

Important Prerequisites and Setup

- 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/git-fun-labs.pdf>

(Above is automatic download link. Can also go to

<https://github.com/brentlaster/safaridocs> and get git-fun-labs.pdf.

Git Fundamentals

Brent Laster

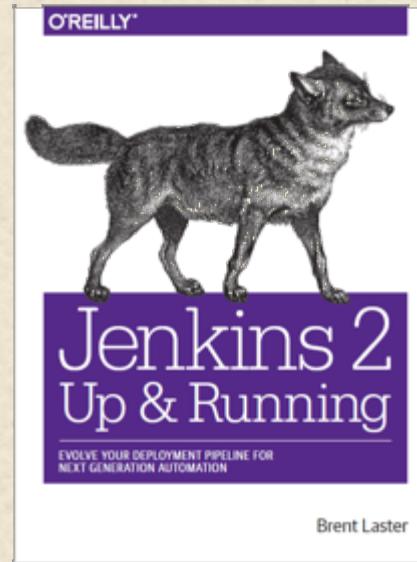
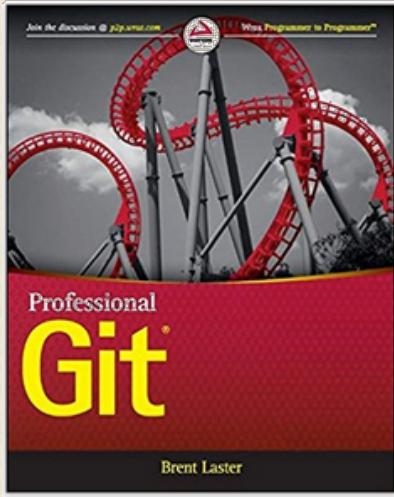
About me

- Senior Manager, R&D
- Part-time trainer
- Git, Gerrit, Gradle, Jenkins
- Author - NFJS magazine, O'Reilly Media, Professional Git book, Jenkins 2 book (April, 2018)
- <https://www.linkedin.com/in/brentlaster>
- @BrentCLaster

Professional Git 1st Edition

by Brent Laster (Author)
4.5 stars - 7 customer reviews

[Look inside](#) ↓



Book – Professional Git

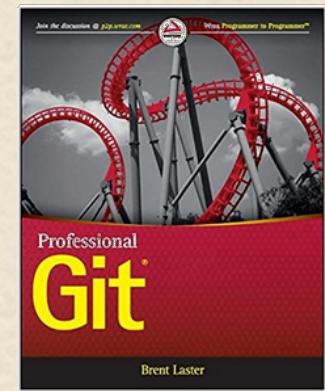
- Reference for material in this class and much more
- First part for non-technical
- Beginner and advanced reference
- More labs
- Please leave a review!

Professional Git 1st Edition

by Brent Laster (Author)

4.5 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.

What is Git?

- Distributed Version Control System
- Open source
- Available for multiple platforms
- Roots in source control efforts for Linux Kernel
- Primary developer – Linus Torvalds
- Been around since 2005

Git Design Goals

- Speed
- Simple design
- Strong support for branching
- Distributed nature
- Ability to handle large projects efficiently
- Disconnected support

Installation

Debian/Ubuntu

```
$ apt-get install git
```

Fedora (up to 21)

```
$ yum install git
```

Fedora (22 and beyond)

```
$ dnf install git
```

FreeBSD

```
$ cd /usr/ports/devel/git
```

```
$ make install
```

Gentoo

```
$ emerge --ask --verbose dev-vcs/git
```

OpenBSD

```
$ pkg_add git
```

Solaris 11 Express

```
$ pkg install developer/versioning/git
```

Ecosystem: Public hosting

- Github, Bitbucket, etc.

The screenshot shows a GitHub repository page for the user 'brentlaster' named 'calc2'. The repository description is 'Simple 2 number web calculator for git demos — Edit'. Key statistics shown include 5 commits, 4 branches, 0 releases, and 1 contributor. A pull request button is visible. A message at the top right says 'File uploading is now available' with a 'Dismiss' button. The repository has 103 forks. At the bottom, there's a call to action to 'Add a README'.

This repository Search Pull requests Issues Gist

brentlaster / calc2 Watch 0 Star 0 Fork 103

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Simple 2 number web calculator for git demos — Edit

5 commits 4 branches 0 releases 1 contributor

Branch: features New pull request New file Upload files Find file SSH git@github.com:brentlaster/calc2 Download ZIP

This branch is 4 commits ahead of master. Pull request Compare

sasbcl add avg function Latest commit b372aa6 on Sep 4, 2012

Dismiss

File uploading is now available You can now drag and drop files into your repositories. Learn more

calc.html add avg function 4 years ago

Add a README

© 2016 GitHub, Inc. Terms Privacy Security Contact Help Status API Training Shop Blog About Pricing

Ecosystem: Self-hosting

- GitLab, Enterprise GitHub

The screenshot shows a self-hosted GitLab interface for a repository named 'Administrator / calc2'. The repository is public, as indicated by the 'Public' button. The main page features a large circular profile picture with a letter 'C' in the center. Below it, the repository name 'calc2' is displayed. The repository statistics show 1 commit, 1 branch, 0 tags, and 0.12 MB of size. There are buttons for 'Add Changelog', 'Add License', and 'Add Contribution guide'. A commit history entry for '32d7b928 initial version · 3 years ago by Brent Laster' is shown. A message at the bottom states 'This project does not have README yet' and provides instructions to add a README file.

Administrator / calc2

Public

C

calc2

1 commit 1 branch 0 tags 0.12 MB Add Changelog Add License Add Contribution guide

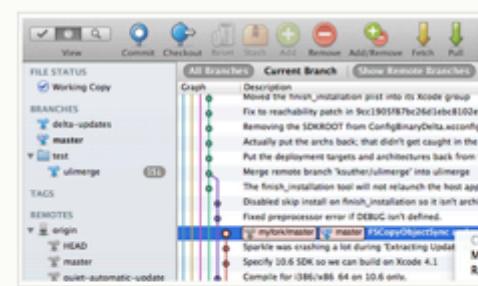
32d7b928 initial version · 3 years ago by Brent Laster

This project does not have README yet

A [README](#) file contains information about other files in a repository and is commonly distributed with computer software, forming part of its documentation.

We recommend you to [add README](#) file to the repository and GitLab will render it here instead of this message.

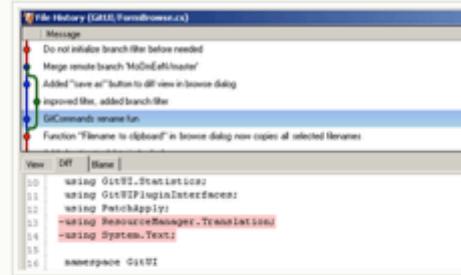
Ecosystem: Ease-of-use Packages



SourceTree

Platforms: Mac, Windows

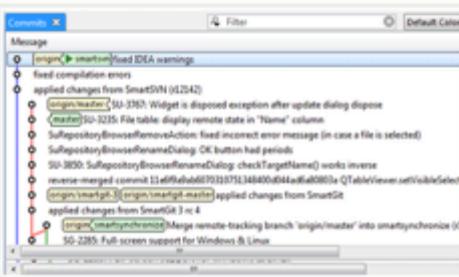
Price: Free



Git Extensions

Platforms: Windows

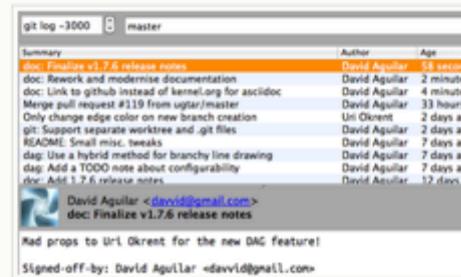
Price: Free



SmartGit

Platforms: Windows, Mac, Linux

Price: \$79/user / Free for non-commercial use



git-cola

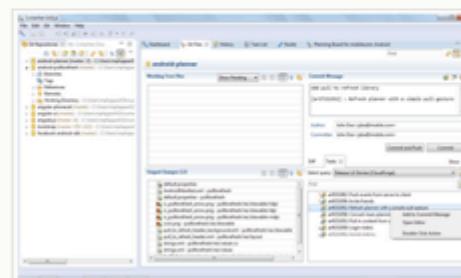
Platforms: Windows, Mac, Linux

Price: Free



GitUp

Platforms: Mac



GitEye

Platforms: Windows, Mac, Linux

Ecosystem: Tools that Incorporate Git

The screenshot shows a web-based interface for managing code reviews. At the top, there's a navigation bar with links for 'All', 'My' (which is selected), 'Projects', 'People', and 'Documentation'. Below the navigation is a search bar and a user dropdown for 'McCoy (Reviewer)'. The main content area displays a pull request titled 'Change 1 - Needs Code-Review' with the commit message 'it's a fact'. The commit ID is I8502da0e6b037cc9dafde806c46c6e6c66f5b301. A reply box contains the text 'Looks fine.' Below the reply box are statistics: -1 0 +1. Underneath, there are radio buttons for 'Code-Review' with options: 'Looks good to me, but someone else must approve' (selected), 'Needs work', and 'Unreviewed'. There are 'Post' and 'Cancel' buttons. A note says 'Post reply (Shortcut: Ctrl-Enter)'. Below the reply box, there's a section for 'Code-Review' with a 'Post' button. The bottom half of the screen shows the commit details: Author and Committer are 'Workshop User <NjjsUser1@gmail.com>', both dated Oct 19, 2015 1:53 PM. The Commit hash is 2f9778a9d7b935990a61fc6139ed7ae668d8c5af. The Parent(s) are cd4e6d1aa38922c9d5135f3463d345efe0506a30. The Change-Id is I8502da0e6b037cc9dafde806c46c6e6c66f5b301. The 'Files' section lists 'captains.log' with a file path, a commit message checkbox (unchecked), and a 'Comments Size' bar showing 1 green segment and +1, -0. The 'History' section shows 'Spock (Contributor)' uploaded patch set 1 at 1:54 PM.

Author	Workshop User <NjjsUser1@gmail.com>	Date
Committer	Workshop User <NjjsUser1@gmail.com>	Oct 19, 2015 1:53 PM
Commit	2f9778a9d7b935990a61fc6139ed7ae668d8c5af	
Parent(s)	cd4e6d1aa38922c9d5135f3463d345efe0506a30	
Change-Id	I8502da0e6b037cc9dafde806c46c6e6c66f5b301	

Files

File Path	Comments Size
captains.log	1 +1, -0

History

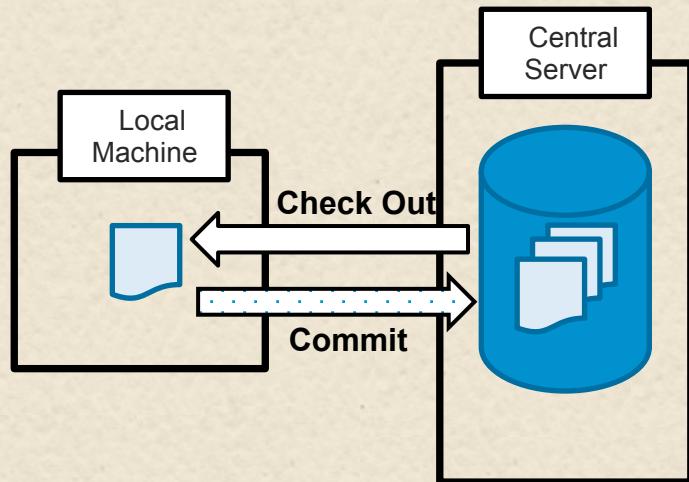
Contributor	Event	Date
Spock (Contributor)	Uploaded patch set 1.	1:54 PM

Git Interfaces

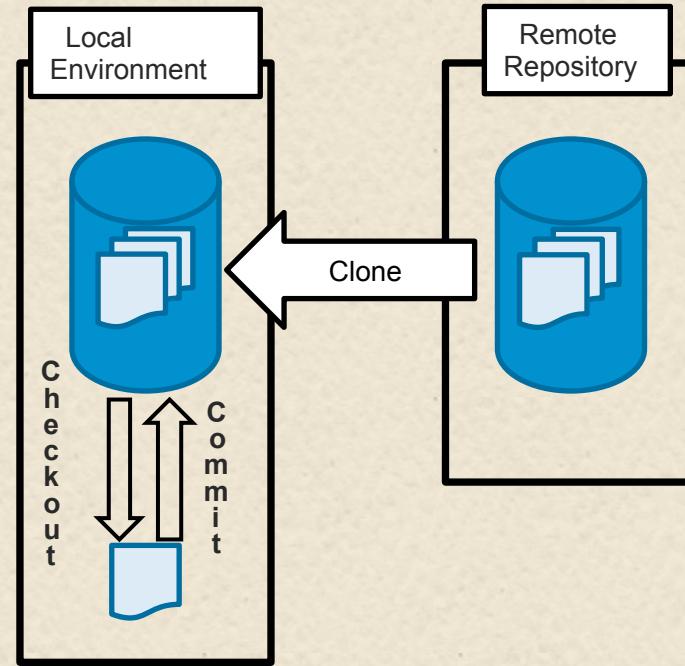
- Many different interfaces exist for working with Git
- Many GUI interfaces
- Command line interface
- Products that script the command line
- Other products such as Gerrit that wrap around Git
- We teach the command line interface because
 - GUI interfaces can all be different - there is not a standard
 - GUI interfaces may change or go away
 - If a GUI interface does not provide some piece of Git functionality, the functionality can be done with the command line
 - If you understand the command line, you can usually find which command in a GUI is used to do the operation

