Git merging

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

1

Merging!

- Once you've finished working in your branch, you want to merge it back into "master"
- Remember that "master" is the default branch, but you can use any branch for releases
- (But most people use "master")

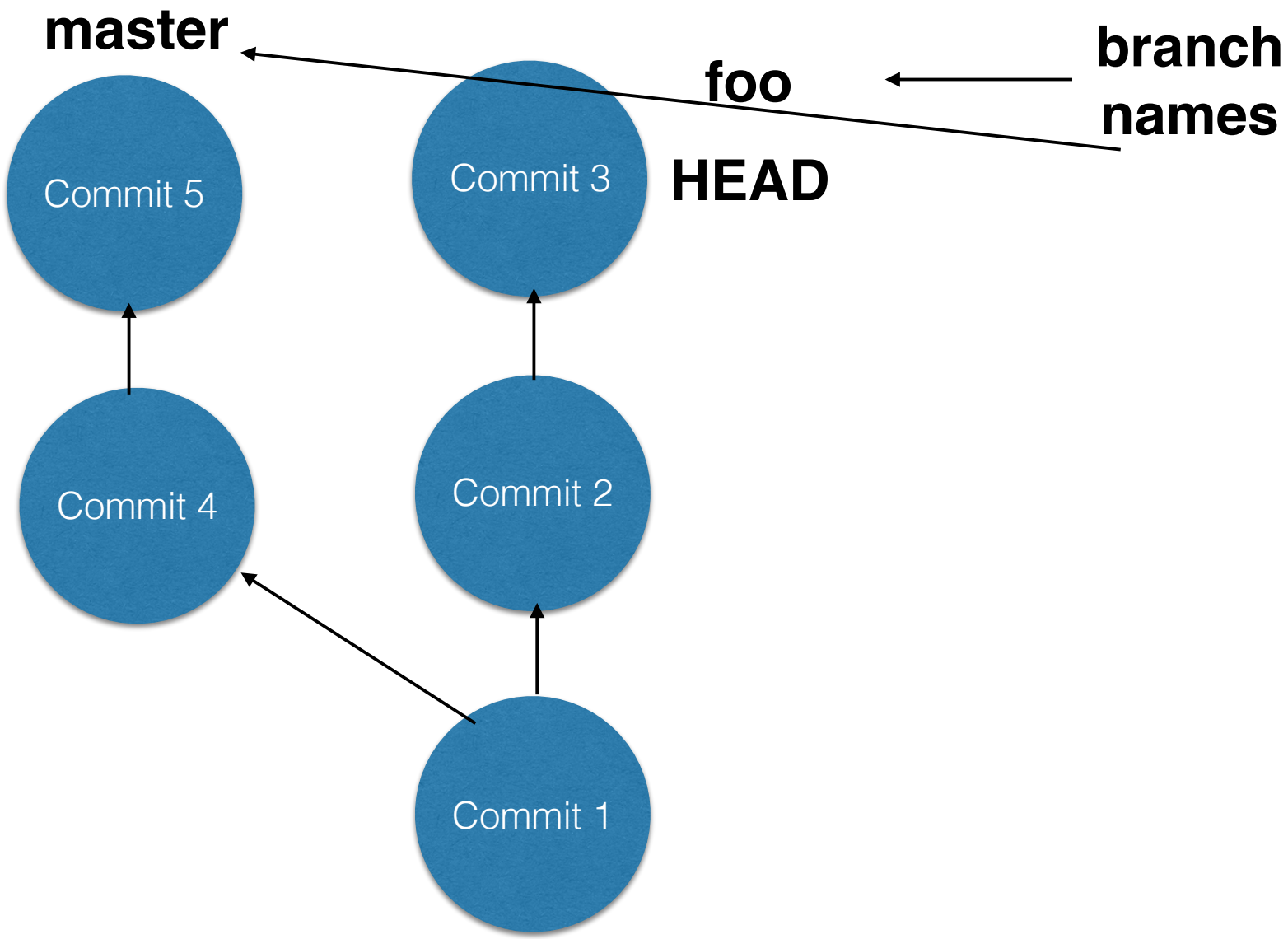
Merging

- Moving files from “foo” into “master” is as easy as “git merge”
- Go into the receiving branch, and type “git merge” and the name of the sending branch
- Merge really means: I want all of the commits from the other branch to now be in my tree, too!

