PayPal Pre-Work Notes

Git and Github

Introduction

Git is the most popular version control system to exist. With a version control system, we can track our **code history** and **work together**. There are centralized as well as distributed version control systems.

Some **centralized** version control systems are:

1) Subversion
2) Microsoft Team Foundation server

Some **distributed** version control systems are:

1) Git
2) Mercurial

Reasons to choose git over other version control systems:

1) Free
2) Open Source
3) Super Fast
4) Scalable
5) Cheap Branching/Merging

Using Git

The Git can be used in several ways. They are mentioned in the points below:

1) The Command Line- Fastest Way
2) Code editors and IDE
3) Graphical User Interfaces(GUI)

Why Command Line(CLI) for Git

1) GUI Tools have limitations
2) GUI Tools are not always available (E.g.- remotely accessing the server)

3) Faster to use Command Line

Configuring Git

Before using Git, we have to specify a few configuration settings:

1) Name
2) Email
3) Default Editor
4) Line Endings

We can apply these settings at these levels:

1) System: The settings here apply to all the users
2) Global: The settings here apply to all the repositories
3) Local: The settings here apply to the current repository

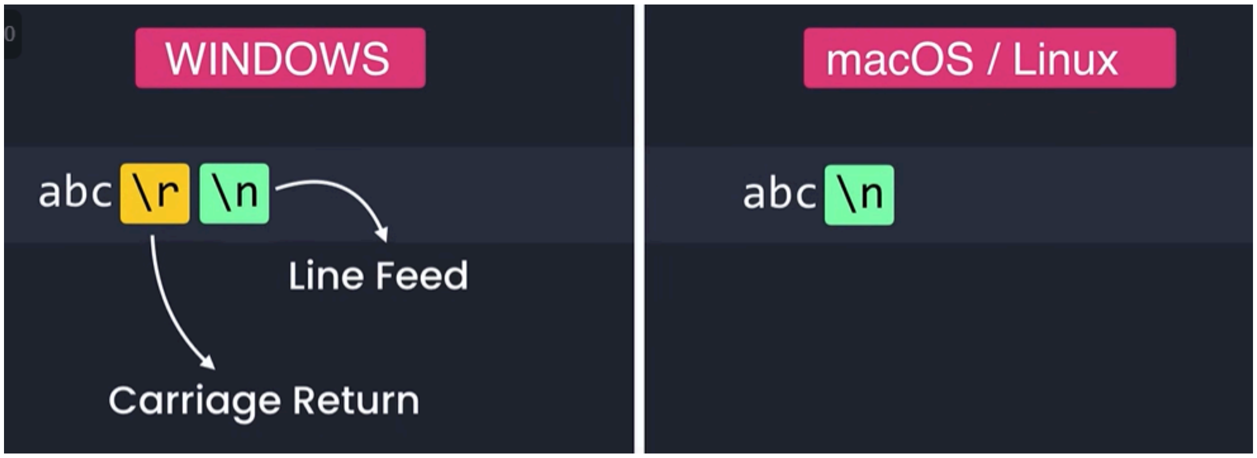
Git Config Commands

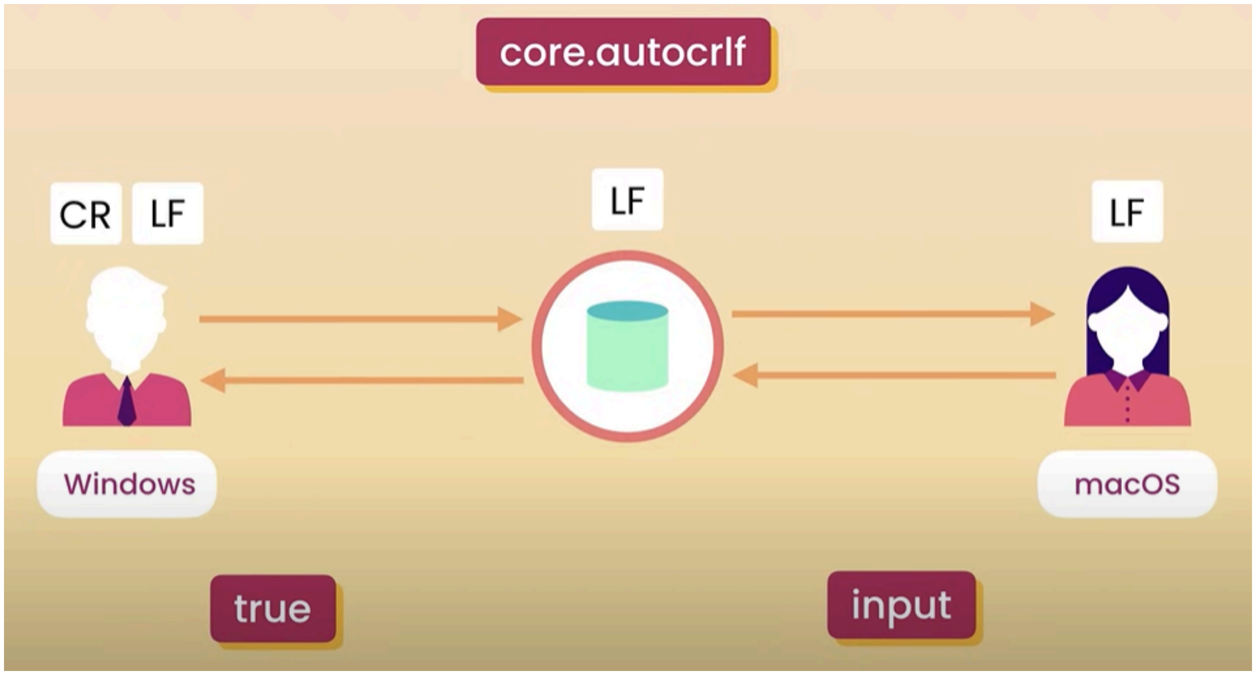
These commands take the configuration and save it in the file named .gitconfig.

Git Command	Action
git config --global user.name "Name"	This configures the user name
git config --global user.email email@gmail.com	This sets the user email in the configuration file
git config –global core.editor "code –wait"	This will set the code editor as VS code(VS code is named as code in PATH variable). wait will make sure to wait until we close the VScode instance ourself.
git config –global -e	This will open the .gitconfig file in code editor
git config –global core.autocrlf input	This will set the auto carriage return line feed to input(recommended for macOS)(More information provided in the note below)
git config --help	Will provide all possible documentation codes for git config

	command
git config -h	Gives a short summary of what the help command does

Note: In windows system, end of line is marked with two special characters, which are carriage return and Line Feed. In macOS/Linux, end of line is only marked with Line Feed. So we need to properly configure the end of line property.