Centralized vs. Distributed VCS

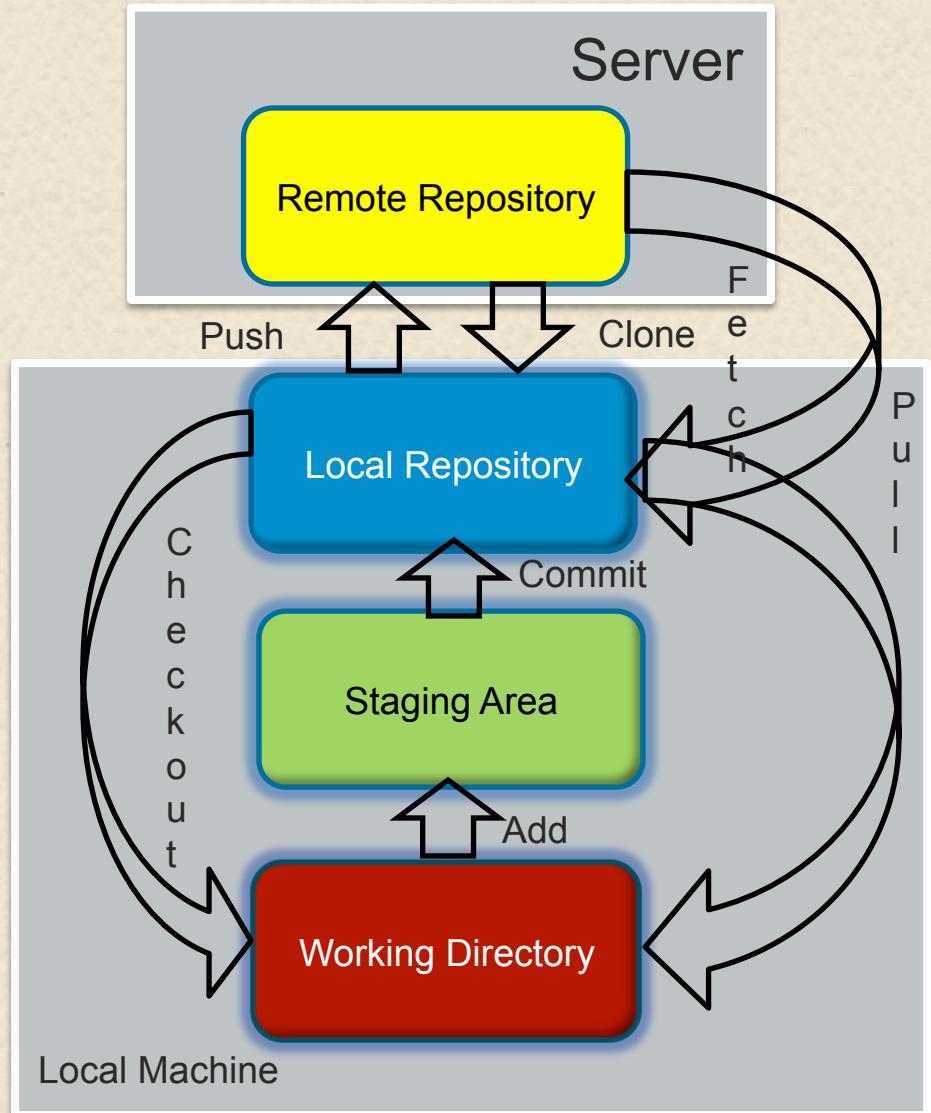
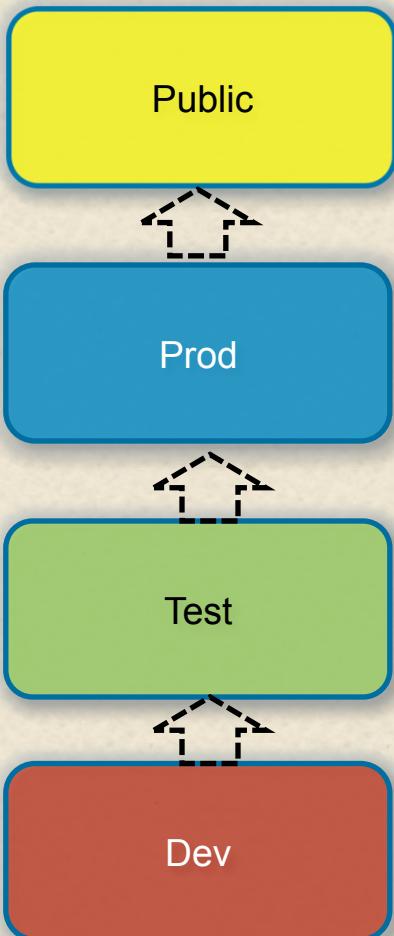
Centralized Version Control Model



Distributed Version Control Model

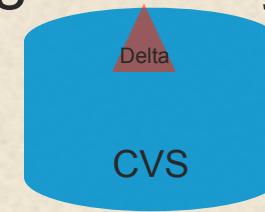


Git in One Picture

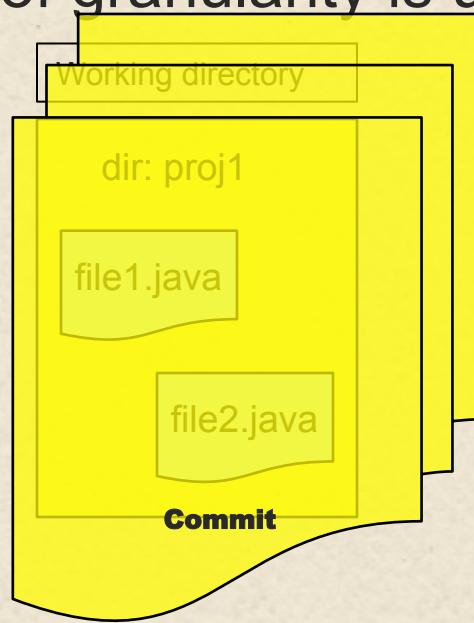


Git Granularity (What is a unit?)

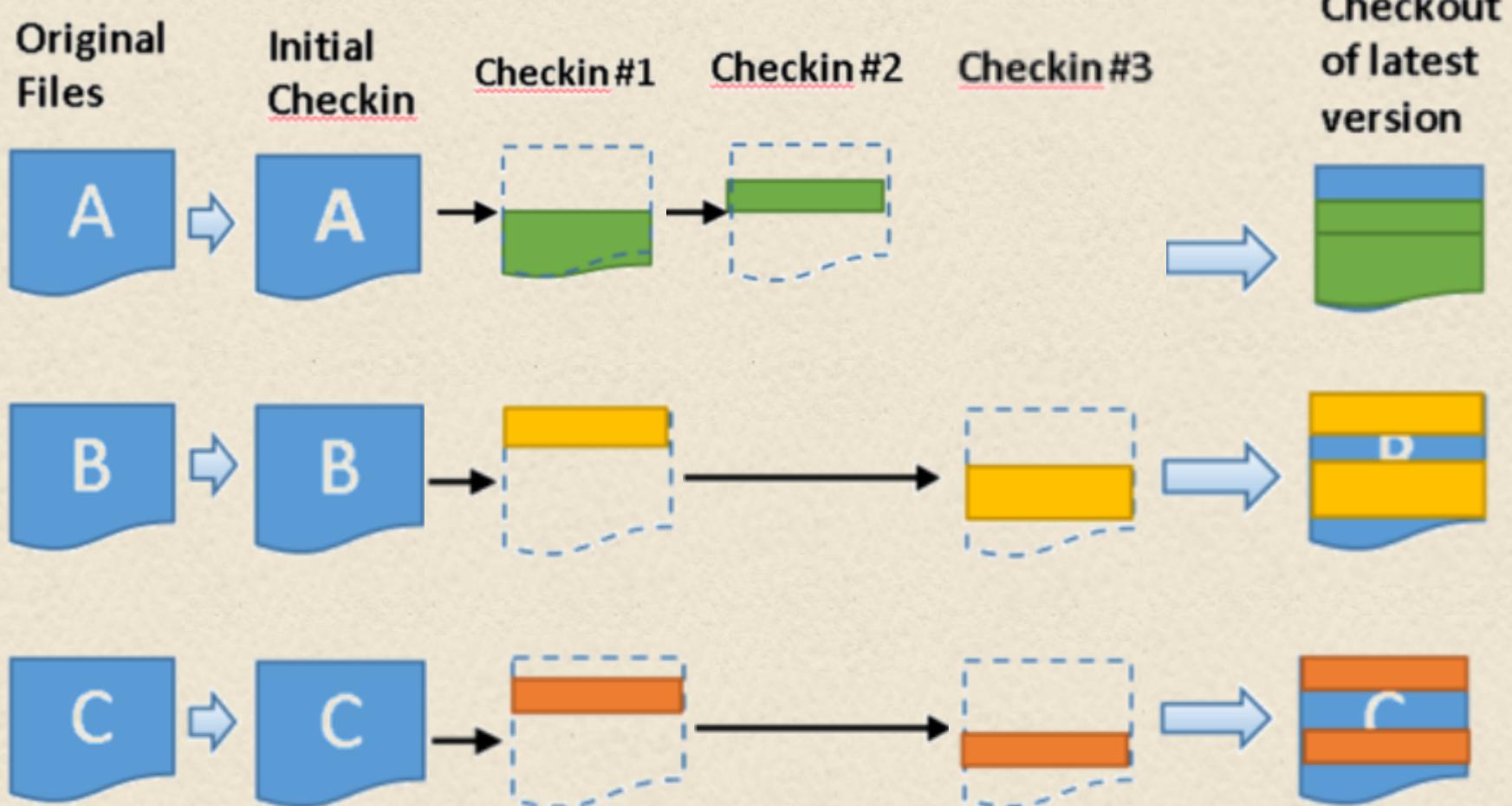
- In traditional source control, the unit of granularity is usually a file



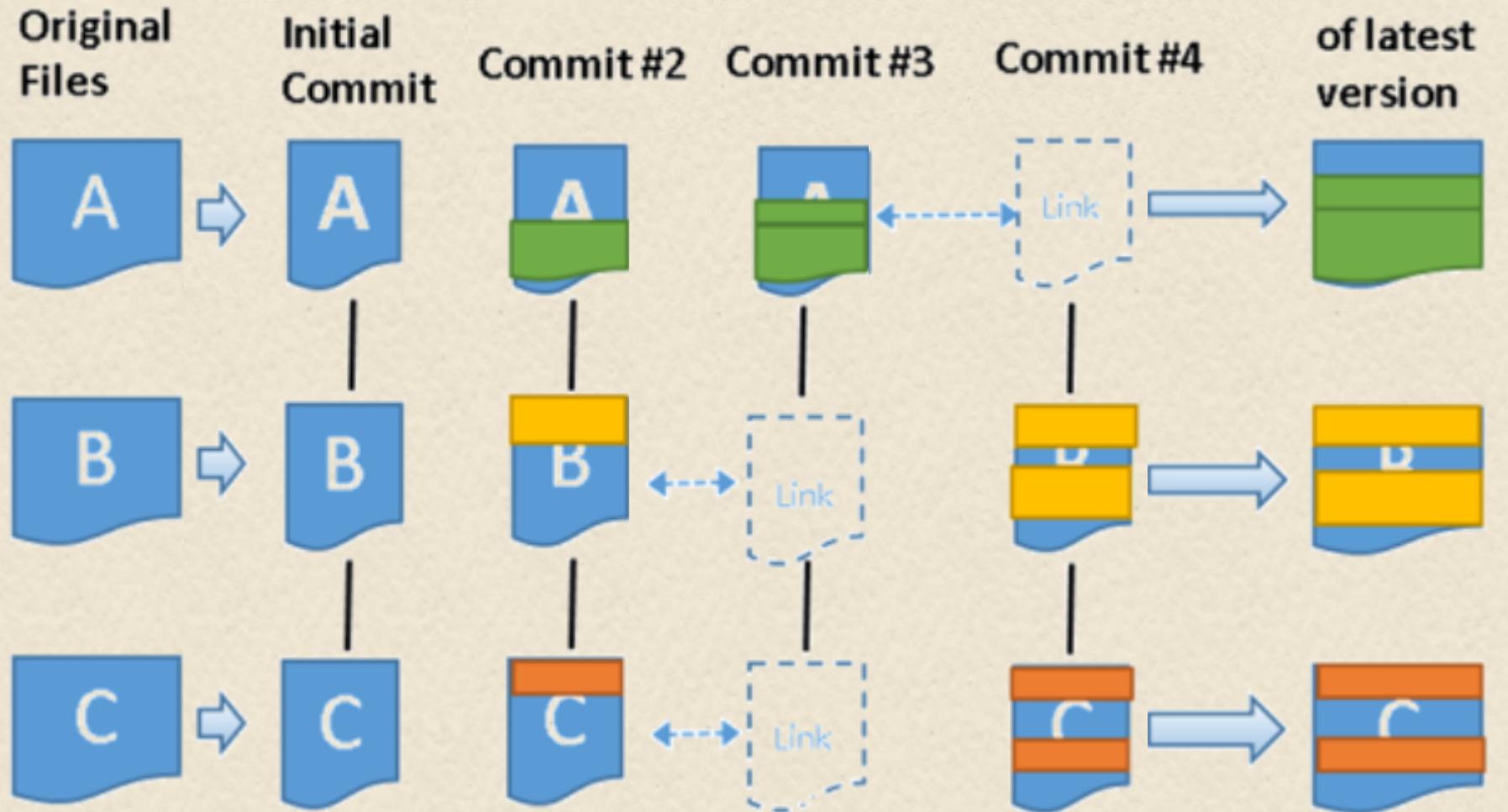
- In Git, the unit of granularity is usually a tree



Delta Storage Model

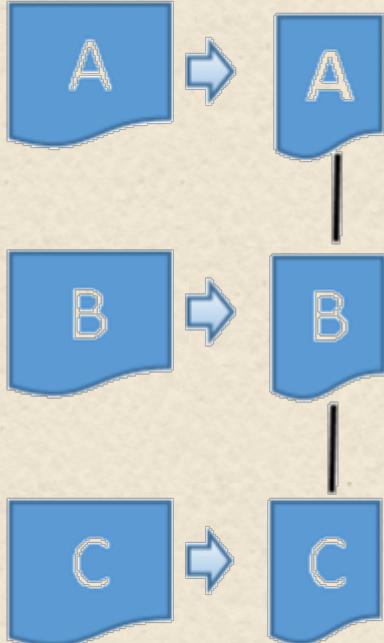


Snapshot Storage Model

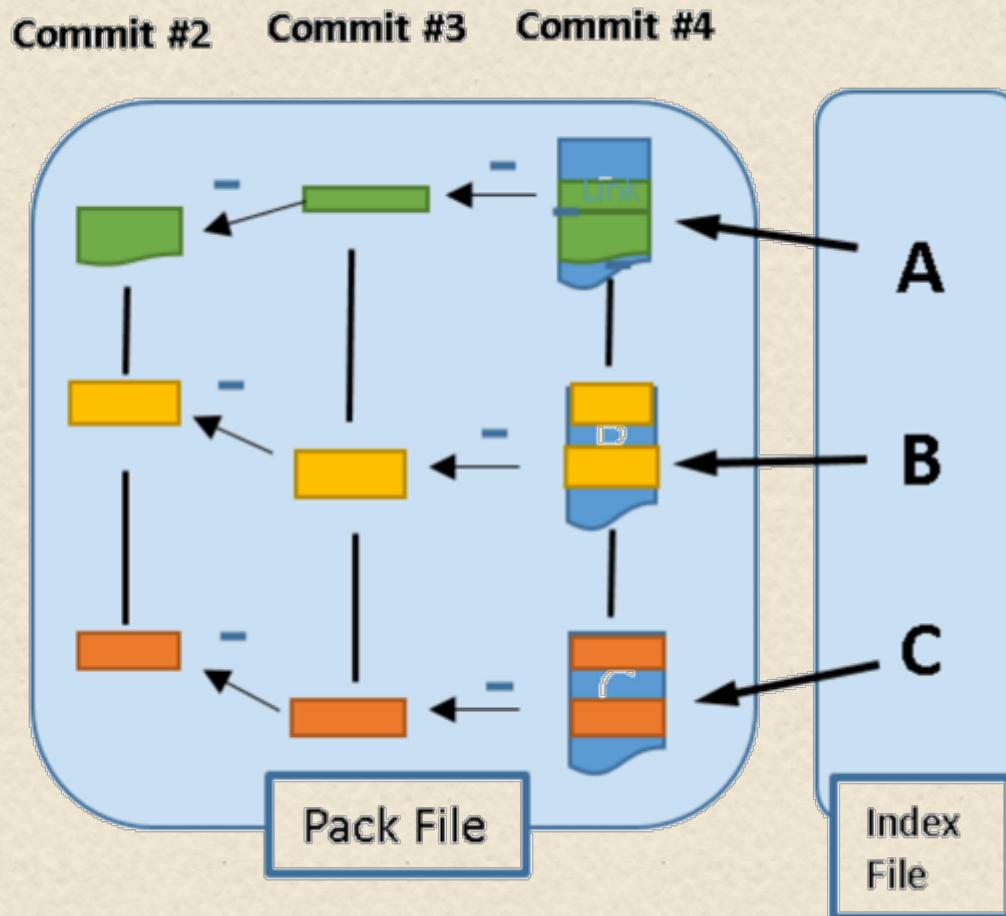


Compressing Files

Original
Files



Initial
Commit



Staging Area: Why?

