Exercise: Collaborative Book Writing with Git and GitHub

Requirements:

Before starting the exercise, make sure you have the following:

1. **Git Bash** installed on your Windows machine. You can download it from https://git-scm.com/.
2. A **GitHub account** created at https://github.com/.

   It is suggested to have a github personalized, with your profile setup (including a picture) and a readme file describing you (see here -> https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/customizing-your-profile/managing-your-profile-readme)

Objective:

Students will learn how to collaborate on a project using Git and GitHub. They will experience working with branches, performing merges, and resolving conflicts both locally and on GitHub, by simulating the writing of a book with different chapters and a shared prologue.

Part 1: Initial Setup

0.- Personalize your profile and data on GitHub (Optional)

- Add basic information on your GitHub profile about yourself (see here -> https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/customizing-your-profile/personalizing-your-profile)
- Manage your Profile README and setup some initial info (see here -> https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/customizing-your-profile/managing-your-profile-readme)

Note: Your GitHub is your "portal" to the portfolio of your activities, is highly recommended to have **personalized** GitHub info with your picture photo, and some basic information about you and your projects on the README file of GitHub.

Note also that you can use MarkDown language to improve the formatting of the README. You can find some info about MD language here -> https://medium.com/@saumya.ranjan/how-to-write-a-readme-md-file-markdown-file-20cb7cbcd6f

As examples of GitHub Profiles you can check:

- https://github.com/sofiacostamagna/
- https://github.com/agorosti

1. Create a repository on GitHub

- One student (Student 1) creates a new repository on GitHub called CollaborativeBook.
- Student 1 invites their partner (Student 2) as a collaborator to the repository.

2. Clone the repository

Both students should clone the repository to their local computers:

1. Open **Git Bash**.
2. Run the following command, replacing [URL] with the actual repository link:

```
git clone [URL]

cd CollaborativeBook
```

3. Create the main file for the book

- In **File Explorer**, both students should navigate to the cloned folder CollaborativeBook.
- **Student 1** creates a new text file named book.md by right-clicking inside the folder, selecting **New > Text Document**, and renaming it to book.md. Make sure the file is saved as a .md file (Markdown format) and not .txt.
- Open book.md in any text editor (like Notepad), and **Student 1** should add the following content:

```
# Prologue

(Write the prologue here)

# Chapter 1: (chapter title)

## Section 1.1

## Section 1.2

## Section 1.3

## Section 1.4
```

- Save the file and go back to **Git Bash**. Student 1 should now add and commit the file to the repository:

```
git add book.md
git commit -m "Add basic book structure"
git push origin main
```

Part 2: Collaborating on Chapters and Prologue

4. Create branches for the chapters

Both students should create separate branches to work on different sections of Chapter 1:

- **Student 1** (working on sections 1.1 and 1.2):

```
git checkout -b chapter-1-student1
```

- **Student 2** (working on sections 1.3 and 1.4):

```
git checkout -b chapter-1-student2
```

Each student should add content to their respective sections of Chapter 1 and save their work.

For example:

- **Student 1** edits book.md and writes content for sections 1.1 and 1.2.
- **Student 2** edits book.md and writes content for sections 1.3 and 1.4.

5. Modify the chapter title (to generate merge conflicts)

To create a situation that generates a conflict during the merge, both students should modify the **chapter title** differently:

- **Student 1** changes the title to:

  ```
  # Chapter 1: Introduction to GitHub
  ```

- **Student 2** changes the title to:

  ```
  # Chapter 1: Using GitHub for Collaboration
  ```

Each student should then commit their changes:

```
git add book.md

git commit -m "Update chapter title and add sections"
```

6. Work on the prologue (to generate local conflicts)

Both students should create a new branch to work on the prologue:

```
git checkout -b prologue
```

Each student adds their own content to the prologue.

For example:

- **Student 1** writes the following in the prologue:

  ```
  # Prologue


  This book explains how to use Git and GitHub...
  ```

- **Student 2** writes something different:

  ```
  # Prologue


  In this book, we will learn about collaboration using GitHub...
  ```

Each student should commit their changes:

```
git add book.md
git commit -m "Add prologue"
```

7. Push changes to GitHub and resolve local conflicts

- **Student 1** pushes their prologue changes to the repository:

  ```
  git push origin prologue
  ```

- **Student 2** attempts to push their prologue changes, but Git will notify them that they cannot push because their local branch is out of date. **Student 2** must first pull the latest changes:

  ```
  git pull origin prologue
  ```

During the pull, a **local conflict** will occur because both students edited the same part of the file (the prologue). **Student 2** must manually resolve the conflict by editing book.md to keep the desired changes.

