GIT

aiimranh

aiimranh

aiimranh

VCS

INTRO

VCS

INTRO

Intro:
Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time. As development environments have accelerated, version control systems help software teams work faster and smarter.

VCS

TYPES

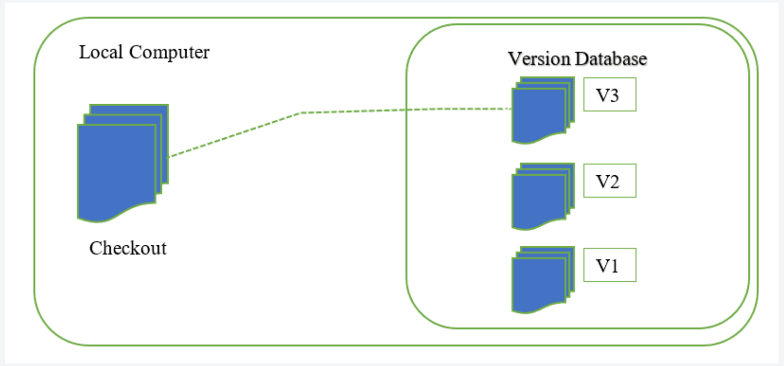
The various types of the version control systems are:

1. Local Version Control System
2. Centralized Version Control System
3. Distributed Version Control System

VCS

Local Version Control System

Local version control system maintains track of files within the local system. This approach is very common and simple. This type is also error prone which means the chances of accidentally writing to the wrong file is higher.

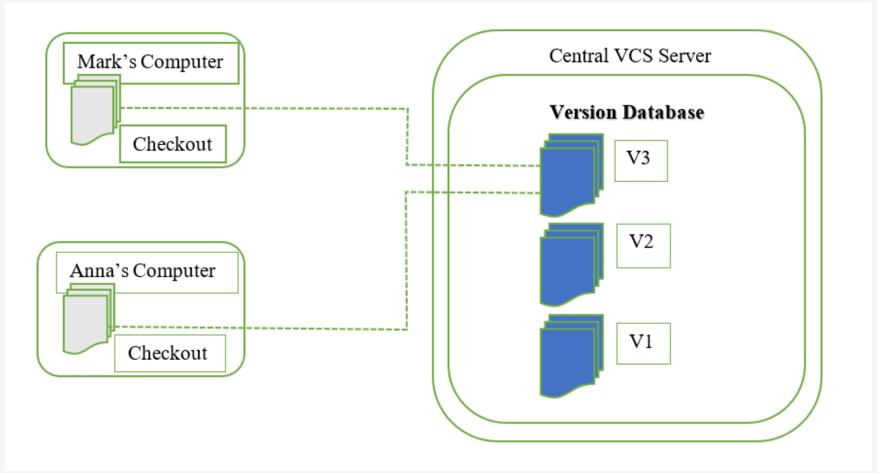


VCS

Centralized Version Control System

In this approach, all the changes in the files are tracked under the centralized server. The centralized server includes all the information of versioned files, and list of clients that check out files from that central place.

Example: Tortoise SVN, CVS and Subversion

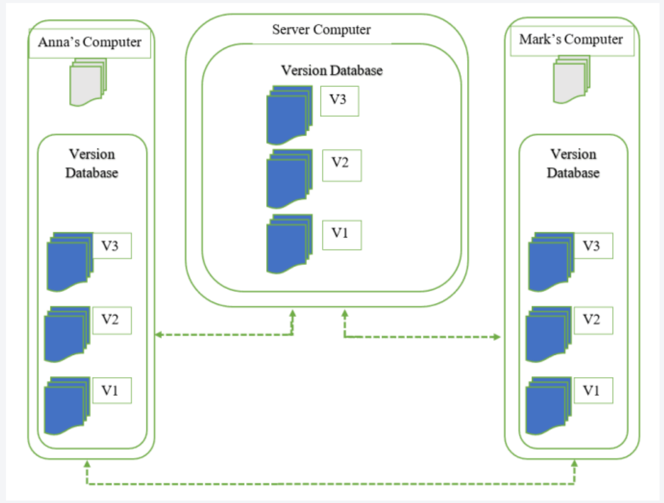


VCS

Distributed Version Control System

Distributed version control systems come into picture to overcome the drawback of centralized version control system. The clients completely clone the repository including its full history. If any server dies, any of the client repositories can be copied on to the server which help restore the server.

Example: Git, Mercurial & Bazaar or Darcs



GIT

INTRO

GIT

INTRO

- ✓ **Git** is a Distributed Version Control System (DVCS) designed to make it easier to have multiple versions of a code base, sometimes across multiple developers or teams.
- ✓ It allows you to see changes you make to your code and easily revert them.
- ✓ It is not **GitHub.**

GIT

PRE-REQUIRED

COMMAND

GIT[1]

DIRECTORY MANAGEMENT

View List Directories

$ **ls**	[to normal view]
$ **ls --help**	[to get all ls command details]
$ **ls -a**	[to all view including hide files]
$ **ls -la**	[to view all files in more details]
$ **ls -la /home/imran**	[to view all files in imran folder]

GIT[2]

DIRECTORY MANAGEMENT

Changing Directories

$ **pwd**	[to know current directory]
$ **cd ..**	[to change directory one level up]
$ **cd Desktop**	[to change directory one level down]
$ **cd ..\Desktop**	[to change current directory another folder]
$ **cd ~**	[~ indicate user directory]
$ **cd ~\Desktop**	[to change Desktop directory]
$ **cd E:\DocumentX**	[to go DocumentX folder]

GIT[3]

DIRECTORY MANAGEMENT

Make New Directories

$ **mkdir “xx xx”**	[to make [xx xx] folder in current location]
$ **mkdir ‘xx xx’**	[to make [xx xx] folder in current location]

Copy Files/Folder to Other Directory

$ **cp new ~\Desktop**	[to copy only new folder]
$ **cp new ~\Desktop -Verbose**	[to make same name directory on target folder]
$ **cp *.jpg ~\Desktop**	[to copy all jpg file]
$ **cp new ~\Desktop -Recurse -Verbose**	[to copy new folder including sub folder and file]

GIT[4]

DIRECTORY MANAGEMENT

Move and Renaming Directory

$ **mv xx.txt xy.txt**	[to change name xx.txt to xy.txt]
$ **mv videos ~\Desktop**	[to move videos to Desktop]
$ **mv *.jpg ~\Desktop**	[to move all jpg at a time]

Removing and Directory

$ **rm rr.txt**	[to remove rr.txt]
$ **rm system -Force**	[to remove administration file]
$ **rm new -Recurse**	[to remove the folder with sub folder]

GIT[5]

DIRECTORY MANAGEMENT

Command History

$ **history**	[to check all command that are used, up and down tor navigate]
Ctrl + R	[to search used command]

Display file content

$ **touch x.txt**	[to create an empty content file]
$ **cat ttt.txt**	[to view content file]

GIT[6]

DIRECTORY MANAGEMENT

Display file content

```
$ less xx.txt
```

[to view file in one page]
[Enter key enhanced to one line]
[space key enhanced to one page]
[q key to quit the file]
[g – to begging of the file]
[G- to end of the file][page up and page down key]
[/word_search to search word]

GIT[7]

DIRECTORY MANAGEMENT

Display file content

$ **head xx.txt**	[to view first 10 lines of the file]
$ **tail xx.txt**	[to view last 10 lines of the file]

Modifying Text File

$ **nano xx.txt**	[to view xx.txt file] [^G indicate Ctrl+G]
$ **notepad xx.txt**	[to view xx.txt file on notepad]

Searching within Files

$ **grep cow xx.txt**	[to view cow word in xx.txt]
$ **grep cow *.txt**	[to view cow word all .txt]

GIT[8]

DIRECTORY MANAGEMENT

Input, Output & Pipeline

$ **echo woof > dog.txt**	[to add woof word in dog.txt]
$ **echo woof 1> dog.txt**	[to add woof word in dog.txt] [1 indicate stdout]
$ **echo woof >> dog.txt**	[to add woof word in dog.txt without erasing]
$ **cat dog.txt | grep new**	[to view new matched word in dog.txt] [| indicates the pipeline operator]

GIT[9]

DIRECTORY MANAGEMENT

Input, Output & Pipeline

$ **cat < dog.txt | grep new** [to view new matched word in dog.txt] [**|** indicates the pipeline operator][

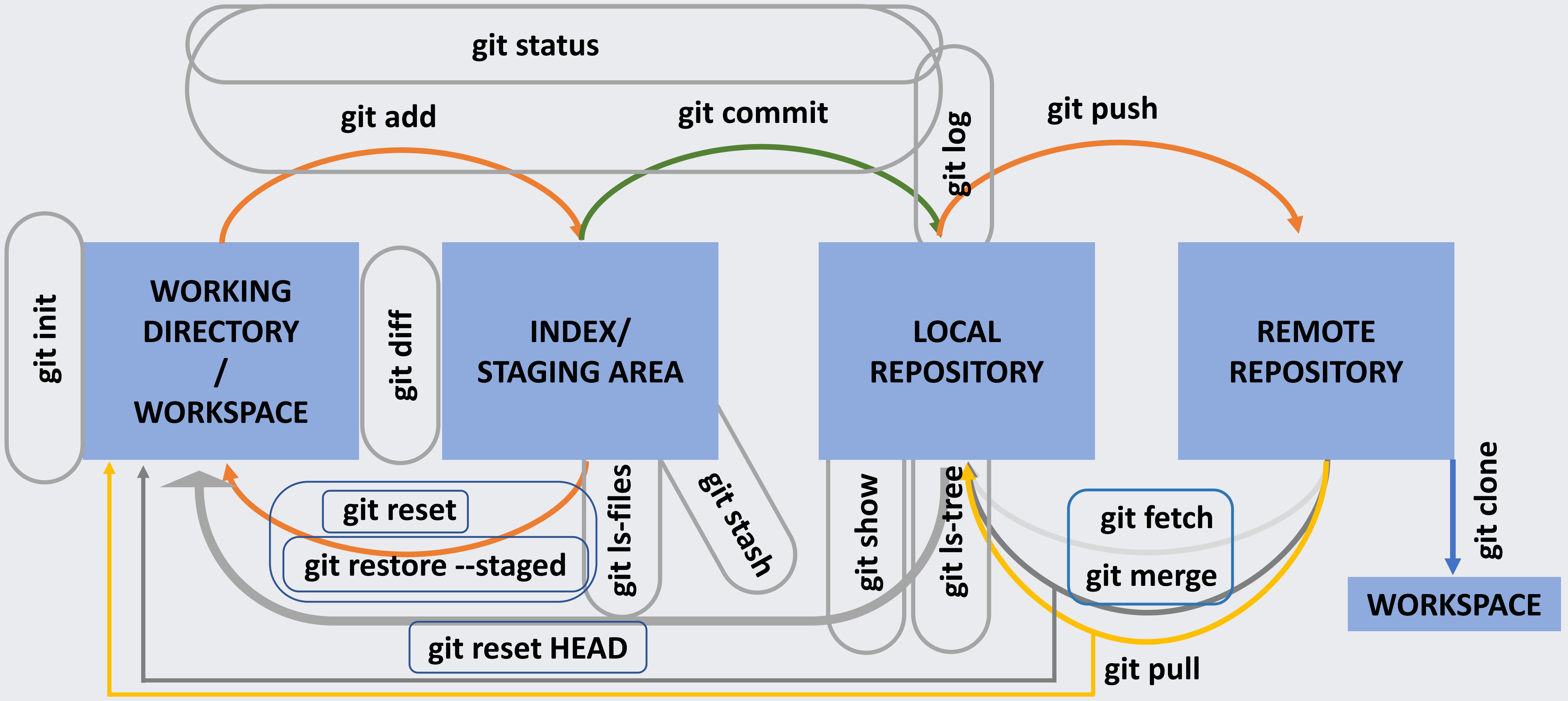
$ **cat dog.txt | grep new > dog2.txt** [to save new matched word in dog2.txt]

$ **rm secure_file 2> error.txt** [to save error message in error.txt] [2 indicate stderr]

