Git branches

Reuven M. Lerner, PhD
reuven@lerner.co.il • http://lerner.co.il/

1

Branching

- Branches are fabulous
- But in most other version control systems, they are painful to deal with
- Not so in Git!
- Git makes branching and merging painless
  - (OK, mostly painless)

How does Git make it easy?

- Branching in Git is easy because it's a trivially fast and easy operation
- It just adds a name (the branch name) as an alias for an existing commit!
- Yes, really. That's it.
- So when you're "in" a branch, the branch name indicates the commit to which the new commit will be added.

Why branch?

- New release
- Working on a new feature
- Working on a bug fix
- Trying an upgrade, to see how it works

Branching in general

- In Git, you should branch whenever you do something new or different
- Every feature or bug fix goes on a new branch!
- I try to do almost no work on the “master” branch, if I can help it
- When I fail to do this, I almost always regret it!

Branching strategies

- The basic approach in Git is: Everything should go on a new branch
  - Fixing a bug? New branch.
  - New feature? New branch.
  - Experimenting? New branch.
- Then, when you're done with your experiment, you merge back onto the "master" branch.
- Branching and merging are very cheap operations in Git, and that's part of its magic!

git branch

- “git branch” with no options shows the available branches, marking the current one
- “git branch NAME” with a name creates a branch NAME; switch to it with "git checkout NAME"

```
git branch foo
```

```
git checkout foo
```

- You can combine the two with

```
git checkout –b foo
```

Deleting branches

- “git branch -d” with a name deletes a branch by that name
- Git will warn you if there are commits on that branch which haven't been merged elsewhere
- Use "git branch -D" if you want to delete anyway

```
$ git branch                            # show all branches, *current
* master

$ git branch foo                        # create a new branch

$ git branch                            # show all branches, *current
  foo
* master

$ git branch -d foo                     # delete the "foo" branch
Deleted branch foo (was 22d0108).

$ git branch                            # show all branches, *current
* master
```

Notice!

- In CVS and SVN, creating a branch means checking out a new subdirectory.
- Working on two branches means having two checked-out subdirectories.
- Not so in Git: When you switch branches, Git rewrites the filesystem to show you the current state

Yes, you got that right!

- You will have one, and only one, copy of a repository on your computer.
- Want to work in another branch? Switch branches *within the existing repository*.
- Switching branches is nearly instantaneous!
  - Git reorders files to reflect the state of the branch
  - HEAD now points to your new branch

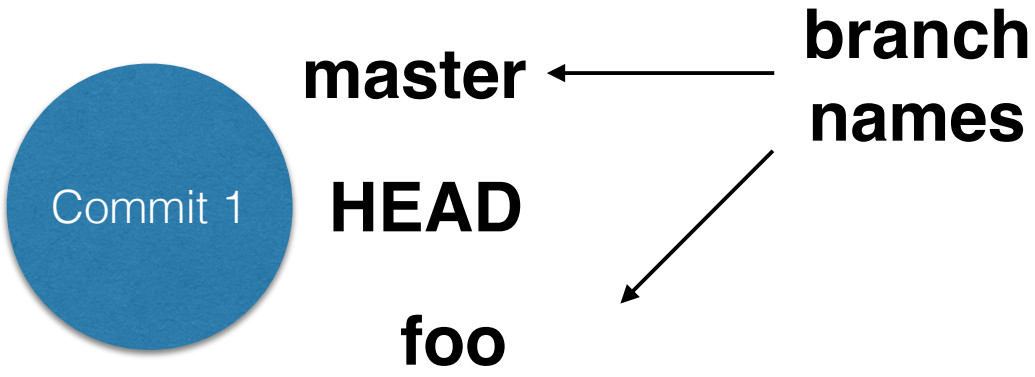
Switching branches

```
$ git branch foo                # Create branch "foo"

$ git checkout foo              # Switch to branch "foo"
Switched to branch 'foo'

$ git branch                    # List all branches, *current
* foo
  master
```

