency.

Exercise 20: Git Internals Investigation
Scenario: Participants delve into the internals of Git by examining its object storage and data structures.

1. Provide participants with a Git repository to explore at the object level.
2. Instruct them to examine the .git directory, Git objects, and Git's internal data structures.
3. Challenge them to reconstruct parts of the commit history or recover lost data from Git objects directly.
4. Discuss Git's underlying data model, including how it stores blobs, trees, and commits.
5. This exercise provides a deep dive into Git's inner workings and enhances participants' understanding of Git's architecture.