Week 11: Git Tutorial

- 11.1 Git Basic
- 11.2 Git Branching
- 11.3 GitHub

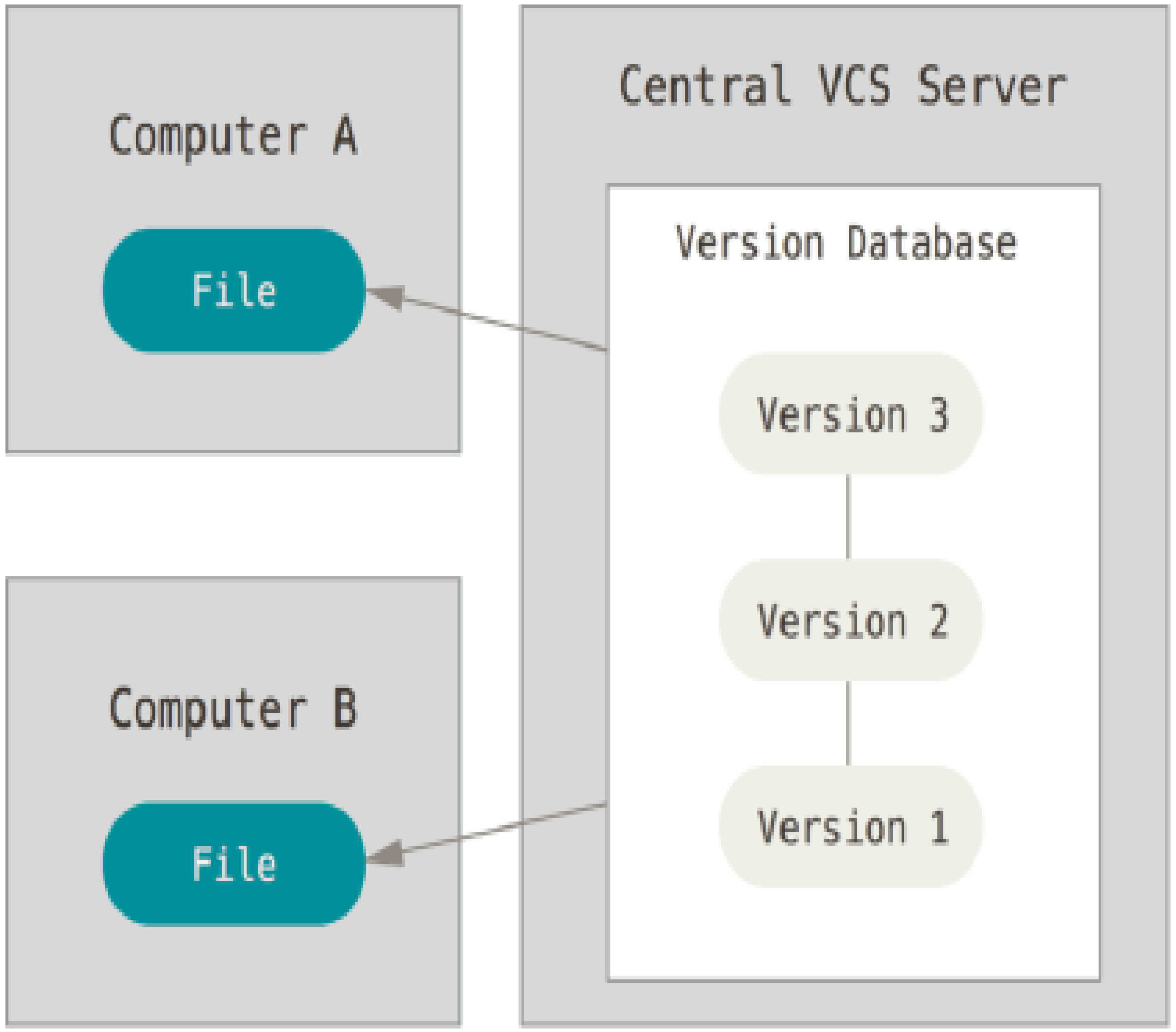
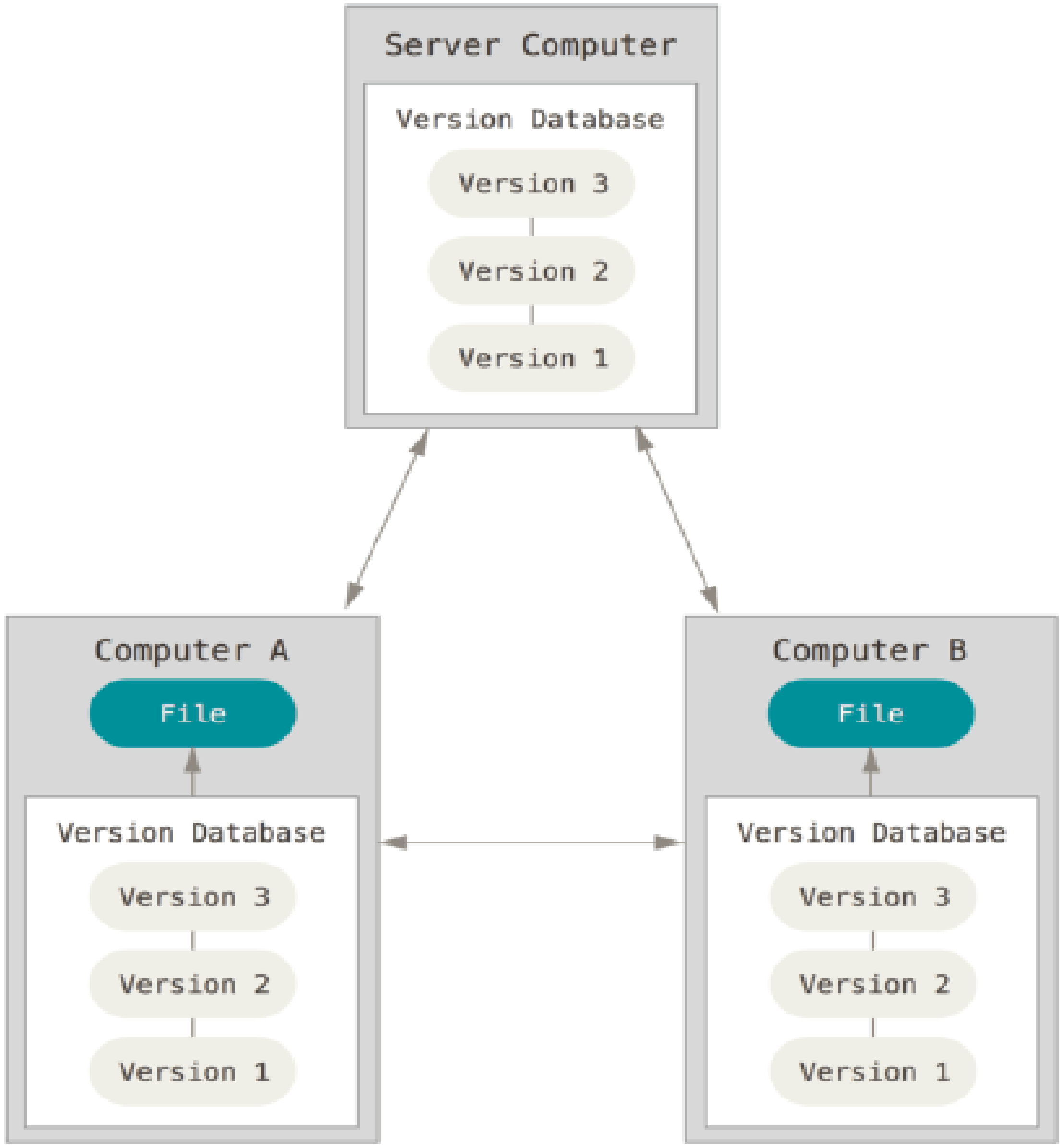


FIGURE 1-2

Centralized version control.

FIGURE 1-3

Distributed version control.



11.1 Git Basics

- The Three States
  - Git has three main states that your files can reside in: *committed*, *modified*, and *staged*.
  - *Committed* means that the data is safely stored in your local database.
  - *Modified* means that you have changed the file but have not committed it to your database yet.
  - *Staged* means that you have marked a modified file in its current version to go into your next commit snapshot.
- This leads us to the three main sections of a Git project: the Git directory, the working directory, and the staging area.

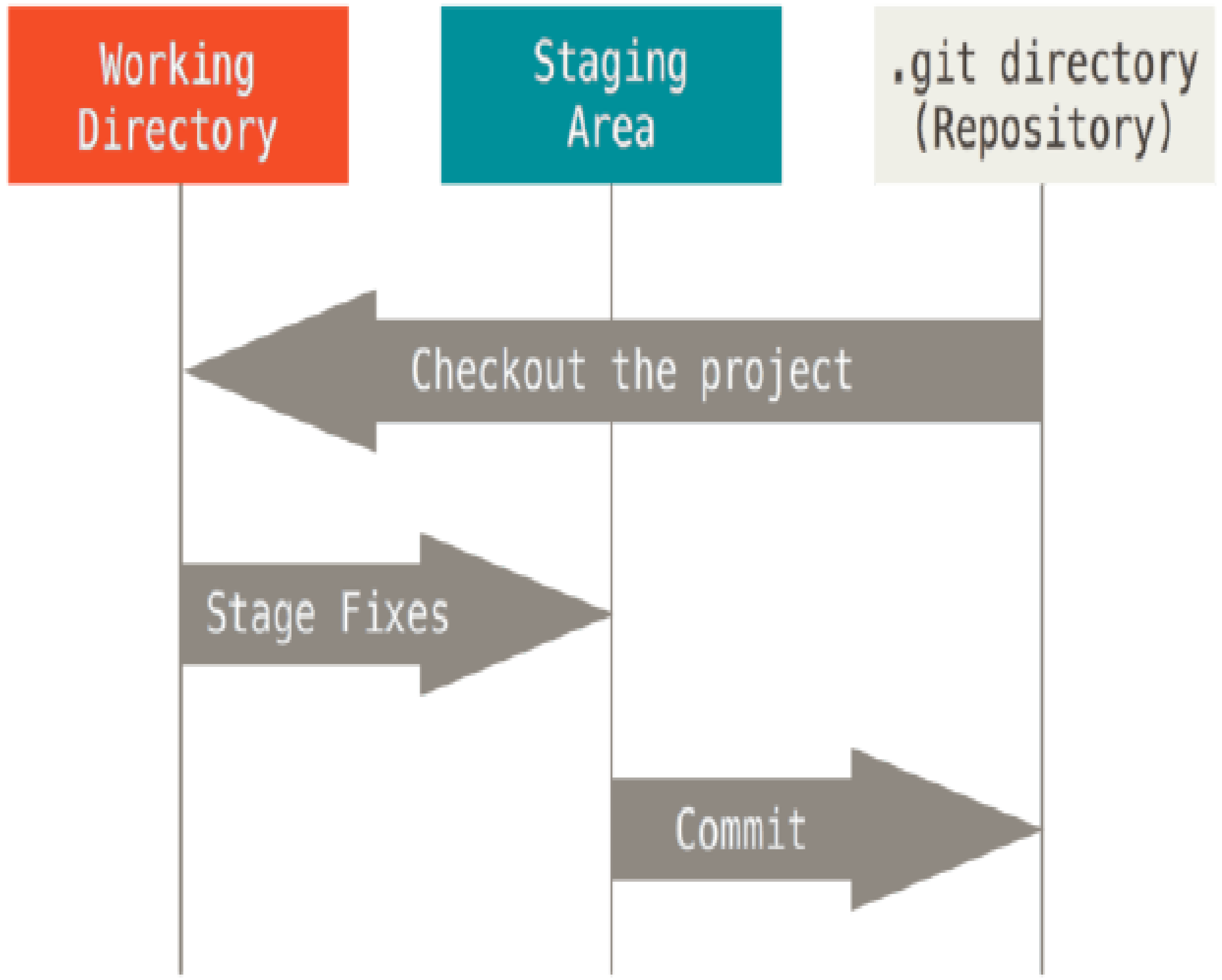


FIGURE 1-6

Working directory, staging area, and Git directory.

- The *Git directory* is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

- The *working directory* is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

- The *staging area* is a file, generally contained in your Git directory, that stores information about what will go into your next commit.

- The basic Git workflow goes something like this:
  - 1. You *modify* files in your working directory.
  - 2. You *stage* the files, 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.

First-Time Git Setup

- Download Git at https://git-scm.com/downloads
- Your Identity
  - The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

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

- Getting Colored Output

```
$ git config --global color.ui auto
```

- Copying and Pasting in Git Bash
  - Go to Menu->Properties and in the Options tab select QuickEdit Mode.
- To clone a repository, run *git clone* followed by a space and the repository URL.

```
$ git clone https://github.com/larthorn/asteroids.git
```

- Downloading necessary files (already in asteroids.git)
  - git-completion.bash
    - https://raw.githubusercontent.com/git/git/master/contrib/completion/git-completion.bash
  - git-prompt.sh
    - https://raw.githubusercontent.com/git/git/master/contrib/completion/git-prompt.sh
  - bash_profile_course, then rename it to .bash_profile
  - Save these file in your home directory.

- Making Git configurations

```
$ git config --global core.editor
"'C:/Program Files/Sublime Text 3/sublime_text.exe`
 -n -w"
$ git config --global push.default upstream
$ git config merge.conflictstyle diff3
```

- Make sure you can start your editor from Git Bash. If you use Sublime, you can do this by adding the following line to your *.bash_profile*:

```
alias subl="C:/Program\ Files/Sublime\ Text\
2/sublime_text.exe"
```

- Restart Git Bash

- Checking Your Settings
  - If you want to check your settings, you can use the

    ```
    $ git config --list
    ```

    command to list all the settings Git can find at that point.

