

- Labs
- <https://github.com/brentlaster/conf/blob/master/oscon2018/pg-labs-2018.pdf>
- GitHub account (github.com)
- Recent version of Git
 - <http://git-scm.com/download/<platform>>



Attendee Networking at OSCON

3

- Introduce yourself to the person next to you
- Say what company or organization you're with
- What you're looking to learn from this tutorial
- 2 to 3 minutes

Power Git: Rerere, Bisect, Subtrees, Filter Branch, Worktrees, Submodules, and more

Brent Laster (author of Professional Git)

About me

- Senior Manager, R&D
- Part-time trainer
- Git, Gerrit, Gradle, Jenkins
- Author - O'Reilly Media, Professional Git book, Jenkins 2 book, OpenSource.com
- <https://www.linkedin.com/in/brentlaster>
- @BrentCLaster

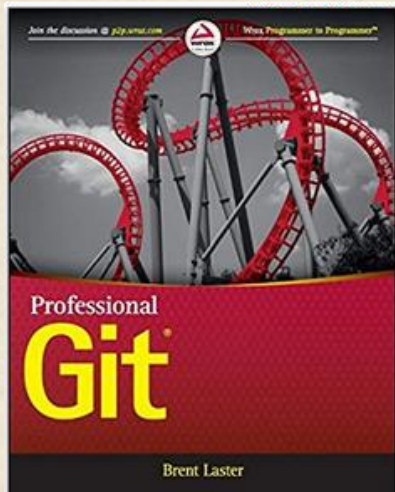
Professional Git 1st Edition

by Brent Laster (Author)



7 customer reviews

Look inside ↴

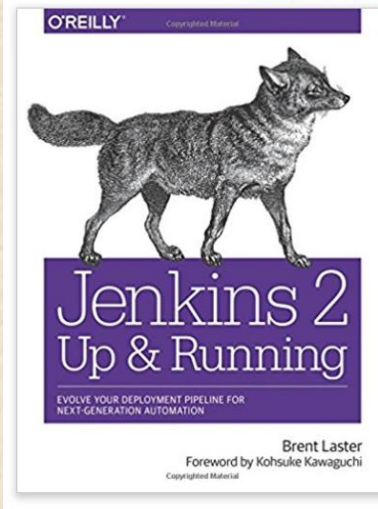


by Brent Laster (Author)



5 customer reviews

Look inside ↴



Book – Professional Git

6

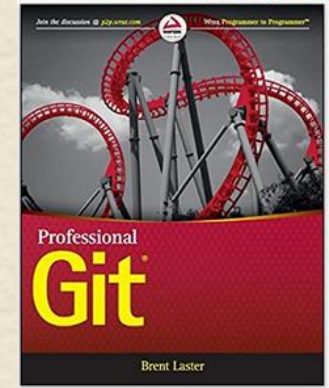
- Additional reference for this material and much more
- First part for non-technical
- Beginner and advanced reference
- More labs
- Please leave a review!

Professional Git 1st Edition

by Brent Laster * (Author)

★★★★★ 7 customer reviews

[Look inside ↴](#)



Amazon Customer

★★★★★ I can't recommend this book more highly

February 12, 2017

Format: Kindle Edition

Brent Laster's book is in a different league from the many print and video sources that I've looked at in my attempt to learn Git. The book is extremely well organised and very clearly written. His decision to focus on Git as a local application for the first several chapters, and to defer discussion about it as a remote application until later in the book, works extremely well.

Laster has also succeeded in writing a book that should work for both beginners and people with a fair bit of experience with Git. He accomplishes this by offering, in each chapter, a core discussion followed by more advanced material and practical exercises.

I can't recommend this book more highly.

★★★★★ Ideal for hands-on reading and experimentation

February 23, 2017

Format: Paperback | [Verified Purchase](#)

I just finished reading Professional Git, which is well organized and clearly presented. It works as both a tutorial for newcomers and a reference book for those more experienced. I found it ideal for hands-on reading and experimentation with things you may not understand at first glance. I was already familiar with Git for everyday use, but I've always stuck with a convenient subset. It was great to be able to finally get a much deeper understanding. I highly recommend the book.

6

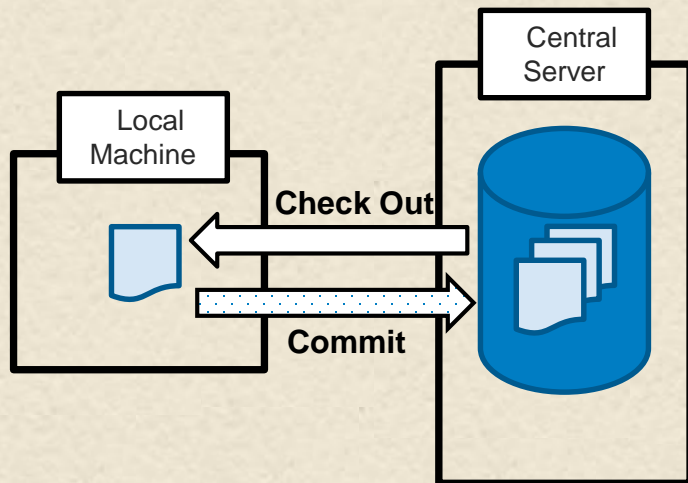


Agenda

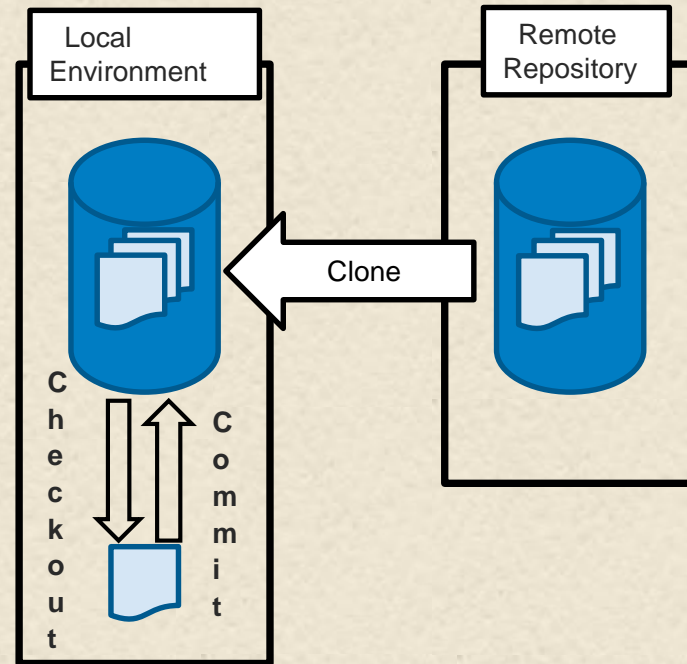
- Core concepts refresh
- Merging and Rebasing
- Stash
- Reset and Revert
- Rerere
- Bisect
- Worktrees
- Submodules
- Subtrees
- Filter Branch
- Interactive Rebase
- Notes
- Grep

Centralized vs. Distributed VCS

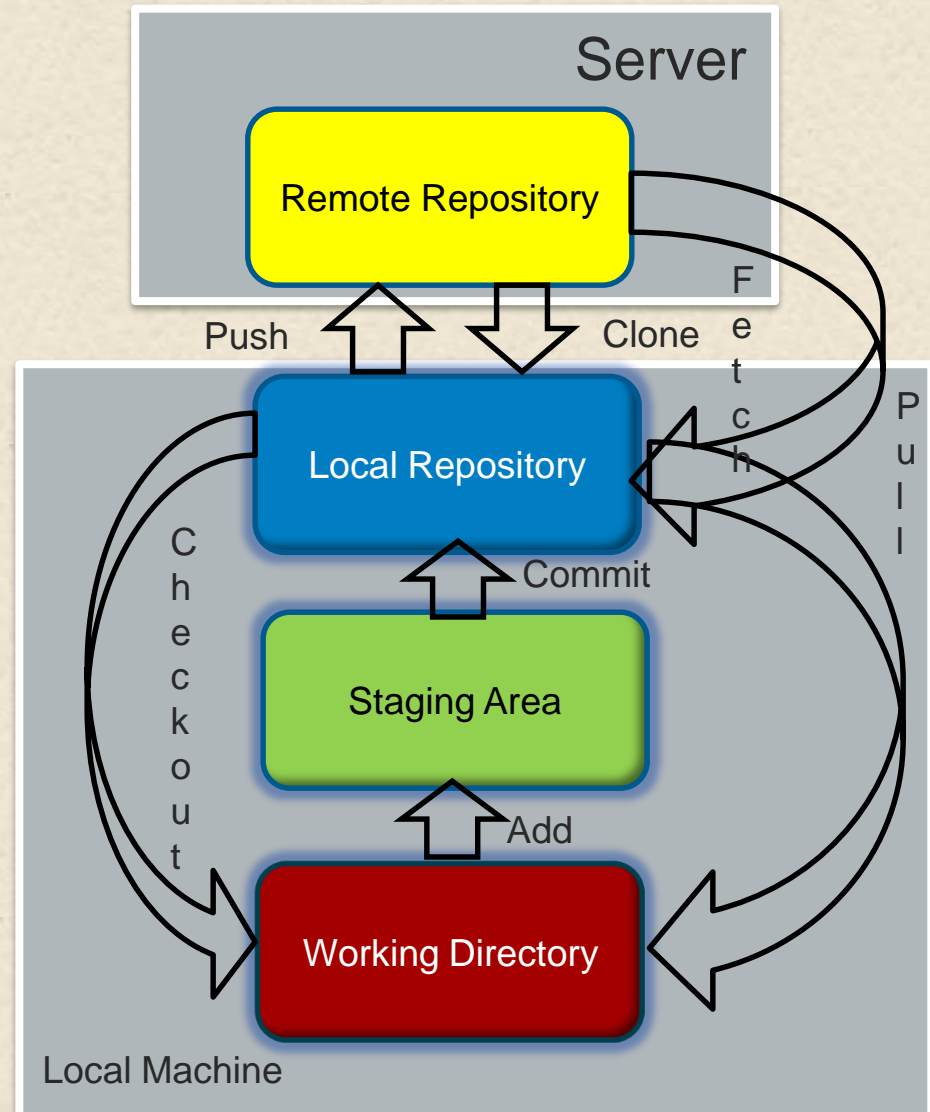
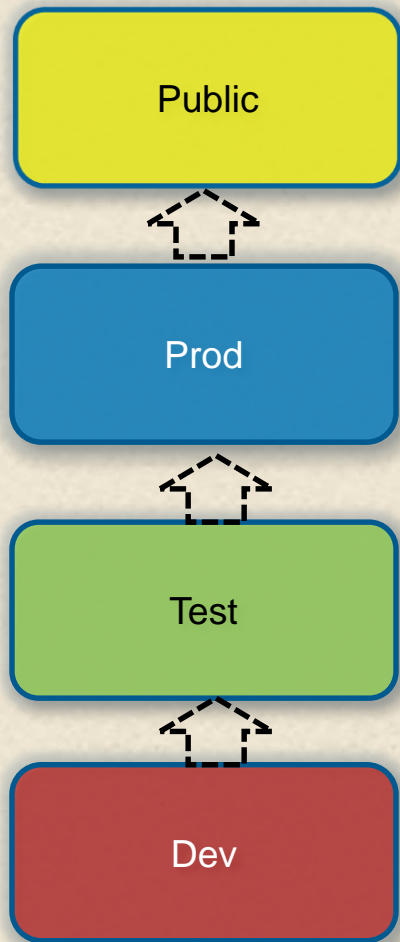
Centralized Version Control Model



