Git Workshop

SENG302

Precursor

- Have your tutorial repository and IntelliJ open
  - We will use IntelliJ for this workshop
- Work in pairs
- Use one computer for the exercises
  - Screenshare if you're working with someone over Zoom!
- Help each other

Pairs

1: lcg37, dkj23
2: smc397, ajs418
3: gca73, aco155
4: ssi162, dkp33
5: jel119, ckh35
6: hli165, tmo89
7: abc56, hma112
8: led43, dzh80
9: sae53, mga114
10: kvn17, mwe50
11: dpa107, krm97
12: rho66, yyu69
13: ycr14, jth141
14: daa59, amh284
15: sne69, jte52
16: tfi18, pis19
17: zkh22, cal127
18: lra63, zhu45
19: hre56, pob16
20: emo58, bsa57
21: aca170, scl113
22: ojo26, fma107
23: bcl7, cdu51
24: rlh89, bjs185
25: dal113, jbh110
26: wjo34, jcl165
27: sva73, jdm183
28: ieb15, gha95
29: jdt59, yya139

Scope of Workshop

- Git Repositories
- Making Changes
- Reset, Checkout, Revert
- Branches and Merge Conflicts
- Mailmap
- Tagging
- Bonus Topics
  - Rebasing
  - Interactive Rebasing

Git Repositories

What is Git

Git helps teams collaborate on source code during software development by tracking changes to files.

- Git is a distributed **Version Control System (VCS)**
- Industry standard
- Enables **collaboration**
- Records changes made to code in a **repository**
  - Tracks **project history**
  - Allows you to **revert changes**

Git Repositories

- **Remote** Repository
  - Hosted on the internet (i.e. in GitLab)
- **Local** Repository
  - Local version of remote repository on your machine

Setting Up Git Credentials

Setting up global **configuration** for username and email

- `git config --global user.name "My Name"`
- `git config --global user.email "nme123@uclive.ac.nz"`

Setting configuration for project level → Just remove `--global`

- `git config user.name "My Name"`
- `git config user.email "nme123@uclive.ac.nz"`

- These need to be setup on every computer you develop from (e.g. laptop, lab machine)
- To check if your credentials are stored → `git config --list`

How to Set Up a Local Repository

- Initialising a Repository
  - `git init`
  - **Initialises** a Git repository within a local directory

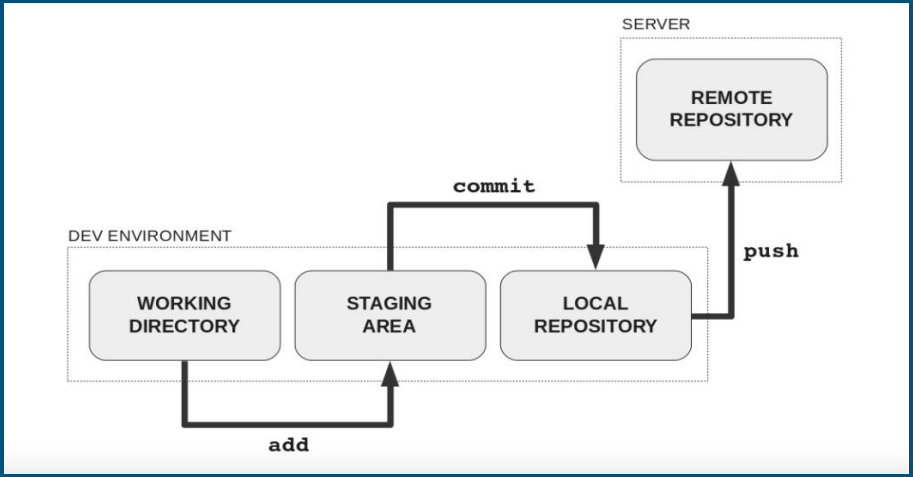
- Cloning an existing Git Repository
  - `git clone`
  - **Initialises then clones/**copies a remote Git repository to a local directory

Exercise 1: Cloning a Repository

- Open up terminal and `cd` to a directory you want to use for this workshop
- Clone the repository url from GitLab to this directory using `git clone`
- Open this directory on IntelliJ
- Have either your previous terminal or the terminal **through IntelliJ** opened for the next exercises
- If you have problems, ask your teammates or one of the teaching staff

