

# Version Control with Git Cheat Sheet

## Using Git with a Feature-Branch Workflow

1. Create, checkout branch.
- 2A. Create, modify, delete code. (Not shown)
- 2B. Add files to stage and review work.
- 2C. Commit files to history often. Repeat 2A.
3. Ready? Clean up and combine commits.
4. Merge branch into master; tag master
5. Push, delete branch.

## 1. Create

### **git clone** <url>

From remote history; creates local folder  
`git clone git@github.com:mikec/myproj.git`

### **git init**

Create a new Git repository from your current directory; add and commit files then:

### **git remote add** origin <url>;

**git push --set-upstream** origin master

From local history to blank remote history

`git remote add origin`

`git@github.com:mcombs964/myproject.git`

### **git remote --verbose** # verify origin url

If the url is wrong you can use:

**git remote remove** origin

## 1. Branch

### **git branch** [--all | --verbose]

List branches. --all shows local and remotes

### **git branch** <new-branch> [source-branch]

Create local branch based on HEAD or source

### **git checkout** <branch1>

Switch to branch

### **git branch --move** <branch1> <branch2>

Rename branch1 to branch2

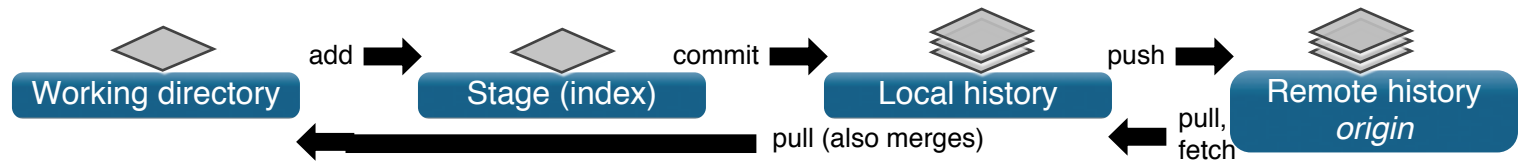
### **git branch --delete** <branch1>

Delete local branch1

### **git stash** [list | apply | drop]

Stores code so you can switch branches without commit. Restore with apply, remove with drop.

<https://mikec964.github.io/>



## 2B. Add/Reset

### **git add** . | --update | --patch <files> | <files>

Add all new/modified/deleted or specified files to stage. --update skips new files. --patch has interactive prompts to add parts of files.

### **git mv** <files>

Rename or move files; update stage

### **git rm** [--cached] <files>

Delete file from working area and index. --cached removes from history only.

### **git reset** [files]

Unstage uncommitted work; remove all (or specified) files from stage without changing working directory.

### **git reset --hard**

**DANGER!** Delete uncommitted work.

### **git clean** [--dry-run] [--force]

**DANGER!** Delete unstaged files.

## 2B. Review Work

### **git status**

Files staged and in working directory.

### **git diff** [file] # working vs stage code changes

Code changes between working and stage

### **git diff HEAD** # working vs last commit code changes

Code changes between working and last commit

### **git diff --cached** # stage vs last commit code changes

Code changes between stage and last commit

### **git diff** <commit1> <commit2> # history code changes

### **git log** [--oneline] [--graph] [--decorate]

History of commits

### **git log** <branch> --not master

History of commits for branch

### **git show** <commit>[:<file>]

History of commits and code changes. :file narrows scope

### **git reflog** [--relative-date] [--all]

Show changes to HEAD and SHA-1s

@mike3d0g

## 2C. Commit/Revert

### **git commit** [--all] [--message "<description title>"]

Copy staged changes into local history. See *Commit Message Style on back*.

### **git commit --amend**

Combine new changes with last commit, overwrite last description

### **git reset HEAD^**

Undo local, unpushed commits. Roll back code (move HEAD and branch) to prior commit.

### **git revert** <commit>

Unapply pushed, remote commits. Undoes changes in commit, then creates a new commit.

### **git checkout** <commit> <file>

Bring file from specified commit in local history to working directory.

## 3. Clean Up Commits

**DANGER!** Rewriting history (this) is for private, not public branches.

### **git rebase --interactive** master

Clean up/combine commits and modify history of current branch. In editor, change command per line to pick, squash, etc.

