

# **GIT Refresher**

What is GIT? *Git is version-control software for working on group project collaboration*

What is Version Control? *Version control enables creating backup versions of your code*

## How does GIT work?

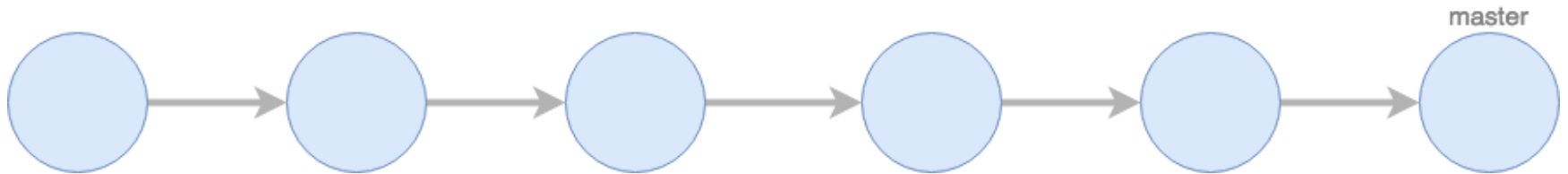
GIT represents history in a Directed Acyclic Graph (DAG). Each Node on the graph is a hashed GIT commit. By the backing data structure being a DAG, this guarantees Nodes representing later commits (in terms of time) will be organized after earlier commits.

# What you should already know...

By now, you should be familiar with basic GIT commands. Just in case, though:

- `git init` : initializes a git repository
- `git clone` : clones the contents of one repo to another
- `git status` : returns the current status of the repo
- `git add` : adds files to the staging area
- `git commit` : records the staged changes as a commit in the local repo
- `git push` : pushes any local commits to any downstream remote repos
- `git fetch` : synchronizes the local repo with its remote
- `git pull` : pulls new remote commits from the remote to the local repo

So far, the DAGs for you repos look like this:

