git merging !

Branching and Merging in `git`

- Branches enable safe exploration of ideas in a code base
  - Workflow:
    1. Checkout a new branch
    2. Make commits
    3. Was the branch a good idea?
       - Yes: Merge it back into the **mainline** branch (git's idiomatic mainline branch is **master**)
       - No: Discard the branch and go back to step #1

- In this lesson, we'll mechanisms for merging:
  1. Fast-forward
  2. Merge commits

- Merging branches with conflicting changes requires a resolution.

Workflows

- Different teams will have different workflows for organizing repositories

- Rules will inform:
  - When and why should you establish a branch?
  - When are you allowed to merge a branch back in?
  - Do you merge with fast-forwarding or not?
  - How do you *release* a version of a project?
  - How do you *catch up* a repository?

- Often in organizations these rules will be influenced by:
  - Are you passing all tests?
  - Have you done a code review?
  - Will your branch merge cleanly?

- Today we'll explore the most important skill in these workflows: merging.
  - For the full story on how large teams operate, read more here:
  - https://nvie.com/posts/a-successful-git-branching-model/

Cloning a `git` Repository with Branches

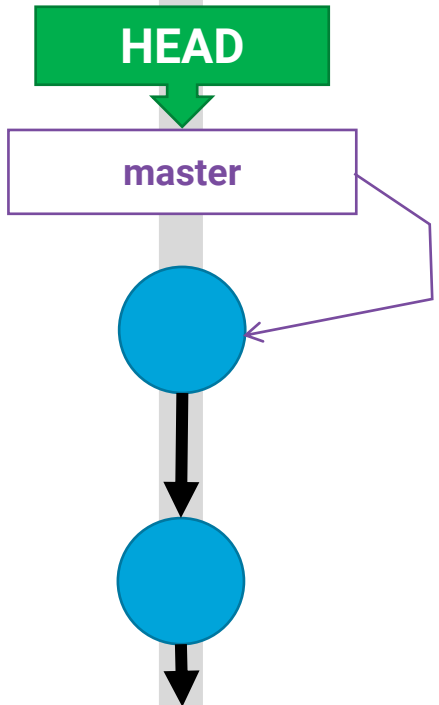
```
learncli$ git clone https://github.com/comp211/git-workflow.git

learncli$ cd git-workflow

# List all branches (including branches only established on remotes)

learncli$ git branch --all

# Notice many branches on the default remote named origin (the GitHub remote)

learncli$ git log --graph --oneline --all
```

Today's Project Overview

- Simple C project
  - main.c – Command-line interface number guessing game
  - Makefile – Build configuration file
  - README.md – Simple readme file for the project
  - .gitignore – git configuration file

- `make` and `Makefile`
  - The program **make** reads the **Makefile** which tells it how to build the project
  - The default rule builds an executable named **app** using the gcc compiler (**make**)
  - The **clean** target deletes the built files (**make clean**)

- Try building the program and running it:
  - **`make`**
  - **`./app`**



Branches

- Commonly you'll use feature branches
  - Working on a new feature? Establish a branch!

- To create a branch, in two steps:

```
git branch feature-challenge
git checkout feature-challenge
```

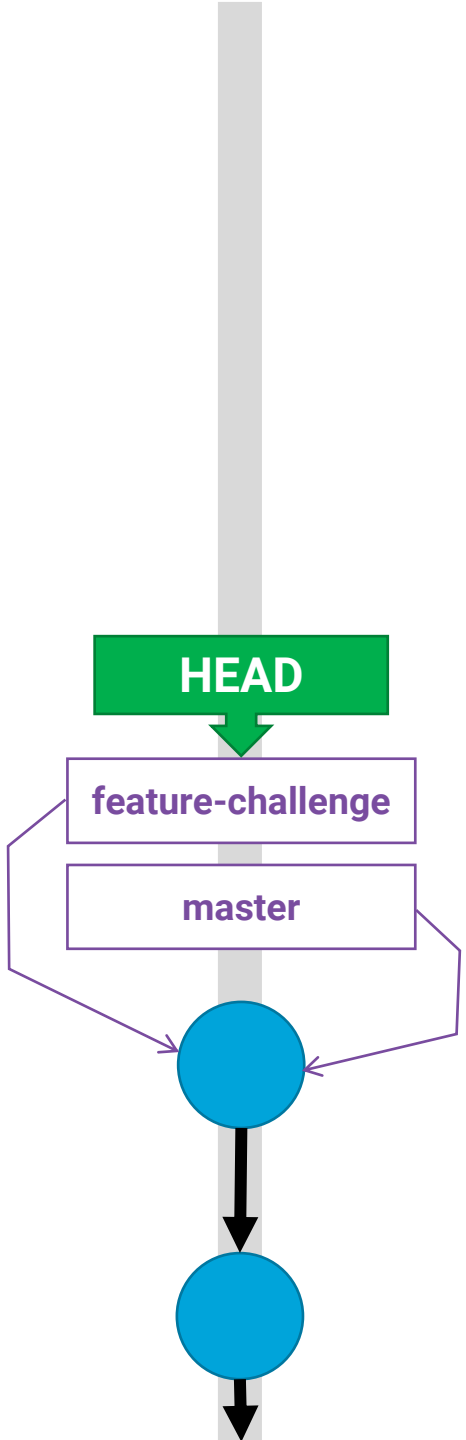
- These steps are usually combined:

```
git checkout -b feature-challenge
```

- Confirm branch checked out:

```
git branch
```

- Remember, a branch is just a pointer to a commit. So a new branch doesn't diverge until commit(s) are made.



