Version control and git

CSE 403 Software Engineering

Winter 2025

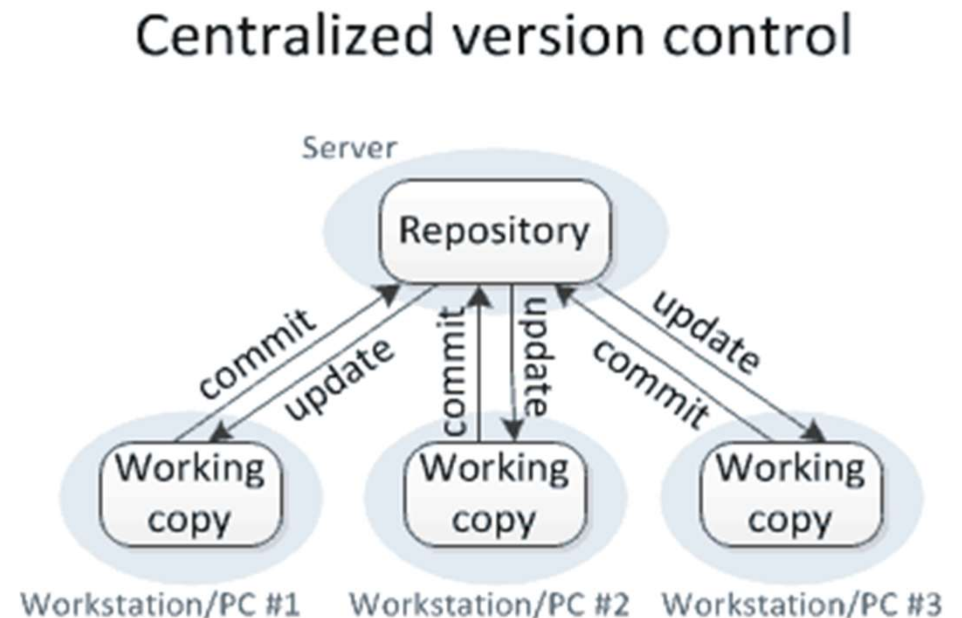
Today's Outline

1. Version control: why, what, how
2. Git: basic concepts for working with a team

See git references and readings on the Calendar

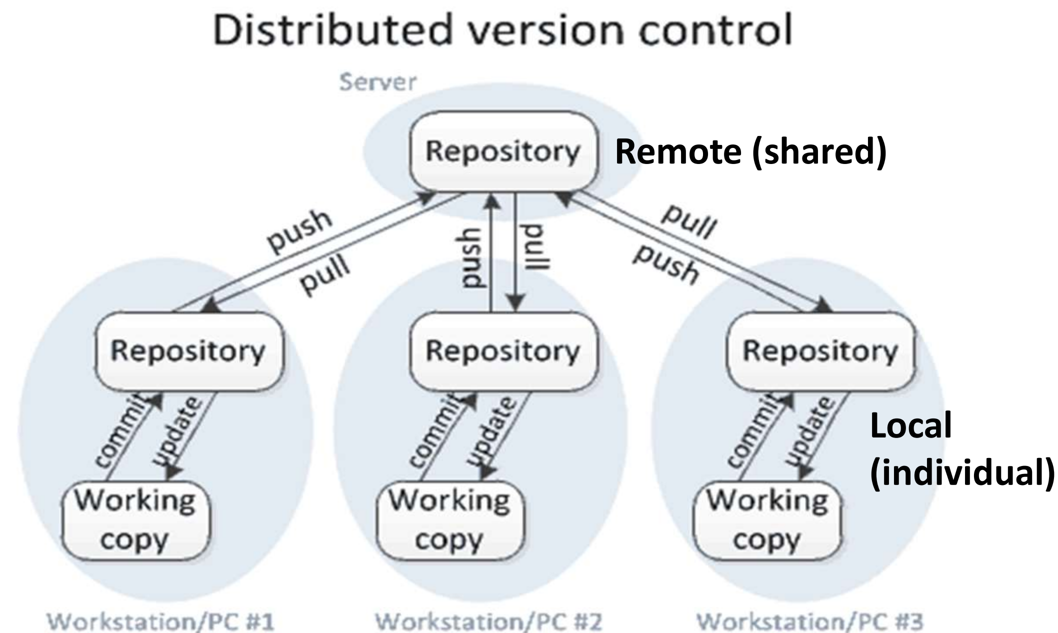
Centralized version control

- **One central repository**
  It stores a history of project versions
- Each user has a **working copy**
- A user **commits** file changes to the repository
- Committed changes are immediately visible to teammates who **update**
- Examples: SVN (Subversion), CVS



Distributed version control

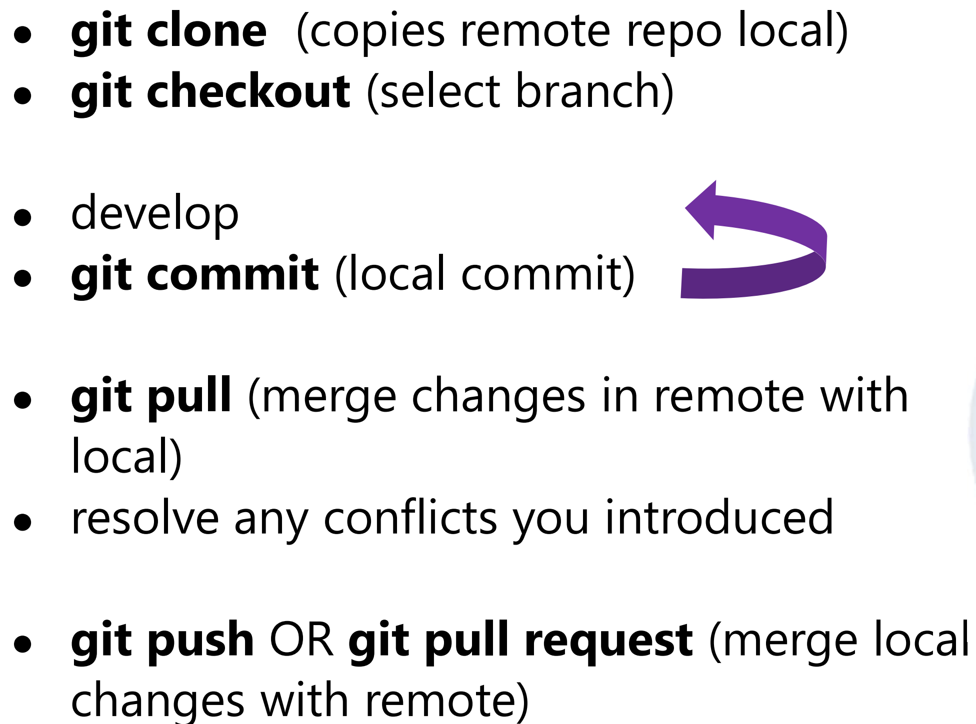
- **Multiple copies of a repository**
  Each stores its own history of project versions
- Each user **commits** to a **local** (private) repository
- All committed changes remain local unless **pushed** to another repository
- No external changes are visible unless **pulled** from another repository
- Examples: Git, Hg (Mercurial)



An example git workflow

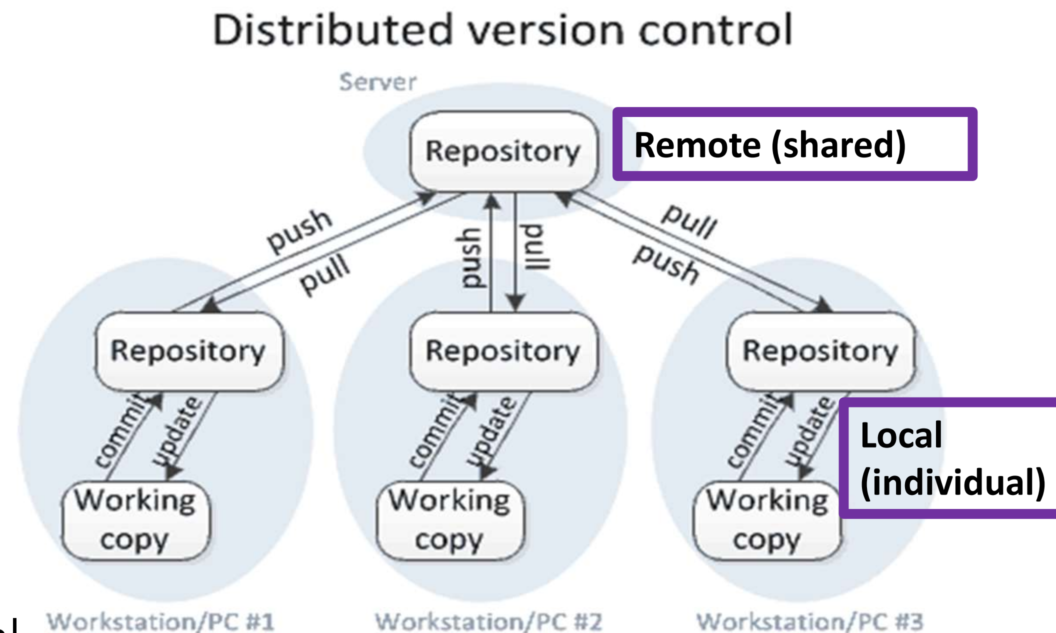
- **git clone** (copies remote repo local)
- **git checkout** (select branch)

- develop
- **git commit** (local commit)



- **git pull** (merge changes in remote with local)
- resolve any conflicts you introduced

- **git push** OR **git pull request** (merge local changes with remote)



Git quiz commands (short definitions)

- **git clone** – copy remote repo to local for development
- **git fork** (github command) – make a new remote repo
- **git cherry-pick** – apply identified commits to the branch
- **git fetch** – create a local branch with latest from the remote repo for comparison
- **git pull** – merge latest from the remote repo into your local branch (= git fetch + git merge)

Using git with a team for a product delivery

What if you have to support:

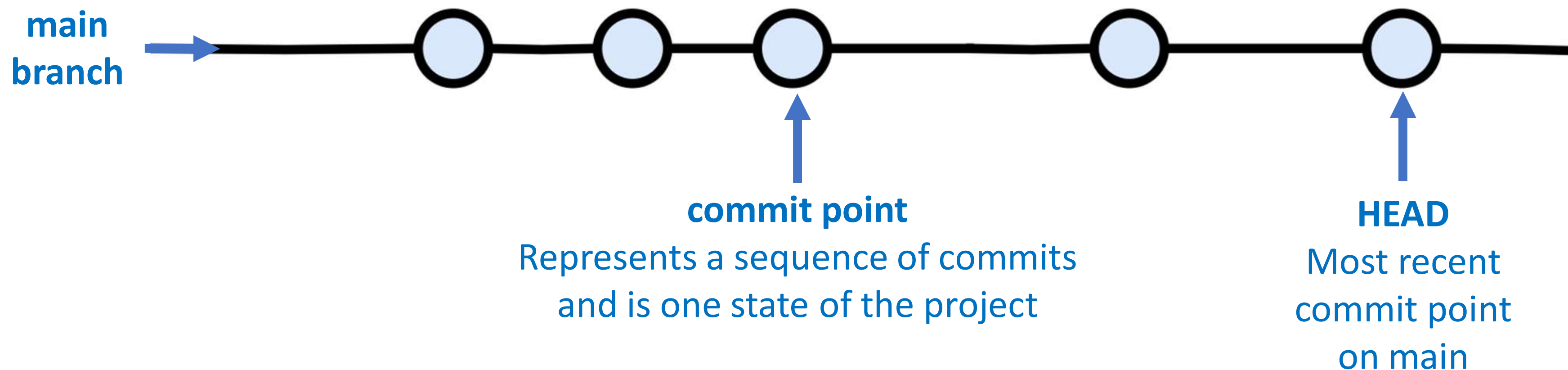
- Version 1.0.4 and version 2.0.0
- Windows and macOS
- Adding a feature
- Fixing a bug

Git has 3 ways to represent multiple histories:

- **Branch**: Start a parallel history of changes to the code in the repository
- **Clone**: Make a copy of the repository locally to work on code changes
- **Fork**: Make a copy the repository that will not necessarily be merged back with original (but can be through a pull request)

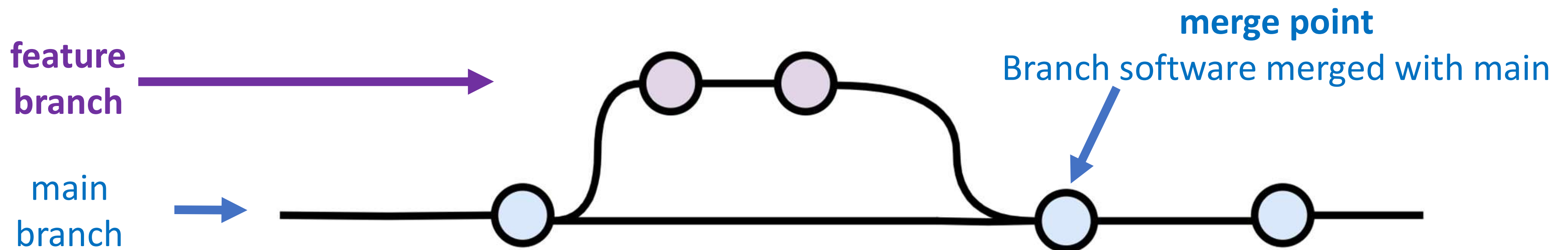
Branches

- Git has a basic concept of a branch
- There is one **main** development **branch** (main, master, trunk)
- You should always be able to ship “**working software**” from **main**



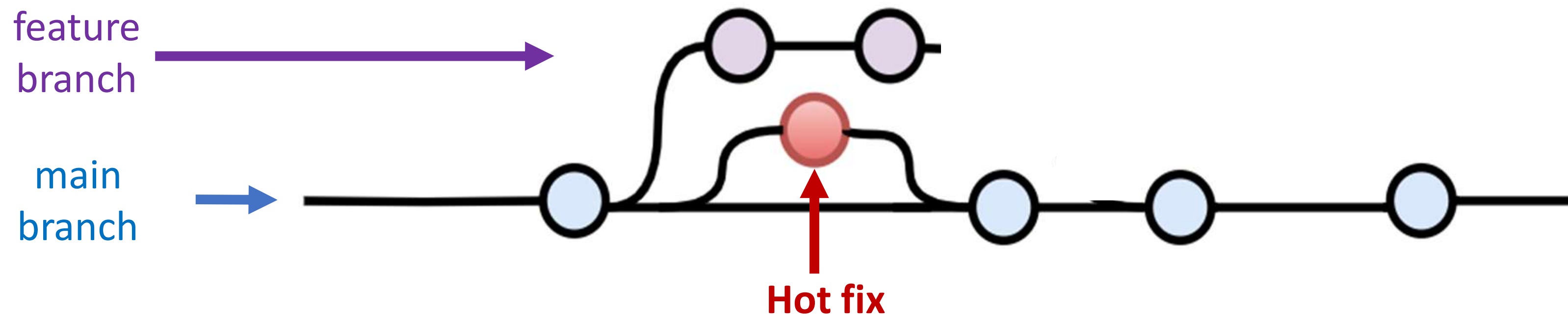
Branches

- To develop a feature, add a new branch
  - And then later merge it with main
  - Lightweight, as (conceptually) branching simply copies a pointer to the commit history
  - **Why is this a good practice?**



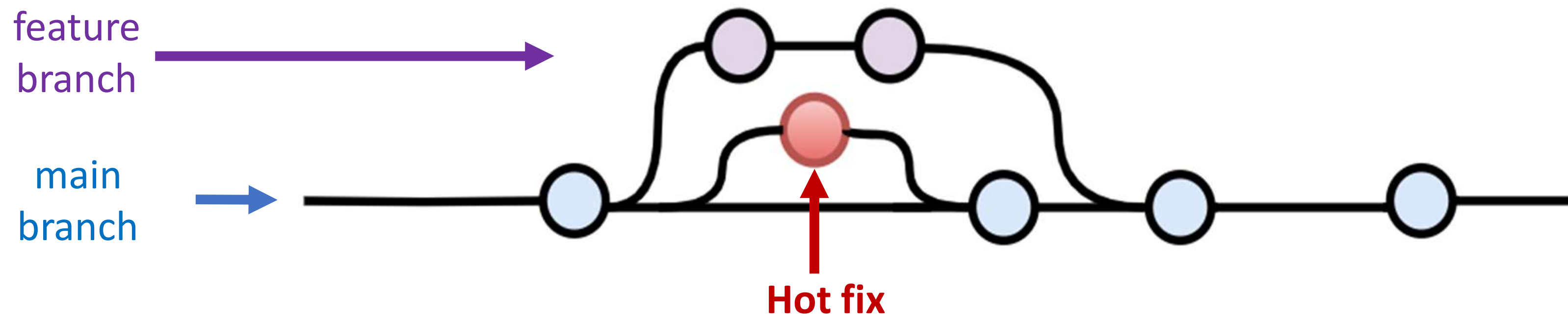
Branches

- To develop a feature or bug fix, add a new branch
  - Why? Keeps main **always working** and allows for **parallel development**



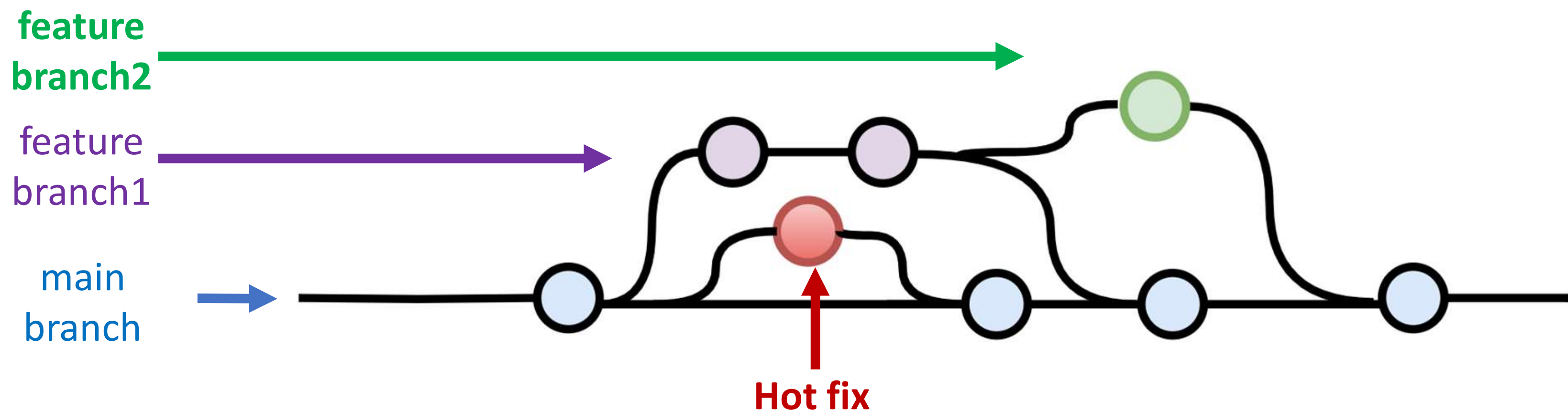
Branches

- To develop a feature or bug fix, add a new branch
  - Why? Keeps main **always working** and allows for **parallel development**



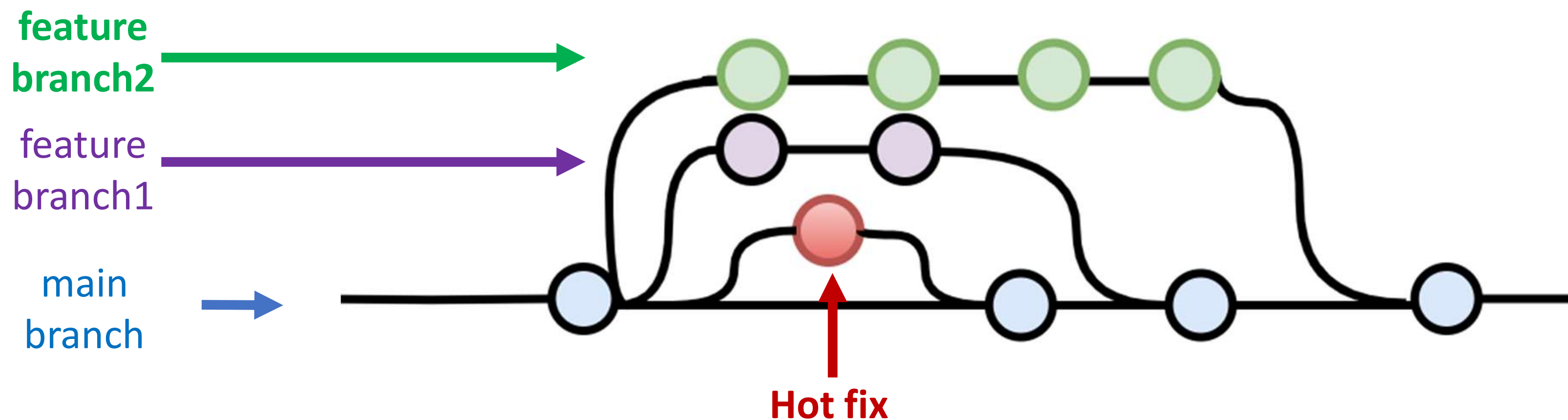
Branches

- To develop a feature or bug fix, add a new branch
  - Why? Keeps main **always working** and allows for **lots of parallel development**



