Git is a distributed version control and source code management system.

It does this through a series of snapshots of your project, and it works with those snapshots to provide you with functionality to version and manage your source code.

Versioning Concepts

What is version control?

Version control is a system that records changes to a file(s), over time.

Centralized Versioning VS Distributed Versioning

- Centralized version control focuses on synchronizing, tracking, and backing up files.
- Distributed version control focuses on sharing changes. Every change has a unique id.
- Distributed systems have no defined structure. You could easily have a SVN style, centralized system, with git.

Additional Information

Why Use Git?

- Can work offline.
- Collaborating with others is easy!
- Branching is easy!
- Merging is easy!
- Git is fast.
- Git is flexible.

Git Architecture

Repository

A set of files, directories, historical records, commits, and heads. Imagine it as a source code data structure, with the attribute that each source code "element" gives you access to its revision history, among other things.

A git repository is comprised of the .git directory & working tree.

.git Directory (component of repository)

The .git directory contains all the configurations, logs, branches, HEAD, and more. Detailed List.

Working Tree (component of repository)

This is basically the directories and files in your repository. It is often referred to as your working directory.

Index (component of .git dir)

The Index is the staging area in git. It's basically a layer that separates your working tree from the Git repository. This gives developers more power over what gets sent to the Git repository.

Commit

A git commit is a snapshot of a set of changes, or manipulations to your Working Tree. For example, if you added 5 files, and removed 2 others, these changes will be contained in a commit (or snapshot). This commit can then be pushed to other repositories, or not!

Branch

A branch is essentially a pointer to the last commit you made. As you go on committing, this pointer will automatically update to point the latest commit.

Tag

A tag is a mark on specific point in history. Typically people use this functionality to mark release points (v1.0, and so on)

HEAD and head (component of .git dir)

HEAD is a pointer that points to the current branch. A repository only has 1 *active* HEAD. head is a pointer that points to any commit. A repository can have any number of heads.

Stages of Git

- Modified - Changes have been made to a file but file has not been committed to Git Database yet
- Staged - Marks a modified file to go into your next commit snapshot
- Committed - Files have been committed to the Git Database

Conceptual Resources

- Git For Computer Scientists
- Git For Designers

Commands

init

Create an empty Git repository. The Git repository's settings, stored information, and more is stored in a directory (a folder) named ".git".

```
$ git init
```

config

To configure settings. Whether it be for the repository, the system itself, or global configurations (global config file is `~/.gitconfig`).

```
# Print & Set Some Basic Config Variables (Global)
$ git config --global user.email "MyEmail@Zoho.com"
$ git config --global user.name "My Name"
```

Learn More About git config.

help

To give you quick access to an extremely detailed guide of each command. Or to just give you a quick reminder of some semantics.

```
# Quickly check available commands
$ git help

# Check all available commands
$ git help -a

# Command specific help - user manual
# git help <command_here>
$ git help add
$ git help commit
$ git help init
# or git <command_here> --help
$ git add --help
$ git commit --help
$ git init --help
```

ignore files

To intentionally untrack file(s) & folder(s) from git. Typically meant for private & temp files which would otherwise be shared in the repository.

```
$ echo "temp/" >> .gitignore
$ echo "private_key" >> .gitignore
```

status

To show differences between the index file (basically your working copy/repo) and the current HEAD commit.

```
# Will display the branch, untracked files, changes and other differences
$ git status

# To learn other "tid bits" about git status
$ git help status
```

add

To add files to the staging area/index. If you do not `git add` new files to the staging area/index, they will not be included in commits!

```
# add a file in your current working directory
$ git add HelloWorld.java

# add a file in a nested dir
$ git add /path/to/file/HelloWorld.c

# Regular Expression support!
$ git add ./*.java
```

This only adds a file to the staging area/index, it doesn't commit it to the working directory/repo.

branch

Manage your branches. You can view, edit, create, delete branches using this command.

```
# list existing branches & remotes
$ git branch -a

# create a new branch
$ git branch myNewBranch

# delete a branch
$ git branch -d myBranch

# rename a branch
# git branch -m <oldname> <newname>
$ git branch -m myBranchName myNewBranchName

# edit a branch's description
$ git branch myBranchName --edit-description
```

tag

Manage your tags

```
# List tags
$ git tag
# Create a annotated tag
# The -m specifies a tagging message,which is stored with the tag.
# If you don't specify a message for an annotated tag,
# Git launches your editor so you can type it in.
$ git tag -a v2.0 -m 'my version 2.0'
# Show info about tag
# That shows the tagger information, the date the commit was tagged,
# and the annotation message before showing the commit information.
$ git show v2.0
# Push a single tag to remote
$ git push origin v2.0
# Push a lot of tags to remote
$ git push origin --tags
```

checkout

Updates all files in the working tree to match the version in the index, or specified tree.

```
# Checkout a repo - defaults to master branch
$ git checkout
# Checkout a specified branch
$ git checkout branchName
# Create a new branch & switch to it
# equivalent to "git branch <name>; git checkout <name>"
$ git checkout -b newBranch
```

clone

Clones, or copies, an existing repository into a new directory. It also adds remote-tracking branches for each branch in the cloned repo, which allows you to push to a remote branch.

```
# Clone learnxinyminutes-docs
$ git clone https://github.com/adambard/learnxinyminutes-docs.git
# shallow clone - faster cloning that pulls only latest snapshot
$ git clone --depth 1 https://github.com/adambard/learnxinyminutes-docs.git
# clone only a specific branch
$ git clone -b master-cn https://github.com/adambard/learnxinyminutes-docs.git --single-branch
```

commit

Stores the current contents of the index in a new "commit." This commit contains the changes made and a message created by the user.

```
# commit with a message
$ git commit -m "Added multiplyNumbers() function to HelloWorld.c"

# automatically stage modified or deleted files, except new files, and then commit
$ git commit -a -m "Modified foo.php and removed bar.php"

# change last commit (this deletes previous commit with a fresh commit)
$ git commit --amend -m "Correct message"
```

diff

Shows differences between a file in the working directory, index and commits.

```
# Show difference between your working dir and the index
$ git diff

# Show differences between the index and the most recent commit.
$ git diff --cached

# Show differences between your working dir and the most recent commit
$ git diff HEAD
```

grep

Allows you to quickly search a repository.