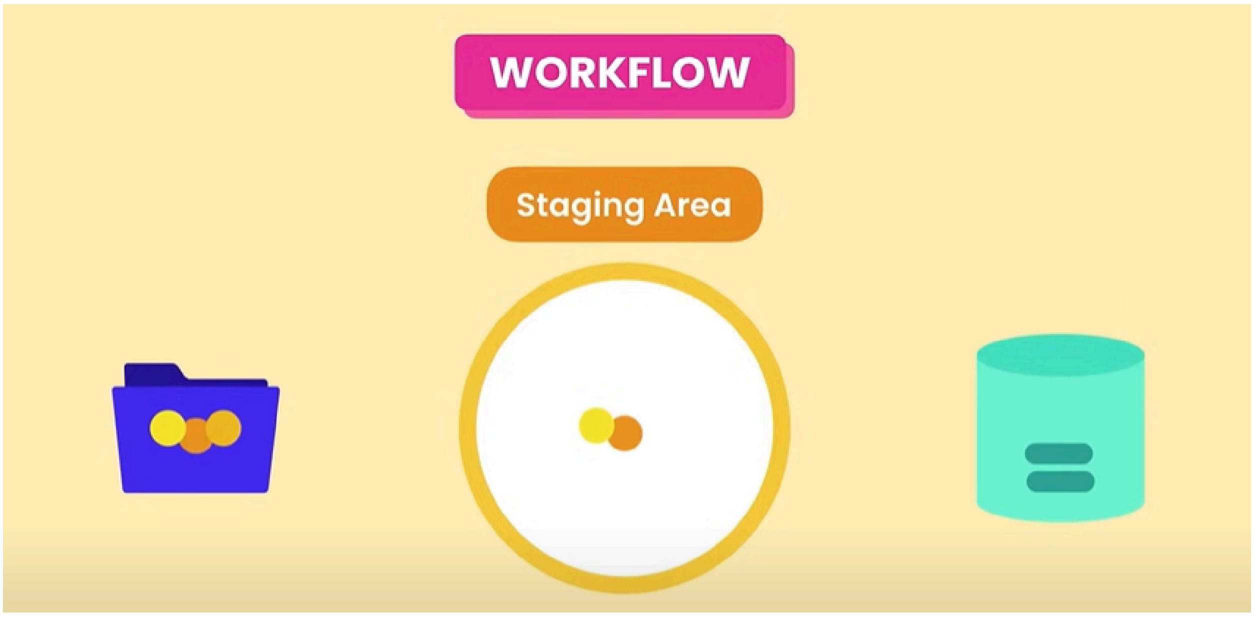
Initializing a Repository

Follow these steps to create a repository:

1) Create a local directory using ***mkdir*** command and go inside the repository.

2) In the directory, initialize the repository by command: ***git init***
   - Note: It creates a new subdirectory .git, which is hidden.
   - This repository is still not committed to github yet

Git Workflow



In Git, we have a folder in our **local system** and we have a **git repository** as a subdirectory in the project folder. In Git, we have a special **staging area**, in which we decide which files to commit to the repository. Then we commit this snapshot to our repository.

WORKFLOW
file1
file2
file1
file2

Git Command	Action
touch file1 file2	This creates a file in the project folder
git add file1 file2	This adds the file1 and file2 to the staging area
git add *.txt	Adds all the .txt files to the staging area
git add .	Add all the files to the staging area. **Note:** As you don't want to add all the files like data files or etc.(Put them in .gitignore)
git commit -m "Initial commit"	This commits the snapshot of the staging area to the git repository. The "Initial commit" is a message which represents what the snapshot represents.
git commit	This will open a commit file in the code editor. You can type your commit message there(one short msg and one long description) if you need.
git status	Give the information about the files which are being tracked and which are not in the staging area.

Note: The staging area remains **unchanged**. We have to manually make changes to it using the add command for all the files (If not, it will have the previous version of the file). Also if we **remove** the file from the folder, we need to use the add command to remove the file from the staging area.

Each commit contains the following:

1) ID: Unique identifier for the commit
2) Message: The commit message
3) Date/Time
4) Author
5) Complete Snapshot: The snapshot of the entire project folder. Stores not only the changes, but the whole project.

Commit Best Practices

1) Don’t commit **too small** or **too large** of a change. A good practice is to record a snapshot around 3-5 times a day, which represent different stages of the project.
2) Commits representing different changes should be done differently. For Example: commit a bug and a typo mistake in different commits.
3) Use past tense in the commit message. For example, use fixed the bug.

Committing without Staging

We can commit the changes without putting the files into the staging area.

Git Command	Action
git commit -am “Message”	This commits all the files without putting them in the staging area with message
git ls-files	This shows all the files which are in the staging area.
git rm file	This removes the file from the local folder as well as the staging area.
git mv file1 file2	This renames the file1 to file2 in both the local folder and the staging area.

Ignoring the Files

Some files need to be ignored to be committed to the git. **For example:** Log files or data files.

Git Command	Action
echo folder/ >> .gitignore	This appends the folder/ in the gitignore extension.
git rm –cached -r folder/	This removes the folder from the staging area.(Used in case you want to remove the

	file from the staging area without removing it from the main folder)

Note: We can type -h in front of any command if we want to know what we can type in front of it.
You can check all the various .gitignore templates here: [github/gitignore: A collection of useful .gitignore templates]

Short status

Gives the short status of the repository.

Git Command	Action
`git status -s`	This gives the short status of the repository.

Changes between staged and unstaged area

Git Command	Action
`git diff --staged`	This gives the difference between files in the repository and the staging area.
`git diff`	This gives the difference between files in the working directory and the staging area.
`git config --global diff.tool vscode`	This sets the default visual tool for seeing differences to vscode.
`git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"`	This sets the difftool in vscode. The PATH variable is set as code for VS Code. (LOCAL and REMOTE tells to compare the local and the remote copy)
`git difftool`	This opens the difftool with files in the working directory and staging area.

