Important Prerequisites and Setup

- Basic Git experience or training such as the Git Fundamentals Safari class.
- Have a recent version of Git downloaded and running on your system

https://www.git-scm.org/

- If you don't already have one, sign up for a free Github account

https://www.github.com

- Download (and print if desired) labs doc from

https://github.com/brentlaster/safaridocs/blob/master/next-level-git-workflow-labs.pdf

- Above is automatic download link. Can also go to https://github.com/brentlaster/safaridocs and get next-level-git-workflow-labs.pdf.

Next Level Git - Master your workflow

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

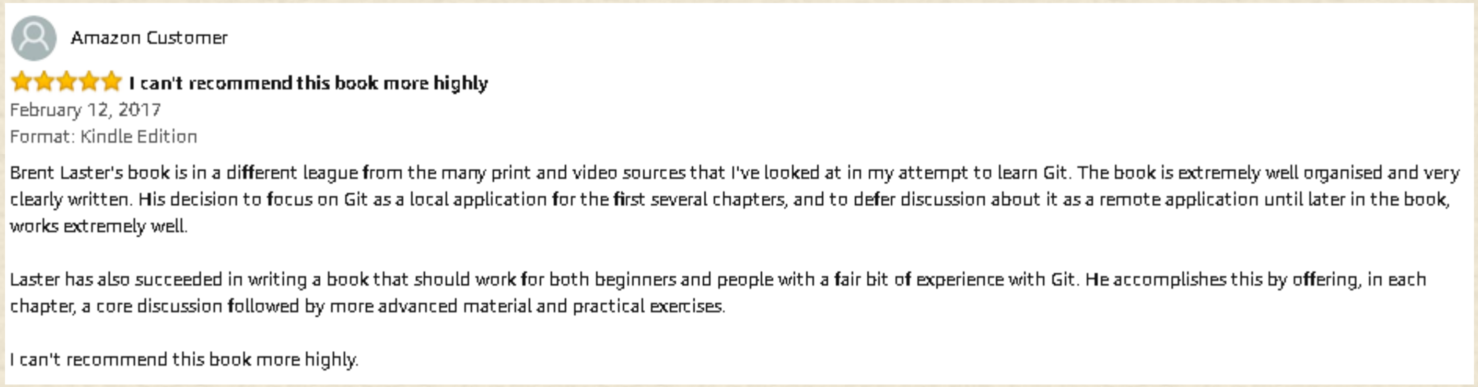
Look inside

O'REILLY
Jenkins 2 Up & Running
EVOLVE YOUR DEPLOYMENT PIPELINE FOR NEXT-GENERATION AUTOMATION
Brent Laster
Foreword by Kohsuke Kawaguchi

Book – Professional Git

- Reference for material in this class 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

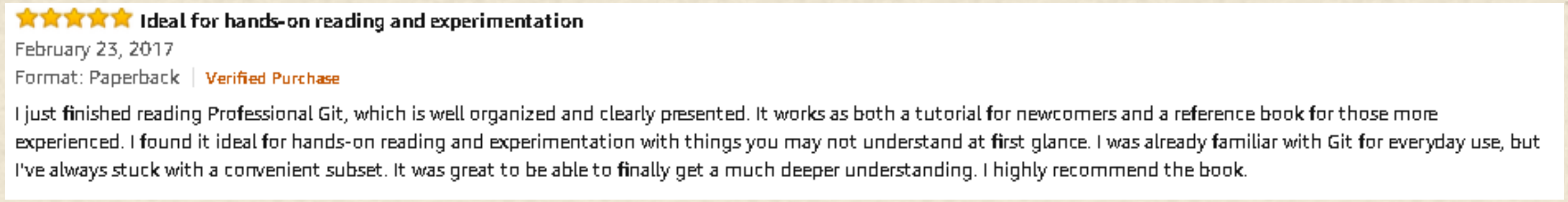

Amazon Customer

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.

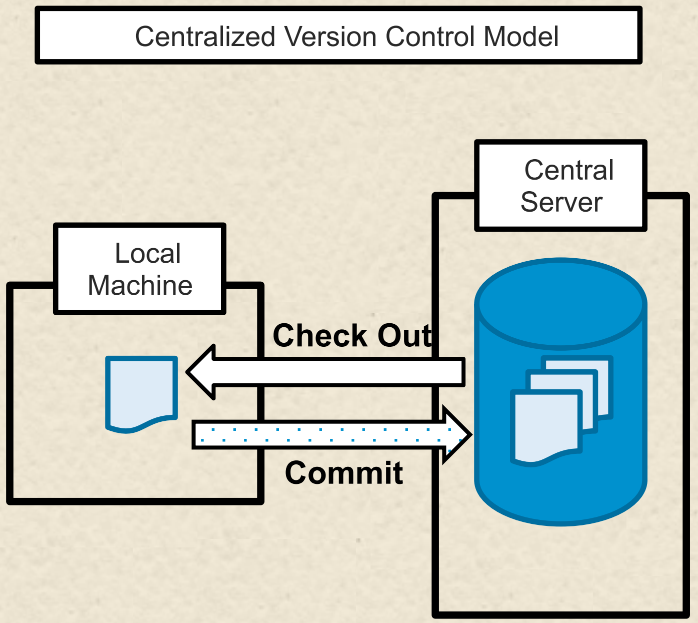

Ideal for hands-on reading and experimentation
February 23, 2017
Format: Paperback | Verified Purchase

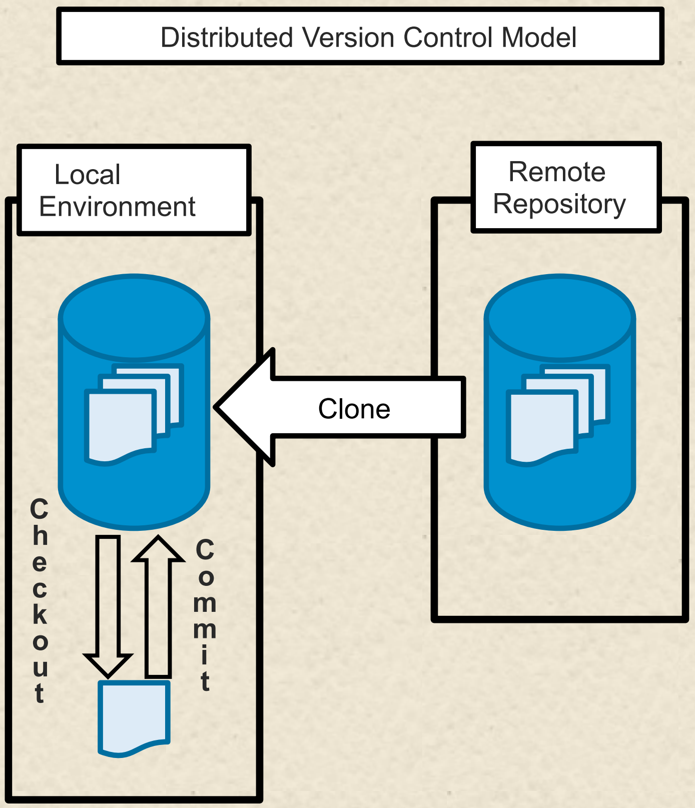
I just finished reading Professional Git, which is well organized and clearly presented. It works as both a tutorial for newcomers and a reference book for those more experienced. I found it ideal for hands-on reading and experimentation with things you may not understand at first glance. I was already 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.

Agenda

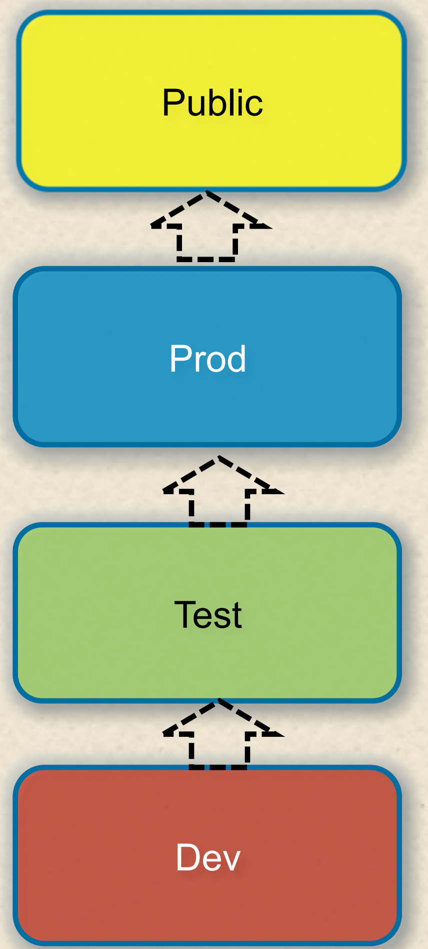
- Core concepts refresh
- Reset and Revert
- Grep
- Bisect
- Worktrees
- Submodules
- Subtrees
- Hooks