GIT

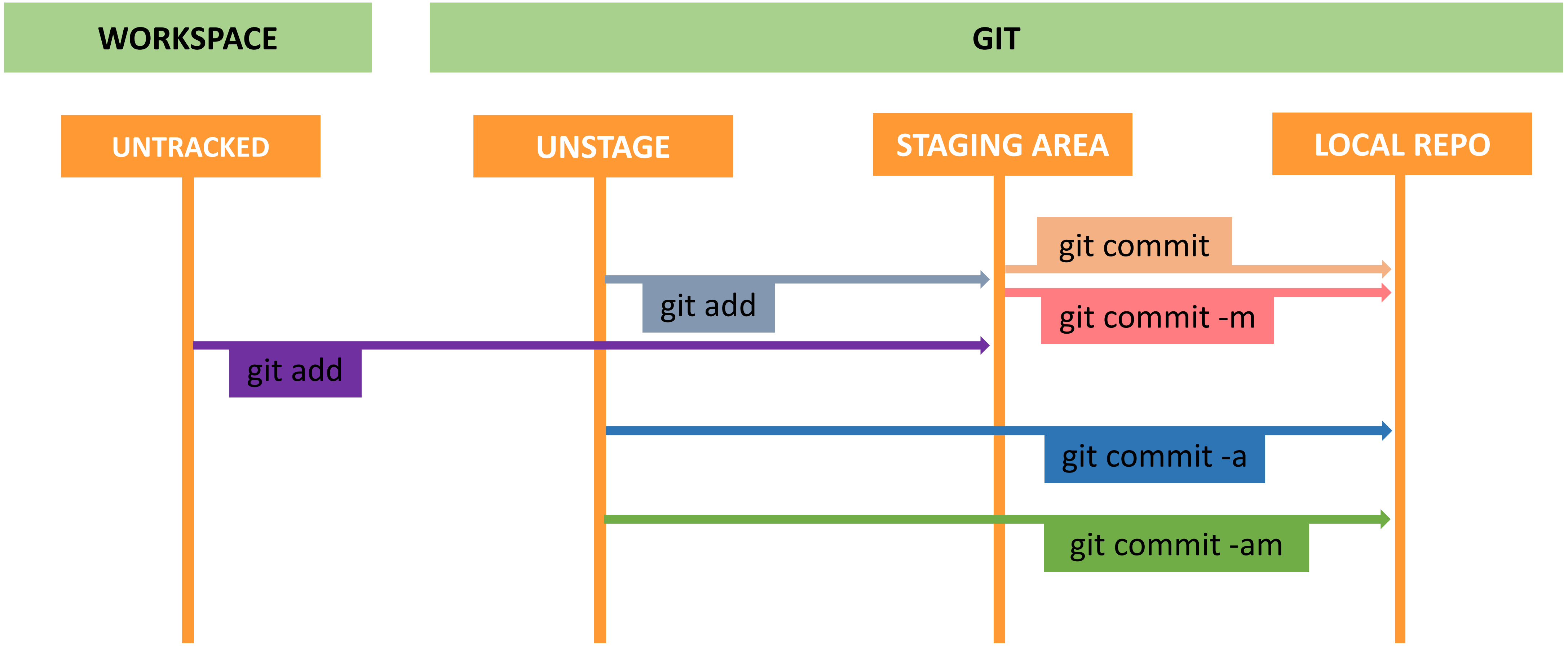
WORKFLOW

GIT WORKFLOW



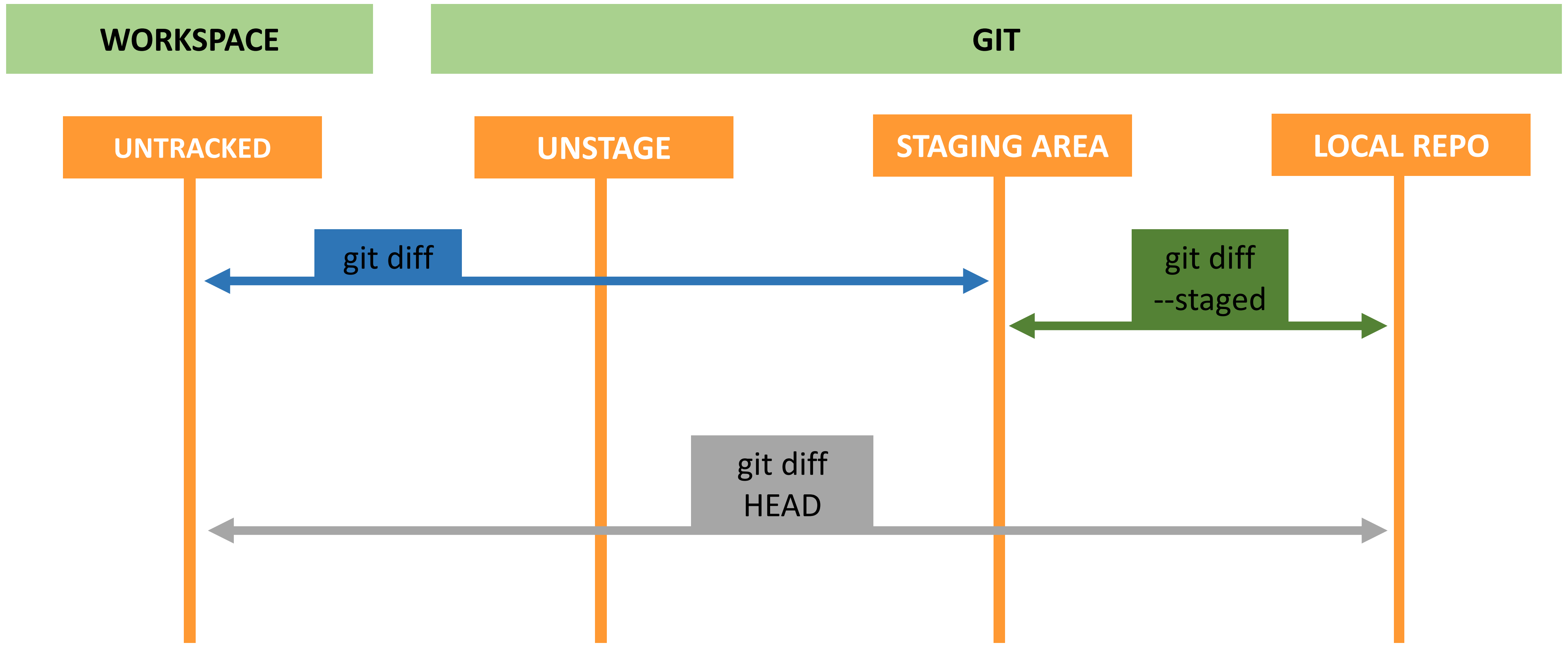
GIT WORKFLOW

FORWARD



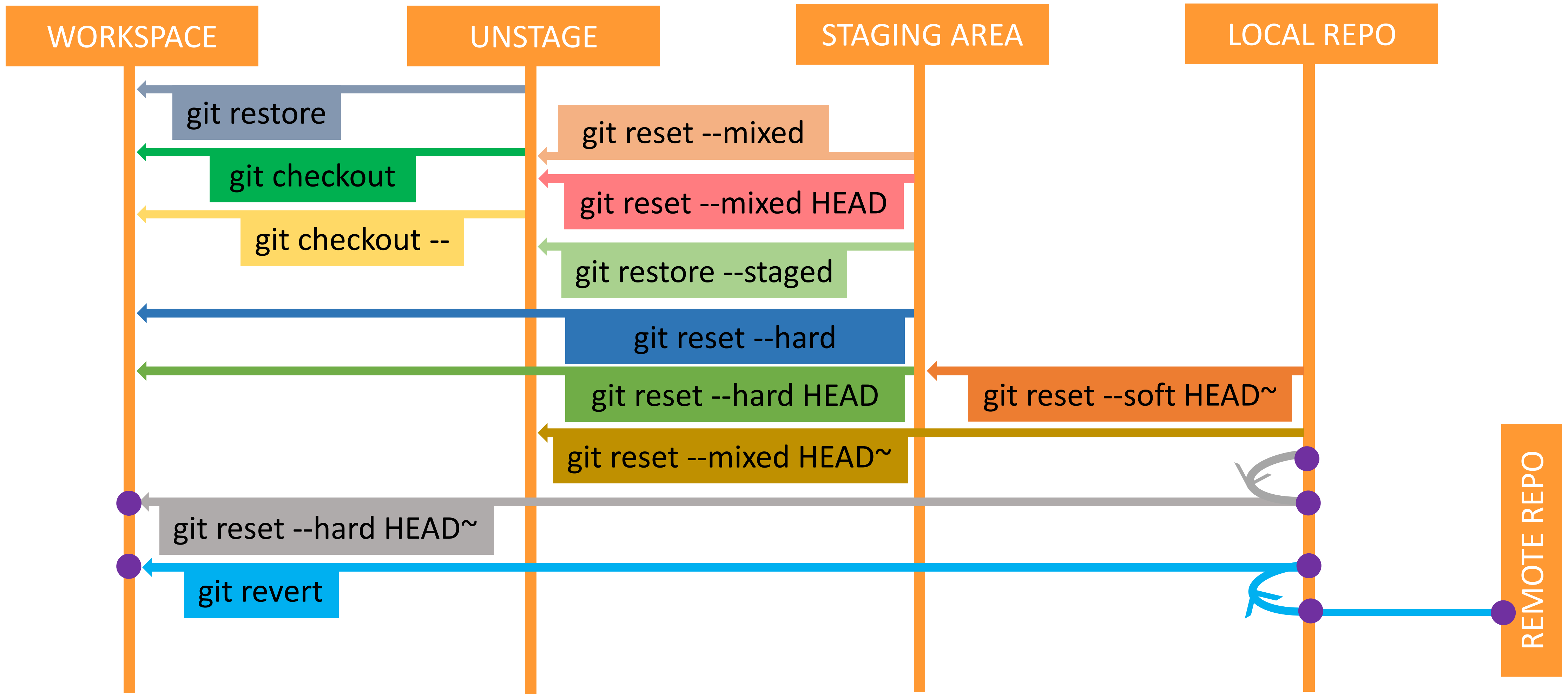
GIT WORKFLOW

DIFF



GIT WORKFLOW

UNDO



GIT
CONFIGURATION
COMMAND

GIT INIT

GIT INITIALIZATION

- Open ***Git Bash***
- *Go To The Desire Directory*
- Create A Blank New Repository & Store Source in Current Folder.

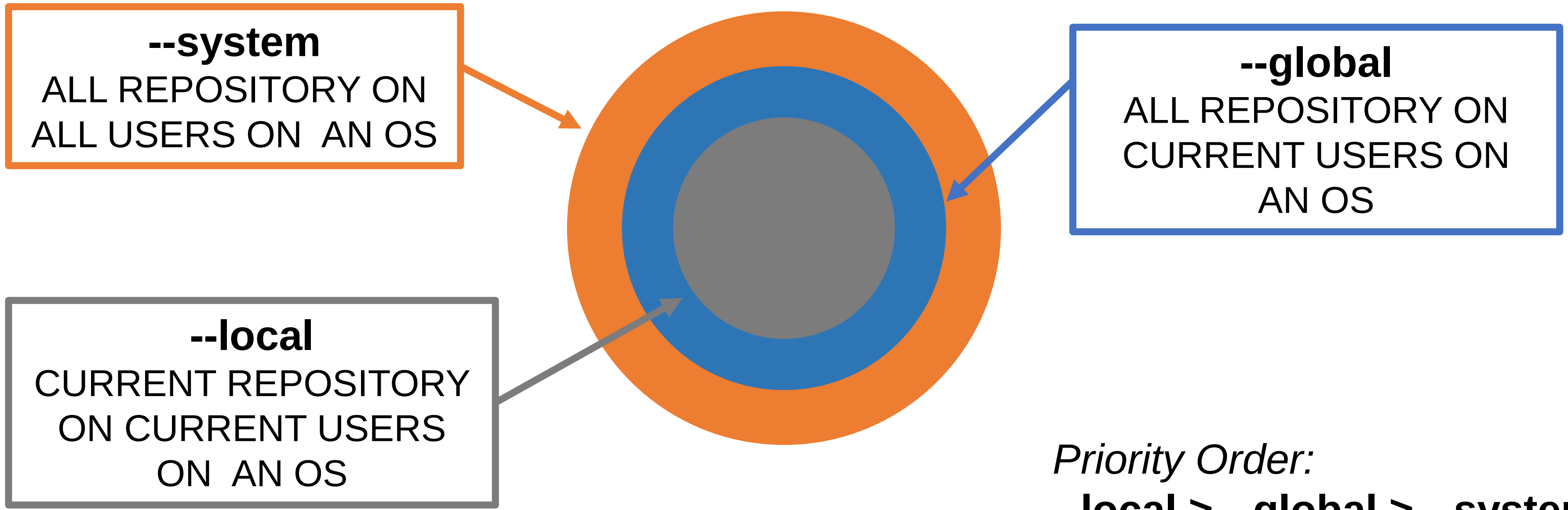
```
$ git init
```

- Create A Blank New Repository & Store Source in A New Folder.

```
$ git init test
```

GIT CONFIG

CONFIGURATION LEVEL


--system
ALL REPOSITORY ON ALL USERS ON AN OS
--global
ALL REPOSITORY ON CURRENT USERS ON AN OS
--local
CURRENT REPOSITORY ON CURRENT USERS ON AN OS

Priority Order:
--local > --global > --system

GIT CONFIG[1]

SYSTEM LEVEL

- ❖ identify yourself at system level

  ```
  $ git config --system user.name “imranh37”
  $ git config --system user.email “hmimran.in@gmail.com”
  ```

- ❖ check system level user & email respectively

  ```
  $ git config --system user.name
  $ git config --system user.email
  ```

GIT CONFIG[2]

SYSTEM LEVEL

- check all system level configuration
  $ **git config --system --list**
- check all the properties of system level configuration
  $ **git config --system --list --show-origin**
- edit the system config file
  $ **git config --system -e**

GIT CONFIG[1]

GLOBAL LEVEL

- ❖ identify yourself at global level

```
$ git config --global user.name “imranh37”
$ git config --global user.email “hmimran.in@gmail.com”
```

- ❖ check global level user & email respectively

```
$ git config --global user.name
$ git config --global user.email
```

GIT CONFIG[2]

GLOBAL LEVEL

- ❖ check all global level configuration[1]
  $ **git config --global --list**
- ❖ check all the properties of global level configuration[2]
  $ **git config --global --list --show-origin**
  [note]: in global level [1] & [2] are equal.
- ❖ edit the global config file
  $ **git config --global -e**

GIT CONFIG[1]

LOCAL LEVEL

- ❖ Required: $ **git init**
- ❖ identify yourself at local level

  $ **git config --local user.name "imranh37"**

  $ **git config --local user.email "hmimran.in@gmail.com"**
- ❖ check local level user & email respectively

  $ **git config --local user.name**

  $ **git config --local user.email**

GIT CONFIG[2]

LOCAL LEVEL

- ❖ check all local level configuration[1]
  $ **git config --local --list**
- ❖ check all the properties of local level configuration[2]
  $ **git config --local --list --show-origin**
  [note]: in local level [1] & [2] are equal.
- ❖ edit the local config file
  $ **git config --local -e**

GIT

EDITOR

COMMAND

GIT EDITOR

USE OF VIM

Hit the **Esc** key to enter "Normal mode". Then you can type **:** to enter "Command-line mode". A colon (**:**) will appear at the bottom of the screen and you can type in one of the following commands. To execute a command, press the **Enter** key.