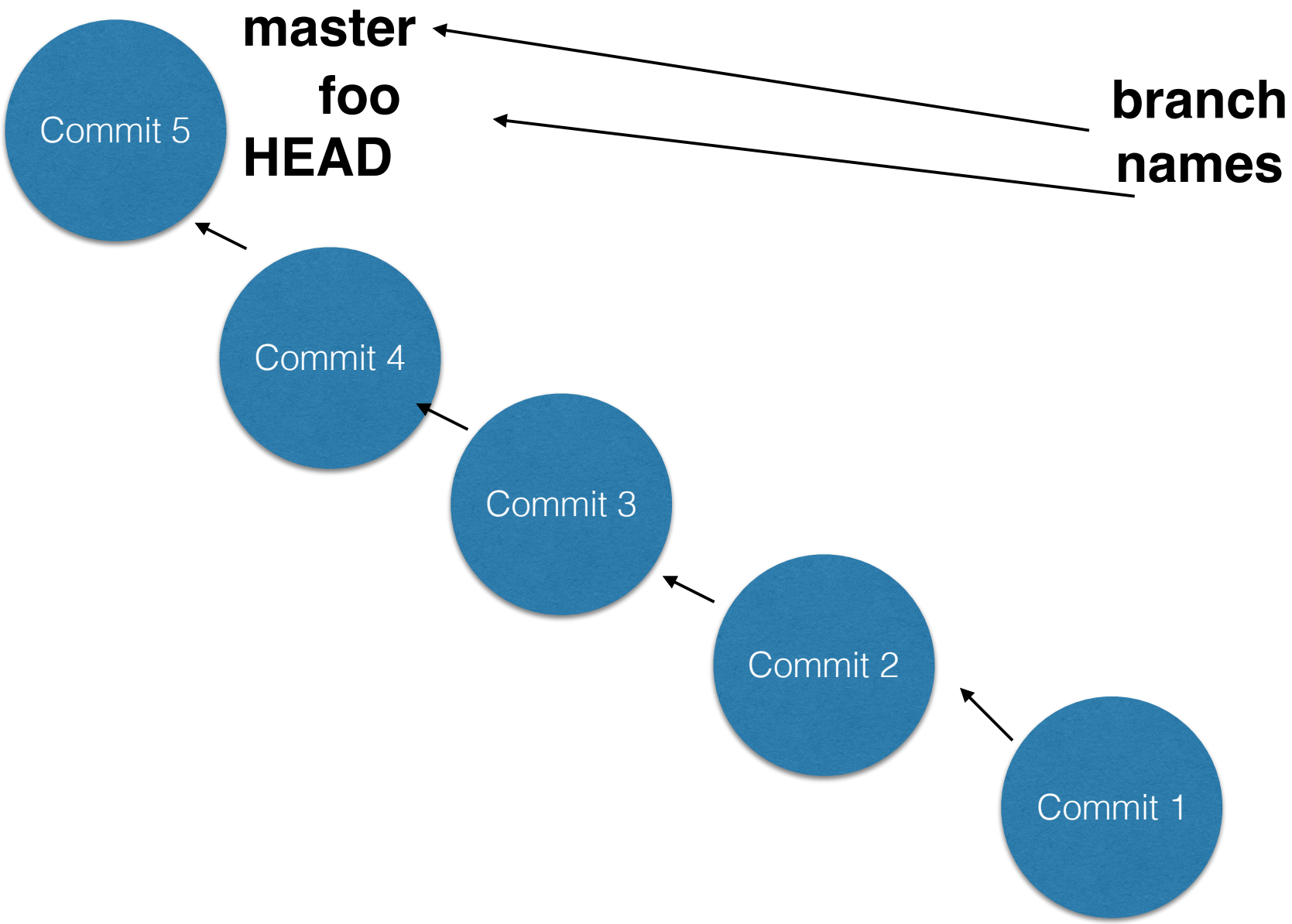
Again: What is merging?

- "git merge foo" means:
  - I want all of the commits in "foo"!
  - If "foo" has any commits that I lack, I want them!
- So we're just comparing, grabbing, and integrating commits
- Yes, this requires some rewriting of the tree. But Git knows how to do that.

Before the merge



After the merge



Merging is one-way!