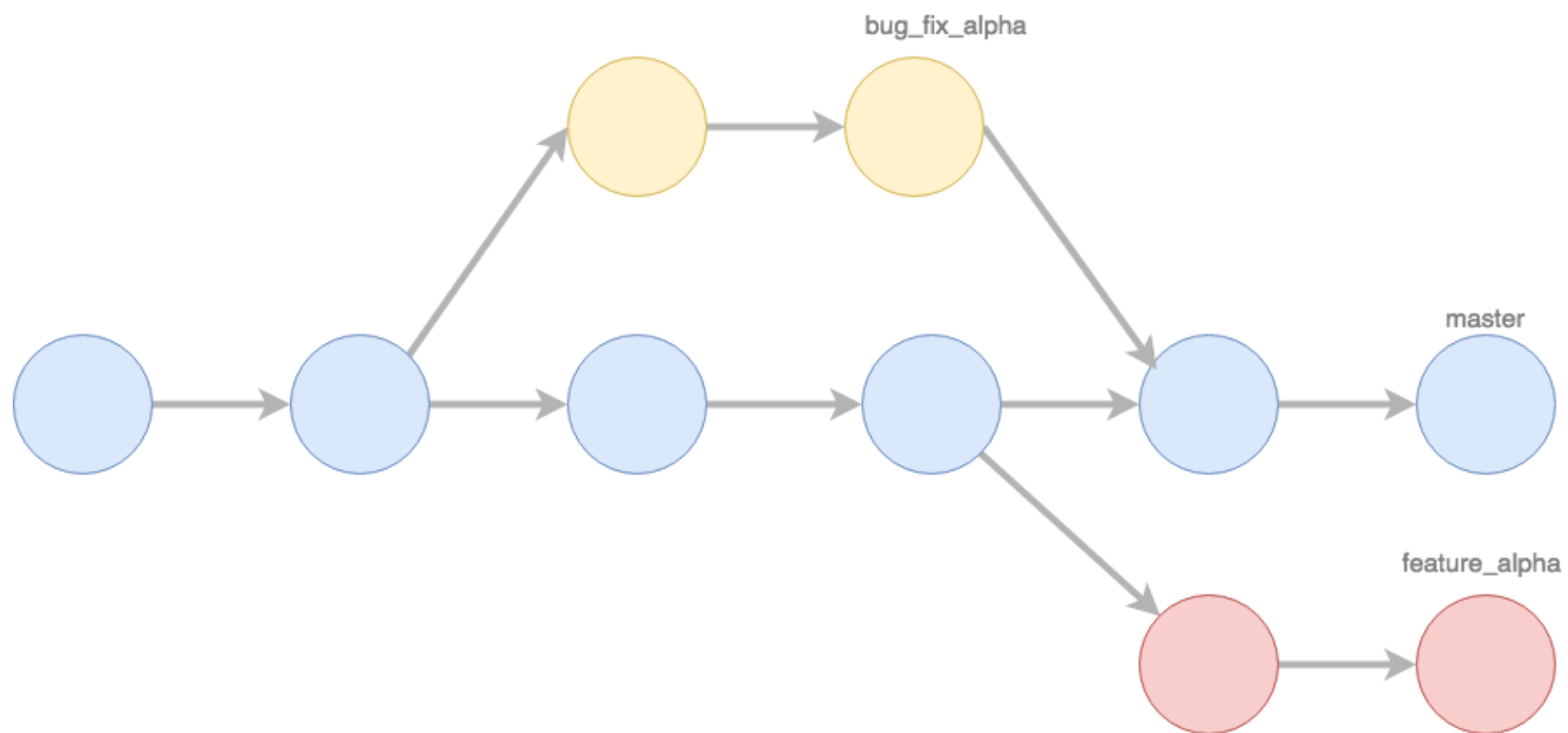


Each circle represents a commit. The right-most circle represents the *latest* commit.

This situation is somewhat ideal, in that it creates a clean *history* to work with. It's very easy to work in such an environment, as there are no conflicts in the GIT DAG.

However, rarely do we work in such an environment and rarely will your DAG look like this.

This is more typical:

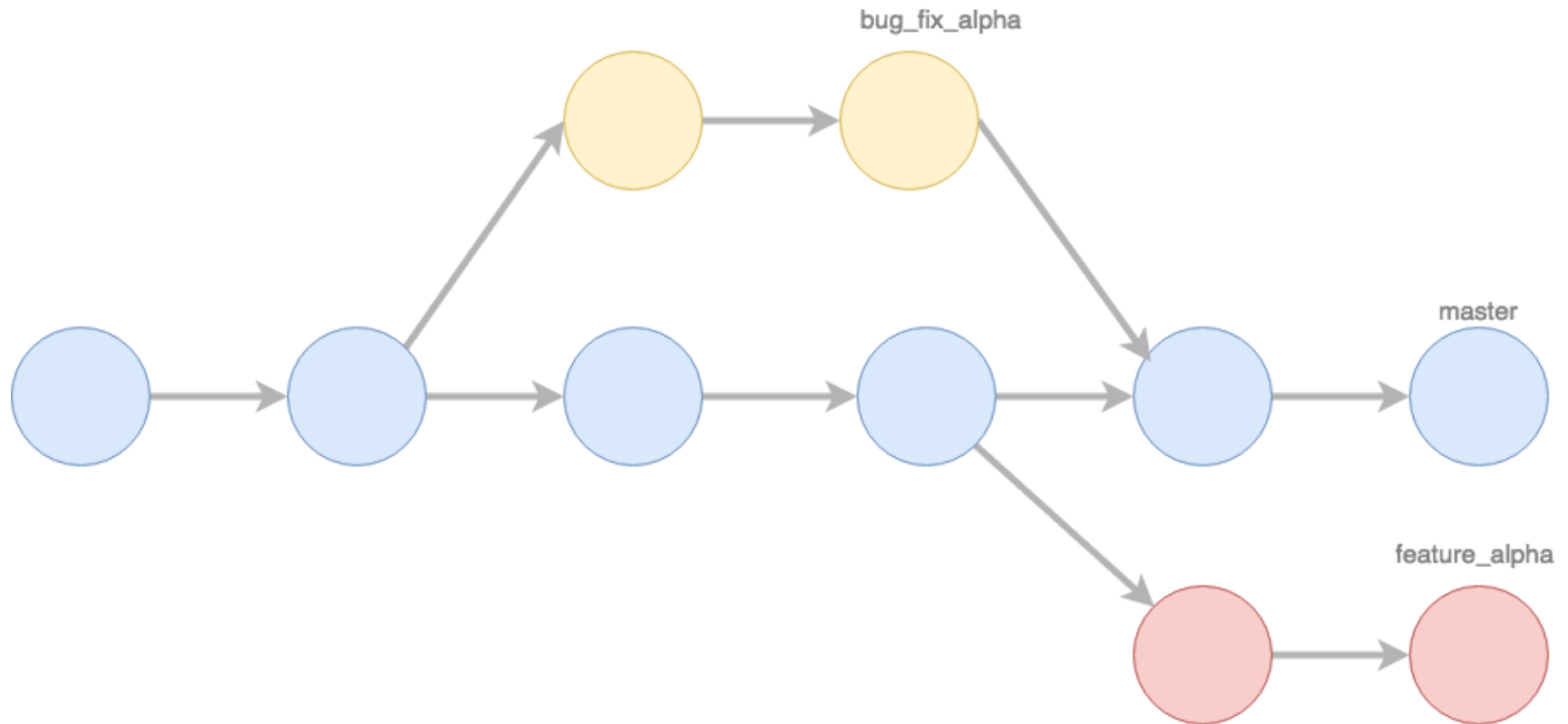


So what's with *bug\_fix\_alpha*, *master*, and *feature\_alpha*?

Those are **branches**.



A branch is a reference to a point where the DAG *diverged*. In other words, we've changed code that is not represented on our mainline, or *master* in our case. This is useful for a number of reasons, but the most useful in our context is it allows multiple people to work on the same piece of code. Let's take a look at our DAG again:

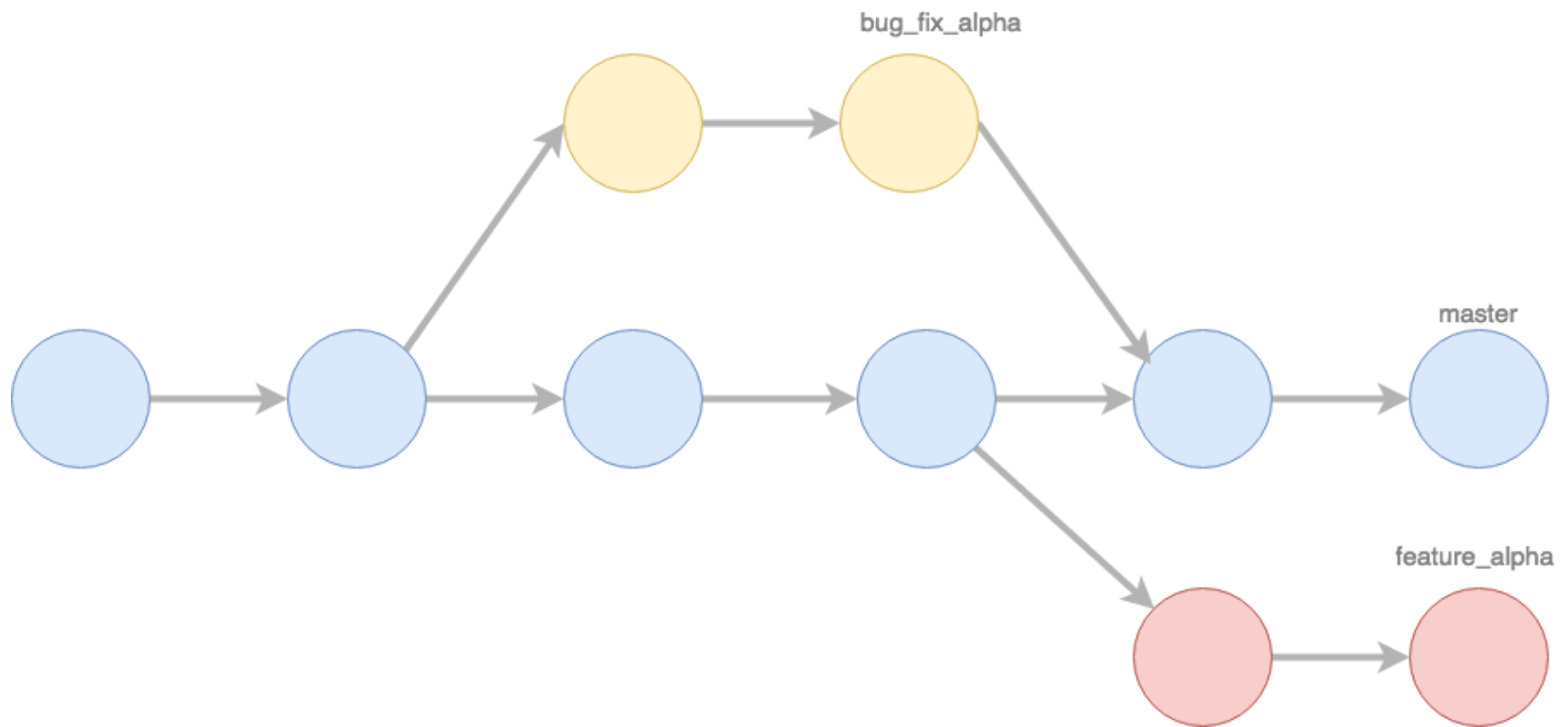


You can think of the blue circles as the shared version of the code, while the yellow and red circles represent teammates.

Let's work our way down our DAG to track what has happened.

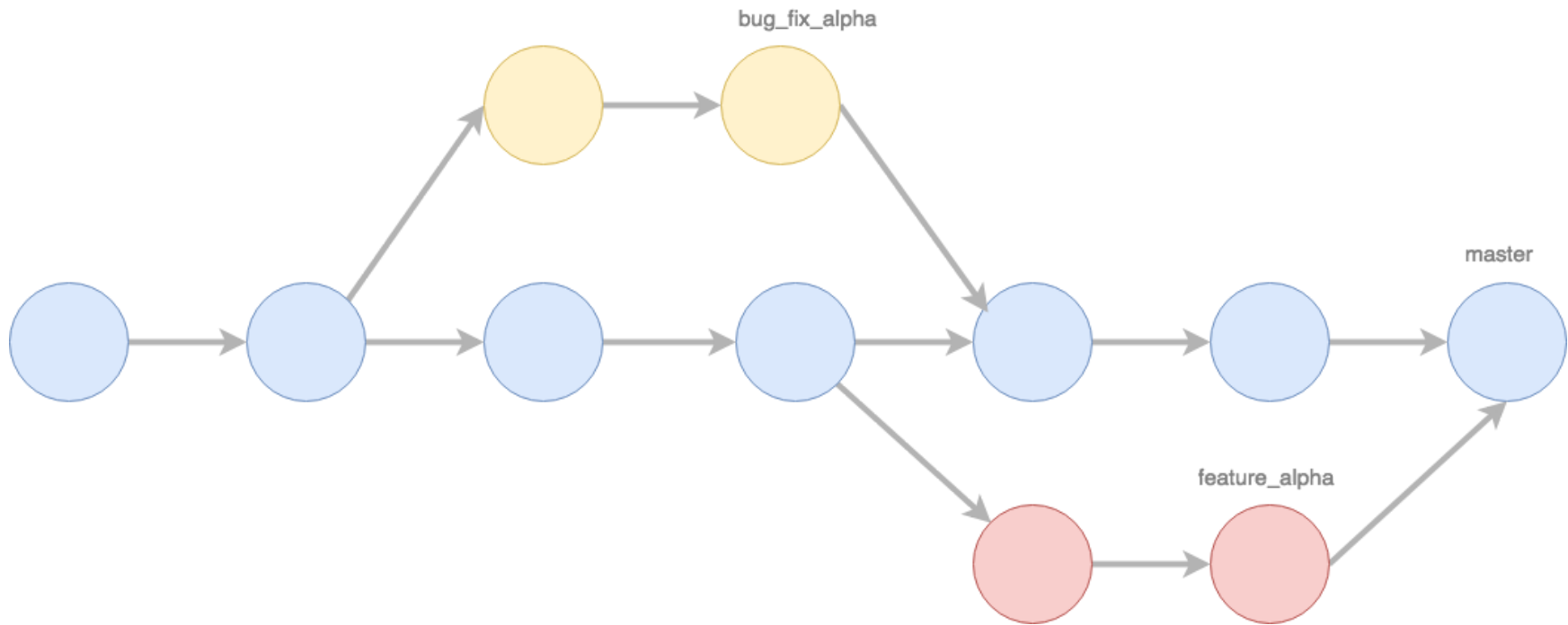
1. A commit is made to *master*
2. A commit is made to *master*
3. A branch (*bug\_fix\_alpha*) is created
4. Commits are made to *bug\_fix\_alpha* and *master*
5. Commits are made to *bug\_fix\_alpha* and *master*
6. A branch (*feature\_alpha*) is created
7. The changes on *bug\_fix\_alpha* are merged into *master* and a commit is made to *feature\_alpha*
8. Commits are made to *master* and *feature\_alpha*

