Hands-on with Git

Sid Anand (LinkedIn)
July 16, 2012

Git Concepts

Git Concepts

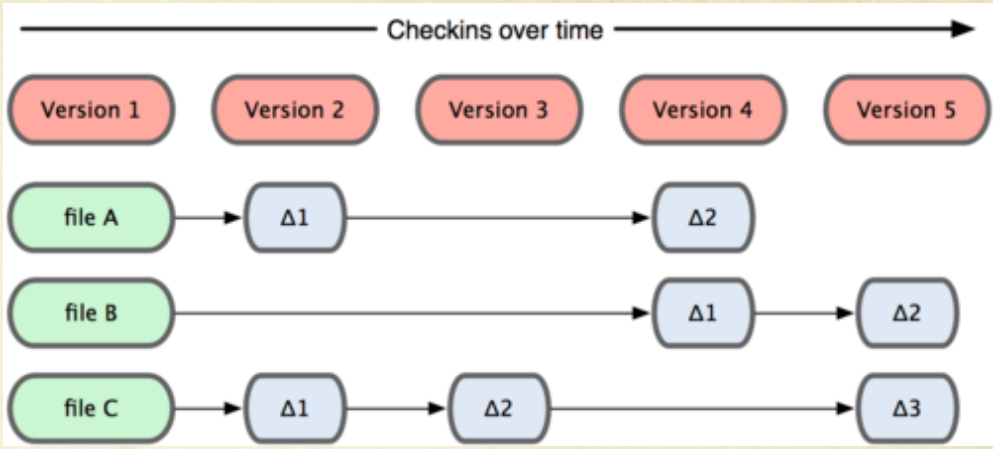
- Overview
  - A distributed VCS (Version Control System) tool to manage version changes in files and directories
  - Other VCS tools include CVS, Subversion, Clearcase, Perforce, etc…
    - They differ more from Git than they do from each other
- **Major Differences**
  - Git stores commits as file system snapshots, not delta files (c.f. next slide)
  - Nearly all operations in Git are local (e.g. lookup change history)
    - Incredibly fast to develop changes
    - Supports off-line work (commits)
  - Most commits are “adds”, so it is very difficult to lose data

Git Concepts (Deltas vs. Snapshots)

Commits Deltas

- Conceptually, most other systems store information as a list of file-based changes
- These systems think of the information they keep as a set of files and the changes made to each file over time

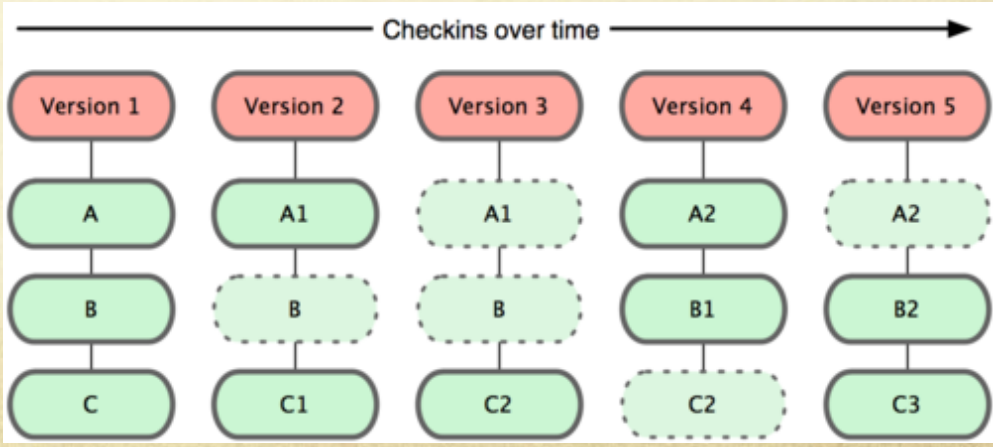
CVS, Subversion, Bazaar, etc...



Commits Snapshots

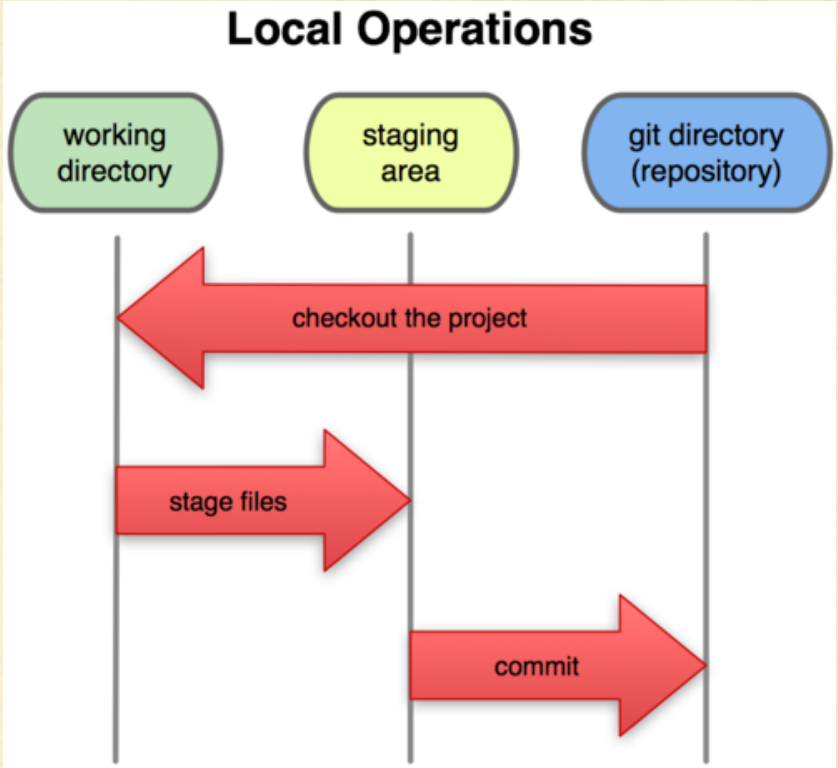
- Git thinks of its data more like a set of snapshots of a mini file system
- Every time you commit, Git takes a snapshot and stores a reference to that snapshot
- If files have not changed, Git doesn't store the file again—just a link to the previous identical file it has already stored