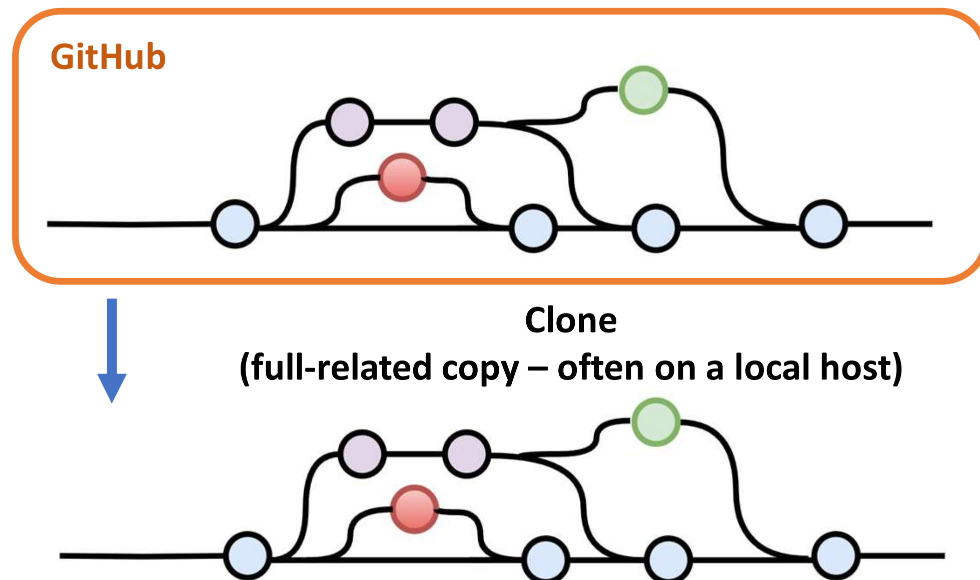
Branches

- To develop a feature or bug fix, add a new branch
  - Why? Keeps main **always working** and allows for **lots of parallel development**



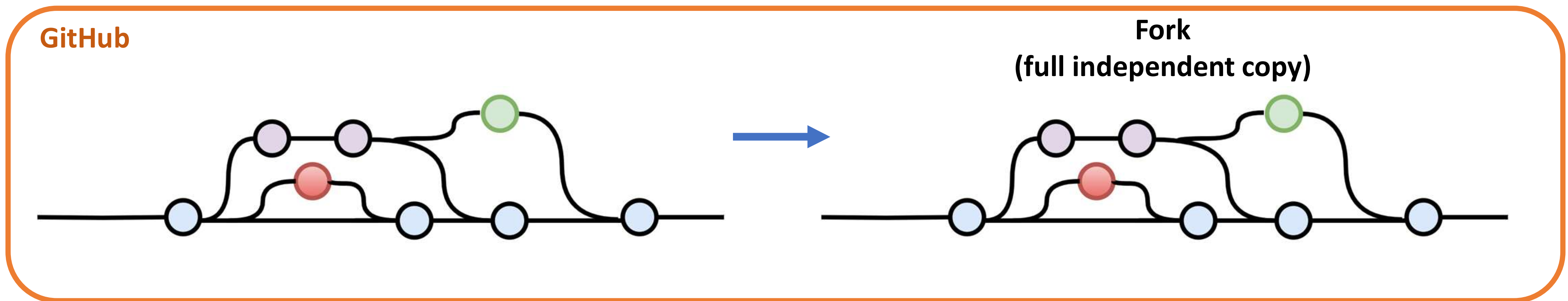
Cloning

- When you **clone** a repo you are creating a **local copy** on your computer that you can sync with the remote
- Ideal for contributing directly to a repo alongside other developers
- Can use all git commands to commit back to remote repo



Forking (github concept)

- Creates a complete **independent copy** of the repository (project)
- Allows you to evolve the repo without impacting the original
- If original repo goes away, forked repo will still exist



- It's possible to update the original but only with **pull requests (original owner approves or not)**

Which would you choose?

Branch (parallel dev), **fork** (in github), **or clone** (to local machine)?

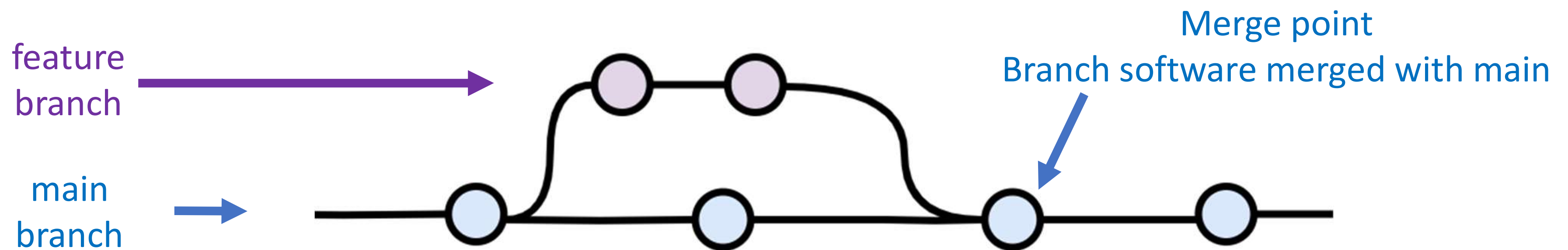
Scenario: CSE403 Class Materials GitHub Repo

1. Fix the bugs in the in-class assignment-1
2. Create instance for working on my laptop
3. Create instance for CSE413 to leverage structure of CSE403
4. Create area for Wi25 specific material

Merging branches

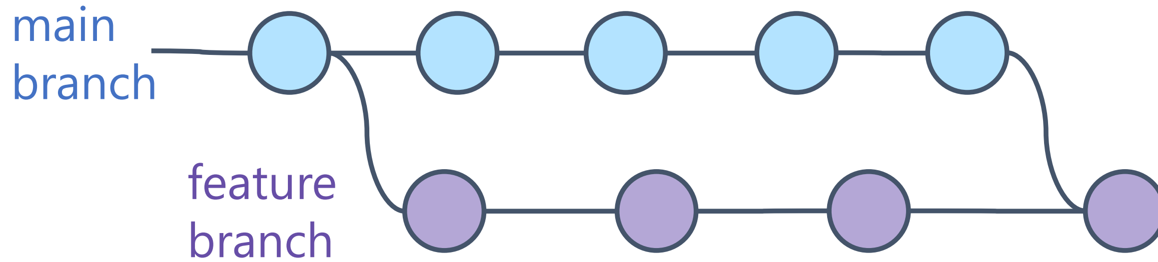
Merging branches

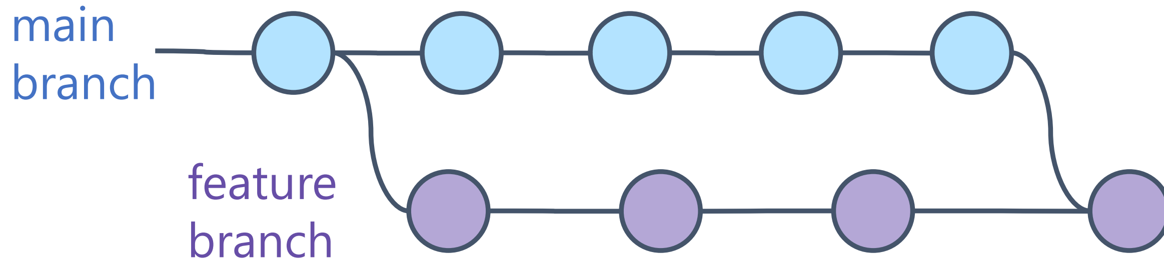
- Branches can get out of sync
  - **merge** incorporates changes from one branch into another
  - Life goal of a branch is to be merged into main as quickly as possible
  - **push** incorporates changes into main* (shared repo)
  - **pull request** incorporates changes into main* (shared repo) after they are reviewed
    - Using pull requests is a CSE403 requirement!