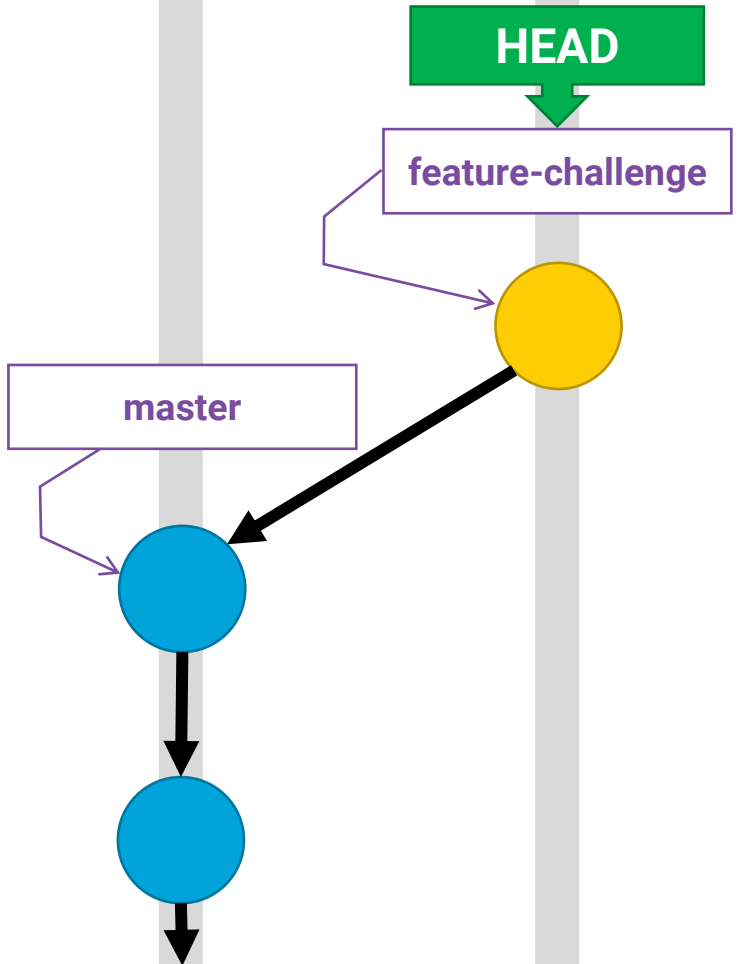
Hands-on: Branching and Committing

1. Change **main.c** such that you are guessing between 0 and 511, rebuild with make, test.

2. Add your changes to **main.c** to staging

3. Make a commit with the message:

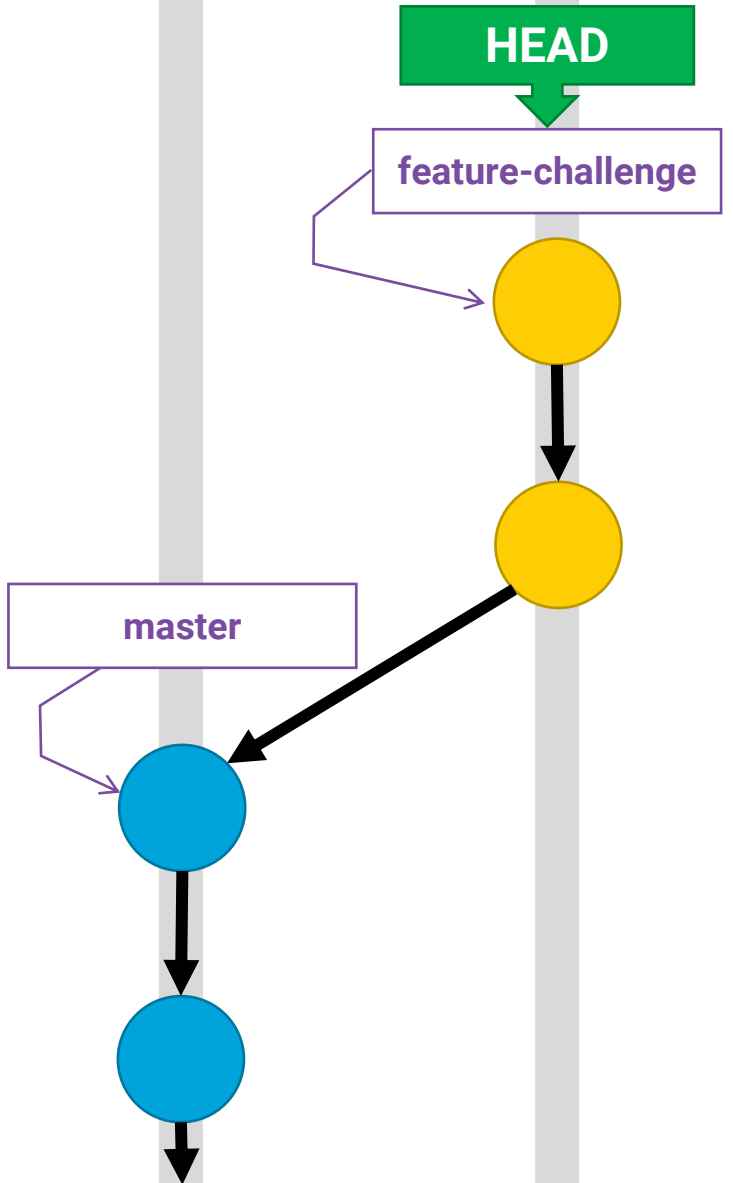
   ```
   Increase the range from 0-511
   ```



Branches: Follow-along

1. In main.c, print a message on a new line with some greeting like "Welcome to my awesome game!" before "I'm thinking of a number..."