Git



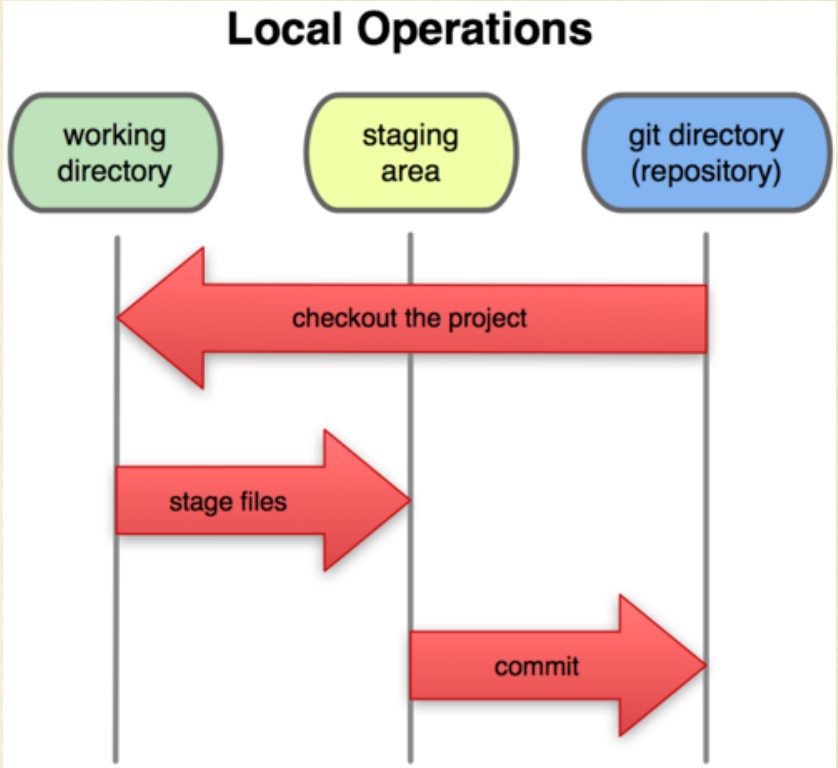
Git Concepts (3 States)

- Files reside in 1 of 3 states:
  - **Committed** means that the data is safely stored in your local repository
  - **Modified** means that you have changed the file but have not committed it to your repository yet
  - **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot



Git Concepts (3 States)

- There are 3 file locations corresponding to these 3 states:
  - **Repository** stores the **Committed** files
  - **Working Directory** stores the **Modified** (and unmodified) files
  - **Staging Area** stores the **Staged** files



In other VCS, there are just 2 locations and 2 states :

- Modified in your local directory
- Committed in the remote repository

Git Example 1

Working with an Existing Repository & Trying out Basic Commands

Git Example 1

Step 1 : Clone a remote repository

Clone (e.g. checkout in other VCS) the "curator" GitHub repository

```
$  cd $HOME/my_projects
$  git clone https://github.com/r39132/curator.git
```

This will download a repository (e.g. Curator) from GitHub, including all project history, to a local repository on your machine.

```
Cloning into 'curator'...
remote: Counting objects: 9216, done.
remote: Compressing objects: 100% (1958/1958), done.
remote: Total 9216 (delta 5752), reused 9060 (delta 5656)
Receiving objects: 100% (9216/9216), 2.33 MiB | 870 KiB/s, done.
Resolving deltas: 100% (5752/5752), done.
```

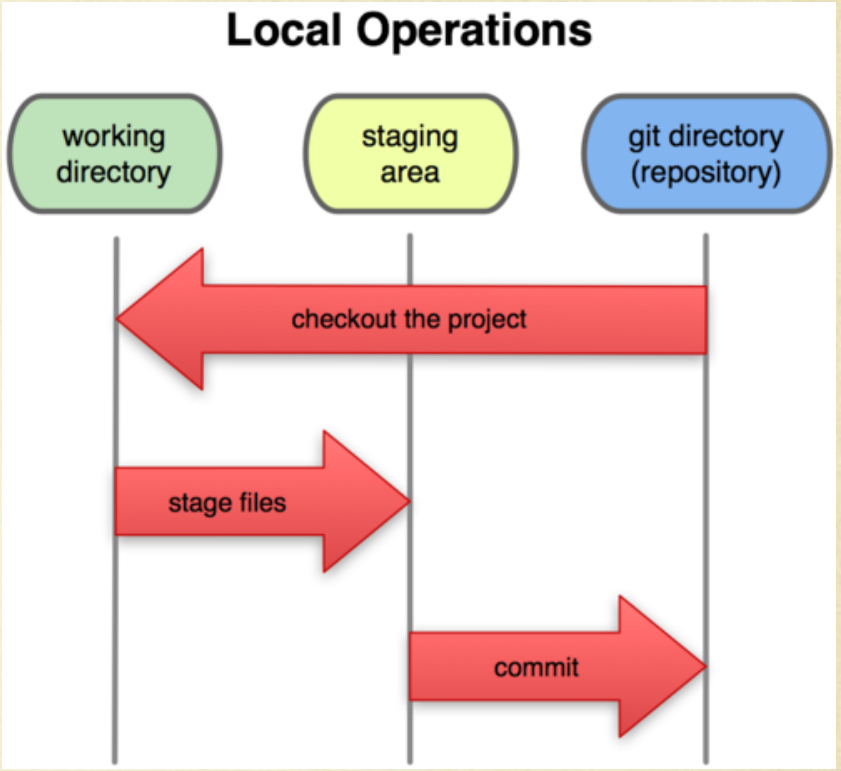
Git Example 1

Step 2 : Create a new file somefile.txt

```
$   cd $HOME/my_projects/curator
$   vim somefile.txt
```

This file is in the working directory only. "git status' shows you this file is untracked and cannot be committed as is.

```
$   git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#               somefile.txt
nothing added to commit but untracked files present (use "git add" to track)
```



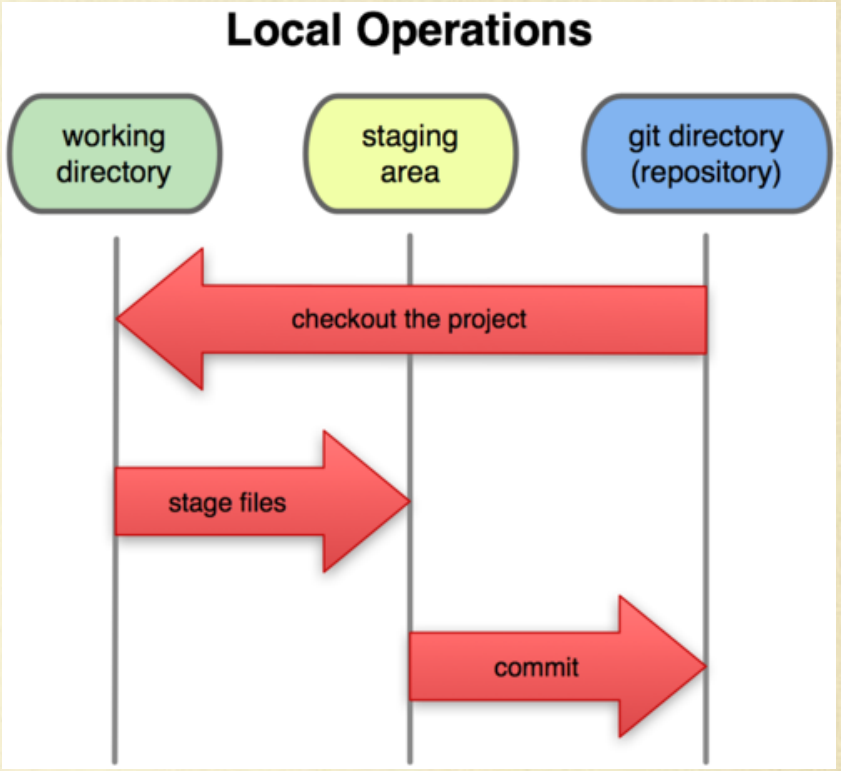
Git Example 1

Step 3 : Add somefile.txt to git

```
$   git add somefile.txt
```

This file is now staged – in effect, this file is now in the staging area. "git commit" will commit all staged files.

```
$    git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#               new file:   somefile.txt
#
```



Git Example 1

Step 4 : Commit all staged files

$ git commit –m "Add somefile"

Commits all files in the staging area to the local repository and adds a commit message. "git status" shows a clean working directory – i.e. matching the local repository.