Let's take another look:

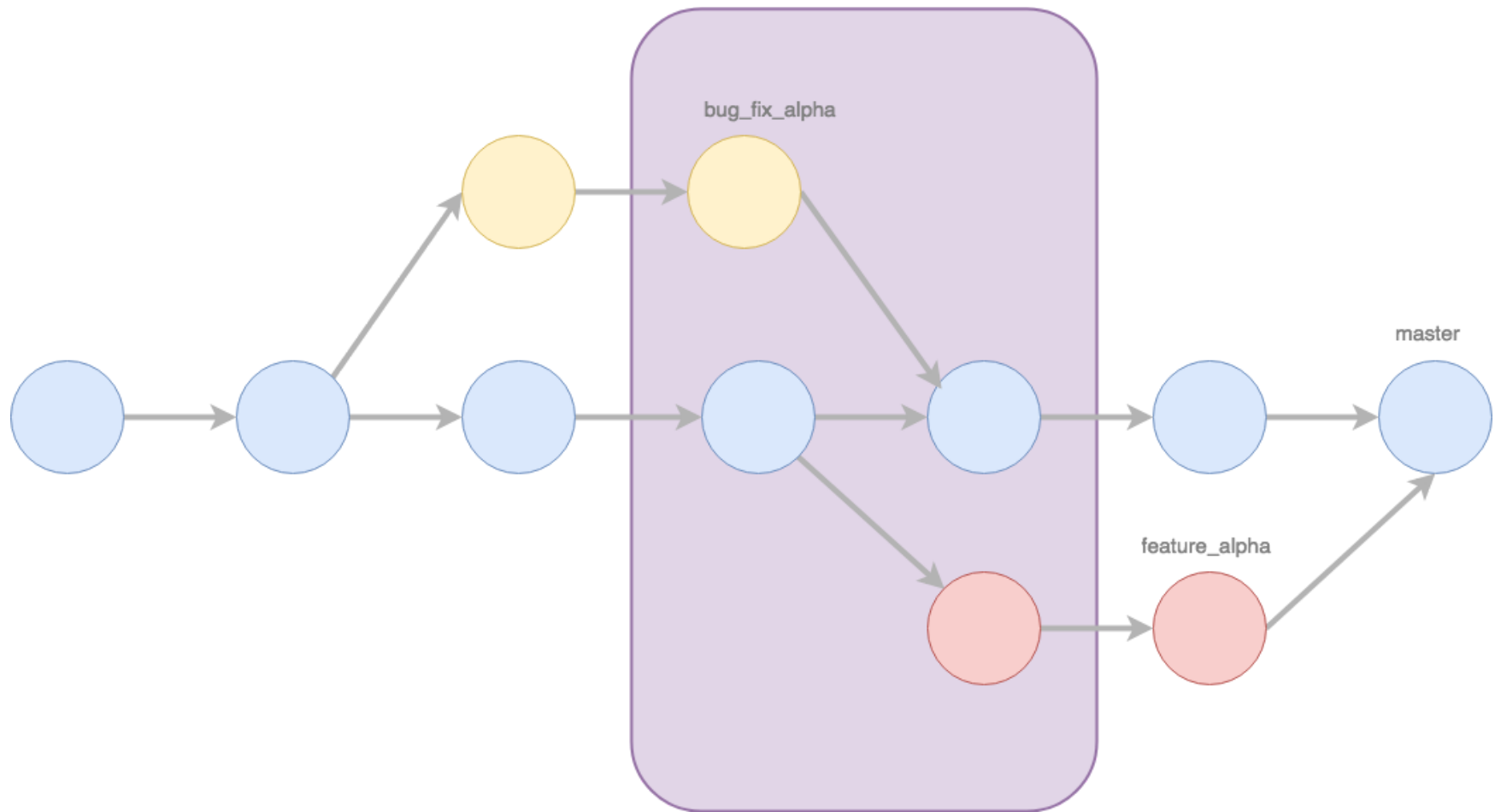


Take a look at *master* and *feature\_alpha*. What do you notice?

The branches have diverged and need to be merged . What would the DAG look like after the merge ?



There's one more detail here we need to discuss.



Do you see any issues with the shaded region? What happens when the changes from *feature\_alpha* are merged in with *master*? Does *feature\_alpha* have the latest changes from *bug\_fix\_alpha*? No, it doesn't. This is called a **conflict**.

When the developer working on *feature\_alpha* tries to merge in, it's likely (although not guaranteed), a conflict will result. If that happens, the conflicts will need to be manually resolved before the merge is completed.

