A distributed version control system

Presented By

Hager Samir

Course Outline

- **Lesson 1:** Case & Problem

- **Lesson 2:** Solution & Types of Version Control System

- **Lesson 3:** git Features

- **Lesson 4:** git Support by commands

- **Lesson 5:** Remote & Branches

- ***** References & Recommended Reading**

Lesson 1
Case & Problem

Your Case & Problem

- If you want to change some file you have two choices:

    1. Write in this file and override the old file.

        - We will get you stuck with the latest changes.

    2. Define new version for this file (perhaps a time-stamped directory, if they're clever).

        - Is also incredibly error prone. It is easy to forget which directory you're in and

          accidentally write to the wrong file or copy over files you don't mean to.

Lesson 2
Solution is Version Control System

Version control systems

- Version control (or revision control, or source control) is a system that records changes to a file or

  set of files over time so that you can recall specific versions later.

  - Almost all "real" projects use some kind of version control

  - Essential for team projects, but also very useful for individual projects

- The Idea is to have Local Version Control System

Local Version Control Systems

- had a simple database that kept all the
  changes to files under revision control
- One of the more popular VCS tools was a
  system called RCS
- Supported by default in most of
  development Environment or Editors (IDEs)

Centralized Version Control Systems

- need to collaborate with developers on other systems.

- clients just check out the latest snapshot of the files

- such as CVS, Subversion, and Perforce

- a single server that contains all the versioned files, and a number of clients that check out files from that central place
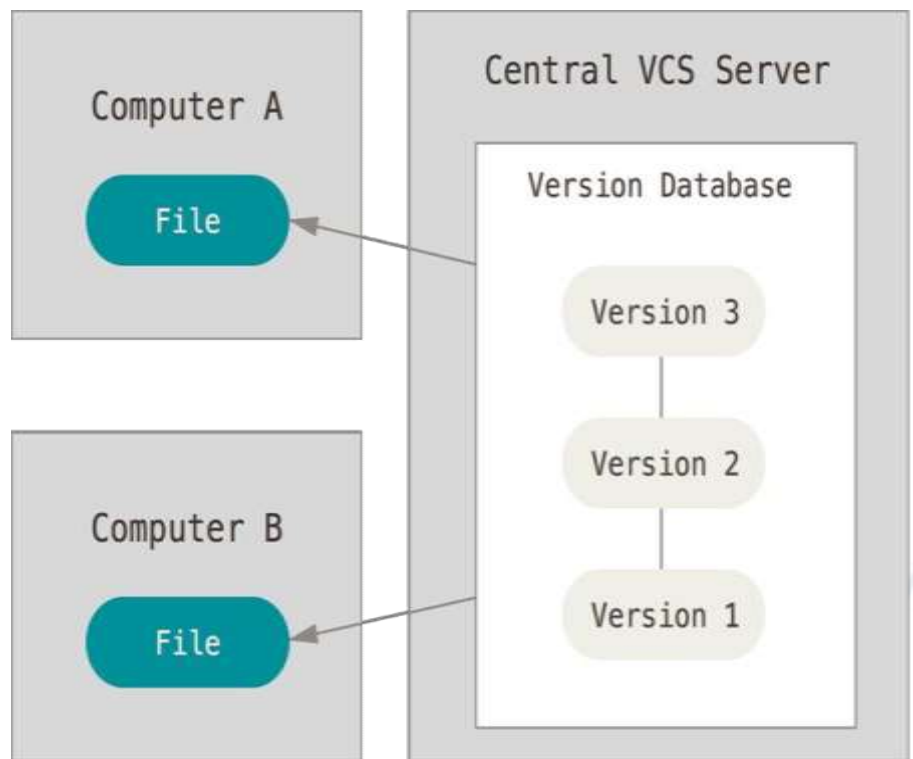
Centralized Version Control Systems (Pros.)

- File's versions shared with all developers.

- Everyone knows to a certain degree what everyone else on the project is doing

- Administrators have fine-grained control over who can do what

- Easier to administer than local databases on every client.

Centralized Version Control Systems (Cons.)

- single point of failure that the centralized server represents.

  - If that server goes down for an hour, then during that hour nobody can collaborate at all or

    save versioned changes to anything they're working on

  - If the hard disk the central database is on becomes corrupted, and proper backups haven't