2. Add changes and make another commit with a message of your choice.



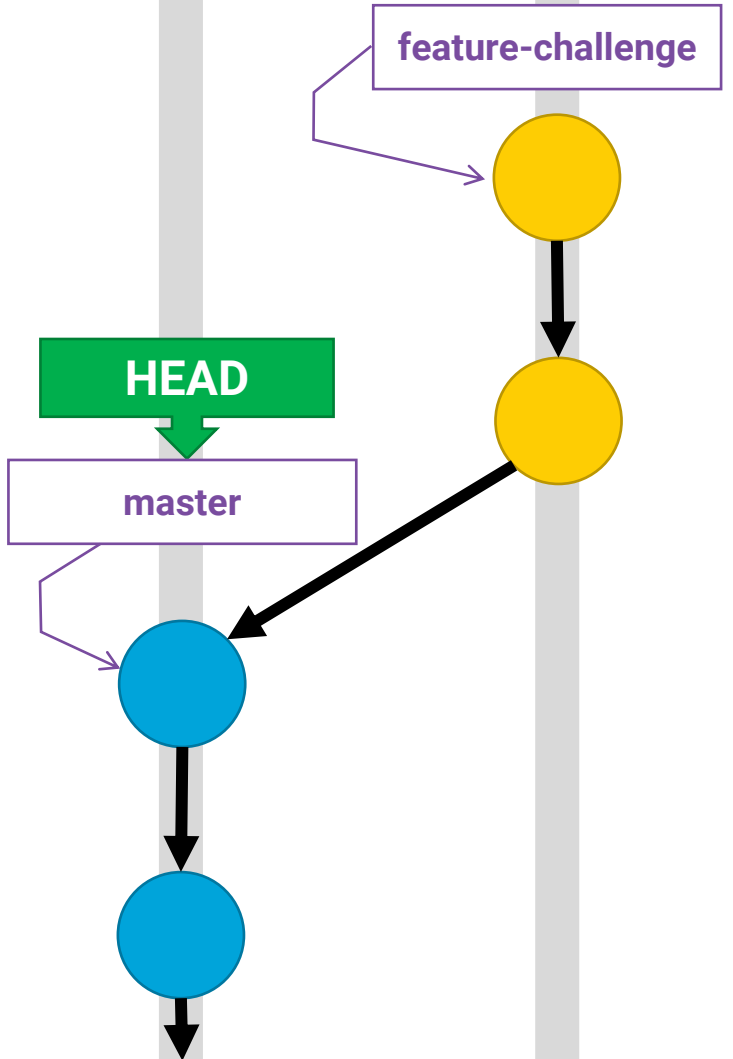
Branches: Changing Branches

- To change to a different branch, check it out:

```
git checkout master
```

- This changes the files in your working directory to exactly match their contents of the branch (commit) checked out.

- If there were uncommitted changes you would risk losing in this process, git requires you to deal with them first.

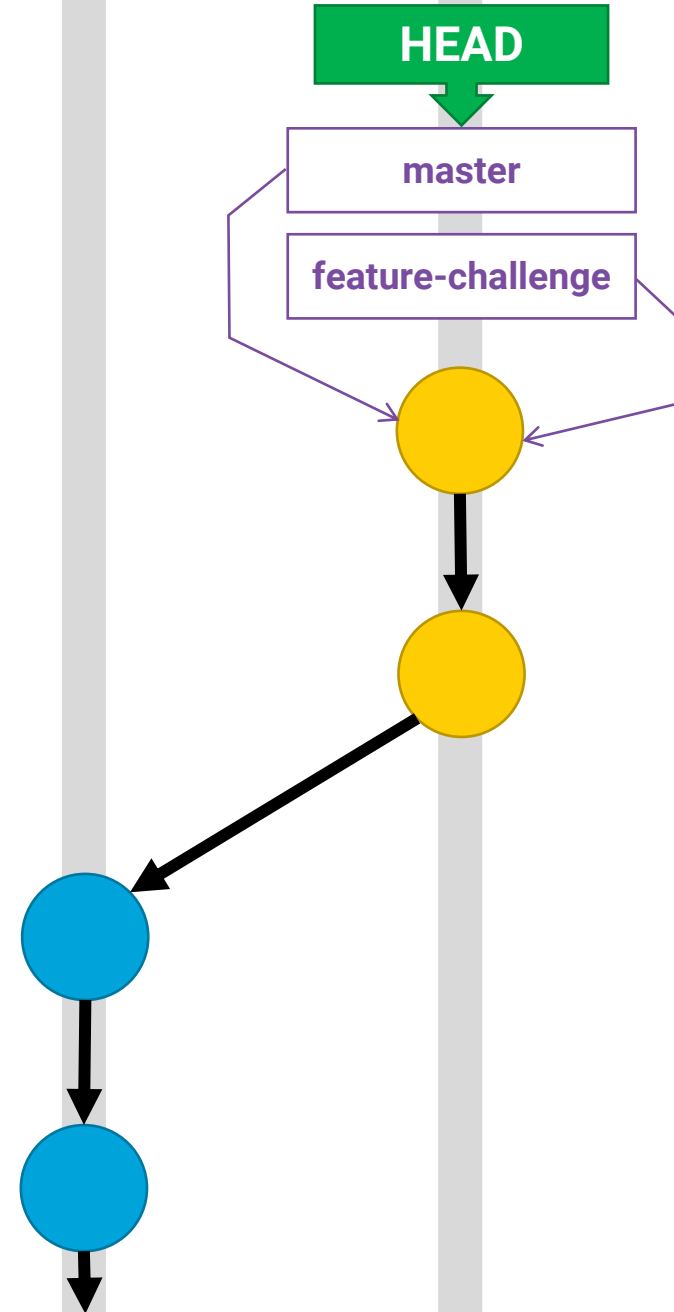


Fast-forward Merges

- As shown in the previous lecture, if we were to merge *feature-challenge* and *master*, by default we would have a single, unified history.
  - It would appear as though there was never a branch!