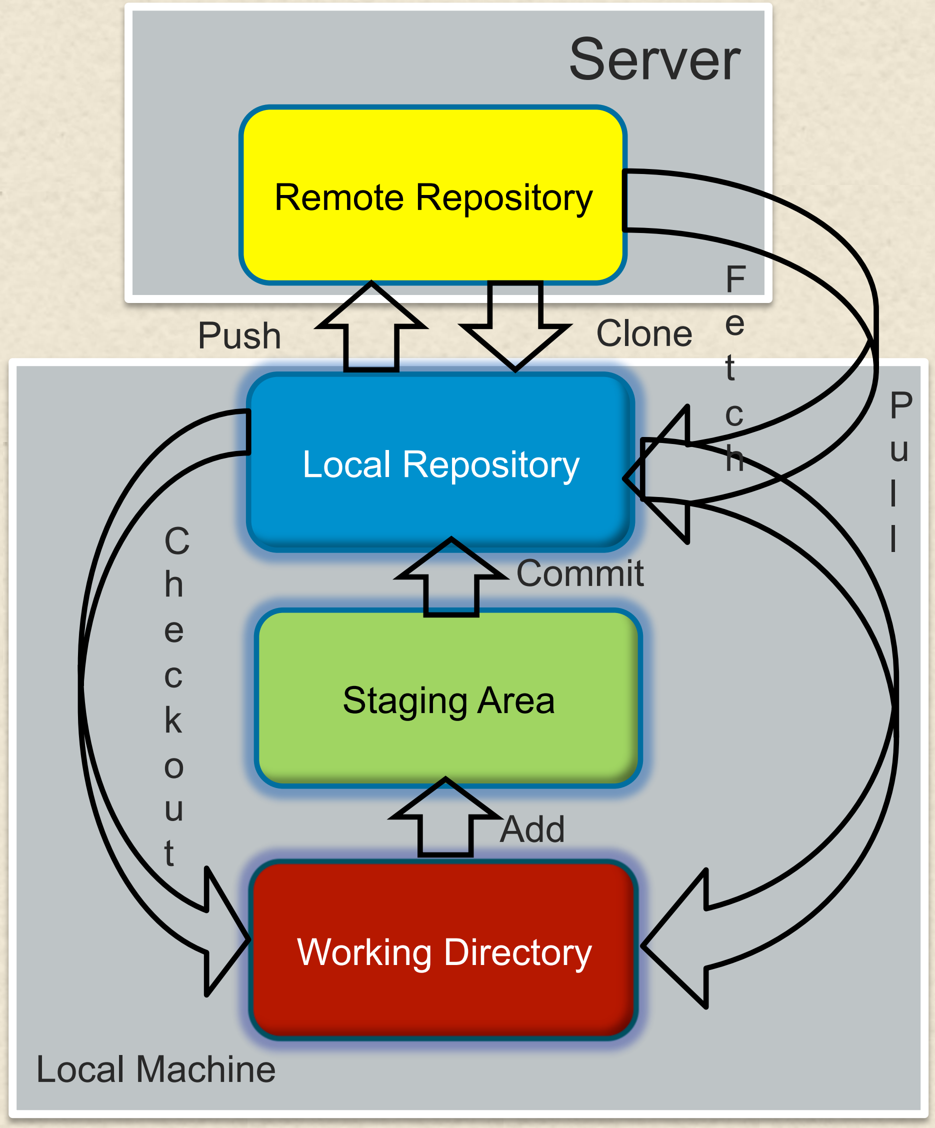
Centralized vs. Distributed VCS





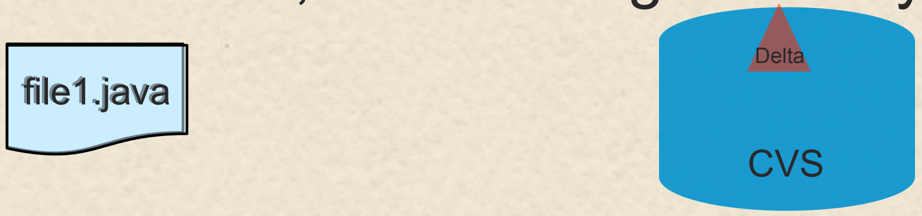
Git in One Picture



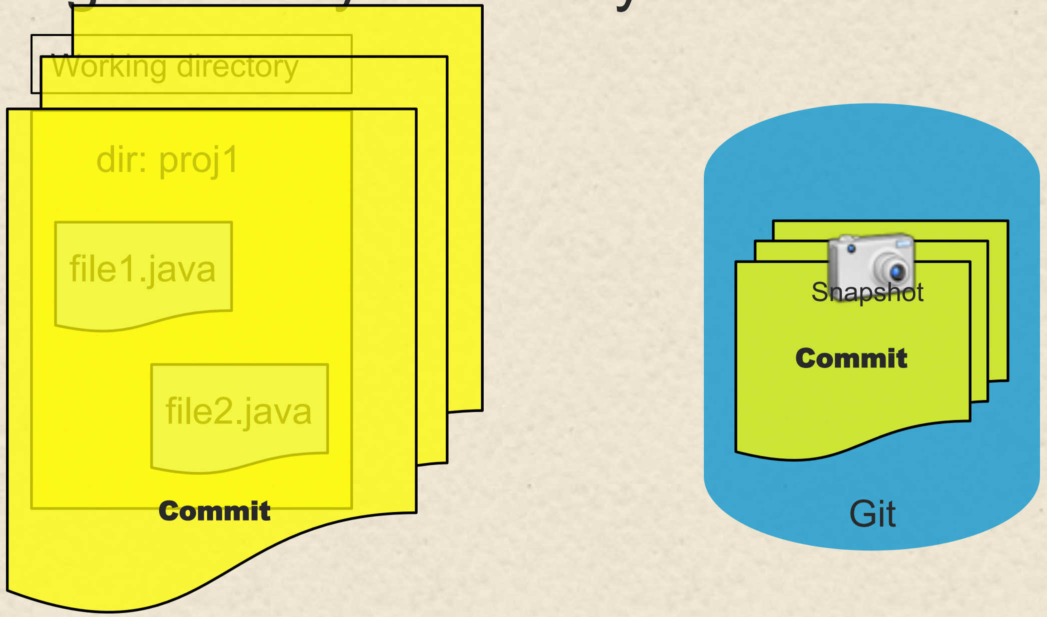


Git Granularity (What is a unit?)

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

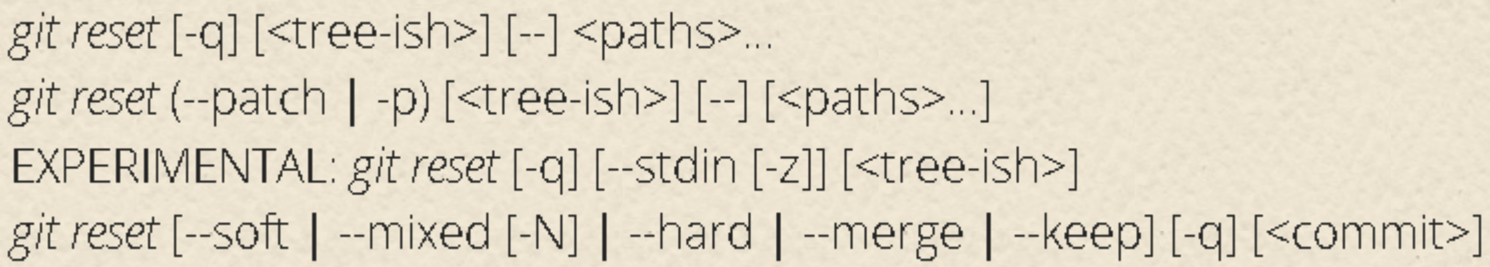


- In Git, the unit of granularity is usually a tree



Command: Git Reset

- 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

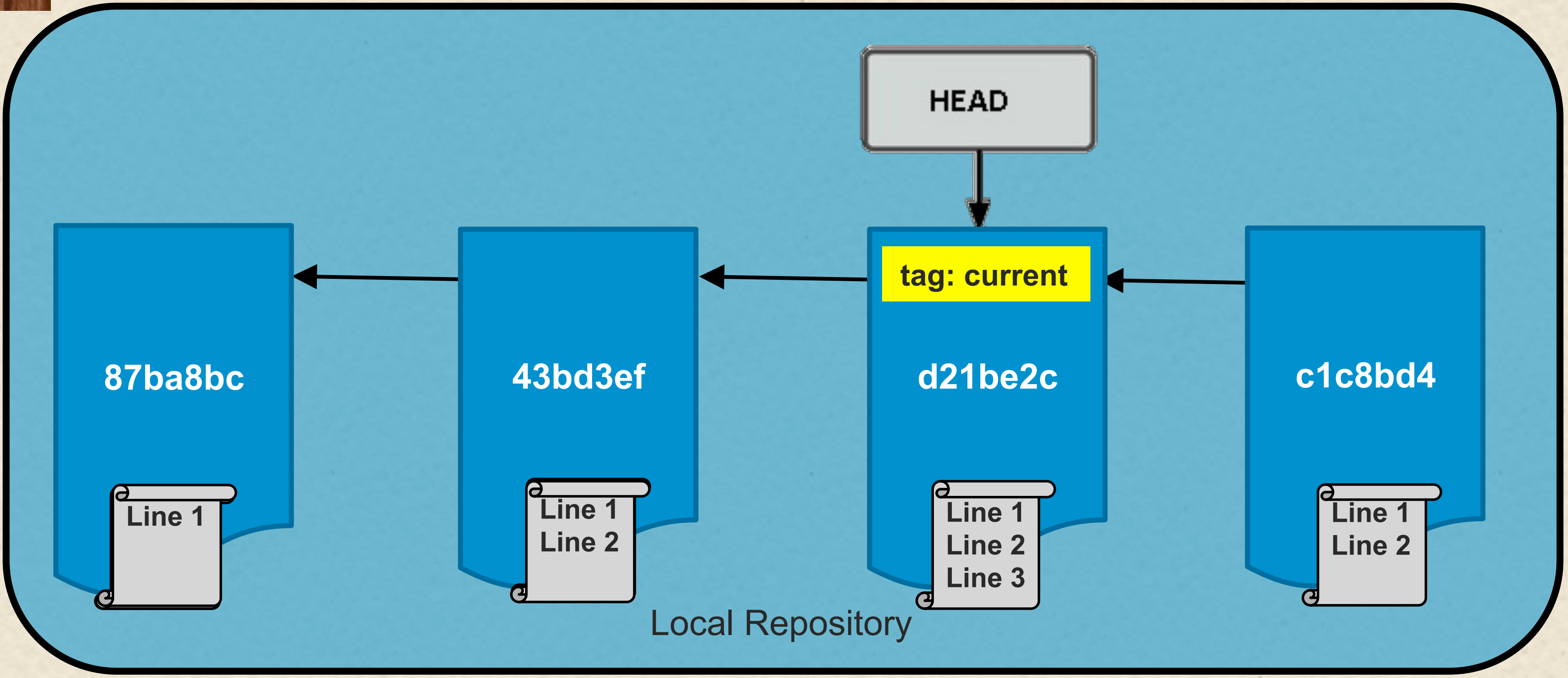
Command: Git Revert

- 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:

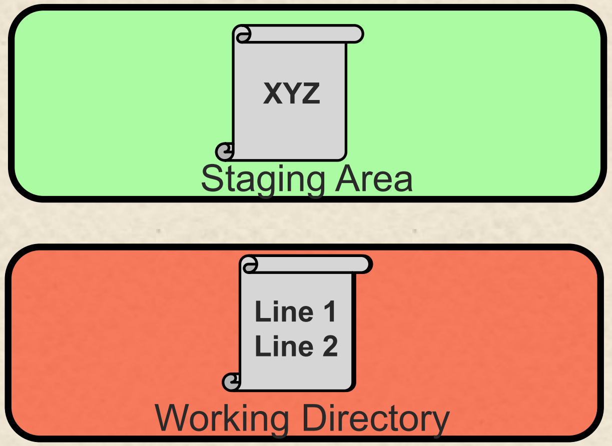
```
git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>...
git revert --continue
git revert --quit
git revert --abort
```

- 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.

Reset and Revert



```
git reset --hard 87ba8bc
git reset current~1 [--mixed]
git revert HEAD
```



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] [-I] [--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

grep

- **Default behavior - search for all instances of an expression across all tracked files in working directory**
- **Search for all instances off 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**

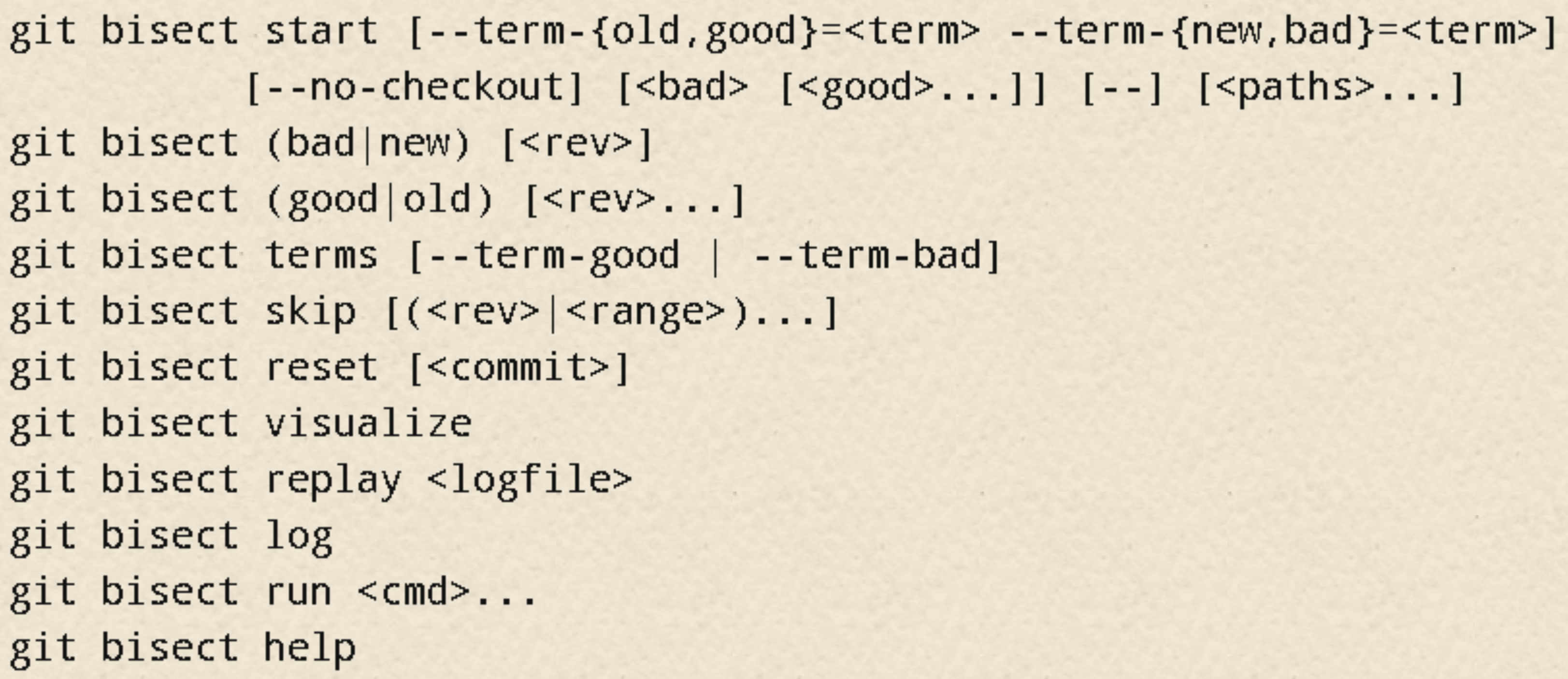
```
$ git grep -e 'config' --cached -- '*.txt'
```

- **search in specific commit(s)**

```
$ git grep -e 'database' HEAD   -- *.java
$ git grep -e 'database' b2e575a  -- *.java
```

Command: bisect

- 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:



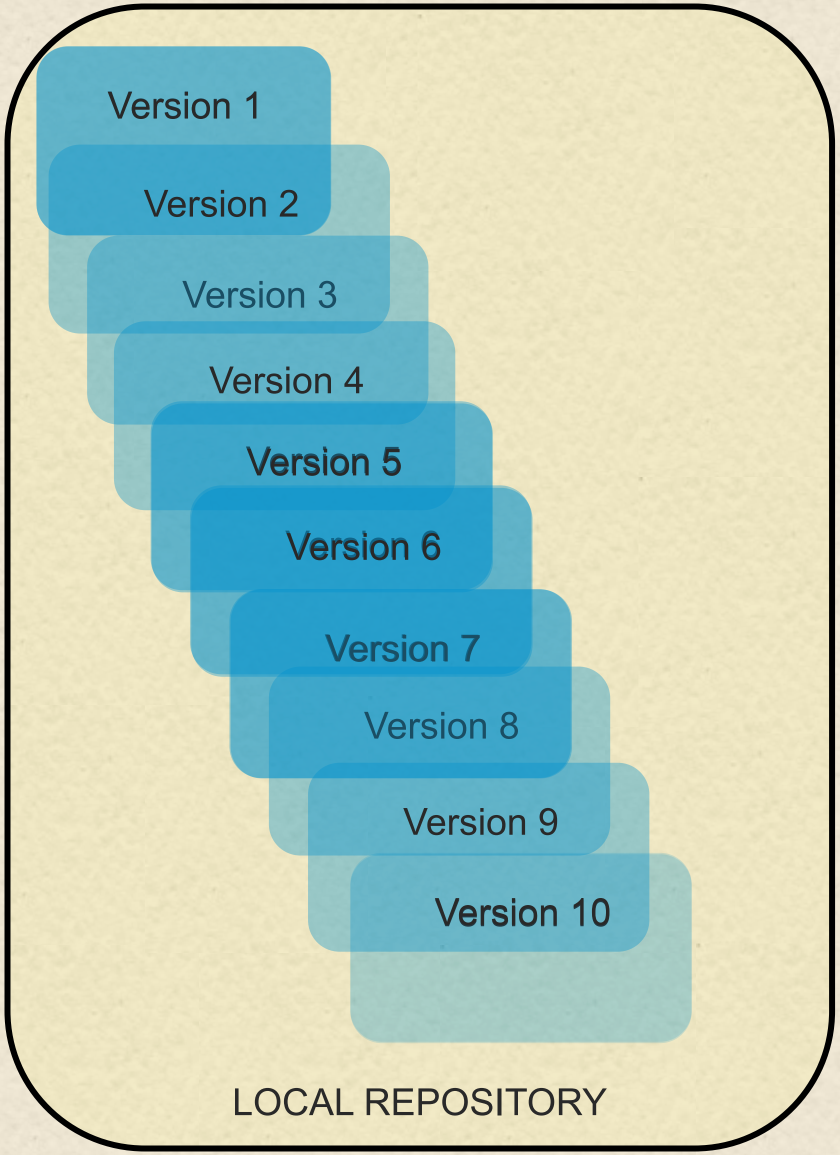
```
git bisect start [--term-{old,good}=<term> --term-{new,bad}=<term>]
          [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new) [<rev>]
git bisect (good|old) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]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

Git Bisect

- 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



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

Lab 1: Using the Git bisect command

Background: Switching between Branches

Command: git checkout <branch>

git checkout master

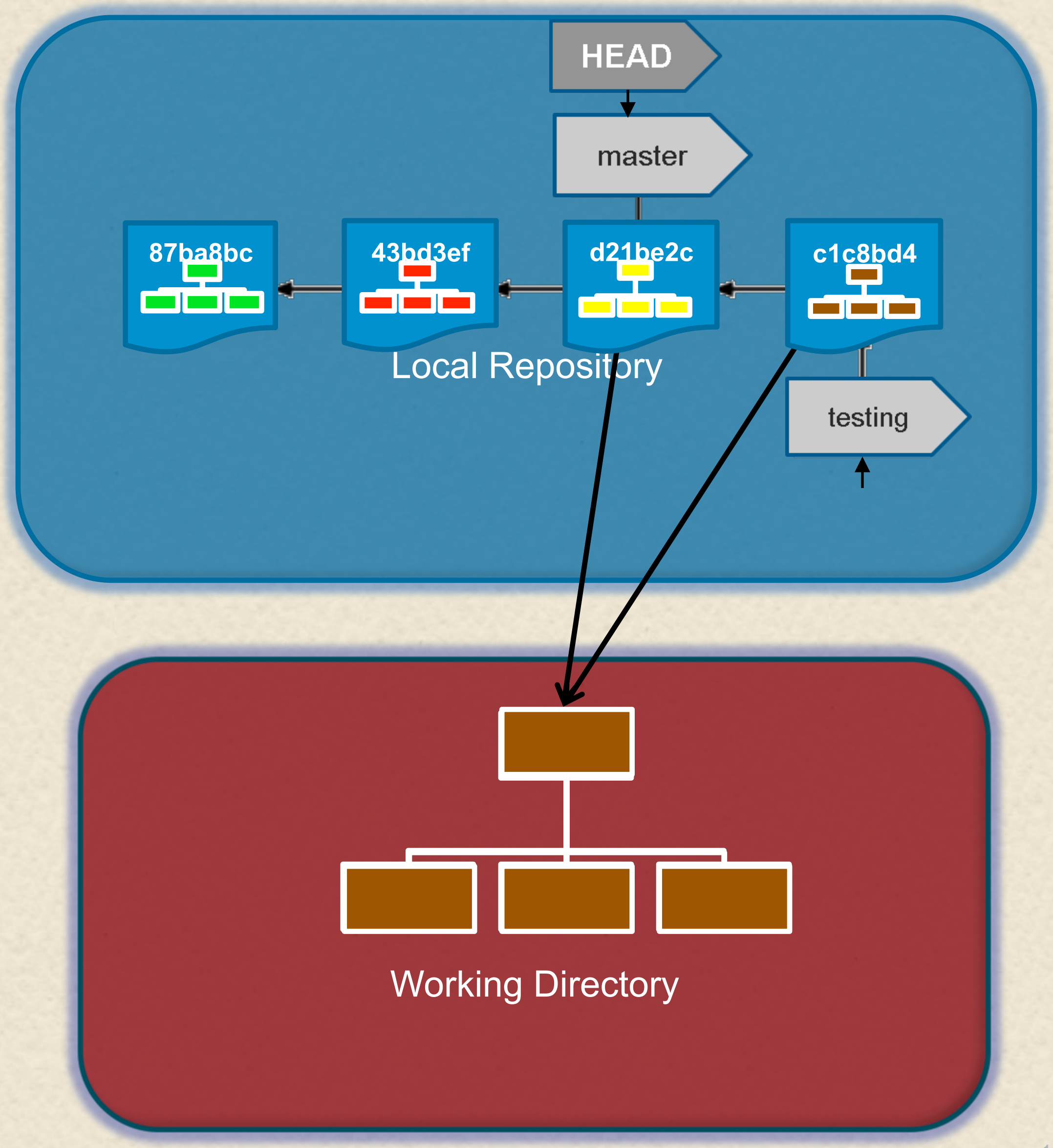
- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**
  - **Updates indicators**

git checkout testing

git checkout master

git checkout testing

- **git branch**
  - * master
  - * testing



Command: Worktrees

- 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 prune* [-n] [-v] [--expire <expire>]
- Notes
  - "Traditional" working directory is called the *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*>.

Worktrees - syntax and usage

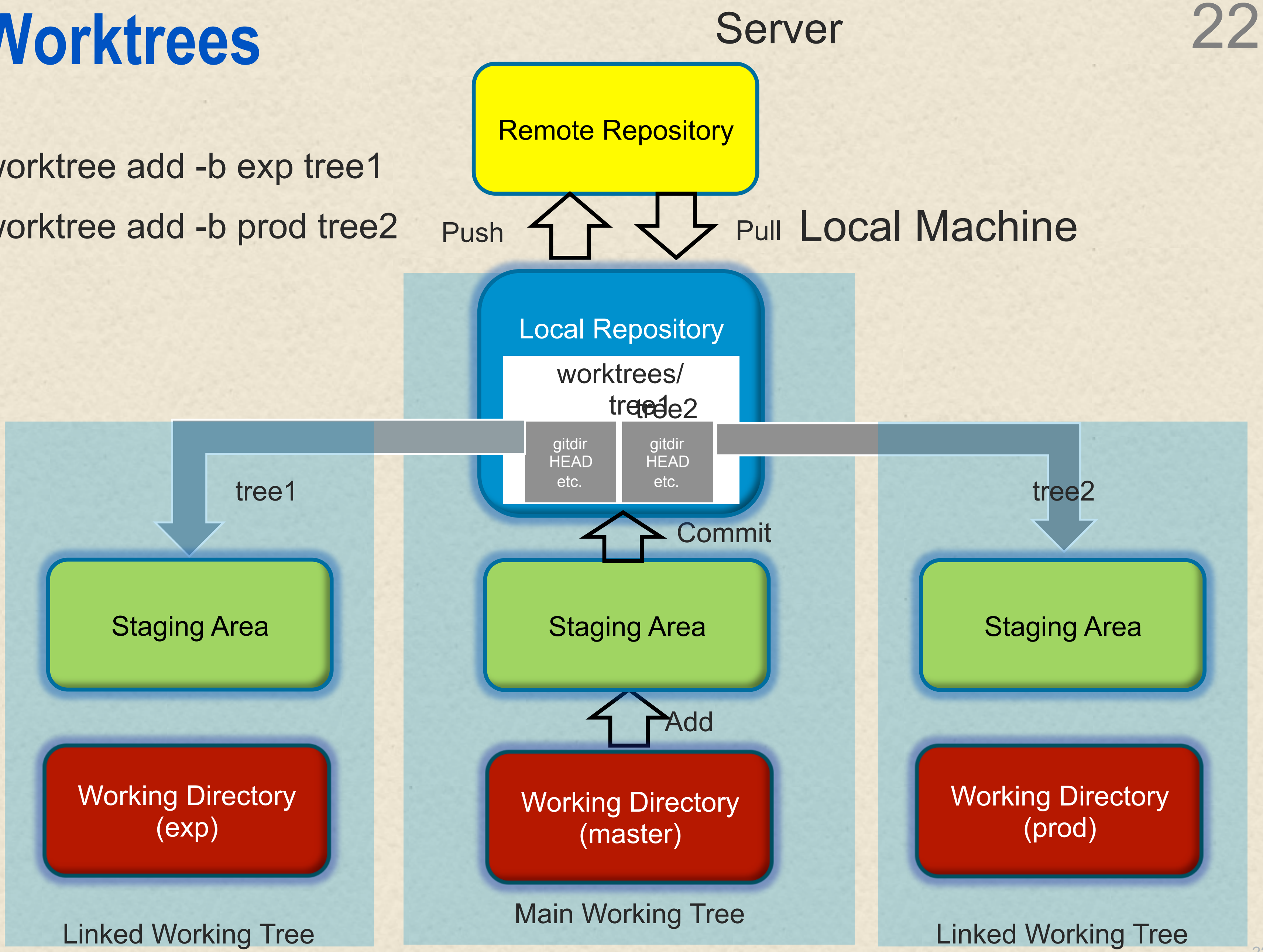
- Syntax
  - *git worktree add* [-f] [--detach] [-b <new-branch>] <path> [<branch>]
  - *git worktree list* [--porcelain]
  - *git worktree prune* [-n] [-v] [--expire <expire>]
- Subcommands
  - Add - create a separate working tree for specified branch
    - » has checked out copy of the branch
    - » 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
  - Prune - removes worktree information from the .git area
    - » only applies after worktree directory tree has been removed

Worktrees - meta-information

- 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*>.
- Working trees can be created on removable media
  - To persist after unmounting and prunes, create "marker" file
    - » Named .git/worktrees/<working tree named>/locked
    - » Convention is to have reason for lock in file

Worktrees

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



Lab 2: Working with Worktrees

Command: Submodules

- 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>…]
git submodule [--quiet] foreach [--recursive] <command>
git submodule [--quiet] sync [--recursive] [--] [<path>…]
```

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

Submodules 1

What happens when you add a submodule?

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.