Distributed Version Control Model



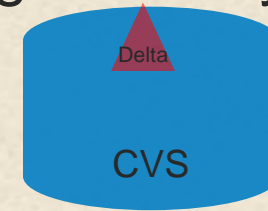
Git in One Picture



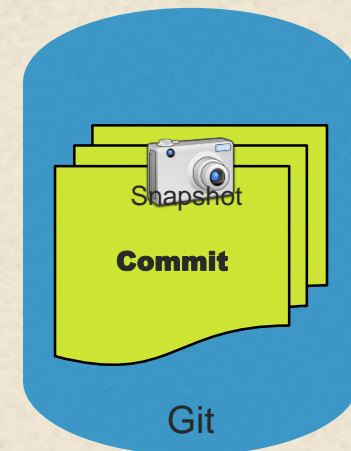
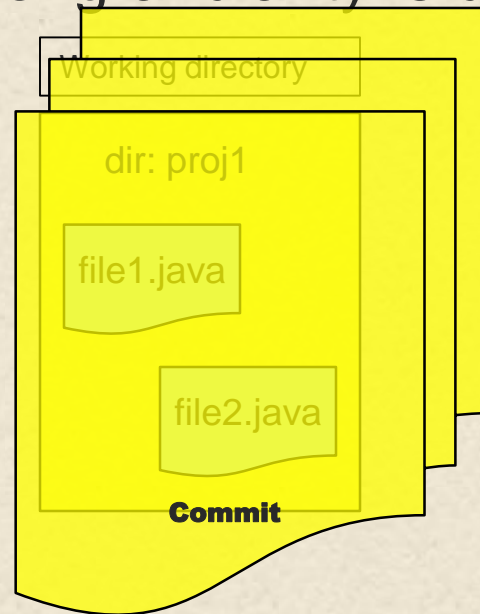
Git Granularity (What is a unit?)

10

- In traditional source control, the unit of granularity is usually a file



- In Git, the unit of granularity is usually a tree



Merging: What is a Fast-forward?

11

- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)

```
$ git checkout master
```

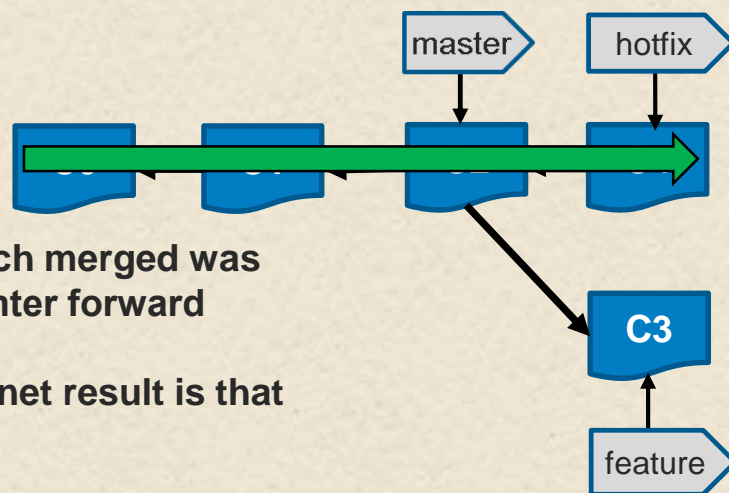
```
$ git merge hotfix
```

```
Updating f42c576..3a0874c
```

```
Fast Forward
```

```
README | 1-
```

```
1 files changed, 0 insertions(+) 1 deletions (-)
```



About “Fast Forward” – because commit pointed to by branch merged was directly “upstream” of the current commit, Git moves the pointer forward

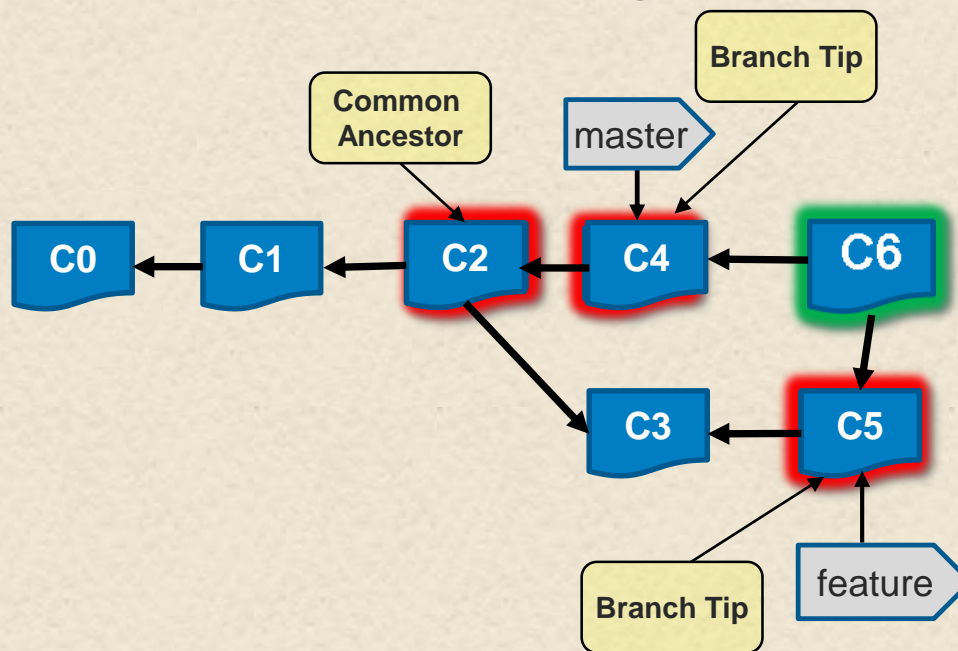
(Both branches were in the same line of development, so the net result is that master and hotfix point to the same commit)

11

Merging: What is a 3-way Merge?

12

- Assume branching scenario below
 - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- Intent is to change to master and merge in feature
- Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)
- Git does 3-way merge using common ancestor
- Instead of just moving branch pointer forward, Git creates a new snapshot and a new commit that points to it called a "merge commit" (two parents)

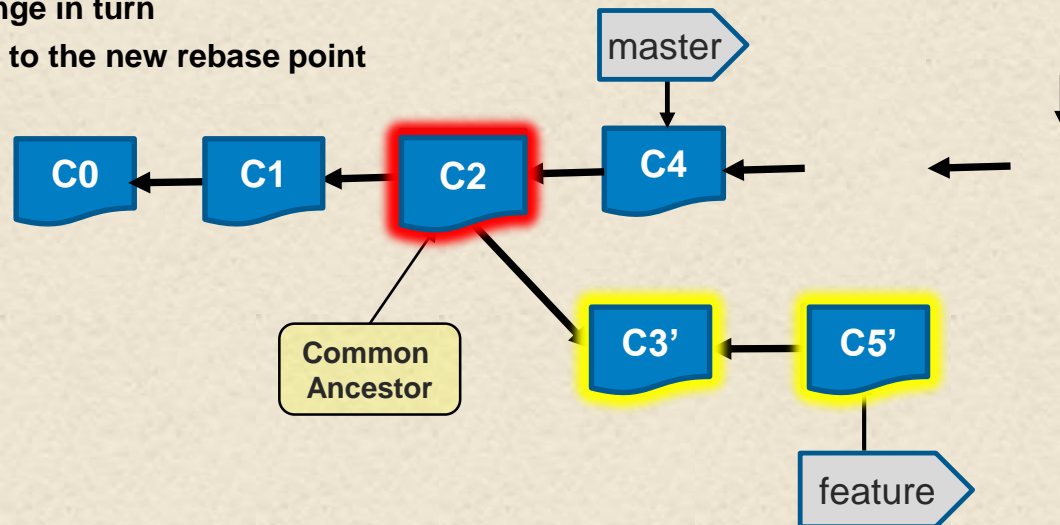


\$ git checkout master
\$ git merge feature

Merging: What is a Rebase?

13

- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (**git rebase branch2 [branch1]**):
 - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
 - Current branch (or branch1 if supplied) is one having its branchpoint moved
 - Branch2 (or commit) provided as first/only argument is the new branchpoint
 - In simple case, think of it as “pick up branch1 entirely and move its branchpoint to be after the tip of branch2)”
- Process:
 - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
 - Gets the diff introduced by each commit of the branch you are on, saving them to temporary files
 - Applies each change in turn
 - Moves the branch to the new rebase point