After resolving the conflict, **Student 2** should commit the changes and push them to the repository:

```
git add book.md
git commit -m "Resolve conflict in prologue"
git push origin prologue
```

8. Create Pull Requests and resolve merge conflicts on GitHub

- **Student 1** creates a **Pull Request** to merge their branch chapter-1-student1 into main.
- Once **Student 1**'s Pull Request is merged, **Student 2** creates a **Pull Request** to merge their branch chapter-1-student2 into main.

9. Generate merge conflicts on GitHub

When **Student 2** tries to merge their Pull Request, GitHub will detect a **merge conflict** due to both students modifying the chapter title differently. The conflict will need to be resolved directly in GitHub.

10. Resolving merge conflicts on GitHub

Both students must work together to resolve the conflict in GitHub:

- Go to the "Conflicts" section in the Pull Request.

- Review the differences and select which changes to keep (either combine both titles or select one).
- Resolve the conflict using GitHub's interface.
- After resolving the conflict, complete the merge of **Student 2**'s Pull Request.

Part 3: Summary of Basic Commands and Concepts

Basic Git Commands:

1. **git clone [URL]:** Clones a remote repository from GitHub to your local machine.
2. **git add [file]:** Adds changes to the staging area to prepare for a commit.
3. **git commit -m "[message]":** Commits your changes to the local repository with a message describing the changes.
4. **git push [branch]:** Pushes your local commits to the remote GitHub repository.
5. **git pull [branch]:** Pulls changes from the remote GitHub repository and automatically merges them into your local repository.
6. **git fetch:** Retrieves the latest changes from the remote repository without merging them into your local branch.
7. **git merge [branch]:** Merges changes from one branch into your current branch.
8. **git checkout -b [branch]:** Creates and switches to a new branch for development.

Understanding Git Fetch vs Git Pull:

- **git fetch:**
    - Fetch only **downloads** changes from the remote repository but does **not** apply or merge them into your current working directory. It allows you to see what changes have been made remotely before deciding how to incorporate them.

- Use **fetch** when you want to **check for updates** from the remote repository but don't want to merge them into your local code yet. This is useful when you want to inspect the changes first or handle them manually.
- Example:

```
git fetch
```

After fetching, you can compare or inspect the changes using git log or git diff.

- **git pull:**
  - Pull is essentially a combination of two commands: **git fetch** followed by **git merge**. It first retrieves the changes from the remote repository, and then it attempts to merge them directly into your current working branch.
  - Use **pull** when you're ready to **update your local branch** with the latest changes from the remote repository and want to do so in one step.
  - Example:

```
git pull
```

Differences and When to Use Fetch or Pull:

- **Pull (fetch + merge) in one step:**
  - Use **pull** when you are ready to incorporate all the remote changes into your local branch without manual intervention.
  - It's a quick and easy way to ensure your local branch is up to date.
- **Fetch (without merging):**
  - Use **fetch** when you are working on something locally and want to check for updates without affecting your current work. It allows you to review changes before deciding whether to merge them into your branch.
  - After using fetch, you can manually review the changes and then apply them with:

```
git merge origin/[branch]
```

  - So, a **fetch followed by a merge** gives you more control compared to a direct pull.

Is git pull equivalent to git fetch and then git merge?

- Yes! git pull is exactly the same as running git fetch followed by git merge automatically.
- However, if you want more control over the merge (e.g., reviewing the fetched changes first), you can run fetch and merge separately instead of doing a pull.

Branching Conventions in Real Development:

In real-world development workflows, branching conventions help keep code organized and maintain a smooth development process. Here are common branch types and conventions:

- **Feature branches (feature/name):**
    - When working on a new feature, developers create a feature branch (e.g., feature/user-login) to isolate changes related to that feature. This keeps the main codebase stable while the feature is being developed.
    - Example:

      git checkout -b feature/user-login

- **Development branch (development):**
    - This branch is used as an integration point for various feature branches. Features are merged into this branch for testing and review before they are pushed to the production environment.
- **Main branch (main or master):**
    - This is the **production-ready** branch. It should always contain stable and well-tested code. Only tested changes from the development branch are merged here.

