GIT

Pau Fernández Rafa Genés Jose L. Muñoz Tapia
Universitat Politècnica de Catalunya (UPC)

Git

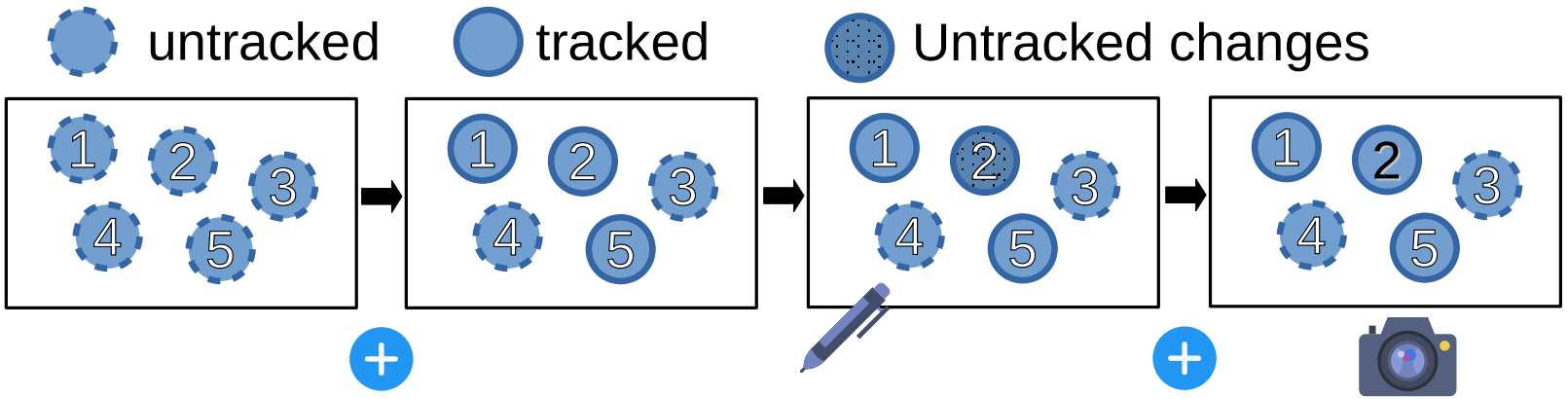
Outline

Naive Version Control

- Version controls can be used for versioning source code files, documentation or with any type of file on a computer.
- Many people's version-control method of choice is:
  - Copy files into another directory (perhaps a time-stamped directory).
  - This approach is very common because it is so simple.
  - But it is also incredibly error prone.
  - It is easy to forget which directory you're in.
  - Or to accidentally write to the wrong file or copy over files you don't mean to.

Version Control System

- A version control system (VCS) is a system that records changes of a set of files over the time.
- Is like ”taking photos”:
  - Need to say who is going to be in the photo **(add)**.
  - Photos will be stored **(commit)**.
  - VCS try hard not to loose any photo.
  - We can see photos later, see differences between photos, name them and so on.



VCS Features

A version control allows us to:

- Revert files back to a previous state.
- Revert the entire project back to a previous state Compare changes over time.
- See who last modified something that might be causing a problem.
- Who introduced an issue, when and more.
- Using a VCS also generally means that if you screw things up or lose files, you can easily recover.

Local VCS

To manage versions programmers long ago developed **local version controls**:

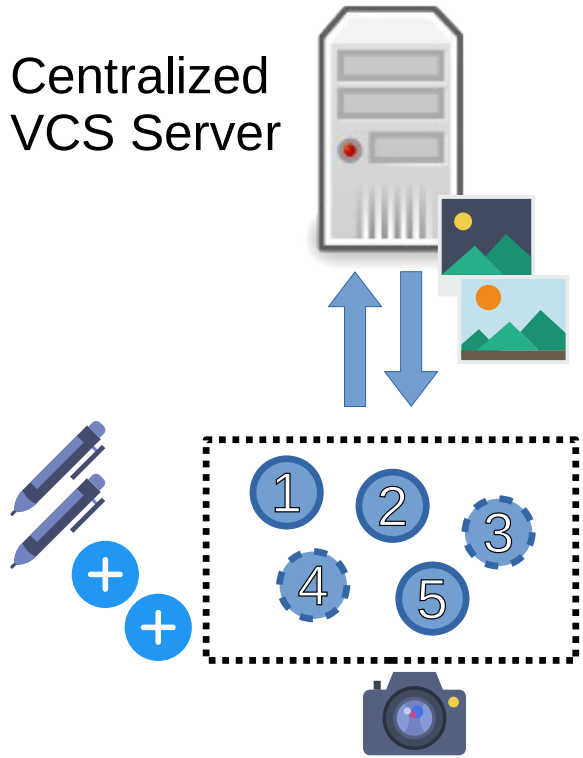
- These had a simple database that kept all the changes to files under revision control.
- One of the more popular VCS tools was a system called RCS, which is still distributed with some computers today.
- RCS works by keeping patch sets (that is, the differences between files) in a special format on disk.
- It can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized VCS i

The problem with local VCS is that there is a need for **collaboration** with developers on other systems.

To deal with this problem, **centralized VCS** were developed:

- These systems have a single server that contains all the versioned files.
- A number of clients check out files from that central place.
- For many years, this has been the standard for version control.



Centralized VCS ii

A centralized VCS has many advantages over a local VCS:

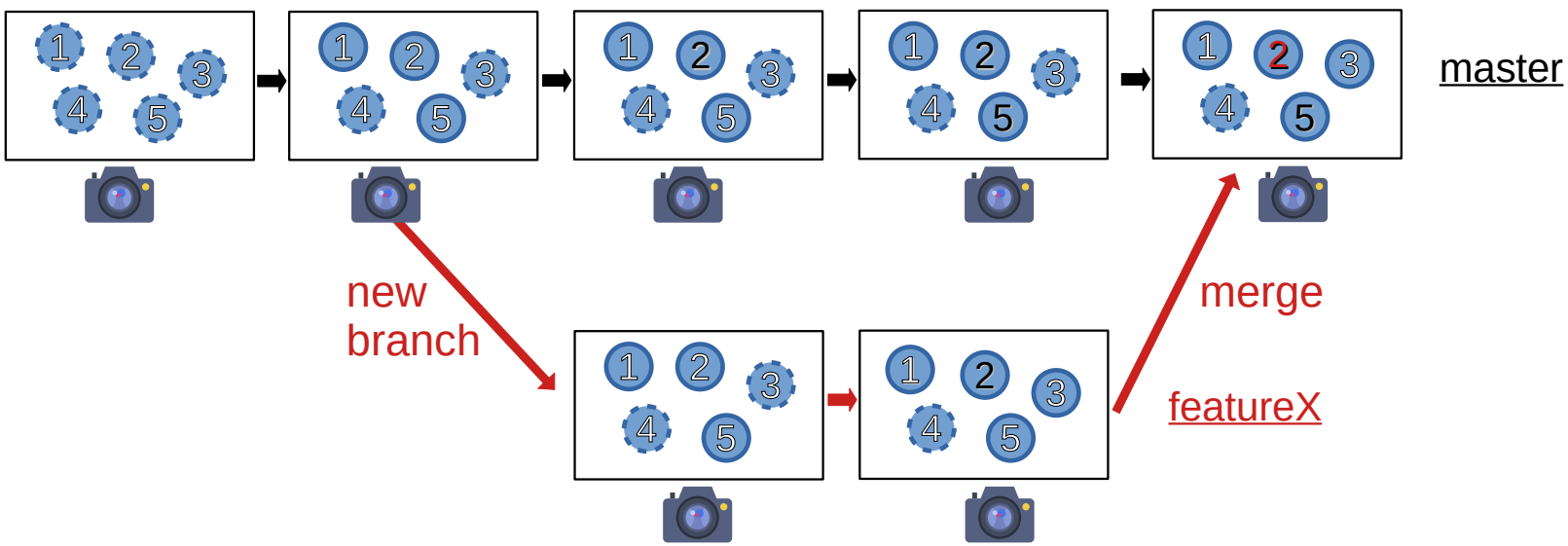
- Everyone knows to a certain degree what everyone else on the project is doing (commits).
- Administrators have fine-grained control over who can do what.
- It's far easier to administer a centralized system than it is to deal with local databases on every client.
- The most famous centralized VCS is SVN (subversion).