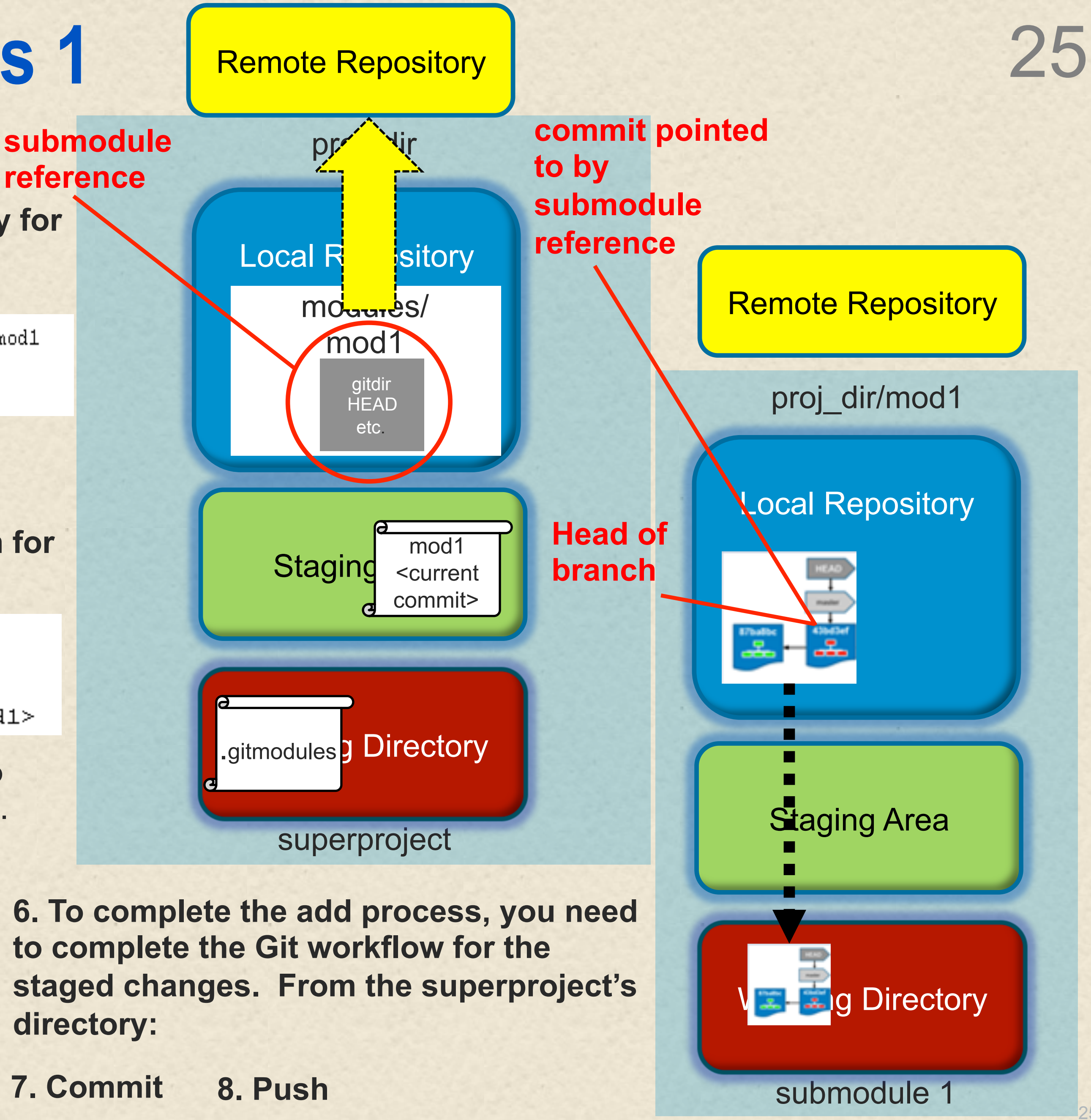
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ 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

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/
```

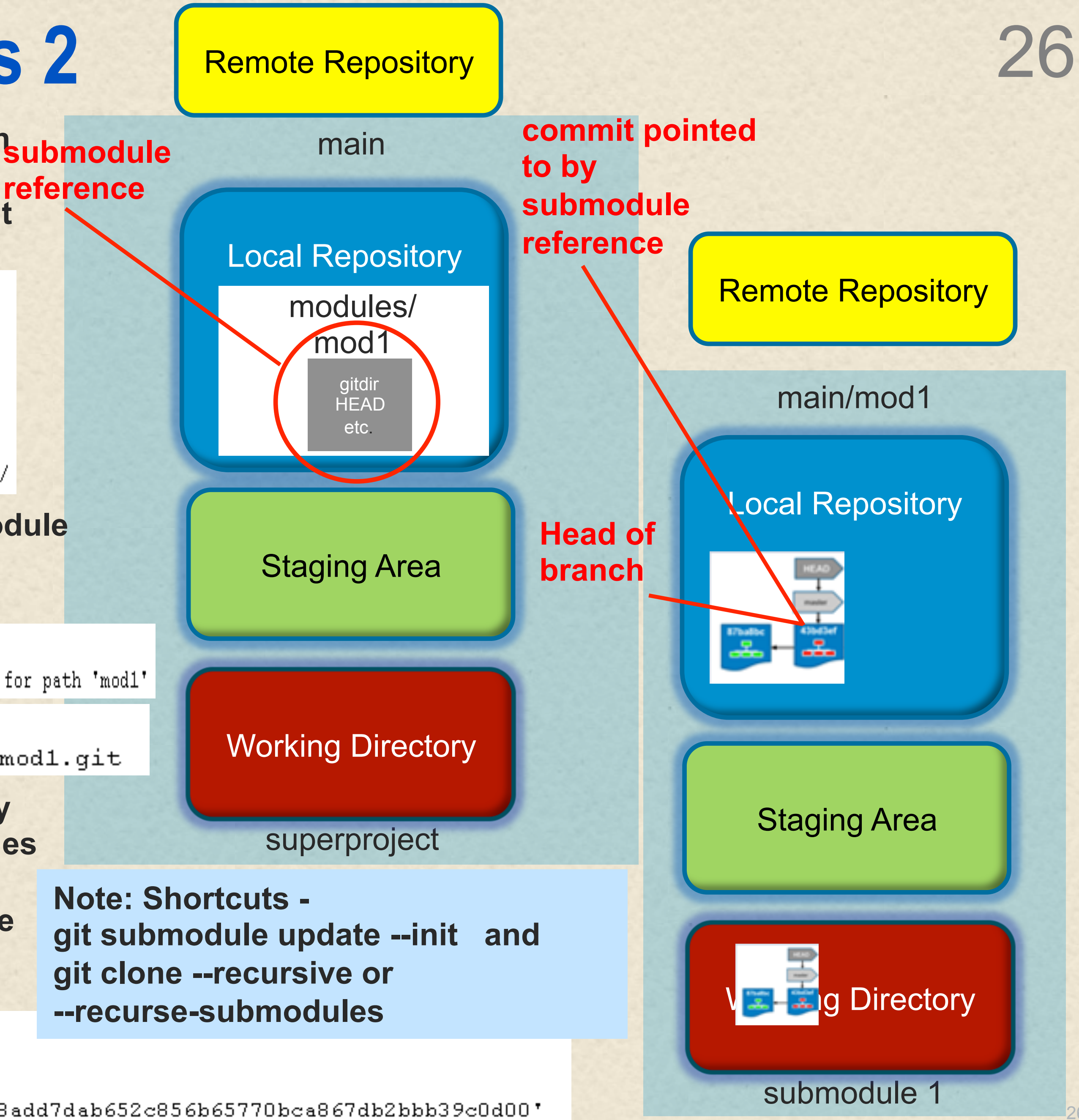
2. git submodule init - puts submodule location information in superproject's config file

```
$ 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'
```



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

Submodules 3

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

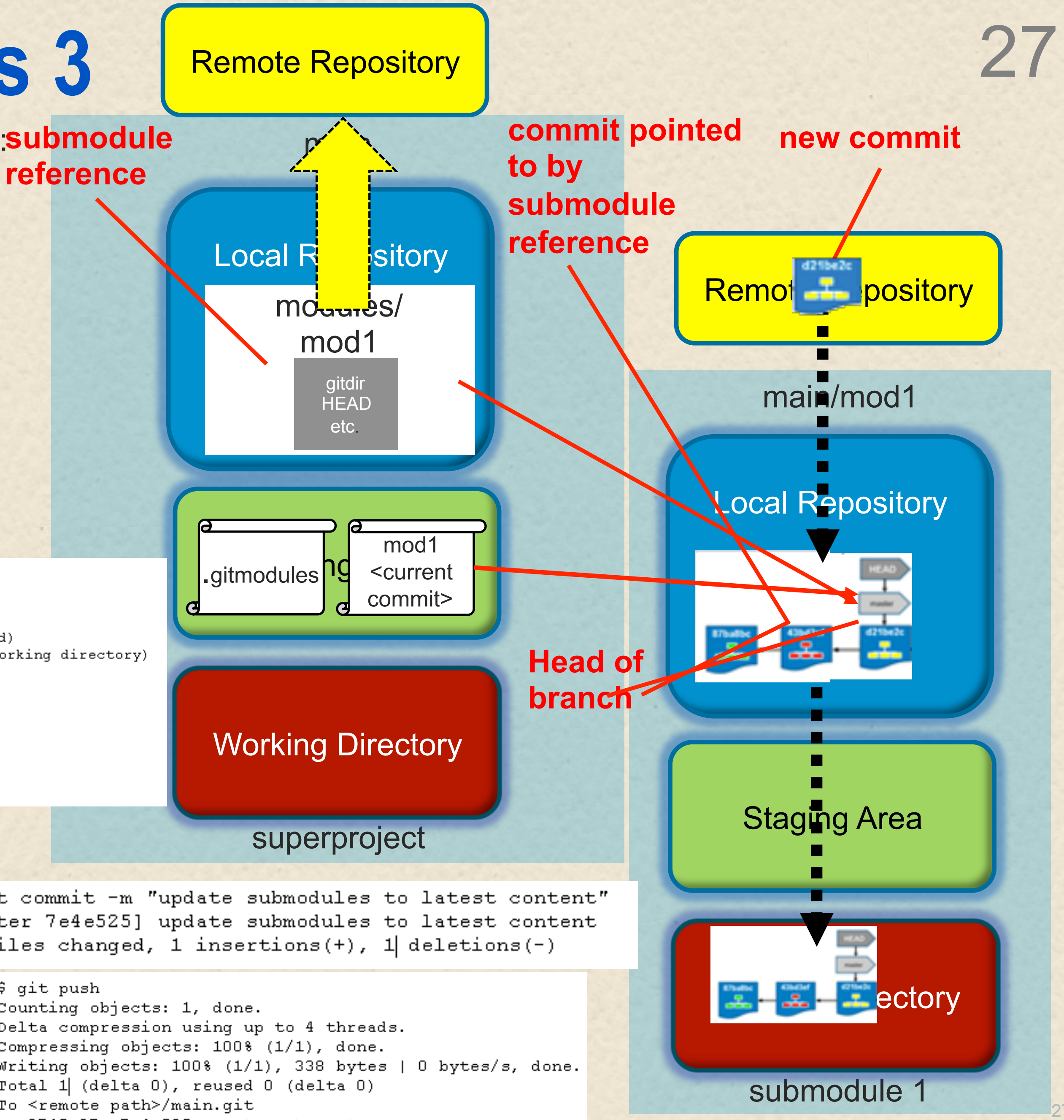
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        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
```



Submodules - merging

- 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.