$ git status

On branch master

nothing to commit (working directory clean)

To see all commit history, use "git log"

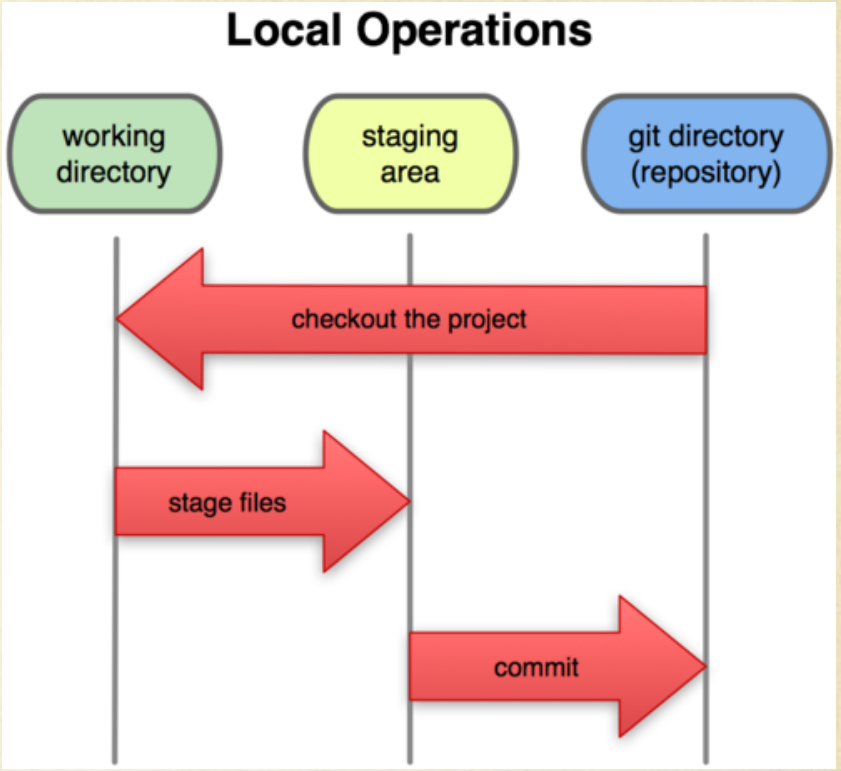
$ git log

commit f3623b7bd4f427b54a41d3597d4a4c29aa9c3e5e

Author: Sid Anand <sianand@linkedin.com>

Date: Sun Jul 15 23:16:30 2012 -0700

Add somefile

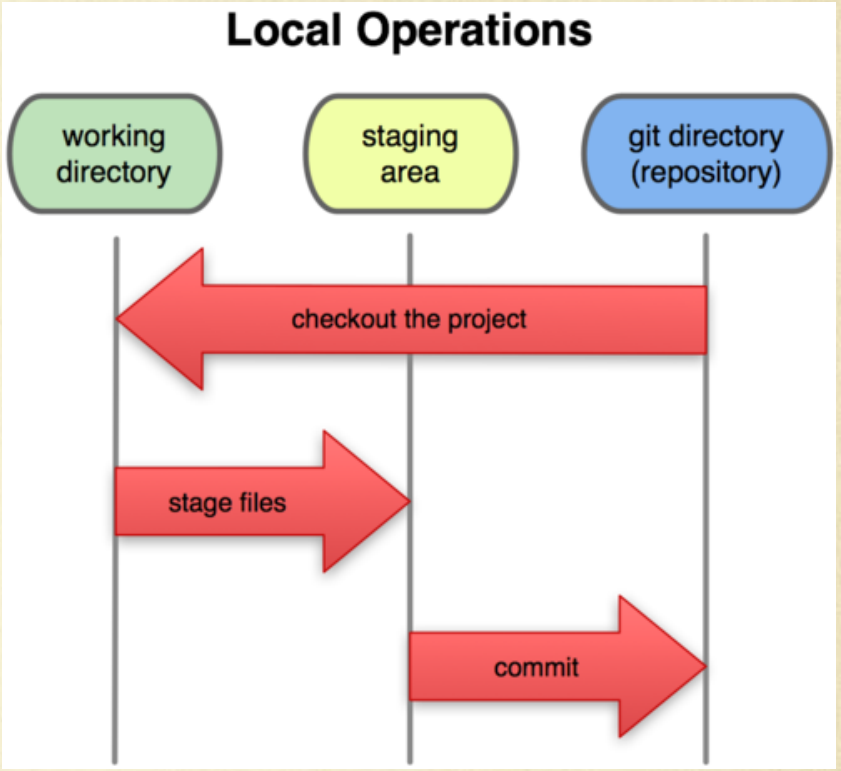


Git Example 1

Step 5 : Modify somefile.txt

```
$    echo "Hello 1" > somefile.txt
```

Git recognizes that the file has been modified, but does not stage it unless you call "git add".




```
$    git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#               modified:   somefile.txt
#
```