- This type of merge is called a fast-forward merge because all that really happens is the *master* branch gets "fast-forwarded" to refer to the same, later commit as *feature-challenge*.

- The downside to a fast-forward commit is you lose the sense of *which* commits were made in the feature branch.



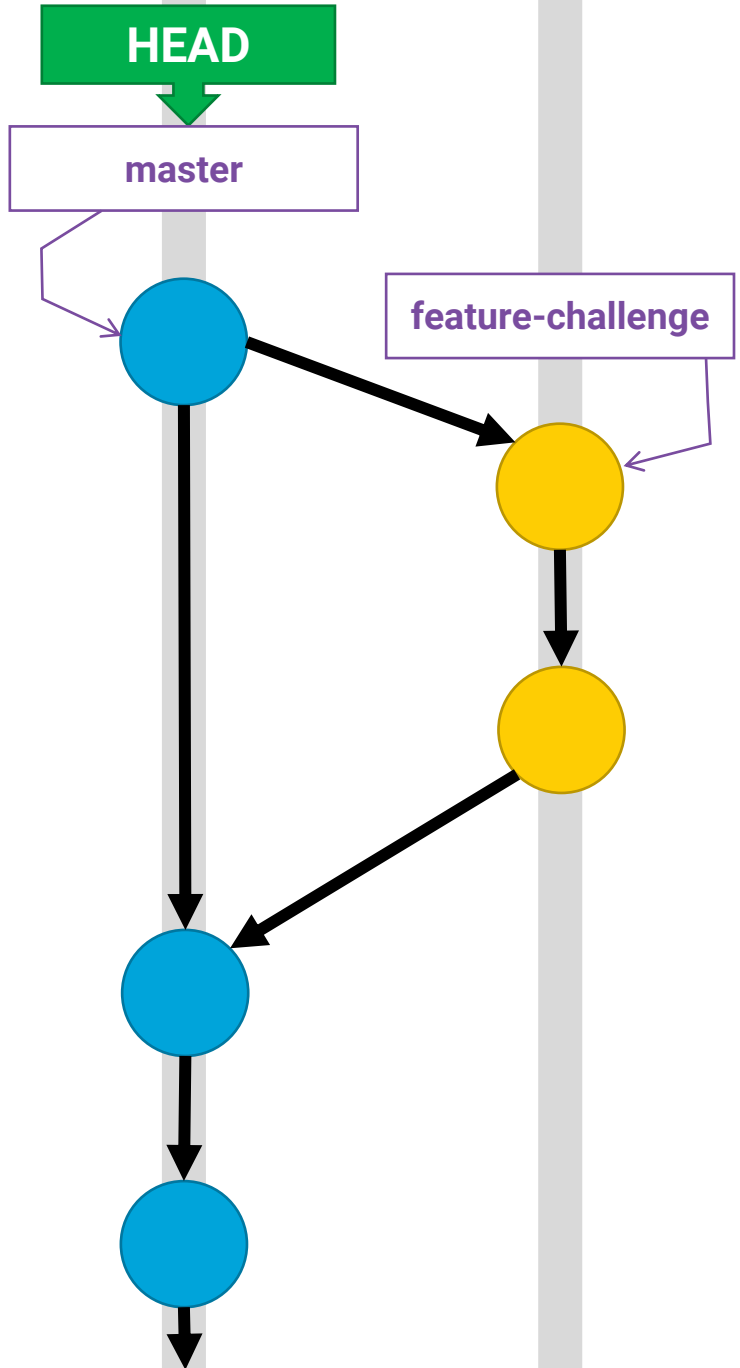
Merge Commits

- When merging feature branches many consider it a best practice to establish a "merge commit"
  - In doing so, the feature branch's commits are kept *separate* from the master branch's.
  - This retains the history of the branch.

- To merge, checkout the branch you're merging into, then:

```
git merge --no-ff feature-challenge
```

- The --no-ff flag is short for no fast-forward.