    been kept, you lose absolutely everything

- Local VCS systems suffer from this same problem

Distributed Version Control Systems

- clients don't just check out the latest snapshot

  of the files: they fully mirror the repository

- such as Git, Mercurial, Bazaar or Darcs

- if any server dies, and these systems were

  collaborating via it, any of the client

  repositories can be copied back up to the

  server to restore it.
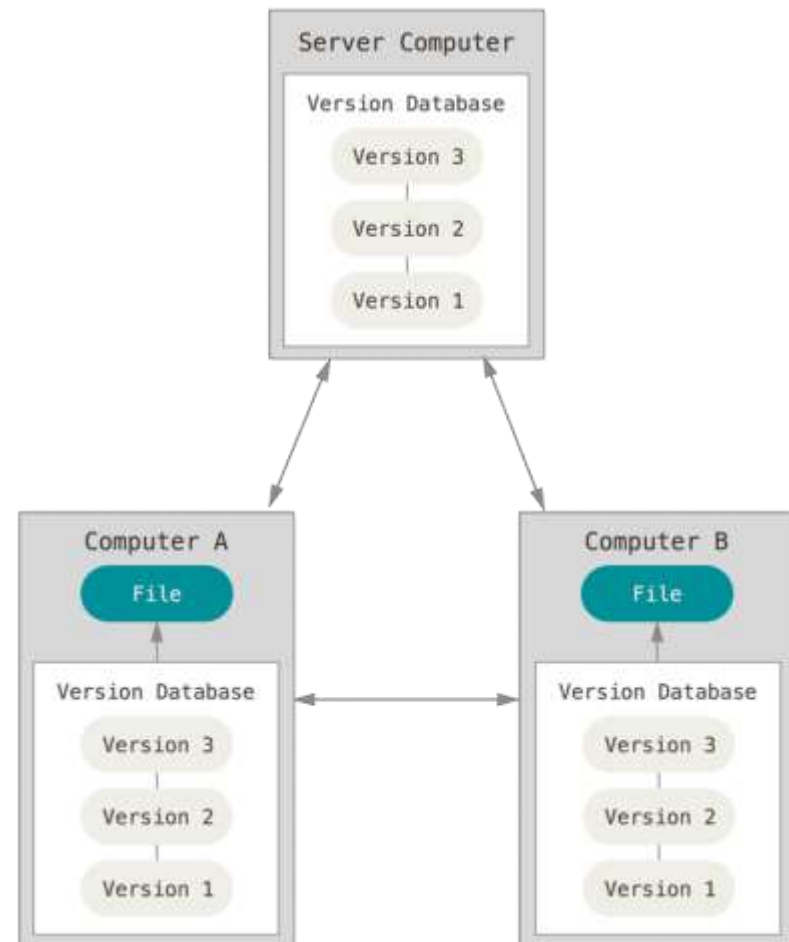
Lesson 3
git Features

Git Features

- Snapshots, Not Differences

- Nearly Every Operation Is Local

- Git Has Integrity

- Git Generally Only Adds Data

- The Three States

Snapshots, Not Differences

- Any other VCS store information as a list of file-based changes.
- These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

Checkins Over Time →

| Version 1 | Version 2 | Version 3 | Version 4 | Version 5 |

File A	→	Δ1	→	Δ2	
File B			→	Δ1	→	Δ2
File C	→	Δ1	→	Δ2	→	Δ3
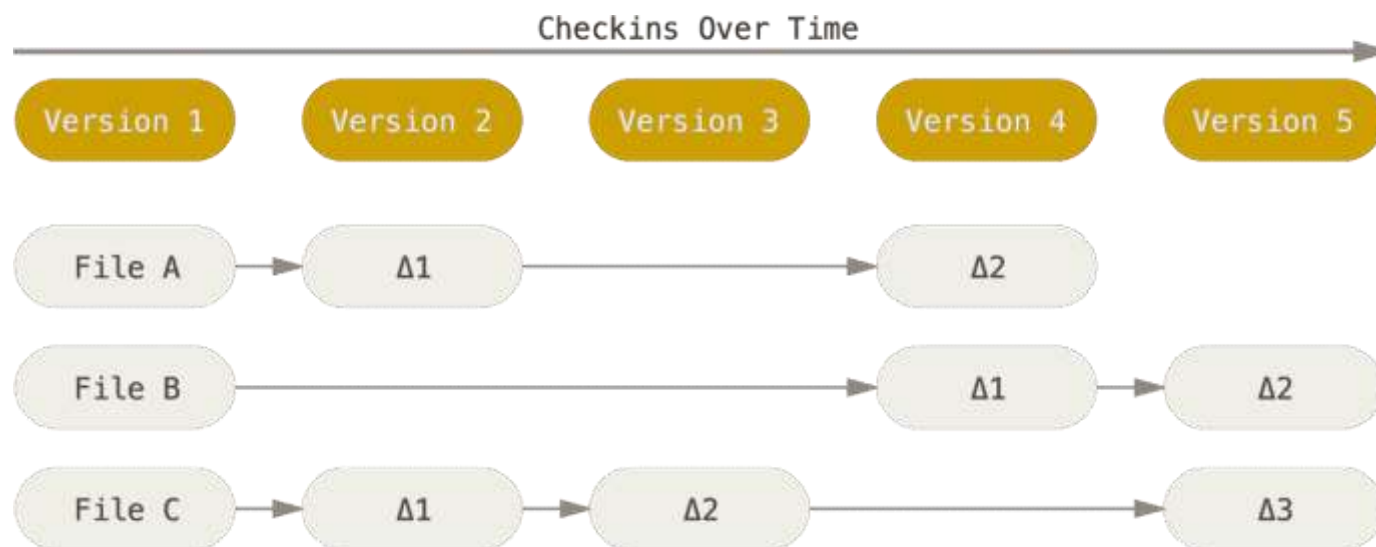
Snapshots, Not Differences (Ex.)

- Git thinks of its data more like a set of snapshots of a miniature file system (**stream of snapshots**).

- Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

Checkins Over Time

Version 1	Version 2	Version 3	Version 4	Version 5
File A	A1	A1	A2	A2
File B	B	B	B1	B2
File C	C1	C2	C2	C3
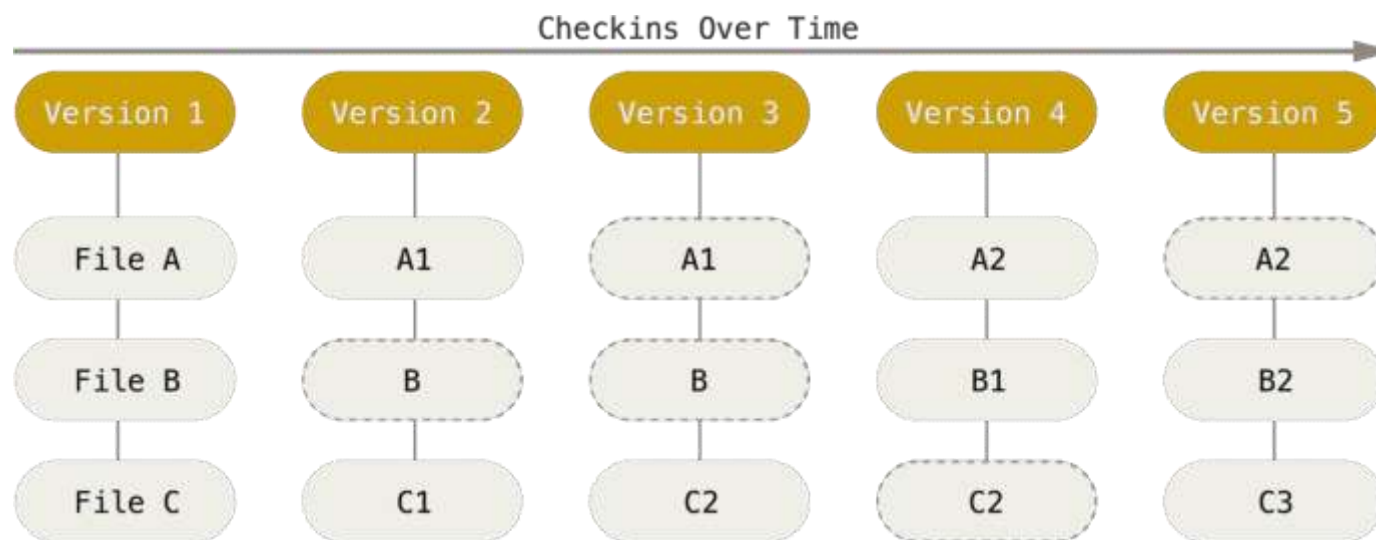
Snapshots, Not Differences (Ex.)

- To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.

Checkins Over Time

Version 1	Version 2	Version 3	Version 4	Version 5
File A	A1	A1	A2	A2
File B	B	B	B1	B2
File C	C1	C2	C2	C3
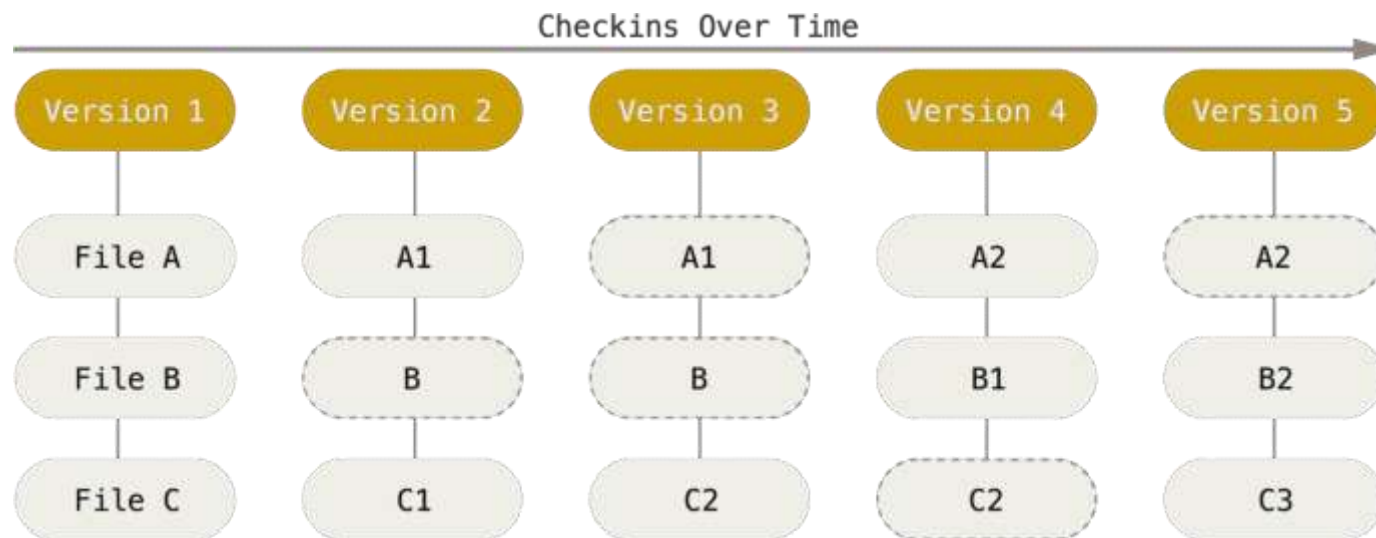
Nearly Every Operation Is Local

- If you're used to a CVCS where most operations have that network latencyoverhead.