Distributed VCS

- Main drawbacks of centralized VCS:
  - Single point of failure.
  - Cannot write history (commits) while offline.
- Git is a **distributed Version Control System**.
  - Each user has a full mirror of the repository.
  - Clients have a local “repository“ and they may also use one or several remote repositories.
  - In the local repository, users can do commits that they can later, if desired, upload to remote repositories.
  - If any server dies, any of the client repositories can be copied back up to the server to restore it.
  - The most famous distributed VCS is GIT.

Branches

- Nearly every VCS has some form of branching support.
- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- This is an important feature that in many VCS is costly but that in GIT is agile and easy.



Outline

Install & Config

- Install in Debian-based systems:

```
$ sudo apt install git
```

- Set your user name and e-mail address:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

- We can change other settings:

```
$ git config --global core.editor "vim"
```

- Check your settings:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
...
```

How to Get a Git Repository

You can get a Git project using two main approaches:

1. Take the directory of your project and import it into Git.

```
$ mkdir myrepo
$ cd myrepo
myrepo$ git init
myrepo$ ls -a
```

2. Clone an existing Git repository:

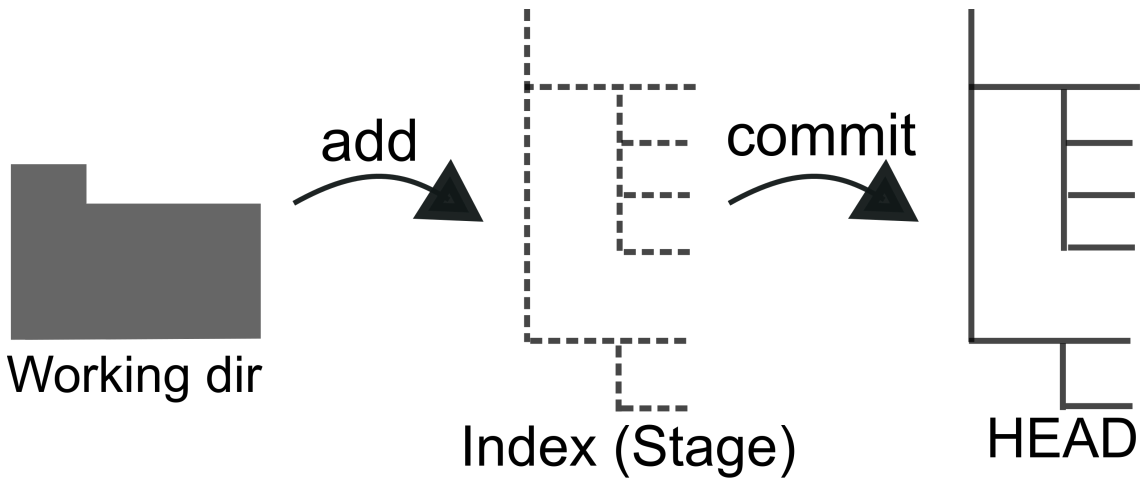
```
$ git clone https://github.com/mylib/mylib mylib
```

Git has a number of different transfer protocols including HTTPS, SSH and the local filesystem.

Outline

Areas

- The local repository is organized in three areas maintained by GIT[1]:
  1. **Working directory**. This area holds the current files of your working directory (tracked and untracked by git).
  2. **Index**. This area acts as a ”staging“ area before commit.
  3. **HEAD**. This area points to the last local commit done.



[1] There is yet another area called stash.

Track Files i

- Add files to be tracked:

```
$ git add <filename>
```

- Creates a snapshot of the specified files **of the working copy at the Index**.
- The content at the Index is said **to be staged for the next commit**.
- By default, the `git add` command adds files recursively.

```
myrepo$ touch file1 file2 readme.txt
myrepo$ mkdir files
myrepo$ touch files/f1
myrepo$ touch files/hello.txt
myrepo$ git add 'f*' # single quote allows file expansion by git
```

Track Files ii

- We can observe the status of the files of our repository:

```
myrepo$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   file1
  new file:   file2
  new file:   files/f1
  new file:   files/hello.txt
Untracked files:
 (use "git add <file>..." include what to be committed)
  readme.txt
```

- Each time you modify the contents of a file, you should execute **`git add`** to add the changes to the staging area.

Unstage Files

- Unstage file1:

```
myrepo$ git rm --cached file1
rm 'file1'
myrepo$ git status
...
Untracked files:
  (use "git add <file>..." to include in what
      will be committed)
  file1
  readme.txt
```

- Without `--cached` Git will remove the file from being tracked and also from your working directory.

More on Staging Files

We have several options to stage (add) files[2]:

Command	New Files	Modified Files	Deleted Files	Description
git add -u	-	✓	✓	Stage modified and deleted files only
git add --ignore-removal .	✓	✓	-	Stage new and modified files only
git add .	✓	✓	✓	Stage all (new, modified and deleted files)
git add -A	✓	✓	✓	Stage all (new, modified and deleted files)

[2] This is for git 2.x version, old 1.x had other behavior.

.gitignore

- You can use the file .gitignore to avoid addding undesired files.
- Examples:

```
node_modules/
*.zip
*.gzip
*.log
**/logs
```

- It's not a good idea in general to version compressed files (better unpacked)
- Double asterisk matches directories anywhere in the repository:

`build/logs/debug.log` `logs/debug.log` `logs/monday/foo.bar`

Commit

- The changes are not really in the Git mini-filesystem until we commit them, Index (staging area) -> HEAD:

```
myrepo$ git commit -m "my first commit"
[master (root-commit) 98c1d8a] my first commit
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2
 create mode 100644 files/f1
 create mode 100644 files/hello.txt
```

- Each commit has a unique identifier (a SHA-1 value), the first numbers are ”98c1d8a”.
- We can see the complete identifier viewing the log:

```
myrepo$ git log
commit 98c1d8a7ac4eeddc91efabc1251549bfc32012bc
Author: jlmunoz <you@example.com>
Date:   Mon Jul 27 03:52:26 2015 +0200
    my first commit
```

File States

- Versioned (tracked) files can be in three states:
  1. **Unmodified**. When a file is the same at your working directory, the Index and the HEAD it is said to be unmodified.
  2. **Modified**. When a file is modified in the working directory, its contents differ from the contents at Index and it is said to be modified.
  3. **Staged**. When a modified file is staged it means that it has been added to the Index and thus, it is prepared for the next commit.

File States in Practice

- Let's observe the status of our repository:

```
myrepo$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  file1
  readme.txt
nothing added to commit but untracked files present(use "git add" to track)
```

- The files that we previously commit are now unmodified.
- Unmodified files are not shown by the status command.
- Now, we modify a file, e.g. file2 and show the status:

```
myrepo$ echo hello > file2
myrepo$ 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:   file2
Untracked files:
...
```

- Notice that a file can be listed as both staged and unstaged.

Automatically Stage

- In the normal workflow:
  1. We should add to the Index.
  2. We make the commit.
- However, with `-a` we tell commit to automatically stage files that have been modified and deleted (like `git add -u`).
- New files you have not told Git about are not affected:

```
myrepo$ git commit -am "third commit"
[master afe80ec] third commit
 2 files changed, 2 insertions(+), 1 deletion(-)
```

- The previous option is basically a way of **not using the Index tree in our workflow**.

Checkout: Going Back and Forth

- We can go to a previous commit with the following command:

```
git checkout <hash_commit>
```

- Example[3]:

```
myrepo$ git --no-pager log --oneline
9b79078 (HEAD -> master) second commit
4c715c9 first commit

myrepo$ git checkout 4c715
```

- To go back to the last commit:

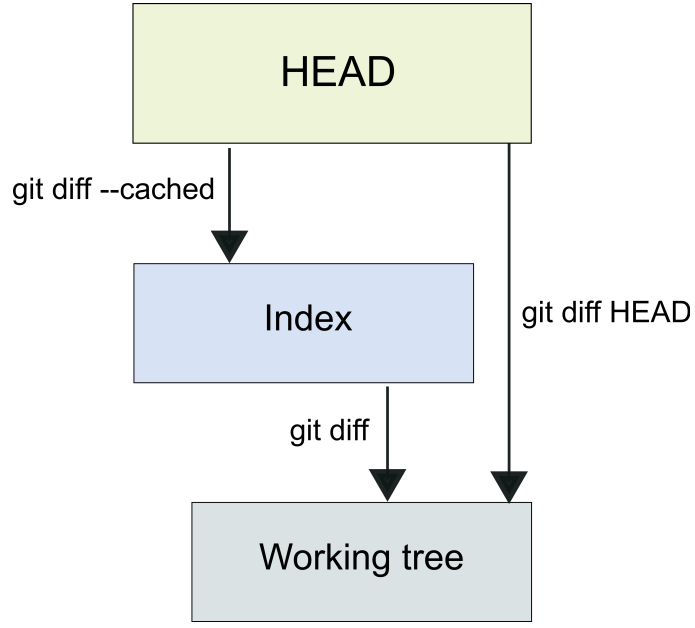
```
$ git checkout master
```

- This is a way to view differences (but painful).

[3] As you might observe, to name a commit, we only need to provide to git the beginning of the commit hash that makes it indistinguishable from other commits.

Viewing Differences

- `git log` shows which files were changed.
- `git diff` shows exactly what changed.



- Appending `-- filename` to the previous commands allows you to view the differences only of a particular file.
- You can view differences between commits with:

```
$ git diff <hash_commit1> <hash_commit2>
$ git diff <hash_commit1> <hash_commit2> filename
```

Graphical Diff Tool & Blame

- In Git, you can configure a graphical diff tool.
- This tool can be configured globally with the following command:

```
$ git config --global diff.tool meld
```

- The previous command configures Git to use **`meld`** as the graphical difftool.
- Then, if you want a graphical diff instead of a textual one, you simply invoke **`git difftool`** instead of **`git diff`**:

```
$ git difftool HEAD -- somefile.py
```

- To know who was the last who edited a line:

```
$ git blame file
```

Outline

Remote Repositories

- You need to know how to manage remote repositories to **collaborate**.
- You add remotes to be able to fetch objects from these remotes (blobs, trees, commits and tags).
- Once you have these objects in your local repository:
  1. You can make changes
  2. Merge your work locally
  3. Upload these changes back to the remote.

Bare and Non-bare Repositories

- Git repositories are bare or non-bare:
  - A **non-bare repository** has a bunch of working files (the tree), and a hidden directory (.git) containing the version control information.
  - A **bare repository** in Git just contains the version control information and no working files, it just does what the ”server” notionally does in a centralized VCS.
- In Git, as a general rule **the repository has to be bare (no working files) in order to accept a push**[4].
- To create a bare repository (it is typical to end the directory with .git for bare repos):

```
$ mkdir myremoterepo.git
$ cd myremoterepo.git
myremoterepo.git$ git init --bare
```

[4] Actually, you can’t push to the currently checked out branch of a repository. With a bare repository, you can push to any branch (none are checked out). Although possible, pushing to non-bare repositories is not common.

Create a Remote

- Let's connect our `myrepo` to the new (bare) `myremoterepo.git`.
- To do so, you have to add the `myremoterepo` as a remote in `myrepo`:

```
git remote add NAME PATH_TO_REPO
```

- For example, to add a remote called "origin" on a directory of the same filesystem:

```
myrepo$ git remote add origin ../myremoterepo.git
```

- We created a remote called "origin", now, we can try to push our history there.

Push and Track

```
myrepo$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use
    git push --set-upstream origin master
```

- The push fails because we need to tell git in which remote branch we want to push our commits.

```
myrepo$ git push -u origin master
```

- The previous command creates:
  - Creates a master branch with our commits in the remote ”origin”.
  - Tracks the remote branch ”origin/master” linking it to our local branch ”master”.
  - Note. The flag `-u` is the short version of `--set-upstream`.
- Note. You can change the remote branch tracked with:

```
myrepo(master)$ git branch -u anotherorigin/master
```

Clone a Remote

- Let's build our repository as a clone of an existing one:

```
$ git clone myremoterepo.git myrepo2
Cloning into 'myrepo2'...
done.
```

- To see which remote servers we have configured:

```
$ cd myrepo2
myrepo2$ git remote -v
origin  /home/myuser/tmp/myremoterepo.git (fetch)
origin  /home/myuser/tmp/myremoterepo.git (push)
```

- To get more information:

```
myrepo2$ git remote show origin
* remote origin
  Fetch URL: /home/myuser/tmp/myremoterepo.git
  Push  URL: /home/myuser/tmp/myremoterepo.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Fetch from Remotes

- We make changes and commit from `myrepo`.
- Now, we can fetch all the objects from `myrepo2` as follows:

```
myrepo2$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ../myremoterepo
   bf374e0..9f5101b  master     -> origin/master
```

- When you **clone** a repository:
  - Git automatically adds that remote repository under the name "origin".
  - Git fetches all the objects of that remote.