Submodule - Challenges

- 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.

Submodules - general rules for updates

- ***If you update something in a submodule, follow these steps:***

1. In the submodule directory, commit and push it out to the submodule's remote.

2. Go back to the superproject. The superproject should show that that particular submodule area has changed—almost like a file in the repository with the submodule name.

3. Stage and commit that changed area (submodule name) in the superproject to ensure that the superproject points to the updated commit in the submodule.

4. Push out that change in the superproject to the superproject's remote. This ensures that anyone cloning or pulling the superproject gets a version that points to the latest updates in the submodules.

- ***If you pull an update of the superproject, follow these steps:***

1. Ensure that you have also pulled the latest versions of the submodules (using the recurse-submodules option or foreach subcommand, or by pulling each area).

2. In the superproject, run the submodule update to check out the commit in the submodule that corresponds to the submodule references in the superproject.

Lab 3: Working with Submodules

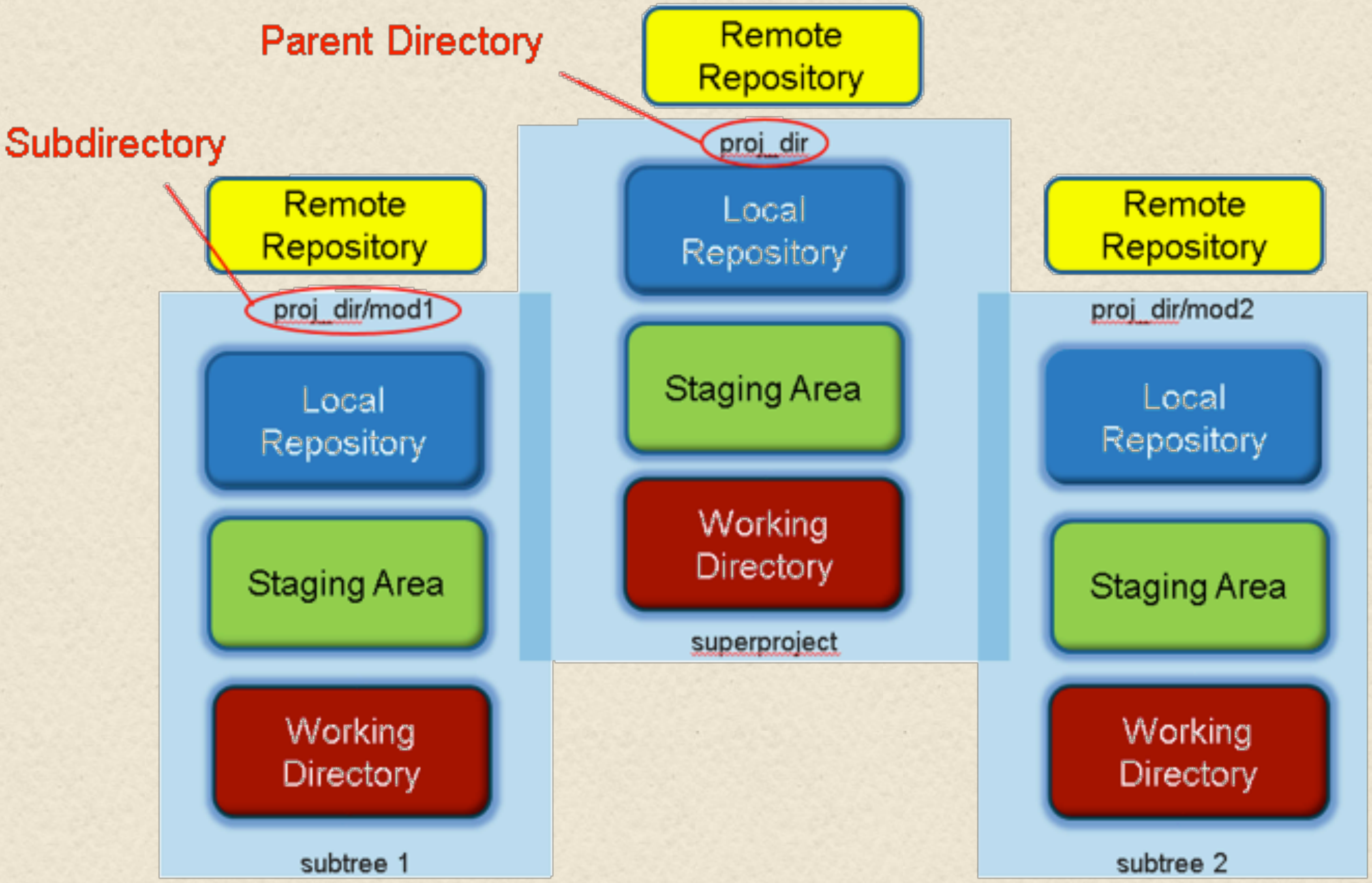
Command: Subtrees

- 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>]
```

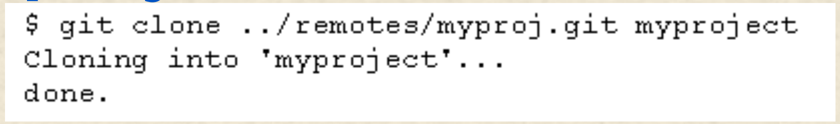
- Notes
  - No links like a submodule - just a copy in a subdirectory
  - Advantage - no links to maintain
  - Disadvantage - extra content to carry around with your project

Subtrees - overall view

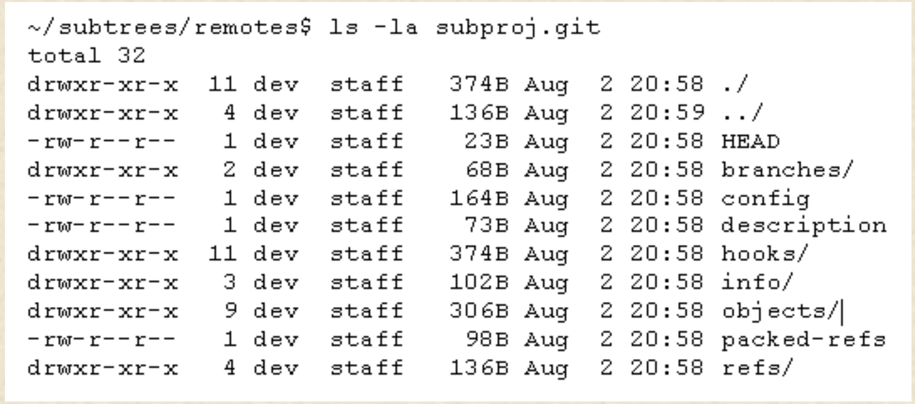


Subtrees - adding a project as a subtree

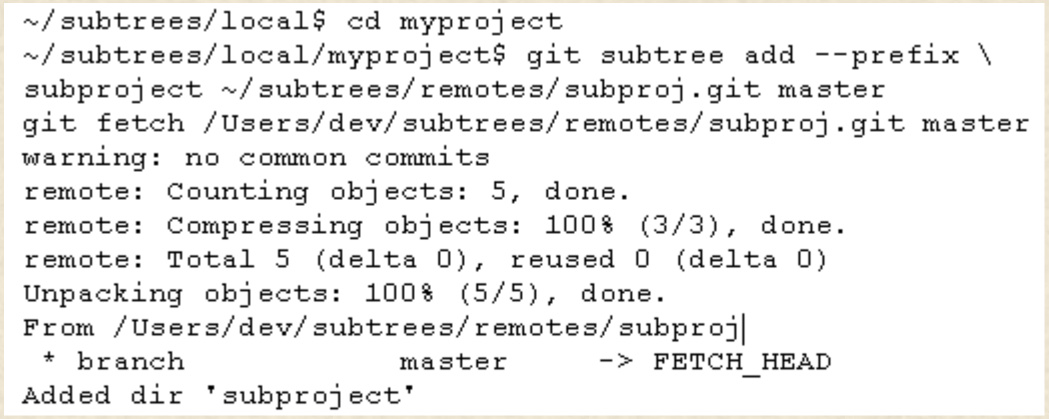
- Simplest form, specify: prefix, remote path to repository, branch (optional)
- Suppose we clone down a project - myproject
- And, on the remote side, we have another project, subproj.
- Now, we can add subproj as a subtree of myproject (--prefix subproject).
- Subtree now shows subproject in structure and history



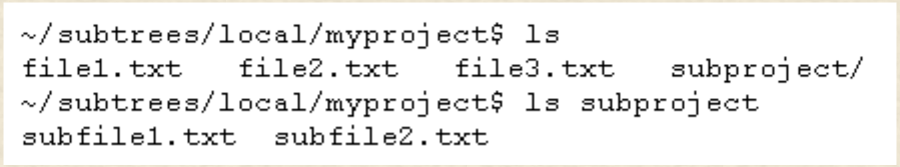
```
$ git clone ../remotes/myproj.git myproject
Cloning into 'myproject'...
done.
```



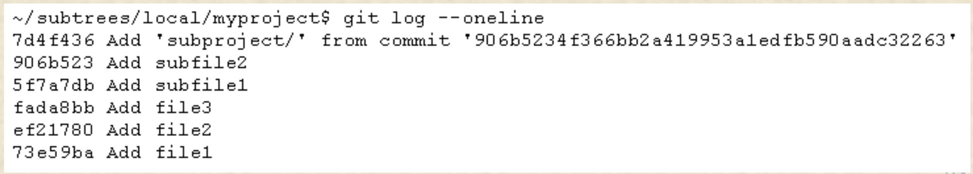
```
~/subtrees/remotes$ ls -la subproj.git
total 32
drwxr-xr-x  11 dev  staff   374B Aug  2 20:58 ./
drwxr-xr-x   4 dev  staff   136B Aug  2 20:59 ../
-rw-r--r--   1 dev  staff    23B Aug  2 20:58 HEAD
drwxr-xr-x   2 dev  staff    68B Aug  2 20:58 branches/
-rw-r--r--   1 dev  staff   164B Aug  2 20:58 config
-rw-r--r--   1 dev  staff    73B Aug  2 20:58 description
drwxr-xr-x  11 dev  staff   374B Aug  2 20:58 hooks/
drwxr-xr-x   3 dev  staff   102B Aug  2 20:58 info/
drwxr-xr-x   9 dev  staff   306B Aug  2 20:58 objects/
-rw-r--r--   1 dev  staff    98B Aug  2 20:58 packed-refs
drwxr-xr-x   4 dev  staff   136B Aug  2 20:58 refs/
```