After rebase, use **push origin [branch] --force**

## 4. Resolve Merge Conflicts

Git will try to resolve merges. Successes will be staged, conflicts will be unmerged. Use git status to list them. See *Figures 1, 2 on back*.

### **git checkout master; git merge** <branch>

Merge branch into master.

### **git checkout master; git merge --no-ff** <branch1>

Merge without fast-forward to create a merge commit. This aids history visualization. See figure 2.

### **git diff** [--base | --ours | --theirs] <file>

Compare file to master (base) file, current master (ours) changes, branch (theirs) changes, or all. See figure 4.

## 5. Push/Pull

### **git push** origin [branch] [--all] [--tags]

Pushes current branch. --all pushes all branches, --tags pushes tags.

### **git push --delete** origin <branch>

Delete branch on origin, retain local branch

### **git pull** origin <branch>

Get changes from remote and merge

### **git fetch** origin <branch>

Get changes from remote without merge

### **git cherry-pick** <commit>

Bring changes (not all files) from a commit in history into working directory

### **git tag** [--list [pattern]] [-n [num]]

List tags. -n shows 1 or num lines of annotation

### **git tag** [--delete] <tagname> [old-commit]

Tag current branch or old commit

### **git push --delete** <tagname>

Delete tag in remote history

## Fix Merge Conflicts Manually

1. Identify which files have merge conflicts with git status.
2. Manually resolve conflicts in each file with **vim** or **mergetool**. Conflicts are marked with <<<< through >>>>.

```
unchanged code for context
<<<< HEAD (current branch
marker)
current code
==== (branch separator)
incoming code
>>>> branch-name (incoming
branch marker)
```

3. Choose current or incoming code or merge, then delete markers and separator.
4. Use git add for resolved file, delete .orig file.
5. Use git commit when all files are resolved.

### **git mergetool** <file>

Launch previously configured GUI mergetool. Make appropriate changes, then confirm at CLI.

Updated: July 13, 2019

## Git Basics

This cheatsheet is organized around the Feature Branch workflow with common commands and undos. The local repository is called *history* and the remote repository is called *remote history* (or *origin*, its most common name). The default branch name in a new history is **master**. Options are shown in long form for better mnemonics; --message instead of -m.

**HEAD:** current commit (and branch)

**HEAD^:** First parent of HEAD

**<commit>:** HEAD, tag name, branch name, or leading substring of the commit SHA-1  
**<file>:** filespec

**<branch>:** branch names cannot contain spaces

**The Feature Branch Workflow** assumes a new branch for every new feature. The master branch never has broken code, and released versions are tagged. Feature branches might be merged into a dev branch, merged into master per release.

**Commit Early and Often** Commits should be atomic (implement one feature or fix).

[Committing a partial file](#) may help. Regardless, commit anytime you want and then use `rebase --interactive` to clean them up.

### Commit Message Style

Title (50 chars), blank line, body (72 chars wrap). Semantic title example: "feat: add play gesture", types are **chore**, **docs**, **feat**, **fix**, **refactor**, **style**, or **test**. See [Semantic Commit Messages](#).

### When to Push

Private branches can be pushed anytime with the understanding that nobody else will check them out. Others should be cleaned up before pushing.

### Merge into the Master Branch

Feature branches with one commit can use fast forward commits; others should use merge commits. After merge, delete the branch.

Don't forget: **git help** [command]

## References

[Git in Practice](#) by Mike McQuaid • [Official Docs](#) at git-scm.com • [A Visual Git Reference](#) • [Visualizing Git Concepts with D3](#) (interactive) • [Atlassian Git Tutorial](#) • [Interactive Git Cheatsheet](#) • [Escape a Git Mess](#)

## Configure

**git config --global user.name** "Mike Combs"

**git config --global user.email** "mike@example.com"

**git config --global core.editor** "vim"

**git config --global merge.tool** "diffmerge"

**git config --global credential.helper** osxkeychain

Use keychain for passwords instead of reprompting. Also: git-credential-gnome-keyring or git-credential-winstore

**git config --global --edit**

Open ~/.gitconfig global config file in editor for editing

### vim .gitignore

Edit this to ignore temporary, object, project, and other files. Find examples at <https://github.com/github/gitignore>. For example:

```
.DS_Store
__pycache__/
*.py[co]d
static/
```