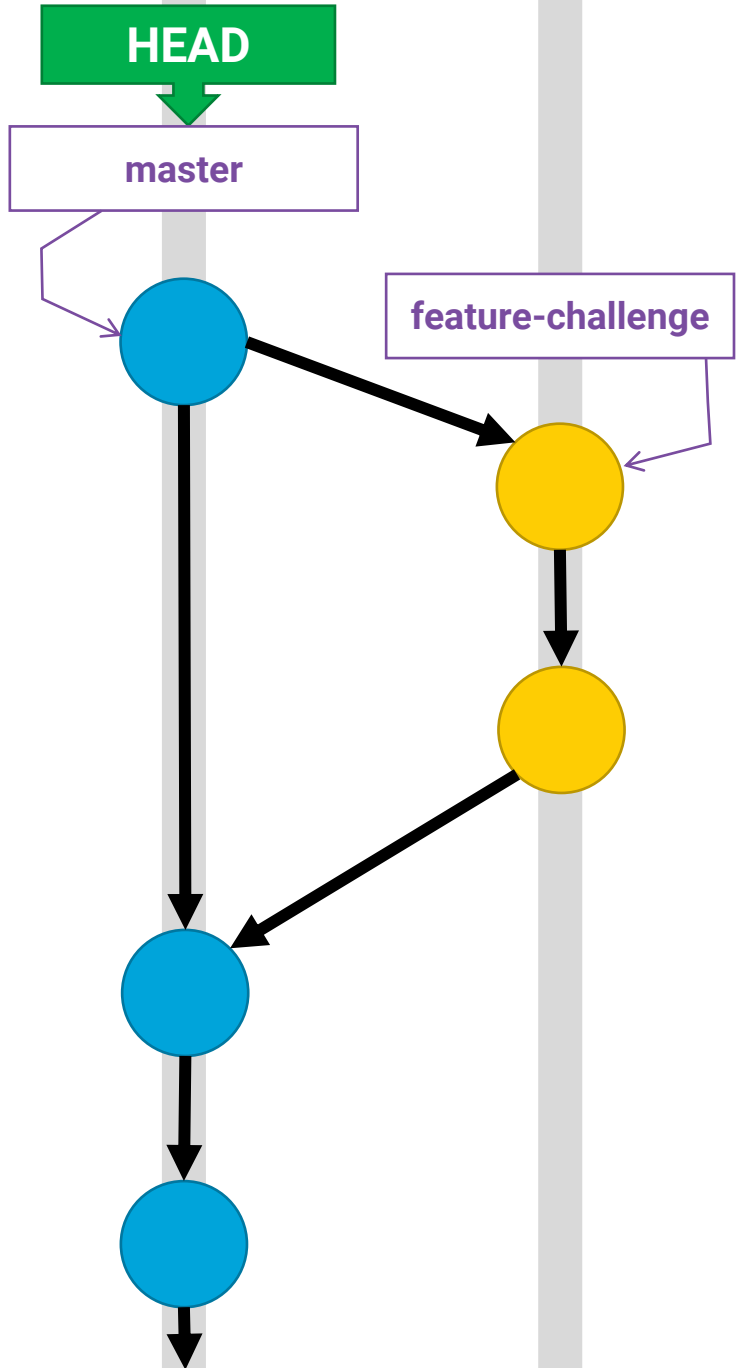


Merge Commits

- Notice a merge commit has two parents!

- Even though git has "branches" commit histories are not usually trees, they're graphs.

- More specifically, git repositories are
  - *directed*
    - each commit points to parent(s)
    - parents do not have references to children
  - *acyclic*
    - you cannot create a self-referential or cyclical history
    - there is a path from the current commit back to the start of the project
  - *graphs*



Checking out a remote branch

- View remote branches with: **`git branch --all`**
  - Has someone pushed a new branch to a remote repo? You'll need to fetch them.
  - The subcommand **git fetch --all** will fetch branches from all remotes.

- To work with a remote branch, there is a specific means for checking it out:
  - If you see a branch such as **`remotes/origin/feature-count`**
  - You can check it out with: **`git checkout --track origin/feature-count`**

- You will see a message indicating a new branch named feature-count is setup to track the remote branch of the same name from remote *origin*.

- Now your repo will match the state of the remote repo's same branch.
  - With write access, you can push and pull to this branch on the remote repository now, as well.

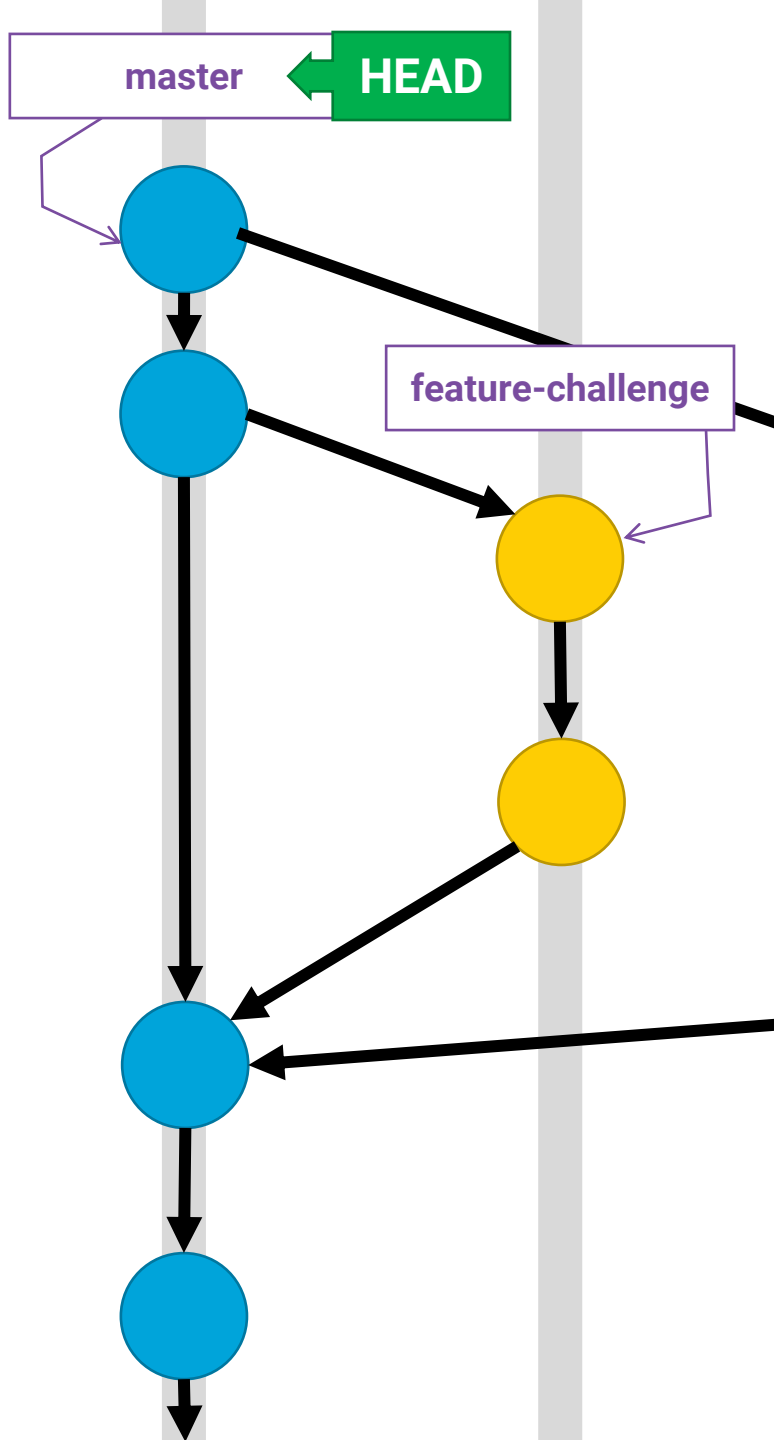
Merge in another branch...