- You can also check what Git thinks a specific key’s value is by typing `$ git config <key>:`
-

```
$ git config user.name
```

- Getting Help
  - If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

- For example, you can get the manpage help for the config command by running

```
$ git help config
```

Getting a Git Repository

- You can get a Git project using two main approaches. The **first** takes an existing project or directory and imports it into Git. The **second** clones an existing Git repository from another server.
- **1) Initializing a Repository in an Existing Directory**
  – If you're starting to track an existing project in Git, you need to go to the project's directory and type:

```
$ git init
```

  – This creates a new subdirectory named *.git* that contains all of your necessary repository files – *a Git repository skeleton*.

- If you want to start version-controlling existing files, you should probably begin tracking those files and do an initial commit. You can accomplish that with a few *git add* commands that specify the files you want to track, followed by a *git commit*:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

2) Cloning an Existing Repository

– If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is *git clone*.

- You clone a repository with *git clone [url]*. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

```
$ git clone https://github.com/libgit2/libgit2
```

- That creates a directory named "libgit2", initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

- If you want to clone the repository into a directory named something other than “libgit2”, you can specify that as the next command-line option:

```
$ git clone https://github.com/libgit2/libgit2
            mylibgit
```

- That command does the same thing as the previous one, but the target directory is called *mylibgit*.

Recording Changes to the Repository

- Remember that each file in your working directory can be in one of two states: *tracked* or *untracked*.
- Tracked files are files that were in the last snapshot; they can be *unmodified*, *modified*, or *staged*.

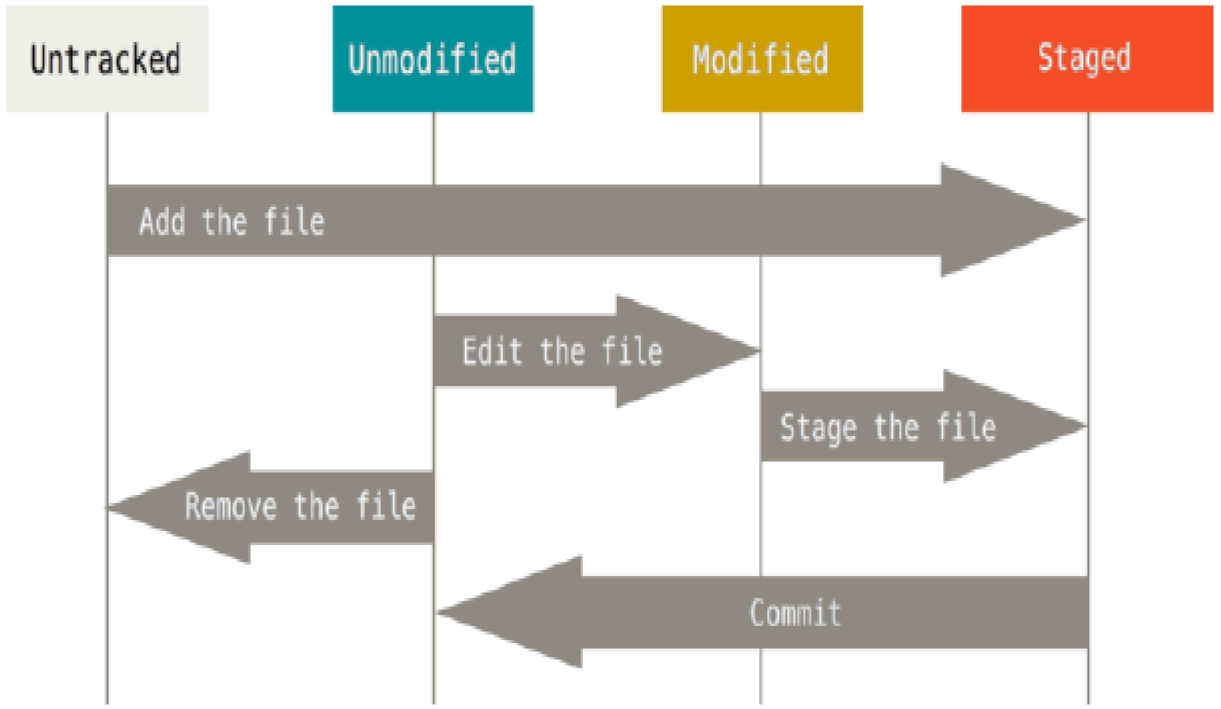


FIGURE 2-1

The lifecycle of the status of your files.

- *Untracked* files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area.

- When you first clone a repository, all of your files will be *tracked* and *unmodified* because Git just checked them out and you haven’t edited anything.

- As you edit files, Git sees them as *modified*, because you’ve changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.

Checking the Status of Your Files

- The main tool you use to determine which files are in which state is the *git status* command. If you run this command directly after a clone, you should see:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

- Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run *git status*, you see your untracked file:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
(use "git add <file>..." to include in what will be
committed)

README

nothing added to commit but untracked files present (use "git
add" to track)
```

- Untracked means that Git sees a file you didn’t have in the previous snapshot (commit); Git won’t start including it in your commit snapshots until you explicitly tell it to do so.
- It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include.
- Tracking New Files: In order to begin tracking a new file, you use the command *git add*. To begin tracking the README file, you can run this:

```
$ git add README
```

- If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

new file: README
```

Staging Modified Files

- Let’s change a file that was already tracked. If you change a previously tracked file called *CONTRIBUTING.md* and then run your *git status* command again, you get something that looks like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

new file: README

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: CONTRIBUTING.md
```

- Let’s run *git add* now to stage the CONTRIBUTING.md file, and then run *git status* again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

new file: README
modified: CONTRIBUTING.md
```

- Both files are staged and will go into your next commit.

Short Status

- If you run *git status -s* or *git status --short* you get a far more simplified output from the command:

```
$ git status -s
 M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

- New files that aren't tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M.

- There are two columns to the output - the *left-hand* column indicates the status of the *staging area* and the *right-hand* column indicates the status of the *working tree*.

- So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

- Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked.
- These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named *.gitignore*. Here is an example *.gitignore* file:

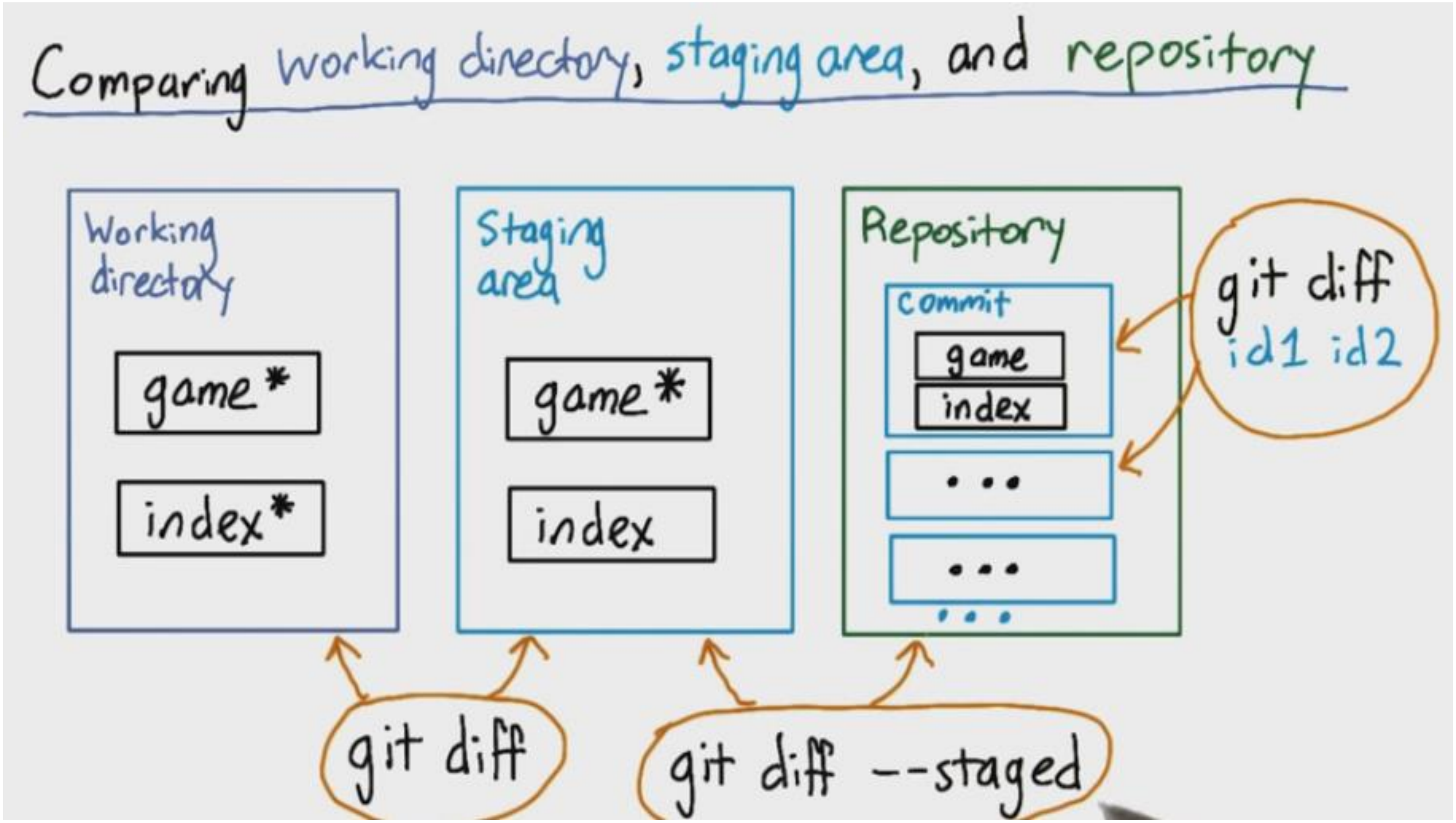
```
$ cat .gitignore
*.[oa]
*~
```

- The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files.

Viewing Your Staged and Unstaged Changes

- *git diff* compares what is in your *working directory* with what is in your *staging area*. The result tells you the changes you’ve made that you haven’t yet staged.

- If you want to see what you’ve staged that will go into your next commit, you can use *git diff --staged*. This command compares your *staged changes* to your *last commit*.


Comparing working directory, staging area, and repository
Working directory
game*
index*
Staging area
game*
index
Repository
commit
game
index
...
...
...
git diff id1 id2
git diff
git diff --staged

- For an example, if you stage the CONTRIBUTING.md file and then edit it, you can use *git diff* to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

modified: CONTRIBUTING.md

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: CONTRIBUTING.md
```

- Now you can use *git diff* to see what is still unstaged:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
## Starter Projects
See our [projects
list](https://github.com/libgit2/libgit2/blob/develo
pment/PROJECTS.md).
+# test line
```

Committing Your Changes

```
$ git commit
```

- Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
2 files changed, 2 insertions(+)
create mode 100644 README
```

- You can see that the commit has given you some output about itself: which branch you committed to (master), what SHA-1 checksum the commit has (463dc4f), how many files were changed, and statistics about lines added and removed in the commit.

Skipping the Staging Area

```
$ git status
On branch master
Your branch is up-to-date with 'origin/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: CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m 'added new benchmarks'
 [master 83e38c7] added new benchmarks
1 file changed, 5 insertions(+), 0 deletions(-)
```

Remove versioned files

- Deletes the file from the working directory and stages the deletion

```
$ git rm [file]
```

- Removes the file from version control but preserves the file locally

```
$ git rm --cached [file]
```

- Changes the file name and prepares it for commit

```
$ git mv [file-original] [file-renamed]
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

deleted: PROJECTS.md

$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
renamed: README.md -> README
```

Viewing the Commit History

- Example repository

```
$ git clone https://github.com/schacon/simplegit-progit
```

- *$ git log* command
- One of the more helpful options is -p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries:

```
    $ git log -p -2
```

- if you want to see some abbreviated stats for each commit, you can use the --stat option:

```
    $ git log --stat
```

Undoing Things

- One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the --amend option:

```
$ git commit –amend
```

- As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

- You end up with a single commit – the second commit replaces the results of the first.

Unstaging a Staged File

- For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type *git add ** and stage them both. How can you unstage one of the two? The *git status* command reminds you:

```
$ git add *
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
renamed: README.md -> README

modified: CONTRIBUTING.md
```

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M CONTRIBUTING.md

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

renamed: README.md -> README

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: CONTRIBUTING.md
```

Unmodifying a Modified File

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README
```

Working with Remotes

- To see which remote servers you have configured, you can run the *git remote* command

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB |
268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.

$ cd ticgit
$ git remote
origin
```

- You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

```
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
```

- To add a new remote Git repository as a shortname you can reference easily, run git remote add <shortname> <url>:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
```

- If you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run *git fetch pb*:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
* [new branch] master -> pb/master
* [new branch] ticgit -> pb/ticgit
```

- When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: *git push [remote-name] [branch-name]*.