Making Changes

Making Changes



- Working Directory
  - Make changes to your files
- Staging Area
  - **Track** new/modified files to staging area → `git add`
- Local Repository
  - **Commit** your changes to your local repository → `git commit -m "your commit message"`
  - Always specify a commit message
- Remote Repository
  - After committing, always **pull** before **pushing** your changes → `git pull`
  - **Push** your changes to your remote repository → `git push`

Git Pull → Pulling and Merge Conflicts

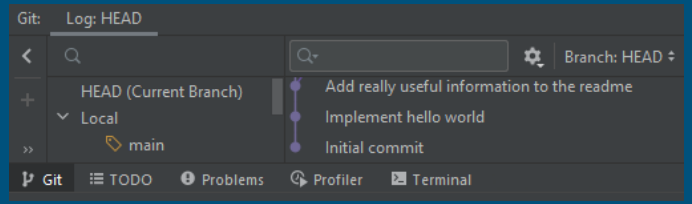
- Always pull before you push
  - Ensures local repository is in sync with remote repository
  - Your teammates might be making changes to the same file
- If they pushed their changes then you can get a **merge conflict**
- Quick tip, use a GUI to resolve merge conflicts because it’s so much easier

Helpful Commands

- `git status`
  - Shows the current state of your Git working directory and staging area
- `git log`
  - Shows the commit history
- `git log --oneline`
  - Shows the commit history with one line per commit

```
d4ae730 (HEAD -> master, origin/master) Remove functionalit
y to subconsciously watch tv when asleep
43def6b Fix functionality to do work when i sleep and add t
est cases for cooking functionality when asleep
6fb244c Implement Functionality to Do Work Automatically wh
ile I sleepcommit
4a59c90 My third commit
```

Command Line



IntelliJ

Exercise 2: Making Some Changes

- **Create a random text file** inside your local repository and `add`, `commit`, and `push` it to the remote repository
- A pop up from IntelliJ may allow you to automatically add new file
  - For the sake of this tutorial, click cancel
- Along the way, use `git status` to see state of your working directory

STEP 1: **Add (can use either commands)**

- Everything : `git add .`
- One file : `git add "filename.txt"`

STEP 2: **Commit**

- `git commit -m "Enter commit message"`

STEP 3: **Pull and Push**

- `git pull`
- `git push`

STEP 4:

- `git log` to display commit history
- Type "q" to quit log mode

Resetting, Checking Out, and Reverting

Git Reset → Undo Changes **Before Commiting**

- Use this to return your working directory to the **last committed** state

- `git reset --mixed`
  - Unstages tracked files but does not revert your changes
  - Essentially undoes `git add`
- `git reset --hard`
  - Restores to the last committed state (HEAD)
  - Removes any changes you've made
  - Useful if you decide to completely revert your changes

Git Checkout → View Previous Commits

- `git checkout <COMMIT-HASH>`
  - Views a previous commit state from your directory
- Find a specific commit hash using `git log`
  (i.e. `git checkout d4ae730`)
- You can look at files, run tests, even edit files and you won't lose the "current" state of the project (HEAD)
- To return to "current" state → `git checkout <BRANCH-NAME>`
  (i.e. `git checkout master`)

Git Revert → Undo Changes **After Commiting**

- `git revert <COMMIT-HASH>`
  - Reverts the changes in a previous commit in your commit history
  - This creates a “revert” commit
  - **Undoes everything** in that particular commit but does not rewrite history
- To revert commits you made **before pushing**
  - `git revert HEAD`

Exercise 3a:
Resetting, Checking Out & Reverting

Resetting (mixed and hard)

1. Make as much changes you want to the files in the repository
2. Stage your changes (`git add .`) **but don't commit anything!**
3. Run `git status` to see your **tracked** files
4. Run `git reset --mixed`
   - Run `git status`
   - Notice how your changes are still present but are just **untracked**
5. Restage your changes **(redo step 2 and 3)**
6. Now run `git reset --hard`
   - Run `git status`
   - Notice how your changes have now been completely **removed**