Prepare

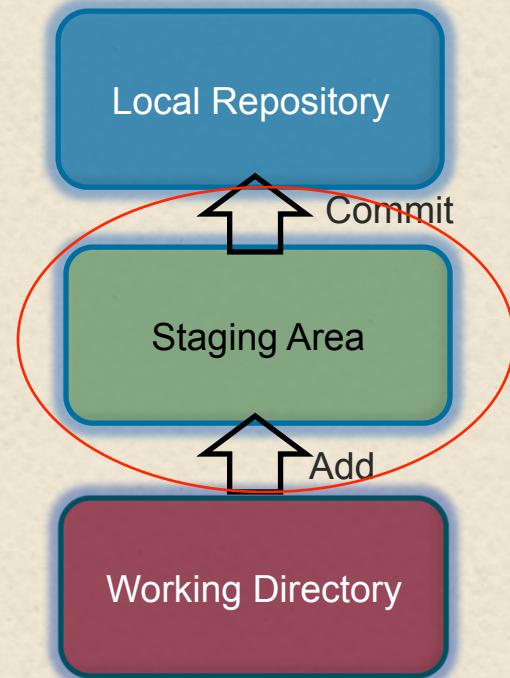
- Stores info about what will go into next commit
- Allows you to control what part of the working tree go into the next commit
- Allows you to queue up individual changes
 - Bundle a collection of specific changes to go into a commit
 - Get them out of your working directory (i.e. “check them off”)
- Helps you split up one large change into multiple commits

Repair

- Allows you to “amend” last commit – redo

Merging with conflicts

- When merge happens, changes that merge cleanly are updated both in working directory and in staging area
- Changes that did not merge cleanly are left in your working directory
- Diff between working directory and staging area shows differences



Example Workflow and Terms

20

1. Modify files in working directory
2. Stage files, adding snapshot to staging area
3. Commit – puts files as they are in the staging area into permanent snapshot in .git

20

Initializing a Repo and Putting Files in

- **Git init**
 - Use in existing directory
 - Creates .git repository skeleton in directory
- **Git add**
 - Start tracking existing files
 - Patterns: git add *.c or git add . (. = files and dirs recursively)
- **Git commit**
 - Uses –m to supply a comment
 - Commits everything unless told otherwise



SHA1



22

```
$ git commit -m "initial add of file1.c"
[master b41b186] initial add of file1.c
 1 file changed, 3 insertions(+)
 create mode 100644 file1.c
```

- Everything in Git is check-summed before it is stored and is then referred to by that checksum. Built into Git low-level functions.
- That means it's not possible to change the contents of any file or directory without Git knowing about it. i.e. avoids file corruption or tampering.
- Mechanism that Git uses for this checksum is called a SHA-1 hash.
- 40-character string of hex characters.
- Calculated based on contents of a file or directory struture.
- Example: **b41b186552252987aa493b52f8696cd6d3b00373**
- Used throughout Git.
- Git stores/points to everything by hash value of the contents in its database.

Remember: You can always get a handle to a snapshot via a SHA1.

Committing Changes

- **Commits to local repo only - not remote**
- **Records a snapshot of your local working area that you can switch back to or compare to later**
- **Requires commit message**
 - Pass by `-m "<message>"`
 - If no `-m`, brings up configured editor so you can type it
 - If you want notepad set as your editor -
<https://github.com/downloads/github/GitPad/Gitpad.zip>
- **Commit output**
 - which branch you committed to
 - what SHA-1 checksum the commit has
 - how many files were changed
 - statistics about lines added and removed in the commit.

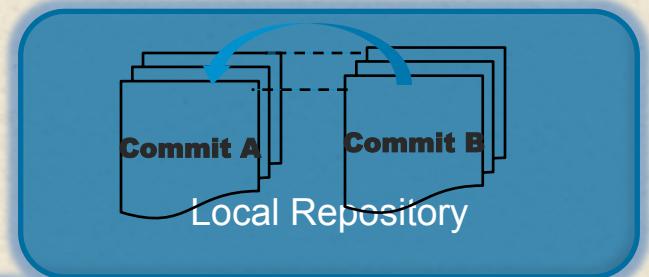
Git Workflow

Staging Area Use: Prepare...

git commit -m “<comment>”

git add .

git add -p



Staging Area



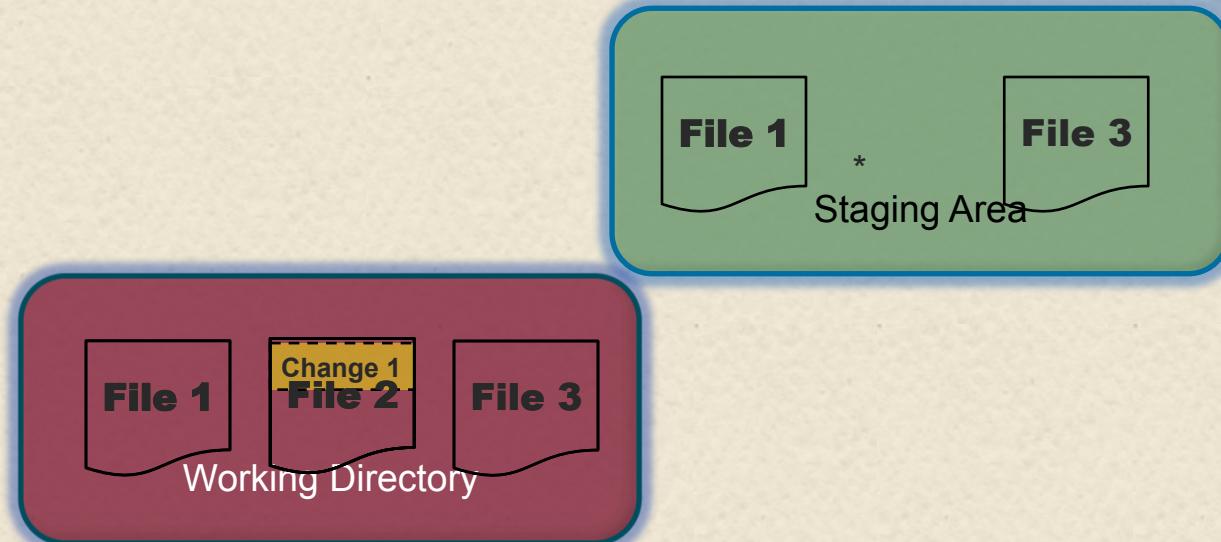
Undoing/Updating Things

- **Git commit --amend**
 - Allows you to “fix” last commit
 - Updates last commit using staging area
 - With nothing new in staging area, just updates comment (commit message)
 - Example:
 - » **Git commit -m “my first update”**
 - » **Git add iforgot.c** (add command stages it into staging area)
 - » **Git commit --amend**
 - » **This updated commit replaces the first (updates one instead of creating a second)**

Git Workflow

Staging Area Use:Repair...

git add .
git commit --amend



Initial User Configuration

- Command: **git config**
- Main things to set
 - User name
`$ git config --global user.name "John Doe"`
 - Email address
`$ git config --global user.email johndoe@example.com`
- Notes
 - Use **--global** to persist options for all repos of this user
 - Check config value with
 - `$ git config --list` (or `git config -l`)
 - `$ git config <property>` as in `git config user.name`
 - Can also set many other things – like diff tool and end-of-line settings

Labs Logistics

Work through the steps – some questions for thought included

Creating Files

In the workshop, we're not concerned about contents of files.
We just need some files to play with.

Lazy way to create a file – in a terminal:

`$ echo stuff > file1.c` (even lazier way `$ date > file1.c`)

Lazy way to append to a file – in a terminal:

`$ echo "more stuff" >> file1.c`

Can use editor if you prefer...

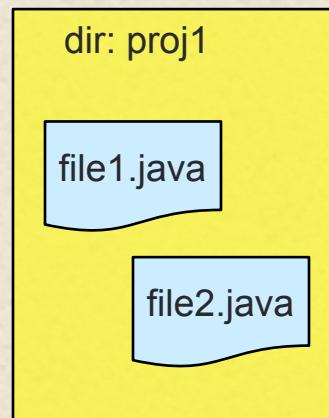
Creating and Exploring a Git Repository and Managing Content

Purpose: In this lab, we'll create an empty Git repository on your local disk and stage and commit content into it.

Git and File/Directory Layouts

Local Repository File Layout

Working directory



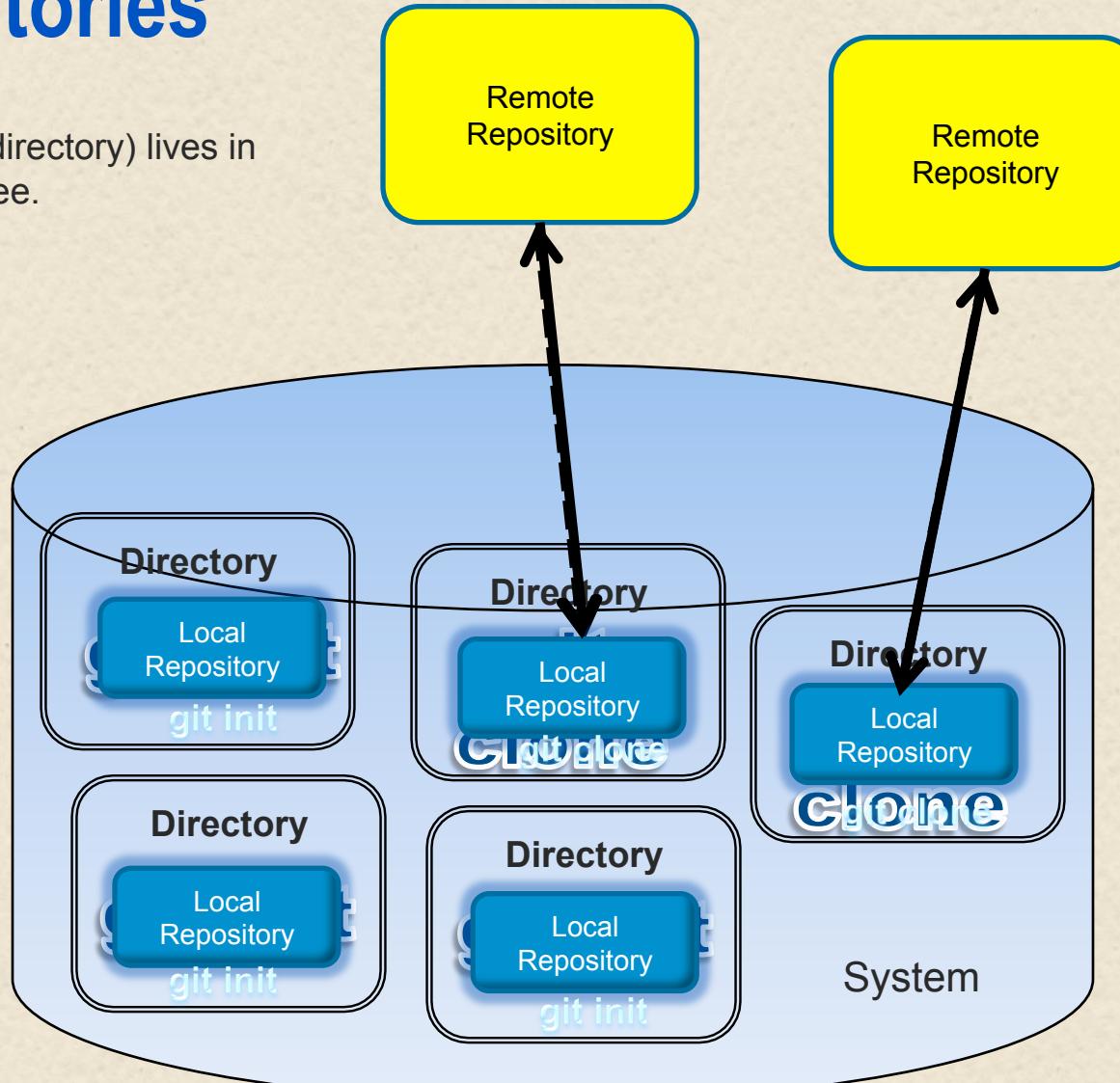
Git snapshot



```
$ tree.sh - * .git
COMMIT_EDITMSG
HEAD
config
description
hooks
  applypatch-msg.sample
  commit-msg.sample
  post-commit.sample
  post-receive.sample
  post-update.sample
  pre-applypatch.sample
  pre-commit.sample
  pre-rebase.sample
  prepare-commit-msg.sample
  update.sample
index
info
exclude
logs
HEAD
refs
  heads
    master
objects
  3a 0e351dc84e32abec5d4dd223c5abdabd57b7f5
  4c 7637a4b66aefc1ee877eaa1afc70610f0ee7cc
  80 7805ea7ae9fdf1b06e876af6e9a69a349b52a3
  88 bee8f0c181784d605eabe35fb04a5a443ae6b7
  ba 2906d0666cf726c7eaadd2cd3db615dedfdf3a
info *
pack *
refs
heads
  master
tags *
```

Starting Work with Git – Multiple Repositories

The repository (.git directory) lives in the local directory tree.