```
    $ git push origin master
```

- If you want to see more information about a particular remote, you can use the *git remote show [remote-name]* command. If you run this command with a particular short name, such as origin, you get something like this:

```
$ git remote show origin
* remote origin
Fetch URL: https://github.com/schacon/ticgit
Push URL: https://github.com/schacon/ticgit
HEAD branch: master
Remote branches:
master tracked
dev-branch 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)
```

Removing and Renaming Remotes

- You can run *git remote rename* to change a remote's shortname. For instance, if you want to rename pb to paul, you can do so with *git remote rename*:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

- If you want to remove a remote for some reason – you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore – you can use *git remote rm*:

```
$ git remote rm paul
$ git remote
origin
```

11.3 Git Branching

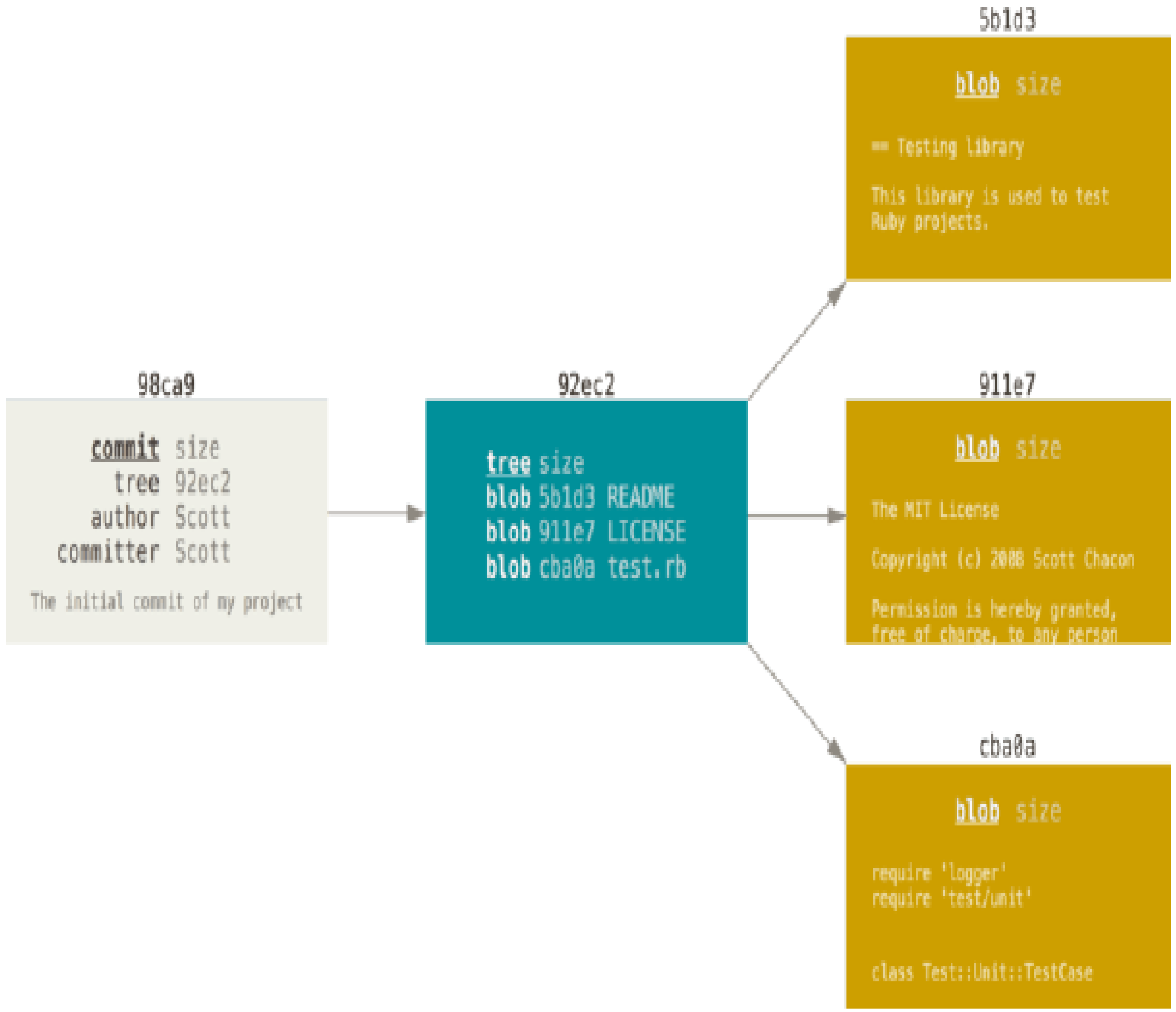
- Git doesn’t store data as a series of changesets or differences, but instead as a series of snapshots.
- When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged.
- This object also contains the author’s name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

- When you create the commit by running *git commit*, Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository.
- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

FIGURE 3-1

A commit and its tree



- If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

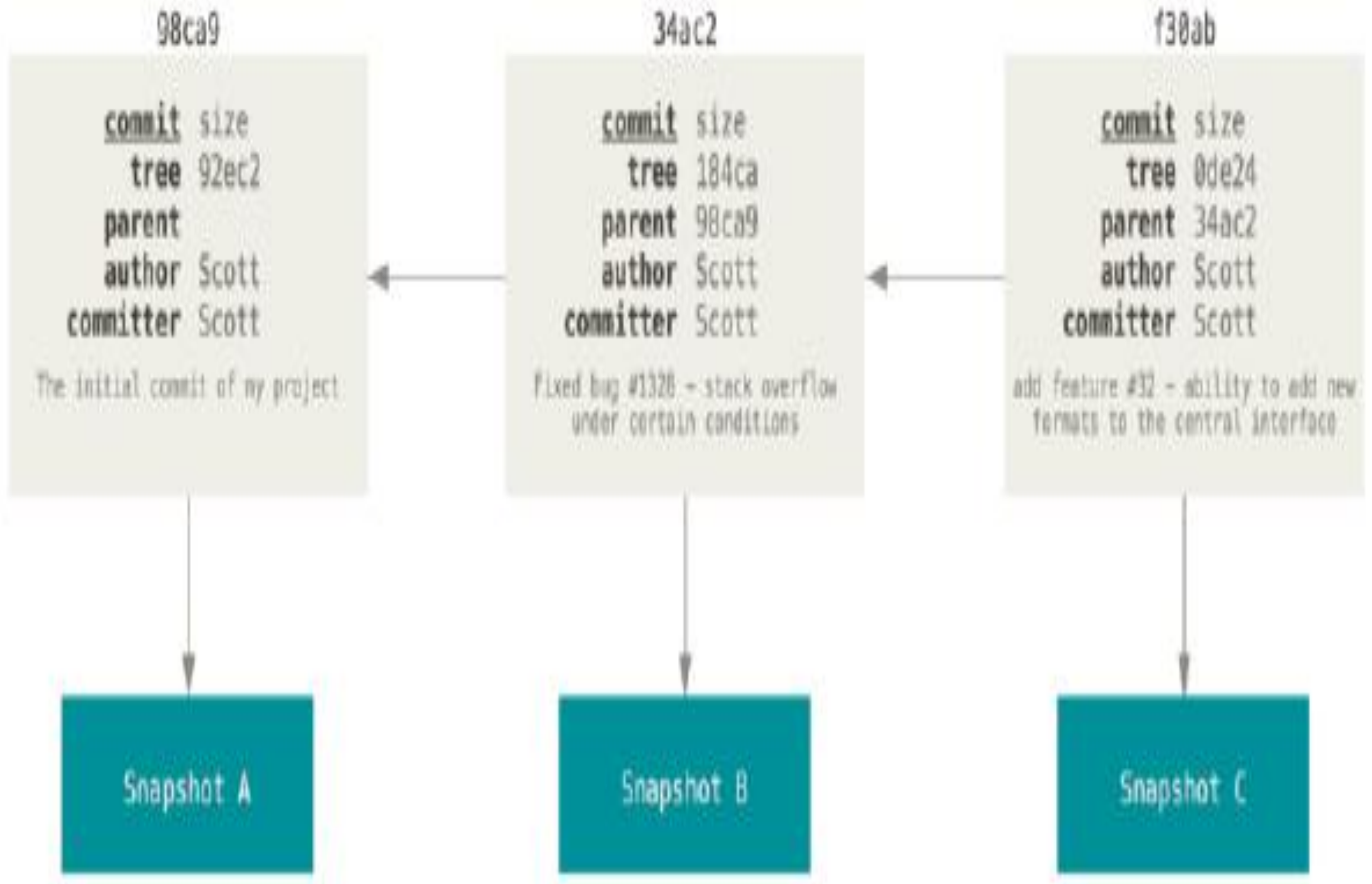


FIGURE 3-2

Commits and their parents

FIGURE 1-4

Storing data as changes to a base version of each file.

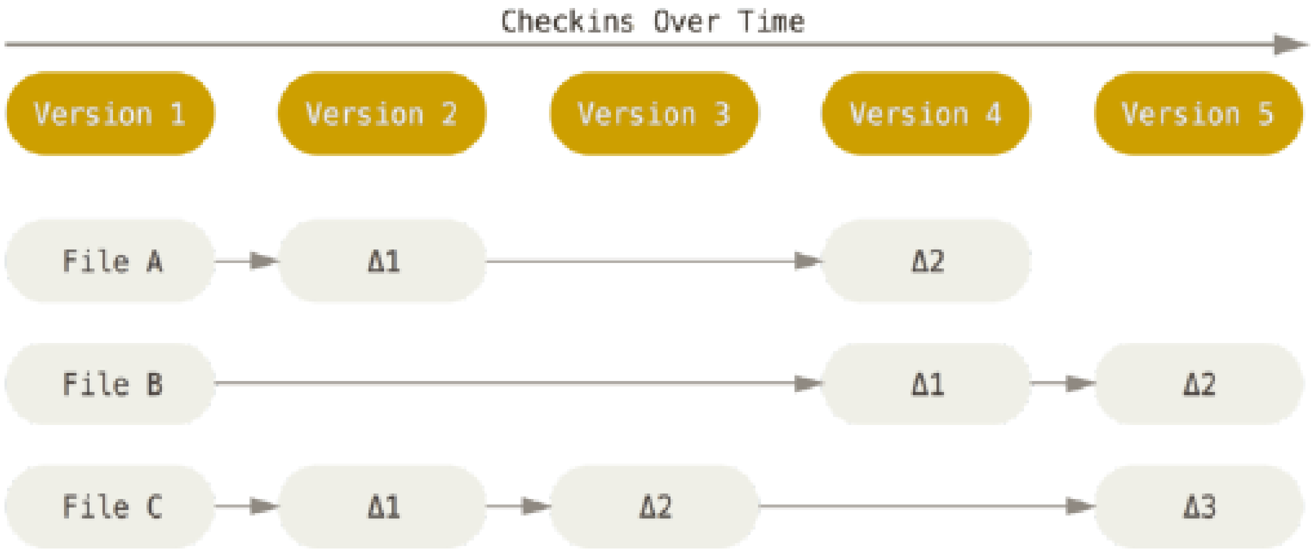
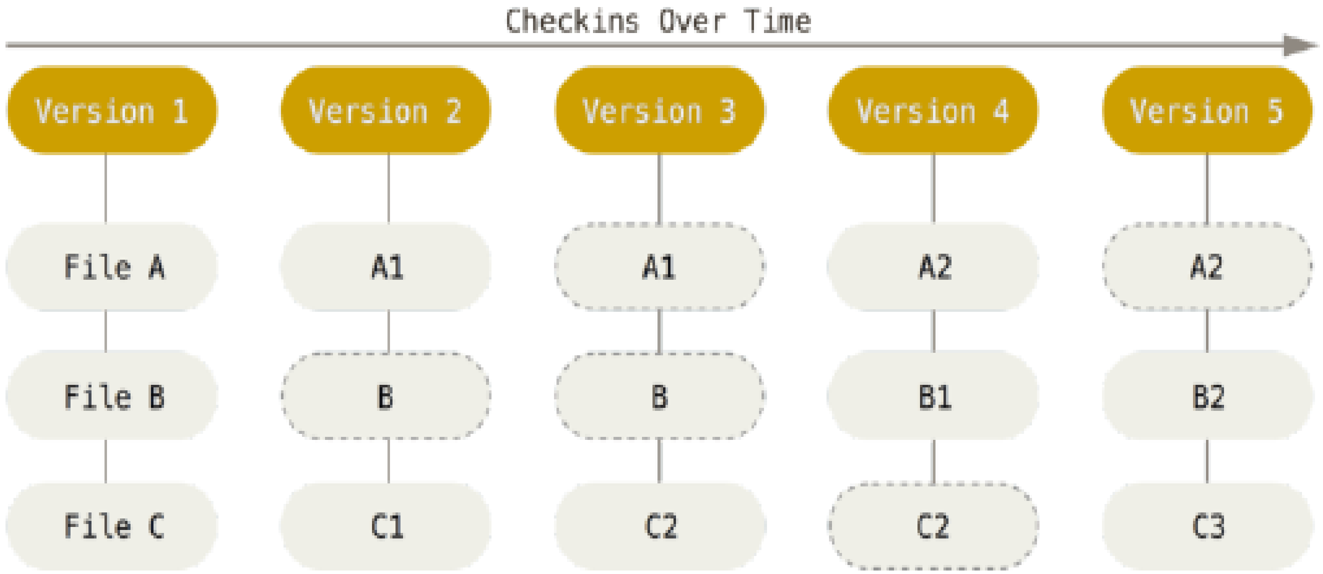


FIGURE 1-5

Storing data as snapshots of the project over time.



- 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 start making commits, you're given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.

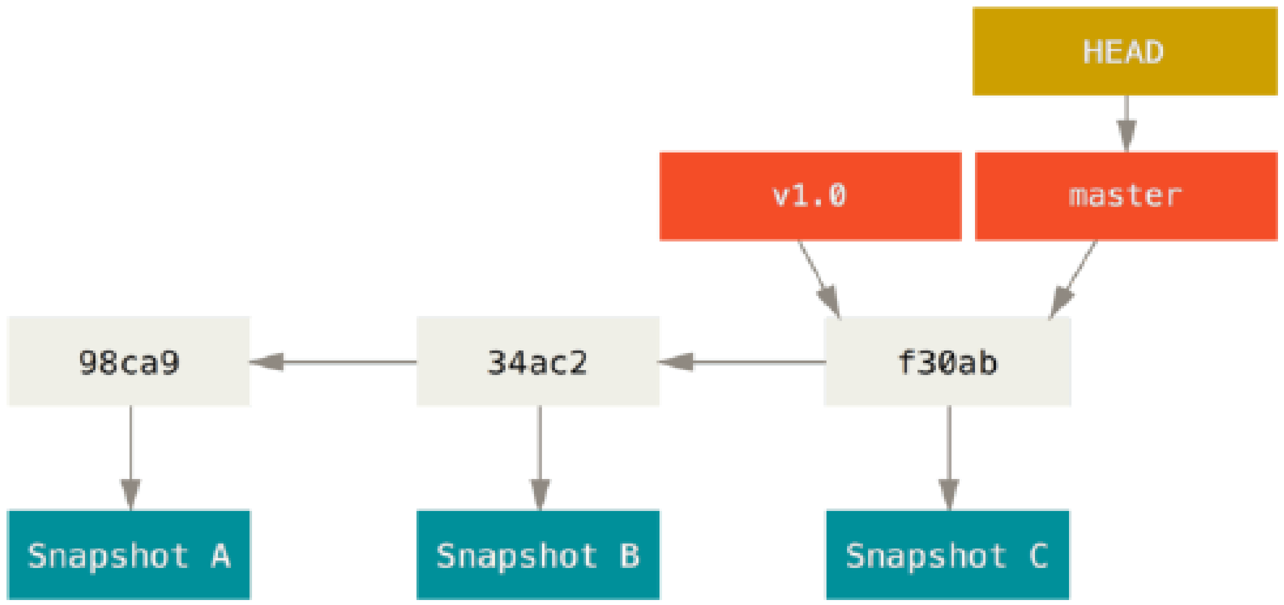


FIGURE 3-3

A branch and its commit history

Creating a New Branch

- What happens if you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you create a new branch called testing. You do this with the *git branch* command:

```
$ git branch testing
```

- This creates a new pointer to the same commit you're currently on.
- The git branch command only *created* a new branch – it didn't switch to that branch.

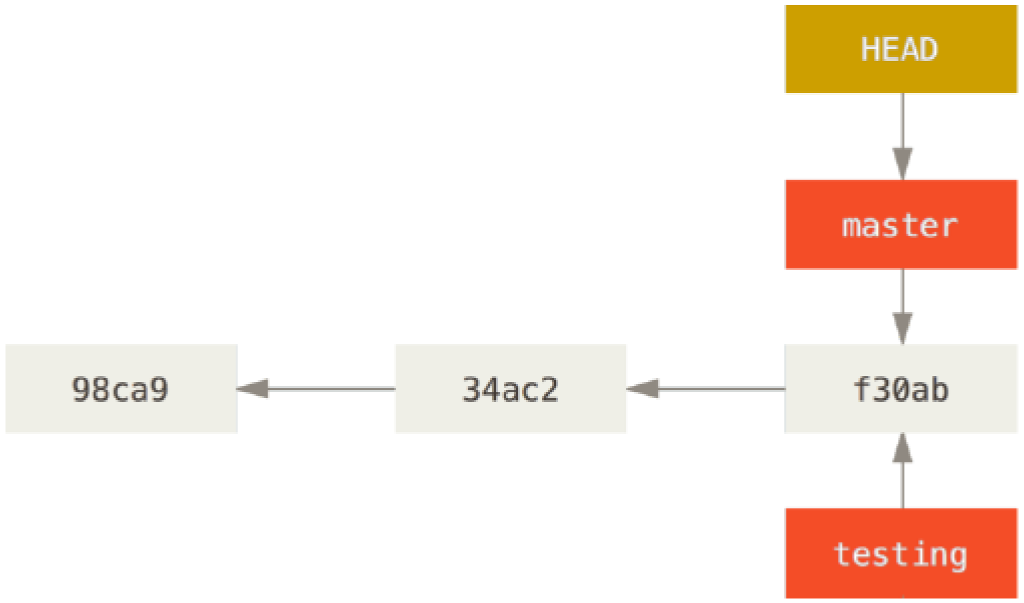


FIGURE 3-5

HEAD pointing to a branch

- You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate.

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

- You can easily see this by running a simple *git log* command that shows you where the branch pointers are pointing. This option is called --decorate.

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

Switching Branches

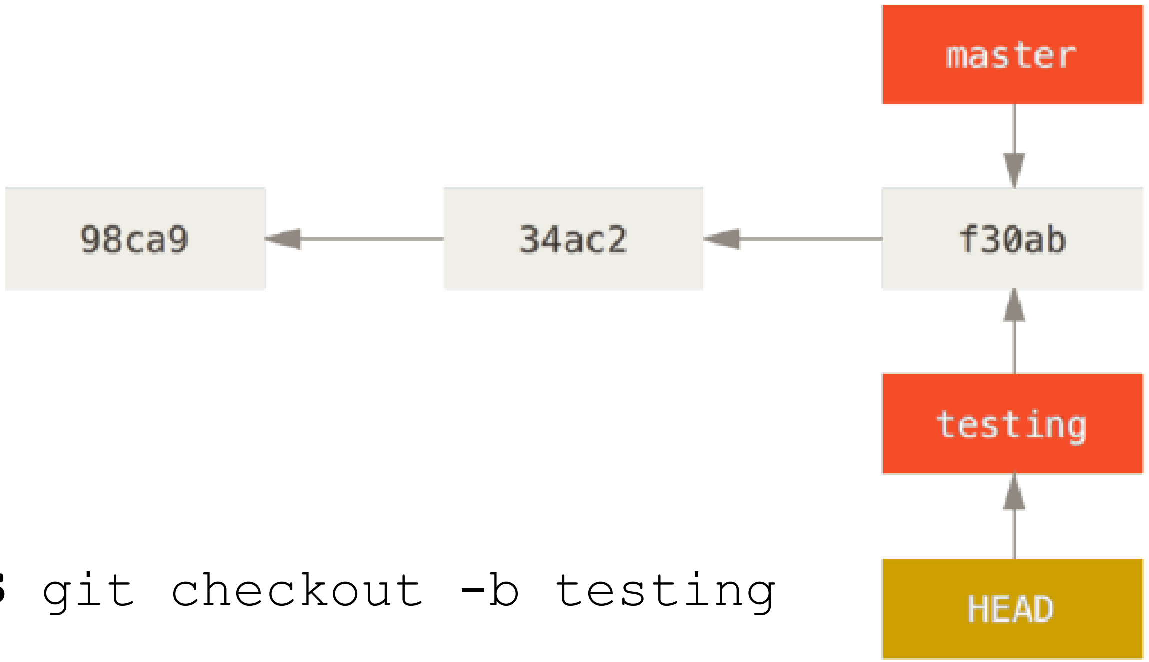
- To switch to an existing branch, you run the git checkout command. Let's switch to the new testing branch:

```
$ git checkout testing
```

- This moves HEAD to point to the testing branch.

FIGURE 3-6

HEAD points to the current branch



- Or using `$ git checkout -b testing`

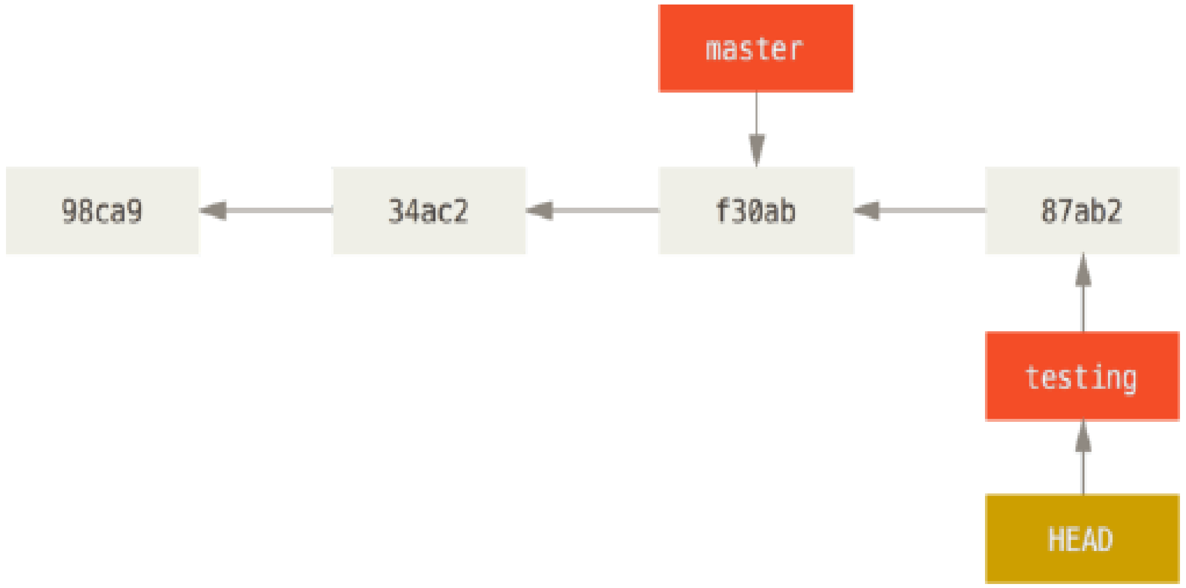
- Well, let’s do another commit:

```
$ vim test.rb
$ git add test.rb
$ git commit -a -m ‘made a change’
```

- This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran *git checkout* to switch branches.

FIGURE 3-7

The HEAD branch moves forward when a commit is made



- Let’s switch back to the master branch:

```
$ git checkout master
```

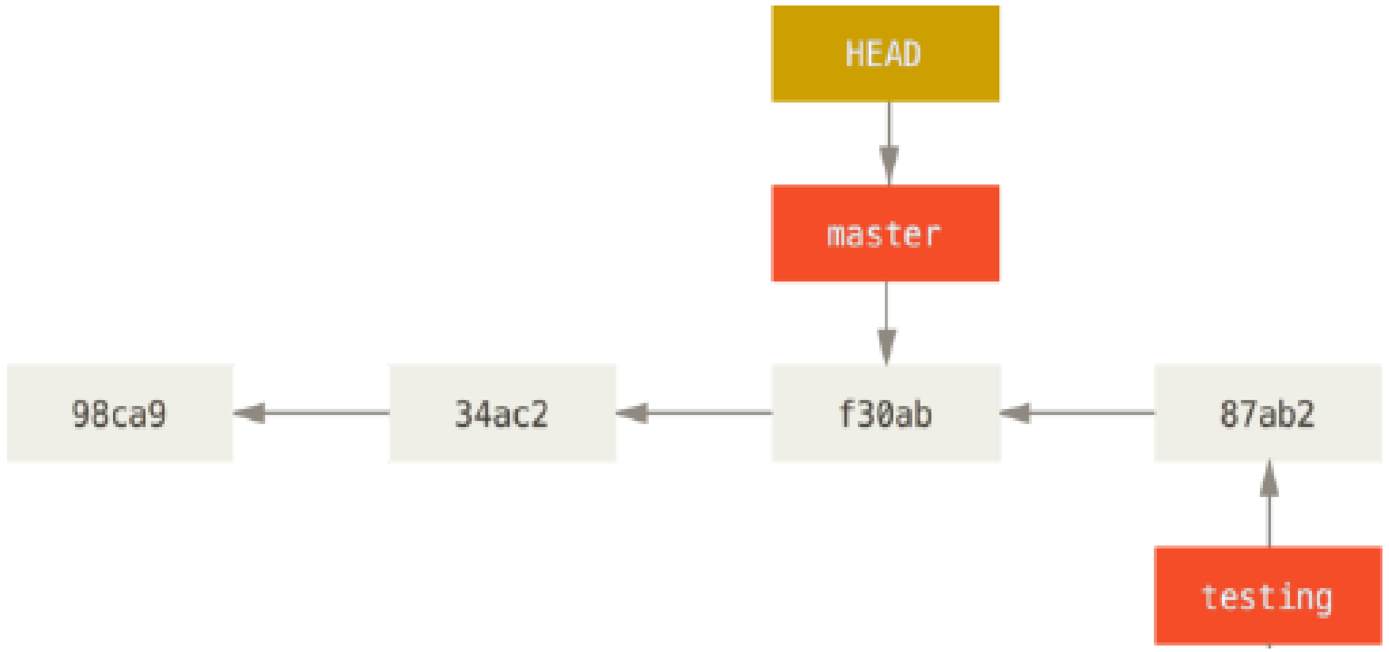


FIGURE 3-8

HEAD moves when you checkout

- This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you’ve done in your testing branch so you can go in a different direction.

- It’s important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch.

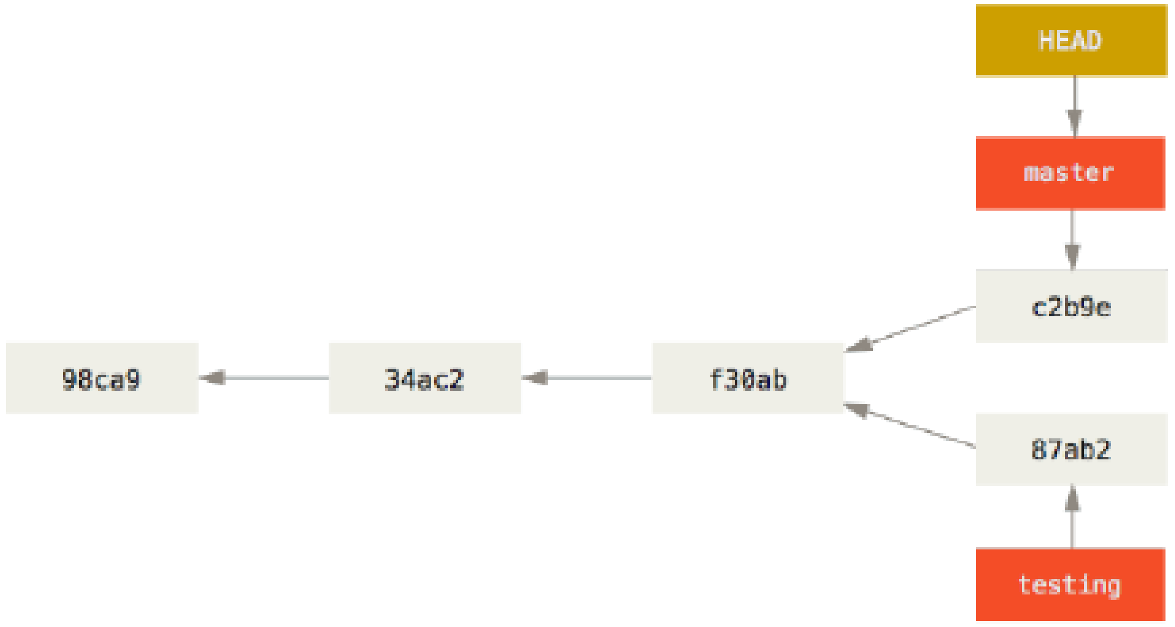
- Let’s make a few changes and commit again:

```
$ vim test.rb
$ git add test.rb
$ git commit -a -m ‘made a change’
```

- Now your project history has diverged

FIGURE 3-9

Divergent history



- You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work.

- Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you’re ready.

- You can also see this easily with the git log command. If you run *git log --oneline --decorate --graph --all* it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new
formats to the
* 34ac2 fixed bug #1328 - stack overflow under
certain conditions
* 98ca9 initial commit of my project
```

Merging

```
$ git checkout master
Already on 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

$ git merge testing
Auto-merging test.rb
CONFLICT (add/add): Merge conflict in test.rb
Automatic merge failed; fix conflicts and then
commit the result.
```

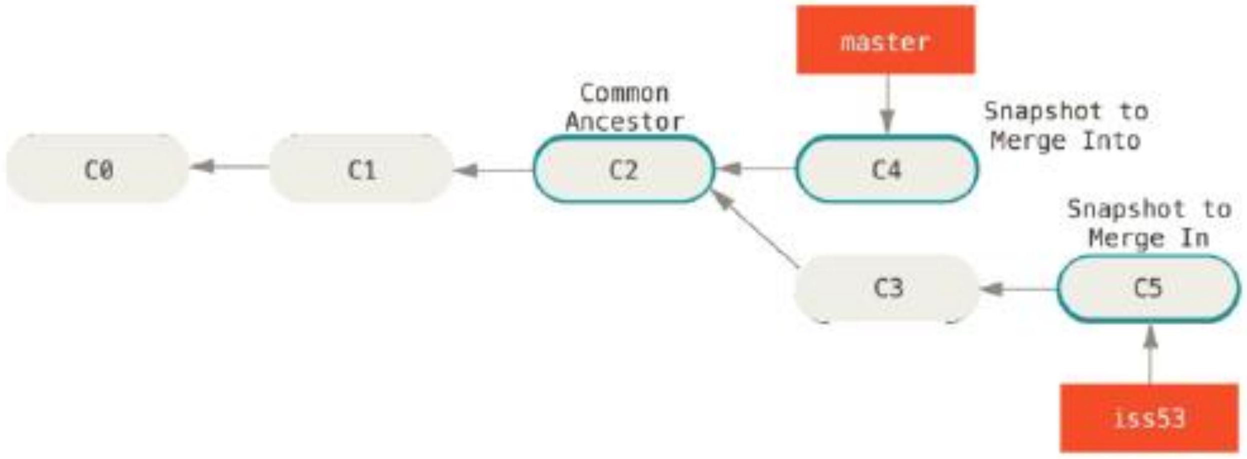


FIGURE 3-16

Three snapshots used in a typical merge

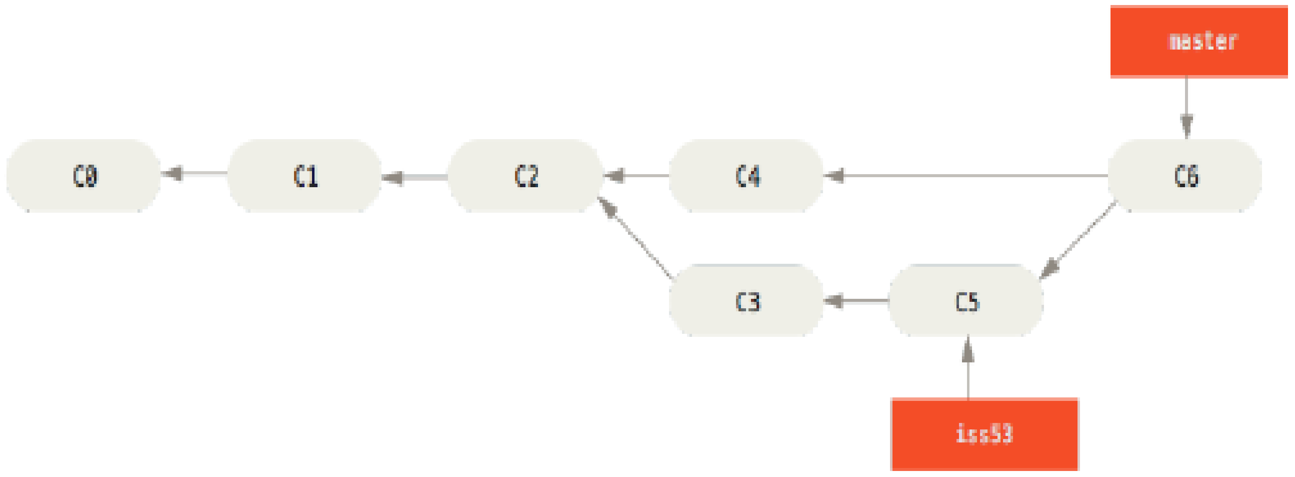


FIGURE 3-17

A merge commit

- Your file contains a section that looks something like this:

```
<<<<<<< HEAD:test.rb
<div id="footer">contact :
email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> testing:test.rb
```

- For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

- This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed.

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both added:      test.rb

no changes added to commit (use "git add" and/or
"git commit -a")
```

```
$ git add test.rb
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   test.rb

$ git commit -m "merge testing into master"
[master dbf9410] merge testing into master
```

```
$ git log --oneline --decorate --graph --all
*   dbf9410 (HEAD -> master) merge testing into
master
|\
| * 314aef9 (testing) made a change
* | 9713f43 made a change2
|/
* 8472568 (origin/master, origin/HEAD) Added note to
clarify which is the canonical TicGit-ng repo
*   cd95b7e Merge branch 'hotfix-1.0.2.2'
|\
| * e493258 Bumped version to 1.0.2.2
| * 470d7af Fixed bug where 'ti comment' was
prepopulated with a ruby object
|/
*   b201539 Merge branch 'hotfix-1.0.2.1'
```

- Delete the label branch ‘testing’

```
$ git branch -d testing
```

Branch Management

- The *git branch* command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