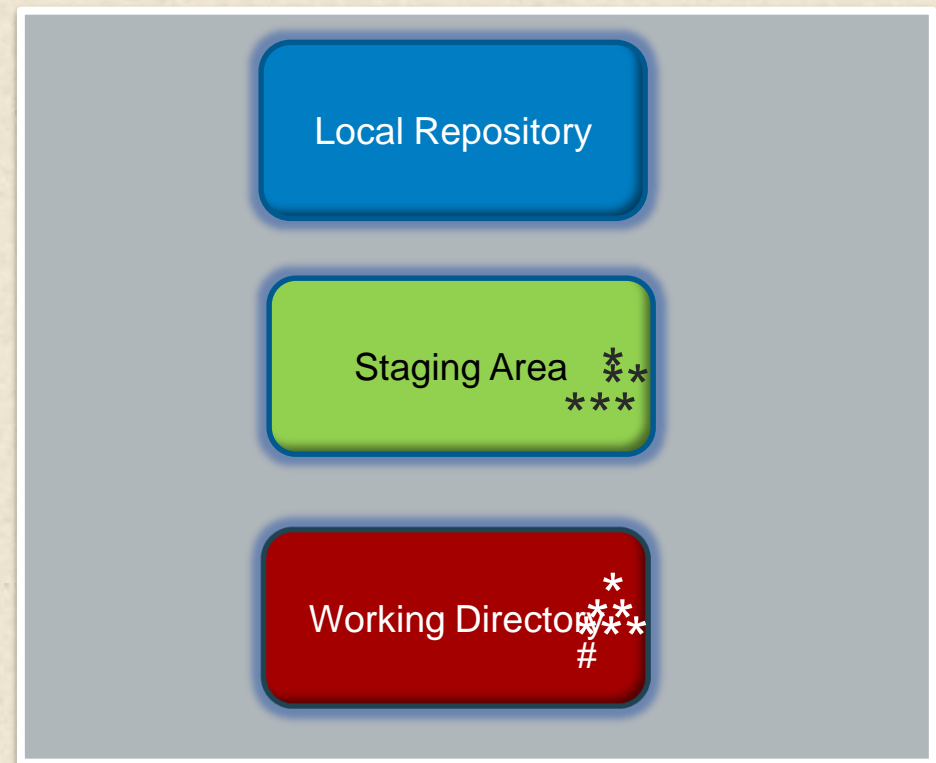
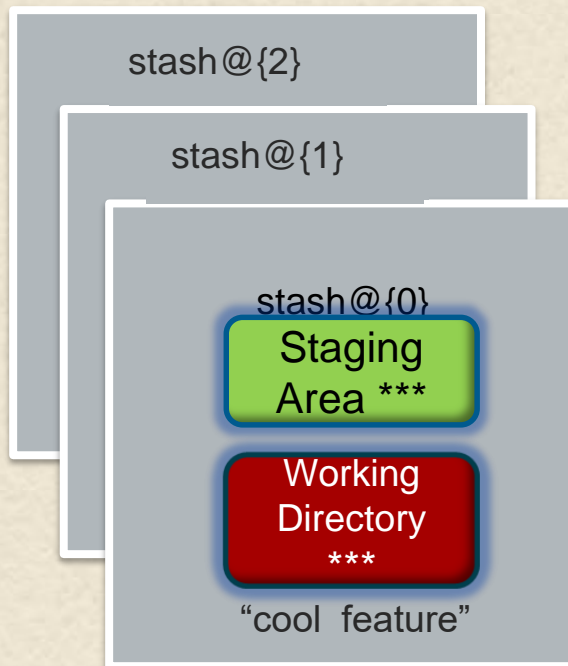
\$ git checkout feature
\$ git rebase master

- Purpose -- allow you to keep a backup copy of your work that hasn't been committed yet
- Use case - you want to switch branches but don't want to lose work that hasn't been committed; you want to save something you've tried and may want to come back to

- Syntax:
 - git stash list* [<options>]
 - git stash show* [<stash>]
 - git stash drop* [-q|--quiet] [<stash>]
 - git stash* (pop | apply) [--index] [-q|--quiet] [<stash>]
 - git stash branch* <branchname> [<stash>]
 - git stash* [push [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet] [-u|--include-untracked] [-a|--all] [-m|--message <message>] [--] [<pathspec>...]]
 - git stash clear*
 - git stash create* [<message>]
 - git stash store* [-m|--message <message>] [-q|--quiet] <commit>

- push (newer) is like save but allows pathspecs
- --keep-index doesn't clean out staging area
- Also separate “git clean” command to remove untracked files/dirs

```
$ git stash pop --keep-index
```



Command: Git Reset

16

- Purpose -- allow you to “roll back” so that your branch points at a previous commit ; optionally also update staging area and working directory to that commit
- Use case - you want to update your local environment back to a previous point in time; you want to overwrite or a local change you’ve made
- Syntax:

```
git reset [-q] [<tree-ish>] [--] <paths>...
```

```
git reset (--patch | -p) [<tree-ish>] [--] [<paths>...]
```

```
EXPERIMENTAL: git reset [-q] [--stdin [-z]] [<tree-ish>]
```

```
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

- **Warning: --hard overwrites everything**

Command: Git Revert

17

- Purpose -- allow you to “undo” by adding a new change that cancels out effects of previous one
- Use case - you want to cancel out a previous change but not roll things back
- Syntax:

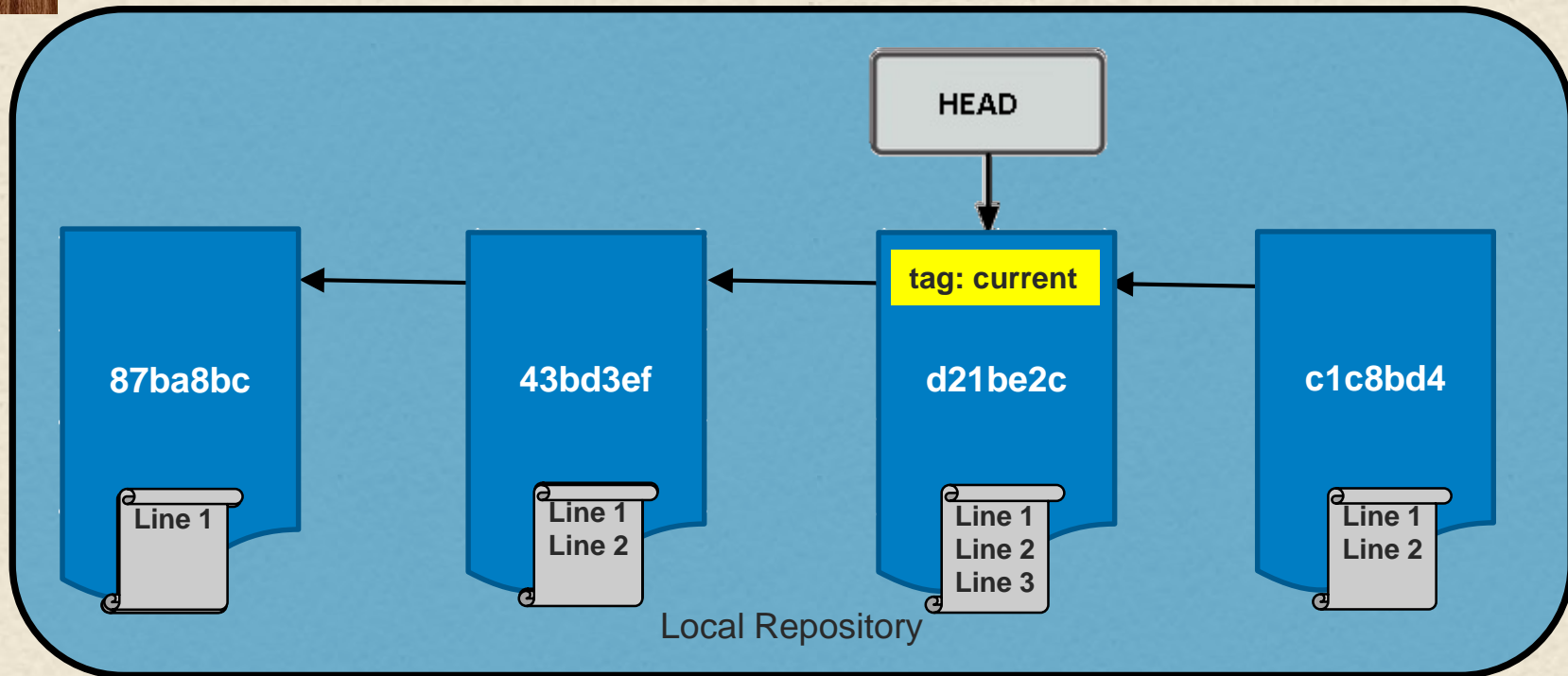
```
git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>...
```

```
git revert --continue
```

```
git revert --quit
```

```
git revert --abort
```

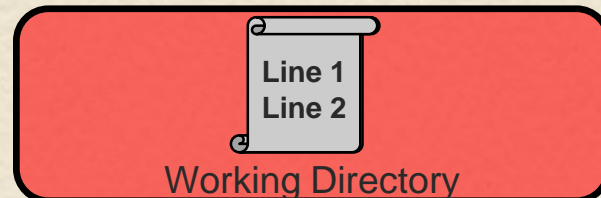
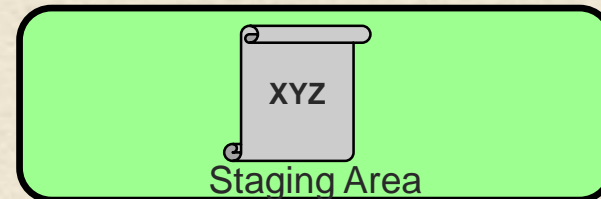
- Note: The net result of using this command vs. reset can be the same. If so, and content that is being reset/revert has been pushed such that others may be consuming it, preference is for revert.



`git reset --hard 87ba8bc`

`git reset current~1 [--mixed]`

`git revert HEAD`



Command: Git Rerere (Reuse Recorded Resolution)

19

- Purpose -- allows recording of how you solve a merge situation and then can automatically resolve the same situation in the same way if needed later
- Use case - trial and repeated merges; merging newer versions of a branch with the same conflicts into a newer one periodically; resolve conflicts after reset or revert; applicable to any repeated merge case: rebase, merge
- Syntax: `git rerere [clear|forget <pathspec>|diff|remaining|status|gc]`
- Note: This is a “state” command. Enabled by turning on a state in Git, rather than just running a command
 - Enabled via `git config --global rerere.enabled 1`
 - » Then runs automatically
 - Invoked directly via `git rerere` for related commands or options

- You want to make sure a long lived topic branch will merge cleanly but don't want to have a bunch of intermediate merge commits.
 - With rerere turned on you can merge occasionally, resolve the conflicts, then back out the merge – or merge to a “throw-away branch”.
 - If done regularly, final merge should be easy because rerere can just do everything automatically.
- You want to keep a branch rebased so you don't have to deal with the same rebasing conflicts each time you do it. (rerere works with rebase too).
- You want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead - you likely won't have to do all the same conflicts again.
- You merge a collection of evolving topic branches together into a testable mainline occasionally.
 - If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

Git rerere – Tips and Tricks

21

- You can also activate git rerere by creating the expected repository directory.
 - `mkdir .git/rr-cache`
- `git rerere status`
 - provides status on what git rerere has been working with
- `git rerere diff`
 - show the current state of the resolution
 - what you started with to resolve and what you've resolved it to.
- You can tell git to forget a recorded resolution
 - `git rerere forget <pathspec>`
 - This resets the conflict resolutions which rerere has recorded for the current conflict in `<pathspec>`.
- You can tell rerere to also update your index
 - `set git config --global rerere.autoupdate true`
- You can tell git merge to not apply rerere for a particular case
 - `git merge --no-rerere-autoupdate`
- You can share your resolves with others by providing access to what's in `.git/rr-cache`

21

Git Rerere 1

