Introduction to git
Part II
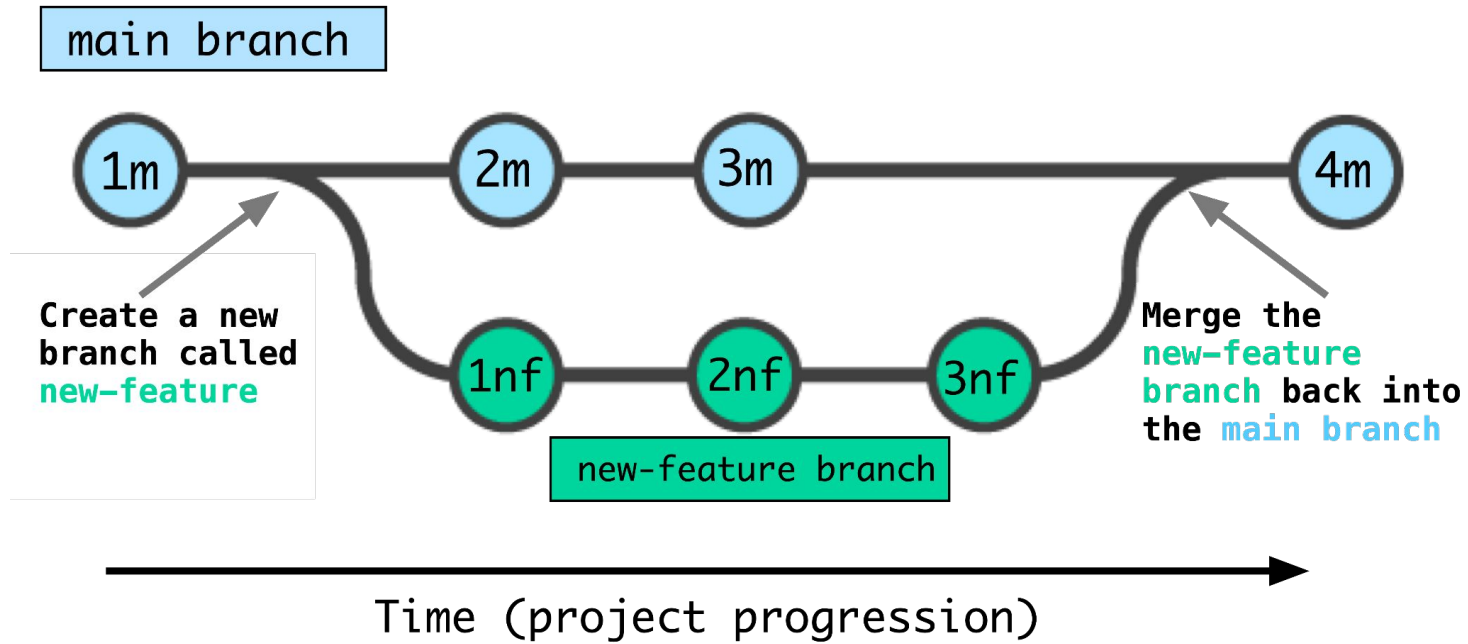
Childhood Cancer
Data Lab x Alex's Lemonade Stand

Branches in git
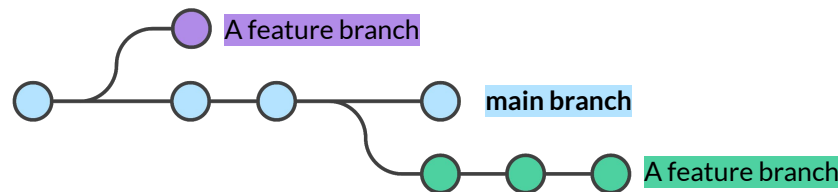
- Branches are like "repositories within repositories" 🤯
- Useful when you want to make changes (maybe experimental!) but you don't want to break the rest of your code
  - You can always switch back to a "clean" branch!
- Keep related changes together
  - All commits for a given new analysis or "feature" can be made within the same branch for easier tracking
  - Helps you to identify which commits are relevant to a given analysis
- If you wreck code in a branch, you've *only wrecked that branch!* Just delete it!
- Branches provide a great framework for collaboration and team science

main branch

1m — 2m — 3m — 4m

Create a new
branch called
new-feature

1nf — 2nf — 3nf

new-feature branch

Merge the
new-feature
branch back into
the main branch

Time (project progression)

main branch history after merge

1m 2m 3m 1nf 2nf 3nf 4m

Let's begin by exploring a real life GitHub repository

https://github.com/alexsLemonade/scpca-nf

...but first, a plug: https://scpca.alexslemonade.org/

Working with multiple branches

Why and when do we use **feature branches**?