```
$ git branch
iss53
* master
testing
```

- Notice the * character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to).

- The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you're currently on. To see which branches are already merged into the branch you're on, you can run *git branch --merged*:

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

- To see all the branches that contain work you haven't yet merged in, you can run *git branch --no-merged*:

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

Remote Branches

- Let’s say you have a Git server on your network at *git.ourcompany.com*. If you clone from this, Git’s clone command 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.

- 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.

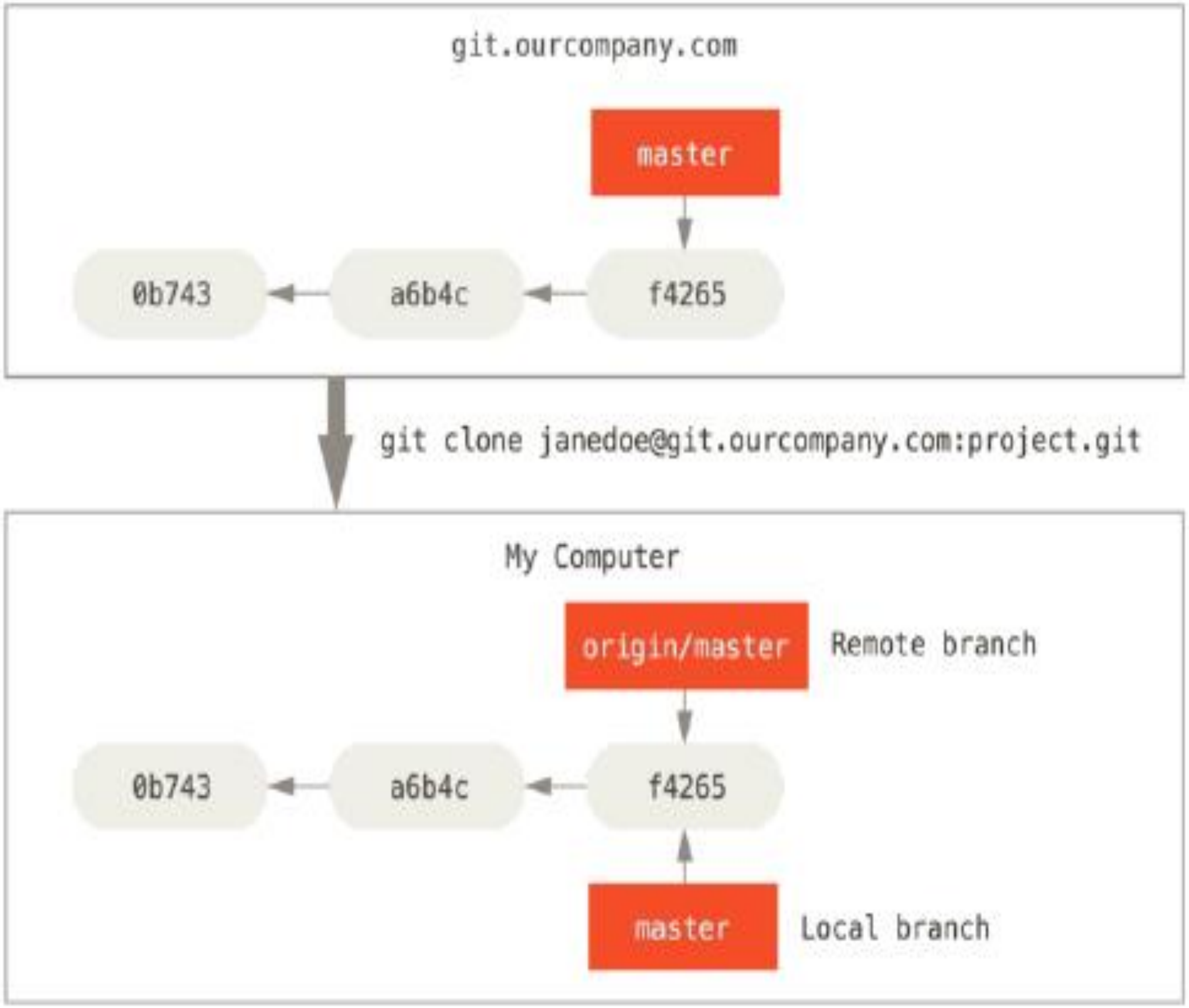


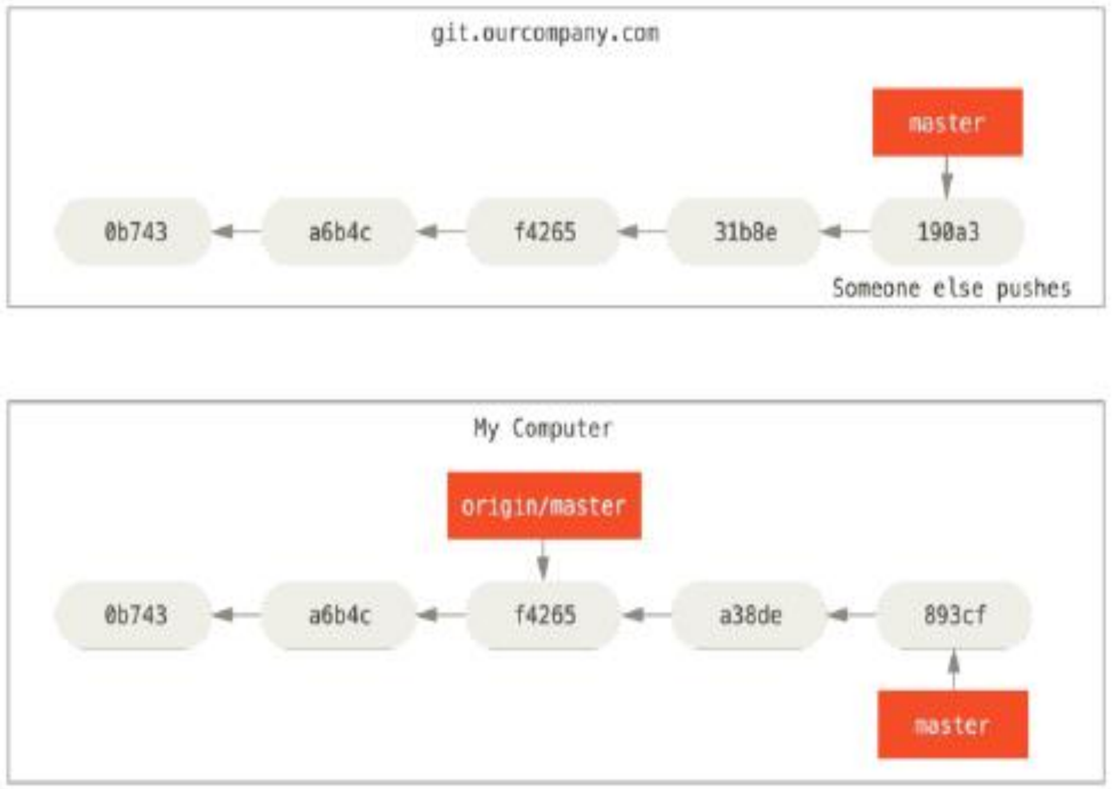
FIGURE 3-22

Server and local repositories after cloning

- 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.

FIGURE 3-23

Local and remote work can diverge



- To synchronize your work, you run a *git fetch origin* command. This command looks up which server "origin" is, 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 position.

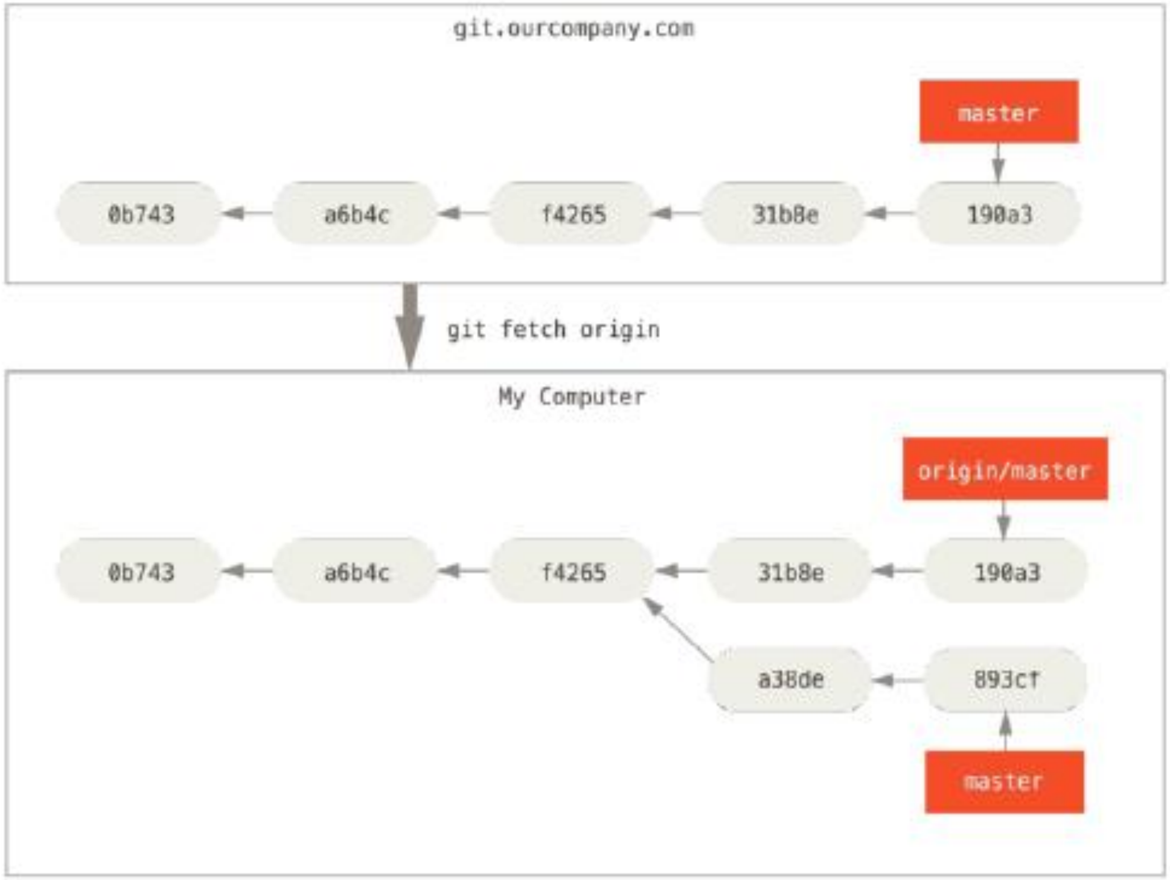


FIGURE 3-24

git fetch updates your remote references

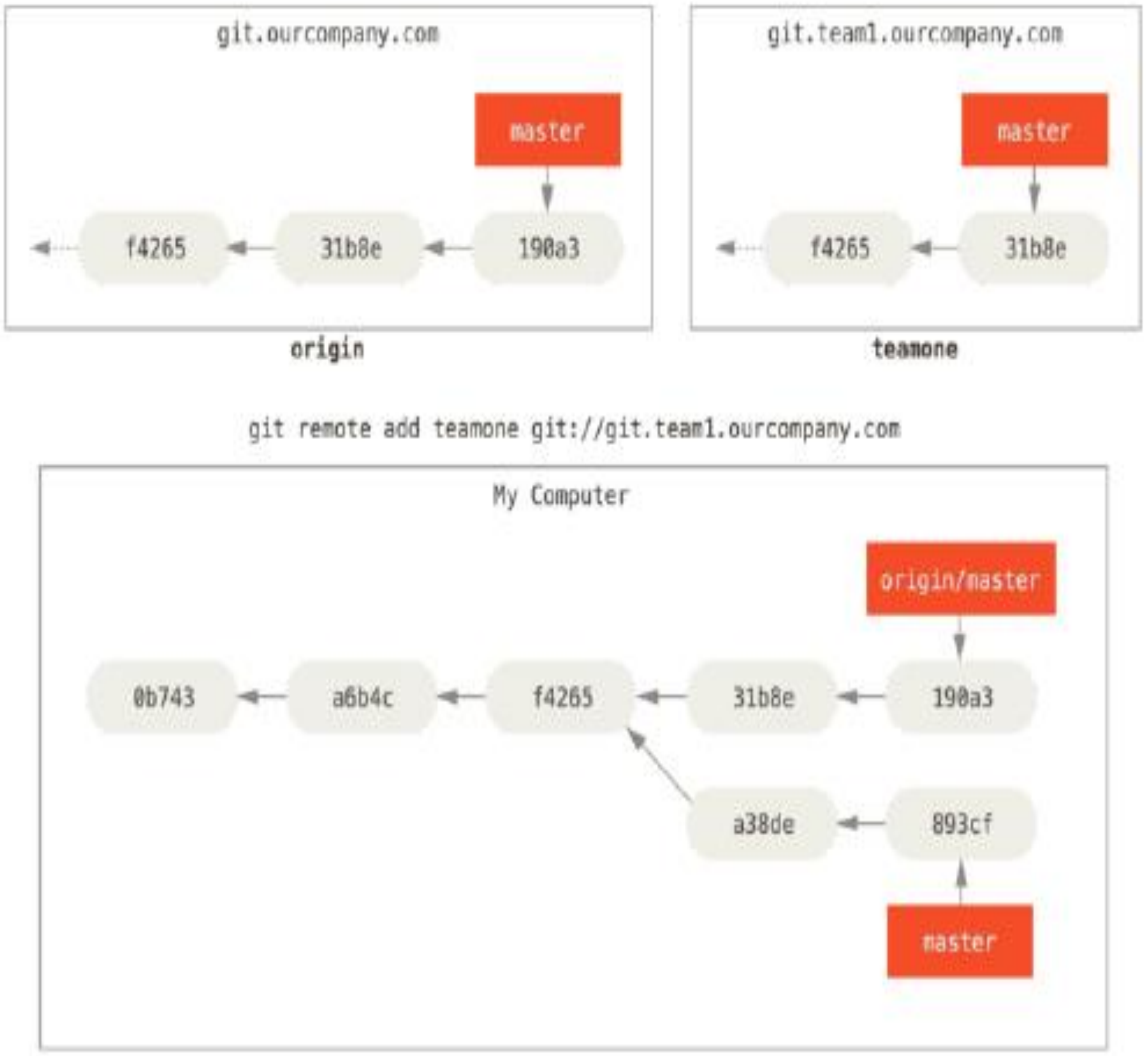
Having multiple remote servers

- let’s assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at *git.team1.ourcompany.com*. You can add it as a new remote reference to the project you’re currently working on by running the *git remote add* command.

- Now, you can run *git fetch teamone* to fetch everything the remote teamone server has that you don’t have yet. Because that server has a subset of the data your origin server has right now, Git fetches no data but sets a remote-tracking branch called teamone/master to point to the commit that teamone has as its master branch.

FIGURE 3-25

Adding another server as a remote



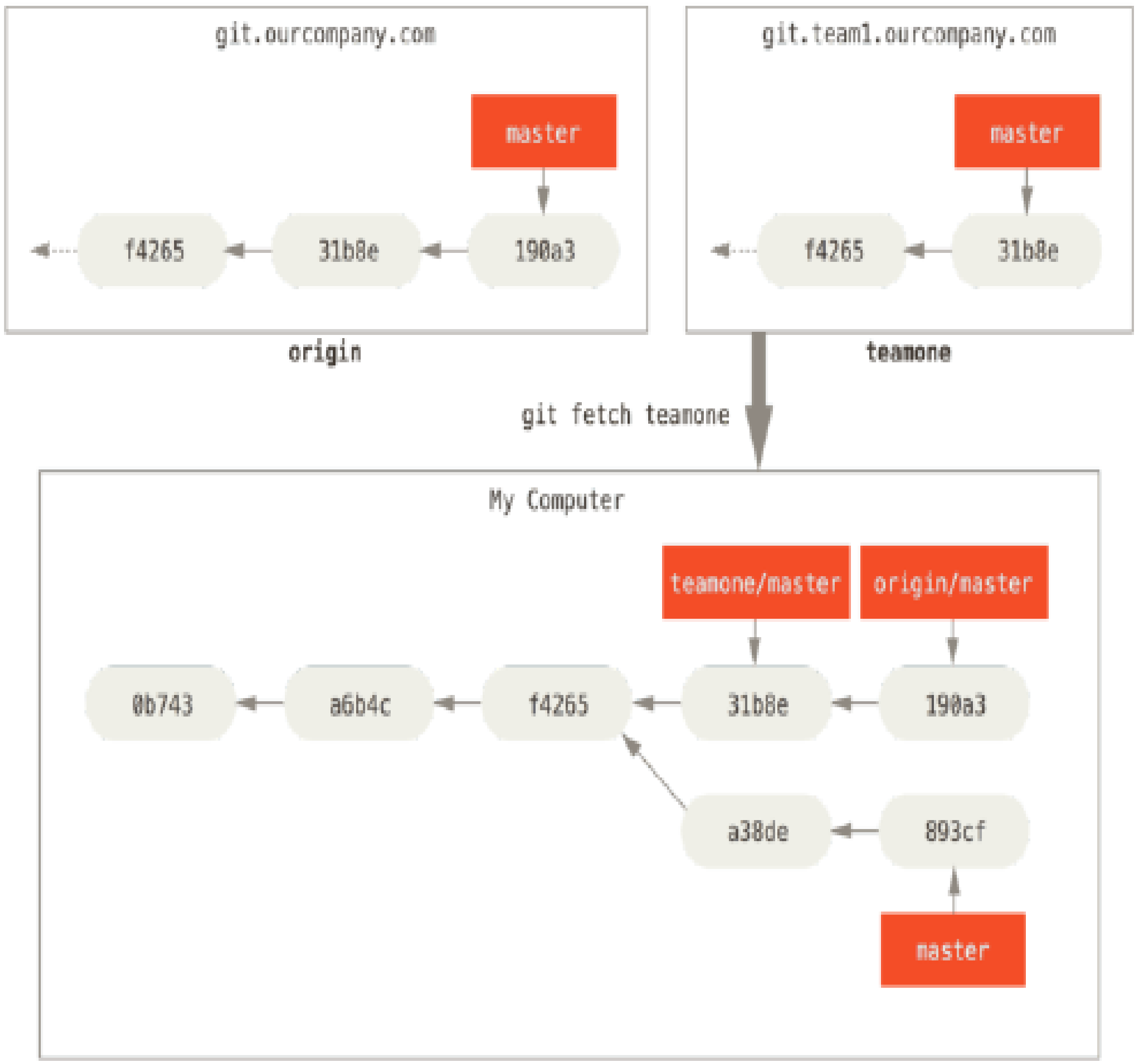


FIGURE 3-26

Remote tracking branch for teamone/master

Pushing

- 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.
- If you have a branch named *serverfix* that you want to work on with others, you can push it up the same way you pushed your first branch. Run git push <remote> <branch>:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s,
done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
* [new branch] serverfix -> serverfix
```

- If you didn’t want it to be called *serverfix* on the remote, you could instead *run git push origin serverfix:awesomebranch* to push your local *serverfix* branch to the *awesomebranch* branch on the remote project.
- The next time one of your collaborators fetches from the server, they will get a reference to where the server’s version of *serverfix* is under the remote branch origin/serverfix: ***$** git fetch origin*
- It’s important to note that when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new serverfix branch –you only have an *origin/serverfix* pointer that you can’t modify.

- To merge this work into your current working branch, you can run *git merge origin/serverfix*. If you want your own *serverfix* branch that you can work on, you can base it off your remote-tracking 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.

Pulling

- While the *git fetch* command will fetch down all the changes on the server that you don't have yet, it will not modify your working directory at all.