Autocomplete

- Enabled in BASH shell
- Can be enabled in other environments via file in git source
- Git <start of command><tab> (may have to hit it twice if multiple options)
- Git command --<start of option><tab>
- Examples:
 - **Git co <tab><tab>**
 - **Git com <tab>**
 - **Git list --s<tab> (note double –'s)**

Getting Help

- Git <command> -h

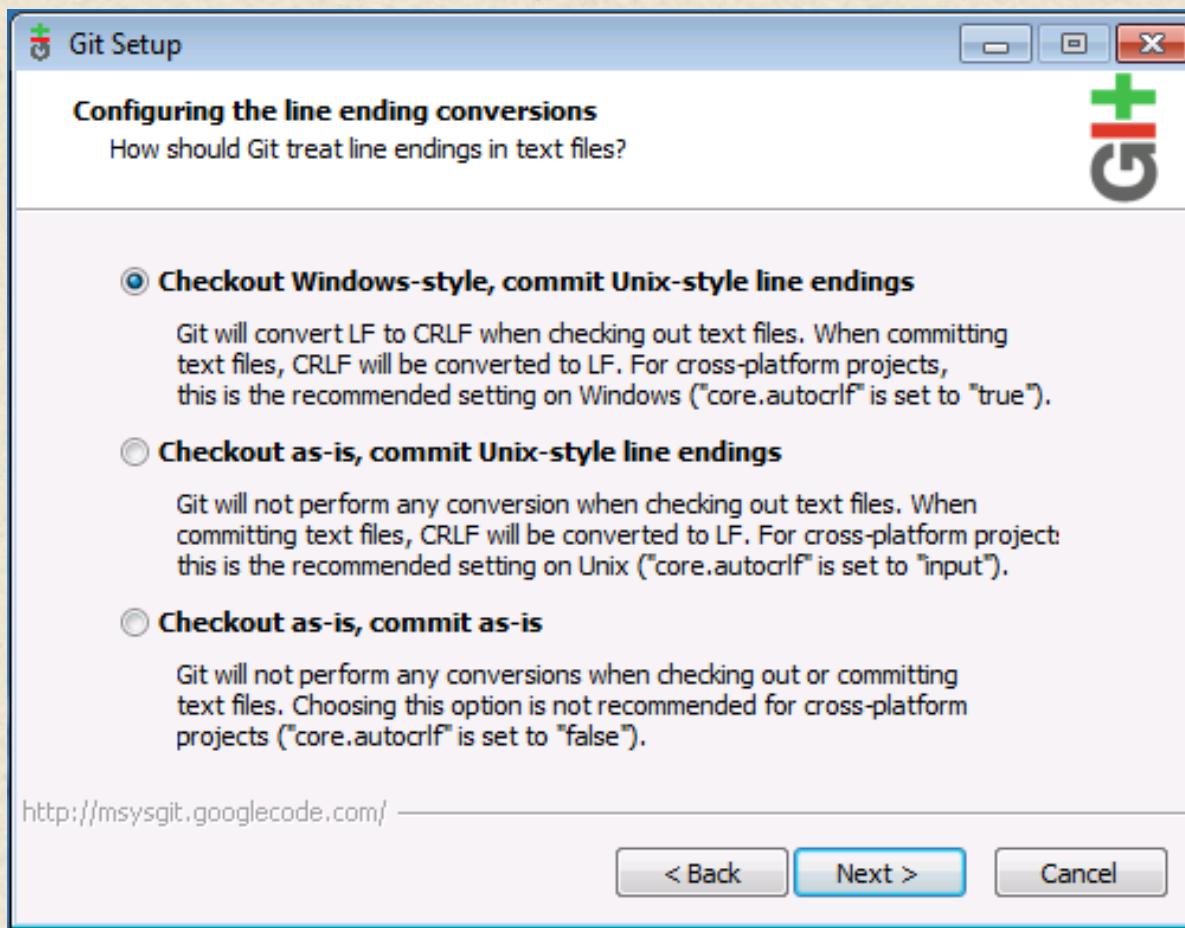
Brings up on screen list of options

- Git <command> --help
- Git help <command>

Brings up html man page

Line Endings

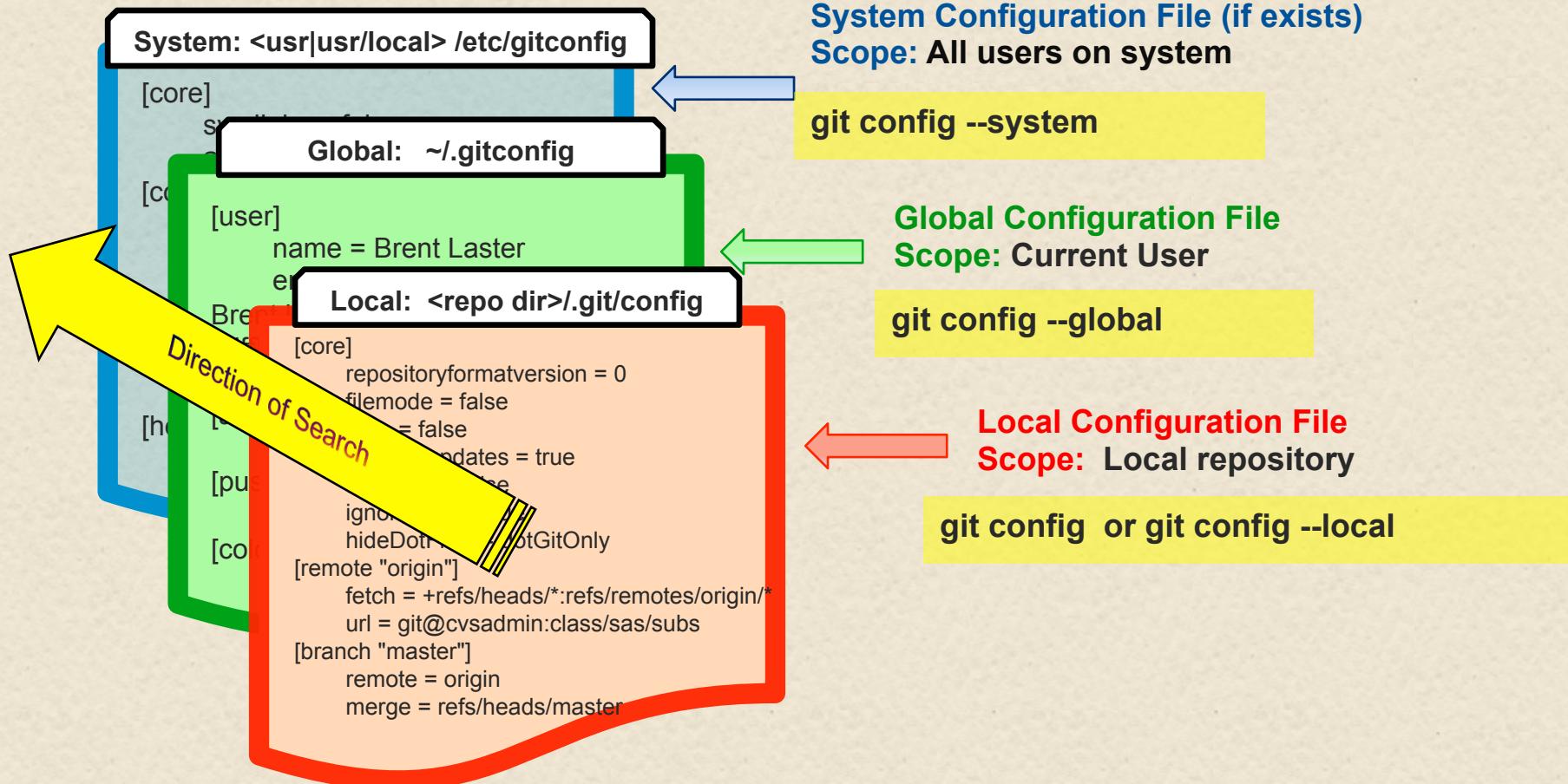
- Git config core.autocrlf



Git Config file

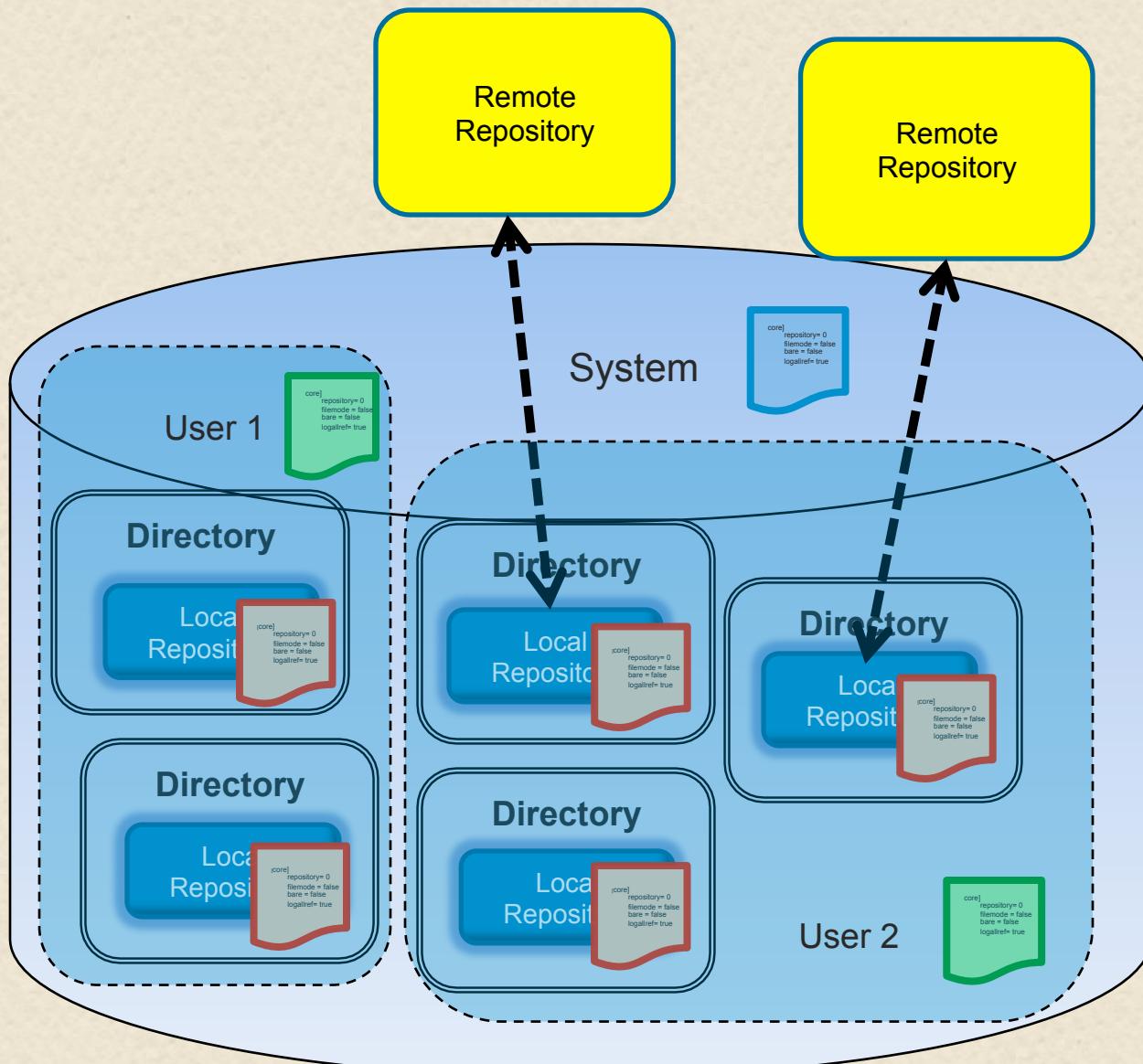
- **Scope**
 - Local (repo)
 - » `.git/config`
 - Global (user)
 - » `~/.gitconfig`
 - System (all users on a system)
 - » `<usr|usr/local> /etc/gitconfig`
- **Divided into sections**
- **Can have user settings, aliases, etc.**
- **Text file that can be edited, but safer to use git config**

Understanding Git Configuration Files Scope



Git Configuration Layout – Multiple Configs

37



37

Terminology: File States in Working Directory

Files in your working directory can be in one of two main states:

- **Tracked**
 - Were in the last snapshot (last commit)
 - Can be
 - Unmodified - same as what's in Git
 - Modified - different from what's in Git
 - Staged
 - OR -
- **Untracked**
 - Everything else
 - Not in last snapshot
 - Not in staging area

3 questions to think about for determining status

- Is Git aware of the file (is it in Git)? tracked or untracked
- What is in the staging area?
- What is the relationship of the latest version in Git to the version in the working directory?

File Status Lifecycle

1. Files start out in wd (rev a)

Git aware? No – **untracked** files

Anything staged? No = Changes **not staged** for commit

Git compared to WD: N/A

2. (Git) Add file.

Git aware: Yes – **tracked**

Anything staged? Rev a staged = Changes **to be committed**

Git compared to WD: Same - **unmodified**

3. Edit in working directory (rev b)

Git aware: Yes – **tracked**

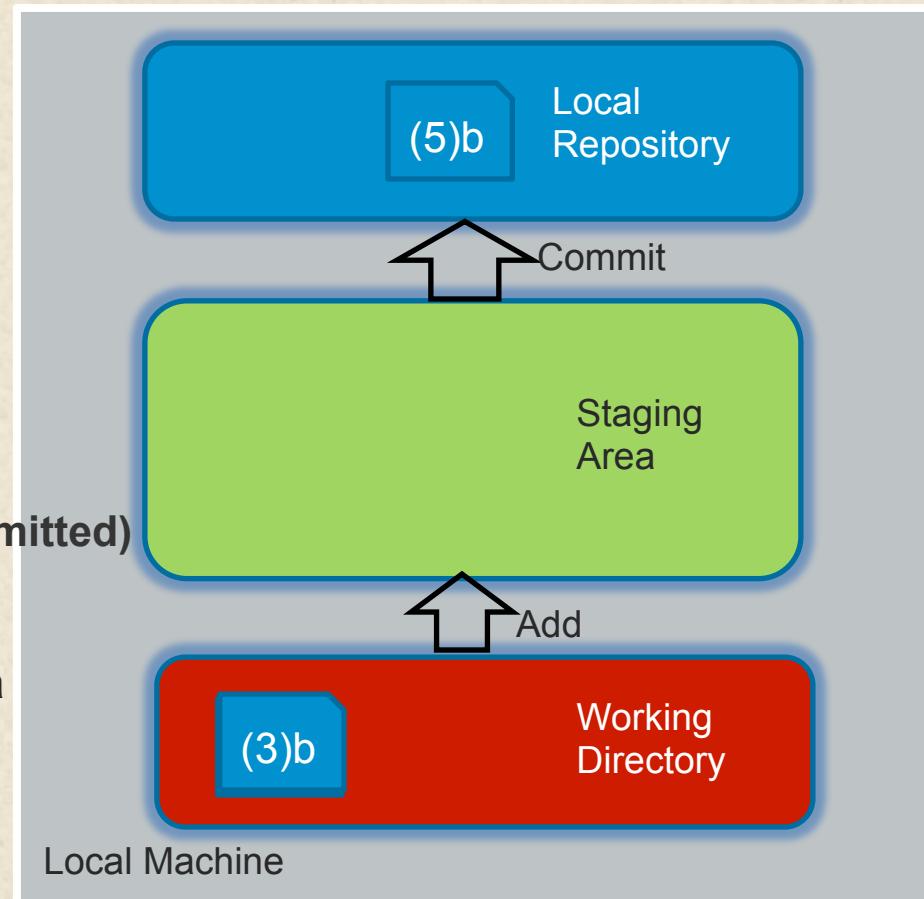
Anything staged? Rev a **staged (to be committed)**

Git compared to WD : Different – **modified**

4. (Git) Add file. **Tracked**, rev b **staged** (rev a overwritten), **unmodified**