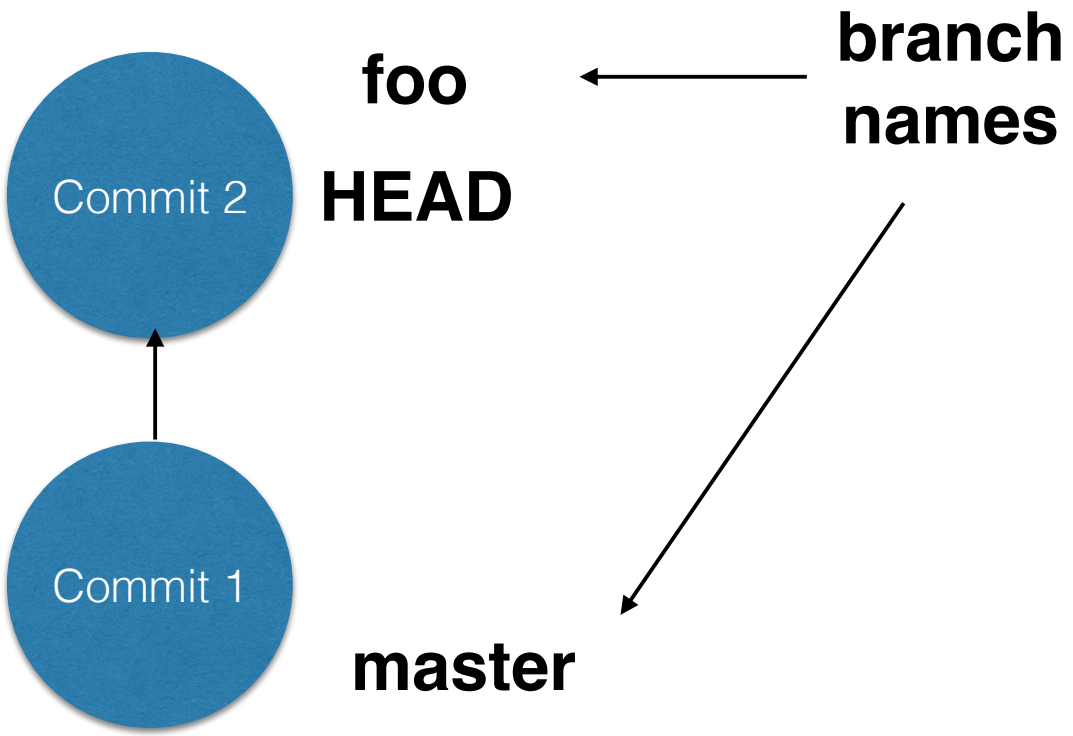
OK, so now what?

- Whatever changes you make on branch “foo” will not be around on “master”
- (And that's because the commits will have "foo" as an ancestor, rather than "master")
- It’s a completely independent workspace