```
~/subtrees/local$ cd myproject
~/subtrees/local/myproject$ git subtree add --prefix \
subproject ~/subtrees/remotes/subproj.git master
git fetch /Users/dev/subtrees/remotes/subproj.git master
warning: no common commits
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/remotes/subproj
 * branch            master     -> FETCH_HEAD
Added dir 'subproject'
```



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



```
~/subtrees/local/myproject$ git log --oneline
7d4f436 Add 'subproject/' from commit '906b5234f366bb2a419953a1edfb590aadc32263'
906b523 Add subfile2
5f7a7db Add subfile1
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
```

Subtrees - tips for adding

- 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: 906b5234f366bb2a419953a1edfb590aadc32263
```

Subtrees - updating and merging

- Update command - similar to add

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

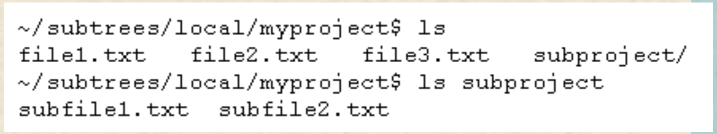
- pulls down the latest content from the remote into the subtree area
- --squash compresses the history again
  - can be omitted, but usually simplifies things
- also a *git subtree merge* command to merge commits up to a desired point into a subproject denoted by the --prefix argument
  - can be used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

Subtrees vs. subtree merge strategy

- In Git, there is also a merge strategy named *subtree*.
- git subtree command (actually a script that's been incorporated into Git) is not the same thing as the subtree merge strategy.
- Git chooses the best strategy depending on the situation.
- The subtree merge strategy is designed to be used when two trees are being merged and one is a subtree (subdirectory) of the other.
- subtree merge strategy tries to shift the subtrees to be at the same level in order to merge similar structures.
- For more information, search for *subtree* in the help page for merge.

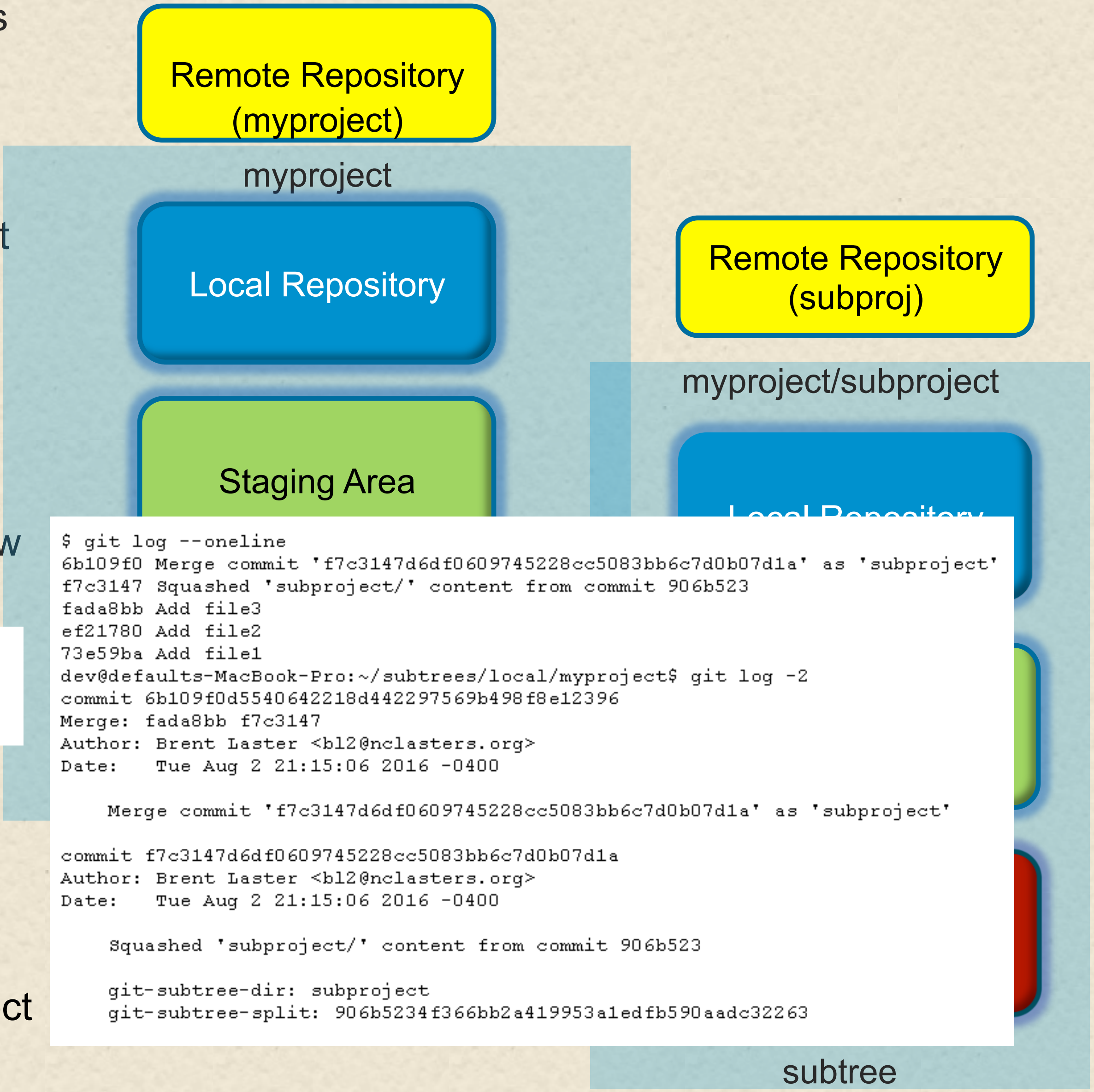
Subtrees

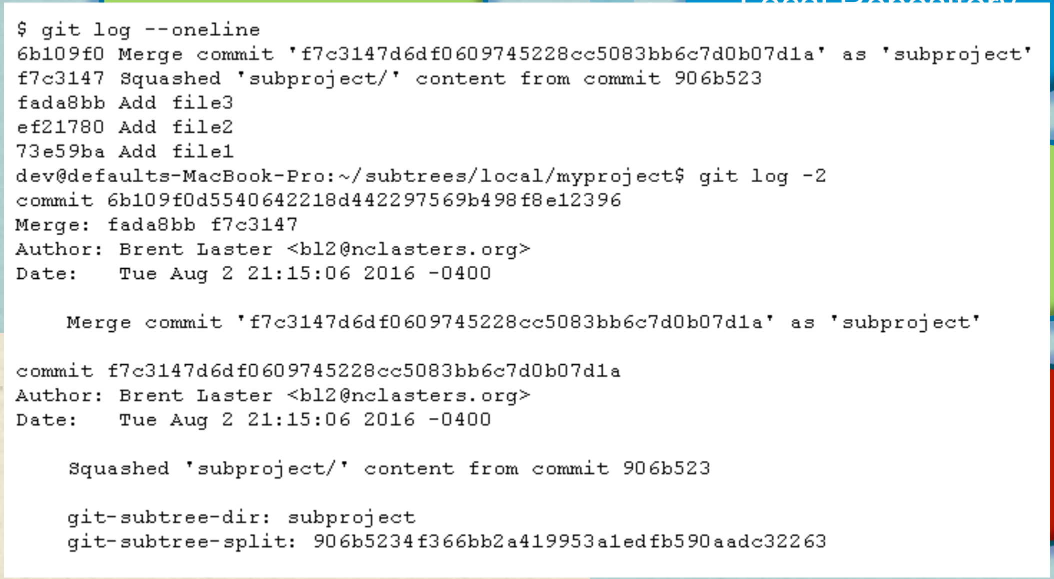
- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use "prefix" option to specify path for subproject
  - Use "squash" option to compress history from remote before adding it
  - branch (master) is optional
- directory listing 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

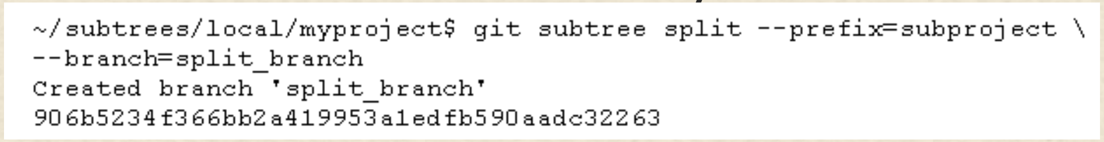




```
$ git log --oneline
6b109f0 Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'
f7c3147 Squashed 'subproject/' content from commit 906b523
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
dev@defaults-MacBook-Pro:~/subtrees/local/myproject$ git log -2
commit 6b109f0d5540642218d442297569b498f8e12396
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: 906b5234f366bb2a419953a1edfb590aadc32263
```