5. (Git) Commit. **Tracked**, Staged: Nothing, **unmodified**

“nothing to commit (working directory clean)”



* “Changes to be committed” = staged

* “Changes not staged for commit” = wd

Git Status

- **Command: git status**
 - Primary tool for showing which files are in which state
 - -s is short option (<1 or more characters><filename>
 - » ?? = untracked
 - » M = modified
 - » A = added
 - » D = deleted
 - » R = renamed
 - » C = copied
 - » U = updated but unmerged
 - -b option – always show branch and tracking info
- **Common usage: git status -sb**

Updating a file that Git knows about

- How do we update a file that git knows about?
 - Stage it (`git add .`)
 - Commit it (`git commit`)
 - Shortcut syntax : `git commit -am`
“<comment>”
- » Doesn't add new files (untracked ones)

Git Special References: Head and Index

- **Head**
 - Snapshot of your last commit
 - Next parent
 - Pointer to current branch reference (reference = SHA1)
 - **Think of HEAD as pointer to last commit on current branch**
- **Index (Staging Area)**
 - Place where changes for the next commit get registered
 - Temporary staging area for what you're working on
 - Proposed next commit
- **Cache - old name for index**
- **Think of cache, index, and staging area as all the same**

Showing Differences

- **Command : `git diff`**
- **Default is to show changes in the working directory that are not yet staged.**
- **If something is staged, shows diff between working directory and staging area.**
- **Option of `--cached` or `--staged` shows difference between staging area and last commit (`HEAD`)**
- **`git diff HEAD` shows differences between working directory and what `HEAD` points to – usually last commit in Local Repository**

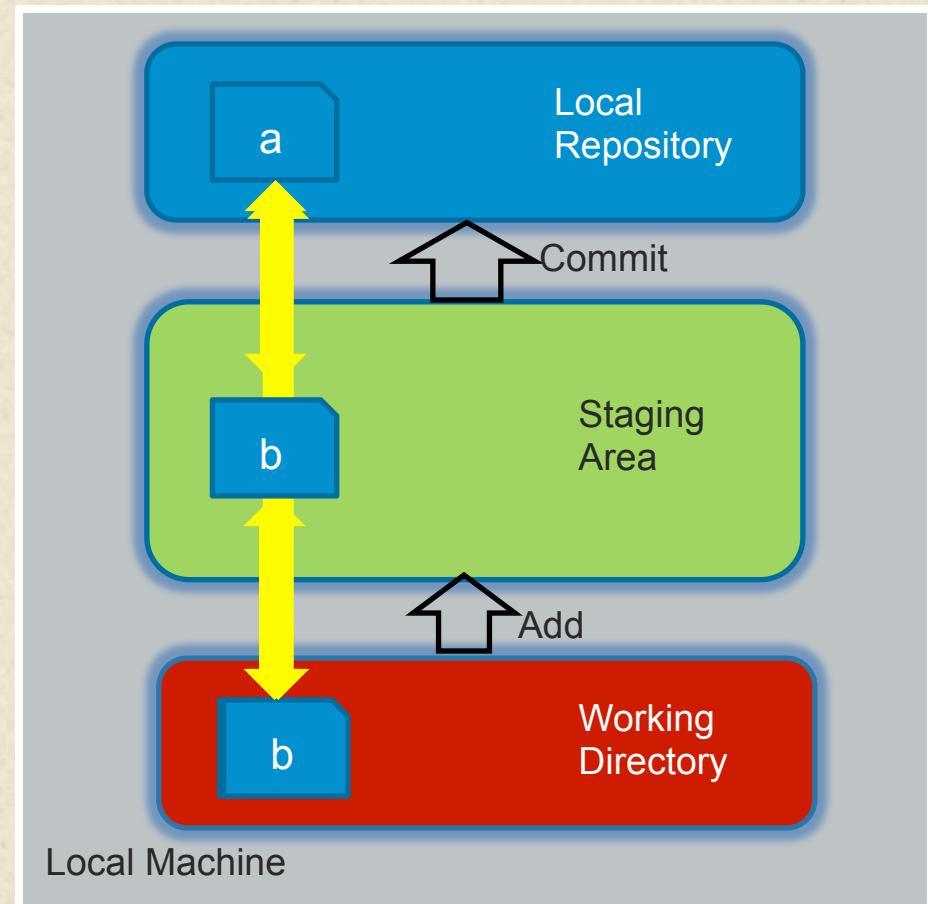
Understanding File Diffs

3. Assume file (rev a) created, added and
4. Detachate compare to local staging
Commit needs to be staged WD -> stashed
against index/local Repo.
WD -> local staging

Git diff compares:
git diff --staged
 WD (not in index area there)
 WD vs. Attached local Repo
 Revs vs. Index to Local Repo
 Revs vs. Different local Repo
 Outputs differences between WD and
 Revs vs. Differences between Staging
 Local Repo Differences between WD and
 Area and Not local Repo
 Local Repo

```
MINGW32:~/diffpp
$ git diff --staged
diff --git c/file1.txt i/file1.txt
index 257cc56..12955d3 100644
--- c/file1.txt
+++ i/file1.txt
@@ -1 +1,2 @@
 foo
+more

sasbc1@L75088 ~/diffpp (master)
$
```



Tracking Content through the File Status Lifecycle

Purpose: In this lab, we'll work through some simple examples of updating files in a Local Environment and viewing the status and differences between the various levels along the way.

History

- **Command: `git log`**
- **With no options, shows name, sha1, email, commit message in reverse chronological order (newest first)**
- **-p option shows patch – or differences in each commit**
 - Shortcut : `git show`
- **-# -shows last # commits**
- **--stat – shows statistics on number of changes**
- **--pretty=oneline|short|full|fuller**
- **--format – allows you to specify your own output format**
- **--oneline and --format can be very useful for machine parsing**
- **Time-limiting options --since|until|after|before (i.e. --since=2.weeks or --before=3.19.2011)**
- **Gitk tool also has history visualizer**

Sample git log format

- **git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short**
- **In detail...**
 - **--pretty="..." defines the output format.**
 - **%h is the abbreviated hash of the commit**
 - **%d commit decorations (e.g. branch heads or tags)**
 - **%ad is the commit date**
 - **%s is the comment**
 - **%an is the name of the author**
 - **--graph tells git to display the commit tree in the form of an ASCII graph layout**
 - **--date=short keeps the date format short and nice**

Git log examples

- **git log --pretty=oneline --max-count=2**
- **git log --pretty=oneline --since='5 minutes ago'**
- **git log --pretty=oneline --until='5 minutes ago'**
- **git log --pretty=oneline --author=<your name>**
- **git log --pretty=oneline --all**
- **git log --oneline --decorate**

Git Aliases

- Can type directly in the .gitconfig file
- Can set using
 - git config alias.<name> <operation>
- Example
 - git config --global alias.co checkout
 - After, “git co” = “git checkout”
- Example aliases (as they would appear inside config file)

[alias]

co = checkout

ci = commit

st = status

br = branch

hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short

Seeing History Visually

- Gitk – graphical interface that comes with most git
- Shows history of local repository (not remote) in working directory
- Also useful to be able to see how changes look graphically

The screenshot shows the Gitk graphical interface for viewing git history. The main pane displays a tree of commits, with the current commit selected. The commit details pane shows the author (Jos Backus), committer (Scott Chacon), date (2009-01-30 17:49:40), and message: "Make the number of bytes to be read from git's stdout configurable.". The commit details also show the SHA1 ID (2a8c38b72380fc64700d0e8e6d7b16d45dfb6ebd), a search bar, and buttons for 'Patch' and 'Tree'. The bottom pane shows the diff for the file lib/grit.rb, comparing revision 7c1785d..fe3d8b5 against the previous commit. The diff highlights changes in the code, such as the addition of a module Grit and changes to the attr_accessor definitions.

```

gitk: gitk
master | Make the number of bytes to be read from git's stdout configurable.
remotes/origin/master | test for current head
| dammit, this is the second time this has reverted
| modified index to create refs/heads if it is not there
| Add diff-lcs dependency
| Add dependency for Open4
| merged recent changes:
| updated the Manifest file
| fixed alternates to accept relative paths and changed f
| added sha() def to commit object
Merge branch 'idx2'
| merged in bryces changes and fixed some testing is
| Clarify how to get a full count out of Repo#comm
| Annoyance: time sleep took file from balloon
| Author: Jos Backus <jos@catnook.com> 2009-01-30 17:49:40
| Committer: Scott Chacon <schacon@gmail.com> 2009-01-30 18:23:08
| Parent: 66938d0e3329c70ebe598c2246a0e6af90d04646 (test for current head)
| Branch: master
| Follows: v0.7.0
| Precedes:

Author: Jos Backus <jos@catnook.com> 2009-01-30 17:49:40
Committer: Scott Chacon <schacon@gmail.com> 2009-01-30 18:23:08
Parent: 66938d0e3329c70ebe598c2246a0e6af90d04646 (test for current head)
Branch: master
Follows: v0.7.0
Precedes:

Make the number of bytes to be read from git's stdout configurable.

Signed-off-by: Scott Chacon <schacon@gmail.com>

index_7c1785d..fe3d8b5_100044
@@ -22,11 +22,19 @@ module Grit
  include GitRuby

  class << self
-    attr_accessor :git_binary, :git_timeout
+    attr_accessor :git_binary, :git_timeout, :git_max_size
  end

-  self.git_binary = "/usr/bin/env git"

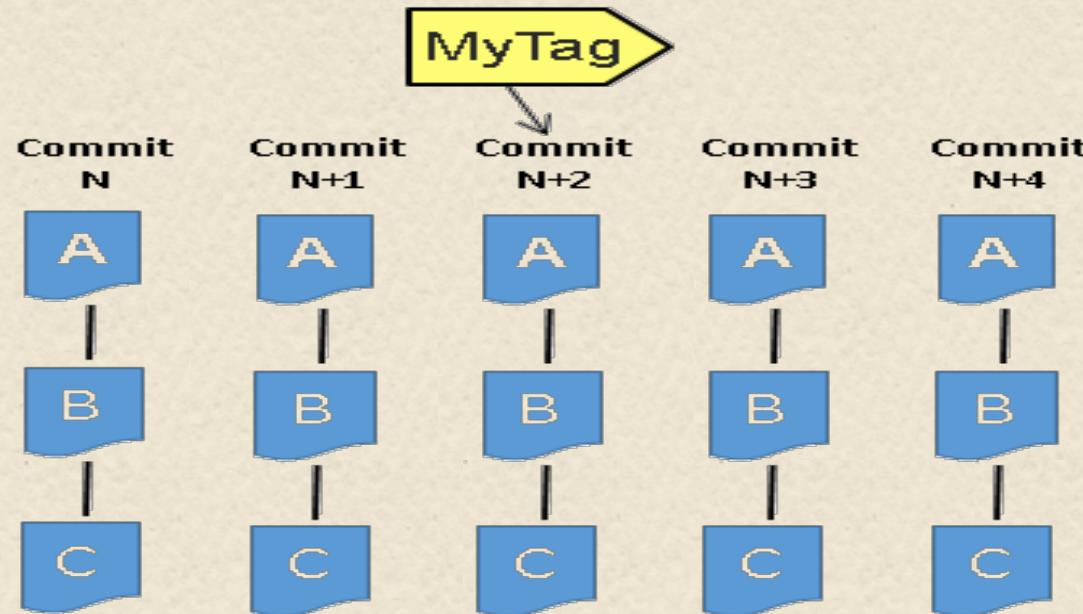
```

Tags in git

- Command: **git tag**
- Two types
 - Lightweight – like regular tag, pointer to specific commit
 - Annotated – stored as full object in the database
 - » Checksummed
 - » Contain full email and date, tagger name, and comment
 - » Created by `git tag -a <tag> -m <message>`
- Create a tag
 - `Git tag <tagname> <hash>`
- Show tags
 - `Git tag` (lists tags out)
 - `Git show <tag>` - shows detail

Notes about Tags

- You tag commits (hashes), not files
- A tag is a reference (pointer) to a commit that stays with that commit
- To reference a file via a tag, you need to qualify with the file name (since the tag refers to entire commit)



Working with Changes Over Time and Using Tags

Purpose: In this lab, we'll work through some simple examples of using the Git log commands to see the flexibility it offers as well as creating an alias to help simplify using it. We'll also look at how to tag commits to have another way to reference them.

Supporting Files - .gitignore

- Tells git to not track files or directories listed in a file named .gitignore
- Operations such as **git add .** will skip files listed in .gitignore
- Rules
 - Blank lines or lines starting with # are ignored.
 - Standard glob patterns work.
 - You can end patterns with a forward slash (/) to specify a directory.
 - You can negate a pattern by starting it with an exclamation point (!).
- What types of things might we want git to ignore?

Supporting Files - .gitattributes

- A git attributes file is a simple text file that gives attributes to pathname – meaning git applies some special setting to a path or file type.
- Attributes are set/stored either in a .gitattributes file in one of your directories (normally the root of your project) or in the .git/info/attributes file if you don't want the attributes file committed with your project.
- Example use: dealing with binary files
- To tell git to treat all obj files as binary data, add the following line to your .gitattributes file:
 - *.obj -crlf -diff
 - (Think of the dash/hyphen as “don't do it”)
 - With this setup, git won't try to convert or fix eol issues. It also won't try to compute or print a diff for changes in this file when you run git show or git diff on your project.
 - In the 1.6 series of Git and beyond, you can also use a macro that is provided that means -crlf -diff:
 - » *.obj binary

Git Attributes File – Example

- Basic example

```
# Set default behaviour, in case users don't have core.autocrlf set.  
* text=auto  
  
# Explicitly declare text files we want to always be normalized and converted  
# to native line endings on checkout.  
.c text  
.h text  
  
# Declare files that will always have CRLF line endings on checkout.  
.sln text eol=crlf  
  
# Denote all files that are truly binary and should not be modified.  
.png binary  
.jpg binary
```

- Advantage is that this can be put with your project in git. Then, the end of line configuration now travels with your repository. You don't need to worry about whether or not all users have the proper line ending configuration.
- Some sample gitattributes files in GitHub for certain set of languages.

