tass
TSS TECHNOLOGY SOLUTIONS

git

Jonas Lejeune

September 17, 2013

Introduction

Introduction

Different Version Control Systems

- Local Version Control Systems
  - Keep local database with changes
  - Not usable for multiple developers
- Centralized Version Control Systems
  - Keep database with revisions on a central server
  - Developers check-out the revision they need
  - Easy to administer access
- Distributed Version Control Systems
  - Every client mirrors the complete repository
  - Thus every node is a backup!
  - Complex operations become much faster

Introduction

Centralized vs Distributed

Centralized

History kept remote
Only working copy available offline
Speed depends on server and connection
Stores data as changes to base files
Possible to lock (binary) files
Easy to learn

Distributed

History local and (optional) remote
Entire repository available
Incredibly fast *always*

Stores a snapshot of data everytime
No support for locking
Somewhat more difficult. . .

Introduction

Speed of Git vs SubVersion

Operation		Git	SVN	
Commit files	Add, commit and push 113 modified files (2164+, 2259-)	0.64	2.60	4x
Commit Images	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
Clone	Clone in Git vs checkout in SVN	107.5	14.0	8x
Size (M)	Size of total client side data and files (in M)	181.0	132.0	

All times are in seconds, SVN was used in ideal conditions (server under no load and 80MB/s line)

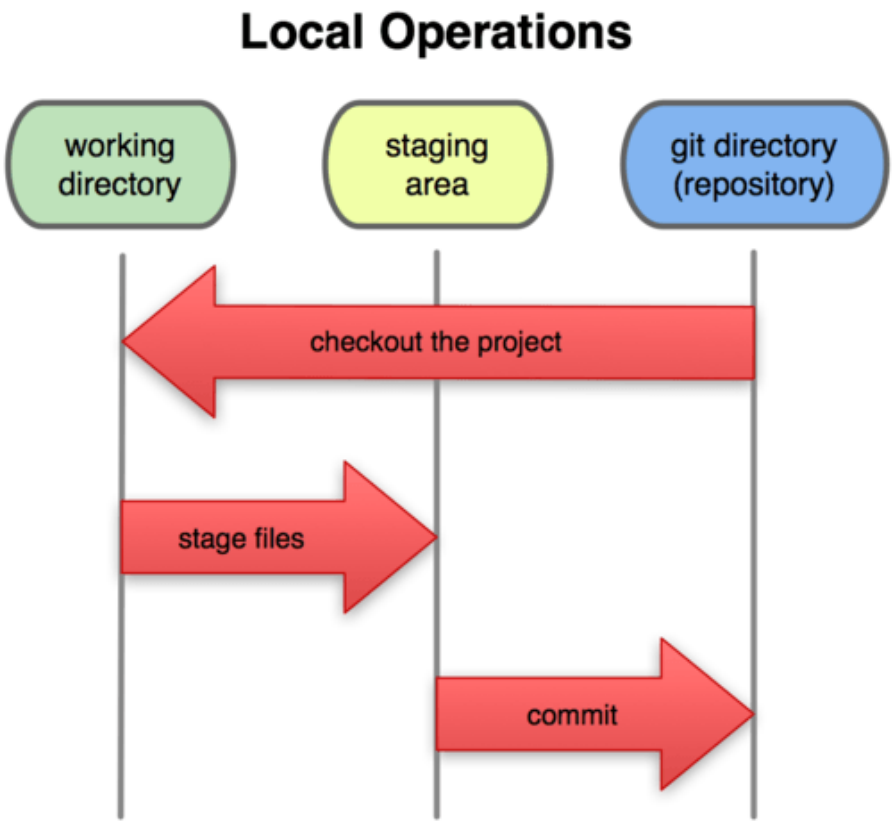
tass TECHNOLOGY SOLUTIONS

Git Basics

Background Information

Git has 3 main states files can be in:

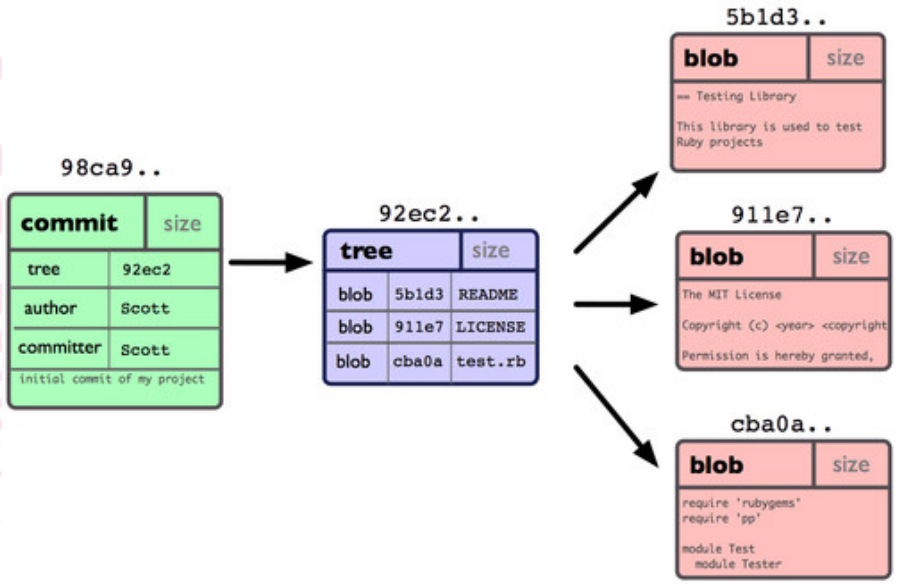
- **committed**: Safely stored in local database
- **modified**: File is changed and not committed
- **staged**: Changed file marked to go in next committed snapshot



Background Information

Git and the SHA1

- Git addresses all content using a SHA1 hash
- All Git objects have a SHA1:
  - When staging a file, Git stores the SHA1 in the staging area
  - When committing, Git stores a SHA1 of a commit object pointing to the directory tree
- In theory two commits can generate the same hash, but a higher probability exists that every member of your team will be attacked and killed by wolves in unrelated incidents on the same night.



Installation

- From source
- Linux binary:
  - `apt-get install git`
  - `yum install git-core`
  - `pacman -S git`
- OS X binary:
  - Download installer from http://code.google.com/p/git-osx-installer
  - Use MacPorts:
    ```
    sudo port install git-core +svn +doc +bash_completion
    +gitweb
    ```
- Windows binary:
  - Download installer from http://msysgit.github.com/

Installation

Initial Configuration

- `git config --global user.name "John Doe"`
- `git config --global user.email johndoe@example.com`
- `git config --global core.editor vim`
- `git config --global core.editor vim`
- Check your configuration with:
  `git config --list`
- Or edit your config-file directly:
  Global: `~/.gitconfig`
  Local per repo: *repo*`/.git/.gitconfig`

Installation

Initial Configuration

Aliases are useful to make a shorthand for often used commands:

- Add them to your `.gitconfig` file
  or use: `git config alias.`*`newalias command`*
- Shorthand: `st = status` allows you to type `git st` to see status
- Add a pretty log function under `git lg` :

```
[alias]
lg = log --graph --pretty=format:'%Cred%h%Creset
-%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold
blue)<%an>%Creset' --abbrev-commit --date=relative
```

Not mine, I also stole it from the wiki ;)

- Overwriting existing git commands is not possible

Installation

Set up GitHub

- Create an account on github.com

  Use the same e-mail address as you used for configuring Git

- Generate ssh-keys on your machine:
  `ssh-keygen -t rsa -C "johndoe@example.com"`

  Not required if you already have `~/.ssh/id_rsa.pub`

- Add the content of the `~/.ssh/id_rsa.pub` file to your SSH Keys on GitHub
  https://github.com/settings/ssh