:q to quit (short for **:quit**)
:q! to quit without saving (short for **:quit!**)
:wq to write and quit
:wq! to write and quit even if file has only read permission (if file does not have write permission: force write)
:x to write and quit (similar to :**wq**, but only write if there are changes)
:exit to write and exit (same as **:x**)
:qa to quit all (short for **:quitall**)
:cq to quit without saving and make Vim return non-zero error (i.e. exit with error)

You can also exit Vim directly from "Normal mode" by typing **ZZ** to save and quit (same as **:x**) or **ZQ** to just quit (same as **:q!**). (Note that case is important here. **ZZ** and **zz** do not mean the same thing.)
Vim has extensive help - that you can access with the **:help** command - where you can find answers to all your questions and a tutorial for beginners.

GIT EDITOR

RESET TO VIM

Set **core.editor** in your Git config:

```
$ git config --global core.editor “vim"
```

Set the **GIT_EDITOR** environment variable:

```
export GIT_EDITOR=vim
```

If you want to set the editor for Git and also other programs, set the standardized VISUAL and EDITOR environment variables*:

```
export VISUAL=vim
export EDITOR="$VISUAL“
```

reset back to default editor.

```
$ git config --global --unset core.editor
```

GIT EDITOR

VSCODE

Set **core.editor** in your Git config:

```
$ git config --global core.editor “code --wait"
```

Set the **GIT_EDITOR** environment variable:

```
export GIT_EDITOR ="code --wait"
```

If you want to set the editor for Git and also other programs, set the standardized VISUAL and EDITOR environment variables*:

```
export VISUAL ="code --wait"
export EDITOR="$VISUAL"
```

GIT EDITOR

SUBLIME

Set **core.editor** in your Git config:

$ **git config --global core.editor "'C:/Program Files (x86)/sublime text 3/subl.exe' -w“**

Set the **GIT_EDITOR** environment variable:

export GIT_EDITOR = "'C:/Program Files (x86)/sublime text 3/subl.exe' -w"

If you want to set the editor for Git and also other programs, set the standardized VISUAL and EDITOR environment variables*:

export VISUAL = "'C:/Program Files (x86)/sublime text 3/subl.exe' -w"

export EDITOR="$VISUAL"

GIT EDITOR

ATOM

Set **core.editor** in your Git config:

```
$ git config --global core.editor “atom --wait"
```

Set the **GIT_EDITOR** environment variable:

```
export GIT_EDITOR =“atom --wait"
```

If you want to set the editor for Git and also other programs, set the standardized VISUAL and EDITOR environment variables*:

```
export VISUAL =“atom --wait"
export EDITOR="$VISUAL"
```

GIT EDITOR[1]

NOTEPAD++

Set **core.editor** in your Git config:

```
$ git config --global core.editor "'C:/Program Files(x86)/
        Notepad++/notepad++.exe' -multiInst -notabbar
        -nosession -noPlugin"
```

Set the **GIT_EDITOR** environment variable:

```
export GIT_EDITOR ="'C:/Program Files(x86)/
Notepad++/notepad++.exe' -multiInst -notabbar
-nosession -noPlugin"
```

GIT EDITOR[2]

NOTEPAD++

If you want to set the editor for Git and also other programs, set the standardized VISUAL and EDITOR environment variables*:

```
export VISUAL ="'C:/Program Files(x86)/
        Notepad++/notepad++.exe' -multiInst -notabbar
        -nosession -noPlugin"
export EDITOR="$VISUAL“
```

GIT
LINE ENDING

COMMAND

GIT

LINE ENDING

The git config **core.autocrlf** command is used to change how Git handles line endings. It takes a single argument.

On Windows, you simply pass **true** to the configuration.

```
$ git config --global core.autocrlf true
```

[note]: configure git to ensure line endings in files you checkout are correct for windows. for compatibility, line endings are converted to UNIX style when you commit files.

GIT VISUAL TOOLS

COMMAND

GIT

VISUAL TOOLS

There are lot of Visual Diff Merge Tool:

- ✓ **WinMerge**
- ✓ **P4Merge**
- ✓ **KDiff3**
- ✓ **Meld**
- ✓ **DiffMerge**

[note]:

- Installed One Of Them
- Integrate With Git

GIT
DIFF TOOLS

COMMAND

GIT

DIFFTOOL

- ❖ Check The Difference Between Working Directory Vs Staging Area

```
$ git difftool
```

- ❖ Check The Difference Between Staging Area Vs Local Repository

```
$ git difftool --staged
```

- ❖ Check The Difference Between Workspace Vs Local Repository

```
$ git difftool HEAD
```

- ❖ Check The Difference Between Two Commit

```
$ git difftool c91e1b1 953a210
```

GIT

DIFFTOOL

- ❖ Check The Difference Between Working Directory Vs Staging Area

  ```
  $ git difftool -t P4Merge
  ```

- ❖ Check The Difference Between Staging Area Vs Local Repository

  ```
  $ git difftool --staged -t KDiff3
  ```

- ❖ Check The Difference Between Workspace Vs Local Repository

  ```
  $ git difftool HEAD -t WinMerge
  ```

- ❖ Check The Difference Between Two Commit

  ```
  $ git difftool c91e1b1 953a210 -t vimdiff
  ```

GIT
DIFF TOOLS
(VSCODE)

COMMAND

GIT

DIFF TOOLS using VSCODE

- ❖Required: VSCODE Installed!
- ❖Setting Visual Diff Tool on VSCODE

```
$ git config --global diff.tool vscode
$ git config --global difftool.vscode.cmd “code --wait
    --diff $LOCAL $REOMTE”
```

- ❖Checking The Config That Actually Save The Info or Not?

```
$ git config --global -e
```

GIT
MERGE TOOLS

COMMAND

GIT

MERGE TOOL

❖Opening With last Config Tool

```
$ git mergetool
```

❖ Open With Specific Merge While 2 or more Tool are Installed

```
$ git mergetool -t vimdiff
```

GIT MERGE TOOLS (VSCODE)

COMMAND

GIT

MERGE TOOLS

- ❖ Required: VSCODE Installed!
- ❖ Setting Visual Merge Tool on VSCODE

```
$ git config --global merge.tool vscode
$ git config --global mergetool.vscode.cmd “code
        --wait --merge  $LOCAL  $REOMTE”
```

- ❖ Checking The Config That Actually Save The Info or Not?

```
$ git config --global -e
```

GIT

HELP

COMMAND

GIT HELP

More Details For Any Command

```
$ git help <command>
$ git <command> --help
$ man git-<command>
```

Short Details For Any Command

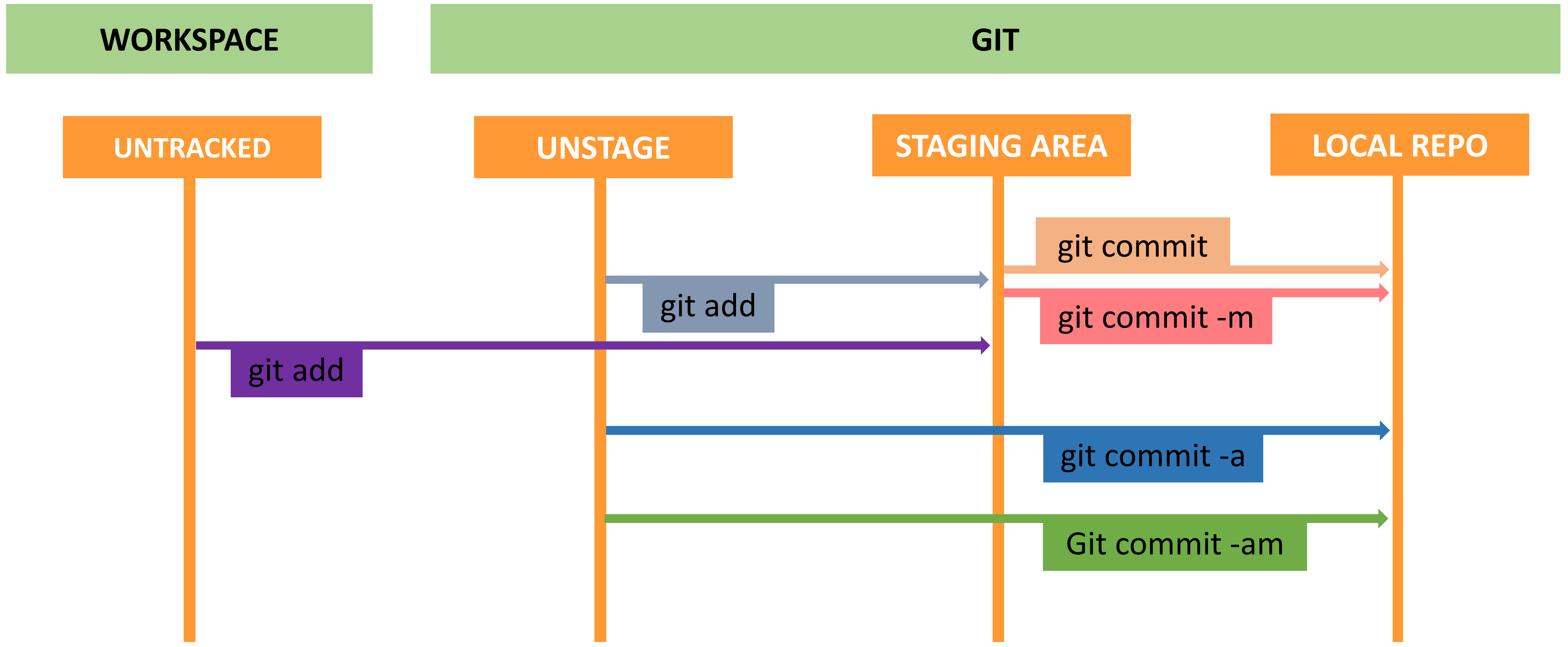
```
$ git <command> -h
```

GIT

ADD

COMMAND

GIT WORKFLOW



GIT ADD[1]

WORKSACE TO STAGING AREA

- ❖ add all untracked, modified, deleted folder, sub folder & it's contain files & also level up folder and files.

```
$ git add --all
$ git add -A
```

- ❖ add all modified, deleted folder, sub folder & it's contain files.

```
$ git add -u
$ git add --update
```

GIT ADD[2]

WORKSACE TO STAGING AREA

- ❖ add specific modified, deleted folder, sub folder & it’s contain files.

```
$ git add -u my-dir/
$ git add --update *.txt
```

- ❖ add current folder & it’s contain files but not level up folder.

```
$ git add .
```

- ❖ add all files & folders to stage without current folder deleted & unstaged files but not upper level or down level.

```
$ git add *
```

GIT ADD[3]

WORKSACE TO STAGING AREA

- ❖ add a single or multiple specific file

```
$ git add my_folder/three.txt
$ git add two.txt four.txt
```

- ❖ add specific extensions files

```
$ git add *.txt
```

- ❖ add a multiple specific files & specific extensions files

```
$ git add *.txt four.pdf
```

GIT ADD PATCH

MODIFIED TO STAGING AREA

- add specific portion from all modified file one by one.

```
$ git add -p
$ git add --patch
```

- add specific portion from a specific file

```
$ git add -p x.txt
$ git add --patch x.txt
```

[note]: after entering the mode ? for help

GIT

LS-FILES

COMMAND

GIT LS-FILES[1]

- ❖ show stage & committed files list in the output

```
$ git ls-files
$ git ls-files -c
$ git ls-files --cached
```

- ❖ Show staged & committed contents “mode bits, object name and stage number” in the output.

```
$ git ls-files -s
$ git ls-files --stage
```

GIT LS-FILES[2]

- ❖ show manual deleted(rm x.txt) [that is an unstage] files in the output.

```
$ git ls-files -d
$ git ls-files --deleted
```

- ❖ show modified files in the output

```
$ git ls-files -m
$ git ls-files --modified
```

- ❖ show other (i.e. untracked) files in the output

```
$ git ls-files -o
$ git ls-files --others
```

GIT LS-FILES[3]

- show only ignored files (that is forcedly added **git add -f**) in the output.

```
$ git ls-files -i --exclude-standard
$ git ls-files --ignored --exclude-standard
```

- show only untracked, ignored files in the output.

```
$ git ls-files -o -i --exclude-standard
$ git ls-files --other --ignored --exclude-standard
```

- show only untracked, not ignored files in the output.

```
$ git ls-files -o --exclude-standard
$ git ls-files --other --exclude-standard
```

GIT

STATUS

COMMAND

GIT STATUS

- ❖ see the full status in all state

```
$ git status
```

- ❖ see the status in all state in short form

```
$ git status -s
$ git status --short
```

GIT COMMIT

COMMAND

GIT COMMIT[1]