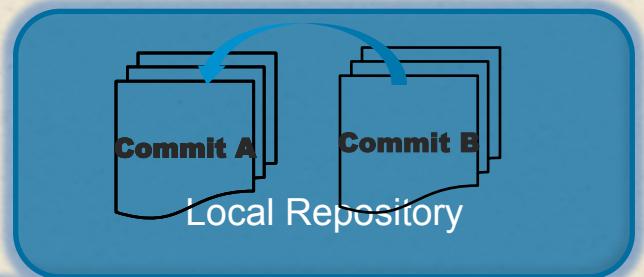
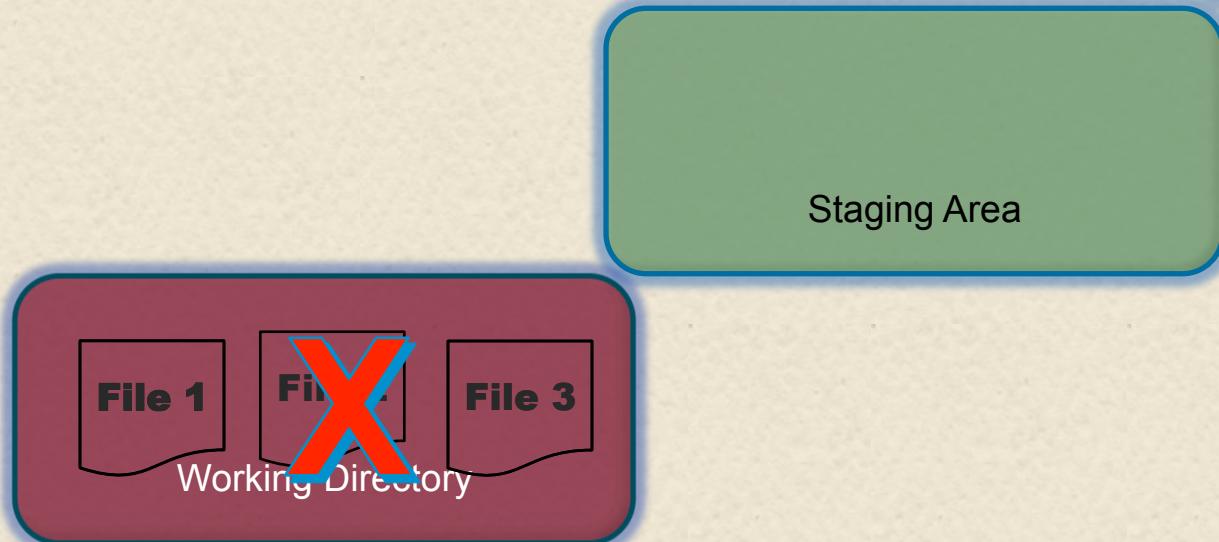
Removing files

- **Command: `git rm`**
- **Process: Remove from tracked files, stage deletion and then commit**
- **Git rm**
 - Removes it from your working directory (via `rm <file>`) so it doesn't show up in tracked files
 - Stages removal
- **If you have a staged version AND a different modified version, git will warn you**
 - Use the `-f` option to force the removal
- **Can remove just from the staging area using `--cached` option on `git rm`**
- **Can provide files, directories, or file globs to command**

Git Workflow for operations like rm

1. Git performs the operation in the working directory
2. Git stages the change
3. User commits to finalize the operation in the repository.

git rm file2
git commit -m “finalize rm”



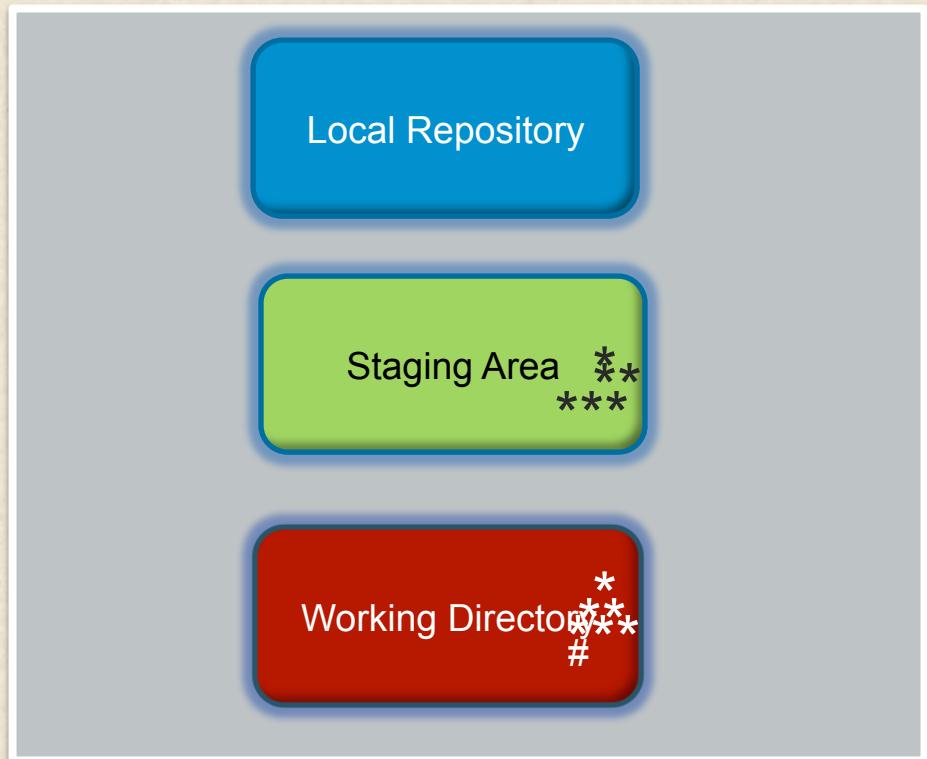
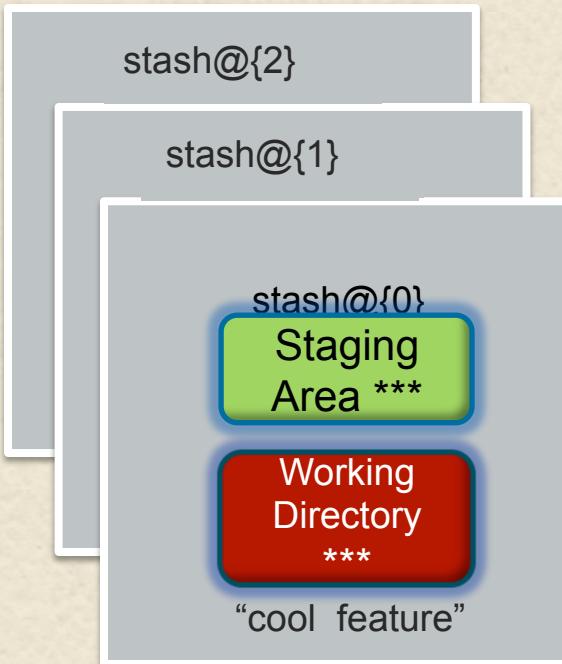
Discarding Changes

- Local changes before staging
 - use git checkout
- Changes already staged
 - git reset HEAD <filename>
 - » Doesn't change contents of directory
 - » Can use manual edit or checkout to update directory
- Getting back to a clean working directory AND staging area
 - git reset --hard HEAD
- Think of reset as a “rollback” (moves HEAD pointer back to a previous commit but does not delete existing commits)
- Supporting command: **git stash**
 - saves off state
 - use git stash pop or git stash apply to get old state back

Git Stash

- Keep a backup queue of your work
 - command: **git stash**
 - saves off state
 - use **git stash pop** or **git stash apply** to get old state back

```
$ git stash apply "cool feature"
```

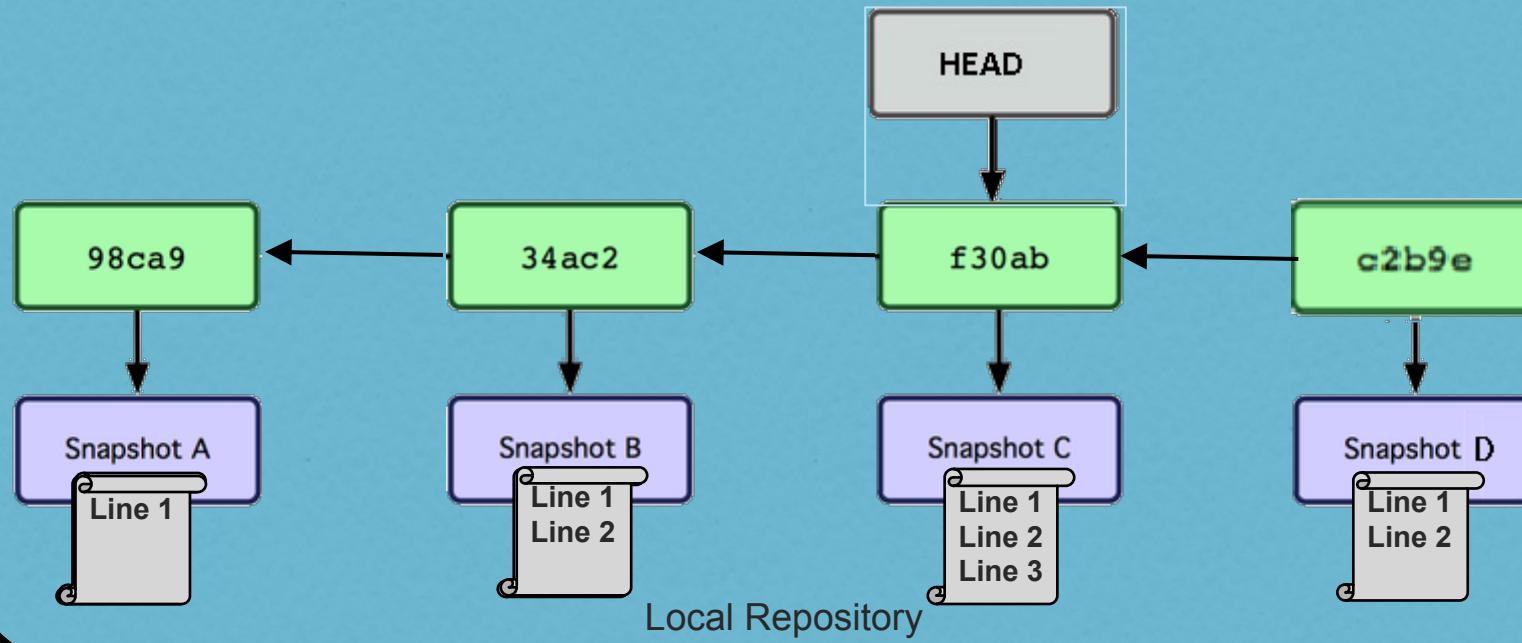


Git Revert

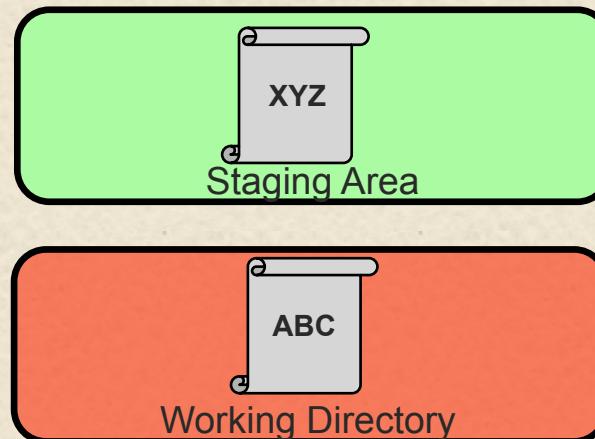
- Net effect is similar to reset, but...
 - Doesn't move pointer to a previous commit
 - Instead, creates new commits that “cancel” out the undesired ones
- Big advantage is that history isn't changed or deleted, it's added on to
- Less surprises and complications for other users who have to merge with the code
- Has a --no-commit option that will only make changes in working tree and index

Reset and Revert

62



- git reset --hard HEAD~2
- git reset 34ac20b [--mixed]
- git revert HEAD~1



62

- **Git doesn't track metadata about renames, but infers it**
- **Git mv command renames a file**
- **Git will show “renamed” in status after a mv**
- **The mv command is equivalent to:**
 - **mv (old local file) (new local file)**
 - **git rm old file**
 - **git add new file**

What is a branch in source control?

- Line of development
- Collection of specific versions of a group of files tagged in a common way
 - `cvs rtag -a -D <date/time> -r DERIVED_FROM -b NEW_BRANCH PATHS_TO_BRANCH`
- End result is I have an easy “handle” to get all of the files in the repository associated with that identifier – the branch name
 - `cvs co -r BRANCH_NAME PATHS`
- So – what you end up with in your working directory when you check out a branch is a set of specific versions of the files from the group, or a ...

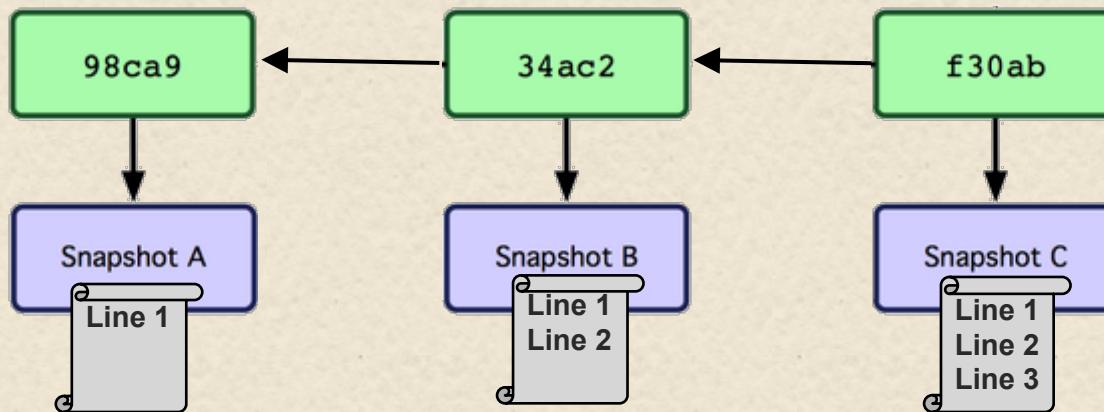
Snapshot! What is a snapshot (in GIT)?

65

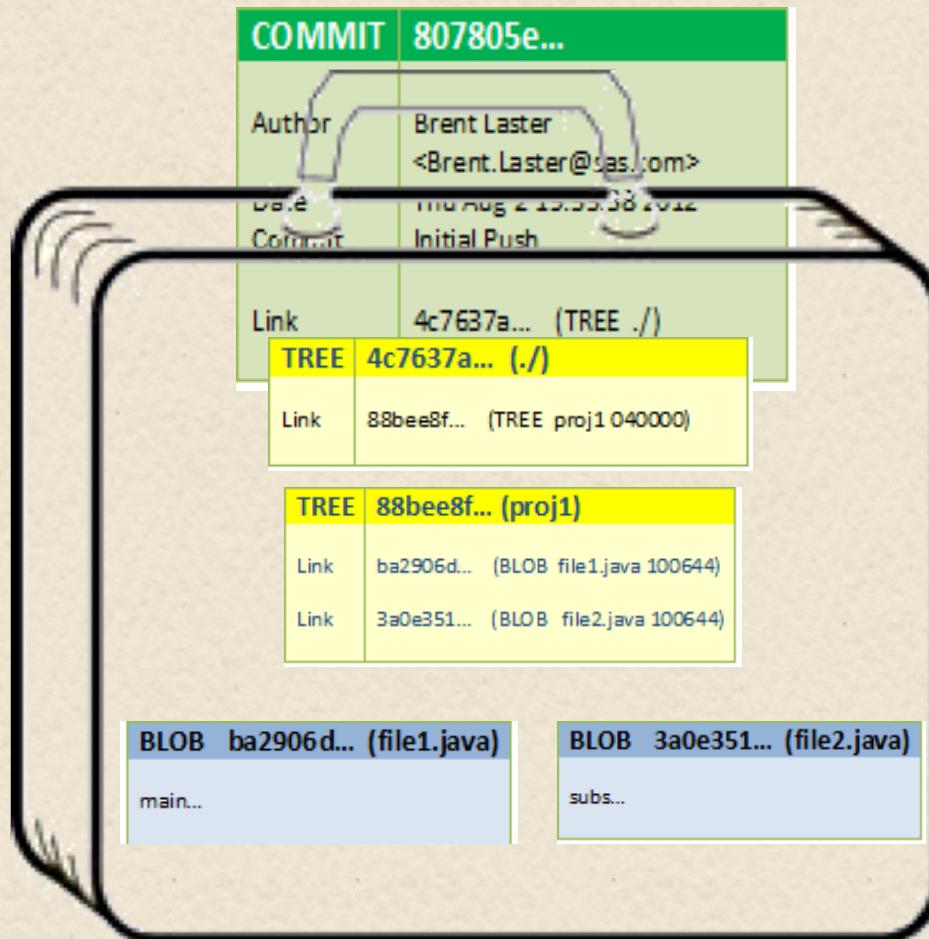
- Line of development associated with a specific change
- Collection of specific versions of a group of files associated with a specific commit
- End result is I have a “handle” to get all of the versions of files in the repository associated with that commit – handle = SHA1 for that commit
- What you end up with in your working directory when you check out a branch is a set of specific versions of the files from the group.