- Compare changes with master branch:
  - git diff master
- Switch to master branch
  - git checkout master
- Merge in feature-count branch with a merge commit (this would happen no matter what)
  - git merge --no-ff feature-count
- Notice the merge succeeded even though *feature-count* was branched from an earlier state of *master*
  - Big idea: This is because commits track changes at the *line* level and the *lines changed* did not overlap or conflict between *feature-count* and *feature-challenge*
- View commit graph history:
  - git log --oneline --graph --all

Conflicts

- What happens when two branches have modify overlapping line segments of a file and you attempt to merge the branches?

- A conflict!



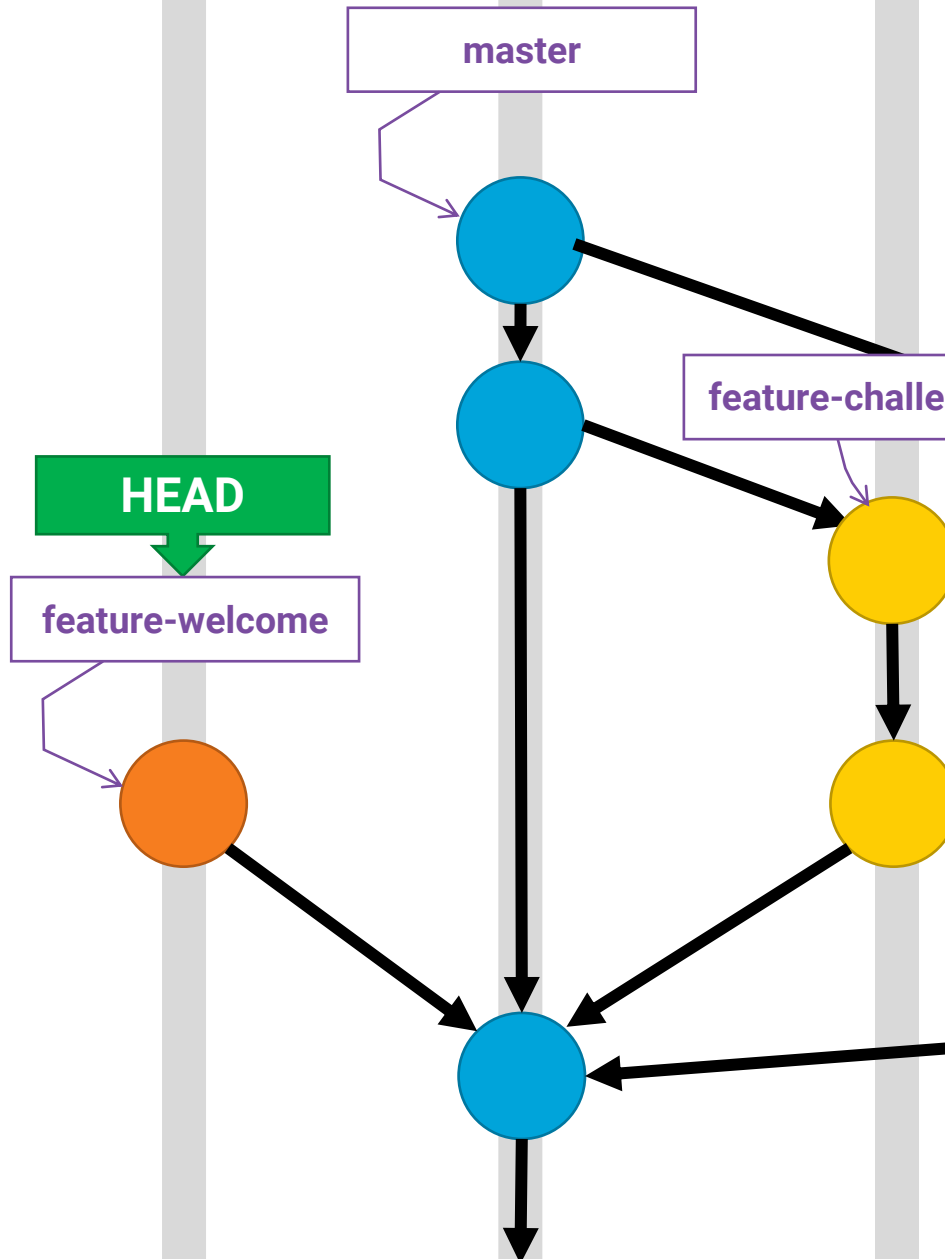
Merging with conflicts (1/3)