- It will simply get the data for you and let you merge it yourself. However, there is a command called *git pull* which is essentially a git fetch immediately followed by a git merge in most cases.

```
$ git pull origin master
```

Deleting Remote Branches

- You can delete a remote branch using the *--delete* option to *git push*. If you want to delete your serverfix branch from the server, you run the following:

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

11.4 GitHub

- Github.com
- Create an account there
- Syncing Repositories to GitHub
  - New a repository on GitHub
  - At your local repository, issue *git remote add origin URL*, e.g.:

```
$ git remote add origin
               https://github.com/larthorn/ticgit.git
```

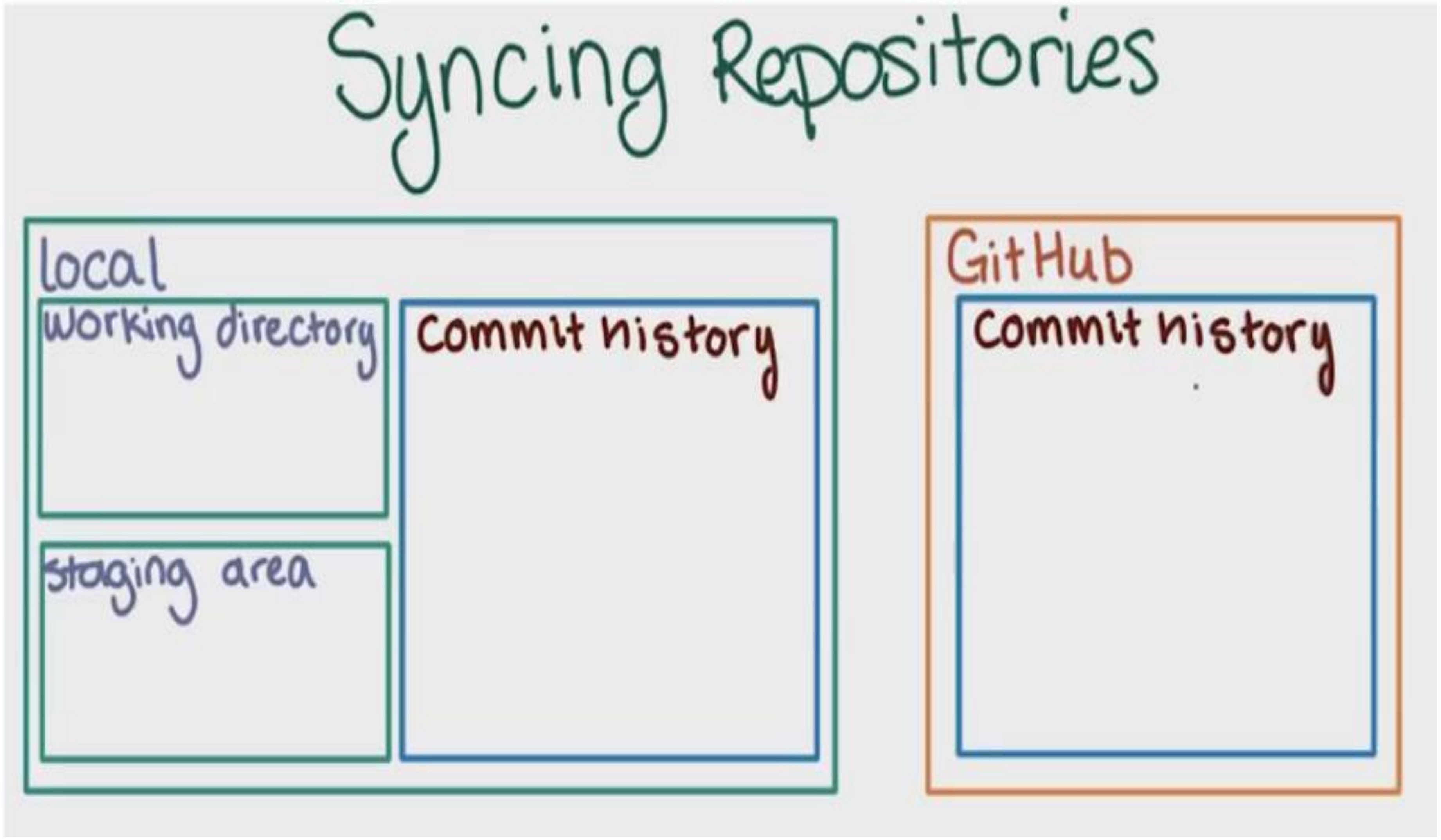
  - Checking with *git remote -v*
  - Copy your content to *origin*

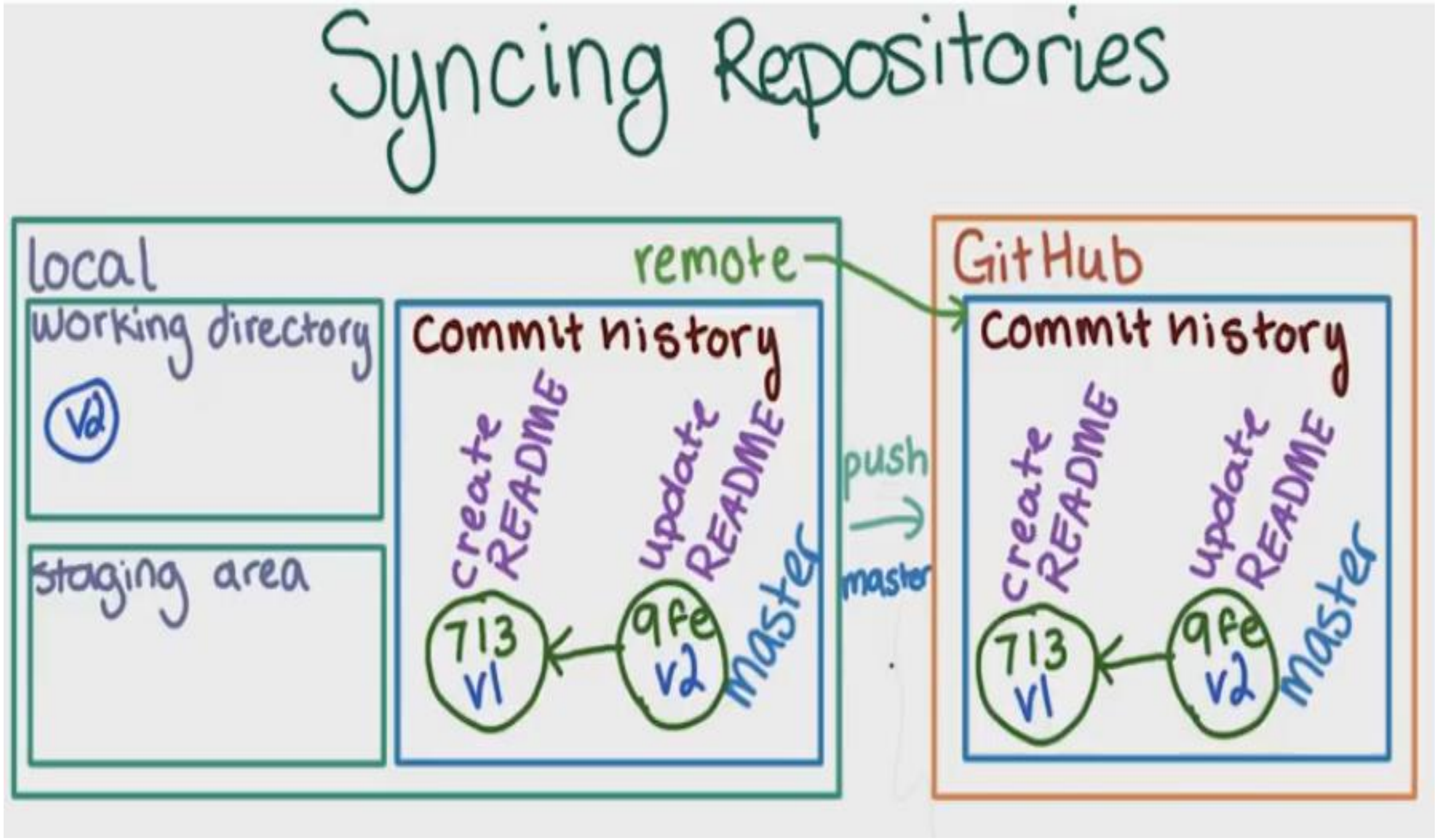
```
         $ git push origin master
```

- Then, creating a new file on GitHub, and commit.
- Pull master (to local machine)

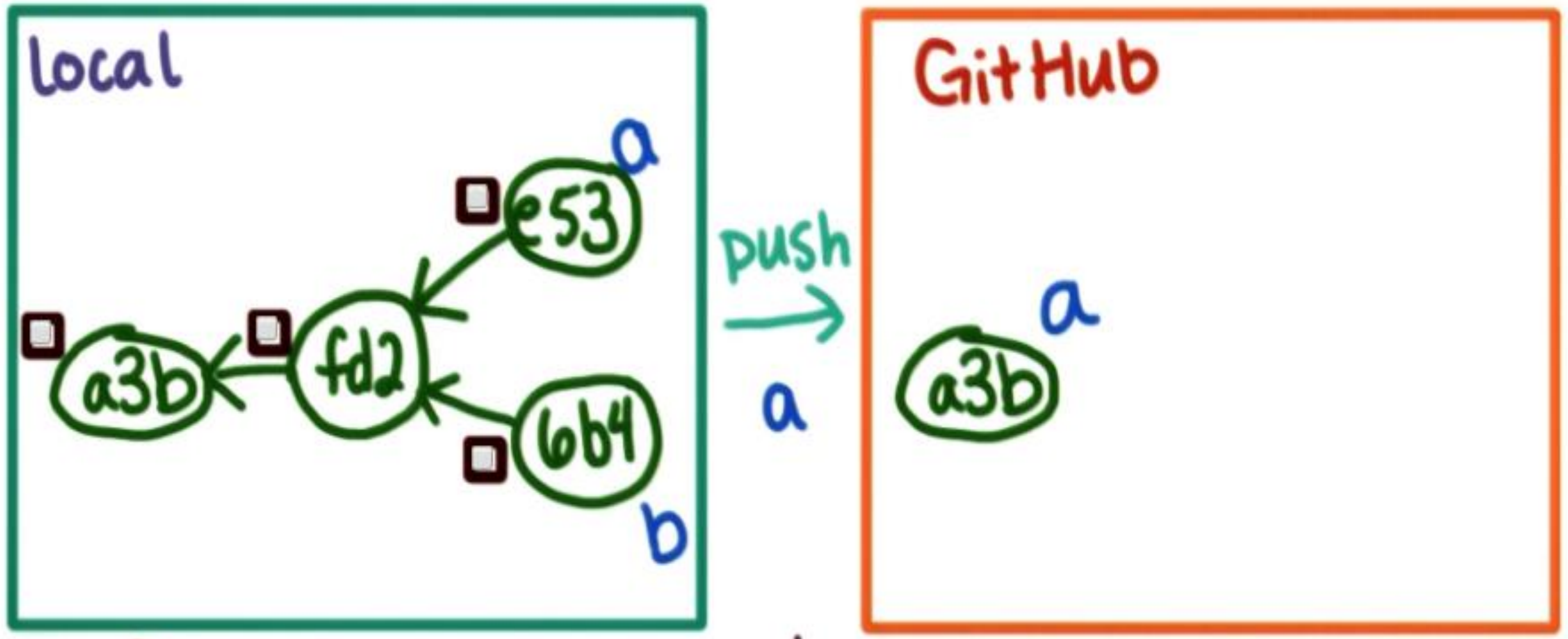
```
    $ git pull origin master
```

- Forking a repository on GitHub

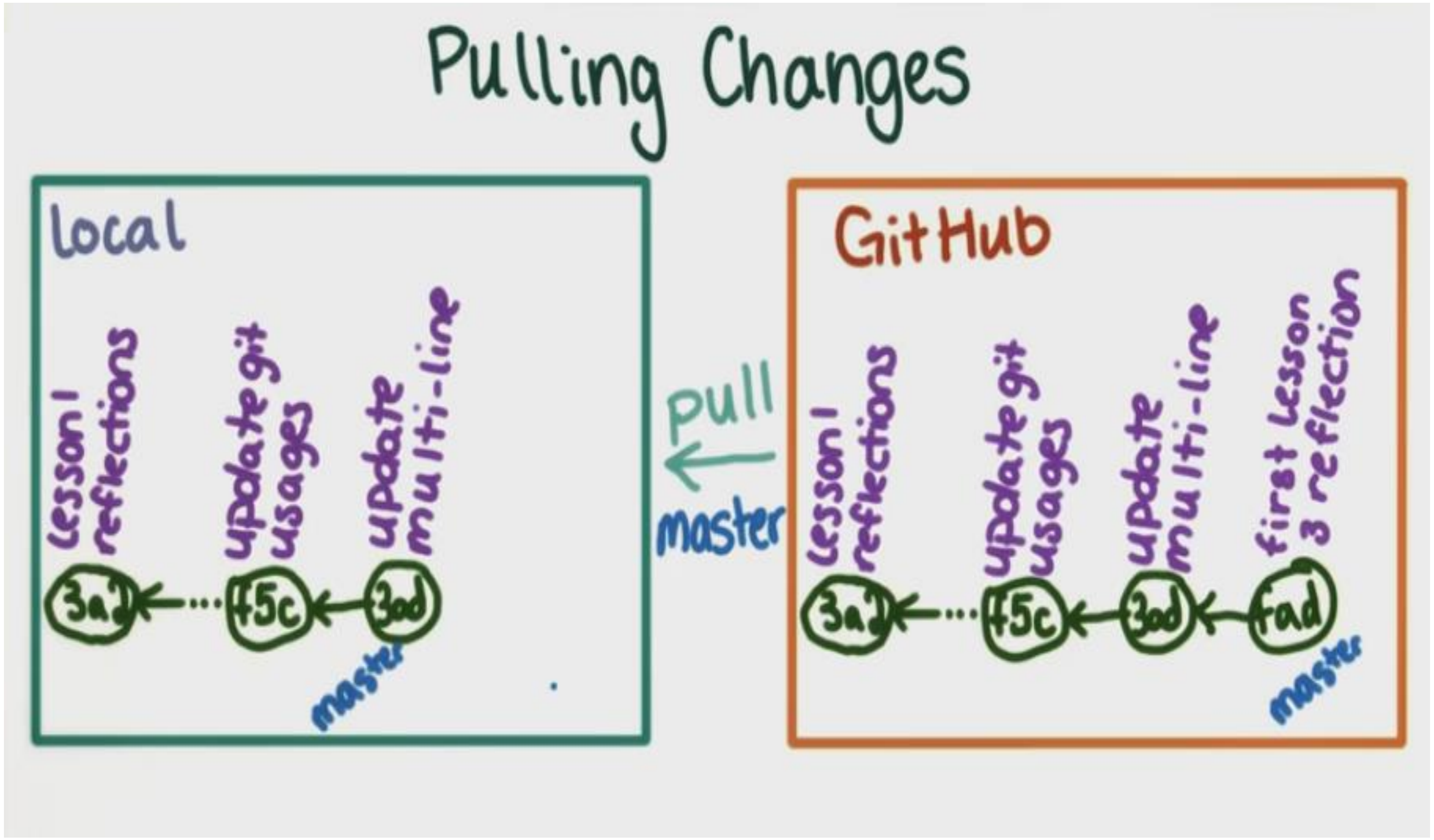

Syncing Repositories
local
working directory
staging area
commit history
GitHub
commit history