Optional Configurations:

```
# Thanks to Travis Jeffery for these
# Set line numbers to be shown in grep search results
$ git config --global grep.lineNumber true

# Make search results more readable, including grouping
$ git config --global alias.g "grep --break --heading --line-number"
```

```
# Search for "variableName" in all java files
$ git grep 'variableName' -- '*.java'

# Search for a line that contains "arrayListName" and, "add" or "remove"
$ git grep -e 'arrayListName' --and \( -e add -e remove \)
```

Google is your friend; for more examples Git Grep Ninja

log

Display commits to the repository.

```
# Show all commits
$ git log

# Show only commit message & ref
$ git log --oneline

# Show merge commits only
$ git log --merges

# Show all commits represented by an ASCII graph
$ git log --graph
```

merge

"Merge" in changes from external commits into the current branch.

```
# Merge the specified branch into the current.
$ git merge branchName

# Always generate a merge commit when merging
$ git merge --no-ff branchName
```

mv

Rename or move a file

```
# Renaming a file
$ git mv HelloWorld.c HelloNewWorld.c

# Moving a file
$ git mv HelloWorld.c ./new/path/HelloWorld.c

# Force rename or move
# "existingFile" already exists in the directory, will be overwritten
$ git mv -f myFile existingFile
```

pull

Pulls from a repository and merges it with another branch.

```
# Update your local repo, by merging in new changes
# from the remote "origin" and "master" branch.
# git pull <remote> <branch>
$ git pull origin master

# By default, git pull will update your current branch
# by merging in new changes from its remote-tracking branch
$ git pull

# Merge in changes from remote branch and rebase
# branch commits onto your local repo, like: "git pull <remote> <branch>, git rebase <branch>"
$ git pull origin master --rebase
```

push

Push and merge changes from a branch to a remote & branch.

```
# Push and merge changes from a local repo to a
# remote named "origin" and "master" branch.
# git push <remote> <branch>
$ git push origin master

# By default, git push will push and merge changes from
# the current branch to its remote-tracking branch
$ git push

# To link up current local branch with a remote branch, add -u flag:
$ git push -u origin master
# Now, anytime you want to push from that same local branch, use shortcut:
$ git push
```

stash

Stashing takes the dirty state of your working directory and saves it on a stack of unfinished changes that you can reapply at any time.

Let's say you've been doing some work in your git repo, but you want to pull from the remote. Since you have dirty (uncommited) changes to some files, you are not able to run `git pull`. Instead, you can run `git stash` to save your changes onto a stack!

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
  HEAD is now at 049d078 added the index file
  (To restore them type "git stash apply")
```

Now you can pull!

```
git pull

...changes apply...
```

Now check that everything is OK

```
$ git status
# On branch master
nothing to commit, working directory clean
```

You can see what "hunks" you've stashed so far using `git stash list`. Since the "hunks" are stored in a Last-In-First-Out stack, our most recent change will be at top.

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

Now let's apply our dirty changes back by popping them off the stack.

```
$ git stash pop
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

`git stash apply` does the same thing

Now you're ready to get back to work on your stuff!

Additional Reading.

rebase (caution)

Take all changes that were committed on one branch, and replay them onto another branch. *Do not rebase commits that you have pushed to a public repo.*

```
# Rebase experimentBranch onto master
# git rebase <basebranch> <topicbranch>
$ git rebase master experimentBranch
```

Additional Reading.

reset (caution)

Reset the current HEAD to the specified state. This allows you to undo merges, pulls, commits, adds, and more. It's a great command but also dangerous if you don't know what you are doing.

```
# Reset the staging area, to match the latest commit (leaves dir unchanged)
$ git reset

# Reset the staging area, to match the latest commit, and overwrite working dir
$ git reset --hard

# Moves the current branch tip to the specified commit (leaves dir unchanged)
# all changes still exist in the directory.
$ git reset 31f2bb1

# Moves the current branch tip backward to the specified commit
# and makes the working dir match (deletes uncommited changes and all commits
# after the specified commit).
$ git reset --hard 31f2bb1
```

revert

Revert can be used to undo a commit. It should not be confused with reset which restores the state of a project to a previous point. Revert will add a new commit which is the inverse of the specified commit, thus reverting it.

```
# Revert a specified commit
$ git revert <commit>
```

rm

The opposite of git add, git rm removes files from the current working tree.

```
# remove HelloWorld.c
$ git rm HelloWorld.c

# Remove a file from a nested dir
$ git rm /pather/to/the/file/HelloWorld.c
```

Further Information

- tryGit - A fun interactive way to learn Git.
- Udemy Git Tutorial: A Comprehensive Guide
- Git Immersion - A Guided tour that walks through the fundamentals of git
- git-scm - Video Tutorials
- git-scm - Documentation
- Atlassian Git - Tutorials & Workflows
- SalesForce Cheat Sheet
- GitGuys
- Git - the simple guide
- Pro Git
- An introduction to Git and GitHub for Beginners (Tutorial)