Exercise 3b:
Resetting, Checking Out & Reverting

Checking Out and Reverting

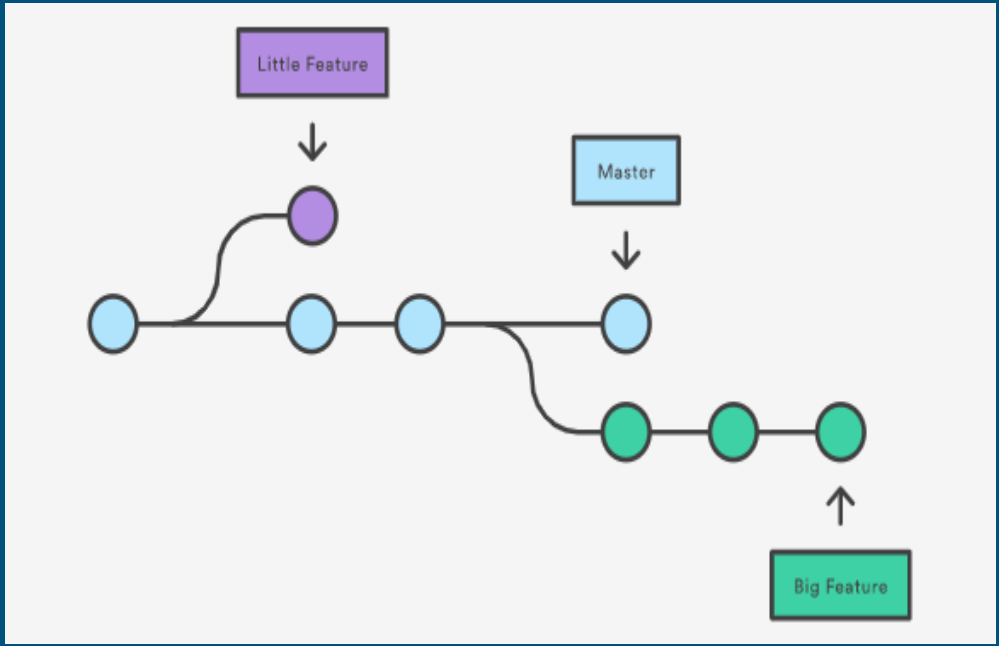
1. Run `git log --oneline` to view your history of commits
2. Checkout the commit hash of the earliest commit → `git checkout <commit-hash>`
   - Observe your project from IntelliJ, your directory is now the exact state of the earliest commit
3. Return to “current state” of the project → `git checkout master`
4. Revert your **most recent commit** → `git revert <commit-hash>`
   - Running this creates a commit and opens up a text editor with a commit message
   - Type in :wq to exit the editor (you don’t need to change anything)
5. Run `git log --oneline` and you can see the revert commit present
6. Run `git push` to apply changes in your remote repository

Branching

Git Branch

- **Branches** are independent lines of development

- Why Branching?
  - Less likely for merge conflict to occur
  - Main line should be bug-free



Git Branch

- `git checkout -b “my-new-branch”`
  - **Creates** and **checkouts** a new branch called “`my-new-branch`” from your current branch
- `git checkout “master”`
  - Checkouts a branch called “`master`”
- `git push -u origin “my-new-branch”`
  - Pushes the branch “`my-new-branch`” to remote repository
- `git branch`
  - Lists branches in repo and highlights branch you are currently in