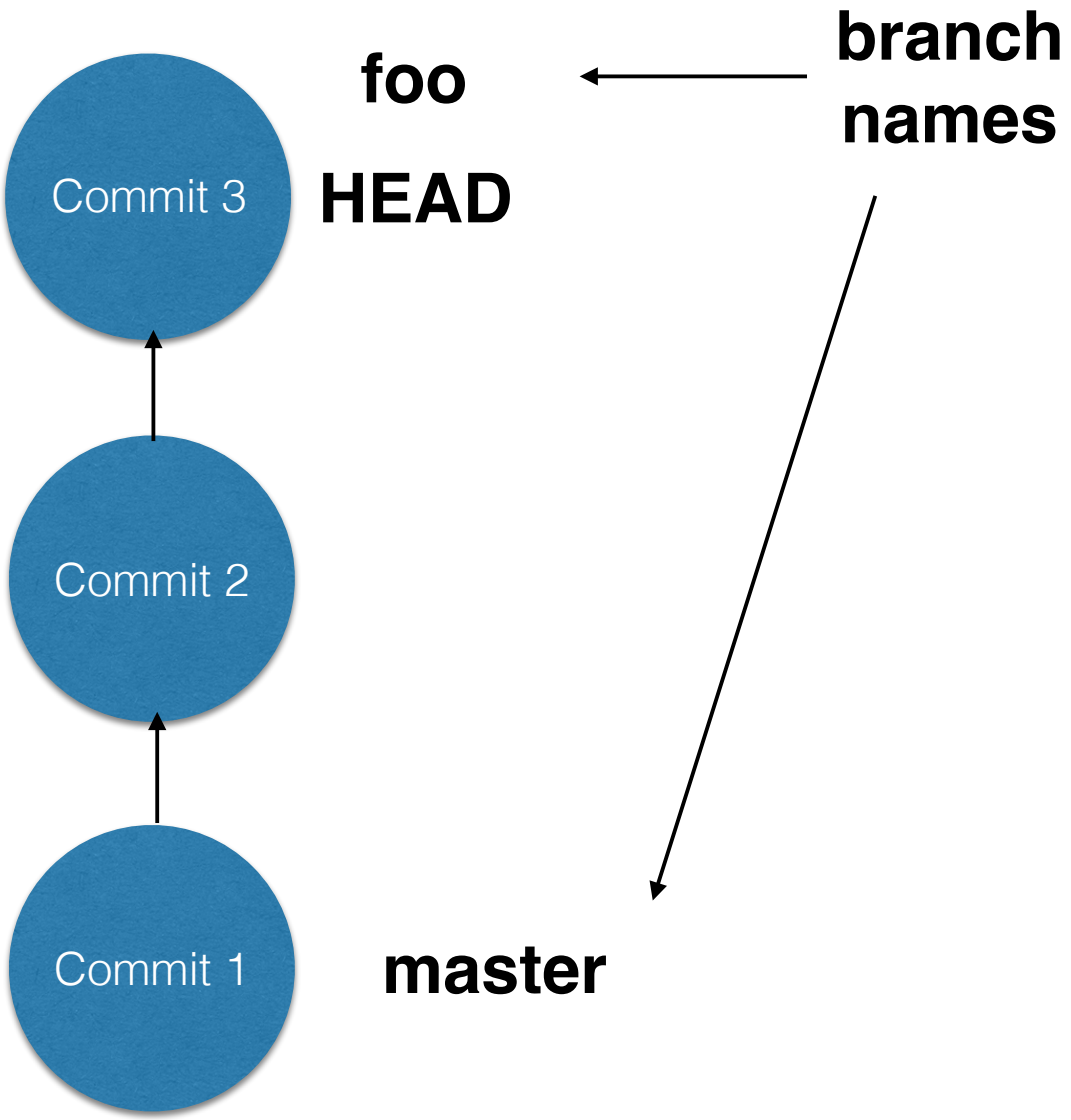
A visual depiction



If we add a commit on "foo"…



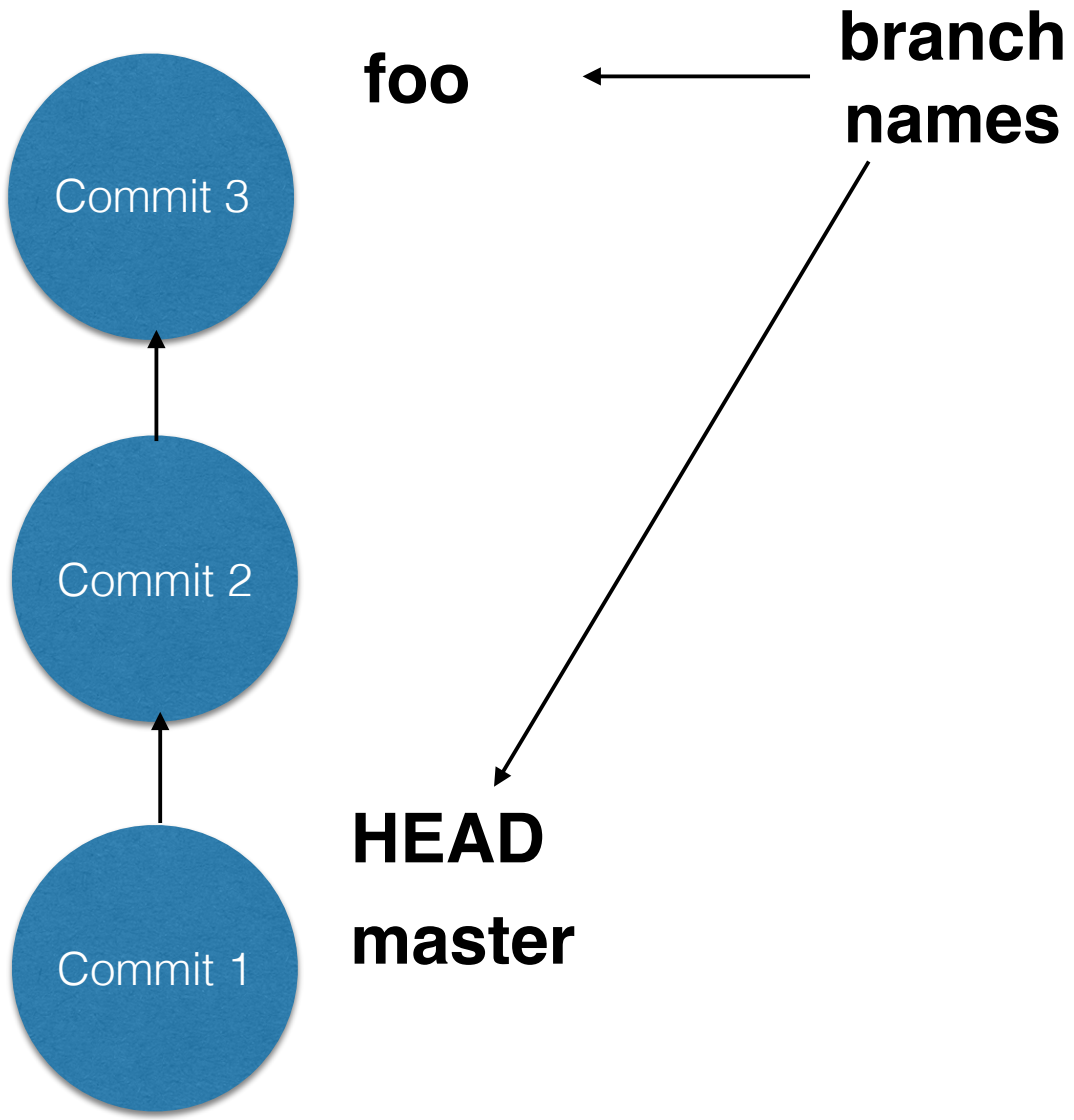