- There is a difference between:
  - going into a ("git checkout a") and merging from b ("git merge b")
  - going into b ("git checkout b") and merging from a ("git merge a")

Merge ancestor

- Git determines, automatically, which common ancestor to use when merging
- You can thus just merge, without having to think about it much more than that!

Merging

```
$ git merge foo                 # Get commits from foo
Updating 22d0108..e17012b
Fast-forward
 stuff/thing1.txt |    1 +
 stuff/thing2.txt |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 stuff/thing1.txt
 create mode 100644 stuff/thing2.txt
$ ls
hello.rb  stuff/
```

We copied a commit!

```
$ git log --pretty=oneline
e17012bef6eff2332b6cef4331f79d92cdc64d44 Added files
22d010830febf7c85184908228e862fb17d2f7ca Revert "Updated to say bye"
f7310fd5064c4cc649f96c9d12faedb8327fa296 Updated to say bye
a4d9e4f924765a1abec8a3ea711c657e8952aff5 Added .gitignore
b53dd549c4891f094ab427852899da958fb9ae21 I added a file
```

Merging in Git

- The reason merging is so easy in Git is that you're simply copying a commit from one branch to another
- You don't need to change directories, communicate with the server, or anything

“Fast forward”

- When you merge, you’ll often see Git say that it’s “fast forwarding.”
- This means that it’s an easy merge case:
  - The branch is a superset of master
  - So the merge can take place by moving the HEAD pointer to the end of the branch

Not just master!

- You can merge between any two branches
- I usually end up merging from a development branch into master, but that's just me
- Just go into the destination branch (typically master), and merge from any other branch

My branching strategy

- New branch (`git checkout -b new-feature`)
- Make my changes
- Commit (`git commit -a -m 'Awesome features!'`)
- Merge from master (`git merge master`)
- Test my code
- Checkout master (`git checkout master`)
- git merge new-feature (merge back!)
- Test my code again
- Done! (`git checkout -d new-feature`)