```
$ git checkout master  
$ git config --global rerere.enabled 1  
$ git merge feature
```

Recorded preimage for 'File B'

```
$ git status
```

Changes to be committed:

modified: File A

modified: File C

Unmerged paths

both modified: File B

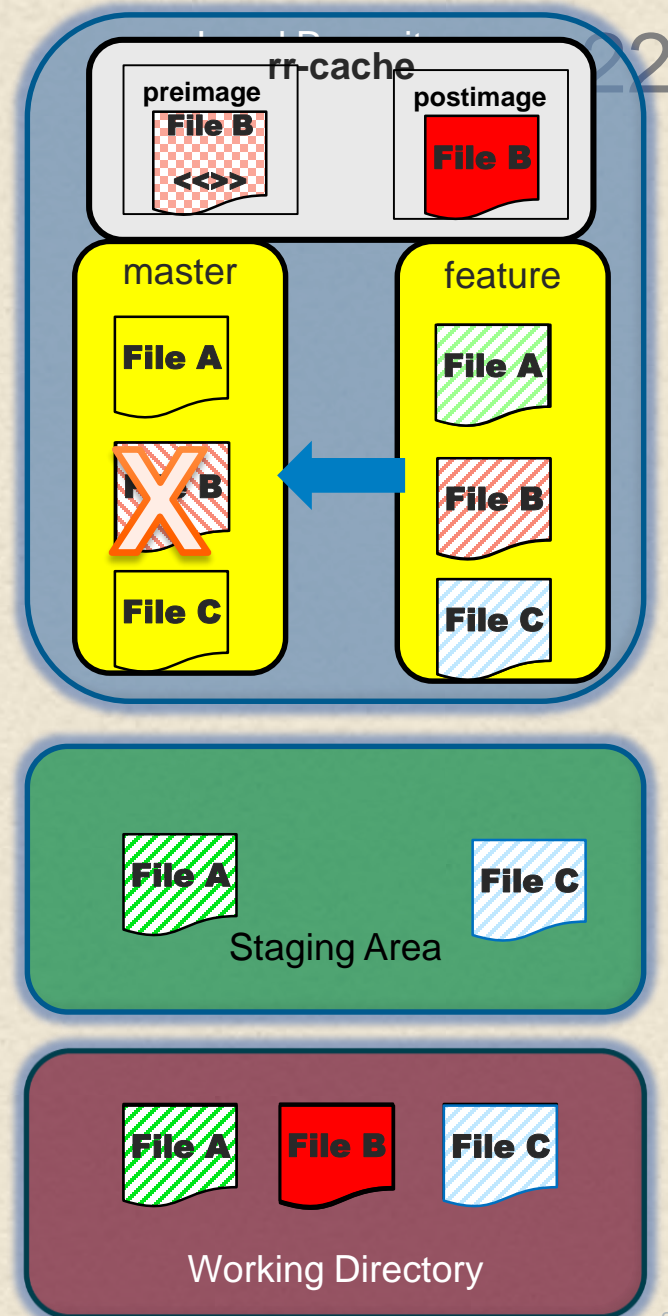
```
[fix conflicts]
```

```
$ git add .
```

```
$ git commit -m
```

```
"finalize merge"
```

Recorded resolution for 'File B'



Git Rerere 2

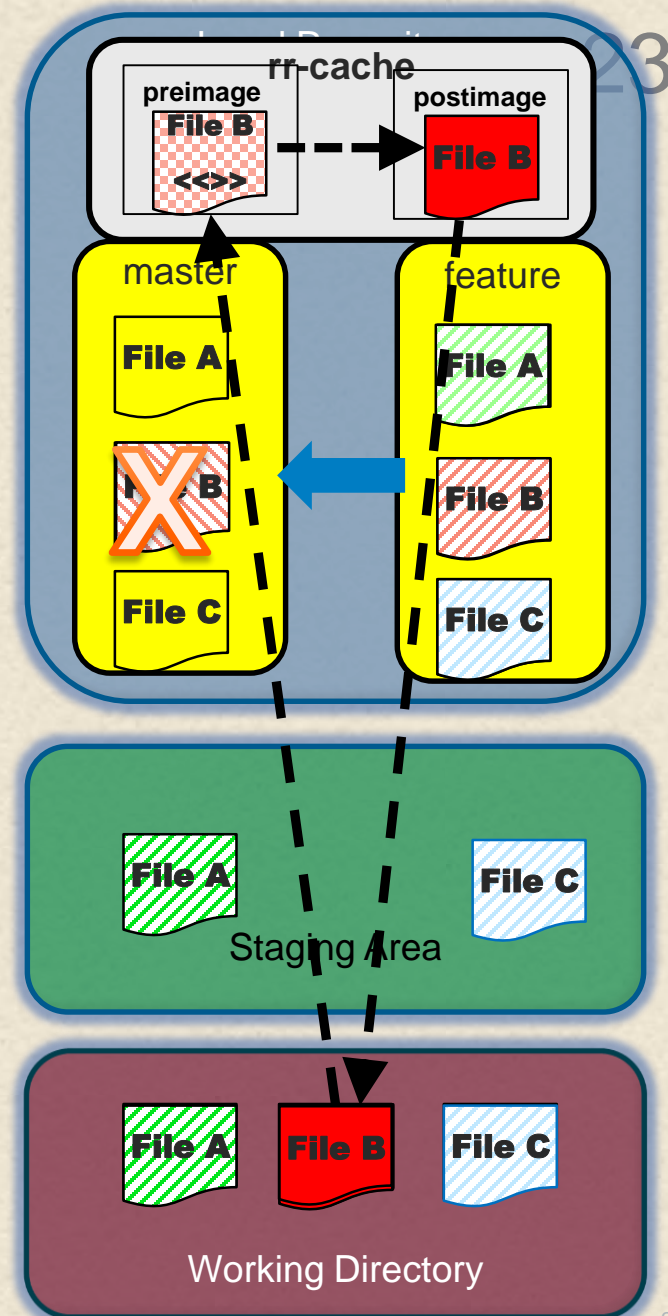
```
$ git reset --hard HEAD~1
```

```
$ git checkout master
```

```
$ git merge feature
```

Merge conflict in 'File B'

Resolved 'File B' using previous resolution



Temporary Checkout

- Can create a “temporary checkout” with command like this:

```
$ git checkout HEAD^0
```

- Puts you into “detached head” state – not on a branch
- Can be useful for temporary activities, trying things out, etc.
- Can create a branch off of it with

```
$ git checkout -b <branch-name>
```


Lab 1 – Using rerere

GitHub project: brentlaster/greetings

<https://github.com/brentlaster/greetings>

- Purpose - Use “automated” binary search through Git’s history to find a specific commit that first introduced a problem (i.e. “first bad commit”)
- Use case - Quickly locate the commit in Git’s history that introduced a bug
- Syntax:

```
git bisect start [--term-{old,good}=<term> --term-{new,bad}=<term>]
                [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new) [<rev>]
git bisect (good|old) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect visualize
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
git bisect help
```

- Use “automated” binary search to find change that first introduced a bug (i.e. “first bad commit”)
- Initiate with `git bisect start`
- Can pass range to start option to identify bad and good revisions – i.e.
`git bisect start HEAD HEAD~10`
- Identify first “bad” commit, usually current one via `git bisect bad`
- Identify a “good” (functioning) commit via `git bisect good`
- From there, git will do a binary search and pick a “middle” commit to checkout.
- Try the checked out version and indicate `git bisect good` or `git bisect bad` depending on whether it works or not.
- Process repeats until git can identify “first bad” commit/revision.
- Can grab previous good revision by specifying “first bad”^
- Can update, create new branch, rebase, etc. to isolate good revisions.
- Other useful options to git bisect: `log`, `visualize`, `reset`, `skip`, `run`

Bisect

28



FIRST BAD COMMIT

- checkout latest version
- try code
- `git bisect start`
- `git bisect bad`
- checkout earlier version (user checks out)
- try code
- `git bisect good` (bisect checks out version 5)
- try code
- `git bisect good` (bisect checks out version 7)
- try code
- `git bisect bad` (bisect checks out version 6)
- try code
- `git bisect bad` (git reports version 6 as the first bad commit)

WORKING DIRECTORY

Exiting or Redoing Bisect

- Stopping bisect

\$ git bisect reset

- Recovering from an incorrect bisect

\$ git bisect log > filename

- Edit filename to correct bad/good

\$ git bisect reset

\$ git bisect replay filename

Lab 2 – Using the Git bisect command

GitHub project: brentlaster/bisect

<https://github.com/brentlaster/bisect>

Background: Switching between Branches³¹

Command: `git checkout <branch>`
`git checkout master`

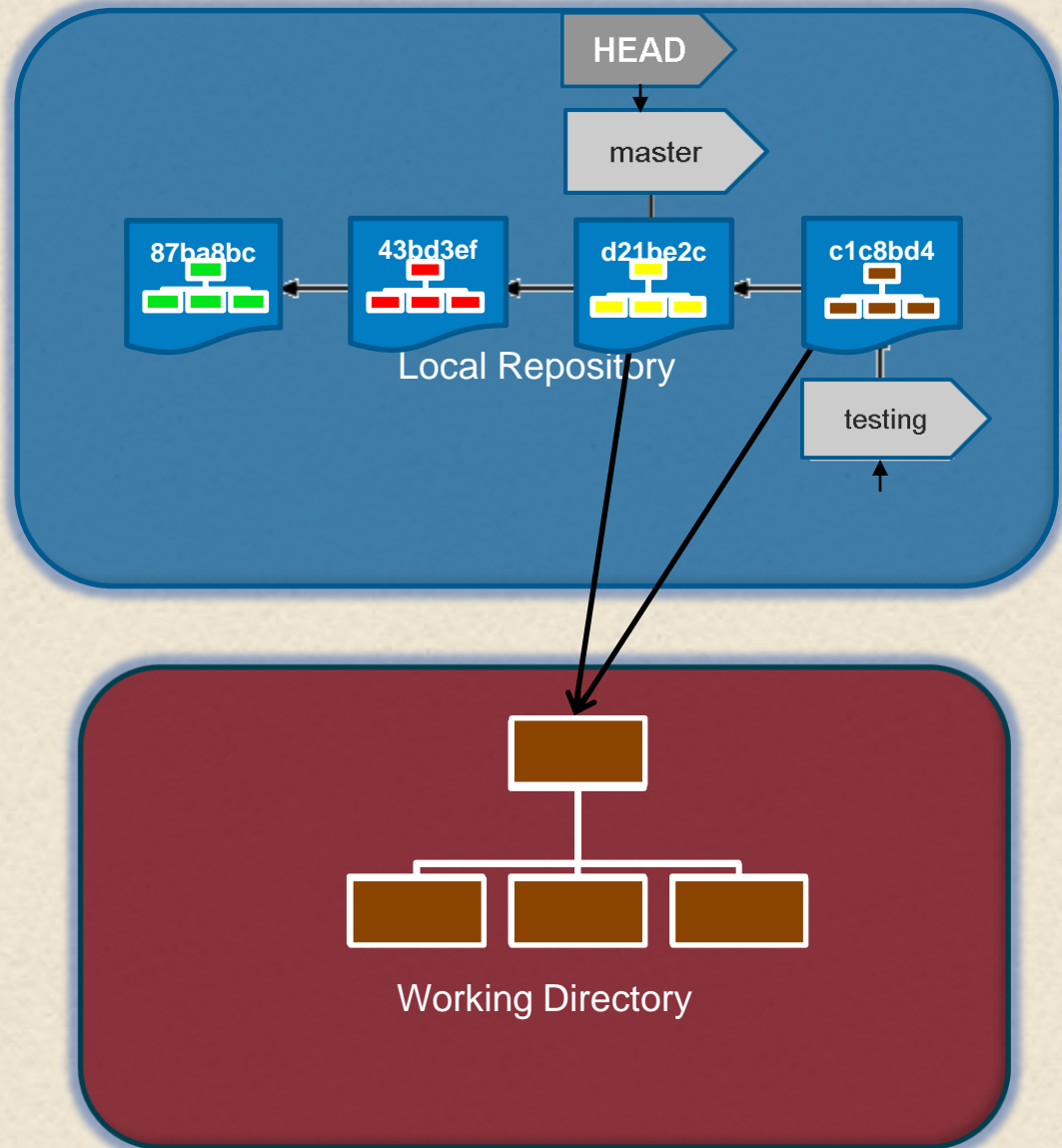
- **Does three things**

- Moves HEAD pointer back to <branch>
- Reverts files in working directory to snapshot pointed to by <branch>
- Updates indicators

`git checkout testing`
`git checkout master`
`git checkout testing`

- **git branch**

- * master
- * testing



Command: Worktrees

32

- Purpose - Allows multiple, separate Working Areas attached to one Local Repository
- Use case - Simultaneous development in multiple branches
- Syntax
 - `git worktree add [-f] [--detach] [-b <new-branch>] <path> [<branch>]`
 - `git worktree list [--porcelain]`
 - `git worktree lock [--reason <string>] <worktree>`
 - `git worktree move <worktree> <new-path>`
 - `git worktree remove [-f] <worktree>`
 - `git worktree prune [-n] [-v] [--expire <expire>]`
 - `git worktree unlock <worktree>`
- Notes
 - “Traditional” working directory is *main working tree*; Any new trees you create with this command are called *linked working trees*
 - Information about working trees is stored in the .git area (assuming .git default GIT_DIR is used)
 - Working tree information is stored in `.git/worktrees/<name of worktree>`.

32

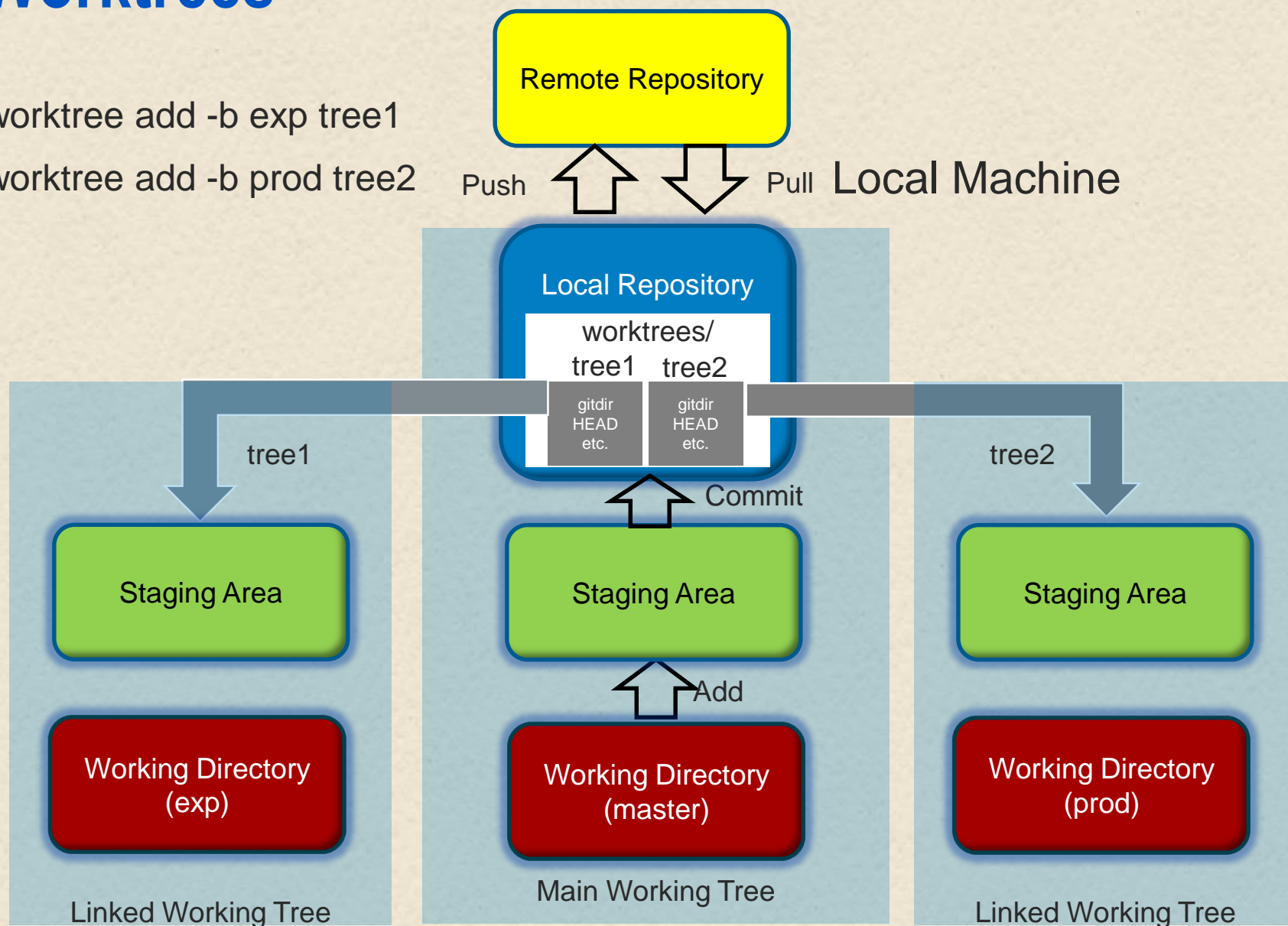
- Subcommands
 - Add - create a separate working tree for specified branch
 - » has checked out copy of the branch
 - » if no branch specified, uses same name as temp area
 - List - lists out current set of working trees active for this repo
 - » porcelain - consistent and backwards-compatible format
 - Remove – removes worktree structure (only if clean – no modified and no untracked)
 - » force – forces removal even if not clean
 - Prune - removes worktree information from the .git area
 - » only applies after worktree directory tree has been removed
 - Lock – if worktree is on removable device/network share then lock administrative files to keep them from being automatically pruned if worktree is not available
 - Unlock – unlock administrative files

Worktrees

Server

34

- `git worktree add -b exp tree1`
- `git worktree add -b prod tree2`



Lab 3 – Working with worktrees

GitHub projects: super_calc, sub_ui,
sub_docs

https://github.com/brentlaster/super_calc

https://github.com/brentlaster/sub_ui

https://github.com/brentlaster/sub_docs

- **Purpose** - Allows including a separate repository with your current repository
- Use case - include the Git repository for one or more dependencies along with the original repository for a project

- **Syntax**

```
git submodule [--quiet] add [-b <branch>] [-f|--force] [--name <name>]
    [--reference <repository>] [--depth <depth>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>...]
git submodule [--quiet] init [--] [<path>...]
git submodule [--quiet] deinit [-f|--force] [--] <path>...
git submodule [--quiet] update [--init] [--remote] [-N|--no-fetch]
    [-f|--force] [--rebase|--merge] [--reference <repository>]
    [--depth <depth>] [--recursive] [--] [<path>...]
git submodule [--quiet] summary [--cached|--files] [(-n|--summary-limit) <n>]
    [commit] [--] [<path>...]
...
```

- **Notes**

- Creates a subdirectory off of original repository that contains clone of another Git repo
- Original repository is typically called *superproject*
- Metadata stored in *.gitmodules* file

Submodules 1

What happens when you add a **submodule reference**?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
    path = mod1
    url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

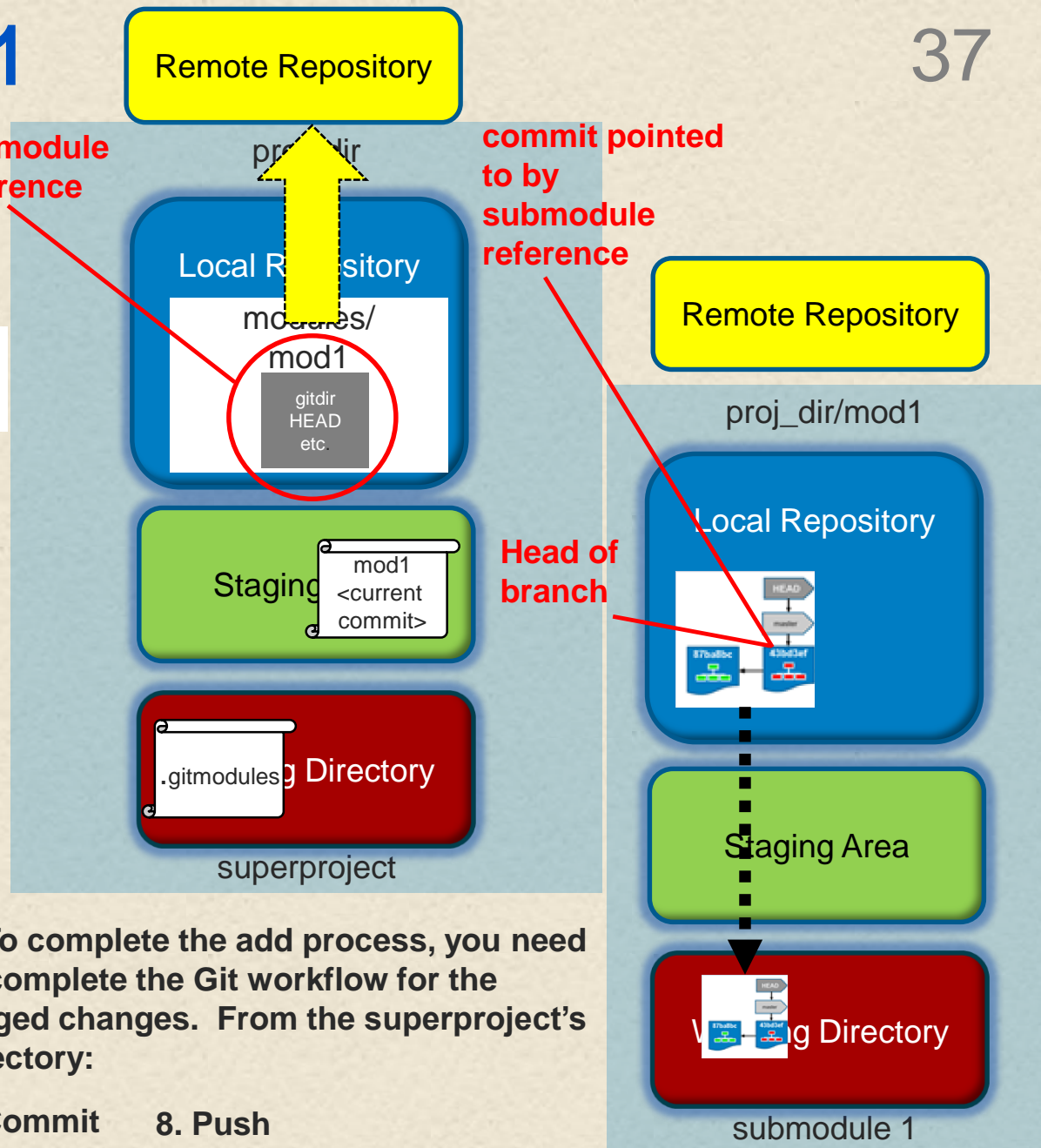
5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
...
    new file:   .gitmodules
    new file:   mod1
```