STAGING AREA TO LOCAL REPOSITORY

- ❖ commit a git

```
$ git commit
```

- ❖ enter i & type “commit message”
- ❖ enter Esc
- ❖ enter :x

GIT COMMIT[2]

STAGING AREA TO LOCAL REPOSITORY

❖ commit a git with message

```
$ git commit -m “initial commit”
```

❖ fixing last commit message

if any mistake in last commit message, run this command:

```
$ git commit --amend -m "corrected message"
```

GIT COMMIT (DIRECTLY)

MODIFIED TO LOCAL REPOSITORY

- ❖ directly commit unstaged changes

```
$ git commit -a
```

  - enter **i** & type “commit message”
  - enter **Esc**
  - enter **:x**

- ❖ directly commit unstaged changes with message

```
$ git commit -am “initial commit”
```

GIT

AMEND

COMMAND

GIT AMEND

- current commit integrate with the last commit

  $ **git commit --amend --no-edit**

  [note]: it will produced new hash

GIT

LOG

COMMAND

GIT LOG[1]

- ❖ check all the commit with details (full commit hash)

  $ **git log**

  [note] : (1) **space** for expanding(2) **q** for exit
- ❖ check all the commit with details (short commit hash)

  $ **git log --abbrev-commit**
- ❖ check all the commit in short

  $ **git log --oneline**
- ❖ check all the commit in one line reverse order

  $ **git log --oneline --reverse**

GIT LOG[2]

- ❖ check the last 3 commit with details

  $ **git log -n 3**
- ❖ check the last 3 commit in short

  $ **git log --oneline -n 3**
- ❖ check the first 3 commit with details

  $ **git log --reverse -n 3**
- ❖ check the first 3 commit in short

  $ **git log --oneline --reverse -n 3**

GIT
SHOW

COMMAND

GIT SHOW[1]

- required $ **git log --oneline**
- check what change in the last commit

  $ **git show head**
- check what change in the commit

  $ **git show 880408e**
- check what change in the 4th commit

  $ **git show head~4**

GIT SHOW[2]

- ❖ required $ **git log --oneline**
- ❖ check specific file content that add in the commit

  $ **git show head~4: .gitignore** or

  $ **git show 880408e: .gitignore** or

  $ **git show 880408e: bin/app.bin**

  [note] : deleted file gives an error

GIT

CHECKOUT

COMMAND

GIT CHECKOUT

- ❖ required $ **git log --oneline**
- ❖ check the commit

  $ **git checkout 880408e**

  [note] : make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.
- ❖ create a new branch to retain commits

  $ **git checkout -b stable** or

  $ **git switch -c unstable**
- ❖ switched to master branch

  $ **git switch –**

GIT
LS-TREE
COMMAND

GIT LS-TREE

❖ Check All The LOCAL REPOSITORY Files

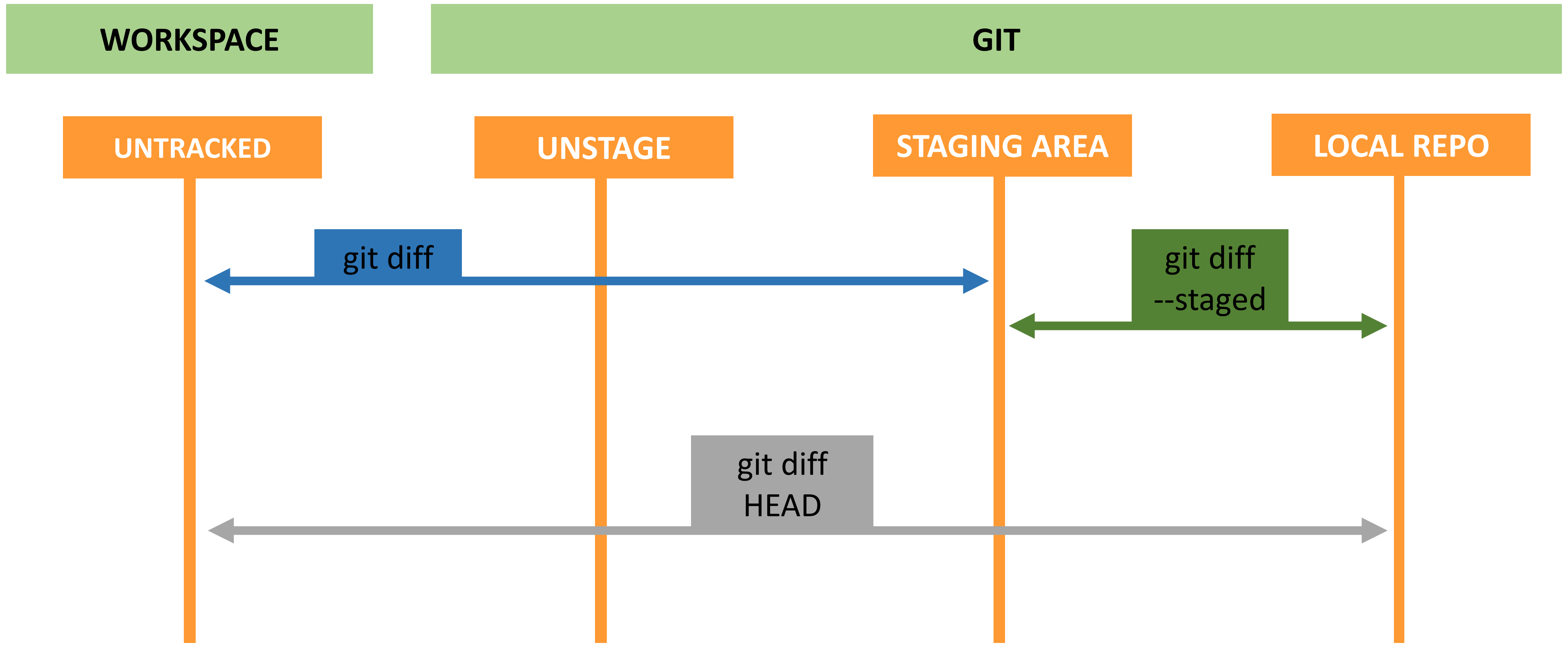
```
$ git ls-tree
```

GIT

DIFF

COMMAND

GIT WORKFLOW



GIT DIFF

- ❖ difference between working directory vs staging area

  $ **git diff**

- ❖ difference between staging area vs local repository

  $ **git diff --staged**

- ❖ difference between workspace vs local repository

  $ **git diff head**

- ❖ difference between two commit

  $ **git diff c91e1b1 953a210**

  [note]: q for exit.

GIT
VISUAL TOOLS

COMMAND

GIT

VISUAL TOOLS

there are lot of visual diff & merge tool:

- ✓ **WinMerge**
- ✓ **P4Merge**
- ✓ **KDiff3**
- ✓ **Meld**
- ✓ **DiffMerge**
- ✓ **vimdiff (default)**

[note]:

- installed one of them
- integrate with git

GIT
DIFF TOOLS
(VSCODE)

COMMAND

GIT[1]

DIFF TOOLS using VSCODE

- required: vscode installed!
- setting visual diff tool on vscode

```
$ git config --global diff.tool vscode
$ git config --global difftool.vscode.cmd “code --wait
        --diff $LOCAL $REOMTE”
```

- checking the config that actually save the info or not?

```
$ git config --global -e
```

GIT[2]

DIFF TOOLS using VSCODE

- ❖ required: vscode installed!
- ❖ open the config file

```
$ git config --global -e
```

- ❖ add the line below it.

```
[diff]
        tool = vscode
[difftool "vscode"]
        cmd = code --wait --diff $LOCAL $REMOTE
```

GIT
DIFF TOOLS

COMMAND

GIT[1]

DIFFTOOL

- check the difference between working directory vs staging area

  $ **git difftool**
- check the difference between staging area vs local repository

  $ **git difftool --staged**
- check the difference between workspace vs local repository

  $ **git difftool head**
- check the difference between two commit

  $ **git difftool c91e1b1 953a210**

GIT[2]

DIFFTOOL

- check the difference between working directory vs staging area

```
$ git difftool -t vscode
```

- check the difference between staging area vs local repository

```
$ git difftool --staged -t kdiff3
```

- check the difference between workspace vs local repository

```
$ git difftool head -t winmerge
```

- check the difference between two commit

```
$ git difftool c91e1b1 953a210 -t vimdiff
```

GIT

RM

COMMAND

GIT RM

REMOVE UNTRACKED FILES

- remove the files which is not tracked although

  ```
  $ rm new.txt
  ```

  [note]: no more command need cause it a untracked file.

GIT RM[1]

REMOVE STAGING AREA FILES

- ❖ remove the file from staging area

```
$ git rm -f new.txt
```

- ❖ remove the file from staging area but not touch in workspace file.

```
$ git rm --cached new.txt
```

[note]: no need to commit.

GIT RM[2]

REMOVE STAGING AREA FILES

- ❖ remove the file from staging area

```
$ rm new.txt
```

- ❖ deleted info updated to the staging area.

```
$ git add new.txt
```

[note]: no need to commit.

GIT RM

REMOVE COMMITTED FILES

- remove the file after commit.

  $ **git rm new.txt**

  [note]: deleted info already in staging area
- commit the info

  $ **git commit -m “new.txt removed”**

GIT RM

REMOVE MODIFIED FILES at UNSTAGE

- remove the file

```
$ rm new.txt
```

- update info to staging area

```
$ git add new.txt
```

- commit the info

```
$ git commit -m “new.txt removed”
```

GIT RM

REMOVE MODIFIED FILES at UNSTAGE

- ❖ remove the file

  $ **git rm --cached new.txt** or

  $ **git rm -f new.txt**

  [note]: deleted info already in staging area
- ❖ commit the info

  $ **git commit -m “new.txt removed”**

GIT RM[1]

REMOVE MODIFIED FILES at STAGING AREA

- ❖ remove the file

  $ **git rm --cached new.txt** or

  $ **git rm -f new.txt**

  [note]: deleted info already in staging area

- ❖ commit the info

  $ **git commit -m “new.txt removed”**

GIT RM[2]

REMOVE MODIFIED FILES at STAGING AREA

- ❖ modified file add to the staging area, then

  $ **rm new.txt**
- ❖ add to staging area

  $ **git add new.txt**

  [note]: before modified file will no longer available in staging area.
- ❖ commit the info

  $ **git commit -m “new.txt removed”**

GIT RENAME

COMMAND

GIT RENAME[1]

- ❖in the [directory]
  - ➢ to rename a file $ **mv one.txt ones.txt**
- ❖ stage the renamed file

  $ **git add one.txt** (as a deleted file)

  $ **git add ones.txt** (as a unstaged file)
- ❖ commit the deleted info

  $ **git commit -m “one.txt rename to ones.txt”**

GIT RENAME[2]

- ❖ stage the renamed file

```
$ git mv one.txt ones.txt
```

- ❖ commit the renamed info

```
$ git commit –m “one.txt rename to ones.txt”
```

GIT MOVE

COMMAND

GIT MOVE[1]

- ❖in the [directory]
  - ➢ to move a file $ **mv one.txt folder/one.txt**
- ❖ stage the moved file

  $ **git add one.txt** (as a deleted file)

  $ **git add folder/one.txt** (as a unstaged file)
- ❖ commit the moved info

  $ **git commit -m “one.txt move to folder/one.txt”**

GIT MOVE[2]

- stage the renamed file

```
$ git mv one.txt folder/one.txt
```

- commit the renamed info

```
$ git commit –m “one.txt move to folder/one.txt”
```

GIT IGNORE

COMMAND

GIT IGNORE

DIRECTORY MANAGEMENT

- ❖ add text in the file with replace the existing text

```
$ echo logs/ > .gitignore
```

- ❖ append text in the file

```
$ echo logs/ >> .gitignore
```

GIT IGNORE

- ❖ make the files or folder that would be ignore

```
$ mkdir logs
$ echo “logs activity” > logs/dev.log
```

- ❖ make the file .gitignore

```
$ touch .gitignore
```

- ❖ ignoring the files & folder

```
$ echo logs/ >> .gitignore
$ git add .gitignore
$ git commit –m “added .gitignore”
```

GIT IGNORE[1]

RULE

the rules for the patterns you can put in the .gitignore file are as follows:

- ✓ blank lines or lines starting with # are ignored.
- ✓ start patterns with a forward slash (/) to avoid recursivity.
  i.e. **/todo** only ignore the todo file in the current directory, not subdir/todo.

GIT IGNORE[2]

RULE

the rules for the patterns you can put in the .gitignore file are as follows:

- ✓ end patterns with a forward slash (/) to specify a directory.
  i.e. **build/** ignore all files in any directory named build
- ✓ negate a pattern by starting it with an exclamation point (!).
  i.e. **!lib.a** even though you're ignoring .a files above

GIT IGNORE[3]

RULE

- ✓ standard glob patterns work, and will be applied recursively throughout the entire working tree.
  - an asterisk (*) matches zero or more characters;
    i.e. ***.txt** ignore all **.txt** files.
  - [abc] matches any character inside the brackets (in this case a, b, or c);
    i.e. **.[oa]** ignoring all **.a, .o** files
  - a question mark (?) matches a single character;

GIT IGNORE [4]

RULE

✓ standard glob patterns work, and will be applied recursively throughout the entire working tree.

- brackets enclosing characters separated by a hyphen ([0- 9]) matches any character between them (in this case 0 through 9).

  i.e. **[0-9].txt** ignoring all **0.txt, 1.txt** & so on.

- use two asterisks to match nested directories;

  i.e. **a/**/z** would match **a/z, a/b/z, a/b/c/z**, and so on.