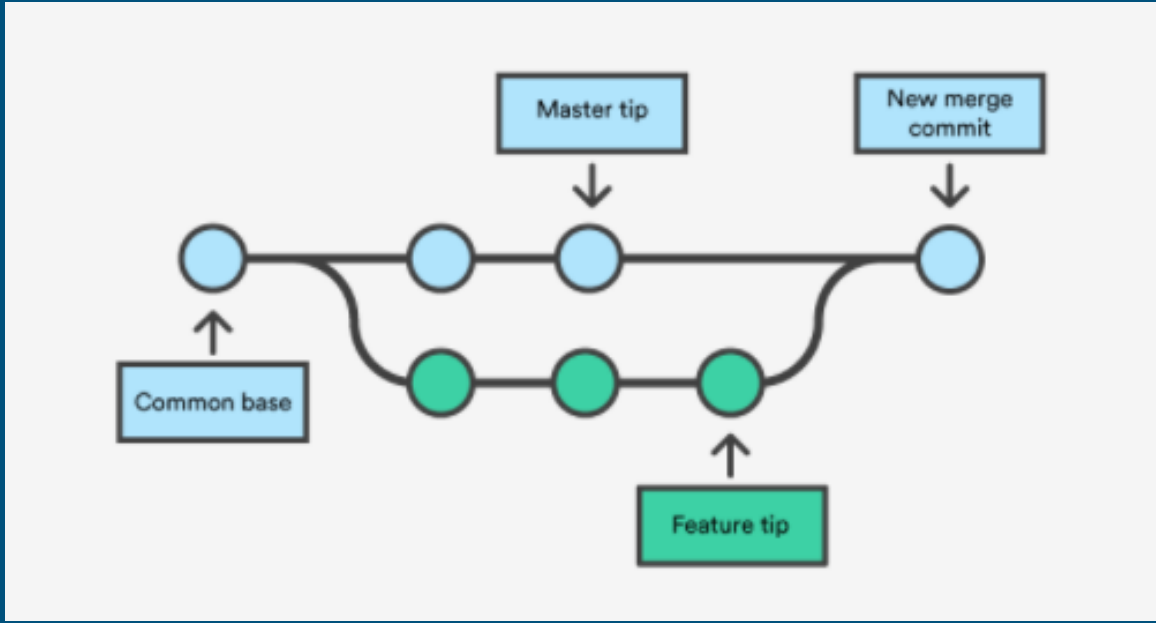
Git Branch

- `git branch -d “My New Branch”`
  - Deletes a branch locally

- **Branching Strategies**
  - Task Branching
  - Feature Branching
  - Story Branching

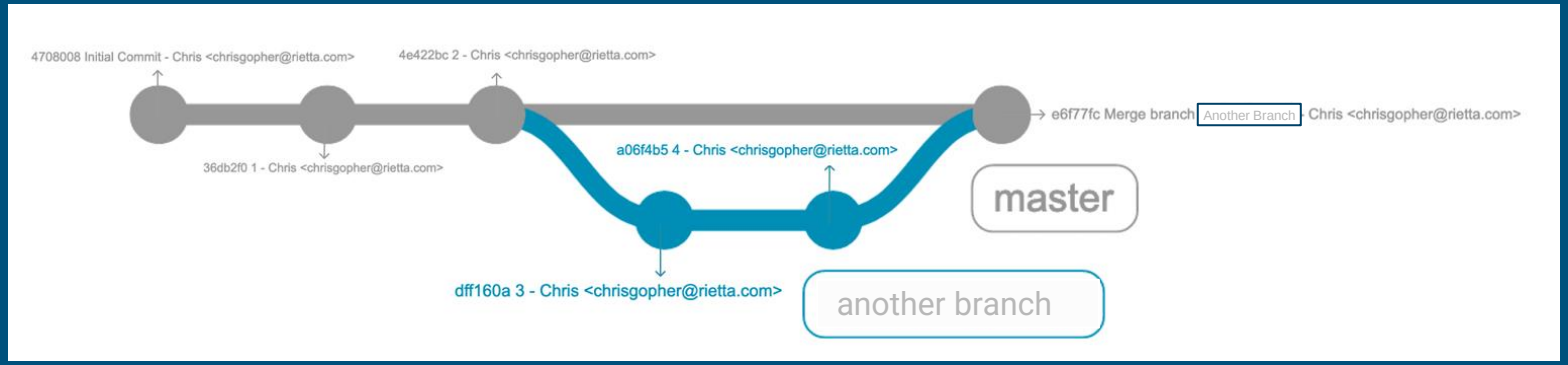
Git Merge → Merging Branches

- Combines multiple sequences of commits into one unified history



Git Merge → Merging Branches

- Merging Branches
  - `git checkout` to the branch you want to merge INTO
  - Then run `git merge <branch-name>`
    where <branch-name> is the branch you want to merge FROM
  - Both branches should be up-to-date with the latest remote changes