If we add a second commit on "foo"…



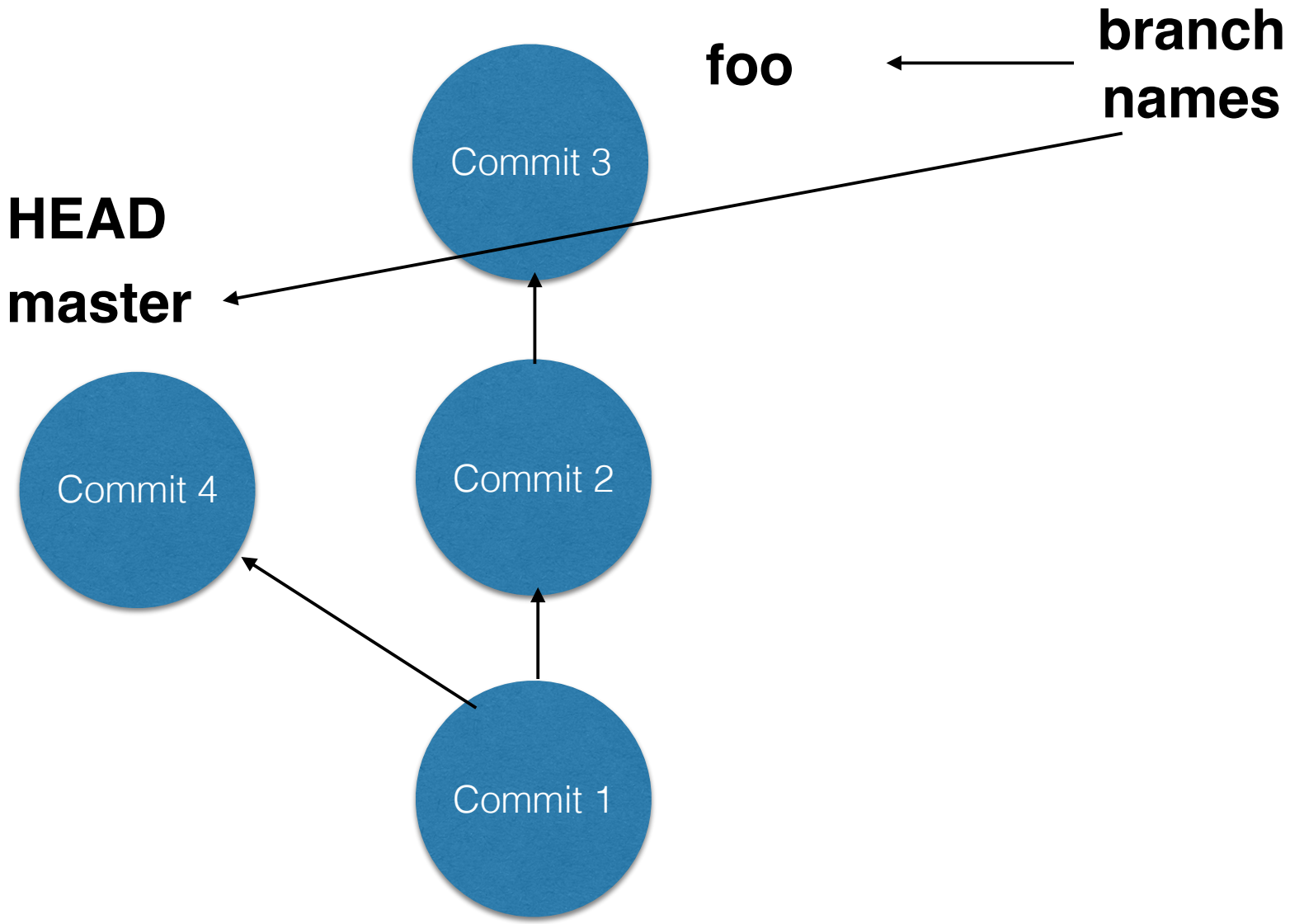
Now for something crazy!