GIT IGNORE[1]

[note]: if accidentally committed file can not be ignored from git

❖ make file in a directory

```
$ mkdir bin
$ echo “bin-tin” >> bin/app.bin
```

❖ make it staging area & commit

```
$ git commit -am “added bin”
```

❖ adding to .gitignore

```
$ echo bin/ >> .gitignore
```

GIT IGNORE[2]

- ❖ git added & change commit

```
$ git commit -am “.gitignore updated”
```

- ❖ now edit the app.bin file

```
$ echo “bin-tin 2” >> bin/app.bin
```

- ❖ checking the git status & its tracking*

```
$ git status
```

GIT IGNORE[3]

but can be fixed by these way:

❖ removing the file from staging area

```
$ git rm --cached –r bin/
```

❖ git added & change commit

```
$ git commit -m “bin folder removed from staging area”
```

now fixed no tracking

GIT EXCLUDE

- force add a **.txt** file that include in **.gitignore** pattern

```
$ git add -f x.txt
```

GIT STASH

COMMAND

GIT STASH[1]

Introduction:
git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

GIT STASH[2]

❖ stashing current state

    $ **git stash**

❖ stashing current state with message

    $ **git stash save “message”**

❖ check the stashing list

    $ **git stash list**

❖ rollback the last stash

    $ **git stash apply**

❖ rollback the specific stash

    $ **git stash apply stash@{1}** or
    $ **git stash apply 1**

GIT STASH[3]

- ❖ show the content of the stash
  - $ **git stash show -p 1**
- ❖ git stash specific file/files
  - $ **git stash one.txt two.txt**
  - $ **git stash push one.txt two.txt**
- ❖ git stash specific file/files with message
  - $ **git stash save “message” one.txt two.txt** or
  - $ **git stash save “message” one.txt two.txt** or
  - $ **git stash push -m “message” one.txt two.txt***

GIT STASH[4]

- ❖ remove the last stash

  $ **git stash drop**
- ❖ remove specific stash

  $ **git stash drop stash@{1}** or

  $ **git stash drop 1**
- ❖ rollback a stash & remove it

  $ **git stash pop**

GIT STASH[4]

- rollback specific stash & remove it

  $ **git stash pop stash@{1}** or

  $ **git stash pop 1**

  [note] : it can be make a stash conflict
- clear all stash

  $ **git stash clear**

GIT STASH CONFLICT

- ❖ Edit The Stashing File
- ❖ Stash The File

```
$ git stash
```

- ❖ Add To STAGING AREA

```
$ git add .
```

- ❖ Now Stash/Drop/Pop It

```
$ git stash
$ git stash drop
$ git stash pop
```

GIT CLEAN

COMMAND

GIT CLEAN[1]

- ❖ remove all the untracked files/folders

```
$ git clean
```

  [note] : git clean remove untracked files & folders. git clean does not work without flag i.e. -f & -i .

- ❖ remove all the untracked files

```
$ git clean -f
```

- ❖ see what untracked files will be removed

```
$ git clean -f -n
```

GIT CLEAN[2]

- ❖ remove all the untracked file & folders

```
$ git clean -f -d
```

- ❖ see what untracked files/folders will be removed

```
$ git clean -f -d -n
```

- ❖ show different option all the untracked files i.e. clean, quit etc.

```
$ git clean -i                    $ git clean -i -n
```

- ❖ show different option all the untracked file & folders

```
$ git clean -d –i   $ git clean -d -i -n
```

GIT BRANCHES

COMMAND

GIT BRANCH[1]

- ❖ list all branches of local repository

```
$ git branch
```

- ❖ see all remote branches of repository

```
$ git branch -r
```

- ❖ create new branch on current revision

```
$ git branch dev
```

- ❖ create branch at specific revision

```
$ git branch other 618dd0a
```

GIT BRANCH[2]

- ❖ rename a branch while pointed to any branch

```
$ git branch -m dev dev
```

- ❖ rename current branch

```
$ git branch -m main
```

- ❖ delete branch

```
$ git branch -d dev
```

[note]: cannot delete current branch/ head branch.

GIT REMOTE BRANCH[3]

- ❖ first delete current / old branch

  $ **git push origin --delete dev**

- ❖ push new local branch

  $ **git push -u origin dev**

- ❖ first delete current / old branch

  $ **git push origin --delete dev**

GIT CHECKOUT

- switch between existing branches

  $ **git checkout dev**
- create and checkout new branch

  $ **git checkout -b dev**
- create and checkout new branch at a specific revision

  $ **git checkout -b dev 618dd0a**

  [note] : only works with no uncommitted changes („clean working tree“)

GIT SWITCH[1]

- switch between existing branches

```
$ git  switch dev
```

- create and checkout new branch

```
$ git switch -c dev
```

- create and checkout new branch at a specific revision

```
$ git switch -c dev 618dd0a
```

- switch to an specific revision

```
$ git  switch --detach 618dd0a or
$ git  switch -d 618dd0a
```

GIT SWITCH[2]

- create a new orphan branch

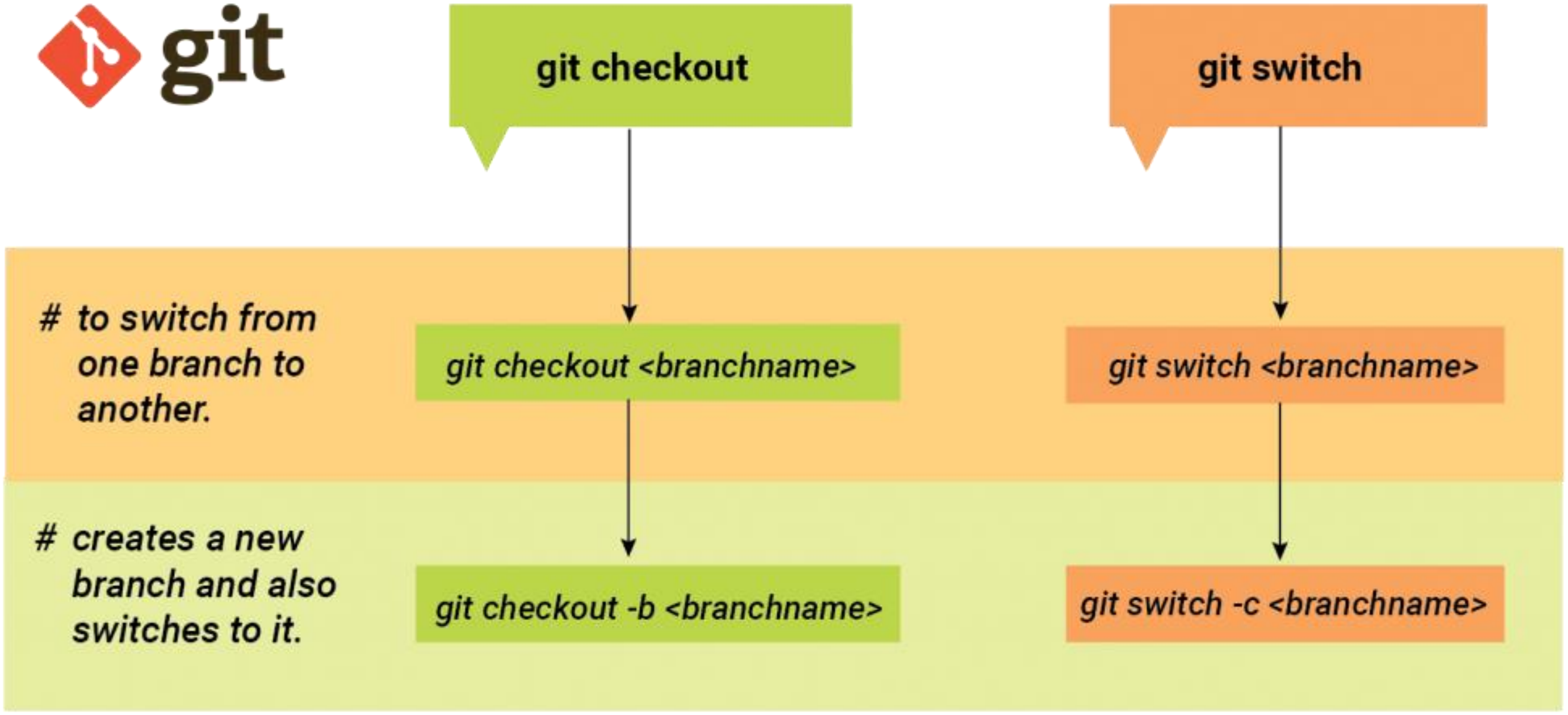
  $ **git switch --orphan feature**

  [note]: all tracked files are removed.
- tracking branch from remote

  $ **git switch -c dev --track origin/dev**

  [note]: if branch is not found but there does exist a tracking branch in exactly one remote with a matching name. execute a **git fetch** command before, to fetch the latest remote updates

GIT SWITCH VS CHECKOUT


git
git checkout
git switch
# to switch from one branch to another.
git checkout <branchname>
git switch <branchname>
# creates a new branch and also switches to it.
git checkout -b <branchname>
git switch -c <branchname>

GIT BRANCHES

COMPARE

COMMAND

GIT BRANCH COMPARE
GIT LOG

- ❖ checking commit differences between two branch in **dev** but in **main**

```
$ git  log main..dev
```

- ❖ checking commit differences between in **main** but in **remote**

```
$ git  log origin/main..main
```

GIT BRANCH COMPARE

GIT DIFF

- ❖ show you all the commits that **dev** has that are not in **main** with two dot

  $ **git diff main..dev**

- ❖ show you all the commits that **dev** has that are not in **main** with three dot

  $ **git diff main…dev**

  [note] : compares the top of the right branch (the head) with the common ancestor of the two branches.

GIT BRANCH COMPARE
GIT DIFF

- ❖ see the differences done to a specific file between two branches

```
$ git diff main..dev -- one.txt
```

GIT BRANCHES

MERGE

COMMAND

GIT MERGE (FAST FORWARD)

- ❖ merge changes in checked out branch

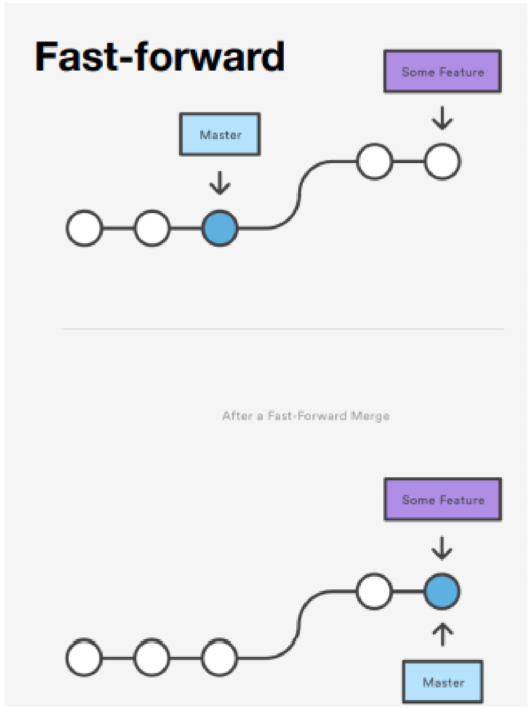
  $ **git merge dev**

  (auto-generated merge commit message)

- ❖ alternative way**:**

  $ **git merge dev –m “merging”**

[note] : fast forward merge happen when nothing change happen in master branch**.** but another branch is forward.



GIT MERGE (THREE-WAY)

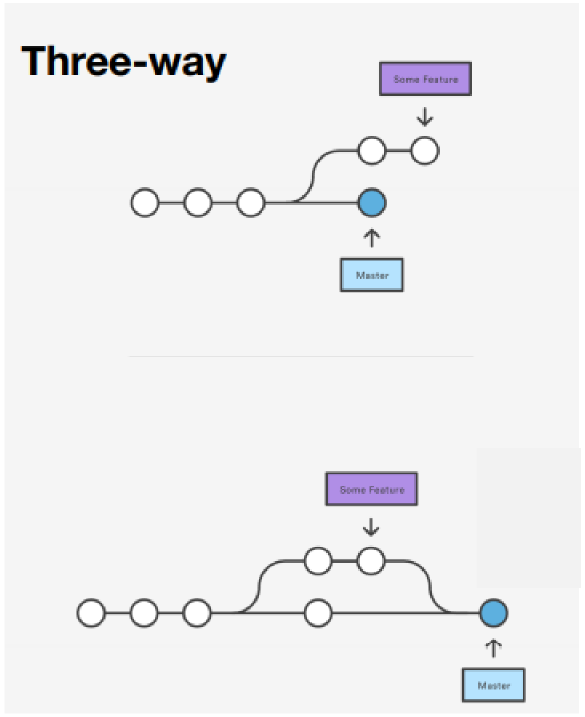
- ❖ merge changes in checked out branch

```
$ git  merge dev
```

(auto-generated merge commit message)

- ❖ alternative way:

```
$ git merge dev –m “merging”
```



GIT MERGE (CONFLICT)

[note] : only three-way merges have merge commits and potential merge conflicts. conflicts if same part of file is changed in both branches.