6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit

8. Push



Submodules 2

38

How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.
```

```
$ cd main
```

```
$ ls -a
./ ../ .git/ .gitmodules file1.txt mod1/
```

2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
```

3. git submodule update - actually clones repositories for submodules into the corresponding subdirectories and checks out the indicated commits for the containing project

```
$ git submodule update
Cloning into 'mod1'...
done.
```

```
Submodule path 'mod1': checked out '8add7dab652c856b65770bca867db2bbb39c0d00'
```

submodule
reference

commit pointed
to by
submodule
reference

Head of
branch

Note: Shortcuts -
git submodule update --init and
git clone --recursive or
--recurse-submodules

Remote Repository

main

Local Repository

modules/
mod1

gitdir
HEAD
etc.

Staging Area

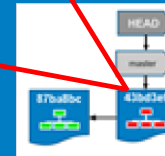
Working Directory

superproject

Remote Repository

main/mod1

Local Repository



Staging Area



Working Directory

submodule 1

Submodules 3

39

Incorporating updates to submodules:

1. You can
`$ cd mod1; git checkout <branch>; git pull`

OR

`$ git pull --recurse-submodules; cd <module dir>; git merge origin/master`

OR

`$ git submodule update --remote`

2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   mod1 (new commits)

Submodules changed but not updated:
  * mod1 8add7da...d05eb00 (2):
    > third update
    > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   mod1

Submodule changes to be committed:
  * mod1 8add7da...d05eb00 (2):
    > third update
    > update info file
```

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
1 files changed, 1 insertions(+), 1 deletions(-)
```

```
$ git push
Counting objects: 1, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1/1), done.
Writing objects: 100% (1/1), 338 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To <remote path>/main.git
 2745a27..7e4e525 master -> master
```

Remote Repository

Local Repository

modules/
mod1
gitdir
HEAD
etc.

.gitmodules

mod1
<current commit>

Working Directory

superproject

commit pointed
to by
submodule
reference

new commit

Remote Repository

main/mod1

Local Repository

Head of
branch

Staging Area

Repository

submodule 1

- Essentially, you can map out the process of dealing with a merge commit in a submodule as follows:
 1. Change into the submodule and resolve the merge in the most appropriate way.
 2. Change back to the superproject.
 3. Verify that the expected values of the submodule updates match the superproject's references.
 4. Stage (add) the updated submodule reference.
 5. Commit to finish the merge.

- Challenges around using submodules nearly always involve keeping submodule content (and “current” commit) in sync with submodule references in superproject
- If references are wrong, operations like “git submodule update” will backlevel submodule content to commits in reference
- If these references are out of sync and that inconsistency is pushed to the remote for the superproject, then other users that pull that version of the superproject can end up back-leveling their submodules, even if they’ve updated their superproject before.

- Shows status of submodules – relative to superproject

\$ git submodule status

- Prints out SHA-1 of currently checked-out commit
- Output can be prefixed with one of the following:
 - - to indicate submodule has not been initialized
 - <space> to indicate initialized and in-sync
 - + to indicate difference in current SHA-1 in submodule vs. what the superproject knows about
 - U to indicate submodule has merge conflicts
- Examples:
 - **\$ git submodule status**
-abaf124e483d8d8bfccae22923c67a794ec6fb66 sub_ui not initialized
 - **\$ git submodule status**
+1d169806636a7d3684984ab3d8641ee15ef37577 sub_ui (heads/master-1-g1d16980) updates

Lab 4 – Working with submodules

GitHub projects: super_calc, sub_ui,
sub_docs

https://github.com/<github id>/super_calc

https://github.com/<github id>/sub_ui

https://github.com/<github id>/sub_docs

- What is it?
 - Step 10: (in submodule) `$ git commit -am "Update title"`
 - Step 14: (in superproject) `$ git commit -am "Update for submodule sub_ui";`
`$ git push`
 - No push of update in submodule, but push of reference to new commit in superproject

```
$ git clone --recurse-submodules https://github.com/gwstudent2/super_calc super_calc3
Cloning into 'super_calc3'...
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 20 (delta 0), reused 2 (delta 0), pack-reused 17
Unpacking objects: 100% (20/20), done.
Submodule 'sub_ui' (https://github.com/gwstudent2/sub_ui) registered for path 'sub_ui'
Cloning into 'C:/Users/sasbcl/test-labs/super_calc3/sub_ui'...
remote: Counting objects: 9, done.
remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 9
error: Server does not allow request for unadvertised object f3a5a299d2f509a1dcb1370043142b21d99fc55e
Fetched in submodule path 'sub_ui', but it did not contain f3a5a299d2f509a1dcb1370043142b21d99fc55e. Direct fetching of that commit failed.
```

- Purpose - Allows including **a copy** of a separate repository with your current repository
- Use case - include **a copy** of a Git repository for one or more dependencies along with the original repository for a project

- Syntax

```
git subtree add -P <prefix> <commit>
git subtree add -P <prefix> <repository> <ref>
git subtree pull -P <prefix> <repository> <ref>
git subtree push -P <prefix> <repository> <ref>
git subtree merge -P <prefix> <commit>
git subtree split -P <prefix> [OPTIONS] [<commit>]
```

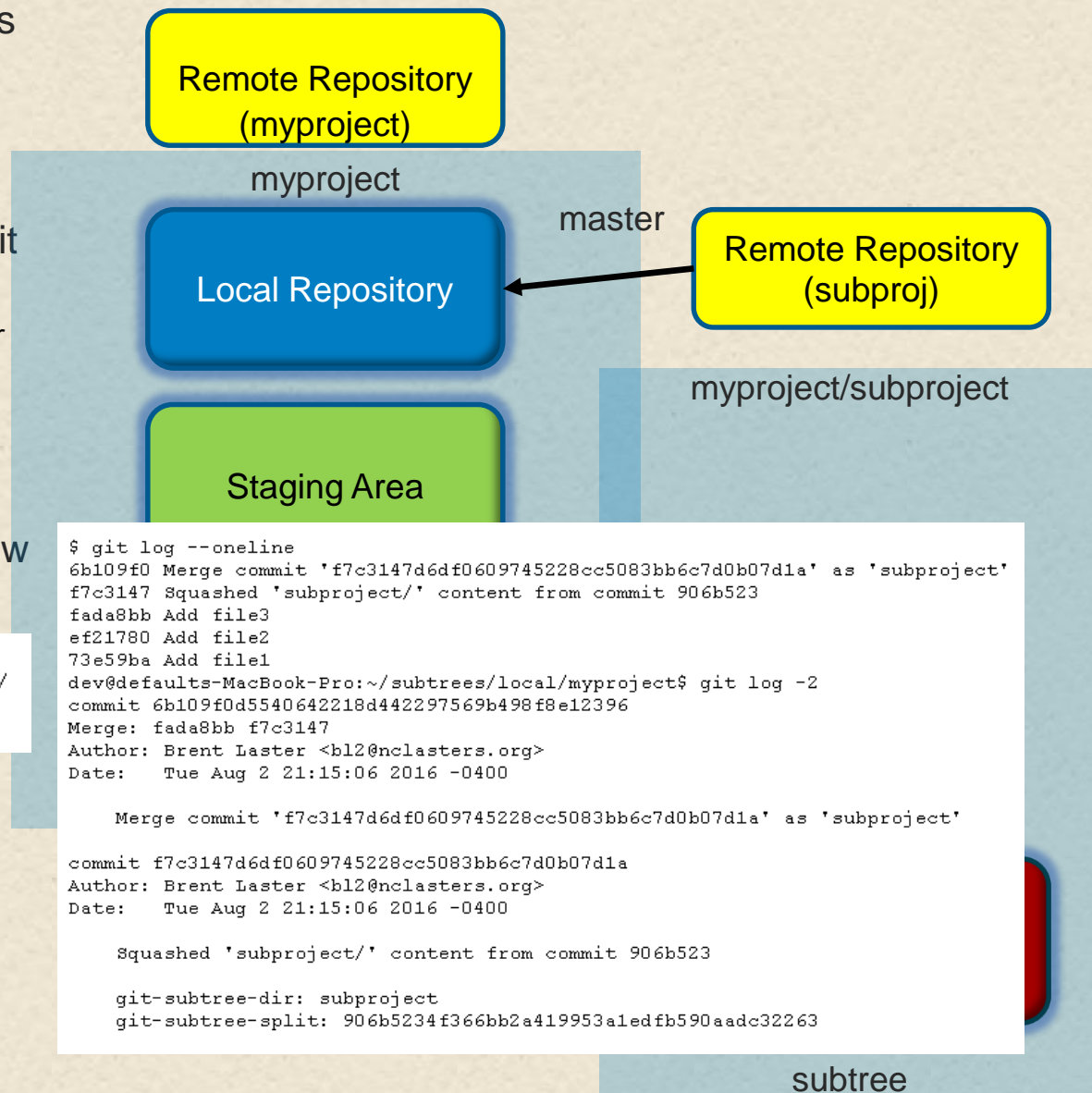
- Notes

- No links like a submodule - just a copy in a subdirectory
- Advantage - no links to maintain
- Disadvantage - extra content to carry around with your project