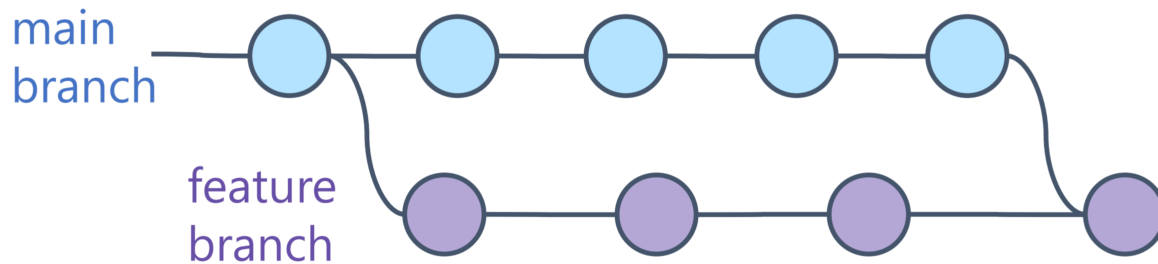


*or another specified branch in the shared repo

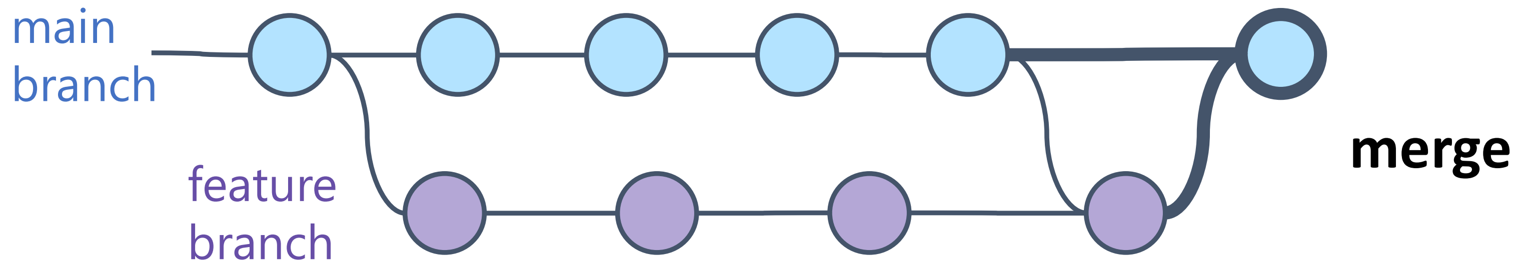
3 ways to resolve a pull request

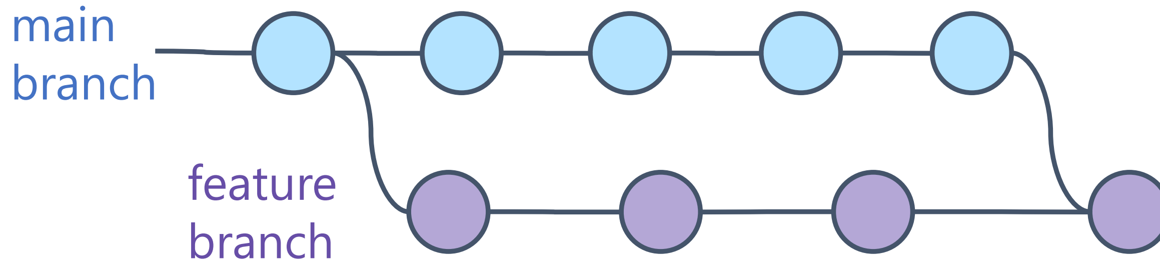


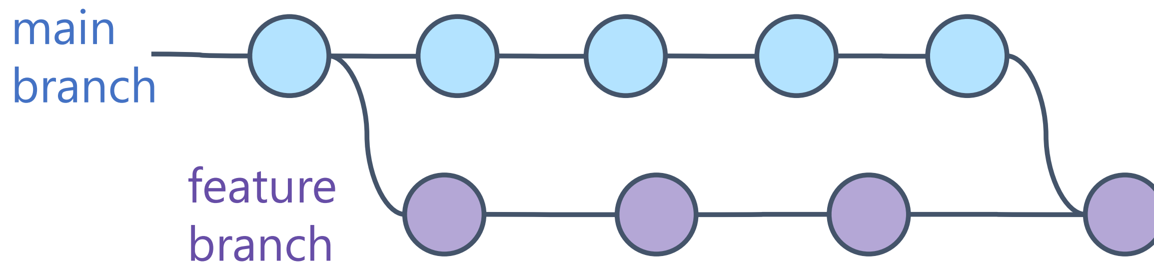




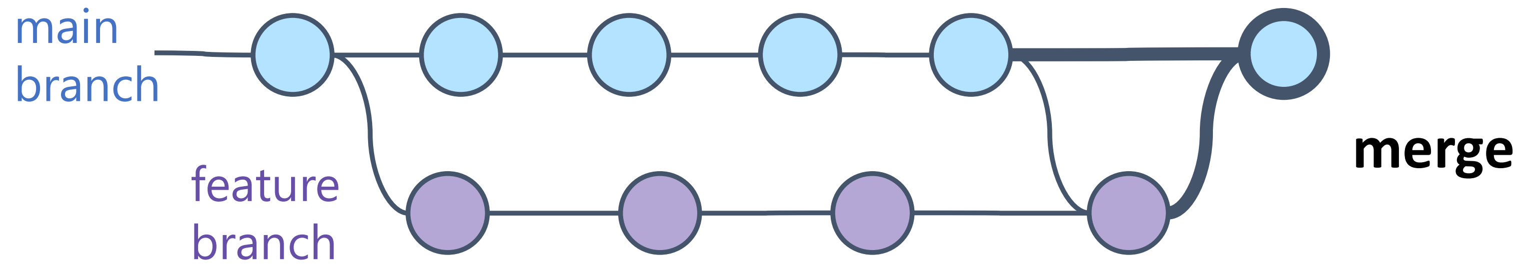
3 ways to resolve a pull request

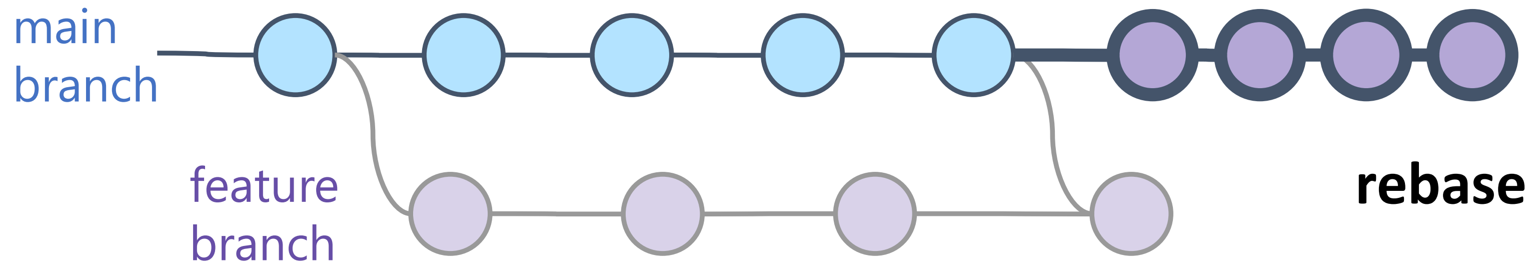


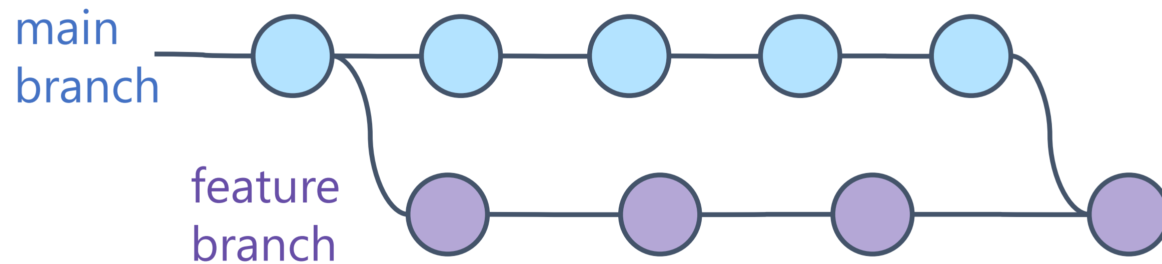




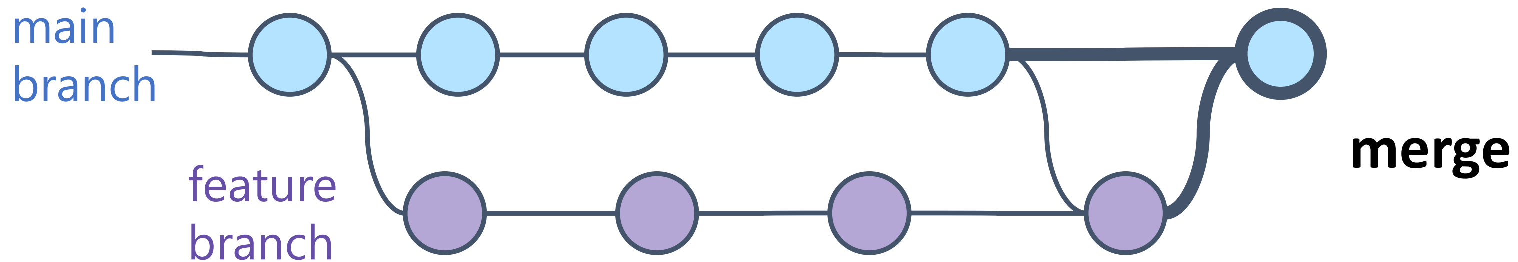
3 ways to resolve a pull request



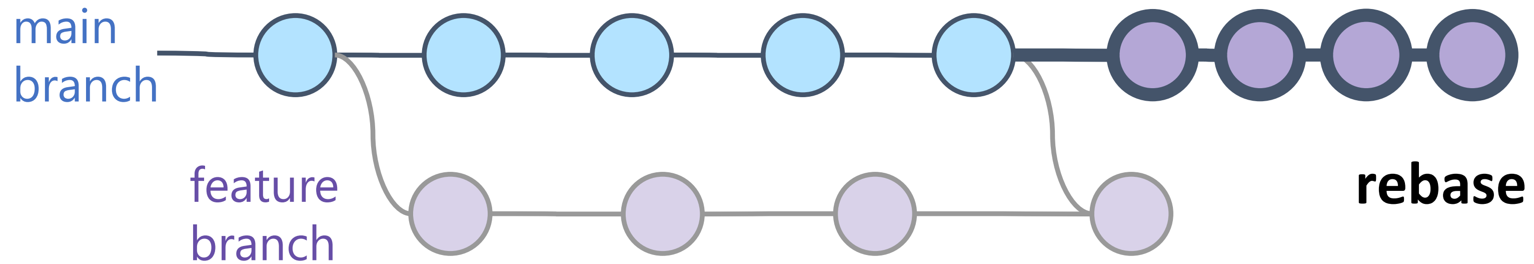




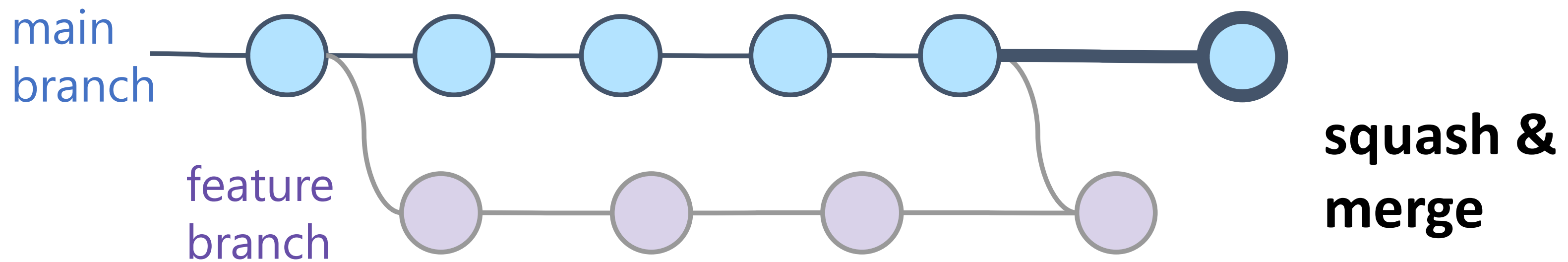
3 ways to resolve a pull request



merge

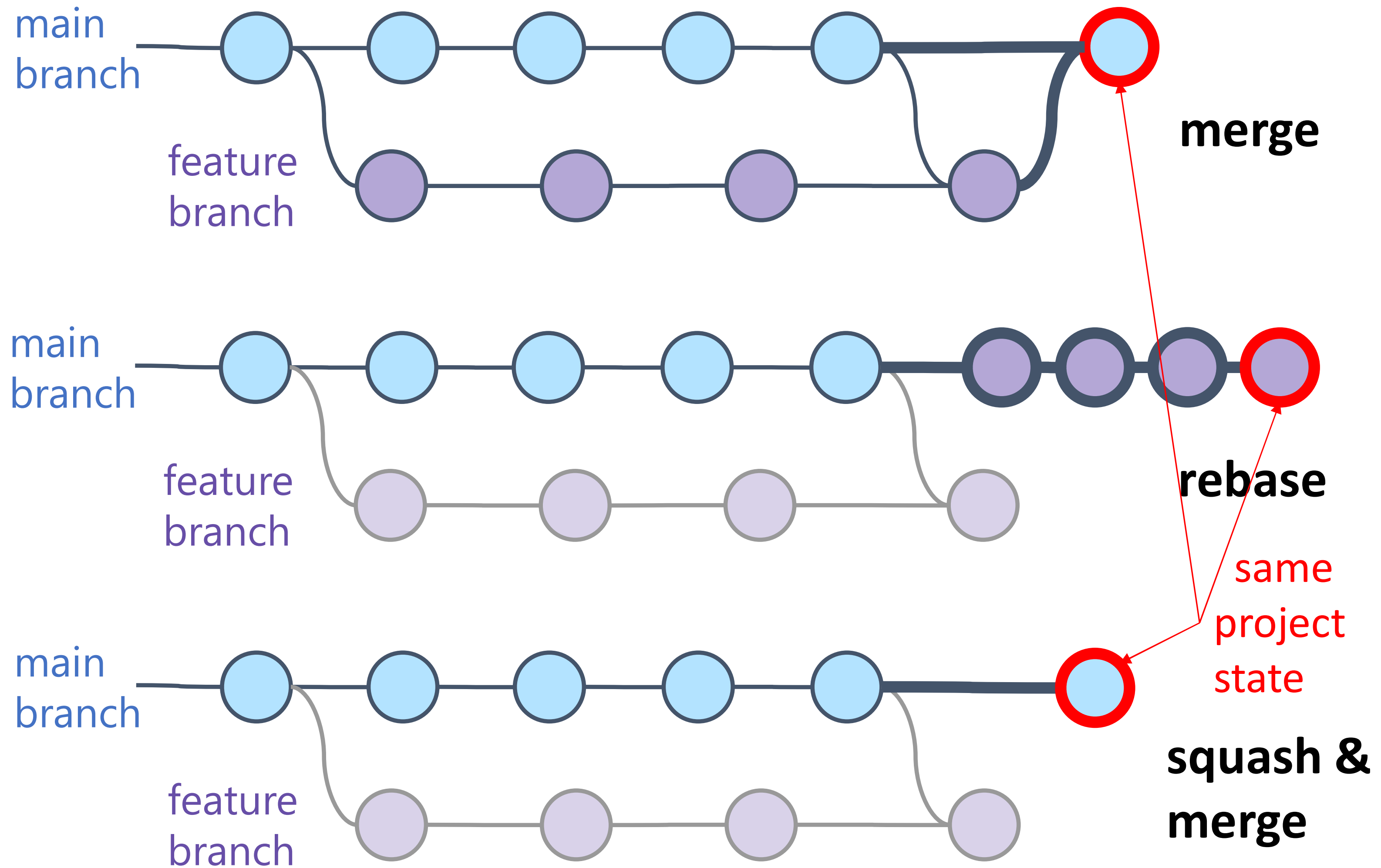


rebase

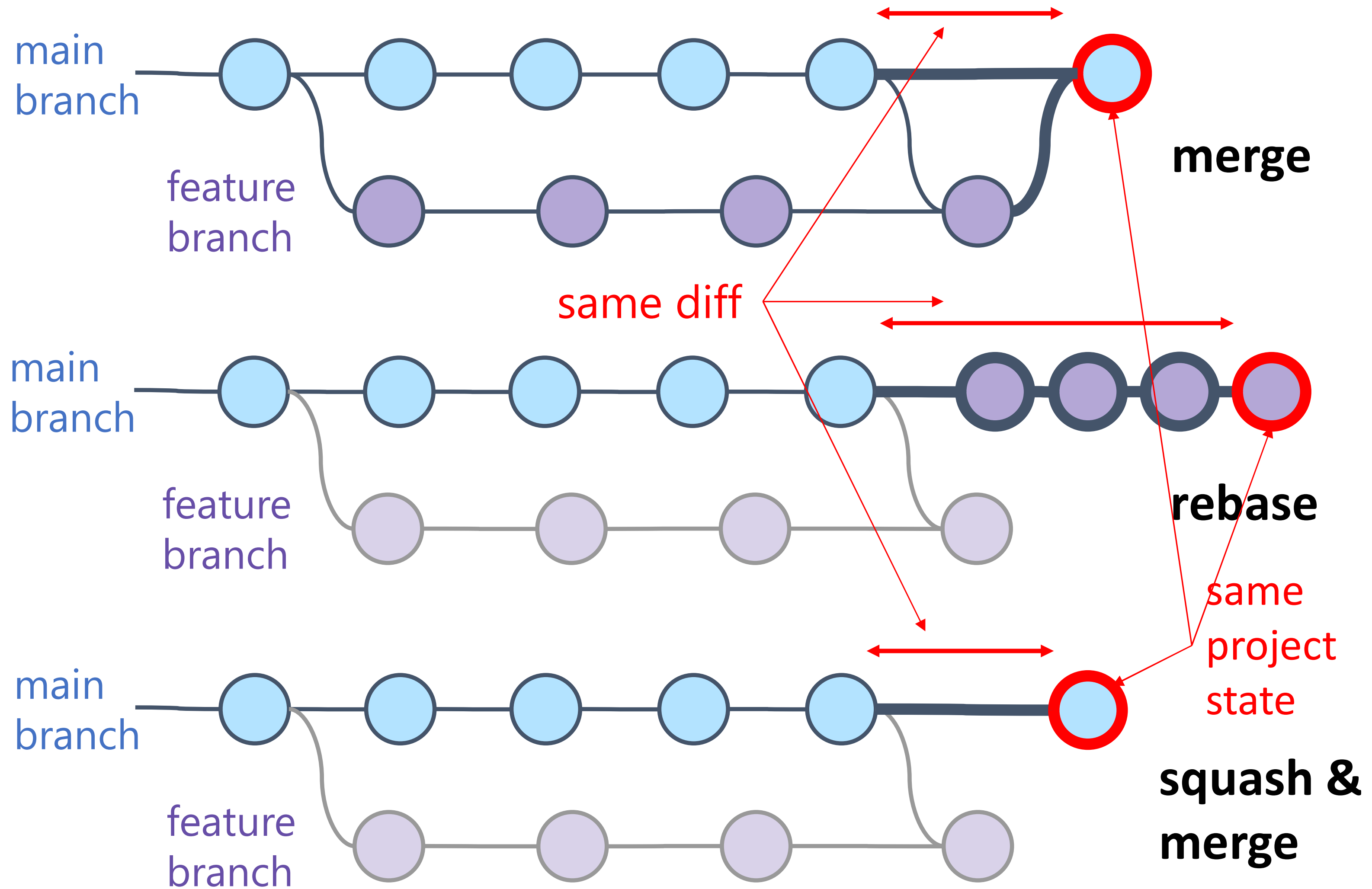


squash &