There are several different models for `git` workflows (stay tuned!), but all make use of a standard paradigm:

- The Project Truth lives in `main` (formerly `master`)
- Code is developed in different branches, which over time get *merged into* the `main` branch
- We want to avoid working directly in the `main` branch
- All of this helps us modularize project development, keep a clear project history, and avoid conflicts with our collaborators

We use the term **feature branch** because each branch should have a specific scope that is limited to a given feature

When you create a branch, it literally *branches off* the branch you are in when you create it. This is called our **base branch**.

We often work with multiple branches at a time

You might be working with more than one feature branch, and your teammates are working in their own branch(es) as well

Image adapted from Atlassian
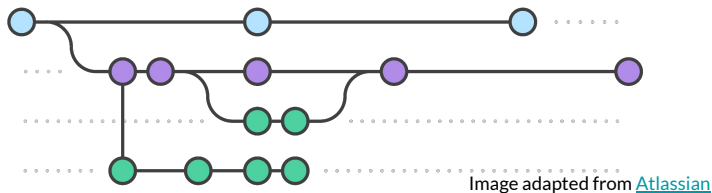
Tips for success:
- Always know what branch you're working in
- Before creating a branch, be cautious you are creating it from the correct base *and* that the base is up-to-date
- As you work, aim to keep your feature branch as up-to-date with its base as possible

… and how do you set yourself up for success? `git status`

Use an informative name for your feature branch

Informative names help **you** stay on track and organize your work, and help **your teammates** quickly get a sense of the scope of your work when reviewing your code

Let's consider the code we wrote for the histogram demonstration….

- **Bad names**
  - `feature`, `bins`, `patch-1`
- **A better name**
  - `add-bins-option-to-histogram`
- **An even better name**
  - `<username>/add-bins-option-to-histogram`
- **If you want to be very organized**
  - `<username>/<issue #>-add-bins-option-to-histogram`

Creating and switching between branches

See all local branches with `git branch -a`

To switch to a different branch...
- `git checkout <different-branch>`
- `git switch <different-branch> (git >= 2.23)`

To create a new branch...
- First, make sure you are in right branch you as your base with (surprise!) `git status`, and switch as needed!
- `git branch <new-branch-name>`

More fun with branches

🔥 To simultaneously create and switch into a new branch…
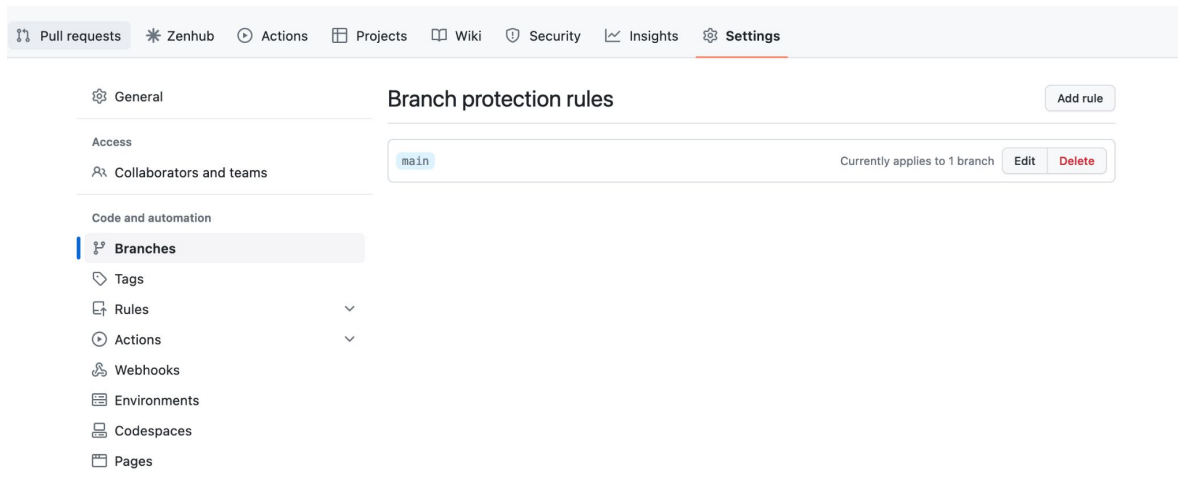- `git checkout -b <new-branch-name>`
- `git switch -c <new-branch-name> (git >= 2.23)`

Change your branch name: `git branch -m <updated-branch-name>`

⚠️ *Caution!* If you've already pushed your branch, this will *not rename the remote branch*. You'll also need something like…

```
git push origin -u <updated-branch-name>        # change your remote target branch
git push origin --delete <original-branch-name> # delete original remote branch
```

Help GitHub help you: Protect your `main` branch

Help GitHub help you: Protect your `main` branch

Branch protection rule

Branch name pattern *

> main

Applies to 1 branch

`main`

Protect matching branches

☑ **Require a pull request before merging**
When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

   ☑ **Require approvals**
   When enabled, pull requests targeting a matching branch require a number of approvals and no changes requested before they can be merged.