Merging

- We can view the differences between local and remote with:

```
git diff origin/master -- [local-path]
```

```
$ git difftool origin/master -- README.md
```

- We can merge our work with:

```
$ git merge origin master
```

- We can **fetch+merge** with a single command:

```
$ git pull
```

Conflicts

- VCS are very good merging text files:
  - If two people don't touch the same text line merge will proceed without problems.
  - However, we might have conflicts if a line is double edited.
- You can visualize and resolve conflicts with Visual Studio Code.
- When you fix the conflict, save the file and commit as usual:

```
$ git status
$ codium myfile.txt
$ git add myfile.txt
$ git commit -am "conflict fixed"
$ git push
```

- Note. Git (and the other VCS) **cannot merge binary files**, they just replace them.

Pushing to your Remotes

- When you have your project at a point that you want to share, you have to push it upstream:

```
$ git push [origin master]
```

- Note. Cloning generally sets up both of those names (origin and master) for you automatically.
- If there is a previous commit your push will rightly be rejected.
- You'll have to pull down before push.
- There can be conflicts.

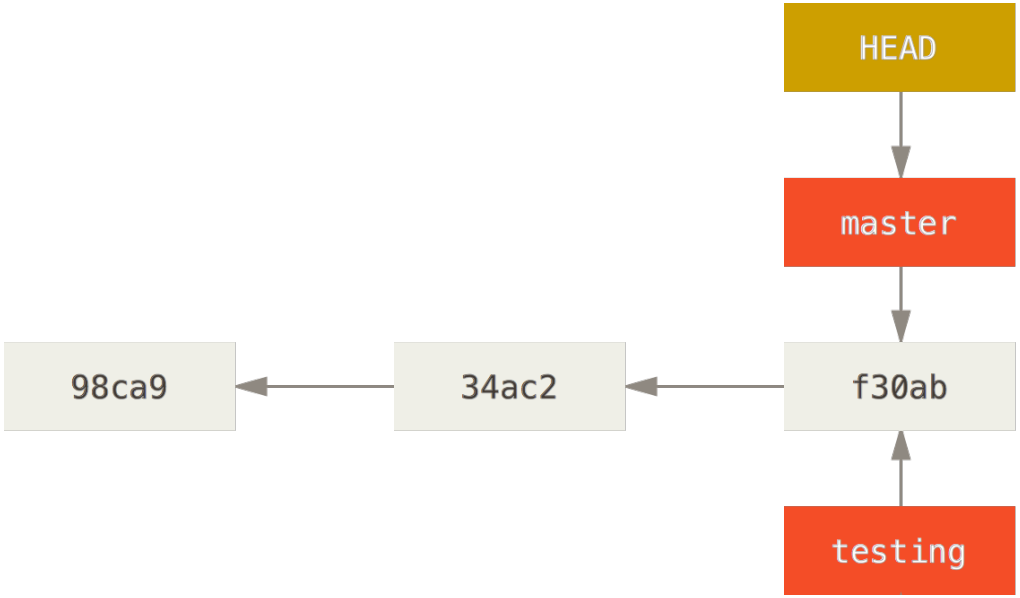
Outline

Creating a New Branch i

- Create a branch:

```
$ git branch testing
```

- When you create a new branch, Git creates a new pointer for you to move around.



- Now, we have two branches pointing into the same series of commits.

Creating a New Branch ii

- You can see the ”master” and ”testing” branches that are right there next to the f30ab commit with:

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

- Git also uses the HEAD to know a what branch are you.
- Nice graphical tools to see the git history are: `gitg` (simple), `gitkraken` or `visual studio code`.

Switching to a Branch i

- We created the branch but we did not switch to that branch.
- To switch to an existing branch:

```
$ git checkout testing
```

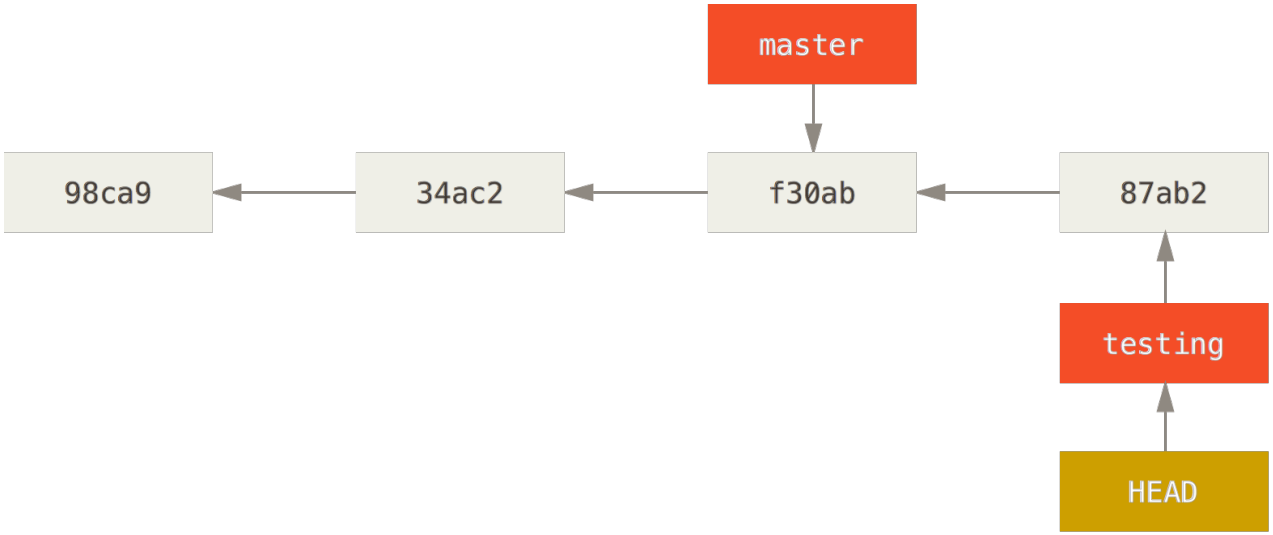
- Previous command moves HEAD to point to the testing branch.
- If you want to create a branch AND checkout you can add `-b`:

```
$ git checkout -b testing2 # This is equivalent to:
$ git branch testing2 ; git checkout testing2
```

- Let's do a commit:

```
$ echo hola > hola.txt
$ git add hola.txt
$ git commit -m 'made a change'
```

Switching to a Branch ii



- The HEAD branch moves forward when a commit is made.
- Now, your testing branch has moved forward, but your master branch still points to the commit you were.