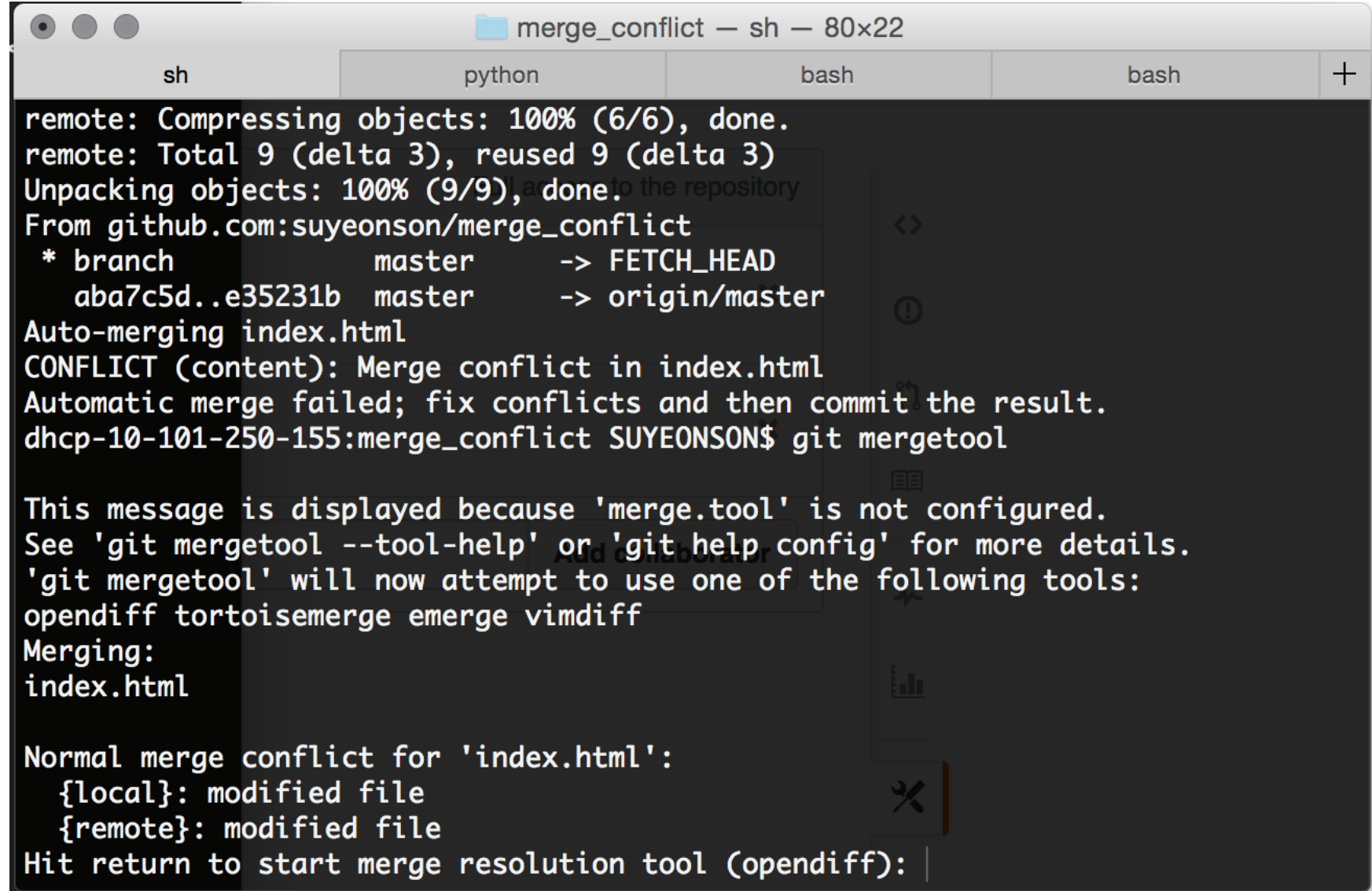
# Terminology

- `git checkout -b branchName` : creates a branch named branchName
- `git checkout branchName` : switches to a previously created branch named branchName
- `git branch` : list available branches
- `git branch -d branchName` : deletes branch named branchName
- `git merge branchName` : while on some branch, merges the changes on the last commit of branchName with the latest commit of the current branch.



# Resolving GIT conflicts

You will undoubtedly run into a GIT conflict. If this happens, you'll see something like this in the terminal window:

A terminal window titled 'merge\_conflict — sh — 80x22' with tabs for 'sh', 'python', 'bash', and 'bash'. The terminal output shows a successful git pull from 'github.com:suyeonson/merge\_conflict' followed by a merge conflict in 'index.html'. The user runs 'git mergetool', which displays a message about the 'merge.tool' configuration and lists available tools: 'opendiff', 'tortoisemerge', 'emerge', and 'vimdiff'. The user selects 'opendiff', and the terminal shows the start of the merge resolution process for 'index.html'.

```
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 3), reused 9 (delta 3)
Unpacking objects: 100% (9/9), done.
From github.com:suyeonson/merge_conflict
* branch                master      -> FETCH_HEAD
  aba7c5d..e35231b      master      -> origin/master
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
dhcp-10-101-250-155:merge_conflict SUYEONSON$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff tortoisemerge emerge vimdiff
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

You'll also see something like this in your code:

```
diff --cc test.txt
index 20b652c,e932d49..0000000
--- a/test.txt
+++ b/test.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<< HEAD
+0804201702
++=====
+ 0804201901
++>>>>>> feature_a
(END)
```

Everything between the first <<<<<<< and ===== is what's on the current branch (or repo). Everything between the ===== and second <<<<<<< is what's on the merging branch (or repo). To fix this, you need to delete **everything** except the code you want to keep. In the above example, if the correct number is 080420702 , then every other line should be removed.

After every conflict has been resolved, a GIT `add`, `commit` and `push` will be needed to complete the merge.

It's important to note a merge conflict can be between branches OR repos. What that means in our case is two developers working on the same branch and making changes to the same file can also be in conflict.

Merge conflicts can be painful. Avoid them by frequently pulling from your remote repository, and merging your mainline into your branches.