Example workflow:

1. Create a new feature branch (e.g., feature/new-functionality).
2. Develop and test the feature.
3. Open a Pull Request to merge the feature into development.
4. After reviewing and testing in development, merge into main for production deployment.

How Does This Apply to the Exercise?

In this exercise, the branches created are more simplistic (prologue, chapter-1-student1, etc.), and they do not follow the standard feature branching model that is typically used in professional development workflows. However, you could easily adapt this exercise to follow best practices by:

- **Creating feature branches for each task:**
    - For example, instead of naming a branch chapter-1-student1, you could name it feature/chapter1-sections-1.1-1.2.
    - Instead of prologue, you could use feature/prologue.
- **Using a development branch:**
    - In the real world, instead of merging branches directly into main, you would merge them into a development branch first for testing. Once all features are stable and have been reviewed, you would merge development into main.
- **Branching Workflow Adaptation:**
    - Adapt the exercise by:
        1. Creating a **development** branch and having students merge their feature branches into development.
        2. Once the chapters are fully written and tested, the students would open a Pull Request to merge development into **main**.

This approach ensures that the codebase remains stable and isolated from incomplete features, which is a best practice for large-scale projects.

Conclusion

1. **Submission**: Once the conflicts are resolved and both Pull Requests are merged, the students should submit the link to their repository as the final deliverable.
2. **Review**: Review the final version of book.md in the main branch to ensure all changes have been correctly merged.

This exercise provides a comprehensive experience with Git workflows, focusing on the fundamental commands, the process of merging changes, resolving conflicts, and best practices for branching.

Simplified Rubric for Evaluating the Git and GitHub Collaboration Exercise

Once you have finished the exercise, you should upload a document to canvas, providing the URL of the GitHub repository with the exercise, and explaining with issues you found during the exercise, and how did you solve them. Please note that the following rubric will be used to grade the exercise:

Criterion	Excellent (9-10 points)	Good (7-8 points)	Acceptable (5-6 points)	Needs Improvement (0-4 points)
1. Use of Git and GitHub	Correct and smooth use of key commands (add, commit, push, pull, merge) with no errors.	Mostly correct use of commands with minor errors.	Basic use of commands with frequent errors, but fixable.	Constant difficulties using commands, requiring continuous assistance.
2. Conflict Resolution	Local and remote conflicts identified and resolved autonomously and effectively.	Conflicts were resolved with some minor errors or occasional help.	Significant help needed to resolve conflicts, with limited understanding.	Conflicts were not resolved properly, requiring total assistance.
3. Understanding of Git Workflow	Clear and complete understanding of branches, merges, and Pull Requests, with precise explanations.	General understanding of the workflow, with minor gaps.	Basic understanding of the workflow, but difficulties with branches or PRs.	Little to no understanding of the Git workflow, frequent errors or confusion.
4. Quality of the Final Project	Final book.md file is well-structured, clear, and error-free. Project	Project well-structured, with minor details to fix. Submitted on time.	Project is readable but contains significant errors. Submitted late.	Project is poorly structured or incomplete. Submitted late

Criterion	Excellent (9-10 points)	Good (7-8 points)	Acceptable (5-6 points)	Needs Improvement (0-4 points)
	submitted on time.			or not submitted.

Optional Criterion (Bonus)

Criterion	Excellent (1 additional point)	Good (0.75 additional points)	Acceptable (0.5 additional points)	Needs Improvement (0.25 additional points)
GitHub Profile Quality	Profile fully complete with photo, well-written bio, and an attractive and functional README.	Profile is properly configured with minor shortcomings.	Profile created with basic information but lacking significant details.	Profile created but incomplete or with very little information.

Total Score: /40 points + 1 optional bonus point

Grading Scale (out of 10 points):

- 36-40 points: 10
- 32-35 points: 9
- 28-31 points: 8
- 24-27 points: 7
- 20-23 points: 6
- 16-19 points: 5
- Less than 16 points: 4 or below

Bonus: Up to **1 additional point** for GitHub profile quality (photo, bio, well-structured README, etc.).