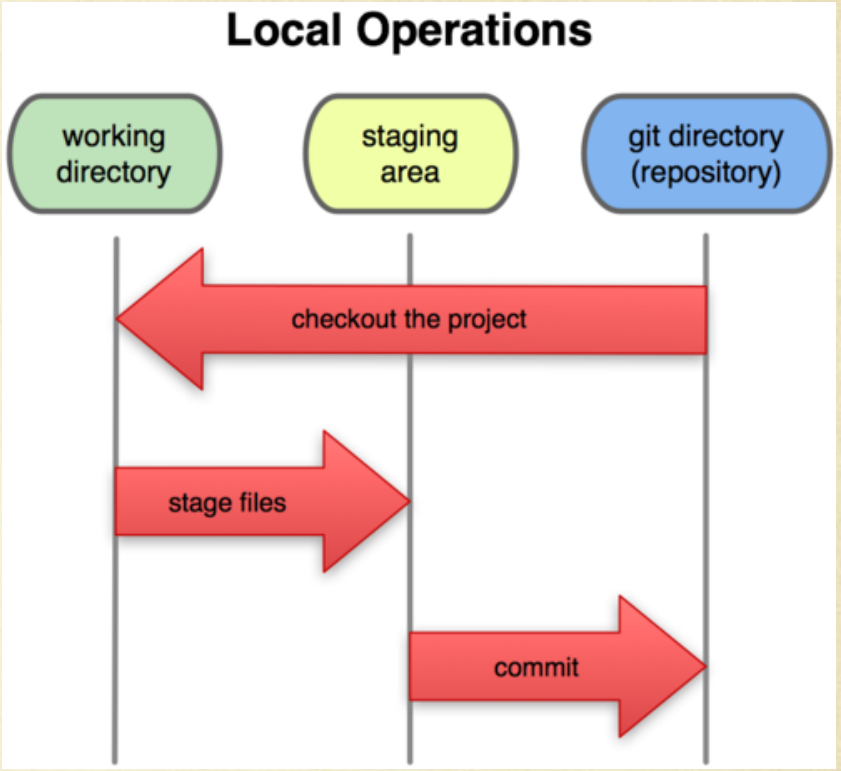
Git Example 1

Step 6 : Stage somefile.txt

```
$    git  add somefile.txt
```

Git stages a snapshot of the file at the time when "git add" was called. "git status" shows the file will be committed in the next call to "git commit".

```
$    git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#               modified:   somefile.txt
#
```



Git Example 1

Step 7 : Make subsequent modifications to somefile.txt

```
$    echo "Hello 2" >> somefile.txt
```

If you make subsequent modifications to the previously staged file, those recent changes are not staged automatically. "git status" shows that a version of somefile.txt is ready for commit and that a subsequent version is unstaged.

```
$    git  status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#               modified:   somefile.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#               modified:   somefile.txt
#
```

Git Example 1

Step 8 : Staging subsequent modifications to somefile.txt

```
$    git add somefile.txt
```

Staging is “snapshotting”. You need to call “git add” to stage the latest snapshot. Now all changes to this file are staged.

```
$    git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#               modified:   somefile.txt
#
```

Git Example 1

Step 9 : Making a modification and undoing changes

$ git "Hello 3" >> somefile.txt

- To discard staged changes :
  - First, unstage it : git reset HEAD somefile.txt
  - Then, discard it : git checkout -- somefile.txt

Good news! You do not need to remember these commands. "git status" always offers you all the options available.

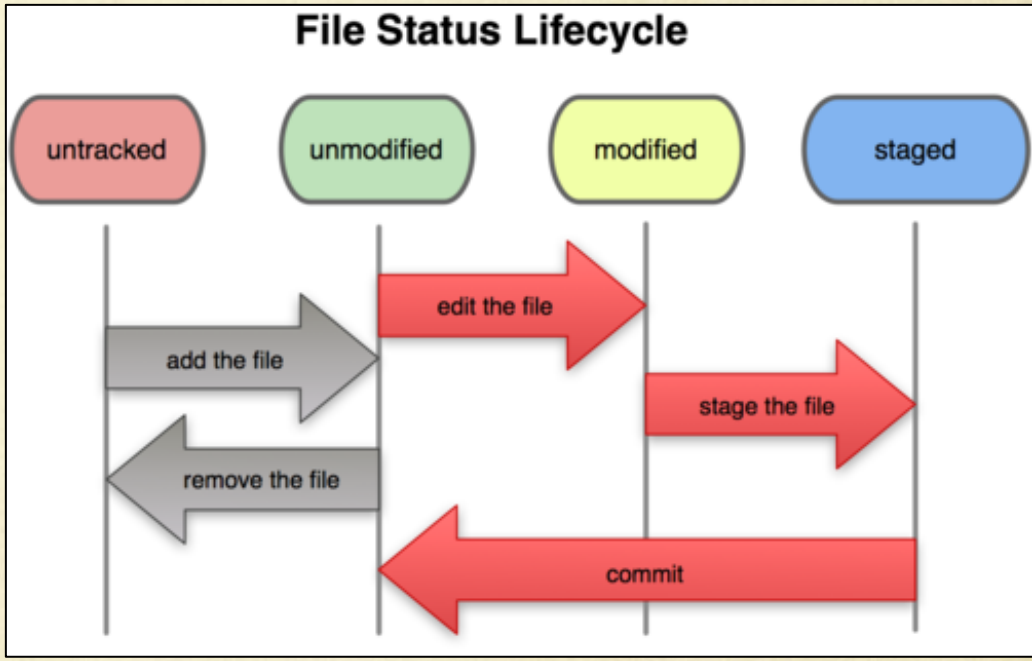
$ git status **(assuming we had staged the "hello 3" changes)**

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#               modified:   somefile.txt
#
```

Git Example 1

Summary of a File Life Cycle

- New files are untracked until you call "git add". This also stages them.
- Existing files that are modified are not staged until you call "git add"
- "git add" stages files. This is a snapshot operation and needs to be repeated if you alter staged files.
- "git commit" moves staged files to the local repository and returns the file status to "unmodified"
- "git status" shows you what files are untracked, modified, or staged. It also let you know the actions available on those files!



Git Setup

Setting up your environment

Git Setup

Step 0 : Install Git on your machine
https://help.github.com/articles/set-up-git

Step 1 : Tell Git who you are
```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Step 2 : Set up SSH keys on GitHub or Gitorious
https://help.github.com/articles/generating-ssh-keys

Step 3 : Either clone an existing remote repository or add an existing local project to Git

Clone an existing remote repository
```
$ cd $HOME/my_projects
$ git clone https://github.com/r39132/curator.git
```

Add an existing local project to Git
```
$ cd $HOME/my_projects/myProject
$ git init
$ vim .gitignore
```
This will start tracking the new project in the local repository and create a .gitignore file.

Git Setup

Setting up the .gitignore file

In a .gitignore file in your working directory, specify files and directories that Git should not version control.

What should you never check-in?

- Eclipse files (e.g. .classpath, .project, .settings/)
- Build artifacts (e.g. anything under the Maven target directory and Ant build and dist directories)