merge

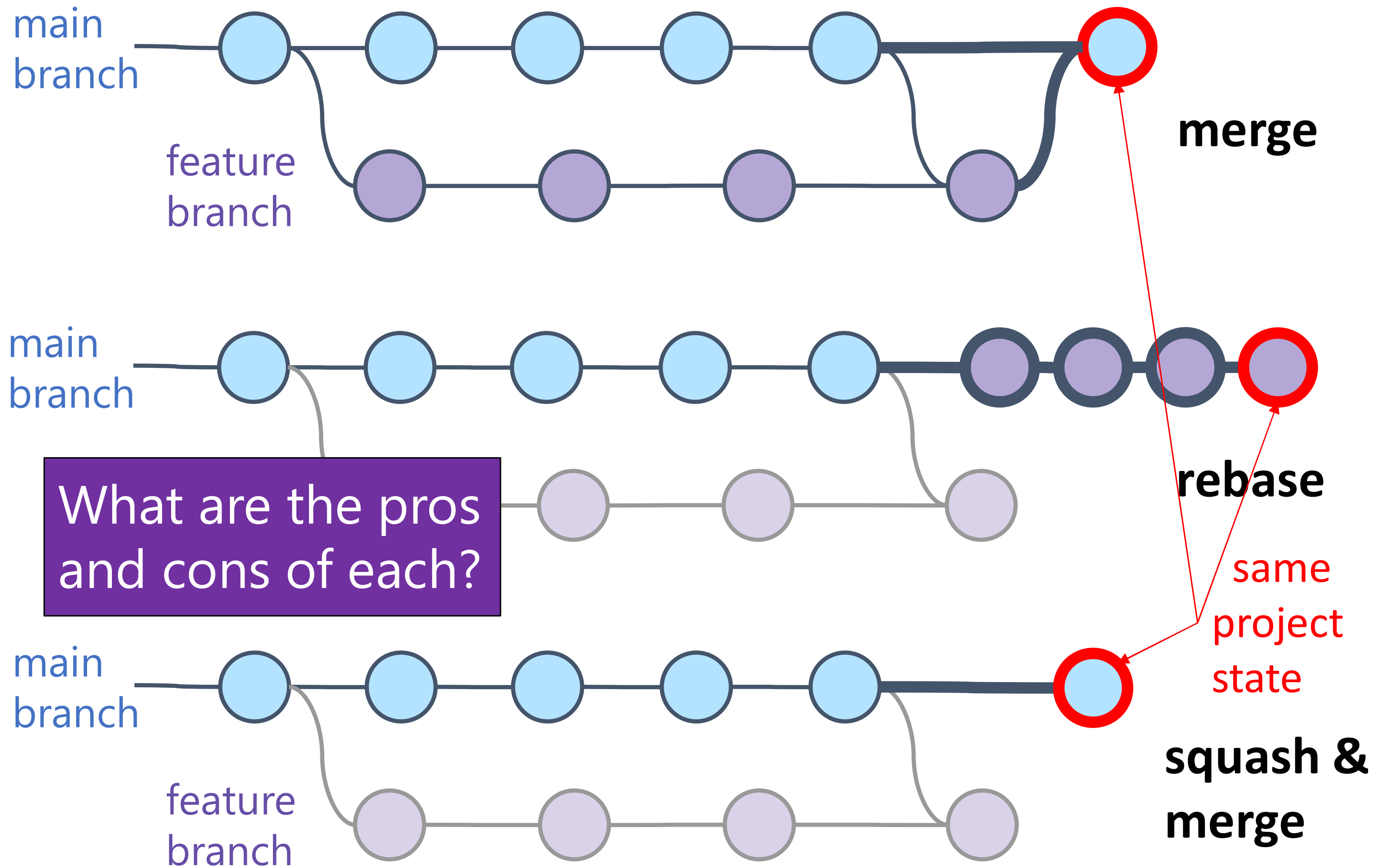
3 ways to resolve a pull request



3 ways to resolve a pull request



3 ways to resolve a pull request



Rebase is a powerful tool, but be careful

- Results in a sequential linear commit history
- Changes the commit history
- Others may be working on copy of original tree - painful for them to sync/merge!

Do not rebase public branches in general
(especially not with a force-push!)

Github has standard options for these useful operations for **pull requests**

Create a merge commit

All commits from this branch will be added to the base branch via a merge commit.

✓ **Squash and merge**

The 14 commits from this branch will be combined into one commit in the base branch.