- Test it! `ssh -T git@github.com`

Basic commands

Getting a repository

- `git init` *`reponame`*
  To create a new repository
- `git -bare init` *`reponame`*
  If you don't want a working tree (used for server)
- `git clone` *`url [dir]`*
  Clone an existing repository

Basic commands

.gitignore

- Each line represents a pattern to ignore
- `*.o` tells git to ignore object-files
- Files already tracked are not affected
- `!important.o` negates the ignoring of important.o
- Comments begin with a #
- Find many example .gitignore files on github.com/github/gitignore

Basic commands

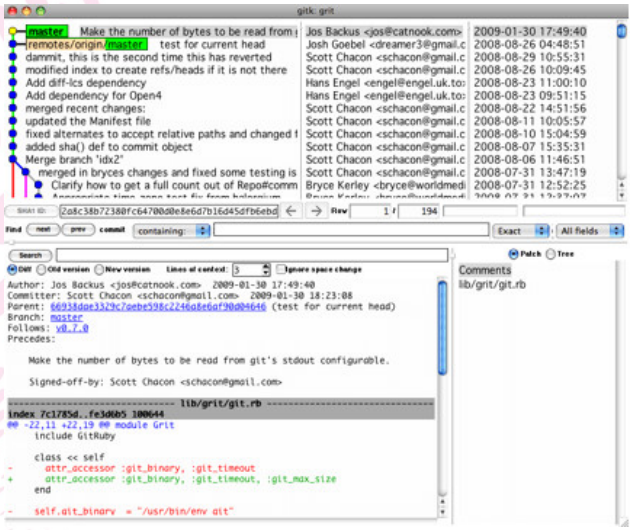
Recording Changes

- `git status`
  Check the status of your files
- `git add` *`filename`*
  Add a new file or stage a modified file
- `git diff`
  Check what you have changed but not staged
- `git diff --staged`
  See what changes are staged
- `git commit`
  Commit your staged changes
- `git commit -a`
  Automagically adds tracked files to commit

Basic commands

Viewing History

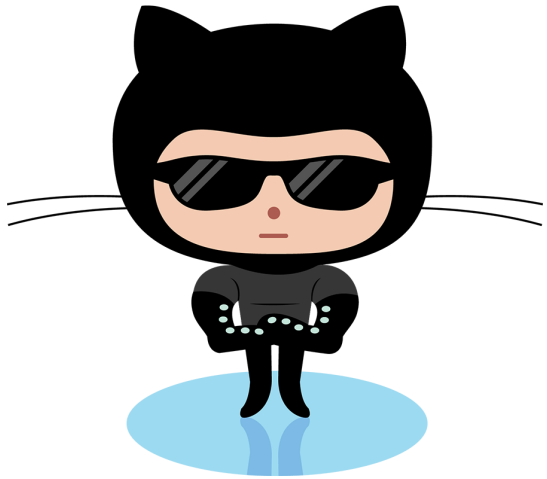
- `git log`
  List commits in reverse chronological order
- `git log -p`
  Shows diff introduced in each commit
- `gitk`
  Graphical tool to visualize history



Basic commands

Undo

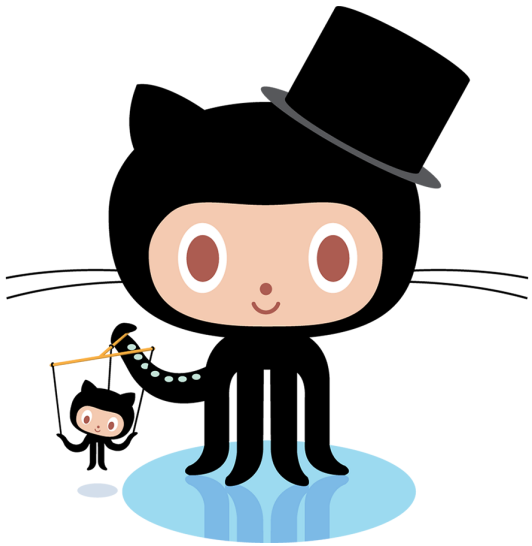
- `git commit --amend`
  Changes the last commit
- `git reset HEAD` *`file`*
  Unstage a changed file (does not undo modifications)
- `git checkout --` *`file`*
  Undo local modifications to a file
- **Careful, anything committed in git can be recovered, everything else is likely never to be seen again!**