Some visual tools for seeing difference are:

1) KDiff3
2) P4Merge
3) WinMerge (Only on Windows)
4) VS Code

History

Git Command	Action
git log	This command helps us view the different log commands.
git log --oneline	This commands shows us the short summary of all the commits.
git log --oneline --reverse	This command shows us the short summary of all the commits in the reverse order(The initial commit at the top).
git log --all --graph	Show branching visually in the command line
git show 09ufd	This shows the commit information using the id(We don't need to type full ID)
git show HEAD	This shows the commit corresponding to the head pointer(which is on the last commit)
git show HEAD~x	This shows the commit corresponding to the headpointer -x commit.(If x= 1, it shows the previous commit)
git show HEAD~x:.gitignore	This command shows the .gitignore file in the xth commit preceding the latest commit.
git ls-tree HEAD	This shows all the files and directories stored in the latest commit.(Files represented using BLOB and directories are represented using trees).

Using the show command, we can view any object in git's database, which can be:

1) Commits
2) Blobs : Files in the commit
3) Trees: Directories in the commit
4) Tags

Unstaging Files

Git Command	Action
`git restore --staged file1`	This command restores the file1 from the repository (last commit) to the staging area.

Discarding Local Changes

Git Command	Action
`git restore file1`	This command restores the file1 from the staging area to the working directory.
`git clean -fd`	This command allows git to remove the untracked files from the staging area.(f stands for force and d for whole directories)

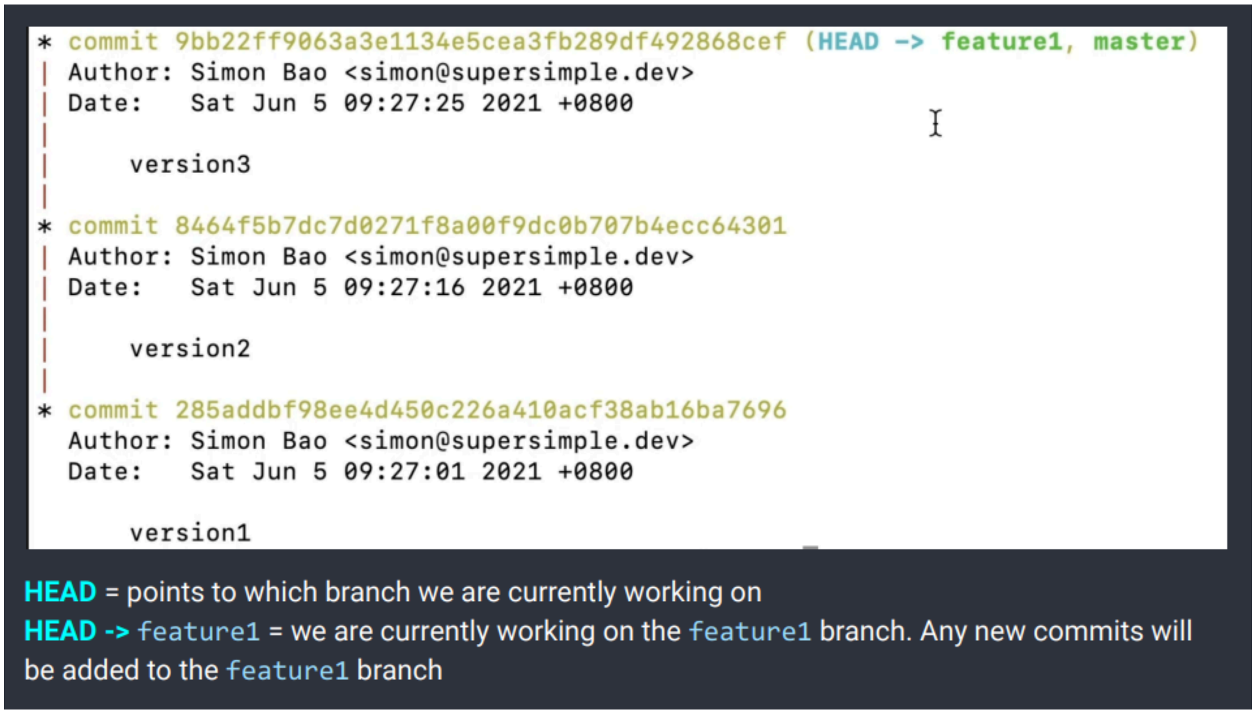
Restoring a File to a Earlier Version

Git Command	Action
`git restore --source=HEAD~1 file1`	This command restores the file1 from the previous commit the working directory