- If you want to differentiate between current version & version from month ago, we have to ask a

  remote server to do it or pull this older version of the file from the remote server to do itlocally.

- in Subversion and CVS, you can edit files, but you can't commit changes to your database

  (because your database is offline).

Nearly Every Operation Is Local (Ex.)

 Java™ Education and Technology Services

- This aspect of Git make it Incredibly fast

- You have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

All of these times are in seconds.

Operation		Git	SVN	
Commit Files (A)	Add, commit and push 113 modified files (2164+, 2259-)	0.64	2.60	4x
Commit Images (B)	Add, commit and push 1000 1k images	1.53	24.70	16x
Diff Current	Diff 187 changed files (1664+, 4859-) against last commit	0.25	1.09	4x
Diff Recent	Diff against 4 commits back (269 changed/3609+,6898-)	0.25	3.99	16x
Diff Tags	Diff two tags against each other (v1.9.1.0/v1.9.3.0)	1.17	83.57	71x
Log (50)	Log of the last 50 commits (19k of output)	0.01	0.38	31x
Log (All)	Log of all commits (26,056 commits - 9.4M of output)	0.52	169.20	325x
Log (File)	Log of the history of a single file (array.c - 483 revs)	0.60	82.84	138x
Update	Pull of Commit A scenario (113 files changed, 2164+, 2259-)	0.90	2.82	3x
Blame	Line annotation of a single file (array.c)	1.91	3.04	1x

Git Has Integrity

- If you're using a CVCS the information about versions saved in configuration file which couldbe

  get corrupted

- Everything in Git is check-summed before it is stored and is then referred to by that checksum.

- This means it's impossible to change the contents of any file or directory without Git knowing

  about it.

- Checksum is built into Git at the lowest levels and is integral to its philosophy.

Git Has Integrity (Ex.)

- The mechanism that Git uses for this check summing is called a SHA-1 hash.

- This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

- A SHA-1 hash looks something like this:

  - 24b9da6552252987aa493b52f8696cd6d3b00373

- Git stores everything in its database not by file name but by the hash value of its contents

Git Generally Only Adds Data

- If you're using a CVCS, It's Easy to erase data in an accidental way (ex. Rollback).

- As in any VCS, you can lose or mess up changes you haven't committed yet.

- When you do actions in Git, nearly all of them only add data to the Git database (locally until commit).

- It is hard to get Git to do anything that is not undoable or to make it erase data in any way.

- Using Git become a joy because we know we can experiment without the danger of severely screwing things up.