- Let's checkout another feature branch

```
git checkout \
   --track \
   origin/feature-welcome
```

- Compare with *master* to see what's changed:

```
git diff master
```

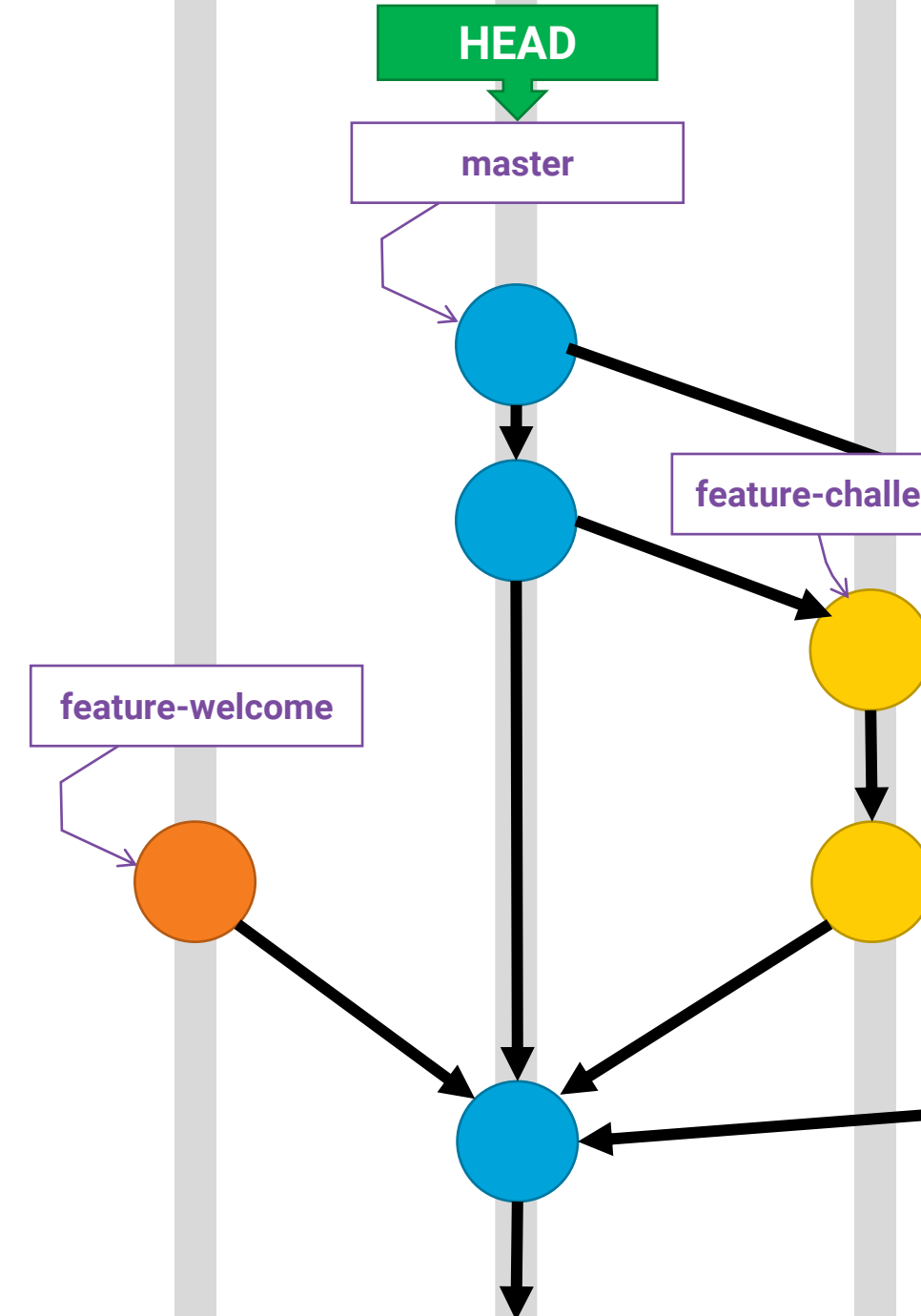


Merging with conflicts (2/3)

- Let's switch back over to master and merge
  ```
  git checkout master
  git merge --no-ff feature-welcome
  ```

- Uh oh...
  - Auto-merging main.c
  - CONFLICT (content): Merge conflict in main.c
  - Automatic merge failed; fix conflicts and then commit the result.