- ❖ to see „unmerged paths“

```
$ git status
```

- ❖ find problematic hunks, create the intended code version and remove <<<<<<< , =======, >>>>>>>

```
$ git add new.txt
$ git commit -m “merging”
```

GIT
VISUAL TOOLS

COMMAND

GIT

VISUAL TOOLS

There are lot of Visual Diff & Merge Tool:

- ✓ **WinMerge**
- ✓ **P4Merge**
- ✓ **KDiff3**
- ✓ **Meld**
- ✓ **DiffMerge**

[note]:

- Installed One Of Them
- Integrate With Git

GIT
MERGE TOOLS

COMMAND

GIT

MERGE TOOL

- ❖Opening With last Config Tool

```
$ git mergetool
```

- ❖ Open With Specific Merge While 2 or more Tool are Installed

```
$ git mergetool -t vimdiff
```

GIT
MERGE TOOLS
(VSCODE)

COMMAND

GIT

MERGE TOOLS

- ❖ Required: VSCODE Installed!
- ❖ Setting Visual Merge Tool on VSCODE

```
$ git config --global merge.tool vscode
$ git config --global mergetool.vscode.cmd “code
        --wait --merge  $LOCAL  $REOMTE”
```

- ❖ Checking The Config That Actually Save The Info or Not?

```
$ git config --global -e
```

GIT BRANCHES

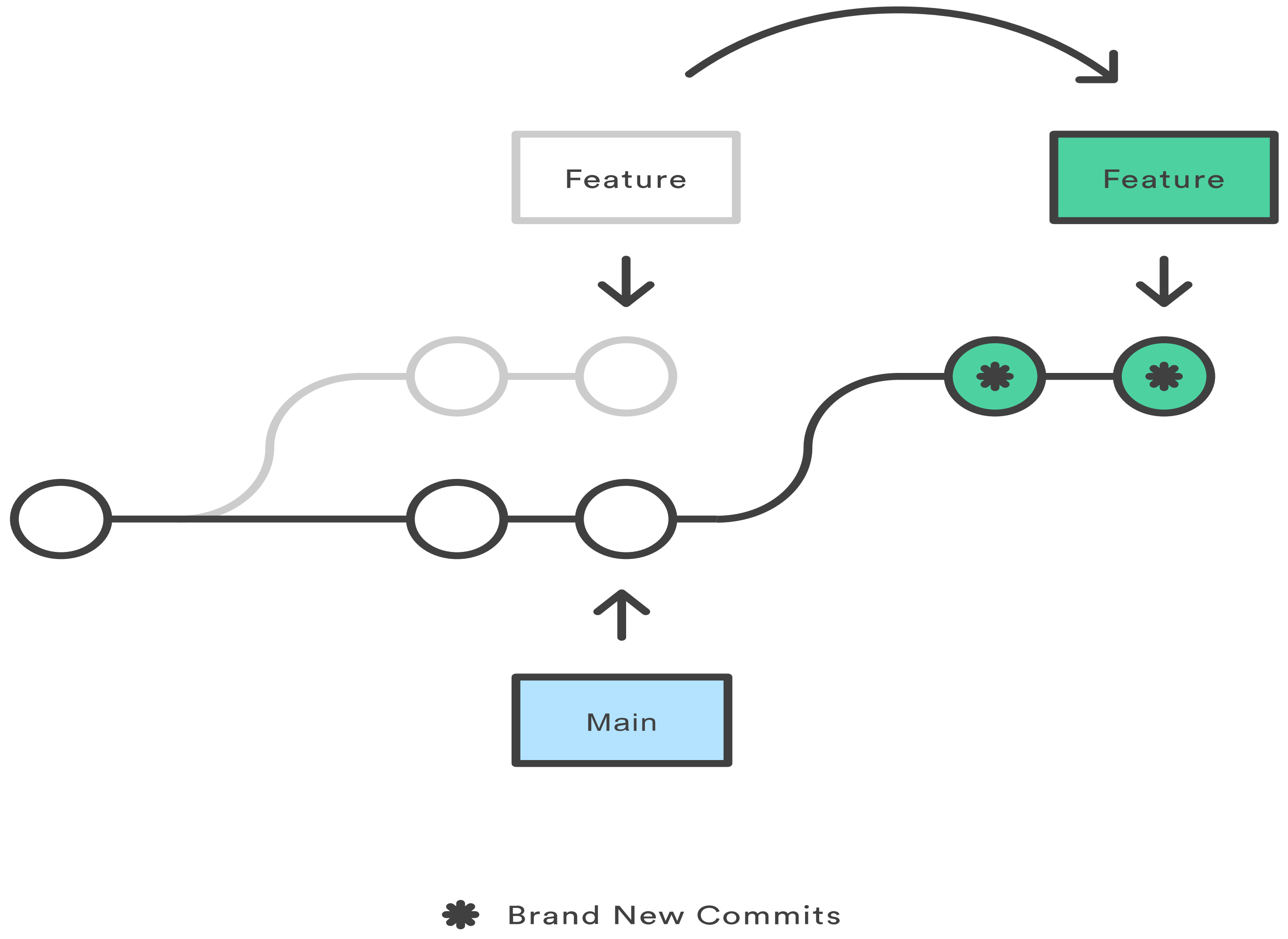
REBASE

COMMAND

GIT REBASE

Intro:
Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow.

GIT REBASE



GIT REBASE

- rebase two branch

  $ **git rebase master**

  [note]: master branch will be base of current branch. hashes will change of current branch logs.
- rebase two branch

  $ **git rebase feature**

  [note]: feature branch will be base of current branch. hashes will change of current branch logs.

GIT REBASE CONFLICT

- ❖ solve conflict.
- ❖ if added any files

```
$ git add .
```

- ❖ continue rebasing

```
$ git rebase --continue
```

- ❖ commit can be skipped

```
$ git rebase --skip
```

GIT
INTERACTIVE REBASE

COMMAND

GIT REWORD

COMMAND

GIT REWORD

- ❖ show log

  $ **git log --oneline**
- ❖ rewording a commit

  $ **git rebase -i HEAD~4**

  $ **git rebase -i fbc20aa**
- ❖ chose the line/lines what would be rewording &

  replace **pick** with **reword**
- ❖ update the commit & done

GIT REORDER

COMMAND

GIT REORDER

- ❖show log

  $ **git log --oneline**
- ❖ rewording a commit

  $ **git rebase -i HEAD~4** or

  $ **git rebase -i fbc20aa**
- ❖ chose the line/lines what would be reordering & switch the line.
- ❖ done.

GIT DROP

COMMAND

GIT DROP

- ❖show log
  - $ **git log --oneline**
- ❖ rewording a commit
  - $ **git rebase -i HEAD~4** or
  - $ **git rebase -i fbc20aa**
- ❖ chose the commit line/lines what would be remove & **replace** pick with **drop**.
- ❖ done.

GIT FIXUP

COMMAND

GIT FIXUP

❖show log

$ **git log --oneline**

❖ fixup commit

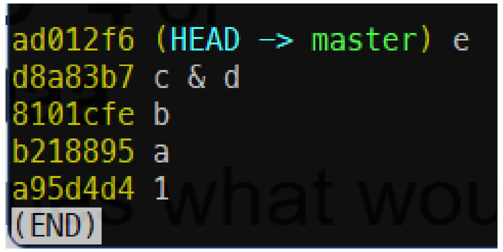
$ **git rebase -i HEAD~4** or

$ **git rebase -i a95d4d4**

❖ replace **pick** with **fixup**.

- enter **i** & edit
- enter **Esc :x**

[note1]: at least one previous commit remain in **pick** mode, otherwise rebase with face problem.
[note2]: there is no commit option.



```
ad012f6 (HEAD -> master) e
d8a83b7 c & d
8101cfe b
b218895 a
a95d4d4 1
(END)
```

GIT FIXUP

[note1]: at least one previous commit remain in
pick mode, otherwise rebase with face problem.
[note2]: there is no commit option.

GIT SQUASH

COMMAND

GIT SQUASH

Intro:
Decide what files go into which **commits** right before you commit with the **staging area**, you can decide that you didn’t mean to be working on something yet with **git stash**, and you can rewrite commits that already happened so they look like they happened in a different way**.**
[note] : combining multiple commits into one & hashing change.

GIT SQUASH

- ❖ show log

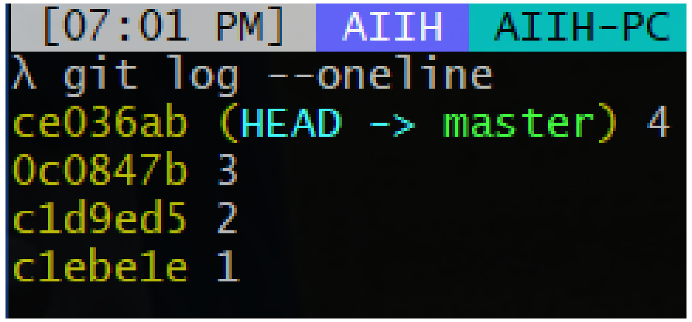
  $ **git log --oneline**

- ❖ opening rebase interactive mode

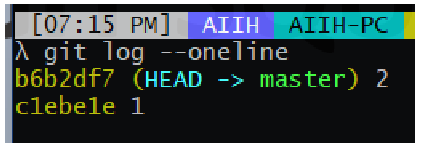
  $ **git rebase -i c1ebe1e** or

  $ **git rebase -i head~3**

- ❖ replace **pick** with **squash**
  - enter **i** & edit
  - enter **Esc :x**



```
[07:01 PM] AIIH AIIH-PC
λ git log --oneline
ce036ab (HEAD -> master) 4
0c0847b 3
c1d9ed5 2
c1ebe1e 1
```



```
[07:15 PM] AIIH AIIH-PC
λ git log --oneline
b6b2df7 (HEAD -> master) 2
c1ebe1e 1
```

GIT SQUASH

- ❖ Add New commit message
    - enter **i** & message
    - enter **Esc :x**

[note1]: at least one previous commit remain in **pick** mode, otherwise rebase with face problem.

GIT SPLIT

COMMAND

GIT SPLIT[1]

- ❖ show log

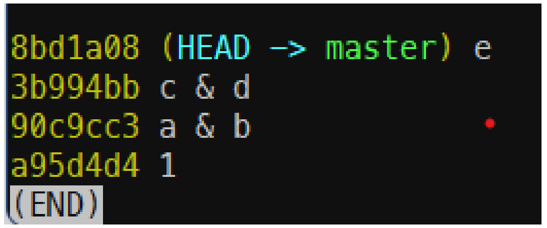
  $ **git log --oneline**

- ❖ changing multiple commit messages

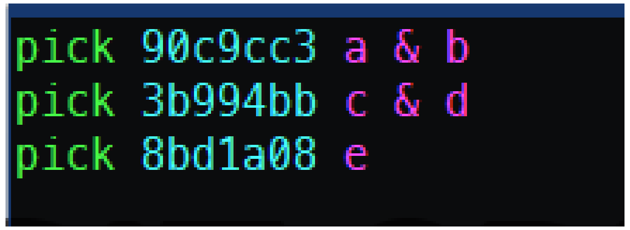
  $ **git rebase -i a95d4d4** or

  $ **git rebase -i head~3**

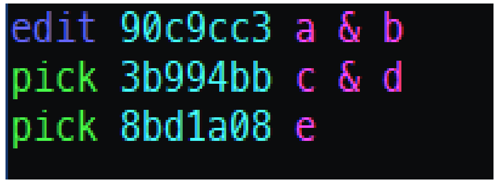
- ❖ replace **pick** with **edit**
  - enter **i** & edit
  - enter **Esc :x**



```
8bd1a08 (HEAD -> master) e
3b994bb c & d
90c9cc3 a & b
a95d4d4 1
(END)
```



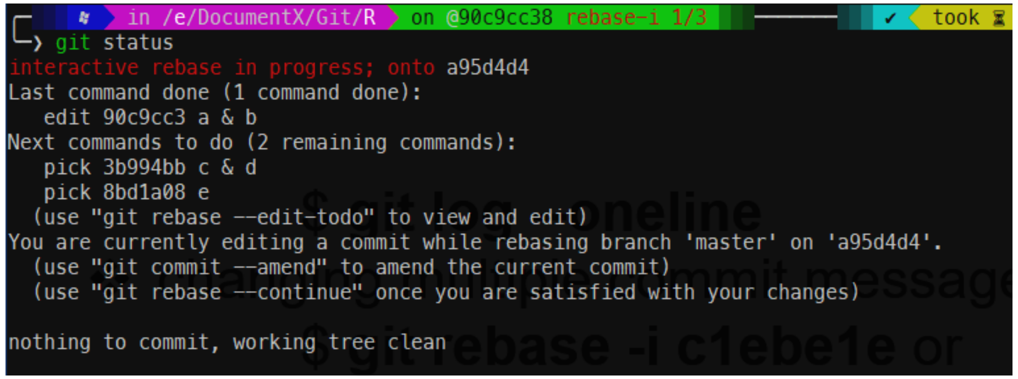
```
pick 90c9cc3 a & b
pick 3b994bb c & d
pick 8bd1a08 e
```



```
edit 90c9cc3 a & b
pick 3b994bb c & d
pick 8bd1a08 e
```

GIT SPLIT

- ❖ Check status
  $ **git status**
- ❖ Undo the commit
  $ **git reset HEAD~**
- ❖ Add file to stage
  $ **git add a.txt**
- ❖ Commit the file
  $ **git commit -m “a”**



```
in /e/DocumentX/Git/R   on @90c9cc38 rebase-i 1/3   ✓ took
╰─❯ git status
interactive rebase in progress; onto a95d4d4
Last command done (1 command done):
   edit 90c9cc3 a & b
Next commands to do (2 remaining commands):
   pick 3b994bb c & d
   pick 8bd1a08 e
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'master' on 'a95d4d4'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working tree clean
```