The Three States
(modified, staged, committed)
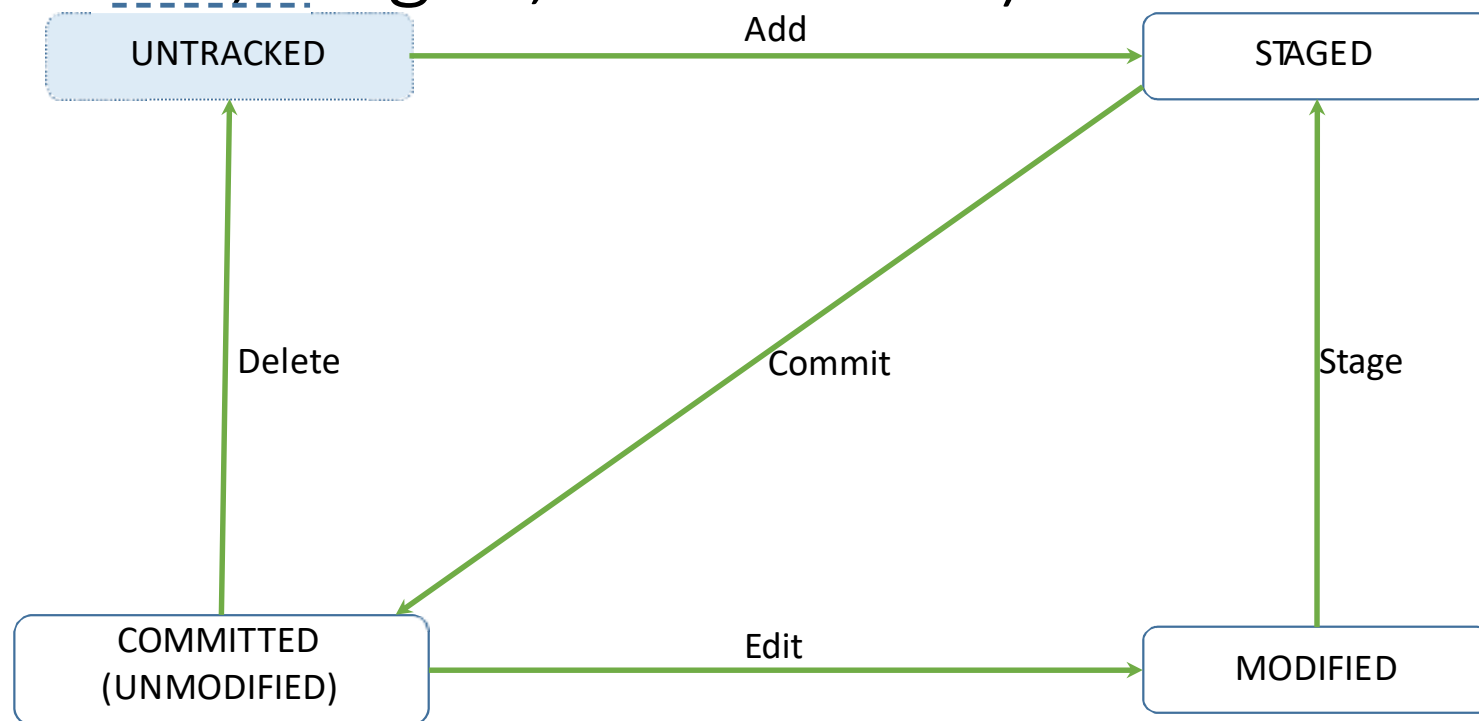
The Three States
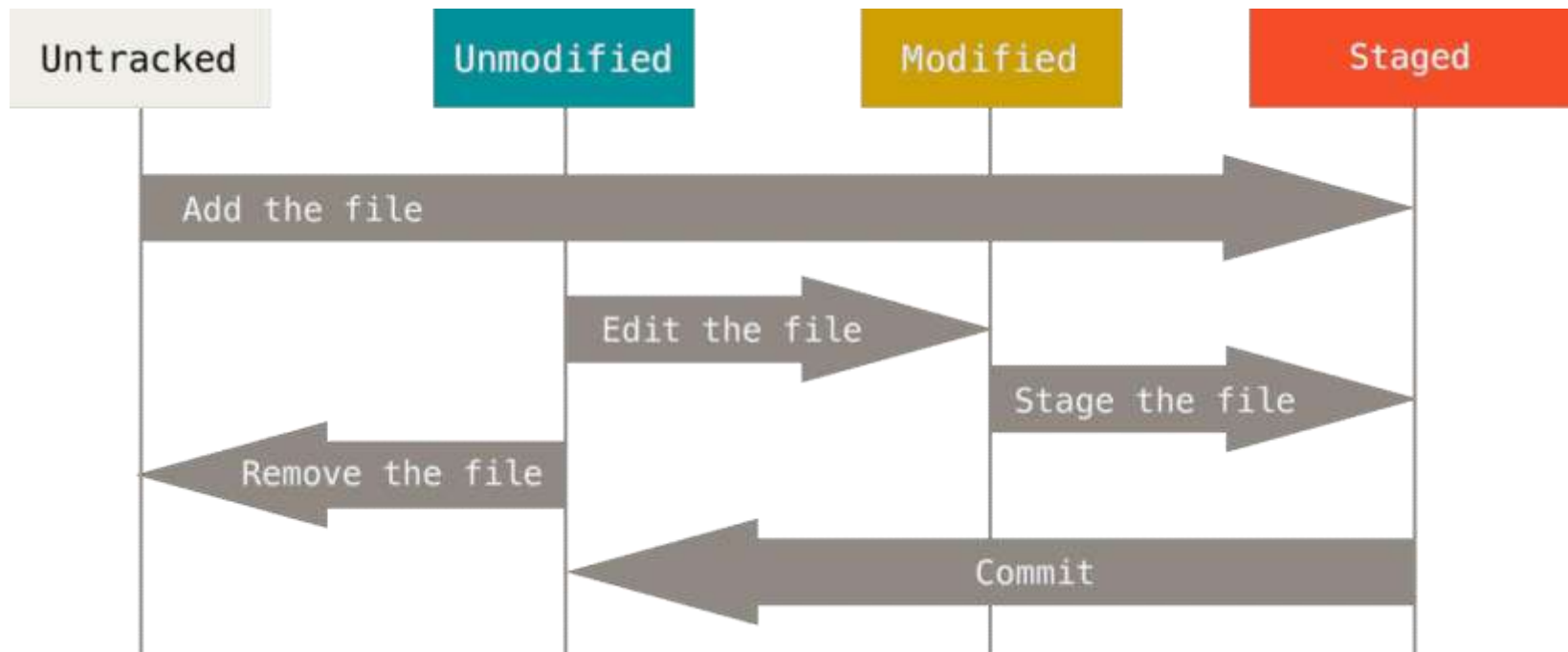
The Three States (Ex.)
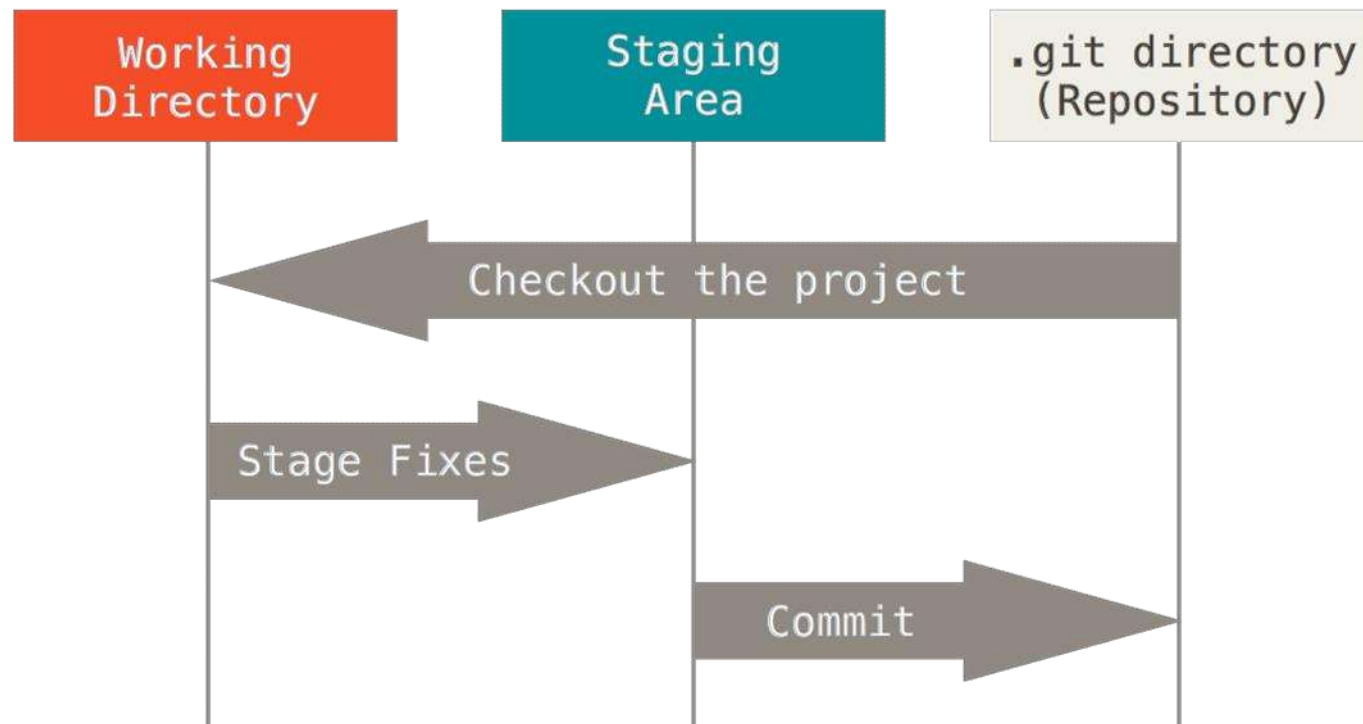
The Three States (Details)