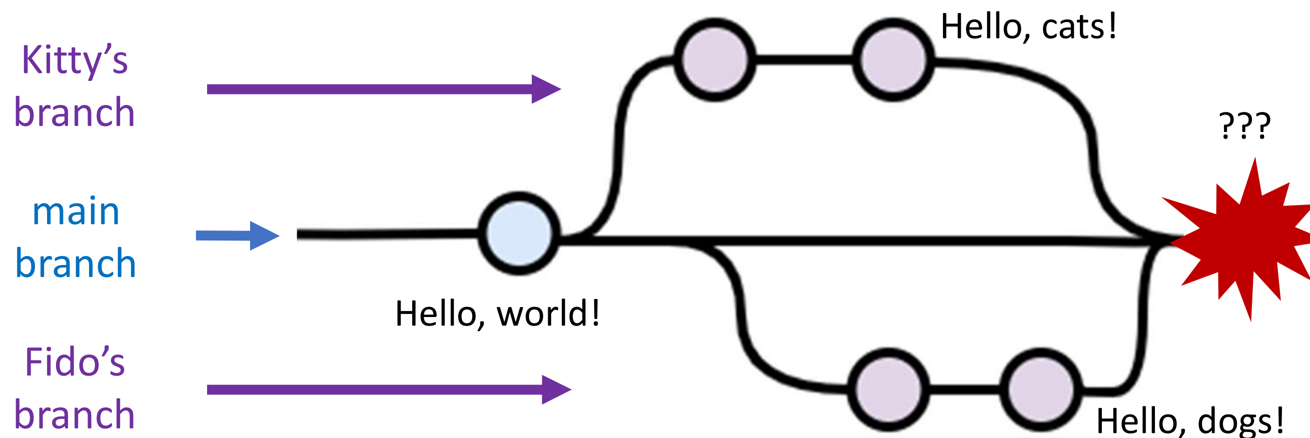
Rebase and merge

The 14 commits from this branch will be rebased and added to the base branch.

Merge conflicts

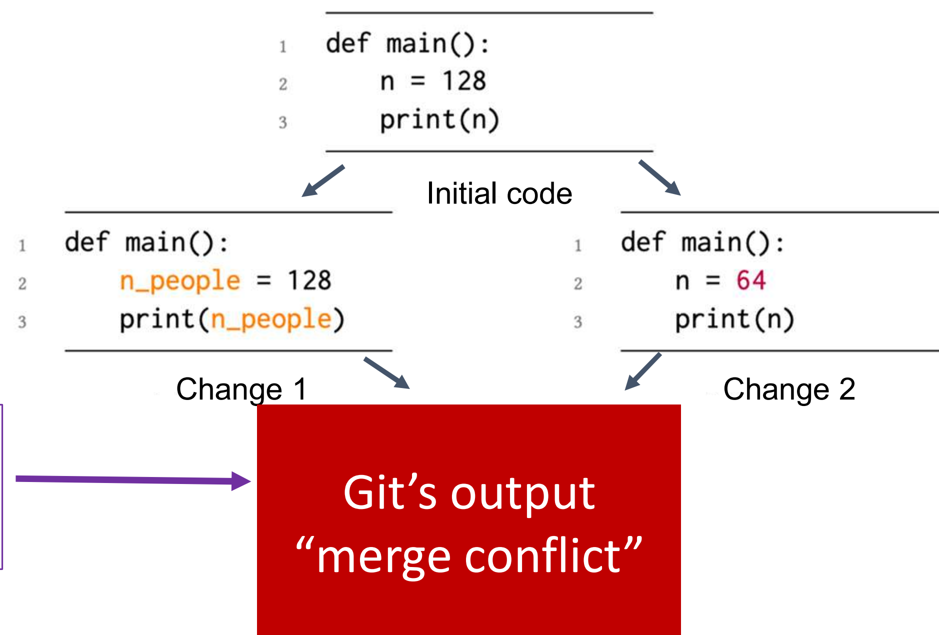
Merge conflicts

- You and a teammate are editting the same file on your own local branches
- You both execute merges to integrate your changes into main
- Git tries to merge the edits for you, retaining edits from both branches
- A **conflict** arises when two users **change the same line** of a file
- The person doing the last merge needs to resolve the conflict by **manual editing**



Merge algorithm: may fail to make a merge

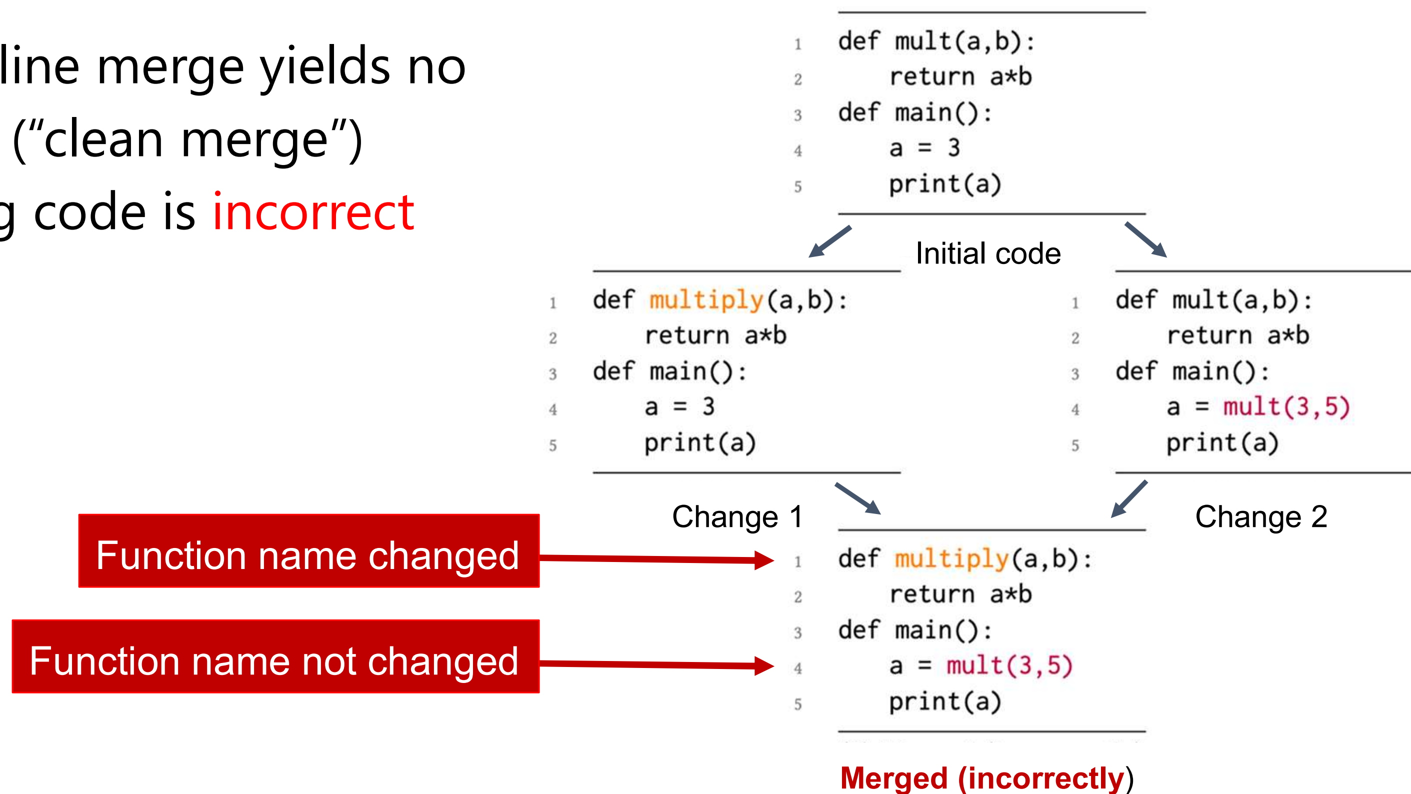
- Line-by-line merge yields a conflict
- Inspection reveals they can be merged



Works despite changes on same line

Merge algorithm: falsely successful merge

- Line-by-line merge yields no conflicts ("clean merge")
- Resulting code is incorrect
- Why?



How to avoid merge conflicts

Synchronize with teammates often

- Pull often
  - Avoid getting behind the main branch
- Push (**via a pull request**) as often as practical
  - Don’t destabilize the main build (don’t break the build!)
  - Use continuous integration (automatic testing on each push, even for branches)
  - Avoid long-lived branches

Commit often

- Every commit should address one concept (be atomic)
- Every concept should be in one commit
- **Tests should always pass before commit**
- Consider squash and merge when appropriate, e.g., bugfix branch that had easily combinable commits

Make single-concern branches and atomic commits

They are easier to understand, review, merge, revert

- Do only one task at a time
  - Commit after each one is completed
- Create a branch for each simultaneous task
  - Easier to share work with teammates
  - Single-concern branch ⇒ Atomic commit on main
  - Requires a bit of bookkeeping to keep track of them all; don't overdo it
- Do multiple tasks in one working copy with multiple branches
  - Commit only specific files, or only specific parts of files (use Git's "staging area" with git add; can interactively choose parts of files)

Do not commit all files

Use a .gitignore file

Don’t commit:

- Binary files
- Log files
- Generated files
- Temporary files

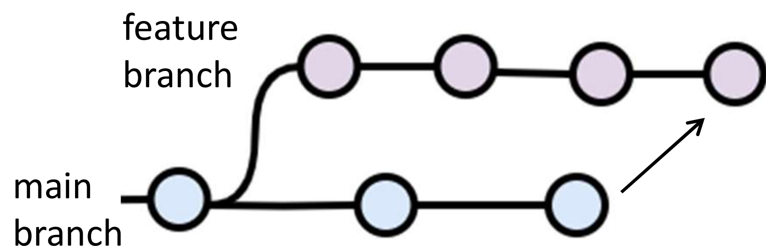
Committing would waste space and lead to merge conflicts

Plan ahead to avoid merge conflicts

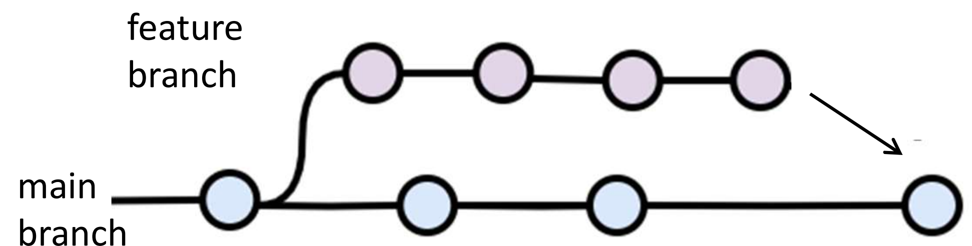
- Modularize your work
  - Divide work so that individuals or subteams “own” parts of the code
  - Other team members only need to understand its specification
  - Requires good documentation and testing
- Communicate about changes that may conflict
  - Examples (rare!): reformat whole codebase, move directories, rename fundamental data structures
- Slow merges down – add some order - to updates to main if main is getting unstable

Tip: lived best practice when merging

1. Integrate changes from main to your branch to make sure no intermediate changes in main have broken your code
2. Merge your branch to main (via a pull request)
3. Not perfect but decreases risk of breaking the build



then



Questions?