Basic commands

Remotes

- `git remote -v`
  List your remotes (with url), ex. output:
  ```
  $ git remote -v
  dan git://danielinux/dan/picotcp.git
  origin git@github.com:tass-belgium/picotcp.git
  ```
- `git remote add` *`remotename url`*
  Add remote with name
- `git fetch` *`remotename`*
  Fetch all data from remote, not often used
- `git pull`
  Fetch data from remote (you cloned from) and merge it into the current branch
- `git push` *`[remotename]`* Push local commits to a remote



Working with Git

Working with Git

Git workflow (commit-level)

- *hack*hack*hack*
- Put files into staging area
- Commit

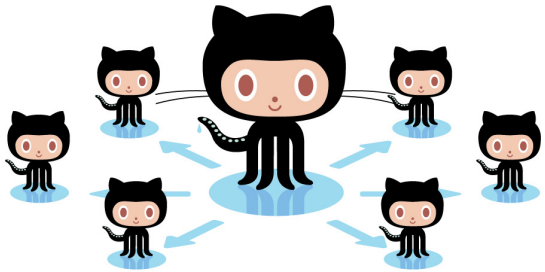
Example

```
$ # edit parameters in 2 config files
$ git add foo.cfg bar.cfg
$ git commit -m "Adjusted velocity parameters in foo.cfg
bar.cfg linked to issue #365"
```

Branch Workflow

Branches

- Branching is diverging from the main line and working without messing up that main line
- `git branch` *`branchname`*
  Create a new branch
- `git checkout -b` *`branchname`*
  Create a new branch and switch to it
  - Because usually you want to work in the branch you just created...
- `git branch -d` *`branchname`*
  Deletes a branch, does not work if you have unmerged commits, use: `-D`



Branch Workflow

Merging

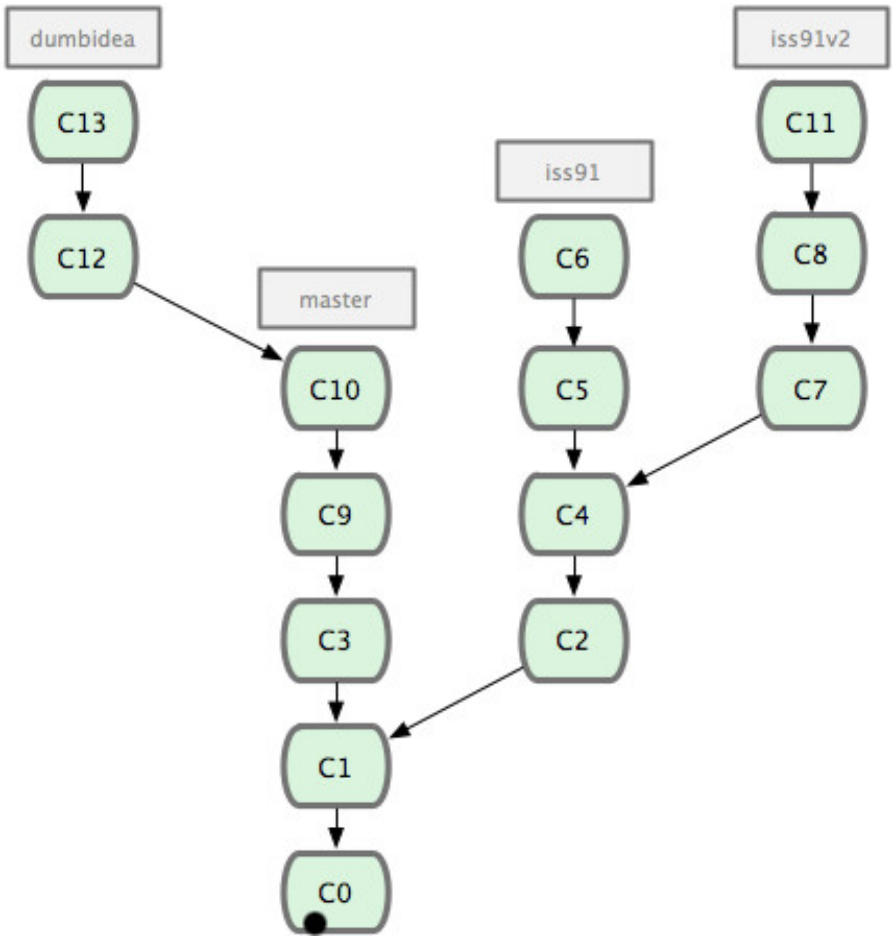
- `git checkout` *`master`*
  `git merge` *`branchename`*
  Merge a branch back into master
- Git determines the best common ancestor between your two branches
- Merge conflicts:
  - Happen when you change the same part of a file differently between branches
  - Git pauses the merge process
  - Look for conflict-resolution markers in the conflicted files
    or use `git mergetool`
  - run `git commit` after resolving the conflict