http://blog.tzolov.net/2014/07/file-lifecycle-under-git-control.html
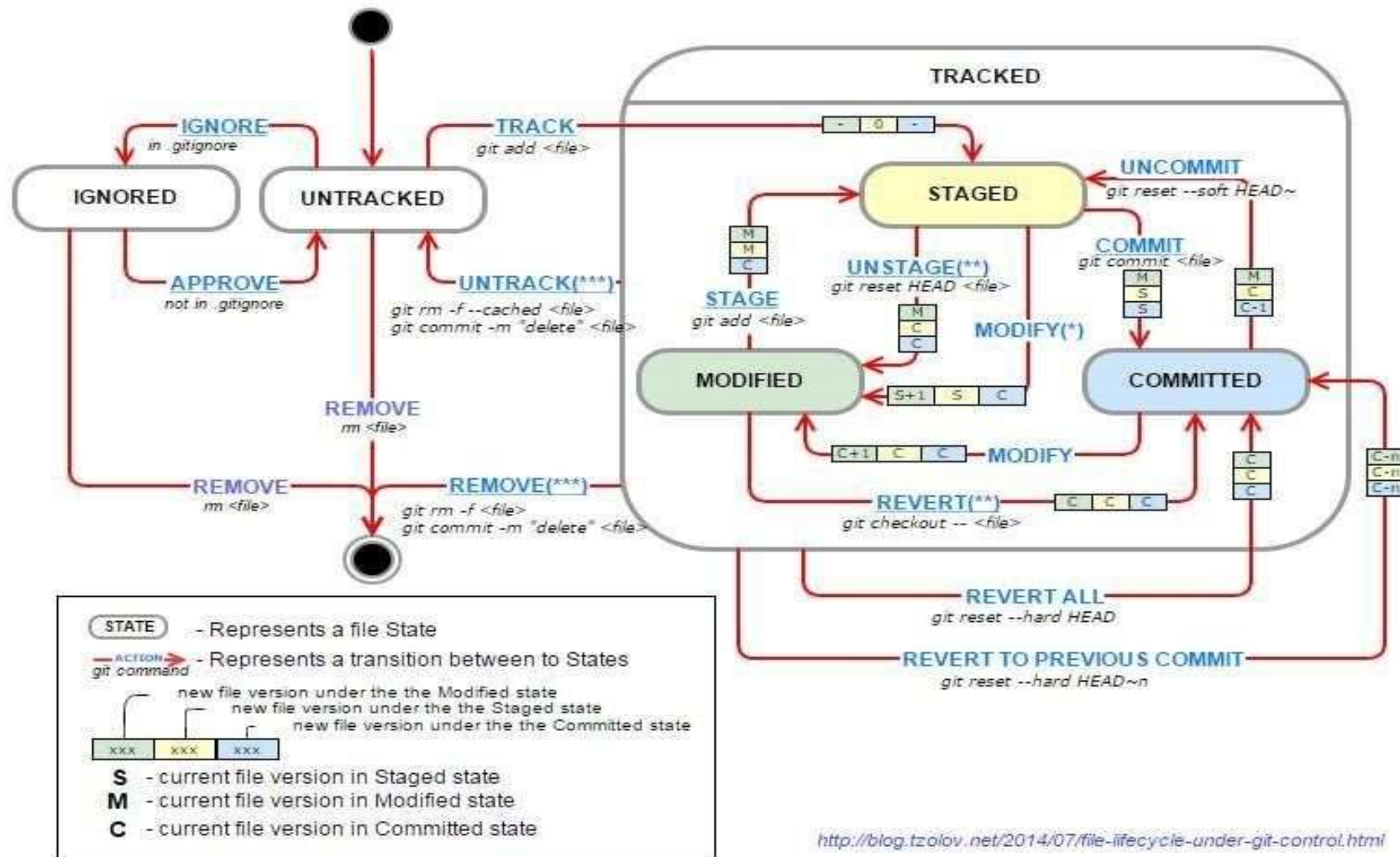
Lesson 4
git Support by commands