Subtrees

48

- Adding a copy from a remote as a subtree
- `cd myproject`
- `git subtree add --prefix subproject --squash subproj.git master`
 - Use “prefix” option to specify path for subproject
 - Use “squash” option to compress history from remote before adding it
 - branch (master)
- directory listing of myproject now shows subproj as subdirectory



```
~/subtrees/local/myproject$ ls
file1.txt file2.txt file3.txt subproject/
~/subtrees/local/myproject$ ls subproject
subfile1.txt subfile2.txt
```

- Looking in the logs of the subproject will show the squashed history
- To get the latest, use pull
- `git subtree pull --prefix subproject --squash subproj.git master`

Subtrees - tips for adding

50

- Can create a remote to simplify pointing to remote to be added as a subtree

```
~/subtrees/local/myproject$ git remote add sub_origin ~/subtrees/remotes/subproj.git
```

- Use “squash” option to compress history from remote before adding it

```
~/subtrees/local/myproject$ git subtree add --prefix subproject --squash \
sub_origin master
```

- History will now show “squashed” reference in record for history

```
$ git log --oneline
6b109f0 Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'
f7c3147 Squashed 'subproject/' content from commit 906b523
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
dev@defaults-MacBook-Pro:~/subtrees/local/myproject$ git log -2
commit 6b109f0d5540642218d442297569b498f8e12396
Merge: fada8bb f7c3147
Author: Brent Laster <bl2@nclasters.org>
Date: Tue Aug 2 21:15:06 2016 -0400

    Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'

commit f7c3147d6df0609745228cc5083bb6c7d0b07d1a
Author: Brent Laster <bl2@nclasters.org>
Date: Tue Aug 2 21:15:06 2016 -0400

    Squashed 'subproject/' content from commit 906b523

git-subtree-dir: subproject
git-subtree-split: 906b5234f366bb2a419953aledfb590aad32263
```

50

Subtrees - updating and merging

51

- Update command - similar to add

```
$ git subtree pull --prefix subproject sub_origin master --squash
```

- pulls down the latest content from the remote into the subtree area
- --squash compresses the history again
 - can be omitted, but usually simplifies things
- also a *git subtree merge* command to merge commits up to a desired point into a subproject denoted by the --prefix argument
 - can be used to merge local changes to a subproject, while git subtree pull reaches out to the remote to get changes
 - different from Git's "subtree" merge strategy

51

- split subcommand can be used to extract a subproject's content into a separate branch
- extracts the content and history related to <prefix> and puts the resulting content at the root of the new branch instead of in a subdirectory

```
~/subtrees/local/myproject$ git subtree split --prefix=subproject \  
--branch=split_branch  
Created branch 'split_branch'  
906b5234f366bb2a419953a1edfb590aadc32263
```

- As output, Git prints out the SHA1 value for the HEAD of the newly created tree
 - Provides a reference to work with for that HEAD if needed
 - New branch shows only the set of content from the subproject that was split out (as opposed to content from the superproject).

```
~/subtrees/local/myproject$ git checkout split_branch  
Switched to branch 'split_branch'  
~/subtrees/local/myproject$ ls  
subfile1.txt  subfile2.txt  
~/subtrees/local/myproject$ git log --oneline  
906b523 Add subfile2  
5f7a7db Add subfile1
```

Subtree - create new project from split content

54

- Since can split out content from a subtree, may want to transfer that split content into another project
- Very simple with Git
 - create new, empty project

```
~/subtrees/local/myproject$ cd ~/
~$ mkdir newproj
~$ cd newproj
~/newproj$ git init
Initialized empty Git repository in /Users/dev/newproj/.git/
```

- pull contents of new branch into the new project (repository)

```
~/newproj$ git pull ~/subtrees/local/myproject split_branch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /Users/dev/subtrees/local/myproject
* branch                split_branch -> FETCH_HEAD
```

- subtree command also supports a push subcommand
- This command does a split followed by an attempt to push the split content over to the remote
- Example: the following command splits out the subproject directory and then pushes it to the sub_origin remote reference and into a new branch named *new branch*:

```
~/subtrees/local/myproject$ git subtree push --prefix=subproject sub_origin new_branch
git push using:  sub_origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
To /Users/dev/subtrees/remotes/subproj.git
 * [new branch]      906b5234f366bb2a419953a1edfb590aad32263 -> new_branch
~/subtrees/local/myproject$
```


Lab 5 – Working with subtrees

GitHub projects: super_calc, sub_ui,
sub_docs

https://github.com/<github id>/super_calc

https://github.com/<github id>/sub_ui

https://github.com/<github id>/sub_docs

- Purpose - provides a way to revise git revisions (rewrite history); different types of operations can be done by applying different built-in filters against revisions
- Use case – remove files from commits, change email, split areas out, etc.
- Syntax:

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]
    [--index-filter <command>] [--parent-filter <command>]
    [--msg-filter <command>] [--commit-filter <command>]
    [--tag-name-filter <command>] [--subdirectory-filter <directory>]
    [--prune-empty]
    [--original <namespace>] [-d <directory>] [-f | --force]
    [--] [<rev-list options>...]
```

- Notes:
 - Usual prohibitions/cautions against changing things already in the repository apply
 - Uses git rev-list to help identify commits in some cases

- Syntax

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]  
    [--index-filter <command>] [--parent-filter <command>]  
    [--msg-filter <command>] [--commit-filter <command>]  
    [--tag-name-filter <command>] [--subdirectory-filter <directory>]  
    [--prune-empty]  
    [--original <namespace>] [-d <directory>] [-f | --force]  
    [--] [<rev-list options>...]
```

- Accepts filters to run against specified commits
- Uses git rev-list to help identify commits in some cases

- Another plumbing command
- Used to support some other commands such as filter-branch
- main use is to list a set of commits bounded by some range or criteria; use the rev-list help page to find out more details on specifying ranges if you're interested

```
git rev-list [ --max-count=<number> ]  
               [ --skip=<number> ]  
               [ --max-age=<timestamp> ]  
               . . . (lots of other omitted options)  
               [ --count ]  
               [ --use-bitmap-index ]  
               <commit>... [ -- <paths>...
```

- For filter-branch, rev-list options can be used to bound the set of things for filter-branch to operate against.
- In most cases, if you need to supply an option for rev-list, you can just use the --all option to indicate all commits.

Filter-branch - subdirectory filter

- Filter-branch expects a subdirectory rather than a command
- Use case: split out subdirectory into repository
- Example:

```
$ git filter-branch -f --subdirectory-filter <subdir> -- --all
Rewrite be42303ffb9356b8e27804ce3762afdeea624c64 (1/6) (0 seconds passed,
remainRewrite c6b5cbd8805bc7b1b411a89be66adccc037df553 (2/6) (1 seconds passed,
remainRewrite f64bd2ce520e9a3df4259152a139d259a763bc31 (2/6) (1 seconds passed,
remainRewrite d8a1065e9709d2c5ee20f62fd4e338fe35666c65 (2/6) (1 seconds passed,
remainRewrite c228ad75ef060707ef2905f8bd46012ed67e718b (5/6) (5 seconds passed,
remainRewrite 4758f9e7a9cbeb8dfea0a37a416b46d823ffa95a (5/6) (5 seconds passed,
remaining 1 predicted)
Ref 'refs/heads/master' was rewritten
```

- Notice that it is going through and processing the set of commits here that have to do with the particular <subdir>
- Ending status of 'refs/heads/<branch>' is indication of successful completion

Filter-branch - undoing a filter-branch

61

- After a filter-branch run, the HEAD points at the rewritten revisions
- Commits that make up the previous structure are still there—just not accessible in this chain
- This is why filter-branch to split or remove content doesn't shrink repos
 - The online help page for filter-branch discusses the remaining steps that you have to follow to actually remove content and shrink the repository.
- Can undo the filter-branch by just using a reset to prior commit
- Git keeps a backup of the previous reference pointed to by HEAD before the filter-branch - SHA1 value in `.git/refs/original/refs/heads/<branch>`
- To undo filter-branch run the command, ***git reset --hard <sha1 from path above>***
- Can also use the *git reflog* command to find the relative reference before the operation and then run the command, ***git reset --hard HEAD@{<relative offset>}***.

