A few words about Git upfront

Git, created by Linus Torvalds for the Linux Kernel in 2005, is a distributed version control system that we will use for project management and group collaboration in this class. If it is completely foreign to you, it is a good idea to take this document as a crash course. Notice that for more in-depth discussion on Git, please go to http://progit.org/book, or other tutorial you may find online.

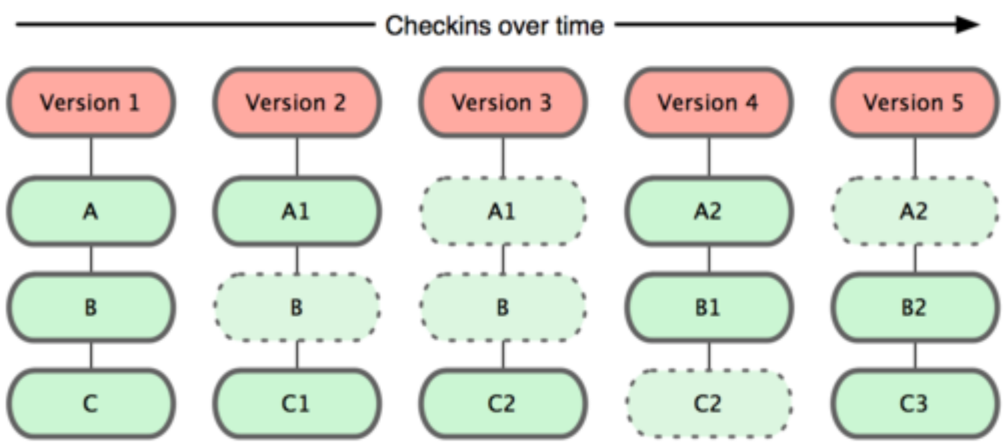
Install Git

1. Go to http://github.com, a Git repository hosting service site in cloud, create a free account.
2. Go to http://git-scm.com/downloads, download and install the latest version of Git.
3. Go to http://help.github.com, click “Set Up Git”. (Don’t skip the guide.) Follow the steps to set up Git on your computer.
4. From your local computer, there are two ways to connect to a remote server like github.com -- https or ssh. I prefer to use ssh. Go to https://help.github.com/articles/generating-ssh-keys, follow the steps to create your SSH key and then add it to http://github.com.
   Note: for step 4, if the “clip” or “pbcopy” command is not found, simply open the file ~/.ssh/id_rsa.pub and copy its contents.

Notice that the installation and configuration procedure of Git may vary between Window, Linux, and Mac OS.

Introduction to git (Part I – Everything is local)

Git thinks of its data (i.e., a project that it keeps track of) more like a set of snapshots of a mini file system. 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. 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.



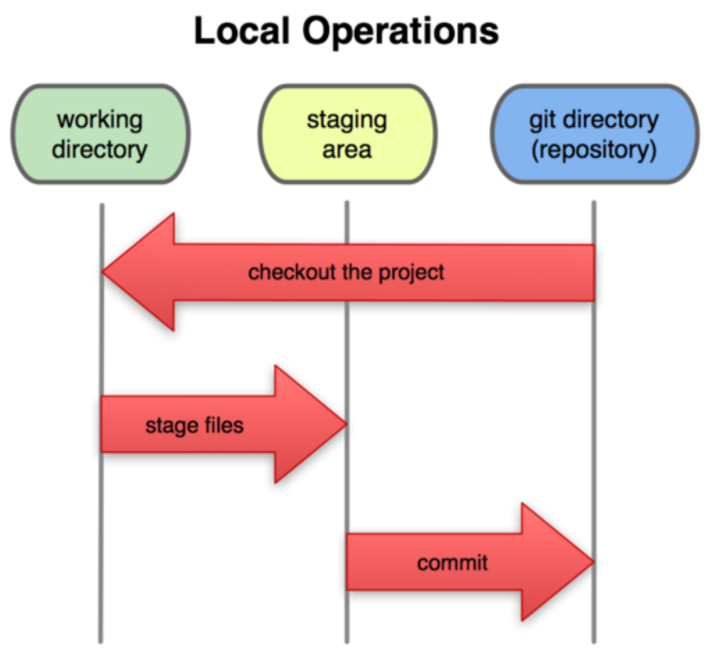
Some Git key concepts are listed below:

The Repository

In Git, your repository exists in the form of a compressed database in the .git/ directory **inside your project's directory** on your local computer. This means you can look at the entire history of the repository and see what has changed without having to communicate with a repository on another remote server.

The Three States

This is one of the main things to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: committed, staged, and modified. **Committed** means that the data is safely stored in your local database. **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot. **Modified** means that you have changed the file but have not committed it to your database yet.



Working Directory/Working Tree/Working Copy in the picture above is your current view into the repository or a single checkout of one version of the project. Checking out is the process Git uses to change your working tree to match a certain point in the repository.