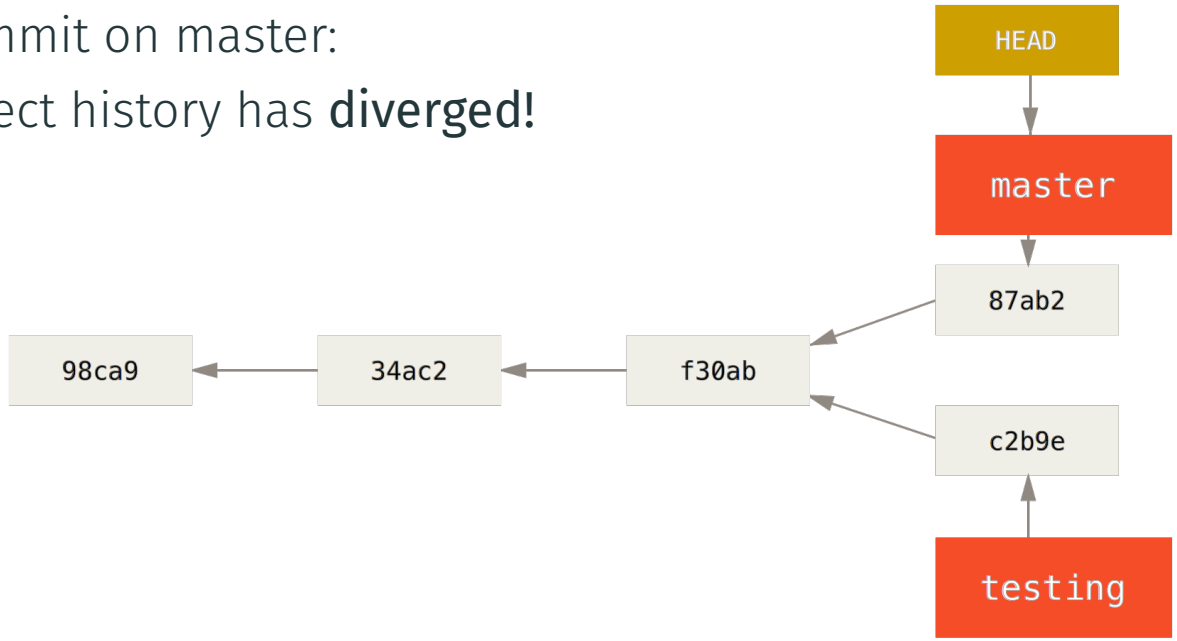
Diverging Branches

- Let's switch back to the master branch:

```
$ git checkout master
```

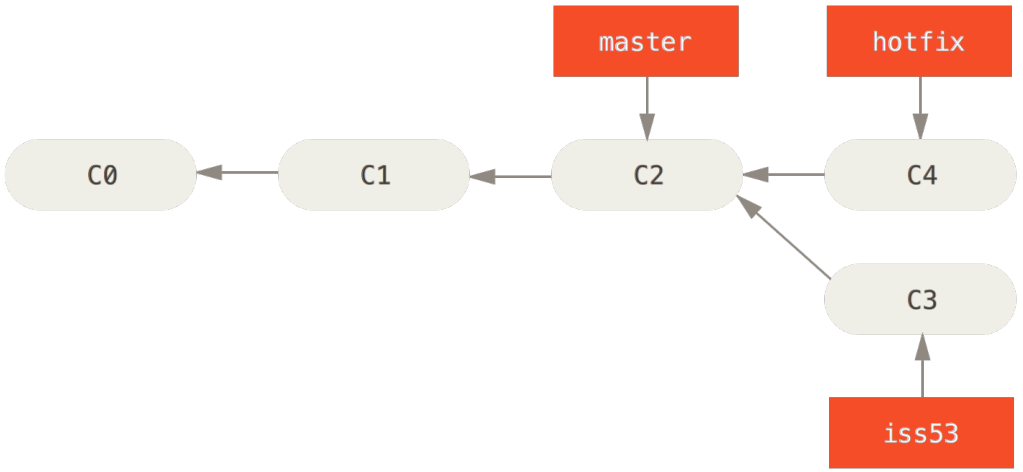
- The previous command did two things:
  1. It moved the HEAD pointer back to point to the master branch.
  2. It reverted the files in your working directory back to the snapshot that master points to.
- If we commit on master:

  Our project history has **diverged!**



Fast Forward Merging

- In the following example we have two branches hotfix and iss53:

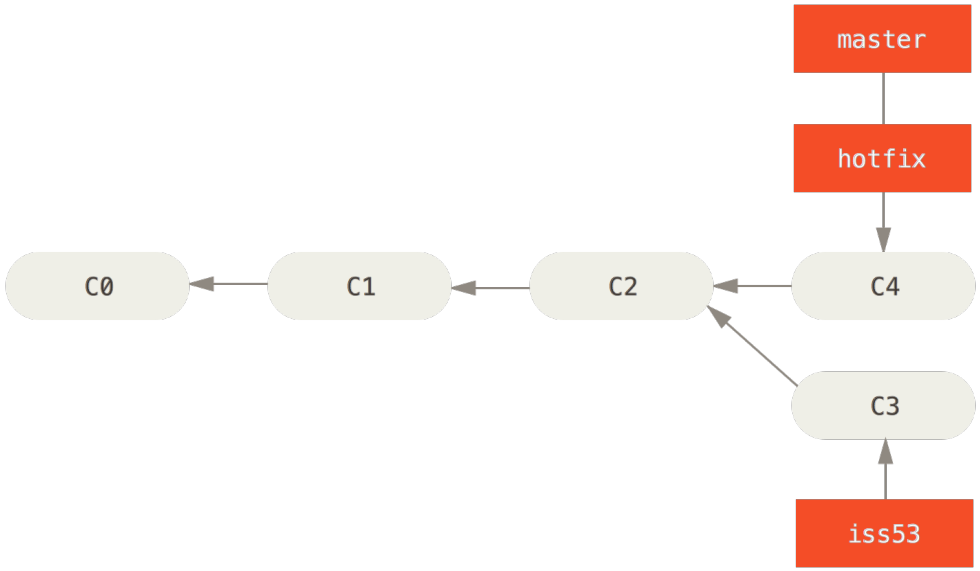


- We merge hotfix into master:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

- You'll notice the phrase **"fast-forward"** in that merge (because Git simply moves the pointer forward).