Git Branching

Git branching allows you to create and manage separate lines of development within a repository, enabling isolated changes and parallel feature development.

Git Command	Action
git branch	Shows a list of available branches
git log --all --graph	Shows the branches visually in the history
git branch feature1	Create a new branch named feature1
git checkout feature1	Switch to the feature1 branch. New commits will now be added to the feature1 branch



```
* commit 9bb22ff9063a3e1134e5cea3fb289df492868cef (HEAD -> feature1, master)
| Author: Simon Bao <simon@supersimple.dev>
| Date:   Sat Jun 5 09:27:25 2021 +0800
|
|     version3
|
* commit 8464f5b7dc7d0271f8a00f9dc0b707b4ecc64301
| Author: Simon Bao <simon@supersimple.dev>
| Date:   Sat Jun 5 09:27:16 2021 +0800
|
|     version2
|
* commit 285addbf98ee4d450c226a410acf38ab16ba7696
  Author: Simon Bao <simon@supersimple.dev>
  Date:   Sat Jun 5 09:27:01 2021 +0800

      version1
```

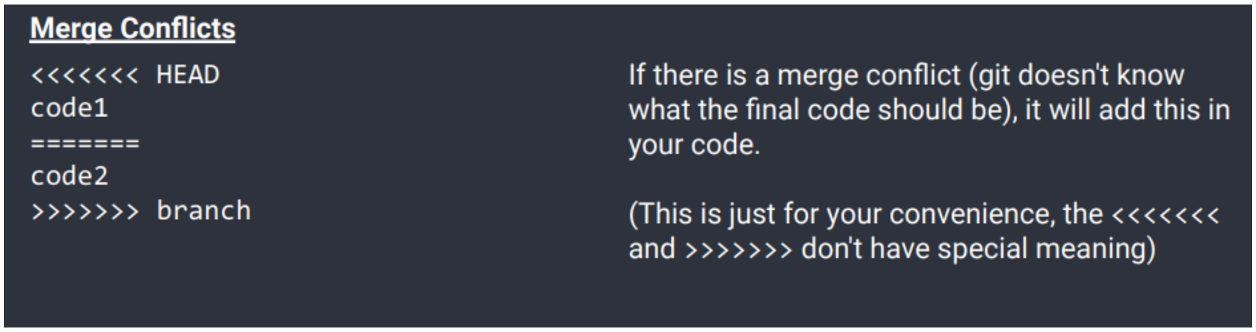
HEAD = points to which branch we are currently working on
HEAD -> `feature1` = we are currently working on the `feature1` branch. Any new commits will be added to the `feature1` branch

Git Merging

Git merging is the process of integrating changes from one branch into another within a Git repository.

Git Command	Action
git merge <branch name> -m "message"	Merge the current branch (indicated by HEAD ->) with another branch (<branch name>). Saves the result of the merge as a commit on the current branch.