Additional material

Some resources

Git concepts and commands (cheatsheets):

- https://training.github.com/downloads/github-git-cheat-sheet/
- https://wac-cdn.atlassian.com/dam/jcr:e7e22f25-bba2-4ef1-a197-53f46b6df4a5/SWTM-2088_Atlassian-Git-Cheatsheet.pdf?cdnVersion=1272

Github concepts and flows:

- https://githubtraining.github.io/training-manual
- https://www.atlassian.com/git/tutorials/

Install

GitHub Desktop

desktop.github.com

Git for All Platforms

git-scm.com

Configure tooling

Configure user information for all local repositories

`$ git config --global user.name "[name]"`

Sets the name you want attached to your commit transactions

`$ git config --global user.email "[email address]"`

Sets the email you want attached to your commit transactions

`$ git config --global color.ui auto`

Enables helpful colorization of command line output

Branches

Branches are an important part of working with Git. Any commits you make will be made on the branch you're currently "checked out" to. Use `git status` to see which branch that is.

Create repositories

A new repository can either be cre
existing repository can be cloned.
initialized locally, you have to pusl
afterwards.

`$ git init`

The git init command turns an exis
new Git repository inside the folde
command. After using the `git in`
local repository to an empty GitHu
following command:

`$ git remote add origin [url]`

Specifies the remote repository fo
The url points to a repository on G

`$ git clone [url]`

Clone (download) a repository tha
GitHub, including all of the files, b

The .gitignore file

Sometimes it may be a good idea
being tracked with Git. This is typi
file named `.gitignore`. You can fi
for `.gitignore` files at github.com

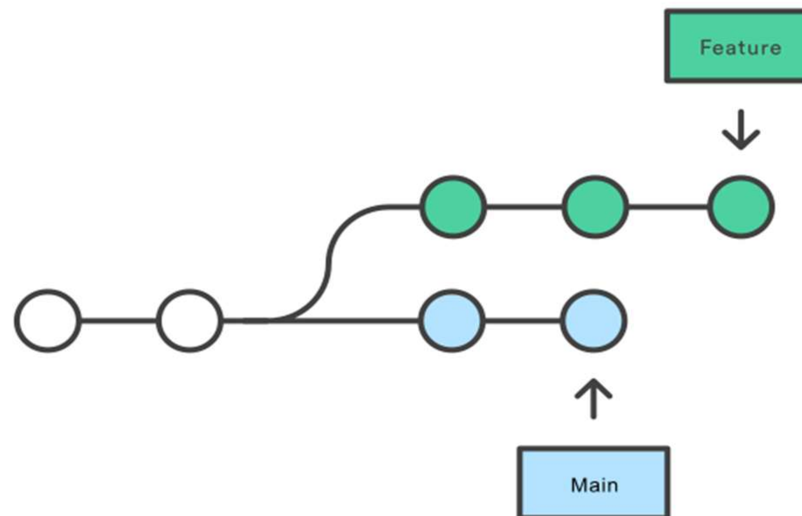
Synchronize changes

More Git vocab 😎

- **index**: staging area (located .git/index)
- **content**: git tracks **a collection of file content, not the file itself**
- **tree**: git's representation of a file system
- **working tree**: tree representing the local working copy
- **staged**: ready to be committed
- **commit**: a snapshot of the working tree (a database entry)
- **ref**: pointer to a commit object
- **branch**: just a (special) ref; semantically: represents a line of dev
- **HEAD**: a ref pointing to the working tree

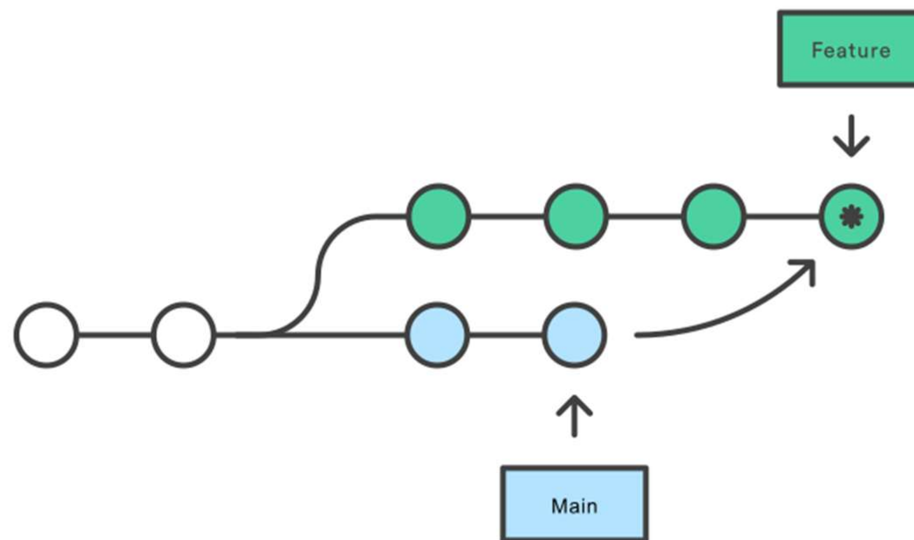
More on rebase

Merge vs Rebase



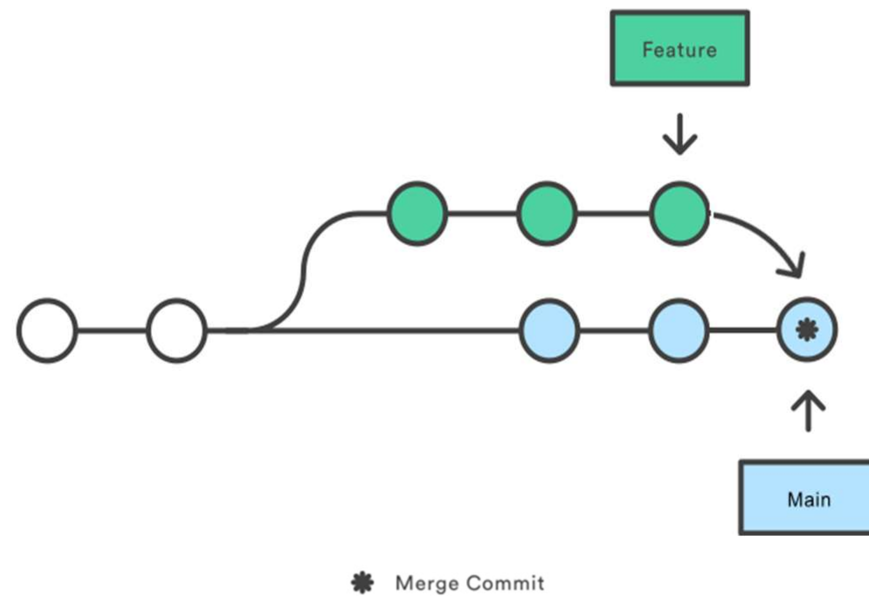
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Merge (integrating changes **from main**)



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

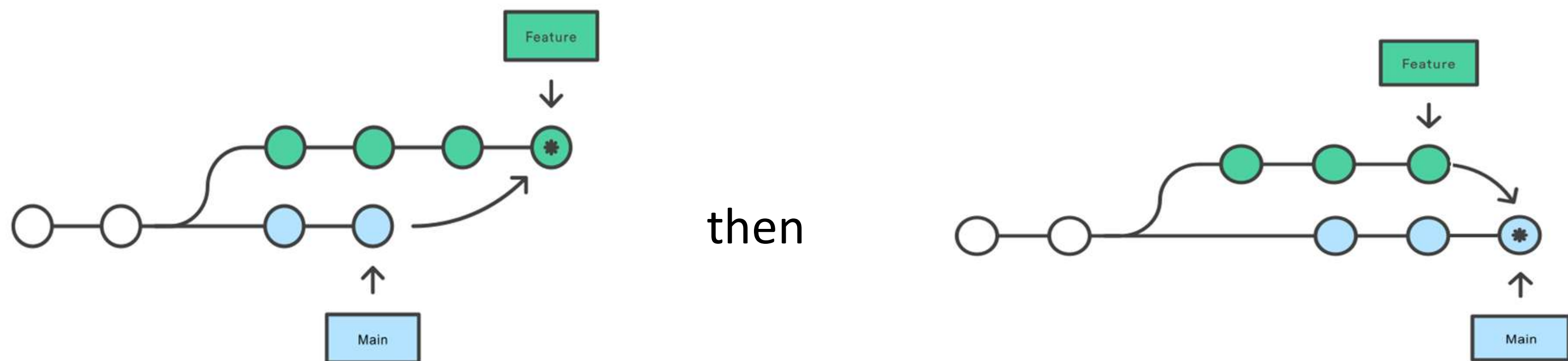
Merge (integrating changes **into main**)



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

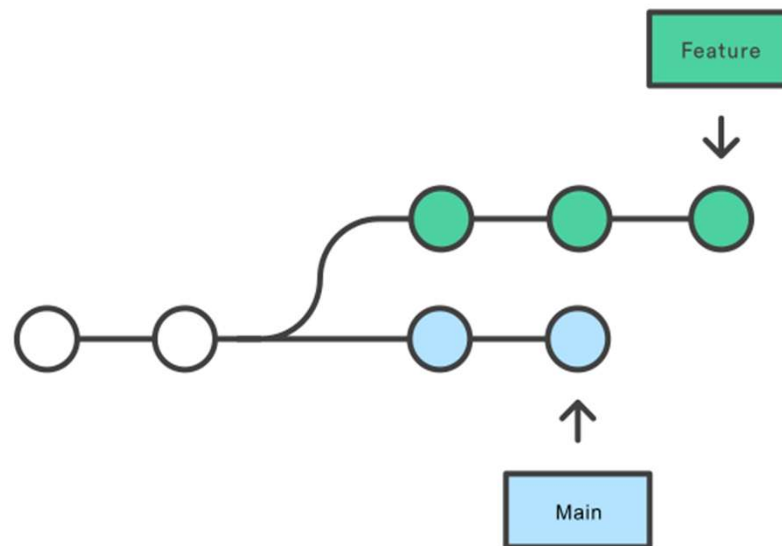
Merge (best practices **do both**)

1. Integrate changes from Main to your branch to make sure no intermediate changes in Main have broken your code
2. Merge your branch to Main
3. Not perfect but decreases risk of breaking the build