Deleting a Branch

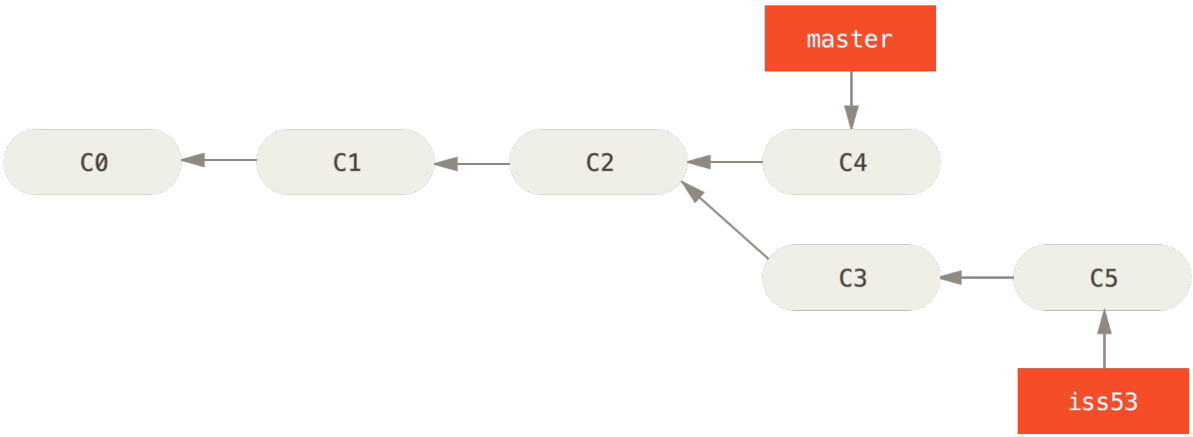


- After your fix is deployed, you're ready to switch back to the work you were doing.
- However, first you'll delete the hotfix branch, because you no longer need it (the master branch points at the same place):

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

About Work on Different Branches

Now, you can switch back to your work-in-progress branch on issue #53 (C5).



- It's worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch.
- If you need to pull it in:
  1. You can merge your master branch into your iss53 branch by running `git merge master`.
  2. You can wait to integrate those changes until you decide to pull the iss53 branch back into master later (this is the typical situation).

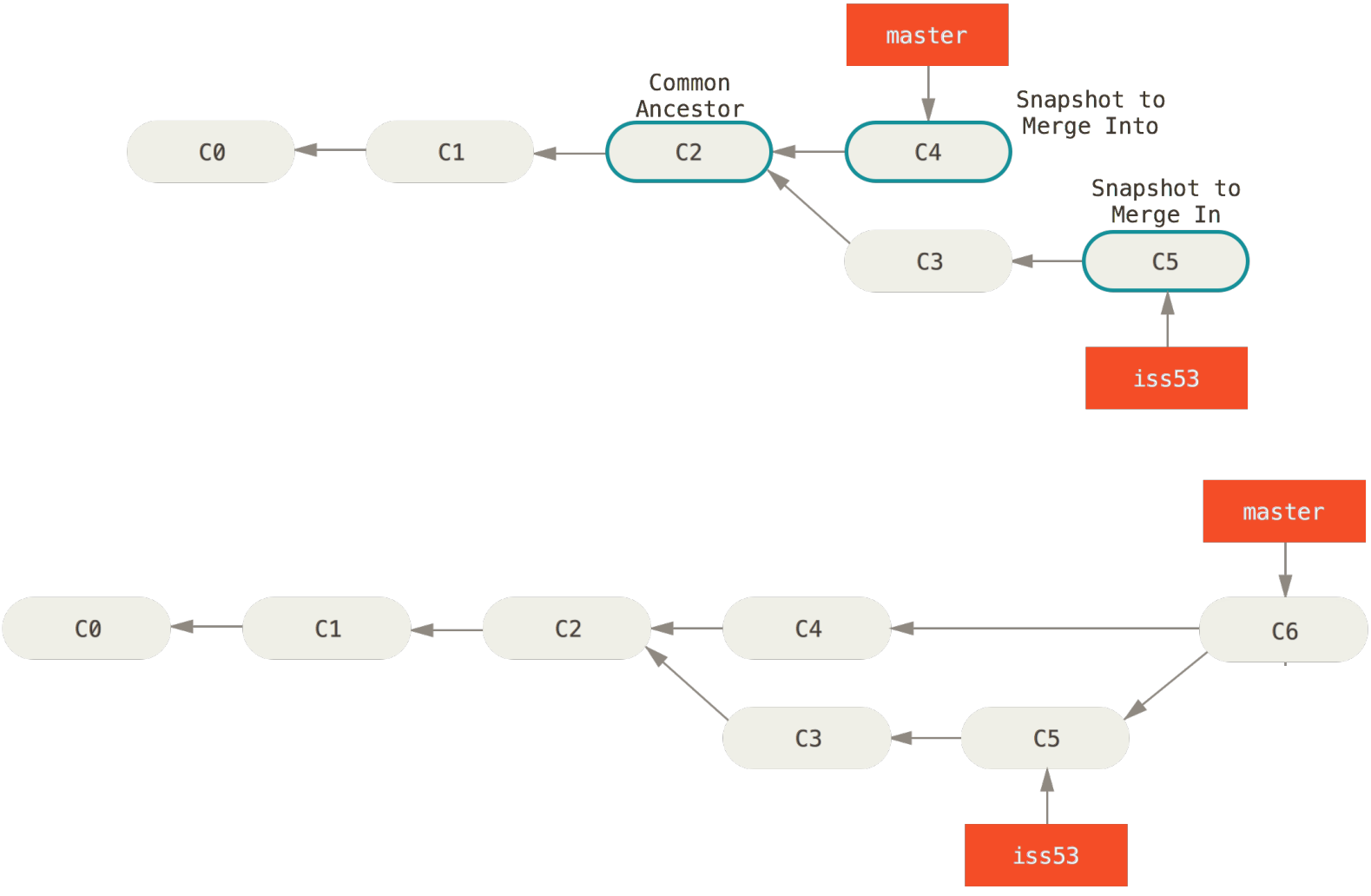
3-Way Merging i

- Suppose you've decided that your issue #53 work is complete and ready to be merged into your master branch:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

- Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some extra work:
  - Git determines the best common ancestor.
  - Calculates and applies the changes from the two branches to the ancestor.
  - This 3-way merge creates a special commit called "merge commit".
  - A merge commit is special in that it has more than one parent.

3-Way Merging ii



Merge Conflicts i

- Occasionally, the merging process doesn't go smoothly.
- If you changed the same part of the same file differently in the two branches you're merging together,
- You'll get a merge conflict that looks 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.
```

- The new merge commit has paused the process while you resolve the conflict.

Merge Conflicts ii

- If you want to see which files are unmerged at any point after a merge conflict:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  (use "git add ..." to mark resolution)
    both modified:      index.html
no changes added to commit (use "git add" and/or "git commit -a")
```

- As usual, to fix the conflict you have to save the correct content, stage your changes, commit and push.

Branch Management i

- To see the last commit on each branch:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

- The previous command shows that you are checked out in ”master”.
- To show branches that are already merged into the branch you’re on:

```
$ git branch --merged
  iss53
* master
```

- Branches in the output of the previous command different from the one you are checked out can be generally deleted.