Branch Workflow

Topic Branches

Create a different branch for each feature
Example:

- Do some work on `master`
- Branching off for issue 91 `iss91`
- Trying a new way of working in `iss91v2`
- Doing some more work in `master` and a new branch with an idea you are not sure of `dumbidea`

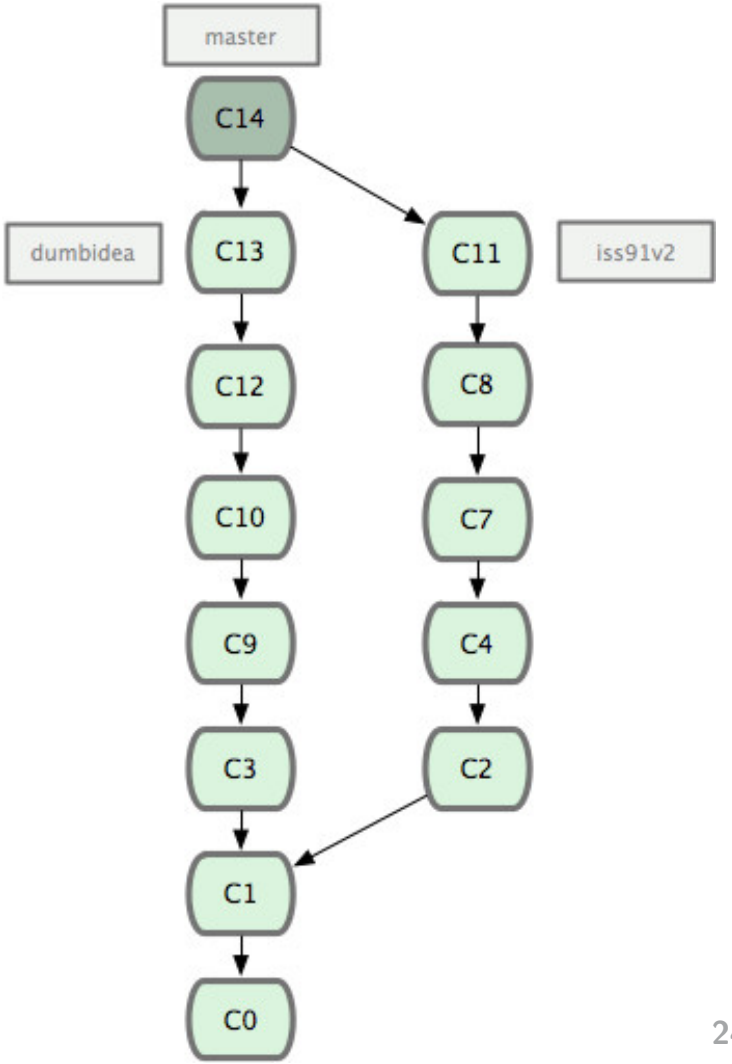


Branch Workflow

Topic Branches

Example:

- The (dumb) idea was genius
- New way of working is preferred on issue 91
- `iss91` can be removed, merge branches `dumbidea` and `iss91v2`



Branch Workflow

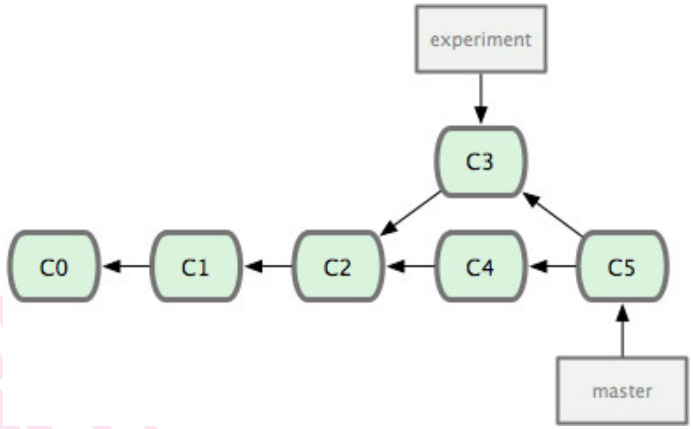
Remote branches

- Very useful if you want to work together **and** keep a stable `master`
- `git push` *`remote localbranch:remotebranch`*
  Pushes your branch to the remote so everyone can fetch it. (Omit *:remotebranch* to give it the same name as *localbranch*)
- `git checkout --track` *`remote/branchname`*
  Creates a local editable copy of the remote branch
- `git push` *`remote`* `:`*`branchname`*
  Deletes the remote branch

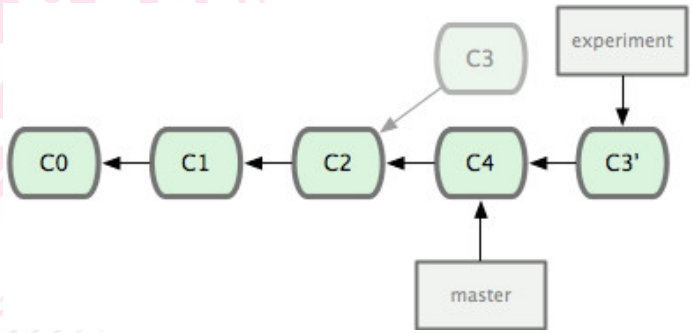
Rebasing

Basic rebase

- `git merge` *experiment*



- `git rebase` *master experiment*



Rebasing

Warning

Never rebase commits that have been pushed to a remote repository!

git-scm.com/book/en/Git-Branching-Rebasing#The-Perils-of-Rebasing

> *And then there's* `git rebase --interactive`*, which is a bit like* `git commit --amend` *hopped up on acid and holding a chainsaw - completely insane and quite dangerous but capable of exposing entirely new states of mind. Here you can edit, squash, reorder, tease apart, and annotate existing commits in a way that's easier and more intuitive than it ought to be.*
>
> – Ryan Tomayko

Power Tools

reflog

- Whenever your `HEAD` moves, Git tracks it in the reflog
- This means you can access everything that was ever stored in Git
- `git reflog` *`[reference]`*
  Shows you the reflog, default: `HEAD`
- `git branch lostandfound` *`commit-sha1`*
  Creates a new branch with the status at the commit of this SHA1
- Only shows **local** modifications!