Set up - Getting Help

- If you ever need help while using Git, there are two ways:

    - By Git documentation on the following directory : "Git_Directory/doc/git/html/index.html"

    - Using Git Commands:

        - $ git help <verb>

        - $ git <verb> --help

Set up - Your Identity

- Config

  - Get and set repository or global options.

  - You can query/set/replace/unset options with this command.

  - The name is actually the section and the key separated by a dot, and the value will be escaped.

  - EX. Set User Name:

    - $ git   config   --global   user.name   "JEDiver"

    - $ git   config   --global   user.email   eng.medhat.cs.h@gmail.com

- global option

  - Then Git will always use that information for anything you do on that system

Set up - Checking Your Settings

- If you want to check all your settings, you can use:

  - **$ git** config --list

- If you want to check all global settings, you can use:

  - **$ git** config --global --list

- If you want to check a specific key's value, you can use:

  - **$ git** config user.name

Git Basics - Getting a Git Repository

- To make Git Repository we has two approaches:

  - Initializing a Repository in an Existing Directory

    1. Go to the project's directory

    2. Type $ git   init

       - creates a new subdirectory named .git that contains all of your necessary repository files.

       - At this point, nothing in your project is tracked yet.

    3. Tracking files (add files)

       - Type $ git   add   file_name

    4. Do an initial commit

       - Type $ git   commit   -m   'initial project version'

Git Basics - Getting a Git Repository (Ex.)

- To make Git Repository we has two approaches:

  - Cloning an Existing Repository

    1. Go to your directory

    2. You clone a repository with $ git clone [url]

       - creates a new directory named with repo name with subdirectory named .git that contains all of your necessary repository files, pulls down all the data for that repository, and checks out a working copy of the latest version.

       - If you want to change the directory then add new name after URL.

       - You'll notice that the command is "clone" and not "checkout".

       - This is an important distinction – instead of getting just a working copy, Git receives a full copy of nearly all data that the server has.

       - Every version of every file for the history of the project is pulled down

Recording Changes to the Repository

- Checking the Status of Your Files:
    - Detailed status:
        - Type   **$ git**   status
    - Short status:
        - Type   **$ git**   status -s
- Viewing Your Staged and Unstaged Changes:
    - you want to know exactly what you changed, not just which files were changed
    - You've changed but not yet staged:
        - Type   **$ git**   diff
    - You've staged :
        - Type   **$ git**   diff   --staged
        - Type   **$ git**   diff   --cached

Recording Changes to the Repository (Ex.)

- Tracking New Files / Staging Modified Files:
  - Detailed status:
    - Type $ git   add   File1_name   File2_name
- Removing Files
  - If you simply remove the file from your working directory, it shows up under the "Changed but not updated" (that is unstaged area)
  - To stages the file's removal:
    - Type $ git   rm   File1_name   File2_name
  - To force the removal:
    - Type $ git   rm   -f   File1_name   File2_name
  - To stages the file's removal but preserves it locally:
    - Type $ git   rm   --cached   File1_name   File2_name

Recording Changes to the Repository (Ex.)

- Moving Files
  - Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file
  - We use mv command to rename file.
  - To rename file:
    - Type $ git   mv   file_from   file_to
  - To rename file (Equivalent way):
    - Type
    - $ mv   file_from   file_to
    - $ git   rm   file_from
    - $ git   add   file_to

Recording Changes to the Repository (Ex.)

- Committing Your Changes

  - General form (commit all staged files):

    - Type $ git   commit

- To type commit message:

    - Type $ git   commit   -m   "Commit Message"

  - The staging area is sometimes a bit more complex than you need in your workflow

  - To commit all files (staged & unstaged):

    - Type $ git   commit   -a   -m   "Commit Message"

Ignoring Files

- Some files that you don't want Git to automatically add or even show you as being untracked.

- Automatically generated files such as log files or files produced by your buildsystem.