Filter-branch - deleting a file from history 62

- Use case: remove undesired file from revisions in a branch
- Either tree-filter or index-filter can be used; index-filter is faster (like staging area)
- index-filter option takes a command to apply when it finds an instance of the thing you are trying to filter out
- Can use the git rm command here to simplify the task
 - Add the --cached option because working in the index
 - Add the --ignore-unmatched option to the git rm command
 - » Tells the command to exit with a zero return code even if none of the files match
- Need to run command for each branch (add -f to filter-branch to overwrite backup reference for previous branch)

```
$ git filter-branch --index-filter 'git rm --cached --ignore-unmatch <relative  
path to file>' <branch name>|
```

Filter-branch - changing email for commits⁶³

- Use case: changing email address for set of commits
- Uses environment filter
 - Allows environment variables to be set and used by commands passed to filter-branch
- Set GIT_AUTHOR_EMAIL environment variable
- Must export it to ensure it is set for operation
- Example:

```
$ git filter-branch -f --env-filter 'GIT_AUTHOR_EMAIL=user@mycompany.com;  
export GIT_AUTHOR_EMAIL' -- HEAD~3..HEAD  
Rewrite 11e90d549c65da2a2c60d790f87bd1ddc1831dfa (1/3) (0 seconds passed,  
remaining 0 predicted)  
Rewrite 1d60e1557b33ea181066df85f3f6b9e633d9e325 (2/3) (1 seconds passed,  
remaining 0 predicted)  
Rewrite f713a23ae271e9f261f4dc25ef15a00d95c9ee41 (2/3) (1 seconds passed,  
remaining 0 predicted)  
Ref 'refs/heads/master' was rewritten
```

Lab 6 – Working with filter-branch

GitHub projects: brentlaster/roarv2

<https://github.com/brentlaster/roarv2>

Command: (Interactive) Rebase

65

- Purpose - allows you to modify commits in the git history
- Use case - you need to make some kind of modification to one or more commits in the repository (rewrite history)

- Syntax

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]  
           [<upstream> [<branch>]]
```

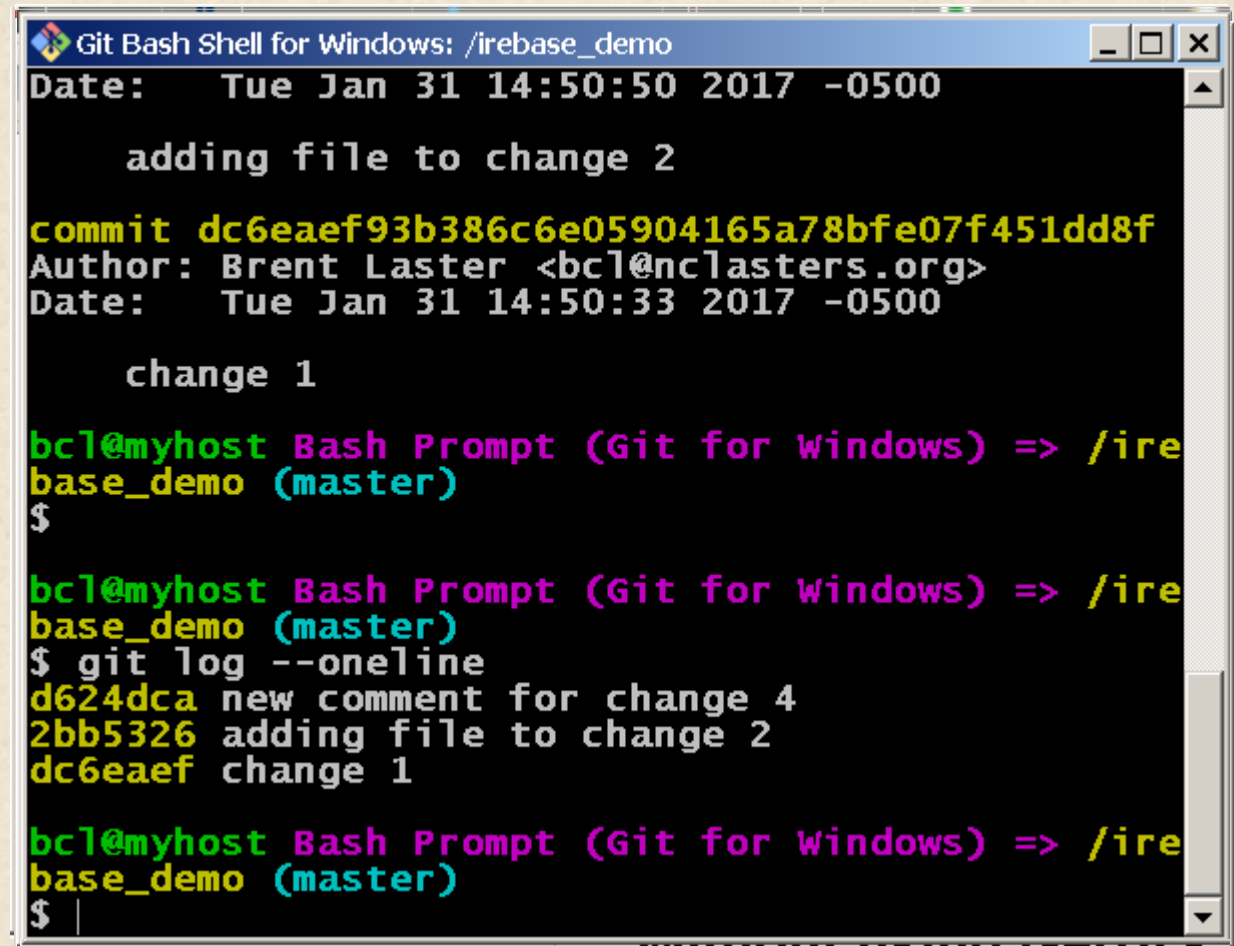
```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]  
           --root [<branch>]
```

- Notes - creates an interactive script/batch file to modify commits
- Cautions - don't use this on anything already pushed

Interactive Rebase

66

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts
- Commits are updated



```
Git Bash Shell for Windows: /irebase_demo
Date:   Tue Jan 31 14:50:50 2017 -0500

    adding file to change 2

commit dc6eaeef93b386c6e05904165a78bfe07f451dd8f
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:33 2017 -0500

    change 1

bcl@myhost Bash Prompt (Git for Windows) => /irebase_demo (master)
$

bcl@myhost Bash Prompt (Git for Windows) => /irebase_demo (master)
$ git log --oneline
d624dca new comment for change 4
2bb5326 adding file to change 2
dc6eaeef change 1

bcl@myhost Bash Prompt (Git for Windows) => /irebase_demo (master)
$
```

- Purpose - Add additional information to objects in the Git repository or look at such information
- Use case - At some point after making a commit, you may decide that there are additional comments or other non-code information that you'd like to add with the commit - *without changing the commit itself*.
- Syntax

`git notes [list [<object>]]`

`git notes add [-f] [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]`

`git notes copy [-f] (--stdin | <from-object> <to-object>)`

`git notes append [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]`

`git notes edit [--allow-empty] [<object>]`

`git notes show [<object>]`

`git notes merge [-v | -q] [-s <strategy>] <notes-ref>`

`git notes merge --commit [-v | -q]`

`git notes merge --abort [-v | -q]`

`git notes remove [--ignore-missing] [--stdin] [<object>...]`

`git notes prune [-n | -v]`

`git notes get-ref`

- Create a note `$ git notes add -m "This is an example of a note" 2f2ea1e`

- Create a note in a custom namespace (add --ref)

```
$ git notes --ref=review add -m "Looks ok to me" f3b05f9
```

- View a note (for a specific revision)

```
$ git notes show 2f2ea1e
This is an example of a note
```

- List notes in log

```
$ git log --show-notes=*
commit 80e224b24e834aaa8915e3113ec4fc635b060771
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Jul 1 13:01:58 2016 -0400

    Removing test subdir on master

commit efl5dca5c6577d077e38a05b80670024e1d92c0a
Author: unknown <bcl@nclasters.org>
Date:   Fri Apr 24 12:32:50 2015 -0400

    update test case

Notes (review):
  Looks ok to me

commit 2f2ea1e30fe4630629477338a0ab8618569f0f5e
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 17:34:57 2015 -0400

    Add in testing example files

Notes:
  This is an example of a note
```

Command: grep

- Purpose - provides a convenient (and probably familiar) way to search for regular expressions in your local Git environment.
- Use case - self-explanatory
- Syntax

```
git grep [-a | --text] [-l] [--textconv] [-i | --ignore-case] [-w | --word-regexp]
        [-v | --invert-match] [-h|-H] [--full-name]
        [-E | --extended-regexp] [-G | --basic-regexp]
        [-P | --perl-regexp]
        [-F | --fixed-strings] [-n | --line-number]
        [-l | --files-with-matches] [-L | --files-without-match]
        [(-O | --open-files-in-pager) [<pager>]]
        [-z | --null]
        [-c | --count] [--all-match] [-q | --quiet]
        [--max-depth <depth>]      [--color[=<when>] | --no-color]
        [--break] [--heading] [-p | --show-function]
        [-A <post-context>] [-B <pre-context>] [-C <context>]
        [-W | --function-context]
        [--threads <num>]
        [-f <file>] [-e] <pattern>
        [--and|--or|--not(|)]-e <pattern>...
        [ [--[no-]exclude-standard] [--cached | --no-index | --untracked] | <tree>... ]
        [--] [<pathspec>...]
```

- Notes
 - Several options are similar to OS grep options

- Default behavior - search for all instances of an expression across all tracked files in working directory
- Search for all instances of expression "database" across all java files (note use of --)

```
$ git grep database -- *.java
api/src/main/java/com/demo/pipeline/status/status.java:      @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://" + strMySQLHost + ":" + strMySQLPort + "/" + strMySQLDatabase, e);
```

- -p option tells Git to try and show header of method or function where search target was found
- --break - make output easier to read
- --heading - prints filename above output

```
$ git grep -p --break --heading database -- *.java

api/src/main/java/com/demo/pipeline/status/status.java
13=public class V1_status {
31:      @Path("/database")

dataaccess/src/main/java/com/demo/dao/MyDataSource.java
18=public class MyDataSource {
64:      logger.log(Level.SEVERE, "Could not access database via
connect string
jdbc:mysql://" + strMySQLHost + ":" + strMySQLPort + "/" + strMySQLDatabase, e);
```

- boolean operators `$ git grep -e 'database' --and -e 'access' -- *.java`
- search in staging area `$ git grep -e 'config' --cached -- '*.txt'`
- search in specific commit(s) `$ git grep -e 'database' HEAD -- *.java`
`$ git grep -e 'database' b2e575a -- *.java`

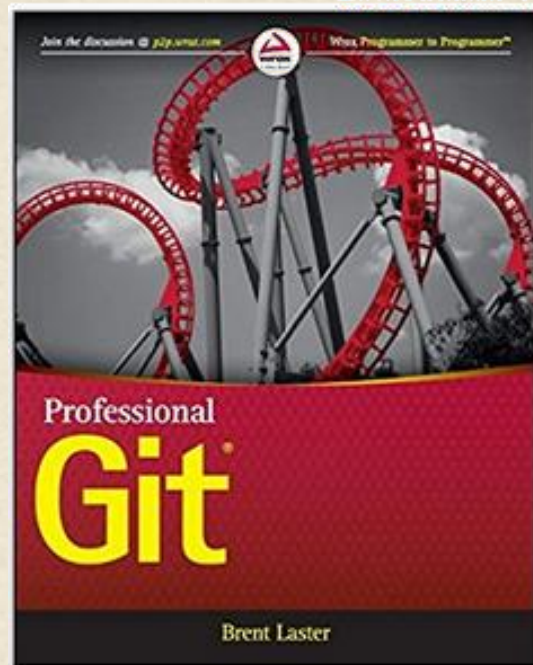
That's all - thanks!

Professional Git 1st Edition

by Brent Laster (Author)

★★★★★ 7 customer reviews

[Look inside](#) ↴



by Brent Laster (Author)

★★★★☆ 5 customer reviews

[Look inside](#) ↴