Merge Conflict

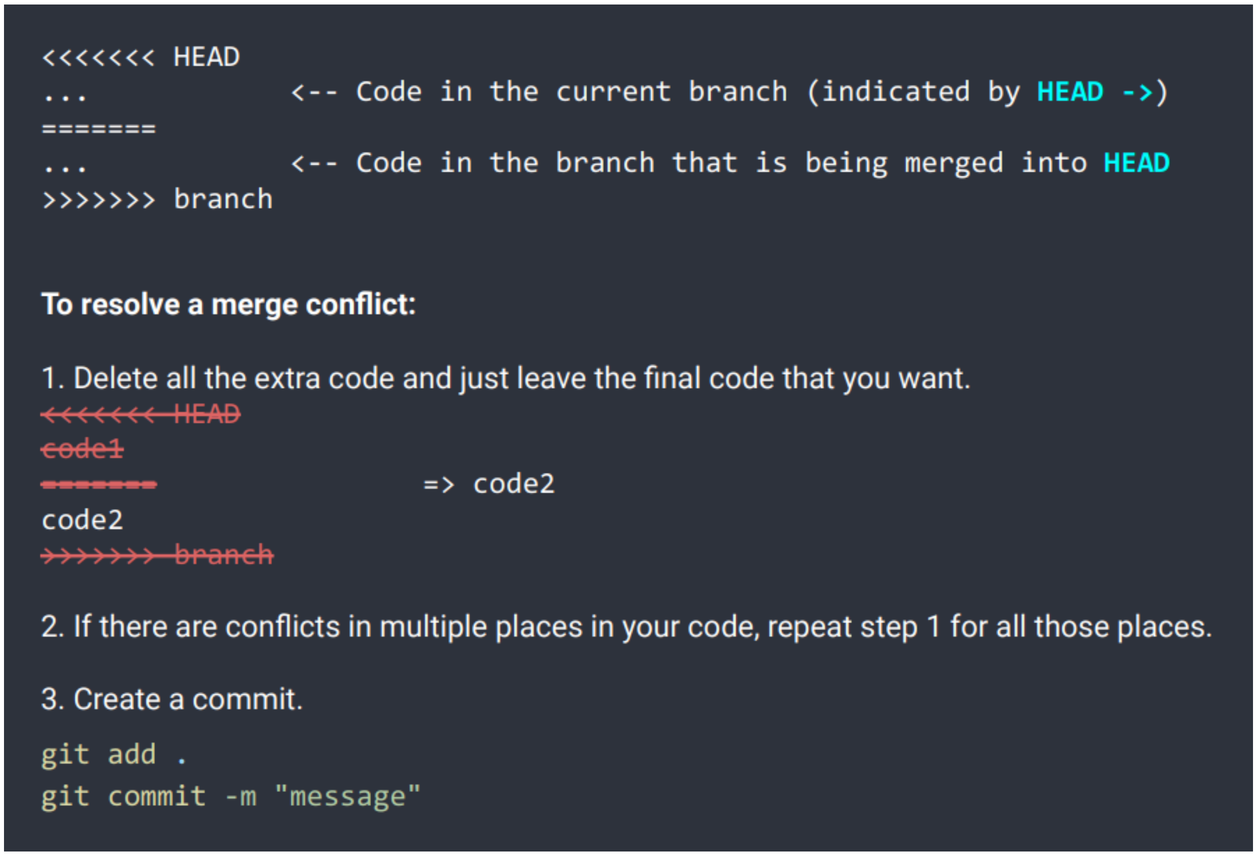


Merge Conflicts

```
<<<<<<< HEAD
code1
=======
code2
>>>>>>> branch
```

If there is a merge conflict (git doesn't know what the final code should be), it will add this in your code.

(This is just for your convenience, the <<<<<<< and >>>>>>> don't have special meaning)



```
<<<<<<< HEAD
...            <-- Code in the current branch (indicated by HEAD ->)
=======
...            <-- Code in the branch that is being merged into HEAD
>>>>>>> branch
```

To resolve a merge conflict:

1. Delete all the extra code and just leave the final code that you want.

~~<<<<<<< HEAD~~
~~code1~~
~~=======~~ => code2
code2
~~>>>>>>> branch~~

2. If there are conflicts in multiple places in your code, repeat step 1 for all those places.

3. Create a commit.

```
git add .
git commit -m "message"
```

Feature Branch Workflow

The feature branch workflow in Git is a development strategy where each new feature is developed in its own branch, separate from the main branch, allowing for isolated development, testing, and review before merging into the main codebase.
You can read it from the second cheat sheet. Also read about the **merging conflicts part.**

Cheat Sheet Link

1) https://drive.google.com/file/d/1Ol5doO3tDxPwj5dtwgIVPB39I_5z3ldO/view?usp=sharing
2) git-github-reference.pdf (supersimpledev.github.io)