More information

https://help.github.com/articles/ignoring-files

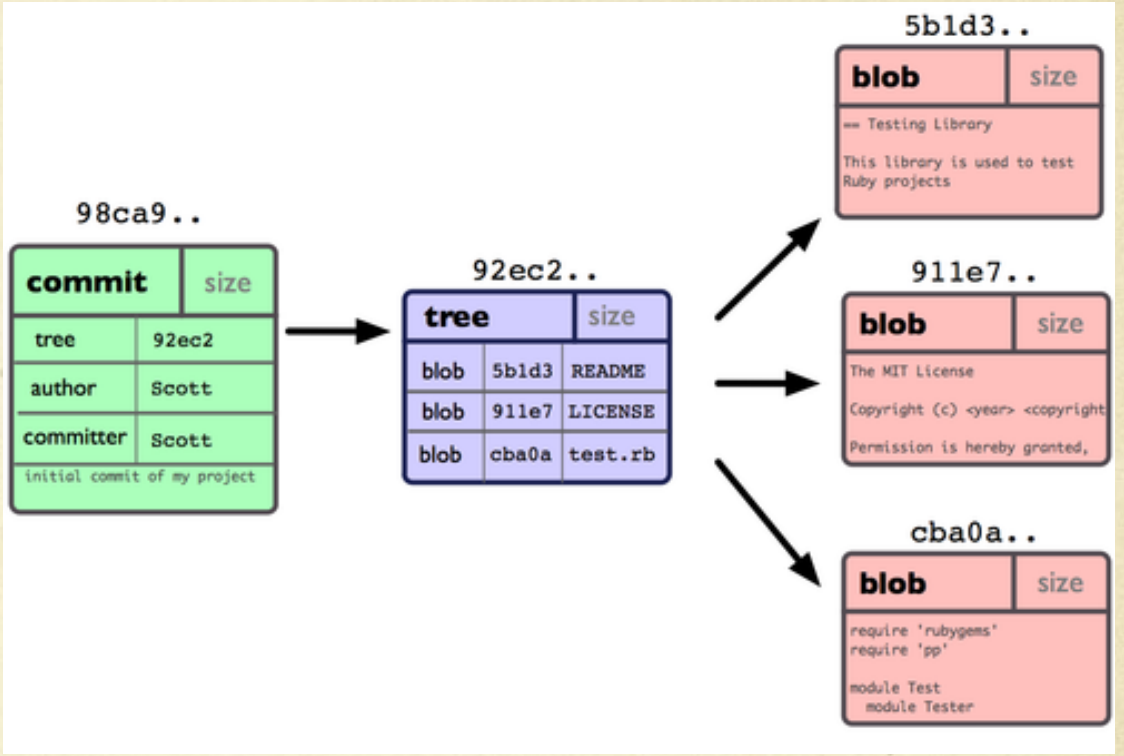
Git Branching

Working with Branches

Git Branching

Let's talk about branching with an illustrated example:

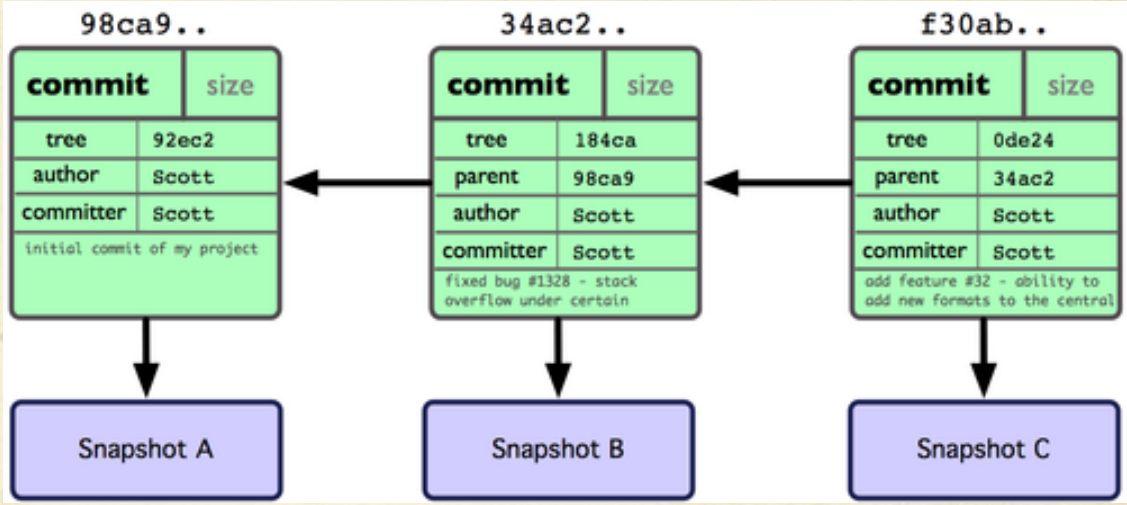
- Imagine that you commit 3 files
- Each file is stored as a blob
- A tree node is created and points to each of those 3 files :
  - for each file, the tree node holds a pointer to the file blob and its md5 hash code
- A commit node is also created and holds the following commit metadata:
  - Author
  - Committer
  - Pointer to tree



Git Branching

... an illustrated example:

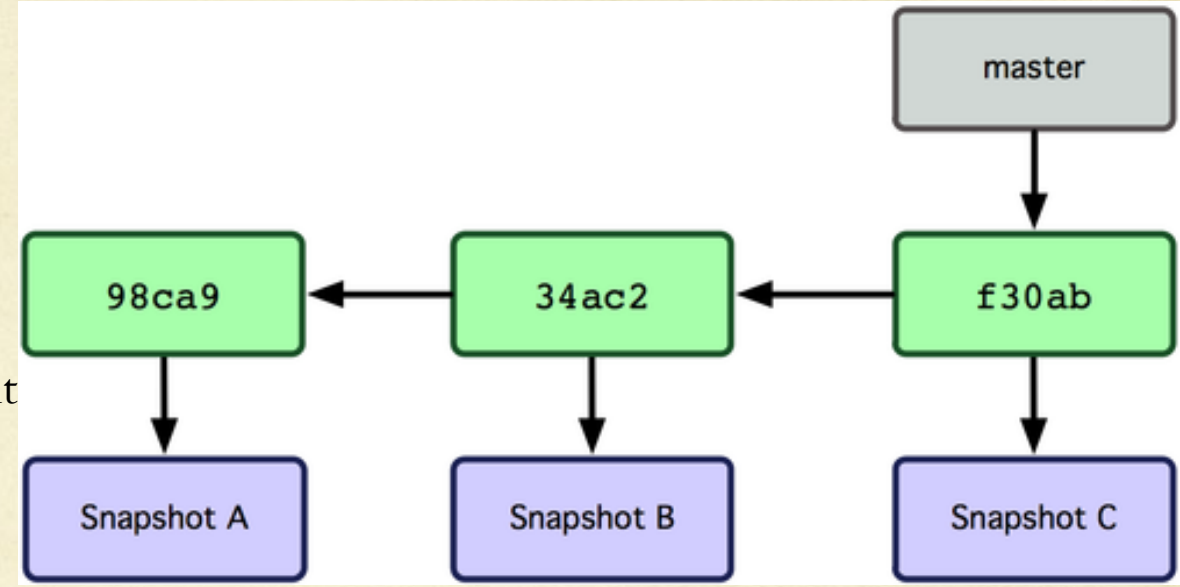
- After 3 commits, the commit nodes are chained together, with child pointing to parent
  - Snapshot A is first
  - Followed by Snapshot B
  - Followed by Snapshot C



Git Branching

… an illustrated example:

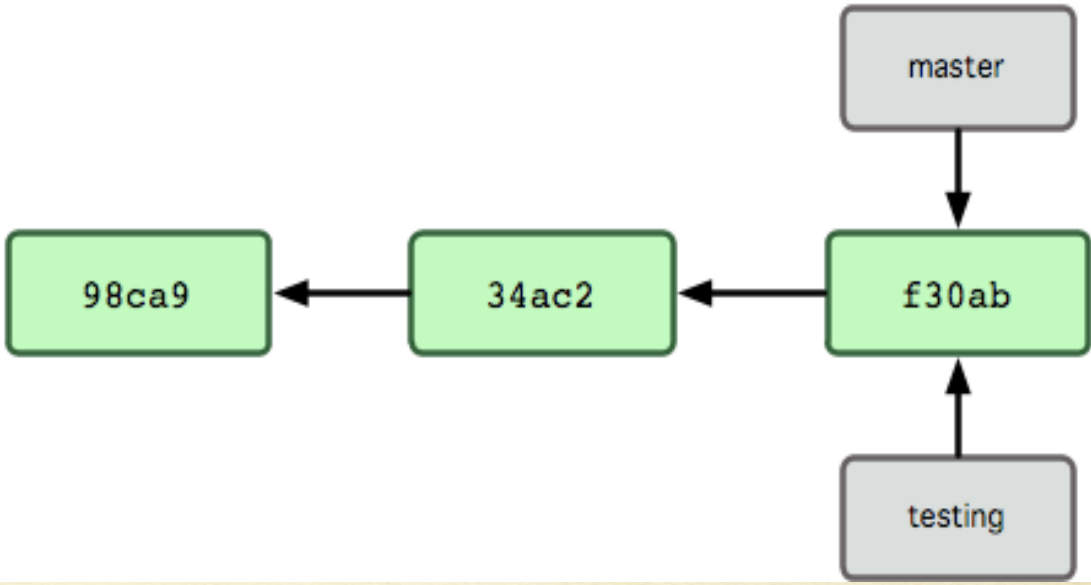
- A branch in Git is simply a lightweight movable pointer to one of these commits
- The default branch name in Git is **master**
- As you initially make commits, you're given a **master** branch that points to the last commit you made
- Every time you commit, it moves forward automatically.



Git Branching

… an illustrated example:

- Create a new branch called **testing**
  $ git branch testing
- Branching in Git is very cheap, unlike in other VCS. Git just writes a 41 Byte file out per branch!
- Git creates a new pointer called testing and points to the commit node you are currently on
- How does Git know which commit you are currently on?

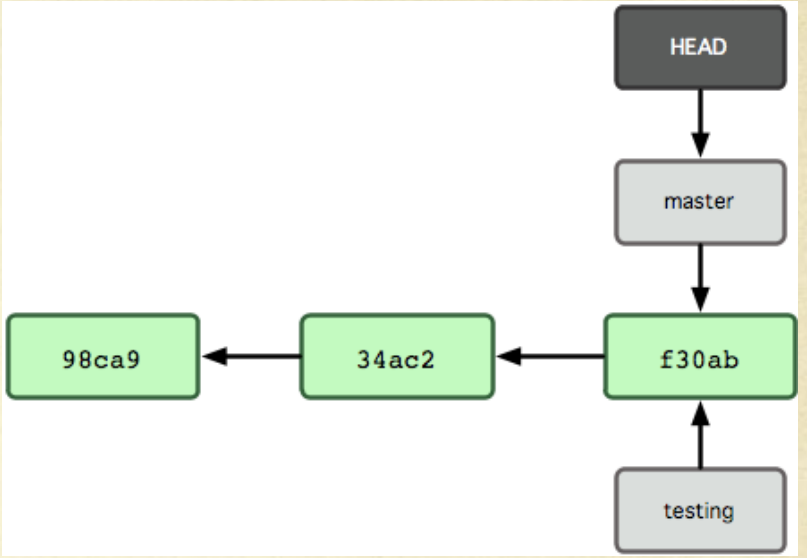


Git Branching

… an illustrated example:

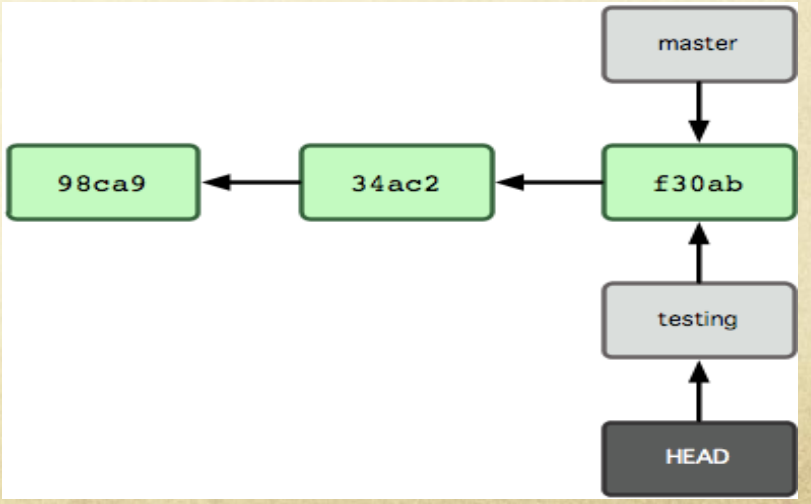
Answer : HEAD pointer

- The HEAD pointer is a special pointer to the current branch you are working on
  - E.g. HEAD is pointing to **master**



- To switch the HEAD to the **testing** branch, use checkout

```
$  git checkout testing
```



Git Branching

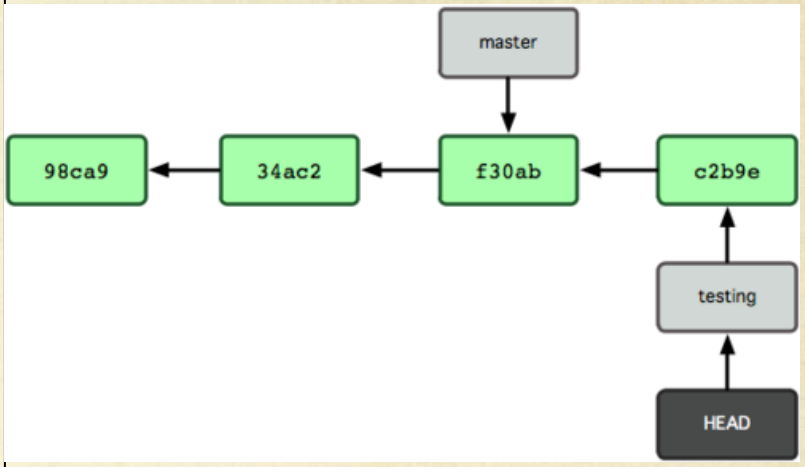
… an illustrated example:

- Now, if you commit, you will move **testing** and HEAD

  $ vim test.rb

  $ git commit -a -m 'made a change'

- The branch that HEAD points to is the one that is automatically advanced on the latest commit