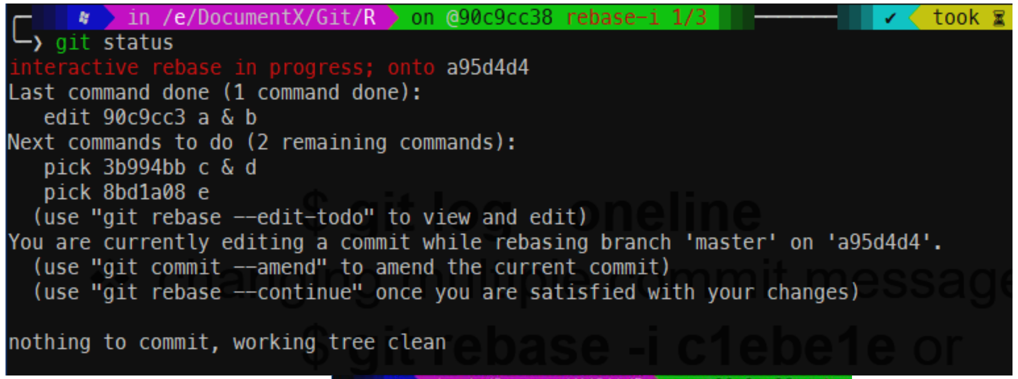
GIT SPLIT

- Add file to stage

  $ **git add b.txt**
- Commit the file

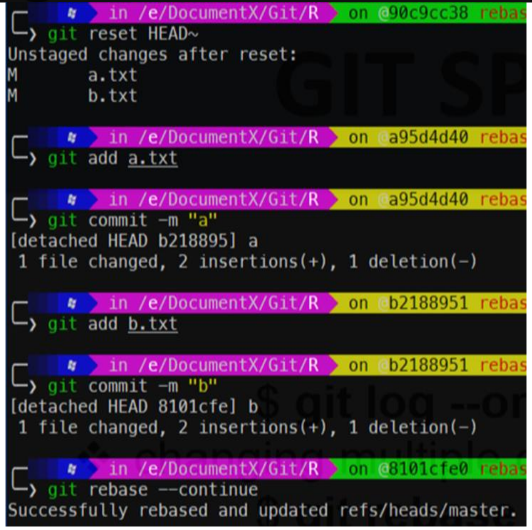
  $ **git commit -m “b”**
- Complete rebasing

  $ **git rebase --continue**



```
in /e/DocumentX/Git/R  on @90c9cc38 rebase-i 1/3      ✓  took
└─❯ git status
interactive rebase in progress; onto a95d4d4
Last command done (1 command done):
   edit 90c9cc3 a & b
Next commands to do (2 remaining commands):
   pick 3b994bb c & d
   pick 8bd1a08 e
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'master' on 'a95d4d4'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working tree clean
```



```
in /e/DocumentX/Git/R  on @90c9cc38 rebas
└─❯ git reset HEAD~
Unstaged changes after reset:
M       a.txt
M       b.txt

in /e/DocumentX/Git/R  on @a95d4d40 rebas
└─❯ git add a.txt

in /e/DocumentX/Git/R  on @a95d4d40 rebas
└─❯ git commit -m "a"
[detached HEAD b218895] a
 1 file changed, 2 insertions(+), 1 deletion(-)

in /e/DocumentX/Git/R  on @b2188951 rebas
└─❯ git add b.txt

in /e/DocumentX/Git/R  on @b2188951 rebas
└─❯ git commit -m "b"
[detached HEAD 8101cfe] b
 1 file changed, 2 insertions(+), 1 deletion(-)

in /e/DocumentX/Git/R  on @8101cfe0 rebas
└─❯ git rebase --continue
Successfully rebased and updated refs/heads/master.
```

GIT BRANCHES

SQUASH IN MERGE

COMMAND

GIT SQUASH

- ❖ merging two branch

  $ **git merge --squash master**

  [note]: squash does not make commit. it use to reduced history to see clean tree.

- ❖ commit the merge history

  $ **git commit -m “merging”**

  [note]: try to avoid commit otherwise **squash** use does not make sense.

GIT REMOTES

BASIC

command

GITHUB

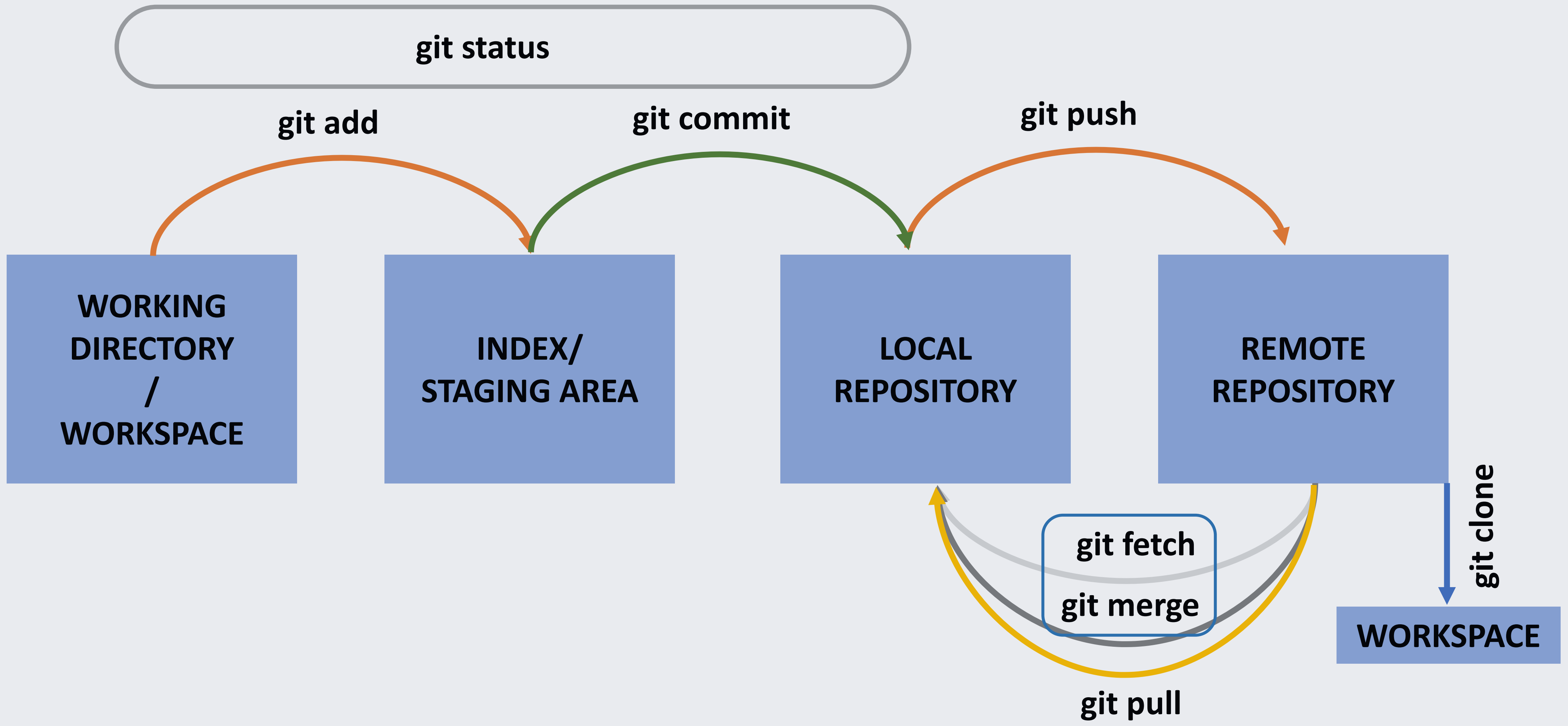
INTRO

GITHUB

INTRO

- ✓ **Github.com** is a website that hosts git repositories on a remote server.
- ✓ Hosting repositories on GitHub facilitates the sharing of codebases among teams by providing a GUI to easily fork or clone repos to a local machine.
- ✓ By pushing your repositories to GitHub, you will pretty much automatically create your own developer portfolio as well.

GIT WORKFLOW



GIT REMOTES

getting started with clone

git clone: create a new local repository by copying a remote repo

```
$ git clone https://github.com/aiimranh/furry-bury.git
```

result: new folder named “furry-bury” in current directory, containing new git repo.

contents identical to repo on server

origin set to **https://github.com/aiimranh/furry-bury.git**

GIT REMOTES

HTTPS

- ✓ Easy to start, no setup required.
- ✓ Must enter username and password every time you pull or push.

SSH

- ✓ Once set up, no username or password required.
- ✓ Need to create an SSH key on your computer and add it to your GitHub account.

GIT FETCH & MERGE[1]

- ❖ checkout a branch of the repository (as usual)

```
$ git checkout
```

- ❖ update a branch with the new version from the remote repository

```
$ git fetch                     only update commit change
$ git merge                     update the local repository
```

[note] : changes your working directory. make sure you pull before committing and merging to stay in sync! (especially on master, maybe someone else updated it) **.**

GIT PULL[2]

- checkout a branch of the repository (as usual)

```
$ git checkout
```

- update a branch with the new version from the remote repository

```
$ git pull
```

[note] : changes your working directory. make sure you pull before committing and merging to stay in sync! (especially on master, maybe someone else updated it) **.**

GIT PUSH

- ❖ create a new branch in the remote repository

```
$ git push -u origin gitone
```

  [note] : only checkout branch will be push.

- ❖ update the remote branch from the local branch afterwards

```
$ git push
```

[note] : only changes that are committed are pushed.
if the remote and local history diverge (e.g. forgot to pull before committing) pushes will be rejected.

GIT REMOTES

Using HTTPS

command

GIT PAT

Required: Personal Access Tokens Key

Go To GitHub

- Account settings/Developer settings/Personal access tokens/Generate new tokens/
- Take A Note & Select Scopes
- Generate Tokens & Copy The Tokens
- Save It on Notepad

GIT REMOTES

CLONE & PUSH

command

GIT HTTPS & PAT

copy https link
run it GitBash

```
$ git clone https://github.com/aiimranh/r.git
```

working on it.

```
$ git push or
$ git push –u origin main
```

enter the pat or just login in with browser.
[note] : pat need once but login may be need every push.

GIT REMOTES

INIT & PUSH

command

GIT HTTPS

create a new repository in GitHub i.e. **r *****
pushing to the remote new repo first time.

```
$ git branch -m main
$ git remote add https://github.com/aiimranh/r.git
$ git push -u origin main
```

enter personal access tokens. or just login in with browser.

[note] : pat need once but login may be need every push.

next time:

```
$ git push  or  $ git push -u origin main
```

GIT REMOTES

Using SSH

command

GIT SSH[1]

checking your ssh version

```
$ ssh -v
```

check if already have any ssh keys:

```
$ ls -la ~/.ssh
```

start the ssh agent in the background

```
$ eval $(ssh-agent -s)
```

generating ssh keys

```
$ ssh-keygen -t rsa -b 4096 -c “hmimran.in@gmail.com"
$ ssh-keygen -t ed25519 -C "hmimran.in@gmail.com"
```

GIT SSH[2]

Copy Public key to Add GitHub

```
$ clip < ~/.ssh/id_rsa.pub
```

Go To GitHub

Account settings/SSH and GPG keys/SSH Keys/
New SSH key/

Give A title & Paste SSH Key

Add SSH Key

GIT SSH[3]

Add Private key to SSH Agent :

```
$ ssh-add ~/.ssh/id_rsa
```

Test GitBash Authentication with GitHub

```
$ ssh -T git@github.com
```

GIT SSH[4]

show the ssh key list

```
$ ssh-add -l
```

changing passphrase of an existing ssh key

```
$ ssh-keygen -p
```

removing a single named private key from the ssh-agent

```
$ ssh-add -d /home/user/.ssh/id_rsa
```

removing all private keys from the ssh-agent

```
$ ssh-add -d
```

GIT REMOTES

CLONE & PUSH

command

GIT SSH[6]

copy ssh link from github repo

run it gitbash

```
$ git clone git@github.com:aiimranh/f.git
```

enter passphrase of ssh

working on it.

```
$ git push or
$ git push -u origin main
```

enter passphrase of ssh

GIT REMOTES

INIT & PUSH

command

GIT SSH[7]

create a new repository in github i.e. **u *****
pushing to the remote new repo first time.

```
$ git branch -m main
$ git remote add git@github.com:aiimranh/u.git
$ git push -u origin main
```

enter passphrase of ssh

next time:

```
$ git push
```

or

```
$ git push -u origin main
```

enter passphrase of ssh

GIT TAGS

command

GIT TAGS

- ❖ show tags list

  $ **git tag**
- ❖ create a tag

  $ **git tag v1.0**
- ❖ create a annotated tag

  $ **git tag -a v1.1 -m “tag message”**
- ❖ create a tag in specific commit hash

  $ **git tag v1.0 f073b04**

GIT TAGS

- show specific tag details

```
$ git show v1.0
```

- create a tag

```
$ git tag v1.0
```

- check same common tag list

```
$ git tag -l “v1.*”
```

GIT TAGS

- ❖ push a specific tag

  $ **git push origin v1.0**
- ❖ push a specific tag

  $ **git push origin --tags** or

  $ **git push --tags**

GIT TAGS

- ❖ remove tag from local

```
$ git tag -d v1.0
$ git tag --delete v1.0
$ git tag --delete v1.0 v1.1
```

GIT TAGS

- remove tag from remote

```
$ git push origin -d v1.0
$ git push origin --deete v1.0
$ git push origin:v1.0
$ git push origin -d v1.0 v1.1
```