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

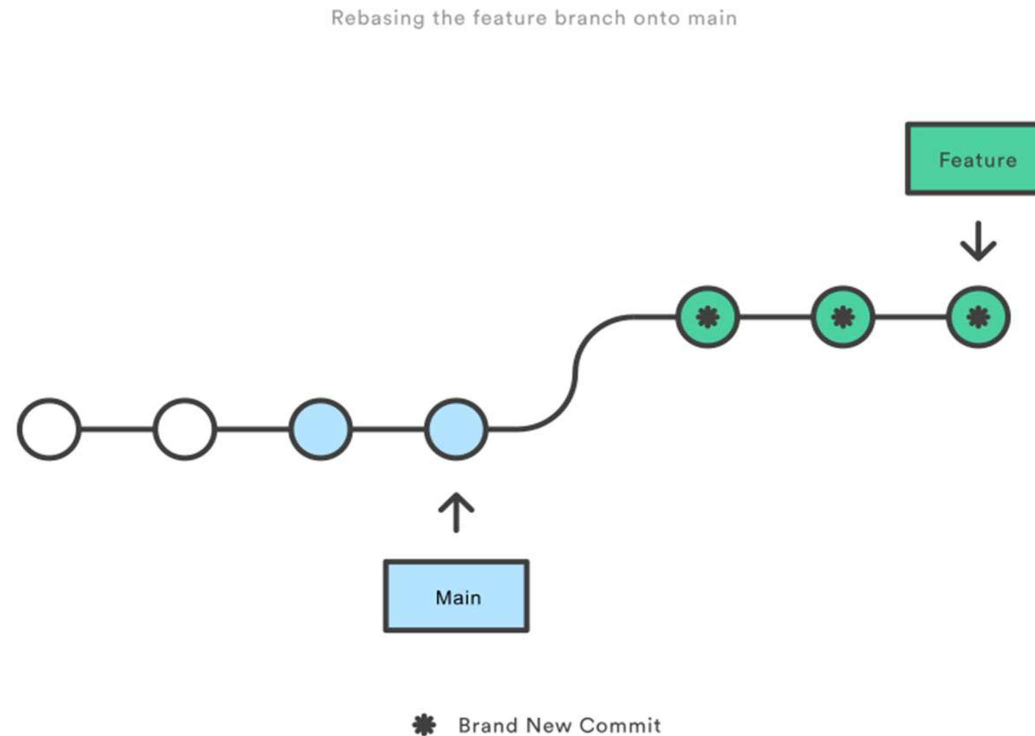
Merge vs **Rebase**



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Merge vs **Rebase**

- Rebase moves the entire feature branch to begin at the tip of the main branch
- It re-writes history by creating new commits, now in the main branch



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

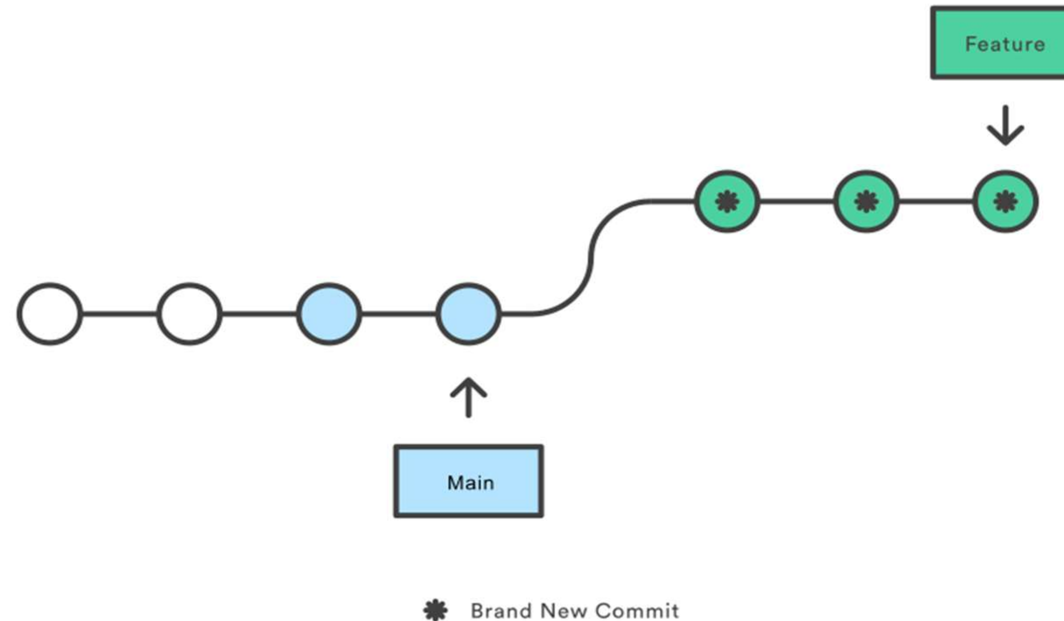
Merge vs **Rebase – why rebase?**

What's a benefit of rebase?

- Clean linear history
- Easier debugging

What's a risk?

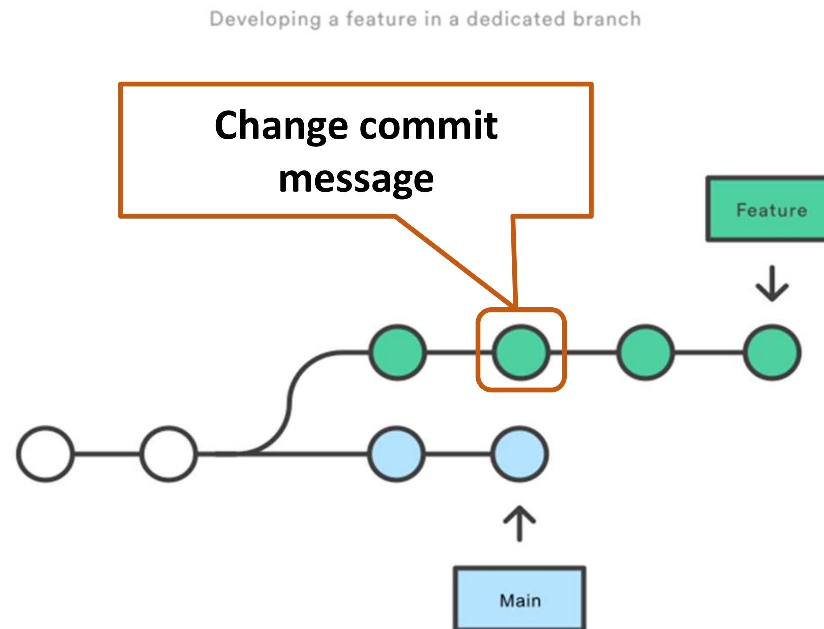
- Losing some commit history
- Others may be working on copy of original tree - painful for them to sync/merge!



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (use to **rewrite** commits)

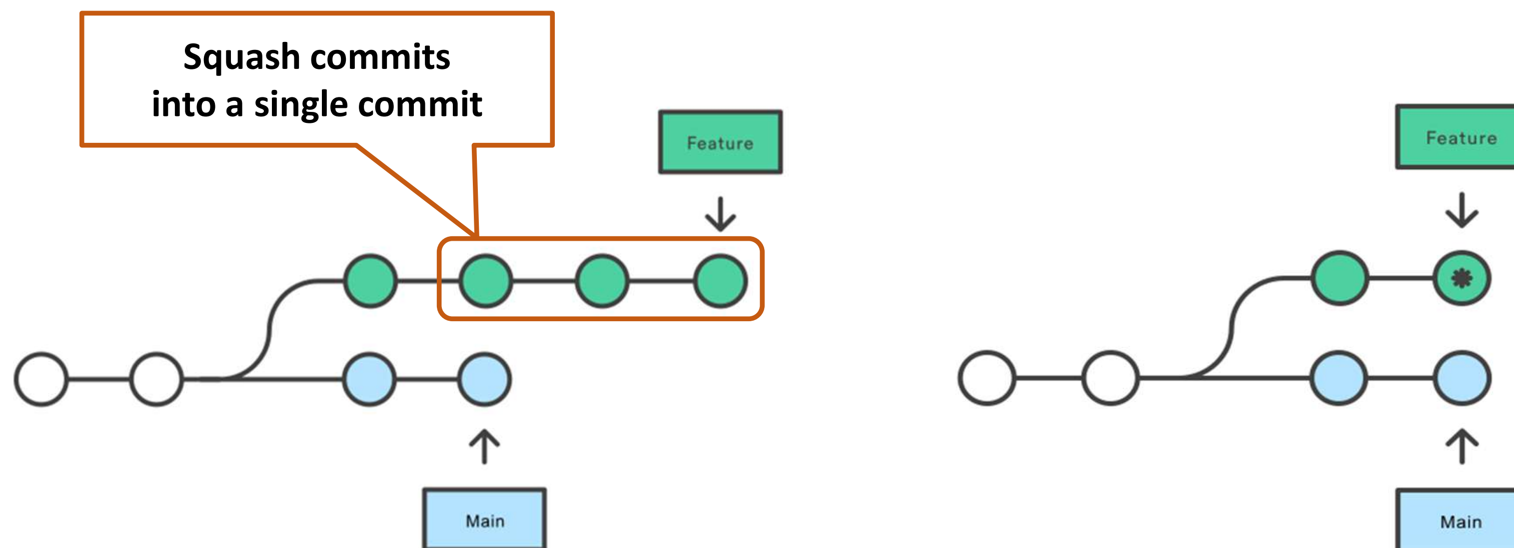
- Can rewrite commits as they move to the main branch



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

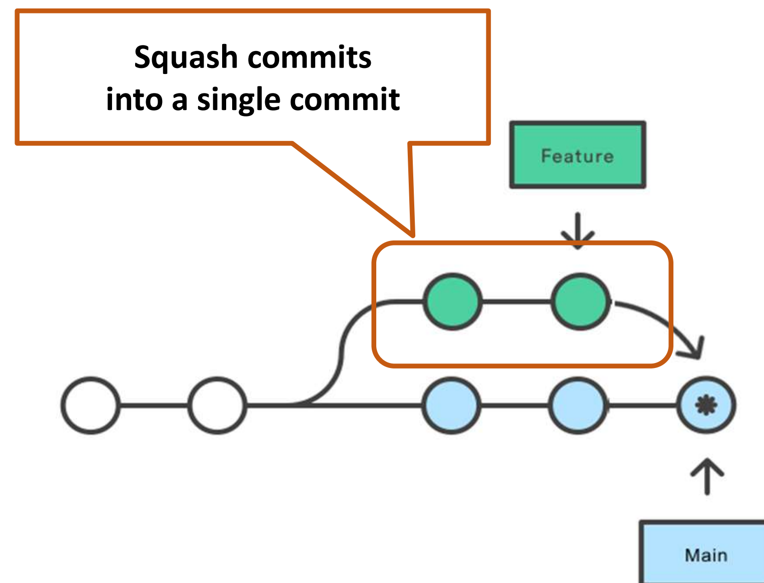
Interactive Rebase (use to **squash**)

- Squash combines commits



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (**squash and merge**)



- Can combine commits before a merge, too!
- Not uncommon to do

https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Rebase: a powerful tool, but ...