Branch Management ii

- To show branches that contain work you haven't yet merged in:

```
$ git branch --no-merged
  testing
```

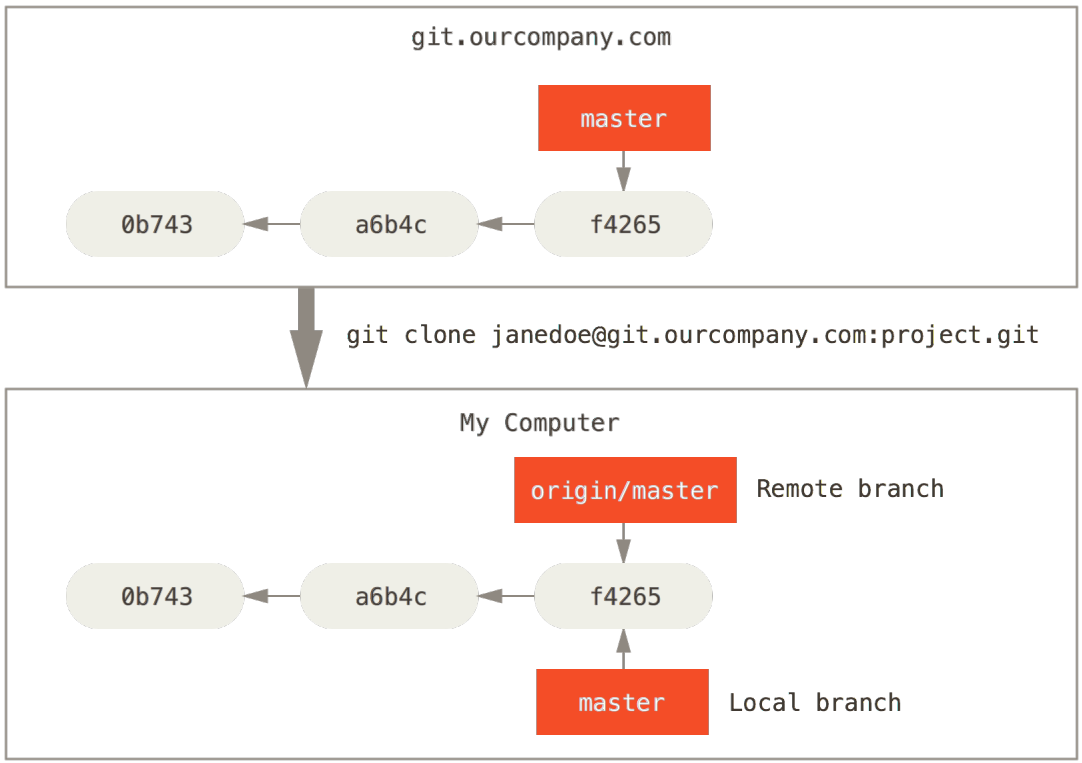
- Because it contains work that isn't merged in yet, trying to delete it with git branch `-d` will fail:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Remote Branches

To list local and remote branches:

```
$ git branch -avv
```



Remote-tracking Branches

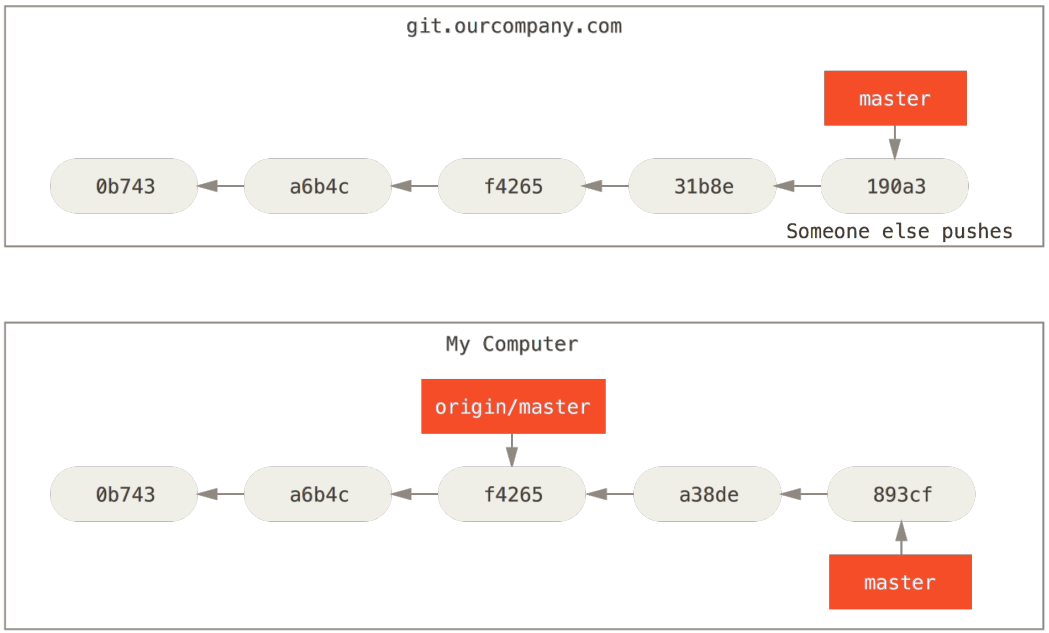
- Remote-tracking branches are references to the state of remote branches:
  - They’re local references that you can’t move.
  - They’re moved automatically for you whenever you do any network communication.
  - Bookmarks to remind you where the branches in your remote repositories were the last time you connected to them.
  - They take the form (remote)/(branch).

Remote Branches and Cloning

- Let's consider you clone git.ourcompany.com:
  - Git's clone command automatically names it origin for you.
  - It pulls down all its data.
  - It creates a pointer to where its master branch is.
  - It names it origin/master locally.
  - Git also gives you your own local master branch starting at the same place as origin's master branch, so you have something to work from.

Remote and Local Branches Diverge

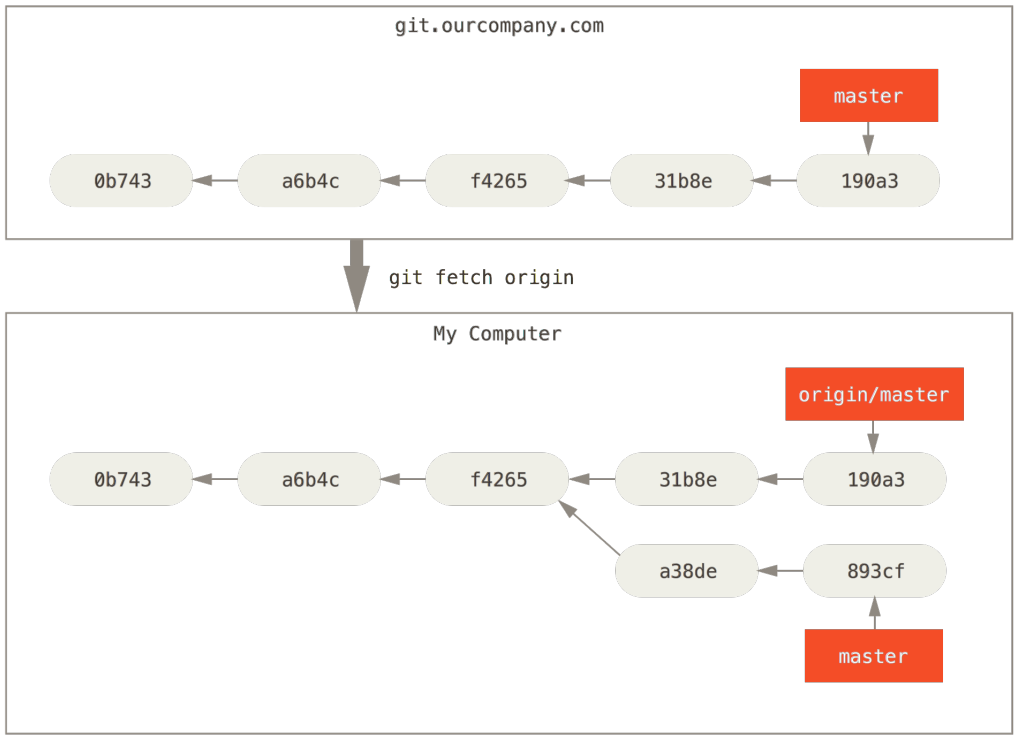
- If you do some work on your local master branch branches will diverge:



Synchronizing (`fetch`)

```
$ git fetch origin
```

The previous comamnd fetches any data from it that you don't yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date.