- To see which files conflict, check status:
  ```
  git status
  ```



Merging with Conflicts (3 / 3)

- Two options:
  1. Abort Merge - **`git merge --abort`**
  2. Fix Conflict and Make Commit

- Opening main.c, you'll see the conflicting lines:

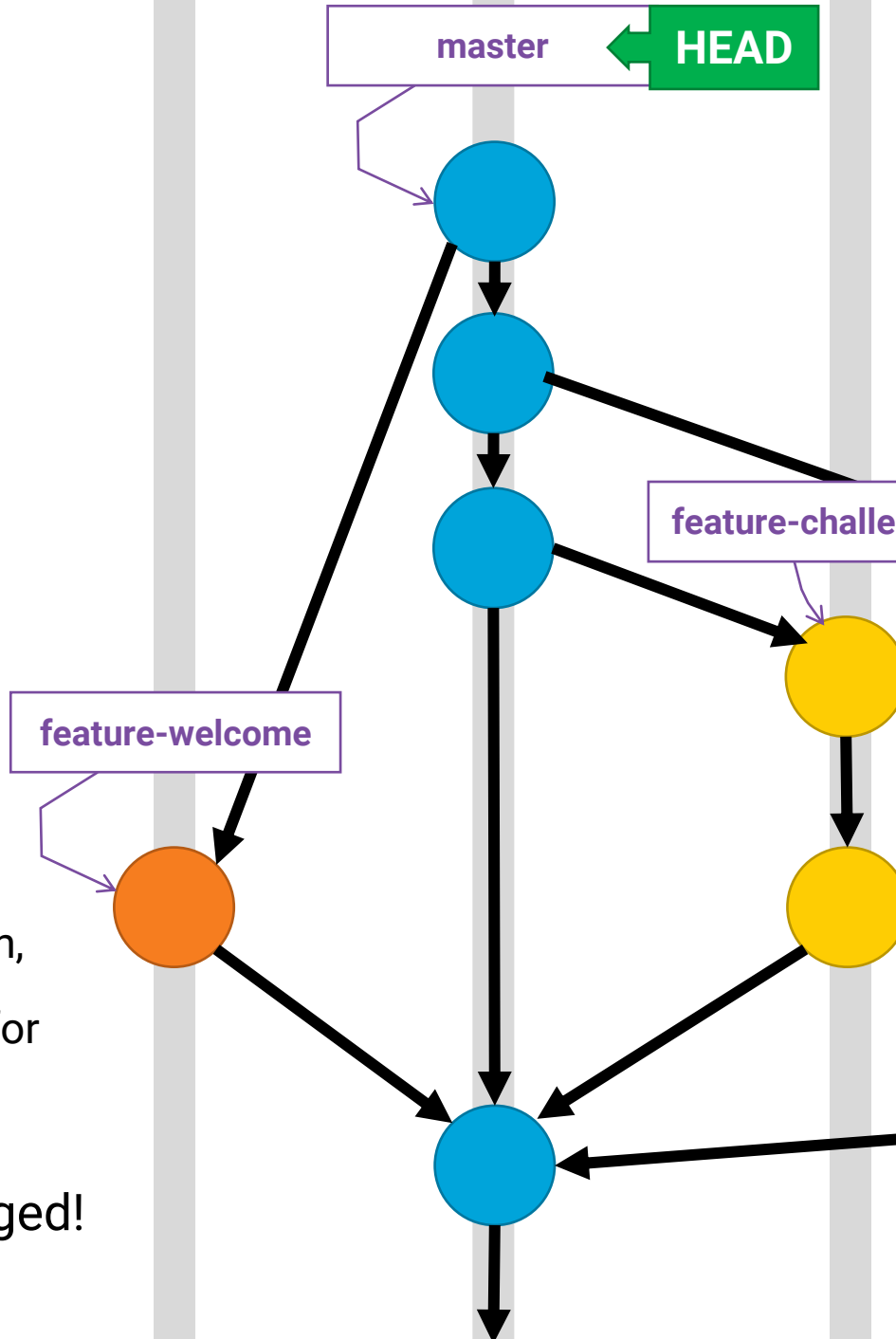
```
13 <<<<<<< HEAD
14     printf("Welcome");
15     printf("I'm thinking of a number between 0 and 511...\n");
16 =======
17     printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
18     printf("Welcome to the binary search game!\n");
19     printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
20
21     printf("I'm thinking of a number between 0 and 255...\n");
22 >>>>>>> feature-welcome
```

master

feature-welcome

- You decide what to keep or delete, make the changes, & save.
  - It's your responsibility to remove the <<<<<<, =======, >>>>>> lines
  - These give you context (HEAD area is your current checked out branch, feature-welcome is the branch you're merging in)
  - Pro-tip: use vim's regex search such as /===== followed by the n key for next match to be sure you handle all conflicting areas. Often there are multiple conflicting areas.

- Add conflicting file(s) to stage, make a commit, and you're merged!



We've created a bit of a mess!

- In an upcoming lecture we'll explore how to clean up some accidental non-linearity in a repository's history before merging.

```
*   d3e15e2 (HEAD -> master) Merge branch 'feature-welcome'
|\
| * 16e23f4 (origin/feature-welcome, feature-welcome) Improve welcome messaging
* |   11b18f0 Merge branch 'feature-count'
|\ \
| * | 61d6cae (origin/feature-count, feature-count) Add tries counter to the game.
| |/
* |   fef2836 Merge branch 'feature-challenge'
|\ \
| |/
|/|
| * e67e01b (feature-challenge) Improve message
| * 71bbb57 Increase the range from 0-511.
|/
* 75af0de (tag: 1.0.0, origin/master, origin/HEAD) Make a simple number guessing game.
* ca154d6 Improve the text.
* cb6cbe2 Add hello world program.
* 5412c74 Initial commit.
```