- Poor "master" is still pointing back to commit #1
- But "foo" (aka HEAD) is pointing to commit #3
- What if I do "git checkout master"? Now HEAD points to "master"!
- And (most importantly) if I make a new commit now, its parent will be "master", not "foo" — in other words, I've created a tree of commits!

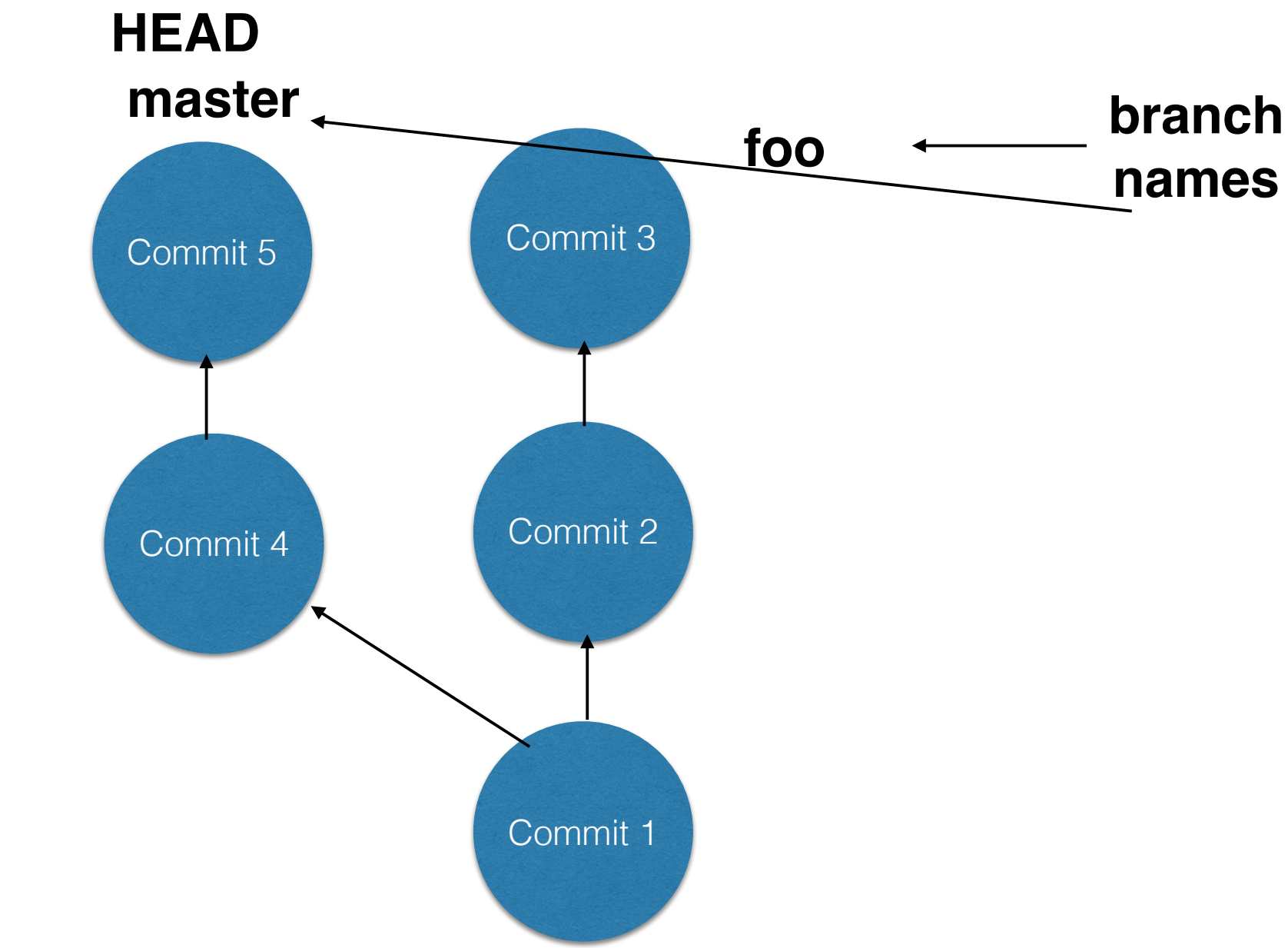
"git checkout master"