Power Tools

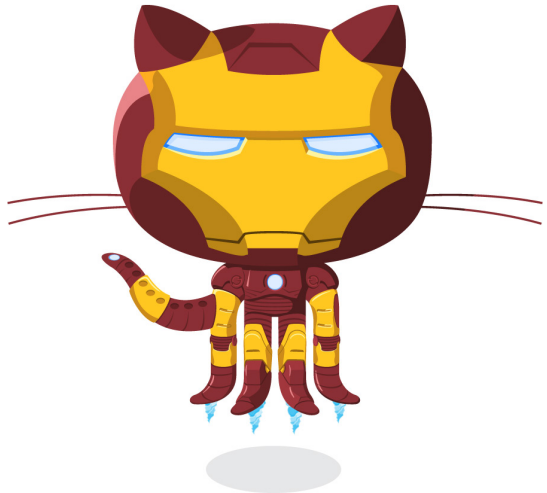
Ancestry Notation

- `^` at the end of a reference means "parent of" that commit
- `^2` means second parent, only useful for a merge commit
- `~` also means "parent of" the commit
- `~2` means first parent of the first parent
- `HEAD^^^` is thus equal to `HEAD~3`

Power Tools

Various

- `git cherry` *`[upstream] [branch]`*
  Compare changesets between forks, shows a + before each commit you have that upstream doesn't
- `git cherry-pick` *`commit-sha1`*
  Creates a copy of the selected commit on top of your `HEAD`
- The **mighty** `-p`:
  `git add` **`-p`** *`file`*
  Choose interactively which changes to commit
  `git checkout` **`-p`** *`file`*
  Interactively select hunks in the difference between the index and your working structure



tass TSS TECHNOLOGY SOLUTIONS

Quick Reference

Quick Reference

Getting Help:

`git help command` or
`git command --help` Show help for a command

Repository creation:

`git init` Create a repository in the current directory
`git clone url` Clone a remote repository into a subdirectory

File operations:

`git add path`	Add file or files in directory recursively
`git rm path`	Remove file or directory from the working tree
`git mv path dest`	Move file or directory to new location
`git checkout [rev] file`	Restore file from current branch or revision

Branches:

`git checkout branch`	Switch working tree to branch
`git checkout -b branch`	Create branch and switch to it
`git branch`	List local branches
`git branch -f branch rev`	Overwrite existing branch, start from revision
`git merge branch`	Merge changes from branch

Working tree:

`git status`	Show status of the working tree
`git diff [path]`	Show diff of changes in the working tree
`git diff HEAD path`	Show diff of staged and unstaged changes
`git add path`	Stage file for commit
`git reset HEAD path`	Unstage file for commit
`git commit`	Commit staged files
`git commit -a`	Stage and commit all modified files
`git commit -m message`	Pass commit message via command line
`git reset --soft HEAD^`	Undo commit, keep changes in the working tree
`git reset --hard HEAD^`	Reset the working tree to the last commit
`git clean`	Clean unknown files from the working tree

Examining History:

`git log [path]`	View commit log, optionally for specific path
`git log [from[..to]]`	View commit log for a given revision range
`git blame [file]`	Show file annotated with line modifications

Quick Reference

Remote repositories - remotes:

`git fetch [remote]`	Fetch changes from a remote repository
`git pull [remote]`	Fetch and merge changes from a remote repository
`git push [remote]`	Push changes to a remote repository
`git remote`	List remote repositories
`git remote add url`	Add remote to list of tracked repositories

Storing your workspace - stash:

`git stash [save] [message]`	Store modifications to tracked files
`git stash -u`	Store tracked **and** untracked files
`git stash list`	List the saved stashes
`git stash apply [stash]`	Restore working state
`git stash pop [stash]`	Restore state and remove stash

Exporting and importing:

`git apply - < file`	Apply patch from stdin
`git format-patch from[..to]`	Format a patch with log message and diffstat
`git archive rev > file`	Export snapshot of revision to file

Tags:

`git tag name [revision]`	Create tag for a given revision
Options:	
`-s`	Sign tag with your private key using GPG
`-l [pattern]`	List tags, optionally matching pattern

File status flags:

??	untracked	File is not tracked by git
M	modified	File has been modified
C	copy-edit	File has been copied and modified
R	rename-edit	File has been renamed and modified
A	added	File has been added
D	deleted	File has been deleted
U	unmerged	File has conflicts after a merge