Branches

- Remember, git stores data as a series of snapshots, not deltas or changesets
- Commits point to snapshots – and the commit that came before them

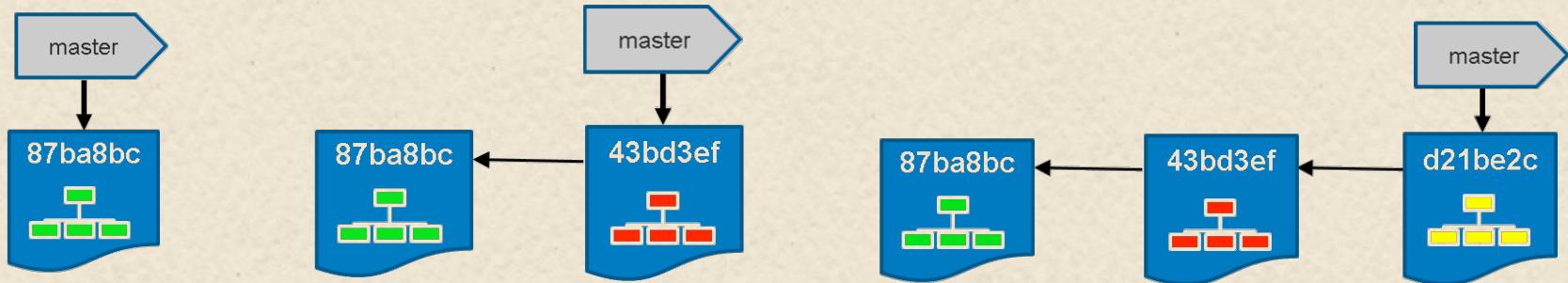


Commit SHA1's are a handle to a snapshot⁶⁷



Lightweight Branching

- A branch in git is simply a lightweight, movable pointer to a commit
- Default branch is named master
- As you initially make commits, you're given a branch pointer that points to the last commit you made. Every time you commit, it moves forward automatically.



- A branch in Git is just a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and easy as writing 41 bytes to a file (40 characters and a newline).
- And, because we're recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do.

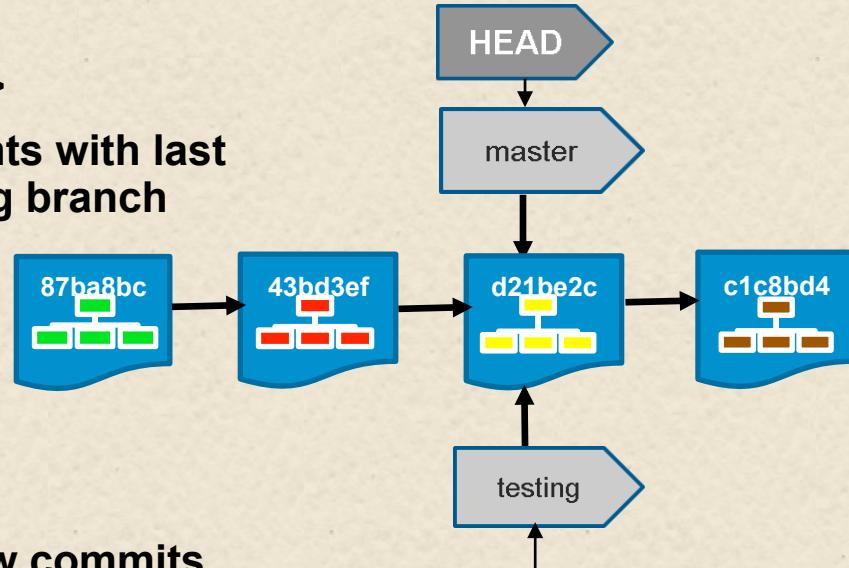
Creating and using a new branch

- Create a branch : **git branch <branch>**
 - Creating a new branch creates a new pointer

Git keeps a special pointer called HEAD that always points to the current branch.

git branch testing

- Change to a branch: **git checkout <branch>**
 - Moves HEAD to point to <branch>
 - Updates working directory contents with last commit from <branch> - if existing branch



git checkout testing

- Branch pointers advance with new commits

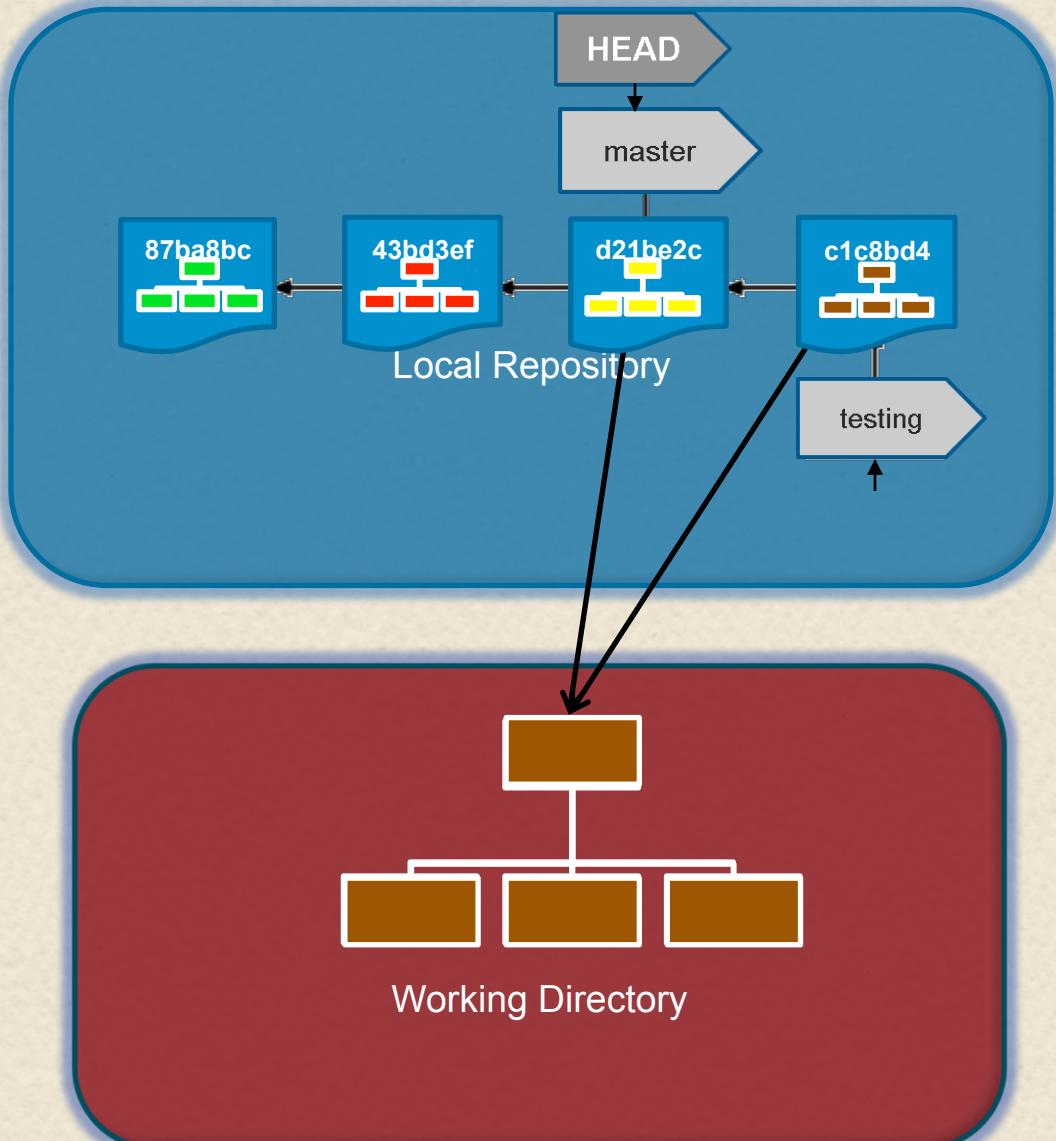
Switching between Branches

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



Which Branch am I on/in?

- Command: **git branch <no arguments>**
- “*” is indicator of which one you’re on
- Prompt will also change in some configurations
 - If it doesn’t, can be setup.

“Topic and Feature Branches”

- **Topic Branches**
 - Term for a temporary branch to try something out
 - Easy to try things in or come back to later
 - Generally, create simple name based on type of work you're going to try – i.e. web_client
 - Maintainer of a project may “namespace” these – as in adding initials on the front – i.e. abc/web_client
 - Create just as any other branch
 - \$ git branch abc/web_client
 - \$ git checkout -b web_client
- **Feature Branches**
 - Term for a branch to develop a feature
 - Intended for limited lifetime
 - Merge back into main line of development

Working with Branches

Purpose: In this lab, we'll start working with branches by creating a new branch and making changes on it.

Merging Branches

- Command: **git merge <branch>**
- Relative to current branch
- Current branch is branch being merged into
- **<branch>** is branch being merged from
- Ensure everything is committed first

Merging: What is a Fast-forward?

- 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
```

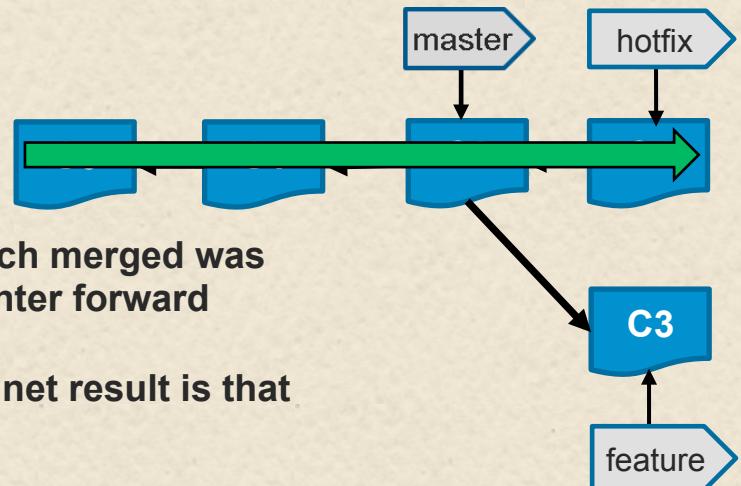
```
$ git merge hotfix
```

Updating f42c576..3a0874c

Fast Forward

README | 1-

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



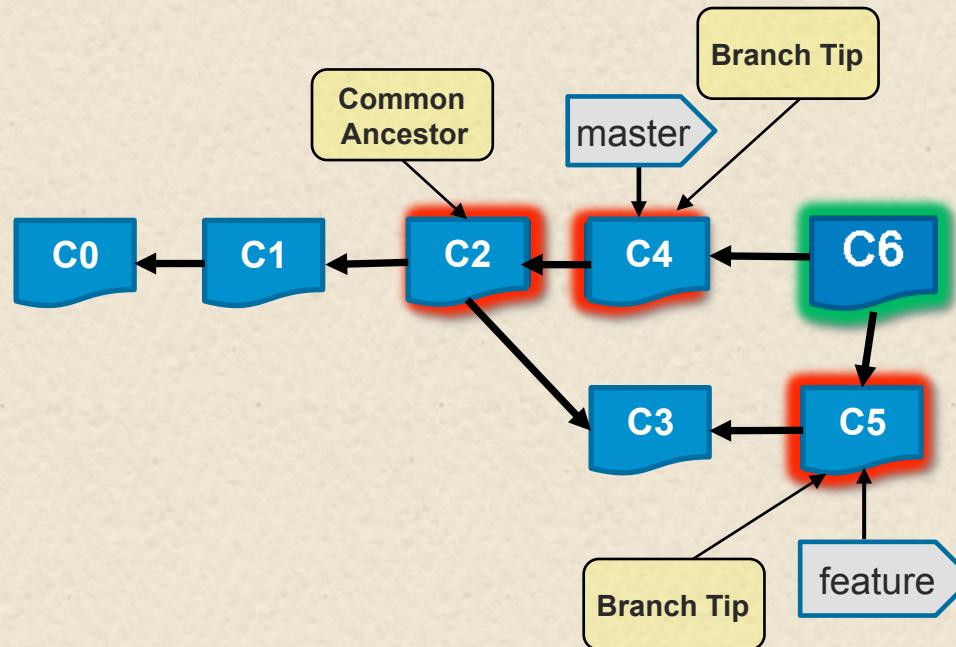
About “Fast Forward” – because commit pointed to by branch merged was directly “upstream” of the current commit, Git moves the pointer forward

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

Merging: What is a 3-way Merge?

76

- 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”