The basic Git workflow goes like this:

1. You modify files in your working directory.
2. You stage the files, i.e., adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

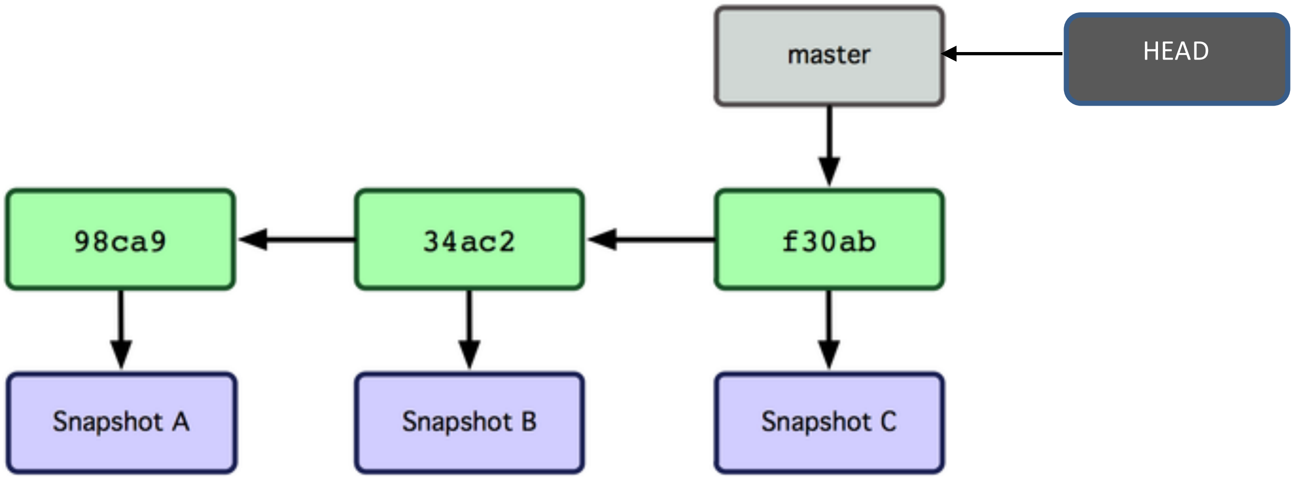
How do you get your repository in the first place? You may ask. You can start your own project and then tell Git to initialize a repository for it by issuing the command `git init`; or you can *clone* an existing repository from another server. Cloning makes a copy of another repository and then checks out a copy of its master branch – its main line of development. We will have more discussion on cloning later.

Branches - Basics

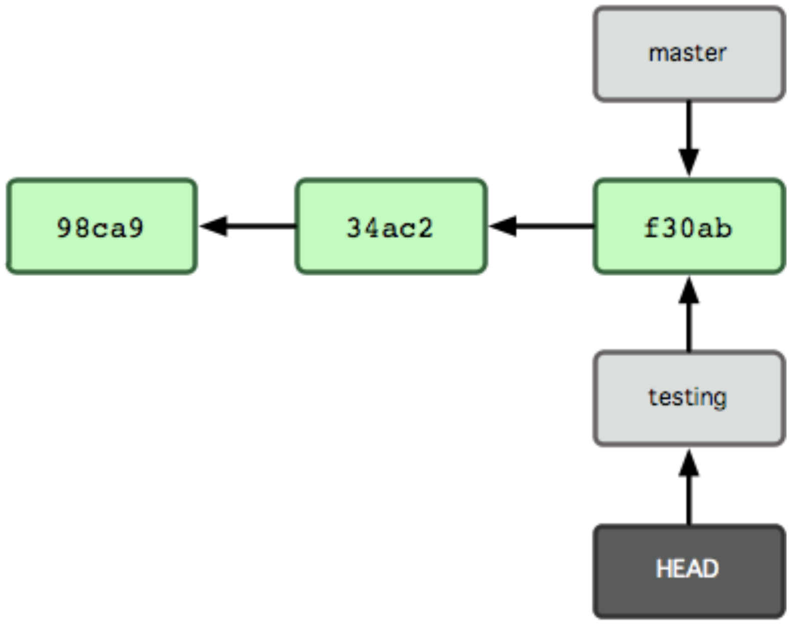
This is Git's killing feature. Branches are to create alternate histories of your project.

A branch in Git is simply a lightweight movable pointer to one of the past 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. E.g. in the following example, you've made three commits so far, first 98ca9, then 34ac2, finally f30ab.

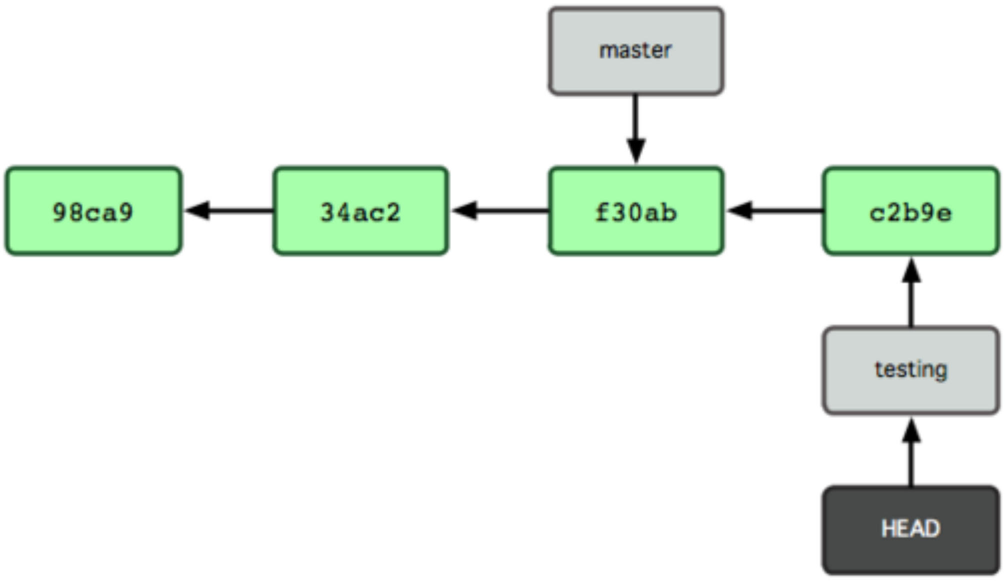
Each commit (except the first one) points to its parent. "HEAD" is a special pointer for the current branch.