- Through Patterns to match them in file named .gitignore

- Setting up a .gitignore file before you get going is generally a good idea so you don't accidentally

  commit files that you really don't want in your Git repository

- List all Ignored files in this project

  - Type $ git   ls-files   --other   --ignored   --exclude-standard

Ignoring Files (Ex.) Rules

- Blank lines or lines starting with # are ignored.

- You can end patterns with a forward slash (/) to specify a directory.

- You can negate a pattern by starting it with an exclamation point (!)

- Standard glob patterns work (Regex Applied).

  - An asterisk (*) matches zero or more characters;

  - [abc] matches any character inside the brackets (in this case a, b, or c);

  - a question mark (?) matches a single character;

  - brackets enclosing characters separated by a hyphen([0-9]) matches any character between them (in this case 0 through 9)

  - You can also use two asterisks to match nested directories; a/**/z would match a/z, a/b/z, a/b/c/z, and so on

Ignoring Files (Ex.) Example

no .a files

*.a

but do track lib.a, even though you're ignoring .a files above

!lib.a

only ignore the root TODO file, not subdir/TODO

/TODO

ignore all files in the build/ directory

build/

ignore doc/notes.txt, but not doc/server/arch.txt

doc/*.txt

ignore all .txt files in the doc/ directory

doc/**/*.txt

Viewing the Commit History

- lists the commits made in that repository in reverse chronological order

  - General form:

    - Type $ git   log

  - Shows the difference introduced in each commit:

    - Type $ git   log   -p

  - Show summary for each commit:

    - Type $ git   log   --stat

  - changes the log output to formats other than the default (oneline/short/full/fuller   Or

    Regex by format):

    - Type $ git   log   --pretty=oneline

Viewing the Commit History (Ex.)

- Lists version history for specific:
  - File:
    - Type $ git   log   --follow   file_name
  - Commit:
    - Type $ git   show   commit_key
- Changes the log output to formats other than the default:
  - By Keywords (oneline/short/full/fuller):
    - Type $ git   log   --pretty=oneline
  - By Regex using format:
    - Type $ git   log   --pretty=format:"%h - %an,   %ar : %s"

Viewing the Commit History (Ex.)

Option	Description of Output	Option	Description of Output
%H	Commit hash	%h	Abbreviated commit hash
%T	Tree hash	%t	Abbreviated tree hash
%P	Parent hashes	%p	Abbreviated parent hashes
%an	Author name	%ae	Author e-mail
%ad	Author date (format respects the –date= option)	%ar	Author date, relative
%cn	Committer name	%ce	Committer email
%cd	Committer date	%cr	Committer date, relative
%s	Subject		

Viewing the Commit History (Ex.)

Common options to log

Option	Description
-p	Show the patch introduced with each commit.
--stat	Show statistics for files modified in each commit.
--shortstat	Display only the changed/insertions/deletions line from the --stat command.
--name-only	Show the list of files modified after the commit information.
--name-status	Show the list of files affected with added/modified/deleted information as well.
--relative-date	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
--graph	Display an ASCII graph of the branch and merge history beside the log output.
--pretty	Show commits in an alternate format.

Viewing the Commit History (Ex.)

- Let you show only a subset of commits

- Limiting Log output:

    - Limits the output to number of entries :

        - Type $ git   log   -3

    - Gets the list of commits made in the last two weeks:

        - Type $ git   log   --since=2.weeks

- You can mix with one or more option for commit limiting.

Viewing the Commit History (Ex.)

Option	Description
-(n)	Show only the last n commits
--since, --after	Limit the commits to those made after the specified date.
--until, --before	Limit the commits to those made before the specified date.
--author	Only show commits in which the author entry matches the specified string.
--committer	Only show commits in which the committer entry matches the specified string.
--grep	Only show commits with a commit message containing the string
-S	Only show commits adding or removing code matching the string

Undoing Things (undo)

- Be careful, because you can't always undo some of these undos

- When you commit too early and possibly forget to add some files, or you mess up your commit message

  - Type in second commit $ git   commit   --amend

  - Example

    - Type $ git   commit   -m   "Initial commit"

    - Type $ git   add   forgotten_file

    - Type $ git   commit   --amend

Undoing Things (undo) (Ex.)

- Reset commit
  - Undoes All commits after this commit, preserve changes locally
    - Type **$ git** reset   commit_key
  - Discard all history and changes back to this commit.
    - Type **$ git** reset --hard   commit_key
- Unstaging a Staged File:
  - The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them
    - When typing **$ git**   status
  - Type **$ git** reset   HEAD   file_name

Undoing Things (undo) (Ex.)

- Unmodifying a Modified File:

    - Modified File / renamed File / moved file / removed file

    - Type $ git   checkout   file_name

    - It's a dangerous command, any changes you made to that file are gone.

LAP 1

- Start new Repo:

  - Create 2 classes and one text file and commit them

    - HelloWorld (method: printHelloWorld)

    - HelloITI (method: printHelloITI)

    - Test.txt

  - Confirm that every thing is commited then

    - Rename HelloWorld class to HelloJava

    - Add printHelloGit to HelloITI class

  - Create .gitignore file and ignore all text files then create new commit

  - Do at least 10 Commits

  - Always use git status and git log to confirm your work