```
$ git checkout master  
$ git merge feature
```

Merge Conflicts

77

- Like CVS
- Will get CONFLICT message
- Git pauses merge in place
- To see which files are unmerged, use git status – will see “unmerged: or both modified: <filename>”
- Uses <<<< and >>> markers
- After resolution, run git add and then commit
- Can also run git mergetool to resolve graphically

Merge Scenario – Branches with Conflicts

\$ git checkout master

\$ git merge feature

\$ git status

Changes to be committed:

modified: File 1

modified: File 3

Unmerged paths

both modified: File 2

[fix conflicts]

\$ git add .

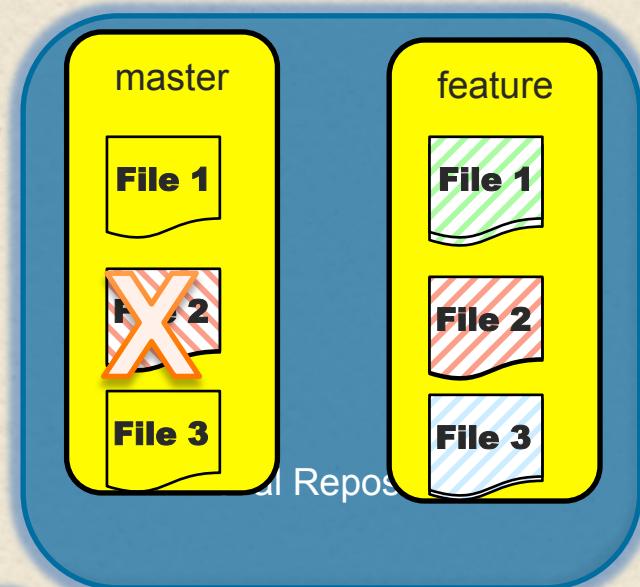
\$ git commit -m

“finalize merge”



Working Directory

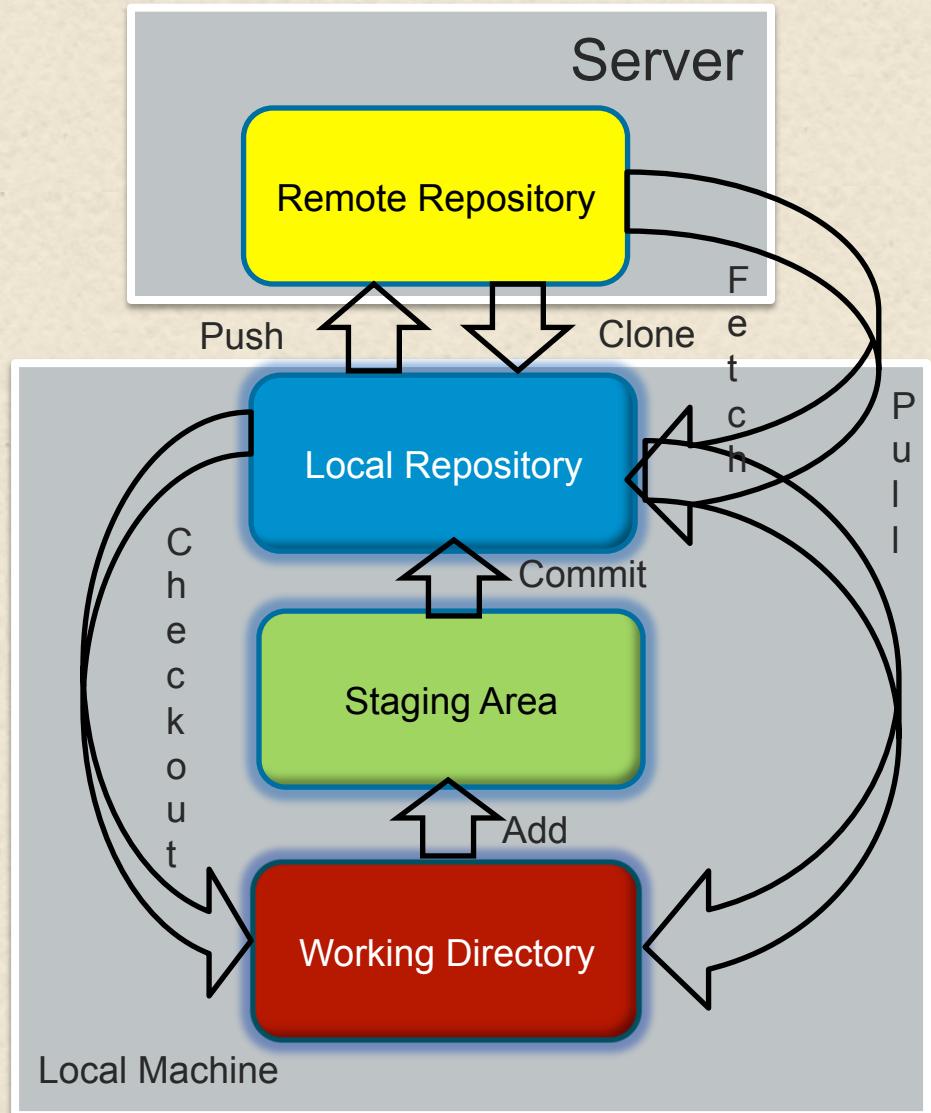
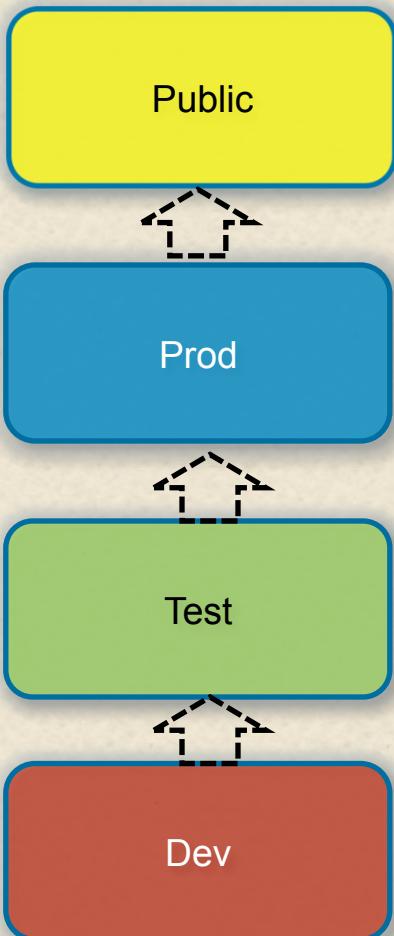
Staging Area



Practice with Merging

Purpose: In this lab, we'll work through some simple branch merging.

Git in One Picture



Cloning a Repo

81

- **Command: `git clone`**
 - Use to get a copy of an existing repository from a system
 - Syntax: `git clone <url> <optional new name>`
 - Clone vs. checkout
 - » Git pulls nearly all data from cloned area – files, history, etc.
 - » Creates a directory with the project name
 - » Checks out working copy of latest version
 - » Does a bit more than fetch and pull – tracks local branch to remote branch
 - After cloning, have full access to files, histories, etc.

Git Remote References “remotes”

82

82

- The term “remote” can either refer to:
 - An actual remote repository
 - A reference to such a repository
- For local use, Git provides references/aliases/nicknames that map to a remote repository location
- The default one of these is “origin”
- The reference “origin” is automatically setup (mapped) for you when you clone
- Example: origin = <https://github.com/<path>>
- This mapped name is what you use in local commands to tell git you want to push/pull/fetch/clone to/from the mapped remote repository
- Example: git pull origin <branch>

82

Git Remote Repositories

- Remote repositories
 - Think “server-side”
 - Versions of projects hosted on network or internet
 - Push or pull from (as opposed to checkout/add/commit)
 - Generally have read-only or read-write access
 - Handle + url
- Public as opposed to private (local)
- Multiple protocols for data transfer
 - Local – shared folder, easy for collaboration, slow, not for widespread use
 - SSH – standard with authentication, no anonymous access
 - Git – fastest, but no authentication
 - Http - easy, but inefficient
 - Https – easy and authenticated (temporarily)

Working with Handles to Remotes

- Seeing what you have access to
 - Git remote (short handle by default, -v shows url)
 - “origin” – special name that git gives to remote you cloned from
 - Ssh urls (git@<blah> vs. git protocol git://) and https indicates ones you can push to
 - Git remote show (shortname) – shows extended info about remote
- Adding a remote
 - Think of adding access to existing one
 - Git remote add [shortname (handle)] [url]
 - » Example : git remote add remote2 https://github.com/userid/projectpath
- Renaming shortname
 - Git remote rename <old> <new>
 - Renames remote branches too
- Removing a reference
 - Git remote rm (shortname)

Working with Remotes

- Retrieve latest from git remote (getting what you don't have)
 - Git fetch (shortname) as in git fetch brent
 - Note that you are getting the full repository (history, branches, basically everything)
 - Fetch pulls the data to the local repository – it doesn't merge or disturb your local content
- Retrieve latest and merge
 - Git pull
 - By default, git clone automatically sets up to track changes on master branch
 - Pull does a “fetch and merge”
- Pushing back to the server (remote)
 - Git push (shortname) (branch)
 - Only works if you have correct access and remote content hasn't been change
 - If remote content has been changed, then would have to fetch latest and merge (like cvs up-to-date check)

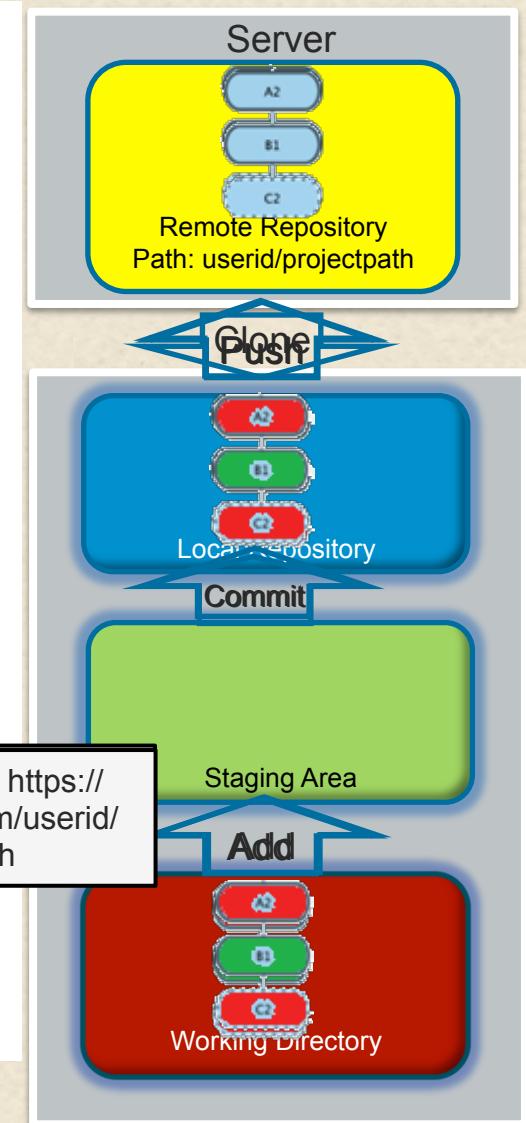
Git Remotes Example

```
$ git clone https://github.com/userid/projectpath
$ git remote -v
origin https://github.com/userid/projectpath(fetch)
origin https://github.com/userid/projectpath (pull)
```

```
$ <edit File A and File C>
$ git commit –am “...”
$ git push origin master
(default is origin and master so could just “git push”)
```

```
$ git remote rm origin
$ git remote add project1 https://github.com/userid/projectpath
$ git remote -v
project1 https://github.com/userid/projectpath(fetch)
project1 https://github.com/userid/projectpath (pull)
```

```
$ <edit File B>
$ git commit –am “...”
$ git push project1
(or git push project1 master)
```



Working with a Remote and Multiple Users

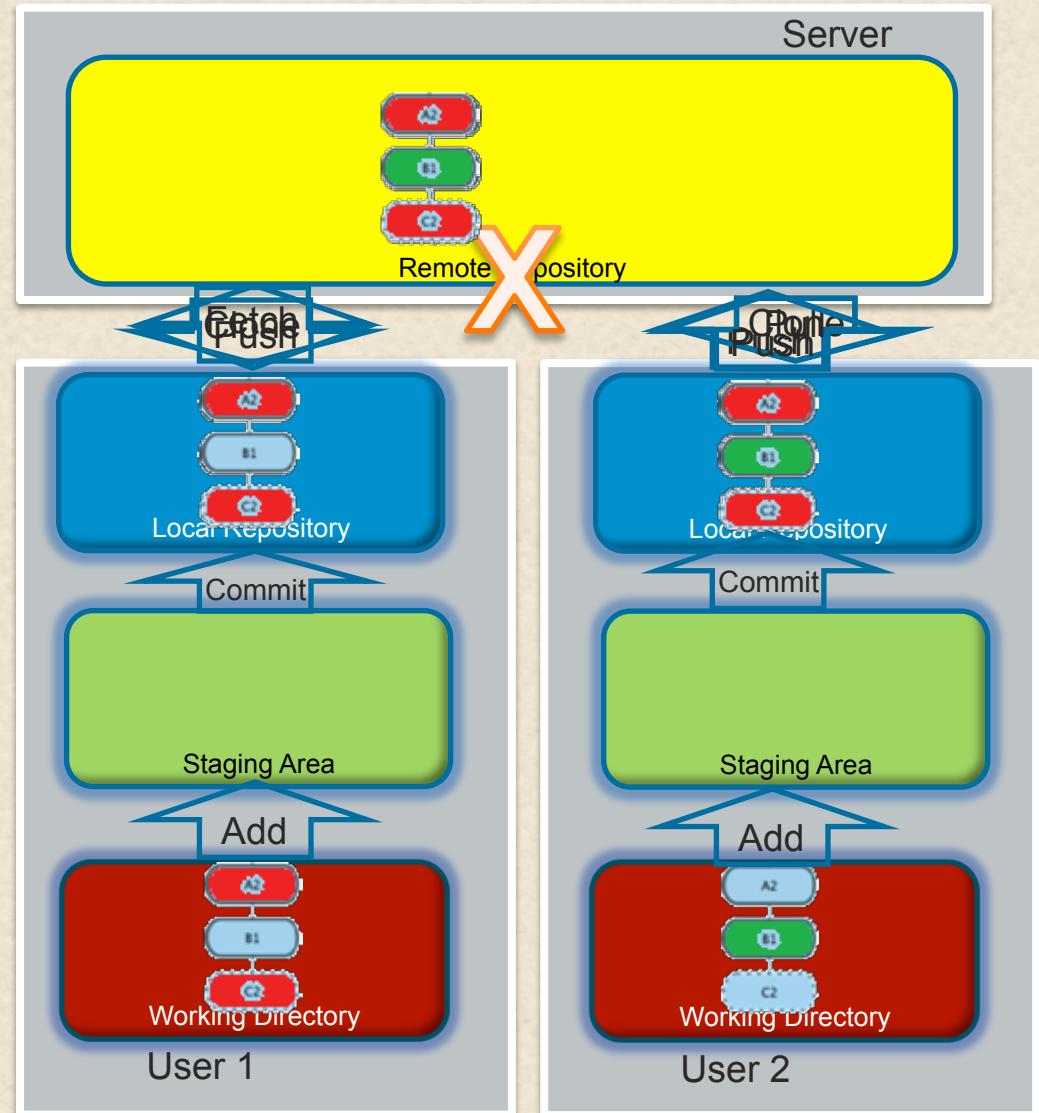
87

User 1

```
$ git clone ...
$ <edit File A and File C>
$ git commit -am "..."
$ git push
$ git fetch
```

User 2

```
$ git clone ...
$ <edit File B>
$ git commit -am "..."
$ git push
$ git pull
$ git push
```



87

Using the Overall Workflow with a Remote Repository

Purpose: In this lab, you'll get some practice with remotes by working with a Github account, forking a repository, cloning it down to your system to work with, rebasing changes, and dealing with conflicts at push time.

That's all - thanks!

Professional Git 1st Edition

by Brent Laster (Author)

4.5 stars 7 customer reviews

[Look inside](#) ↓