To track static dir but not its contents:

`touch static/.gitignore; git add -f static/.gitignore`

**git rm --cached <files>**

Remove files from history in case they got there before you put them in .gitignore

**git status --ignored**

Show files ignored due to .gitignore

**git config --global alias.<name> <cmd>**

Define an alias for <cmd>

`git config --global alias.loga "log --graph --decorate --oneline"`

**git <alias>**

Use previously defined alias

## Log Options

**git log <since id>...<until id>** # show range of commits

**git log <limit>** # limit number of commits shown

**git log --author="<pattern>"**

**git log --decorate** # show branch and tag names

**git log --graph** # show graph of commits

**git log --grep="<pattern>"**

**git log --oneline** # show each commit on 1 line

**git log -p** # show full diff

**git log --stat** # show files and changed line counts

## Regression

**git blame --date=short -w -s -L 40,60 <file>**

For each line in file, show author, date, and commit. -w ignores white space, -s hides author name, -L specifies range of lines

**Bisect** uses a binary search of history to help find a commit that introduced a bug.

**git bisect start; git bisect bad;**

**git bisect good <commit>**

Start regression process, indicate HEAD is bad, identify last known good commit. Bisect will now checkout a revision within these bounds.

**git bisect <bad | good>**

After you check for a problem, use this to checkout the next commit. Repeat until problem is isolated.

**git bisect reset**

End regression process, return to HEAD

**git bisect log**

Show bisect steps

## Rebase

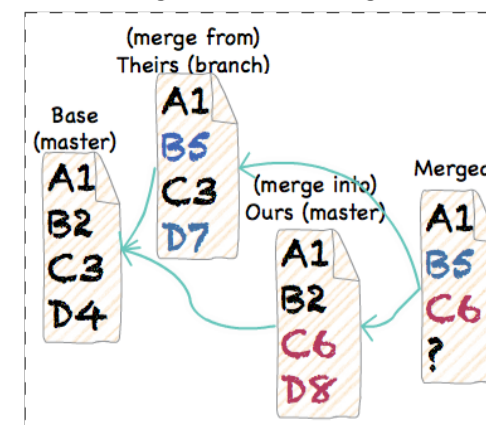
**git rebase master <branch>**

If master changed after branch fork, create new commits to base branch on current master. See figure 3.

## 3-Way Merge

In this merge, Git cannot automatically resolve line D because it has been changed in the branch, and then later in the master. Usually *ours* is the current master branch, and *theirs* is the feature branch.

Fig 4: Three-Way Merge



## FF-Merge, Merge Commit

In the figures, 'A' is the first commit and subsequent commits point to their parents.

Fig 1: Master and Feature branches

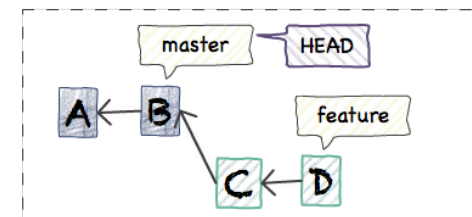


Fig 2A: After merge commit

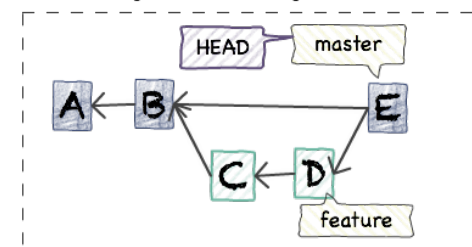


Fig 2B: After fast-forward merge

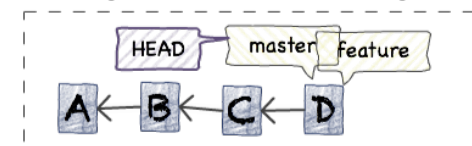


Fig 3A: After branch, commit added to Master

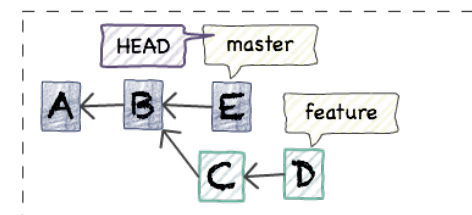


Fig 3B: Feature rebased on Master