      [Required number of approvals before merging: 1 ▾]

   ☐ **Dismiss stale pull request approvals when new commits are pushed**
   New reviewable commits pushed to a matching branch will dismiss pull request review approvals.

   ☐ **Require review from Code Owners**
   Require an approved review in pull requests including files with a designated code owner.

   ☐ **Restrict who can dismiss pull request reviews**
   Specify people, teams, or apps allowed to dismiss pull request reviews.

   ☐ **Allow specified actors to bypass required pull requests**
   Specify people, teams, or apps who are allowed to bypass required pull requests.

   ☐ **Require approval of the most recent reviewable push**
   Whether the most recent reviewable push must be approved by someone other than the person who pushed it.

☐ **Require status checks to pass before merging**
Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

☐ **Require conversation resolution before merging**
When enabled, all conversations on code must be resolved before a pull request can be merged into a branch that matches this rule. Learn more.

☐ **Require signed commits**
Commits pushed to matching branches must have verified signatures.

☐ **Require linear history**
Prevent merge commits from being pushed to matching branches.

☐ **Require deployments to succeed before merging**
Choose which environments must be successfully deployed to before branches can be merged into a branch that matches this rule.

☐ **Lock branch**
Branch is read-only. Users cannot push to the branch.

☑ **Do not allow bypassing the above settings**
The above settings will apply to administrators and custom roles with the "bypass branch protections" permission.

☐ **Restrict who can push to matching branches**
Specify people, teams, or apps allowed to push to matching branches. Required status checks will still prevent these people, teams, and apps from merging if the checks fail.

Rules applied to everyone including administrators

☐ **Allow force pushes**
Permit force pushes for all users with push access.

☐ **Allow deletions**
Allow users with push access to delete matching branches.

Merging feature branch changes back into `main`

Merging itself creates a "merge commit" within the `main` branch (or, in whichever branch you are merging into)

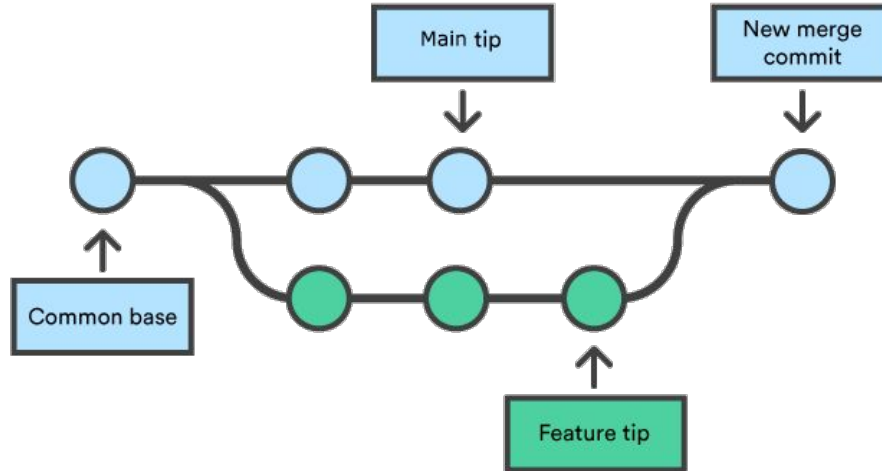If the feature branch is as up-to-date as possible with `main`, merge conflicts will be much less likely!

Image from Atlassian

Keeping your feature branch up-to-date with `main`

1. Locally, switch back to the main branch `git switch main`

2. Pull down `main` branch changes: `git pull main`
    - This will update your local `main` branch to match the remote `main` branch

3. Switch back to your feature branch: `git switch <feature-branch>`

4. Merge in the `main` branch updates: `git merge main`
    - You may enter **vi** as part of the commit that this command creates! Use **:wq** to get out.

Some caveats to the previous slide!

We assumed that the base branch is always `main`, but this is not always the case!
We'll see later a couple scenarios where your base branch is not `main`, but the same
concepts will apply.

This process will differ a little if you are working in a fork! You first have to keep your
`main` branch up-to-date with the **upstream** `main` branch:

```
git switch main          # switch to your main branch
git merge upstream/main  # merge the upstream main into your local main branch
git push                 # update your fork's remote main

# Now, you can sync your feature branch with your main branch
```
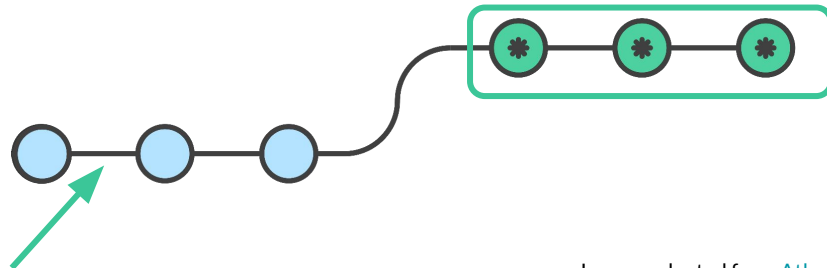
Merging and *rebasing* can be used to combine branch histories