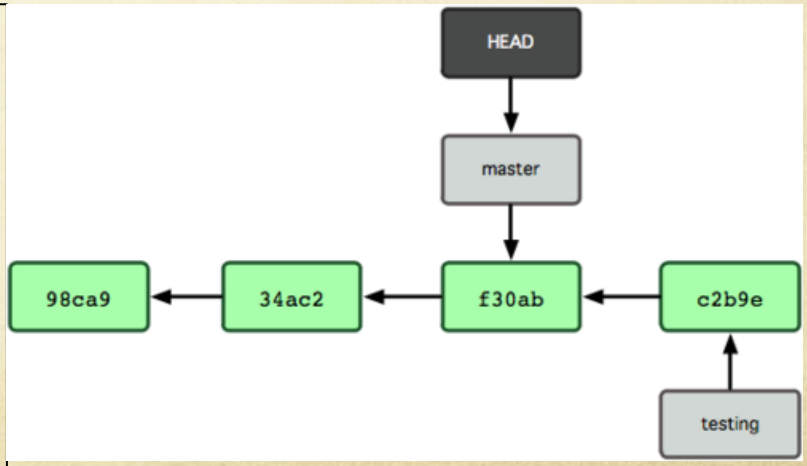


- To switch the HEAD back to the **master** branch:

  $ git checkout master

- This does 2 things:
  - Moves the HEAD to master
  - Reverts your working directory to reflect the branch you are on

NOTE : A revert (branch switch) cannot happen if you have uncommitted conflicting files in the branch you are leaving. Commit first!



Git Branching

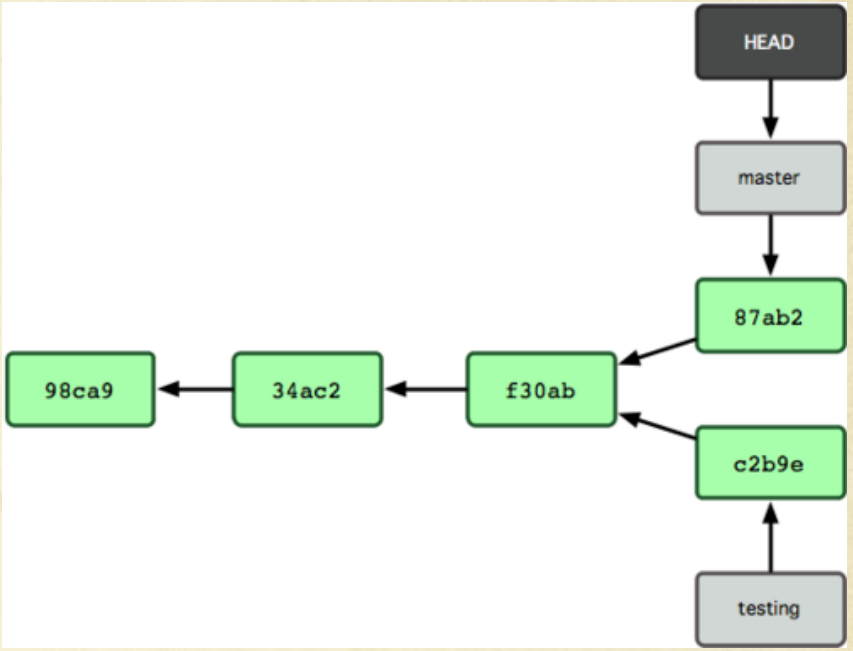
… an illustrated example:

- Now, if you commit another change, you will move **master** and HEAD

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

- **master** and **testing** have diverged

How do you merge branches?



Git Branch Merging

Merging Branches

Git Branch Merging

An illustrated example:

Consider the commits and branches on the right. There are 3 branches:

- **Master**
- **Iss53**
- **hotfix**

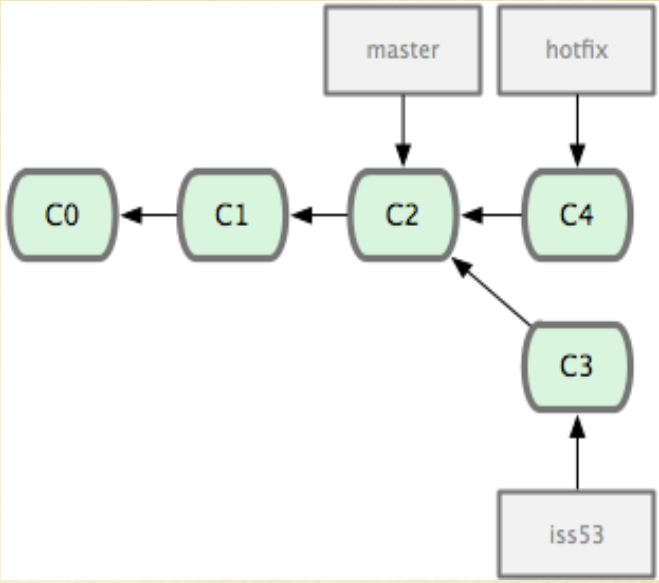
If you want to merge **hotfix** into **master**:

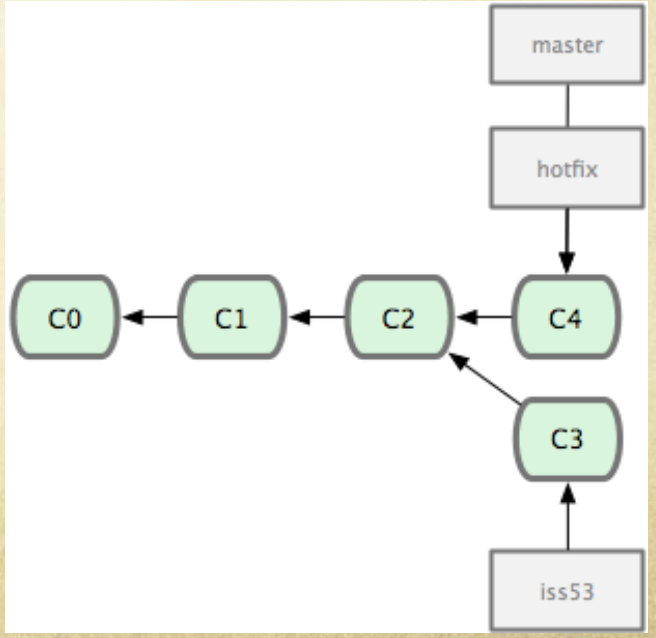
$ git checkout master // HEAD points to master, making it the current branch