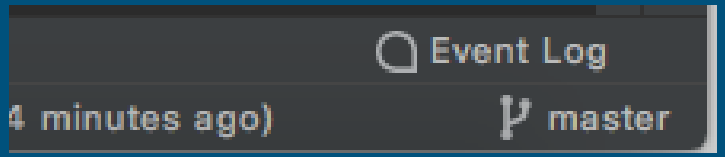
Git Merge → Merge Conflicts

- When merging branches, you may encounter **merge conflicts**
- Merge conflicts occur when:
  - Two people have changed the same lines in a file
  - If one developer deleted a file while another developer was modifying it
- To merge the branches → Resolve conflicts
  - Using the terminal
  - **Using a GUI like IntelliJ**

Exercise 4a: Branching and Merge Conflicts Using Git Commands

1. Create and checkout a new branch from the master branch on your local repository and push it to the remote repository
   - Run `git checkout -b "branch-name"`
   - Then push branch to remote repo `git push -u origin "branch-name"`
   - Run `git branch` to view your list of branches
1. Open the file "MyName.txt" and modify it based on the instructions
2. Try and merge the existing branch "another-branch" into your branch
   - `git merge "another-branch"` → **you will get a merge conflict**
3. To resolve conflict, open up the file "MyName.txt"
   - Modify it so it looks like your changes **from step 2** and save it
4. Track your merging changes (`git add`)
5. Use "`git commit`" to conclude merge

Exercise 4b: Branching and Merge Conflicts Using a GUI (IntelliJ)



1. On the bottom right of IntelliJ, click `master`
   - This will open up a list of branches
2. Under “Local Branches”, click “master”, then click “New Branch from Selected...”
   - Name your branch “my-Branch” and click create
   - This **creates** and **checkouts** “my-Branch”
3. Open the file “MyName.txt” and modify it based on **one of your details**
4. Commit your changes to “my-Branch” **but don’t push**
5. **Repeat Step 2** and create another new branch, by substituting “my-Branch” with **“other-Branch”**
6. **Repeat Step 3** but modify “MyName.txt” with your **partner’s detail**
7. Commit your changes to “other-Branch” **but don’t push**
8. Click the bottom right to view list of branches
9. Click “my-Branch” under “Local Branches”, then click “Merge into Current...”
   - A window pop-up saying you have a merge conflict
10. Select the changes you want to retain. Click **merge** and **accept your changes**
11. Run `git push -u origin my-branch` to push your branch to repo

Mailmap

Mailmap

- Used to correct commit author information
- Commit authors can be checked using the `git log` command

```
$ git log
commit 31cdad6b5486f98579f9eddb42e2a05519618c4f
Merge: 7a17a2a df782e8
Author: Griffin Baxter <grb96@uclive.ac.nz>
Date:   Fri Feb 18 15:10:26 2022 +1300
```

- If the author is not correct, the code may not be marked for grading

Correct format: `First-name Last-name <user-code@uclive.ac.nz>`

- This can be fixed, by adding a new line to the `.mailmap` file in the root of your project repository

```
First-name Last-name <user-code@uclive.ac.nz> Incorrect-name <incorrect-email>
```

Tagging

Git Tag → Tagging Your Sprint Delivery

- `git tag -a <tagname>`

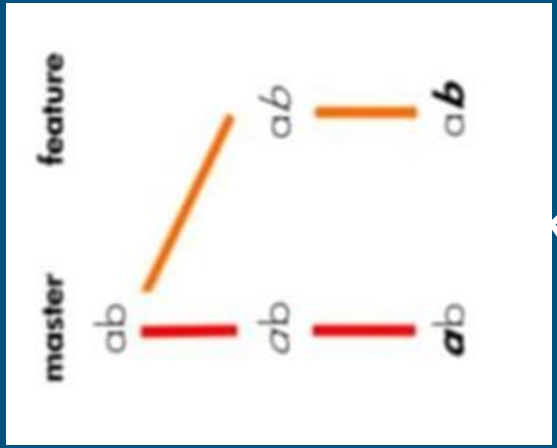
Tagging your sprint delivery

- Make your last commit
- Add a tag
  - `git tag -a sprint_X.Y -m “Deliverable for sprint X”`
  - X is the sprint number, Y is the revision number (should start at 0)
  - `git tag -a sprint_1.5 -m “Deliverable for sprint 1”`
- Push a tag
  - `git push origin --tags`