Add a new commit to "master"!



Add another commit on "master"!



Understand the logo now?

Git is a tree!

- Each commit has a unique name
- Some commits are also branches (i.e., the branch name is an alias for the commit name)
- Branching is fast and easy, because we're just adding names and aliases

Create some new files

```
$ mkdir stuff
$ cd stuff
$ cat > thing1.txt
This is Thing 1.
$ cat > thing2.txt
This is Thing 2.
$ cd ..

$ git status
# On branch foo
# Untracked files:
#  (use "git add <file>..." to include in what will be committed)
#
#    stuff/
nothing added to commit but untracked files present (use "git add" to track)
```

```
# Do things on branch foo
```

Make our commit

```
$ git add .                                          # Add everything
$ git status                                         # What's our status?
# On branch foo
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     new file:   stuff/thing1.txt
#     new file:   stuff/thing2.txt
#

$ git commit -m 'Added files'                        # Commit thing1 and thing2!
[foo e17012b] Added files
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 stuff/thing1.txt
 create mode 100644 stuff/thing2.txt
```

Meanwhile, in master

```
$ ls                                # on branch "foo"
hello.rb  stuff/

$ git checkout master               # switch to "master"
Switched to branch 'master'

$ ls                                # Files from "master" are gone!
hello.rb
```

Seeing the tree

- Yes, you can use a graphical tool
- But if you're stuck, you can always use

```
git log –graph
```

or

```
gitk
```

When you switch branches

- You switch the HEAD. That's basically it.
- If some files were added or changed, but not committed, they come with you.
- If some files were staged, but not committed, they come with you.

Switching branches with modified files

```
git checkout master

echo ‘foo’ > test.txt

git add test.txt

git commit –a –m ‘Added new file’

echo ‘bar’ >> test.txt

git checkout develop   # Not allowed!
```

Error message!

```
$ git checkout develop

error: Your local changes to the following
files would be overwritten by checkout:

   test.txt

Please, commit your changes or stash them
before you can switch branches.

Aborting
```

Why not?

- “git checkout” means: Show me the state of the repository, as of a particular commit (or branch)
- Normally, this means that files will change to reflect how they are (were) in that commit
- But if you have changes in your working directory, then Git has to choose between your working directory, the destination commit, or a merge — not an obvious choice.
- So it refuses to choose!

But if it's a new file…

- This error message won't happen with a new file (untracked or added, but not committed)
- The file doesn't exist in either commit (what you're coming from, or going to), and thus can't cause a conflict. Git doesn't need to decide or choose.

Re-doing a branch

- Let's say you want to remove three commits from a branch. How can you do that?
- You could revert them, of course. Or do a new commit. But we could also do the following:

```
git checkout SHA1          # Detatched head

git branch -d mybranch     # Delete branch

git branch mybranch        # Create branch here
```

Or, more easily..

```
git reset --hard SHA1
```

Renaming a branch

- You can rename a branch with the -m option:

```
git branch OLDNAME –m NEWNAME
```

Comparing branches

- See all of the changes between master

```
git diff master
```

- Or a single file:

```
git diff master afile.txt
```