$ git merge hotfix // HEAD (& master) advances to C4

```
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

This is a FAST FORWARD merge.

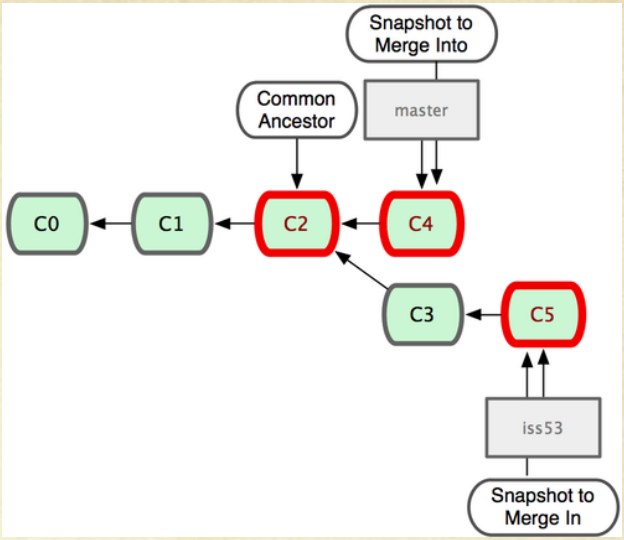




Git Branch Merging

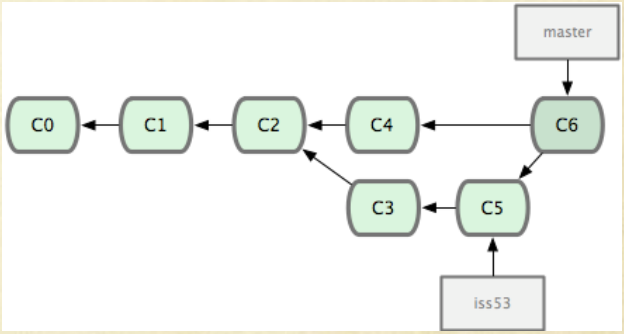
An illustrated example:
Consider the commits and branches on the right. There are 2 branches:

- **Master**
- **Iss53**



If you want to merge **iss53** into **master**:
$ git checkout master // HEAD points to master
$ git merge iss53// HEAD (& master) advances to C4

Git does a 3-way merge of the 3 highlighted commit nodes shown. Unlike the previous example where the **master** (and HEAD) pointer was just advanced to a an existing commit node, a new commit node is created here.



The difference between the 2 examples is that in the latter, a new COMMIT is created, called a MERGE COMMIT (C6).

Git Branch Merging

Resolving Merge Conflicts

What if you encounter merge conflicts?

```
$    git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

$    git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   unmerged:   index.html
#
```

Run a visual merge tool or resolve in vim/emacs as you would for SVN. Then commit.

```
$    git mergetool
$    git commit –m “Merged iss53 branch into the master branch”
```

Git Remotes

Working with Remotes

Git Remotes

Basics

- Remote refers to a remote repository
- This is different from your local repository. The clone command below copies a remote repository to the local file system

  $ git clone https://github.com/r39132/curator.git

- Most of the work that you do is against your local repository
  - E.g. git commit commits to your local repository
- When you are ready to publish your changes to a remote repository, you must "git push" it
- When you are ready to get the latest changes from a remote repository, you must "git pull" it

Git Remotes

Some Commands

List Git Remotes

```
$  git remote
origin
upstream  // Note : Learn about upstream remotes at https://help.github.com/articles/fork-a-repo
```

List Git Remotes in VERBOSE Mode (including their URLs)

```
$  git remote -v
origin     https://github.com/r39132/druid.git (fetch)
origin     https://github.com/r39132/druid.git (push)
upstream     git@github.com:metamx/druid.git (fetch)
upstream     git@github.com:metamx/druid.git (push)
```

Inspect a Remote

```
$  git remote show upstream
remote upstream
  Fetch URL: git@github.com:metamx/druid.git
  Push  URL: git@github.com:metamx/druid.git
  HEAD branch (remote HEAD is ambiguous, may be one of the following):
    master
  Remote branches:
    compression         tracked
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Git Remotes

Working with Remotes

Step 1 : Clone a remote repository

$ cd $HOME/my_projects

$ git clone https://github.com/r39132/curator.git

This creates a local copy in $HOME/my_projects/curator

Step 2 : Commit some changes locally

$ vim somefile.txt

$ git add somefile.txt

$ git commit –m "Adding some file"

Note : the last 2 lines can be done as git commit –a –m "Adding some file"

Step 3 : Pull recent changes from your "origin" remote repository

$ git pull origin master

Note : "git pull" also works

Step 4 : Publish your recent commits back to the origin

$ git push origin master

Note : "git push" also works

Git Remotes

Working with Remotes

Step 1 : Clone a remote repository

$ cd $HOME/my_projects

$ git clone https://github.com/r39132/curator.git

This creates a local copy in $HOME/my_projects/curator

Step 2 : Commit some changes locally

$ vim somefile.txt

$ git add somefile.txt

$ git commit –m "Adding some file"

Note : the last 2 lines can be done as git commit –a –m "Adding some file"

Step 3 : Pull recent changes from your "origin" remote repository

$ git pull origin master

Note : "git pull" also works

Step 4 : Publish your recent commits back to the origin

$ git push origin master

Note : "git push" also works

Git Commands Reference

Git Commands Reference

Life-Cycle Commands Summary

- git add will stage a new (untracked) file
  - For a tracked file, git add will stage changes
  - Any changes made to a staged file need to be restaged via another git add call
  - Staging is like "snapshotting" a version of a file
- git reset HEAD on a staged file will unstage the file
- git checkout -- on a modified, tracked file will return it to an unmodified, tracked state
- git diff compares your working directory to your staging area
- git diff --staged compares your staging area to your repository
- git commit -a will automatically stage all files before commit. This keeps you from doing a git add before
- git rm will remove a file. It will automatically stage it (removes it from your tracked files) and removes it from the working directory.
- git rm --cached will remove the file from the staging area (i.e. no longer tracking the file) but will keep the file in your working copy. You may want to add it to your .gitignore file in order to not see it as an untracked file on future calls to git status

Git Commands Reference

Branch Commands Summary

- **git branch** : lists the branches. a " * " next to a branch means that the branch is checked out
- **git branch --no-merged** : list branches that have changes that need to be merged into the current branch you are on
- **git branch -d <branch_name>** : This will fail for branches returned by the preceding command
- **git branch --merged** : list branches that **do not** have changes that need to be merged into the current branch you are on
- **git branch** <*branch_name*> : creates a new branch based on where HEAD is
- **git checkout** <*branch_name*> : switches HEAD to branch *branch_name*
- **git checkout -b** <*branch_name*>: creates a new branch named *branch_name* and switches HEAD to it

Branch Merging : merge *source_branch* into *target_branch*

- **git checkout** <*target_branch*>
- **git merge** <*source_branch*>

Handling Merge Conflicts

- **git mergetool**

After manual merging, then **git commit -a -m "<your message here>"**

- git branch -d <*branch_name*> : delete a branch. It will fail for branches that are checked out or that are listed in "**git branch --no-merged**"

Git Reference

Git References

Reference Manual

- http://www.git-scm.com/documentation/

99% of this content came from the ProGit book , a link for which can be found above :

Cheat Sheets

- Heroku Cheat Sheets
  - https://na1.salesforce.com/help/doc/en/salesforce_git_developer_cheatsheet.pdf

- Visual Guide : http://marklodato.github.com/visual-git-guide/index-en.html

Questions?