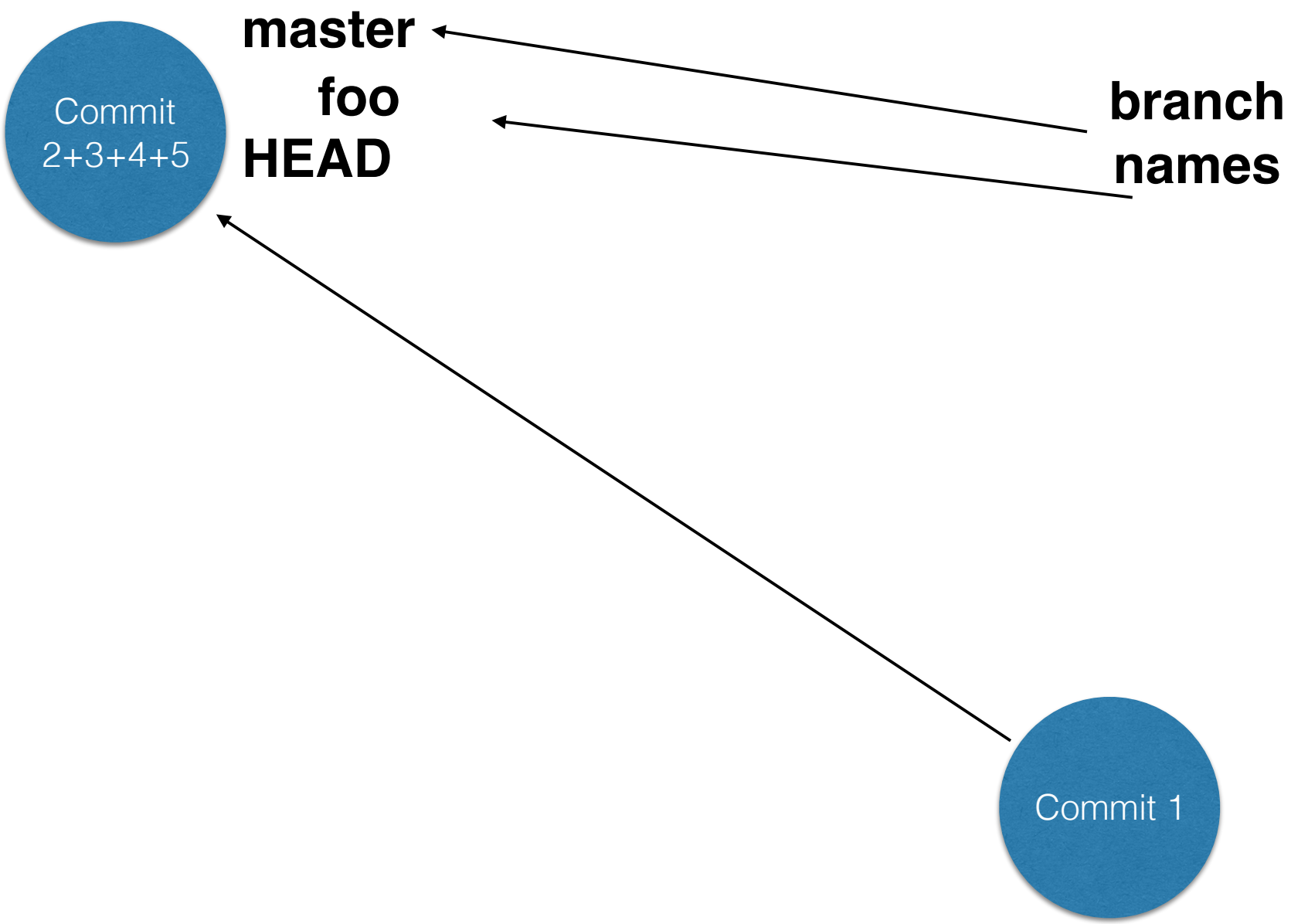
Another strategy

- Want to merge branches A and B?
- Create a third branch, and merge there
  - If things go wrong, you just delete the branch
  - If things go well, merge back into A and/or B

Squashed merges

- Often, you will make many commits in a branch before merging to master
- You don’t want all of these commits to show up separately in master
- You can use the --squash option to “git merge”, to turn them into a single commit

Squashed merge



```
$ git commit squash1.txt -m '1'
[foo 5045326] 1
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 squash1.txt
$ git commit squash2.txt -m '2'
[foo 619d6d3] 2
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 squash2.txt
$ git commit squash3.txt -m '3'
gi[foo ffe581b] 3
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 squash3.txt
$ git log --pretty=oneline
ffe581bab89abbc478d83767134a2453875dc9d5 3
619d6d39c0f885f8897159cf4c3c2b8511437505 2
50453267b7d429162a92db82380a10d6ba6337a0 1
e17012bef6eff2332b6cef4331f79d92cdc64d44 Added files
```

Merge!

```
$ git merge foo --squash
Updating e17012b..ffe581b
Fast-forward
Squash commit -- not updating HEAD
 squash1.txt |    1 +
 squash2.txt |    1 +
 squash3.txt |    1 +
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 squash1.txt
 create mode 100644 squash2.txt
 create mode 100644 squash3.txt

$ git commit -a -m 'Squash!'
[master 104f2ff] Squash!
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 squash1.txt
 create mode 100644 squash2.txt
 create mode 100644 squash3.txt
```

—squash result

- When you use "git merge BRANCH —squash", you don't end up (yet) with a new commit
- Rather, you end up with a staged version of what you need to get the same results
- You still need to commit after squashing!
- If you don't include the -m option to "git commit", the messages from the squashed commits are provided

Another squashing method

- You can also tell Git that you want to merge the last four commits with the "rebase" command
- We'll return to "rebase" later. For now, consider that "rebase" means that you want to rewrite history
- In this case, we want to rewrite the four most recent commits, such that they're really part of a single commit

Squashing

- To squash the four most recent commits, say

```
git rebase -i HEAD~4
```

- -i means "interactive"
- HEAD~4 means "starting with the four commits before HEAD"
- This puts you into an editor, with one line per commit. Choose "pick" to include the commit, or "squash" to squash it into the previous commit.