GIT TAGS

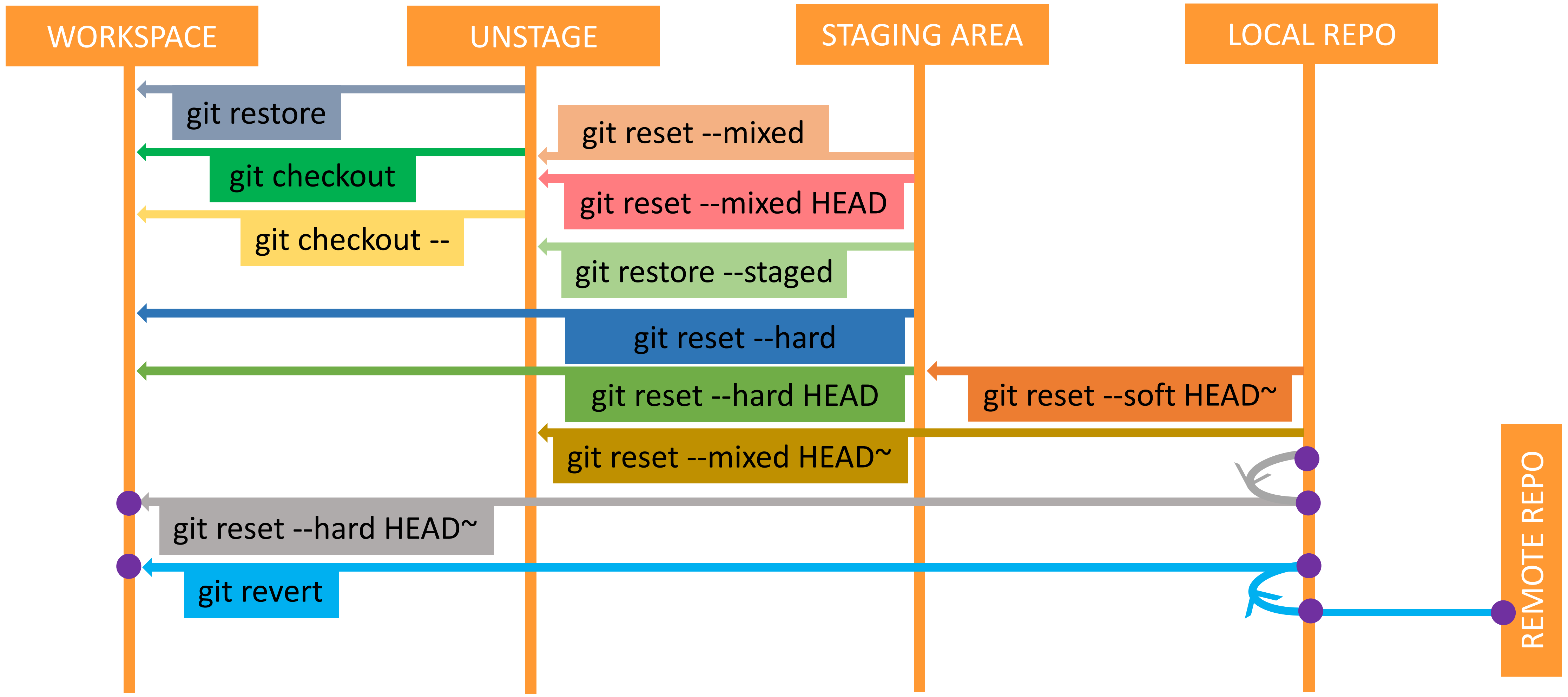
- check tag

```
$ git checkout -b dev_version v1.0
```

GIT UNDO

command

GIT WORKFLOW



GIT UNDO[1]

UNSTAGE TO WORKSACE

❖ unstage to workspace the all unstaged file

```
$ git restore .
$ git checkout .
$ git checkout -- .
```

❖ unstage to workspace the specific file

```
$ git restore one.txt
$ git checkout one.txt
$ git checkout -- one.txt
```

GIT UNDO[2]

UNSTAGE TO WORKSACE

- ❖ unstage to workspace the multiple files

```
$ git restore one.txt two.txt
$ git checkout one.txt two.txt
$ git checkout -- one.txt two.txt
```

- ❖ unstage to workspace the specific extension files

```
$ git restore *.txt
$ git checkout *.txt
$ git checkout -- *.txt
```

GIT UNDO[3]

UNSTAGE TO WORKSACE

- ❖ unstage to workspace the multiple files or specific extension files

```
$ git restore *.txt git.pdf
$ git checkout *.txt git.pdf
$ git checkout -- *.txt git.pdf
```

GIT UNDO[4]

STAGING AREA TO UNSTAGE

- ❖ staging area to unstage the all staging area files & folders

```
$ git reset
$ git reset --
$ git reset --mixed
$ git reset head
$ git reset --mixed head
$ git restore --staged .
```

GIT UNDO[5]

STAGING AREA TO UNSTAGE

- staging area to unstage the specific staging area files & folders

```
$ git reset new/
$ git reset one.txt
$ git reset -- one.txt
$ git reset head one.txt
$ git restore --staged one.txt
```

GIT UNDO[6]

STAGING AREA TO UNSTAGE

- staging area to unstage the specific extension staging area files & folders

```
$ git reset *.txt
$ git reset -- *.txt
$ git reset head *.txt
$ git restore --staged *.txt
```

GIT UNDO[7]

STAGING AREA TO UNSTAGE

- ❖ staging area to unstage the specific extension & multiple staging area files & folders

```
$ git reset *.txt git.pdf
$ git reset -- *.txt git.pdf
$ git reset head *.txt git.pdf
$ git restore --staged *.txt git.pdf
```

GIT UNDO[8]

STAGING AREA TO WORKSPACE

- staging area to workspace all the staging area files & folders

```
$ git reset --hard
$ git reset --hard head
```

GIT UNDO[9]

LOCAL REPO TO STAGING AREA

- local repository to staging area all the local repository files & folders

```
$ git reset --soft head~
```

GIT UNDO[10]

LOCAL REPO TO STAGING AREA

- ❖ local repository to staging area all the local repository files & folders

```
$ git reset --soft baa61c1
```

[note]: it does not touch workspace & unstage but staging area files remain it. so, after next commit will remain latest staging area files. so, all the commit history after **baa61c1** will be remove.

GIT UNDO[11]

LOCAL REPO TO UNSTAGE

- local repository to unstage all the local repository files & folders

```
$ git reset --mixed head~
```

GIT UNDO[12]

LOCAL REPO TO UNSTAGE

- ❖ local repository to unstage all the local repository files & folders

```
$ git reset --hard baa61c1
```

[note]: it does not touch workspace but unstage or modified files remain it. so, after next git add & git commit will remain latest unstage files. so, all the commit history after **baa61c1** will be remove.

GIT UNDO[13]

LATEST LOCAL REPO TO RECENT PREVIOUS LOCAL REPO

- ❖ latest local repo to recent previous local repo

  $ **git reset --hard head~** or

  $ **git reset --hard head^**

GIT UNDO[14]

LATEST LOCAL REPO TO RECENT PREVIOUS LOCAL REPO

- ❖ latest local repo to any previous local repo

```
$ git reset --hard baa61c1
```

[note]: all the commit history after **baa61c1** will be remove & workspace back to **baa61c1** this version & have clean working tree.

GIT UNDO[15]

$ **git reset --soft head~1**
does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). to staging area.

$ **git reset --mixed head~1**
$ **git reset head~1**
resets the index but not the working tree (i.e. the changed files are preserved but not marked for commit) and reports what has not been updated. to unstaging area.

GIT UNDO[16]

```
$ git reset --hard head~1
```

resets the index and working tree. any changes to tracked files in the working tree since <commit> are discarded.

GIT

REVERT

command

GIT UNDO[15]

REMOTE REPO TO LOCAL REPO & WORKSPACE

- latest remote repo to recent previous local repo

```
$ git revert head
$ git push
```

  [note]: new commit will added, no data loss.

- latest remote repo to any previous local repo

```
$ git revert baa61c1
```

  [note]: this make a merge conflict while reverting.

GIT

REVERT CONFLICT SOLVE

command

GIT UNDO[16]

REVERT CONFLICT[1]

- ❖ check current status

```
$ git status
```

- ❖ edit the conflict files
- ❖ add files in staging area

```
$ git add .
```

- ❖ revert continue

```
$ git revert --continue
```

GIT

REVERT CONFLICT

command

GIT UNDO[17]

REVERT CONFLICT[1]

- check current status
```
$ git status
```
- edit the conflict files
- add files in staging area
```
$ git add .
```
- cancel the current operation & set to pre-sequence state.
```
$ git revert --abort
```

GIT

REVERT CONFLICT

command

GIT UNDO[16]

REVERT CONFLICT[]

- skip the current commit and continue with the rest of the sequence

```
$ git revert --skip
```

- forget about the current operation in progress. can be used to clear the sequencer state after a failed cherry-pick or revert.

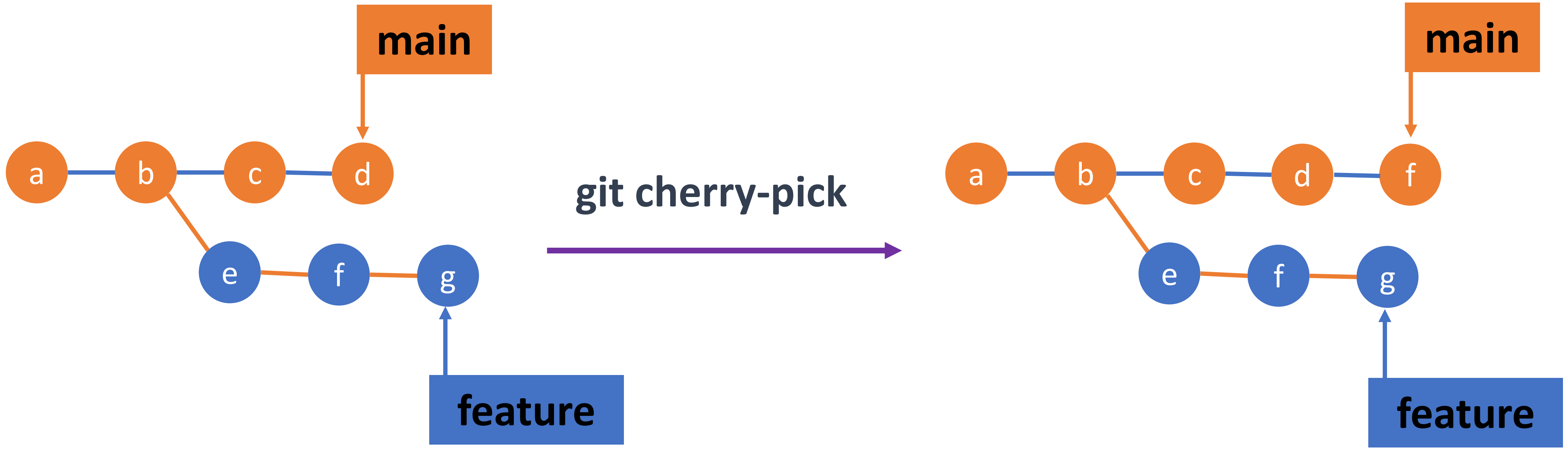
```
$ git revert  --quit
```

GIT CHERRY-PICK

command

GIT

CHERRY-PICK



GIT[1]

CHERRY-PICK

- go to the feature branch & copy the hash that will be picked.

```
$ git switch feature
$ git log --abbrev-commit
```

- go to the main branch where will be merge that picked commit

```
$ git cherry-pick 26a898c
```

GIT[2]

CHERRY-PICK

- the **--edit** option will cause git to prompt for a commit message before applying the cherry-pick operation.

```
$ git cherry-pick 26a898c --edit
```

- the **--no-commit** option will execute the cherry pick but instead of making a new commit it will move the contents of the target commit into the working directory of the current branch.

```
$ git cherry-pick 26a898c --no-commit
$ git commit -m "message"
```

GIT

BISET

GIT

BISET

The idea behind git bisect is to perform a binary search in the history to find a particular regression. Imagine that you have the following development history:

```
... --- 0 --- 1 --- 2 --- 3 --- 4* --- 5 --- current
```

You could try to check out each commit, build it, check if the regression is present or not. If there is a large number of commits, this can take a long time. This is a linear search. We can do better by doing a binary search. This is what the git bisect command does. At each step it tries to reduce the number of revisions that are potentially bad by half.

You'll use the command like this:

```
$ git stash save
$ git bisect start
$ git bisect bad
$ git bisect good 0
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[< ... sha ... >] 3
```

GIT

Reference

Reference:

- ✓ https://www.youtube.com/watch?v=oe21Nlq8GS4
- ✓ https://www.youtube.com/watch?v=8JJ101D3knE
- ✓ https://www.youtube.com/watch?v=JOIL6gof2BA
- ✓ https://git-scm.com/doc
- ✓ https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud
- ✓ https://www.youtube.com/watch?v=eBmG7keE8B4
- ✓ https://www.youtube.com/watch?v=ElRzTuYln0M
- ✓ https://www.youtube.com/watch?v=D7JJnLFOn4A
- ✓ https://www.youtube.com/watch?v=tcd4txbTtAY
- ✓ https://www.youtube.com/watch?v=S0_pOvrfN0U
- ✓ https://www.youtube.com/watch?v=k0B9Lz2Zw38

END!