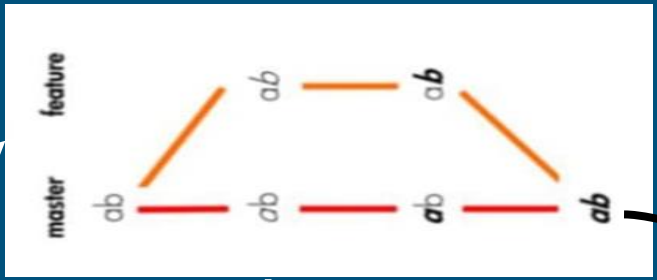
Rebasing

(be careful if used in 302 project)

Git Rebase → An Alternative To Merging

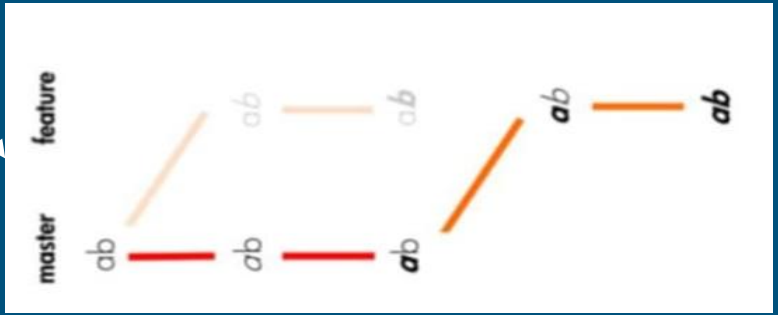


Your project history



Merge commit

`git merge`



No merge commit

`git rebase`

Git Rebase → An Alternative To Merging

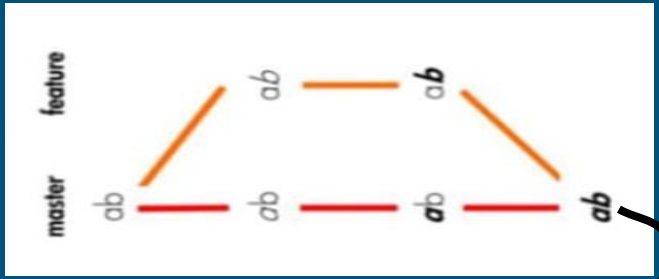
- `git merge`
  - Ties the history of two branches
  - Only target branch is changed, creates a **merge commit**
  - Project/commit history is preserved

- `git rebase`
  - Rewrites the history of target branch to only include feature branch commits → Rewrites these commits, so history is rewritten
  - A linear and much cleaner project history
  - No merge commits are added

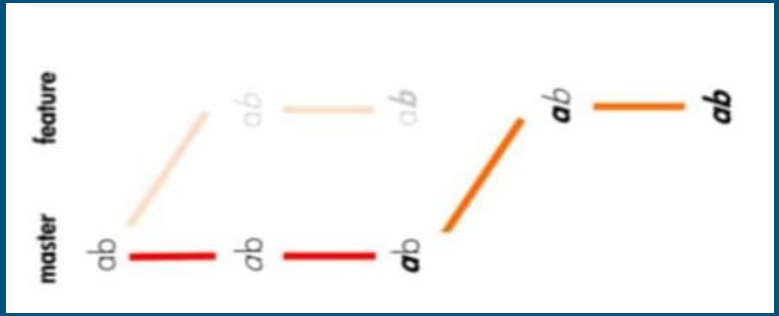
Git Rebase → An Alternative To Merging

Main use of rebasing we will cover

- To tidy up commit history of a branch before merging into the main branch
- Only **rebase** your commits **BEFORE YOU PUSH**