Feature branch created here
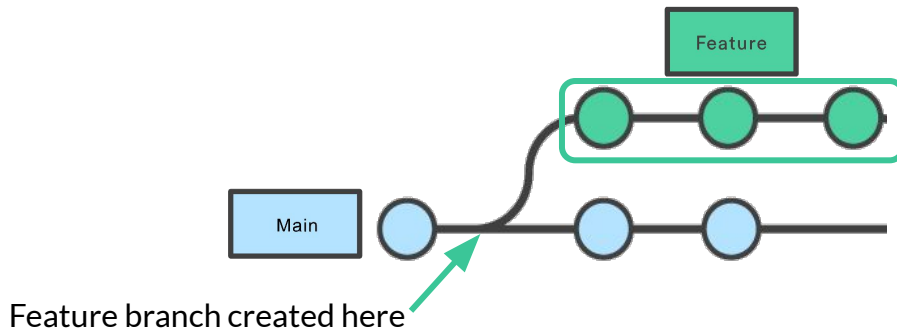
`git merge`
Retains full project history
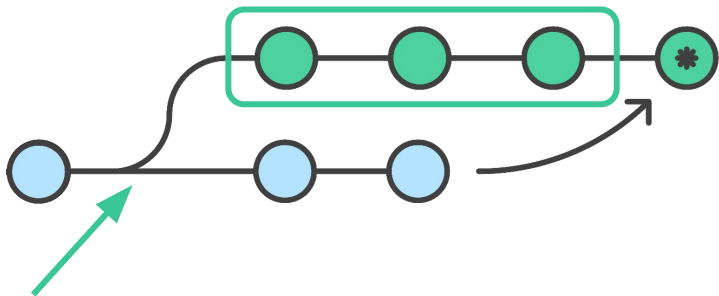
`git rebase`
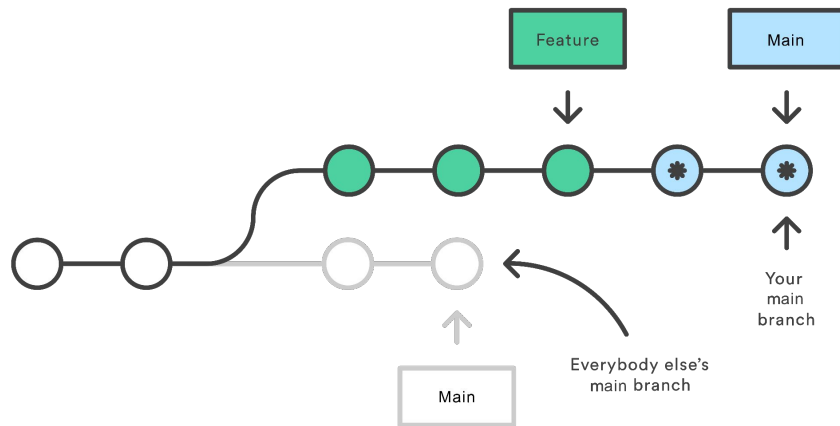*Overwrites* project history

Remember Atlassian's golden rule of rebasing

So, before you run git rebase, always ask yourself, "Is anyone else looking at this branch?"

AKA, never* use `git rebase` on a shared branch
- Public repositories with potential for open contribution
- Private repositories within your organization, even if not meant for external use or consumption

*unless the project maintainers tell you to

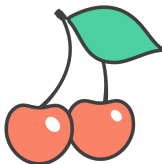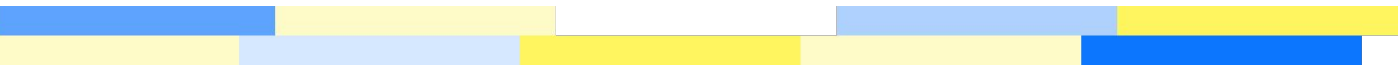# Helpful commands when working in multiple branches

- `git stash`
  - Use this to save "work in progress" code for later *without committing*
  - This commands adds changes since the last commit to the stash, which you can "apply" when you are ready
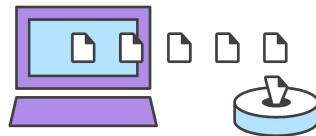
- `git cherry-pick`
  - This command will *copy* (not move!) commit(s) from one branch to another
  - The same commit(s) will now exist in *both branches*, meaning this command results in duplicate history
  - But, you can clean up after yourself if you absolutely need to (we'll see an example...now!)

Image from

Demo: Working with multiple branches