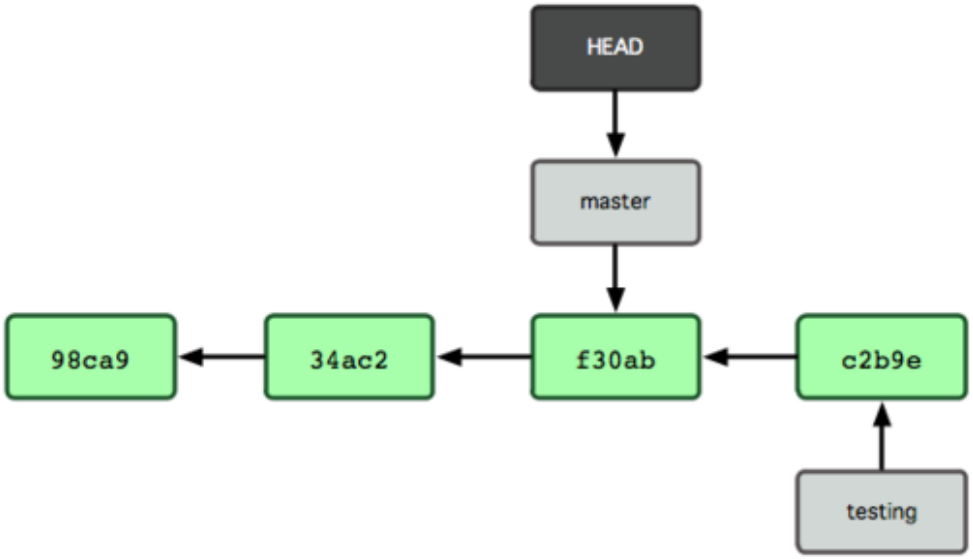
What happens if you create a new branch? Well, doing so creates a new pointer for you to move around. E.g. the following shows the changes if you issue the command `git checkout -b testing`,



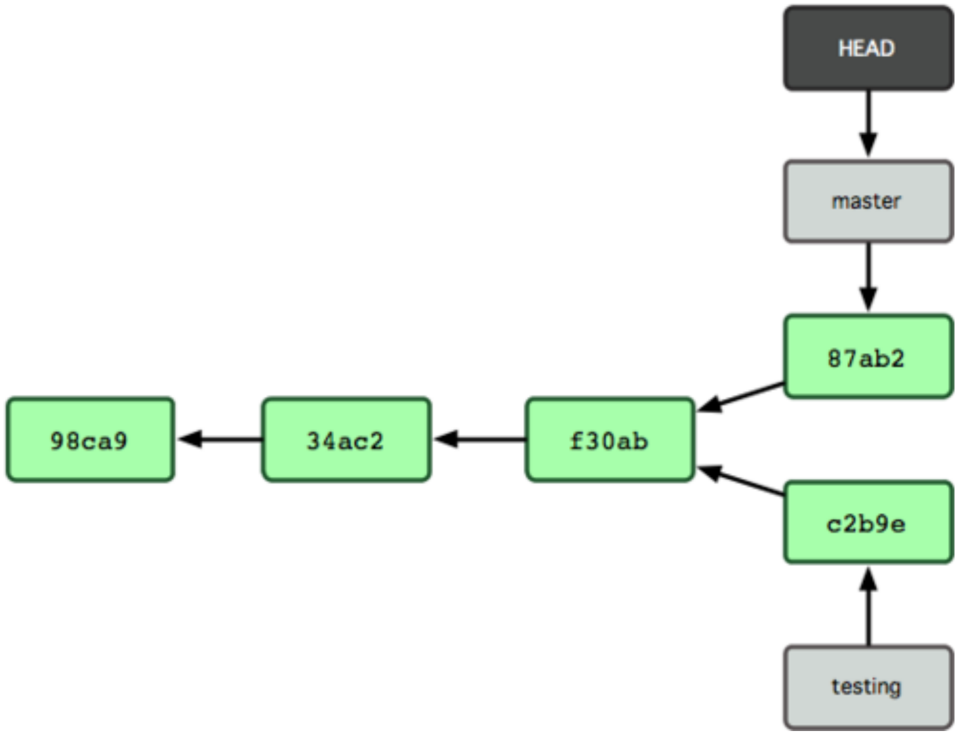
What if you do another commit (now, on the testing branch)? The following figure shows the result:



You can switch back to the master branch by `git checkout master`:

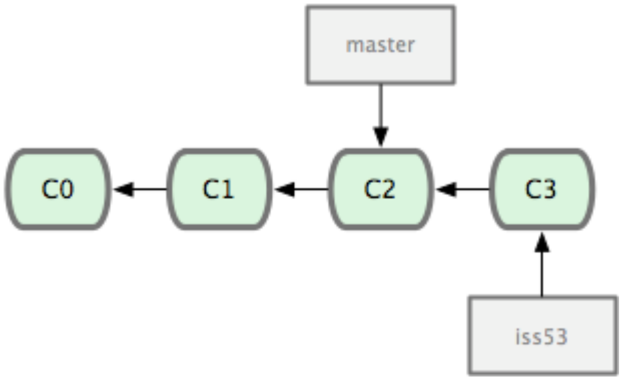


What if you do another commit (now, on the master branch)? The following figure shows the result:

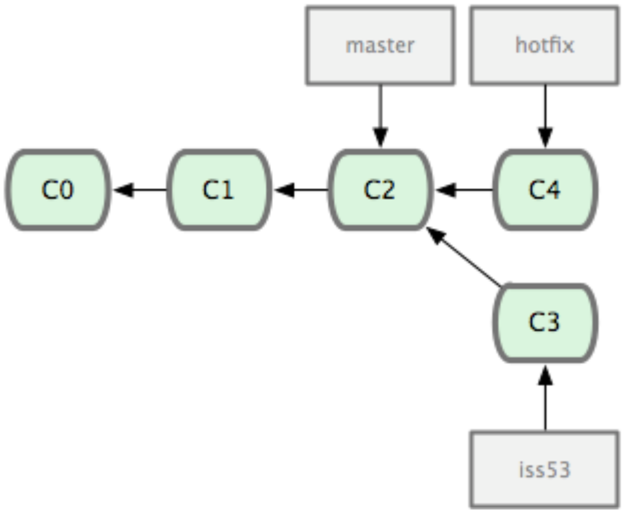


<u>Branches – Merge</u>

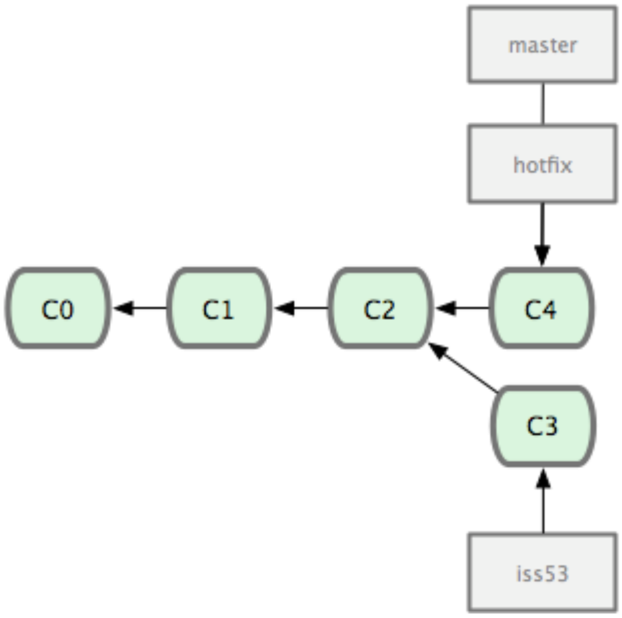
Scenario #1: First, let's say the following is what you have (two branches, `master` and `iss53` – the one that you are currently working on):



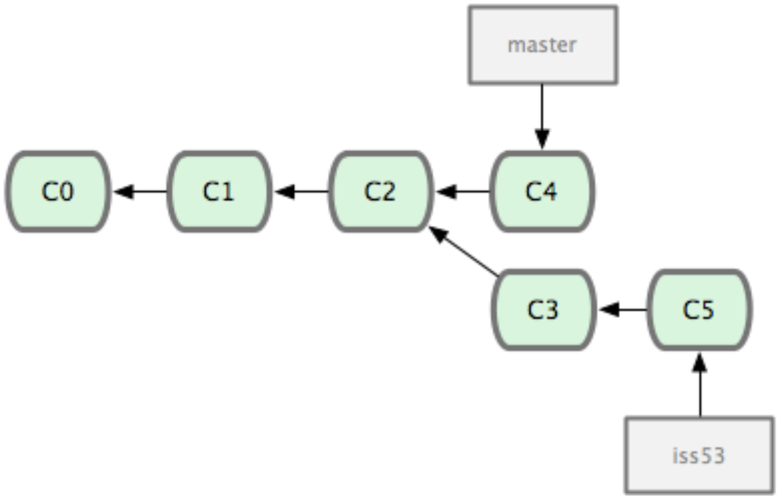
Suddenly, you realize that you need to make a quick fix to the `master` branch. You switch back to `master`, create a new branch `hotfix`, make and commit the changes. The following shows the result:



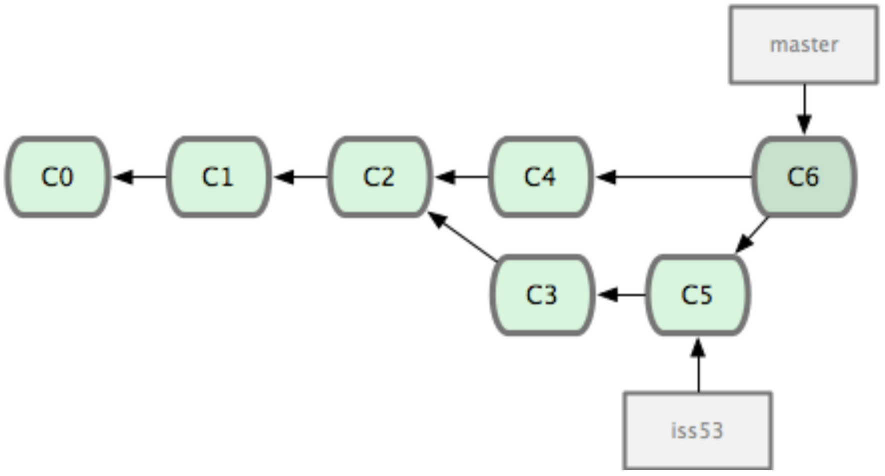
Now, it's time to merge changes on hotfix (C4) into master (`git checkout master` then `git merge hotfix`). Since C4 is a direct upstream of C2, Git simply "Fast Forwards" the HEAD pointer to C4:



Scenario #2: What if the commit on the branch you're on (e.g. C4 in the following picture) isn't a direct ancestor of the branch you are merging in (C5)?



Suppose you do `git checkout master` and then `git merge iss53`, Git will have to do a three-way merging:



Resolve Merge Conflicts

Occasionally, if you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly. You will get something like this:

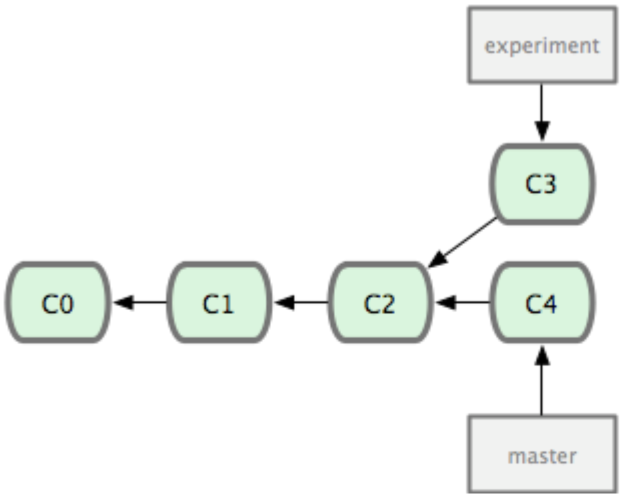
```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

You are now obligated to resolve **ALL** conflicts by modifying the files involved one by one, stage them (`git add -A`), and finalize the merge by committing the changes (`git merge --continue`).

Branches - Rebase

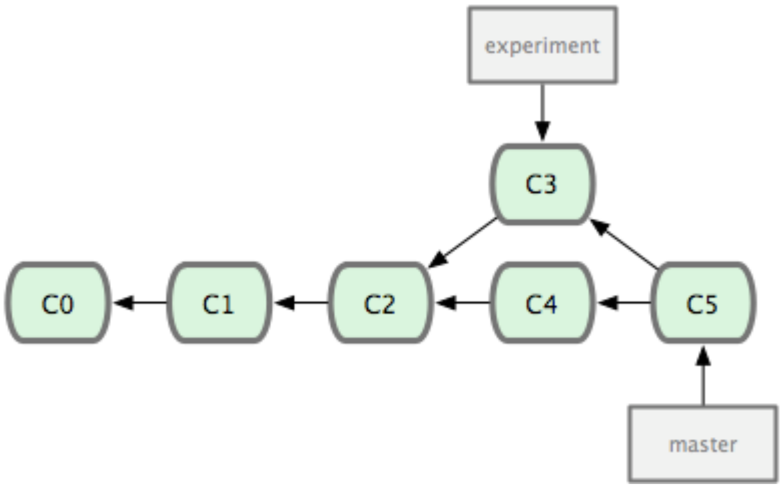
Rebasing is another way to integrate changes from one branch into another. There are major differences between merging and rebasing so misuse of them will result in messy consequences.

Take the following picture as an example (let's call it Fig.0):



Approach 1:

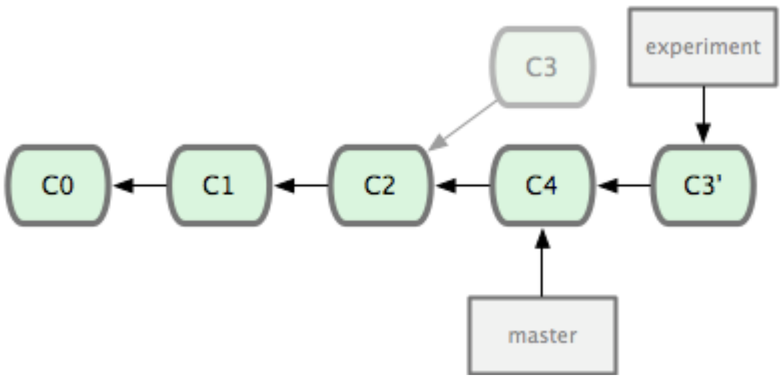
If you do `git checkout master` then `git merge experiment`, you will get this (let's call it Fig.1):



You are now obligated to resolve **ALL** conflicts by modifying the files involved one by one, stage them (`git add -A`), and finalize the merge by committing the changes (`git merge –continue`).

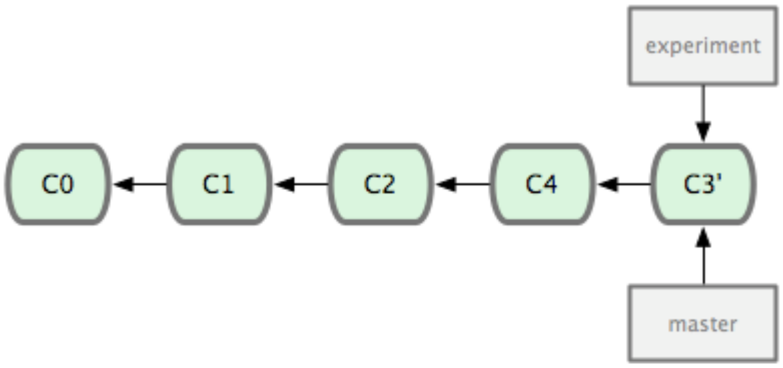
Approach 2:

But from Fig.0, if you do `git checkout experiment` then `git rebase master`, you will get this (let's call it Fig.2):



Notice that just like merging, conflicts may occur when rebasing. You need to resolve **ALL** of them before proceeding to do anything else. I.e. modify the files involved in the conflicts one by one, stage them (`git add -A`), and finalize the rebase by committing the changes (`git rebase –continue`).

Then from Fig.2, you could do `git checkout master` then `git merge experiment`, you will get this (let's call it Fig.3, which is a fast forward of the master pointer):



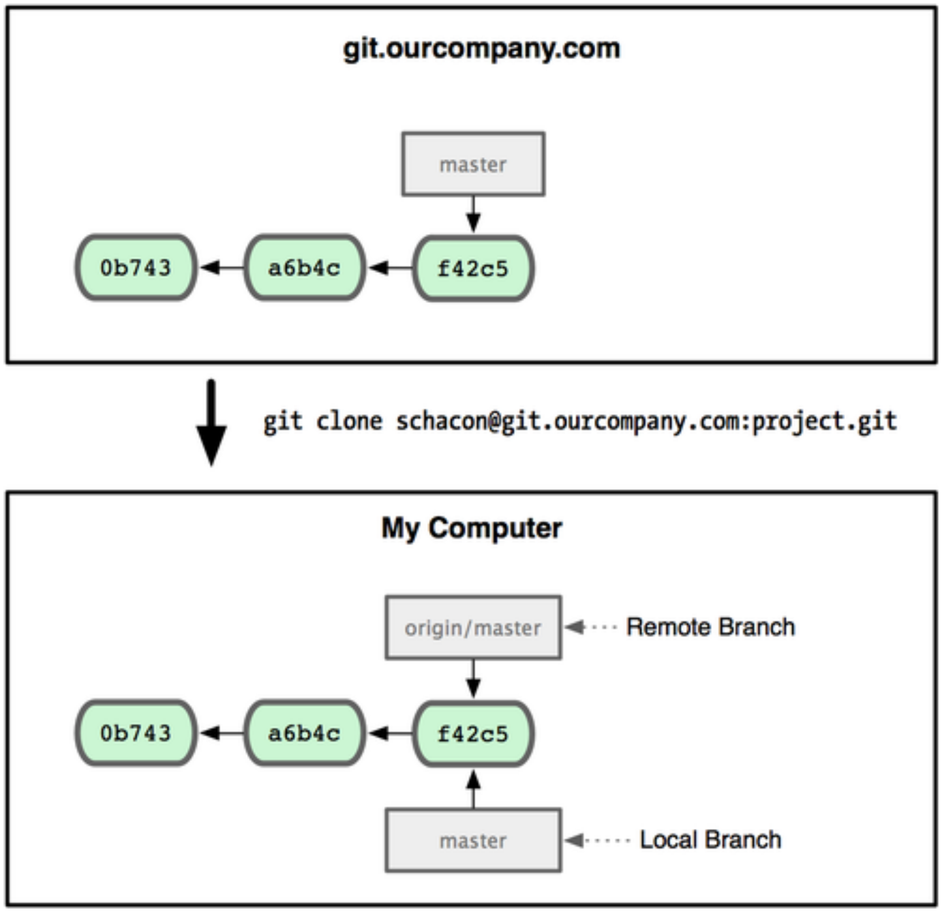
The end product Fig.3 is the same as Fig.1 as far as the final contents of the master branch is concerned, i.e., after C2, all changes in C3 and C4 are captured in the final snapshot, but rebasing makes a cleaner history.

Introduction to git (Part II – Working with remotes)

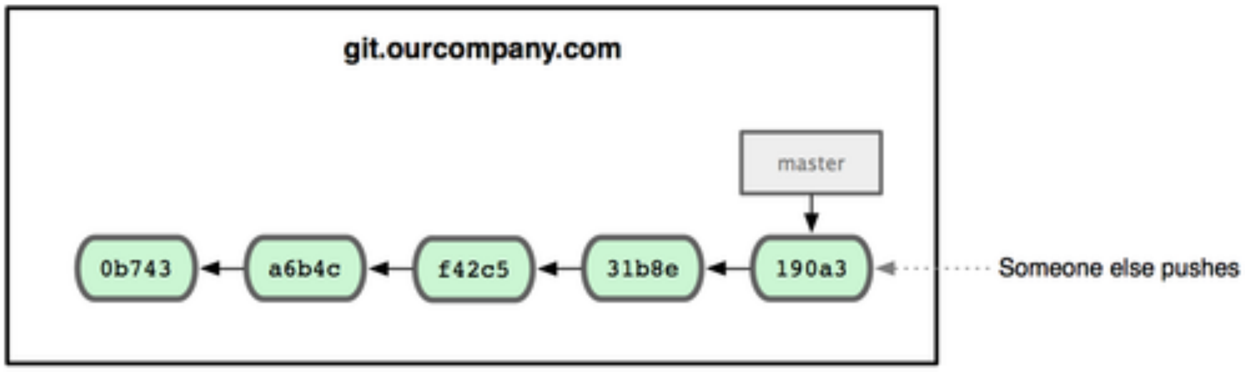
For your individual projects, how can you be able to work from different computers? For your team projects, how can you keep in synch with everyone else's changes and make sure they have yours? Well, you need to know how to manage your remote repositories. In this class, we will use http://github.com.

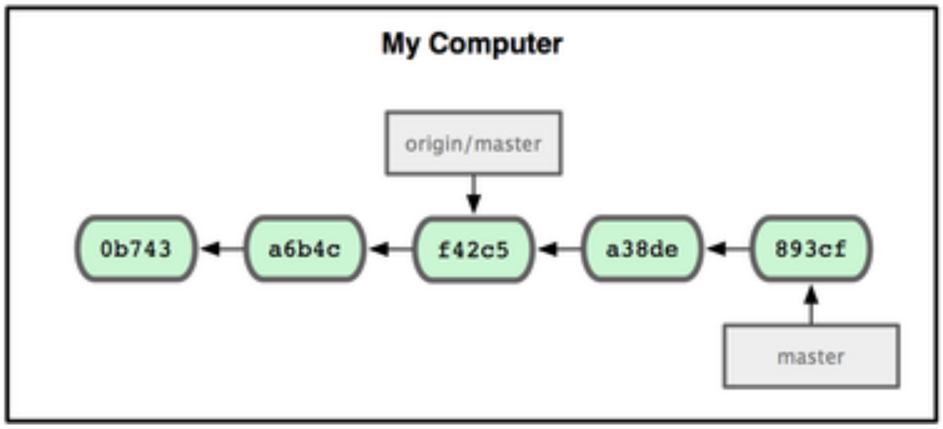
Clone, fetch, and pull

E,g, let's say you have a Git server `git.ourcompany.com`. If you `clone` from this, Git automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally; and you can't move it. Git also gives you your own `master` branch starting at the same place as origin's `master` branch, so you have something to work from:

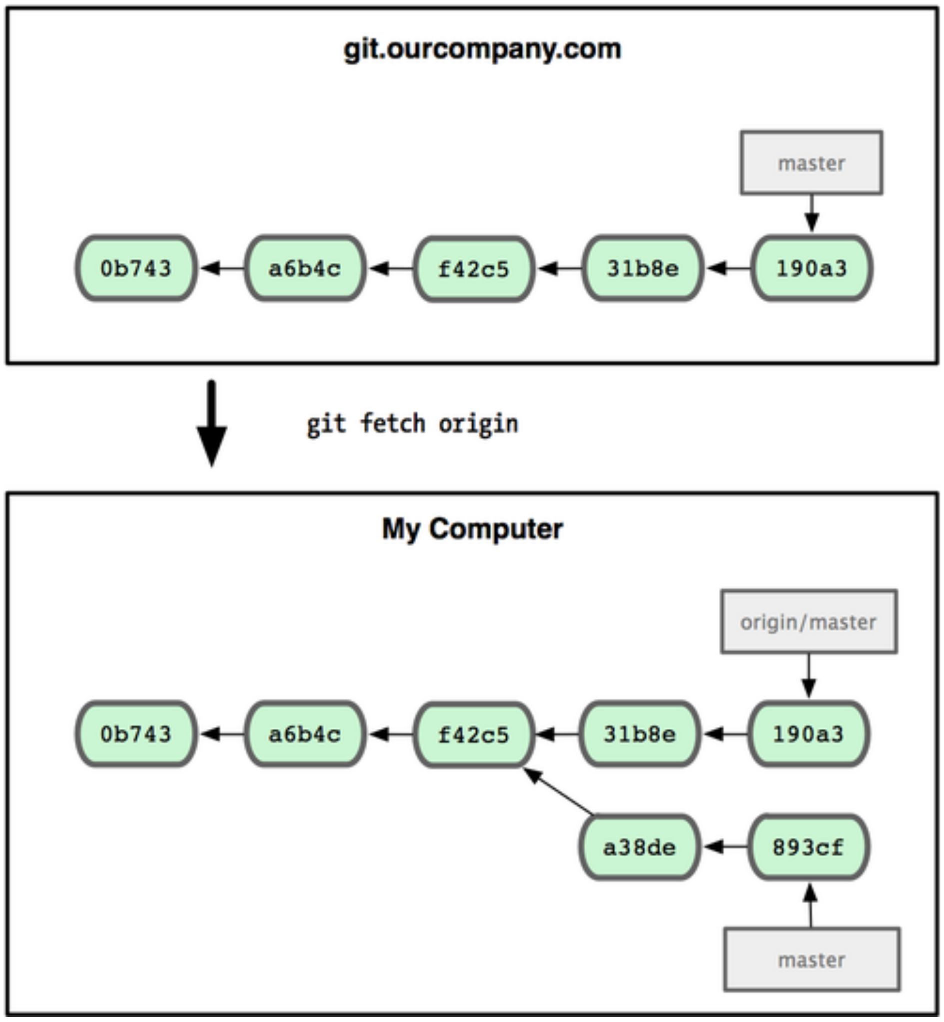


If you do some work on your local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your `origin/master` pointer doesn't move:





To synchronize your work, you run a `git fetch origin` command. The result is show below:



Eventually, you want to merge these two histories into one. You can do so by running `git checkout master` and then `git merge origin/master`. <u>To simplify it, you may do `git pull origin` to do both fetching and merging in one step.</u>

<u>Push</u>