Merge commit



Exercise 5: Rebasing Branches

1. Checkout master → `git checkout "master"`
2. Create a branch from master called "feature" → `git checkout -b "feature"`
3. Checkout master, modify MyName.txt and push a new commit
   - To simulate changes in master while you are working on your own branch
4. Checkout the "feature" branch and **create and commit a new file**
5. Repeat the above step **2 more times.**
6. Run `git log --oneline` and observe the commit hash for those 3 commits
7. Now rebase master into the "feature" branch → `git rebase master`
8. Run `git log --oneline` once more
   - Observe how the commit hashes from step 4 are different and have been ***rewritten***
   - Also the commit hashes are now on top of master's HEAD
9. Checkout master and rebase the "feature" branch into master → `git rebase feature`
10. Check your history (`git log --oneline`) and your master branch has been rebased **without any merge commits**

Interactive Rebasing

(also be careful if used in 302 project)

Interactive Rebasing

- Rebasing is used to rewrite the commit history of a branch
- Interactive rebasing allows you to have more control of rebasing
- Some commands we will cover:
  - `pick` → uses a commit
  - `reword` → rewrites commit message
  - `squash` → combines multiple commits and allows you to rewrite commit message
  - `fixup` → like squash but discards commit message

- `git rebase -i HEAD~X`
  - opens an interactive rebase session for the last X commits

```
pick 6633b5a Message #1
pick 3a03f0d Message #2
pick 657897b Message #3

# Rebase 3598fe2..657897b onto 3598fe2 (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

Interactive Rebasing

- Order of **most recent to least recent commit** is from **bottom to top**

```
pick 9d3cd65 Make changes to activity screen
pick 807f54c Add more changes to activity screen
pick 502548d Implement log1n scr3n prooperly
pick 7d1c683 Irrelevant commit
pick 7b6cf4a Relevant commit
```

Interactive rebasing

- `pick`, `reword`, **`fixup`**, `squash`

- **The word to the left of each commit lets us perform rebase commands**
  - i.e. `pick`, `reword`, `squash`, `fixup`

- **For example, tidying multiple commits that should really only be a single commit → use `fixup`**

```
pick 9d3cd65 Make changes to activity screen
fixup 807f54c Add more changes to activity screen
```

- **`fixup`** → Discards commit message and uses commit message of parent commit (the earlier commit)
- Results to a new commit with a new commit hash

```
550644e Make changes to activity screen
```

Interactive rebasing

- `pick`, `reword`, `fixup`, **`squash`**

`squash`

- Combines multiple commits and allows you to edit commit message
- Results to a new commit with a new commit hash

```
pick a13af06 Implement more things
squash d5c05a0 Test things
```

```
ae7c5f7 (HEAD -> master) Implement and tested things
```

Exercise 6: Interactive Rebasing

1. On your master branch, make 5 separate commits with any changes
2. Enter interactive rebase to rebase those 5 commits → `git rebase -i HEAD~5`
3. Remember, order of commits from **most recent to least recent** is from **bottom to top**
4. Navigate to your list of commits, then **type in "i"** to enter INSERT mode and do the following:
   - Rename your oldest commit (commit on top) with `reword`
   - Squash the third commit from the top with the second commit from the top with `squash`
   - Fixup your most recent commit (on the bottom) the one above it with `fixup`
1. Type in → "esc :x" and then enter "i"
   to begin `rewording` your oldest commit
1. Type in → "esc :x" and then enter "i"
2. to begin `squashing`
3. Run `git log --oneline`
   and observe your changes
1. Push your changes

```
reword 2ef2941 Imploement some changes
pick e13ceeb Implement user model
squash 8f9acd8 Add documentation to user model
pick cabeb40 Implement Activities screen
fixup 9d189c2 Implement Activities screen

# Rebase b1cf685..9d189c2 onto b1cf685 (5 commands)
-- INSERT --
```

Tips on Rebasing

1. Only rebase YOUR COMMITS before pushing to remote repository (origin)
   - Rebasing after pushing means you're rewriting existing history
   - Don't do this in SENG302

1. Use rebase to tidy up commits
   - If you have two similar commits with similar changes
   - i.e. "Implement user model" and "fix typo in user model"

3. Rebasing can help keep your history tidy but BE CAREFUL

Git GUI Clients

- IntelliJ/WebStorm built in
- GitKraken
- SmartGit
- Git Extensions
- Fork

Git Workshop

SENG302