Publishing (push)

- When you want to share a branch with the world, you need to push it up to a remote that you have write access to:
  - Your local branches aren't automatically synchronized with remotes.
  - You have to explicitly push the branches you want to share:

```
$ git push origin serverfix
  ...
 * [new branch]      serverfix -> serverfix
```

  - If you do not want to call your branch serverfix on the remote, you can use the syntax:

```
$ git push origin serverfix:awesomebranch
```

  - If you want to push all your branches:

```
$ git push origin --all
```

Tracking Branches i

- When you do a fetch that brings down new remote-tracking branches, you don't automatically have local, editable copies of them:
  - In this case, you don't have a new serverfix branch but you only have an origin/serverfix pointer that you can't modify.
  - You can execute `git merge origin/serverfix` to merge this work into your current working branch.
- If you want your own local branch based on the remote branch:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

- This gives you a local branch that you can work on that starts where origin/serverfix is.
- Note. After `-b` you can use a different name for the local branch.
- Checking out a local branch from a remote branch automatically creates what is called a **"tracking branch"** (AKA "upstream branch").

Tracking Branches ii

- An equivalent command for tracking a branch is:

```
$ git checkout --track origin/serverfix
```

- Tracking branches are local branches that have a direct relationship to a remote branch:
  - If you're on a tracking branch and type for example **`git fetch`**, Git automatically knows which server to fetch from.
  - When you clone a repository, it generally automatically creates a master branch that tracks origin/master.
  - However, you can set up other tracking branches if you wish.
- You can use the option **`-u`** to change the upstream branch you're tracking:

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

Viewing Tracked Branches

```
$ git branch -avv
  iss53     7e424c3 [origin/iss53: ahead 2] working with functions
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] ..
  testing   5ea463a trying something new
```

- Here we can see that our iss53 branch is tracking origin/iss53 and is "ahead" by two, meaning that we have two commits locally that are not pushed to the server.
- It's important to note that:
  - These numbers are only since the last time you fetched from each server.
  - This command does not reach out to the servers, it's telling you about what it has cached from these servers locally.
- If you want totally up to date ahead and behind numbers:

```
$ git fetch --all
$ git branch -vv
```

Pulling

- Fetch does not modify your working directory at all.
- You have to merge yourself.
- However, there is a command called `git pull`:
  - Is essentially a `git fetch`.
  - Immediately followed by a `git merge` (in most cases).
- If you have a tracking branch, `git pull` will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Deleting Remote Branches

- Suppose you're done with a remote branch.
- You can delete a remote branch using the `--delete` option to `git push`:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

- Basically all this does is remove the pointer from the server.

Outline

Fork

- Consider that you want modify the code of a project in Github or any other cloud provider for git repositories.
- You can clone a the repo and make changes.
- But, you want to contribute to **the original project!**.
- However, by default, you cannot contribute back to the upstream repo (unless you are explicitly declared as ”contributor”).
- To be able to contribute, the first step is to create a ”fork” (which can be done using the Github menu).

Fork

In github and other cloud providers for git repositories, a fork is a clone on the server side (i.e. Github server). Notice that Github will know that the fork is a clone of an existing project and which project.

Pull Request (AKA Merge Request)

- After creating the fork, we clone it locally.
- Then, we create a new branch in which make our changes.
- Next, we push the changes in our branch to Github (we have the rights to do so).
- The fork in Github will detect this new branch and allow us to create a ”pull request” (in gitlab is called ”merge request”).

Pull Request (AKA merge request)

A pull request or merge request allows a fork to send a message to the original project with a contribution in a new branch. If the project accepts our changes they will merge our branch into their code.

Outline

Tags

- Git has the ability to tag specific points in history as being important.
- Typically people use this functionality to mark release points (v1.0, and so on).
- Git uses two main types of tags:
  - A **lightweight tag** is just a pointer to a specific commit.
  - **Annotated tags** are stored as full objects in the Git database. They're checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
- It's generally recommended that you create annotated tags so you can have all this information.

Annotated Tags

- Create an annotated tag:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

- You can list with a pattern:

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
```

- You can use `git show` to view the tag info.
- you can use `-s` to create a GPG-signed tag.

Sharing Tags

- By default, the `git push` command does not transfer tags to remote servers, you have to explicitly push tags:

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

- If you have a lot of tags that you want to push up at once:

```
$ git push origin --tags
Counting objects: 1, done.
...
```

- Tags used to denote versioned releases typically use annotated tags.

Checkout a Tag

- You can checkout a tag with the following command:

```
$ git checkout tags/v2.0.0
```

- Or simply with:

```
$ git checkout v2.0.0
```

- Actually, you can't really check out a tag in Git **to work with** because since they can't be moved around.
- To work from a tag you should create a branch in your repo that looks like a specific tag:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Outline

Basic Git Commands

Install:

```
$ sudo apt install git
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Create local repos:

```
myrepo$ git init
$ git clone REMOTE_PATH
```

Work with the local repos:

```
myrepo$ git add file
myrepo$ git status
myrepo$ git commit -m "first commit"
myrepo$ git commit -am "second commit"
myrepo$ git log
myrepo$ git checkout c2a3c...
myrepo$ git checkout master
myrepo$ git checkout -b mynewbranch
myrepo$ git checkout master
myrepo$ git merge mynewbranch
myrepo$ git branch -d mynewbranch
myrepo$ git branch -vv
myrepo$ git tag -a v2.0 -m 'my version 2.0'
myrepo$ git tag
myrepo$ git checkout v2.0
myrepo$ git checkout -b version2 v2.0
```

Work with remote repos:

```
myrepo$ git init --bare
myrepo$ git remote add REMOTE_NAME REMOTE_PATH
myrepo$ git push -u origin master
myrepo$ git branch -u anotherorigin/master
myrepo$ git push [origin master]
myrepo$ git push origin --all
myrepo$ git checkout --track origin/mybranch
myrepo$ git branch -avv
myrepo$ git push origin mybranch
myrepo$ git push origin --delete mybranch
myrepo$ git pull
myrepo$ git push origin v2.0
myrepo$ git push origin --tags
```