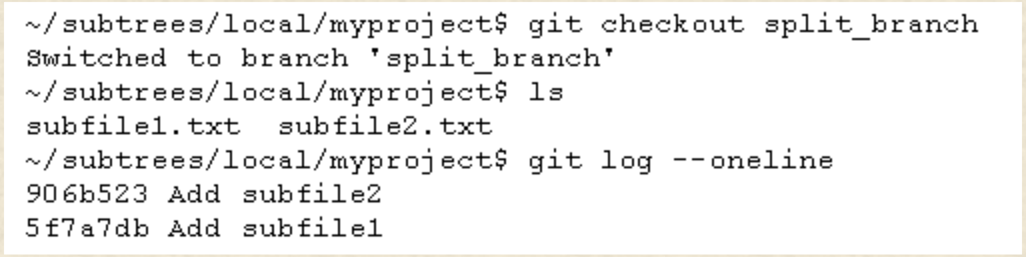
Subtrees - split

- 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

- 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

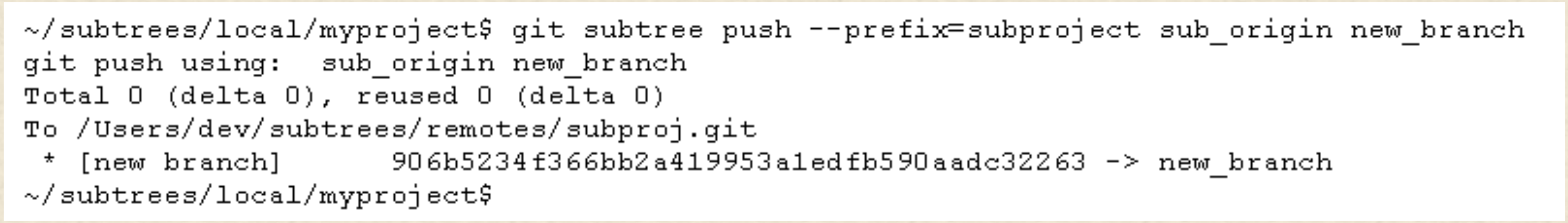
```
~/subtrees/local/myproject$ cd ~/
~$ mkdir newproj
~$ cd newproj
~/newproj$ git init
Initialized empty Git repository in /Users/dev/newproj/.git/
```

  - 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 - push

- 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]      906b5234f366bb2a419953a1edfb590aadc32263 -> new_branch
~/subtrees/local/myproject$
```

Lab 4: Working with Subtrees

Hooks – basic concepts

- Definition – a program or script that runs when a certain event happens
- Common uses
  - Sending email or other notifications when a change is pushed to a repository
  - Validating that certain conventions have been met before a commit
  - Appending items to commit messages
  - Checking the format or existence of certain elements in a commit message
  - Updating content in the working directory after an operation
  - Enforcing coding standards

Hooks - accessing

- By default, hooks come from Git template area and live in .git/hooks
- As of Git 2.9, can have different path to hooks
  - Set by core.hooksPath config value
  - Provides means to have common path to hooks for multiple users or projects (shared hooks)
- After cloning or init, .git/hooks is populated with sample hooks

applypatch-msg.sample	pre-push.sample
commit-msg.sample	pre-rebase.sample
post-update.sample	prepare-commit-msg.sample
pre-applypatch.sample	update.sample
pre-commit.sample	

- Hooks are NOT cloned as part of Git clone
- Can be updated via “git init”

Hooks - attributes

- Domain
  - Local : applypatch-msg, pre-applypatch, post-applypatch, pre-commit, prepare-commit-msg, commit-msg, post-commit, pre-rebase, post-checkout, post-merge, pre-push, post-rewrite, push-to-checkout
  - Remote : pre-receive, update, post-receive, post-update, pre-auto-gc
- Return Code
  - 0 = success; operation should continue
  - Non-0 = not success; operation should abort
- Working Directory Location
  - Non-bare repo – working directory root
  - Bare repo - repository directory
- Environment Variables
  - GIT_DIR, GIT_AUTHOR_DATE, GIT_AUTHOR_EMAIL, GIT_AUTHOR_NAME

Hooks - am

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
am	applypatch-msg	pre-applypatch		post-applypatch	

- am command – takes a patch (may be from email) applies it, and commits the change
- applypatch-msg – operates on commit message for apply part of git am operation (mailing patches)
  - P1 – Name of temporary file with proposed commit message
- pre-applypatch – called after patch is applied, but before committed; allows Git to verify patch results look right
- post-applypatch – called after patch is applied AND committed; useful for notifications

Hooks - commit

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
commit	pre-commit	prepare-commit-msg	commit-msg	post-commit	

- pre-commit - verify that what's about to be committed meets some condition or criteria and is okay to commit
- prepare-commit-msg - Applypatch-msg – operates on commit message for apply part of git am operation (mailing patches)
  - P1: Name of temporary file with proposed commit message
  - P2: Type of operation
  - P3: SHA1 value for certain operations
- commit-msg – called after patch is applied, but before committed; allows Git to verify patch results look right
  - P1: Name of temporary file with proposed commit message
- post-commit – called after patch is applied AND committed; useful for notifications

Hooks - push

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
push	pre-push	pre-receive	update	post-receive	post-update

- pre-push – called prior to a push; can prevent a push
  - P1: Remote reference name
  - P2: Remote URL Extra: STDIN lines of the form <local reference> <local sha1> <remote reference> <remote sha1> LF
- pre-receive – runs once for push before updating references
  - No parameters - Extra: STDIN lines of the form <local reference> <local sha1> <remote reference> <remote sha1> LF
- update – runs once for each reference to be updated
  - P1: Name of reference being updated
  - P2: Old SHA1 value
  - P3: New SHA1 value
- post-receive – runs once after all references are updated; knows old and new
  - No parameters Extra: STDIN lines of the form <old value> <new value> <reference name> LF
- post-update – runs after all references; doesn't know old and new
  - P* (variable number): Name of references being updated

Hooks - rebase

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
rebase	pre-rebase				

- pre-rebase - provides an opportunity to validate that the rebase should go through, issue a warning message, and so on
  - P1: Upstream that the current series of commits come from
  - P2: Branch being rebased (if not the same as P1)
- Example hook – prevents topic branches that have already being merged from being rebased (and so not merged again)

Hooks – push (to non-bare repository)

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
push (to non-bare repo)	push-to-checkout				

- non-bare repository – one that has working directory and staging area and checked out branch
- normally push to bare remote repository only, so receive.denyCurrentBranch setting to prevent pushes to non-bare
- push-to-checkout – allows to override receive.denyCurrentBranch and sync working directory and staging area to make everything consistent
  - P1: Target commit for updating

Hooks – commit --amend or rebase

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
commit – amend or rebase				post-rewrite	

- post-rewrite – runs after operations that change history (rewrite commits)
  - P1: command that called it
  - <stdin> : <old sha1> <new sha1> [<optional extra data>]
- currently no extra data is passed, but might be in future
- runs after commit –amend or rebase (but not filter-branch)
- similar uses to post-checkout and post-merge

Hooks – gc --auto

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
gc --auto	pre-gc-auto				

- gc – run garbage collection (cleaning up objects that aren't used anymore)
  - auto = option to cleanup if there are too many loose objects over a configured threshold
- hook runs first if --auto option is specified
  - No parameters
- can do notification and/or verification before gc

Hooks - checkout

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
checkout				post-checkout	

- post-checkout – runs after a successful checkout (or after clone unless no-checkout option is used)
  - P1: Previous HEAD (before checkout)
  - P2: Current HEAD (after checkout)
  - P3: Checkout type flag: 1= branch, 2 = file
- can be used to cleanout unwanted files

Hooks – merge

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
merge, pull				post-merge	

- post-merge – runs after a successful pull or merge (or after clone unless no-checkout option is used)
  - P1: flag that indicates squash or merge
- can be used to do things like set permissions or kick off supporting processes (such as testing)

Lab 5: Creating a post-commit hook

That’s all - thanks!

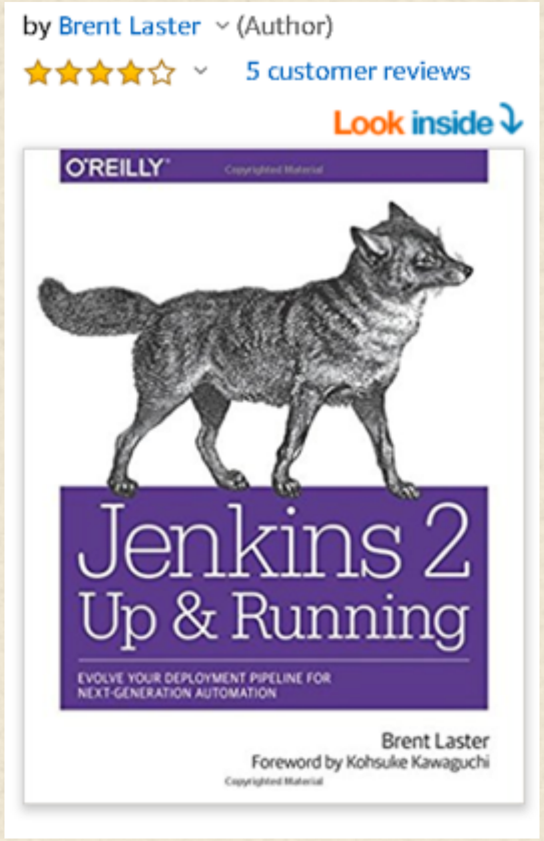
Professional Git 1st Edition

by Brent Laster (Author)

7 customer reviews

Look inside

Join the discussion @ p2p.wrox.com
Wrox Programmer to Programmer™
Professional
Git
Brent Laster



by Brent Laster (Author)

5 customer reviews

Look inside

O'REILLY
Jenkins 2
Up & Running
EVOLVE YOUR DEPLOYMENT PIPELINE FOR NEXT-GENERATION AUTOMATION
Brent Laster
Foreword by Kohsuke Kawaguchi