When you want to share a branch with the team, you need to push it up to a remote that you have write access to. Your local branches aren't automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to upload. The command you use has the syntax the `git push [remotename] [localbranch]:[remotebranch]`. Notice that if you drop the `localbranch` part of the command, you effectively delete the `remotebranch` on the remote server, <u>so be very careful</u>.

Git work flow

1. Create a new project folder if you are starting from scratch. Otherwise, go to step 2.
   Enter the project folder.
   ```
   git init
   git add -A
   git commit -m "created a new project"
   git remote add origin git@github.com:user_name/some_project.git
   ```
   Once again, these git commands above must be issued **inside** your project folder.
   Go to step 3.
2. Clone an existing repository from the remote server which defaults to `origin`. Both the remote and local branches default to `master`:
   ```
   git clone git@github.com:user_name/some_project.git
   ```
   To clone a specific branch e.g. `Test_Branch`,
   ```
   git clone -b Test_Branch git@github.com:user_name/some_project.git
   ```
3. Create a new branch to work on a new feature:
   ```
   git checkout -b new_feature
   ```
   … work on new feature…
   ```
   git add -A
   git commit -m "commit comments…"
   ```
   … work more on this feature…
   ```
   git add -A
   git commit -m "commit comments…"
   ```
   … work more on this feature…
   ```
   git add -A
   git commit -m "commit comments… Done with this new feature"
   ```
   …
   When working on training projects where you are the sole contributor, go to step 6.
4. You have completed the new feature and are ready to merge it into the master branch. But before doing that, first, you need to check if there are other new changes available on the remote server (probably pushed there by your teammates while you were working on your part). This step could be skipped if you are working on an individual project.

   ```
   git checkout master
   ```
   (If this is a new project, i.e. you came here from steps 1 and 3, do this:
   ```
   git remote add origin git@github.com:user_name/some_project.git)
   ```

   ```
   git pull origin HEAD
   ```
   … you might need to resolve conflicts here…

5.
   ```
   git checkout new_feature
   git rebase master
   ```
   … you might need to resolve conflicts here…

6.
   ```
   git checkout master
   git merge new_feature
   ```
   If you are done with all features, continue to Step 7; otherwise, go back to Step 3.
7. Push to the remote server
   ```
   git push --force origin master:master
   ```

Please follow the Git work flow above when developing your project(s). A few last things to mention though –

- You may create a local branch from a specific commit.
  First, list all commits so far:
  ```
  git log
  ```
  Then find the commit from which you want to continue to work on, e.g. 349d39…
  ```
  git checkout 349d39
  git checkout –b my_new_branch
  ```

- In order to prevent git from keeping track of files such as `log`, `swp`, `tmp` unnecessarily, put the following lines in the `.gitignore` file in your project directory:

  ```
  # Ignore bundler config
  /.bundle


  # Ignore the default SQLite database.
  /db/*.sqlite3
  /db/*.sqlite3-journal


  # Ignore all logfiles and tempfiles.
  /log/*.log
  /tmp


  # Ignore other unneeded files.
  database.yml
  doc/
  *.swp
  *~
  .project
  .DS_Store
  .idea
  .secret
  ```

- In order to include the current branch name as part of your command line prompt, add the following lines to the `~/.bash_profile` in your home directory (make sure the `git branch` command is all on one line) and open a new terminal window:

```
# Git branch in prompt.
parse_git_branch() {
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\w\[\033[32m\]\$(parse_git_branch)\[\033[00m\] $ "
```

- Never switch to another branch before staging and committing all the changes you've made to the current branch.
  If you are in the middle of making some changes (say on branch A) and are not ready to commit them, do the following:
  ```
  git add -A
  git stash
  ```
  switch to another branch (say branch B), do whatever your want there, and switch back to branch A, then
  ```
  git stash pop
  ```
  Now, you may continue to work on your job.

- Never rebase the master branch, the one that you use to push/pull commits to/from a remote repo, or you will give everyone in your team a headache down the road. Instead, rebase the local branch (step 5 above) then switch to the master branch and merge (step 6 above).