In other words

- If you start with four commits:

```
pick 01d1124 Adding license

pick 6340aaa Moving license into its own file

pick ebfd367 Jekyll has become self-aware.

pick 30e0ccb Changed the tagline in the binary, too.
```

- You can then squash them into a single one:

```
pick 01d1124 Adding license

squash 6340aaa Moving license into its own file

squash ebfd367 Jekyll has become self-aware.

squash 30e0ccb Changed the tagline in the binary, too.
```

Cherry picking

- Sometimes, you want to bring only one or two commits from another branch
- You can do that with “cherry pick”
- Get the SHA1 of the commit(s) you want to merge
- Instead of “git merge”, do “git cherry-pick” with the SHA1 to merge in

Cherry picking

```
$ git cherry-pick c008427
[master 40b820b] cherry
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 cherry.txt

$ git log --pretty=oneline
40b820ba1d048f16a63982750cd640e8ead8bfbf cherry
104f2ff76e57076bd6cd3849965efd9e40109de8 Squash!
e17012bef6eff2332b6cef4331f79d92cdc64d44 Added files
22d010830febf7c85184908228e862fb17d2f7ca Revert "Updated to say bye"
```

Resolving conflicts

- Sometimes, a merge doesn't go so smoothly
- If Git can resolve a conflict itself, it will
- But if it cannot resolve a conflict, then it will ask you to resolve things
- The files will be marked with >> and << to show the problematic area

Set up the conflict

- In foo:

```
print "Hello, world"      # a comment
```

- In master:

```
print "Hello, world"      # a different comment
```

Conflict!

```
$ git merge foo

Auto-merging hello.rb

CONFLICT (content): Merge conflict in hello.rb

Automatic merge failed; fix conflicts and then commit the
result.
```

The file

```
<<<<<<< HEAD

print "Hello, world"     # a different comment

=======

print "Hello, world"      # comment

>>>>>>> foo
```

Fix as necessary, then commit

- Fix the file, so it doesn't have conflict marks
- Stage and add the file
- Voila! Conflict avoided

How to avoid conflicts

- Merge from your branches into master often
- And/or: Merge from master into your branches!
  - This gives you a chance to find and fix conflicts before they actually occur

Graphical merge

- If there are conflicts in your merge, and you have to resolve them, you can use a graphical client
- "git mergetool" tries to launch the best client that it can find on your system
- You can set your preferred tool in your config under "merge.tool" (i.e., section "merge", name "tool")

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
            araxis
            emerge
            opendiff
            vimdiff
            vimdiff2

      user-defined:
            sourcetree

The following tools are valid, but not currently available:
            bc3
            codecompare
            deltawalker
            diffmerge
            diffuse
            ecmerge
            gvimdiff
            gvimdiff2
            kdiff3
            meld
            p4merge
            tkdiff
            tortoisemerge
            xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

Using mergetool

- “git mergetool” is *not* used instead of “git merge”
- Rather, it’s used when there is a conflict with a merge, to help you resolve the conflict graphically.
- If there are conflicts, run “git mergetool”, and Git will feed the merge problems to your tool.
- Resolve in the tool, use “git add” to indicate resolution, and the merge continues.

mergetool on Windows

- If you like p4merge (from Perforce), say:

```
git config --global merge.tool p4merge

git config --global mergetool.p4merge.cmd
"p4merge.exe \"$BASE\" \"$LOCAL\"
\"$REMOTE\" \”$MERGED\""
```

- Notice that Git will pass a number of variables to the merge tool program

Merge problems?

- You can always stop a merge with

```
git merge --abort
```

- That returns you to the situation you were in before starting the merge

Rebasing

- Let's say you have a lot of commits in your branch, and you merge with another branch
- You can merge with "rebase," which rewinds the commits, applies those from the other branch, and then applies your local commits
- Good when working with other people
- Some people *always* rebase when pulling; this is a bit controversial

Using rebase

```
git checkout develop

git rebase OTHERBRANCH
```

Interactive rebase

- You can even invoke

```
git rebase otherbranch -i
```

- You can choose which commits are merged, and which are ignored!

When shouldn't you rebase?

"**Do not rebase commits that you have pushed to a public repository.** If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family." —

http://www.git-scm.com/book/en/Git-Branching-Rebasing

As a general rule

- Only change history *before* you push!
- Once you push, don't do anything that changes history

commit -a

- You don’t have to manually stage each file
- “git commit -a” stages all tracked files that have been modified
- So if you have made a bunch of changes to files already in the system, you don’t need to add and then commit each one

git commit -a

```
$ git commit -a -m 'Added a print statement'
[master ce0e1f3] Added a print statement
 1 files changed, 1 insertions(+), 0 deletions(-)
```