Syncing Repositories
local
remote
GitHub
working directory
v2
staging area
Commit history
create README
update README
713 v1
9fe v2
master
push
master
Commit history
create README
update README
713 v1
9fe v2
master

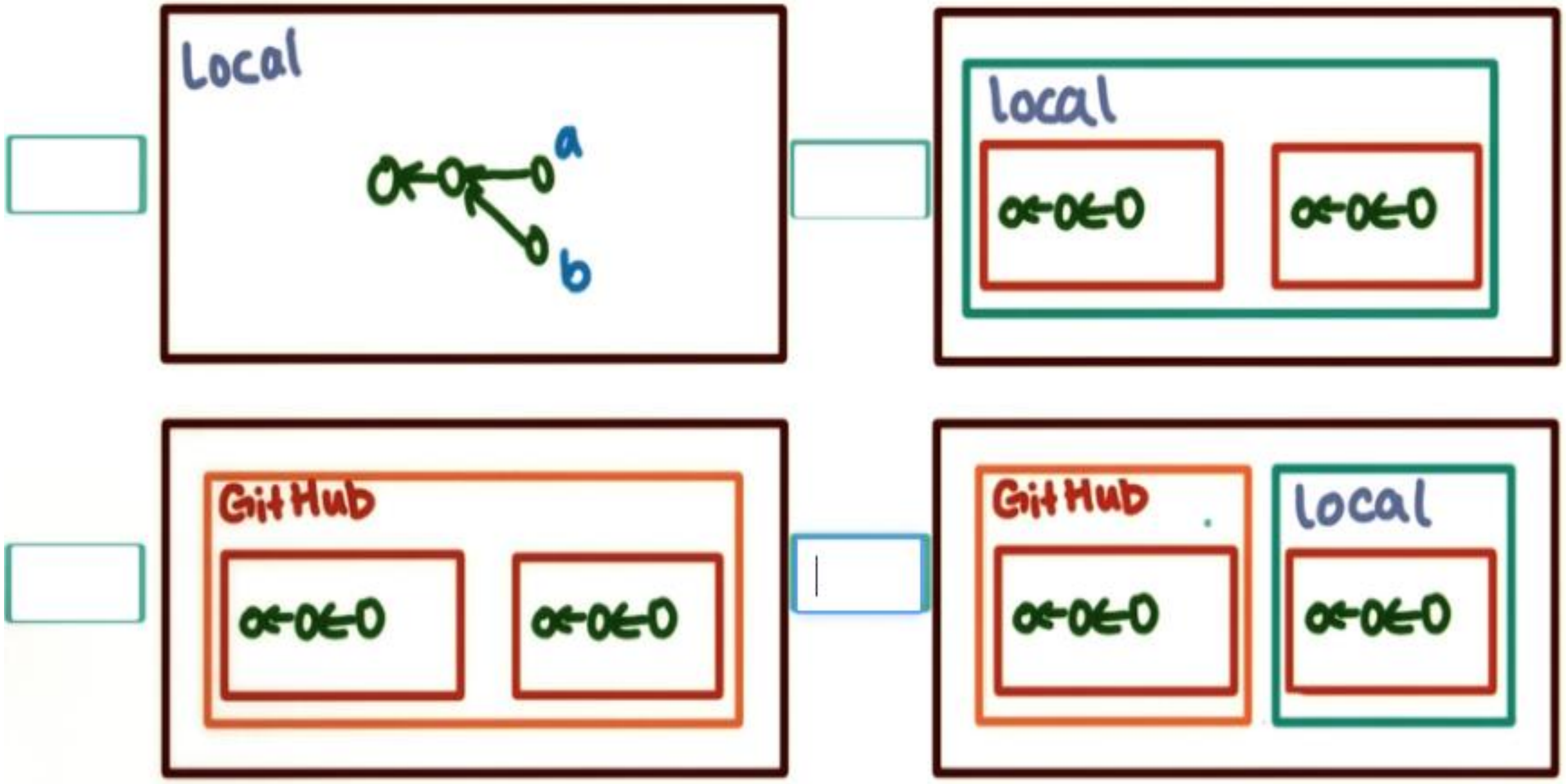
Syncing Repositories



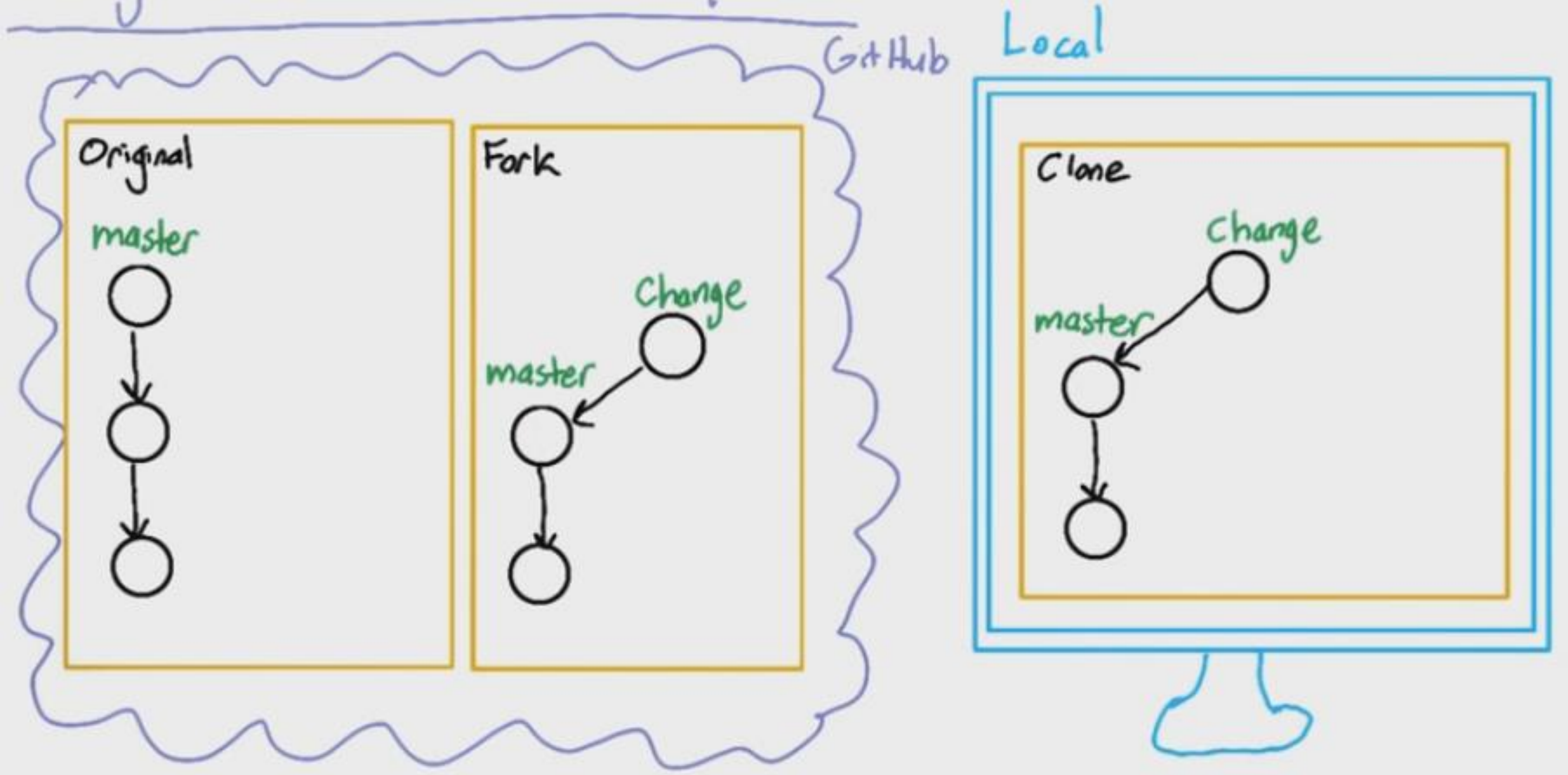
Check the commits that will be sent


Pulling Changes
local
lesson 1 reflections
update git usages
update multi-line
3a2
45c
3od
master
pull
master
GitHub
lesson 1 reflections
update git usages
update multi-line
first lesson 3 reflection
3a2
45c
3od
fad
master

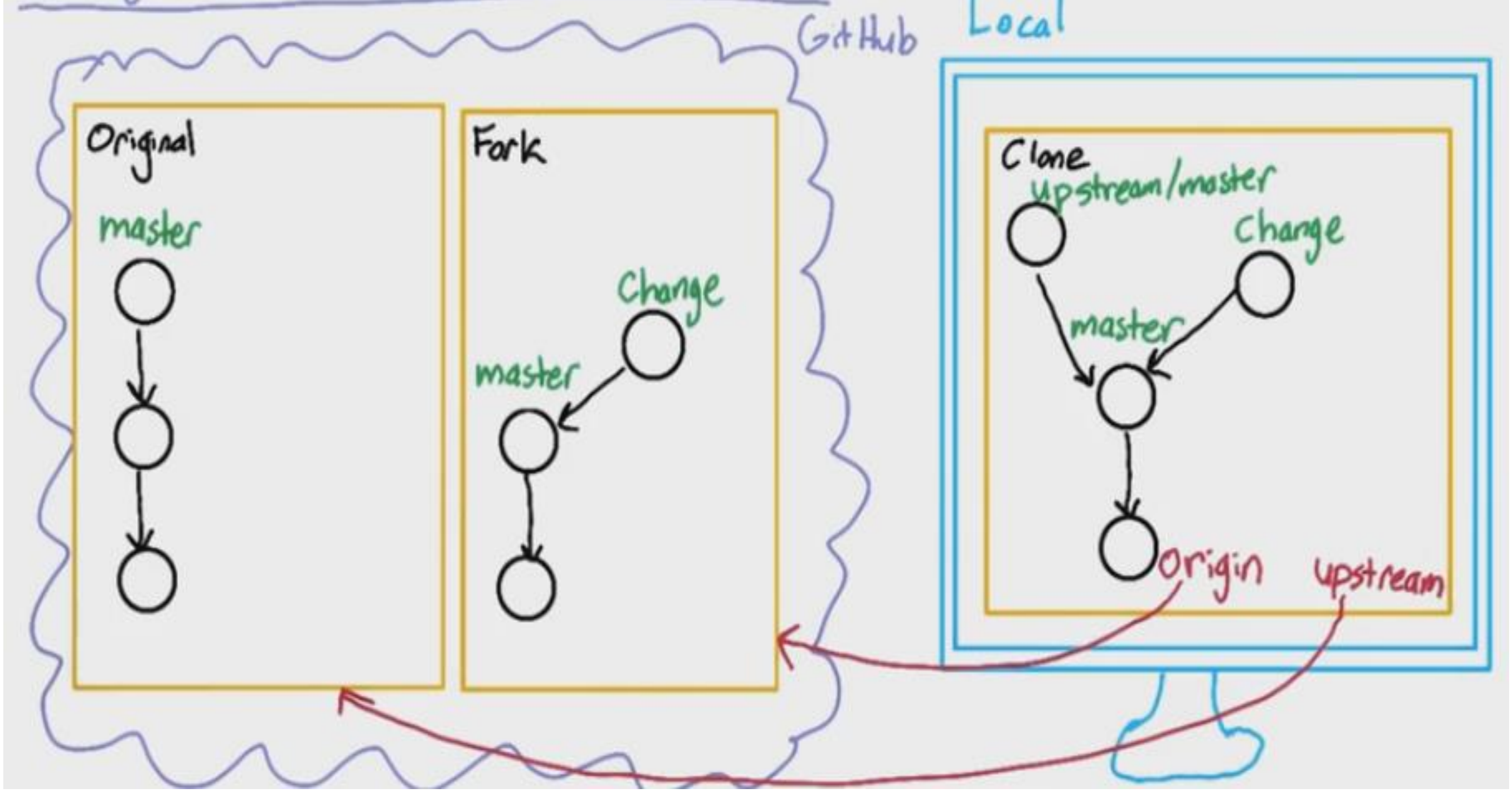
Clone, Branch, or Fork?

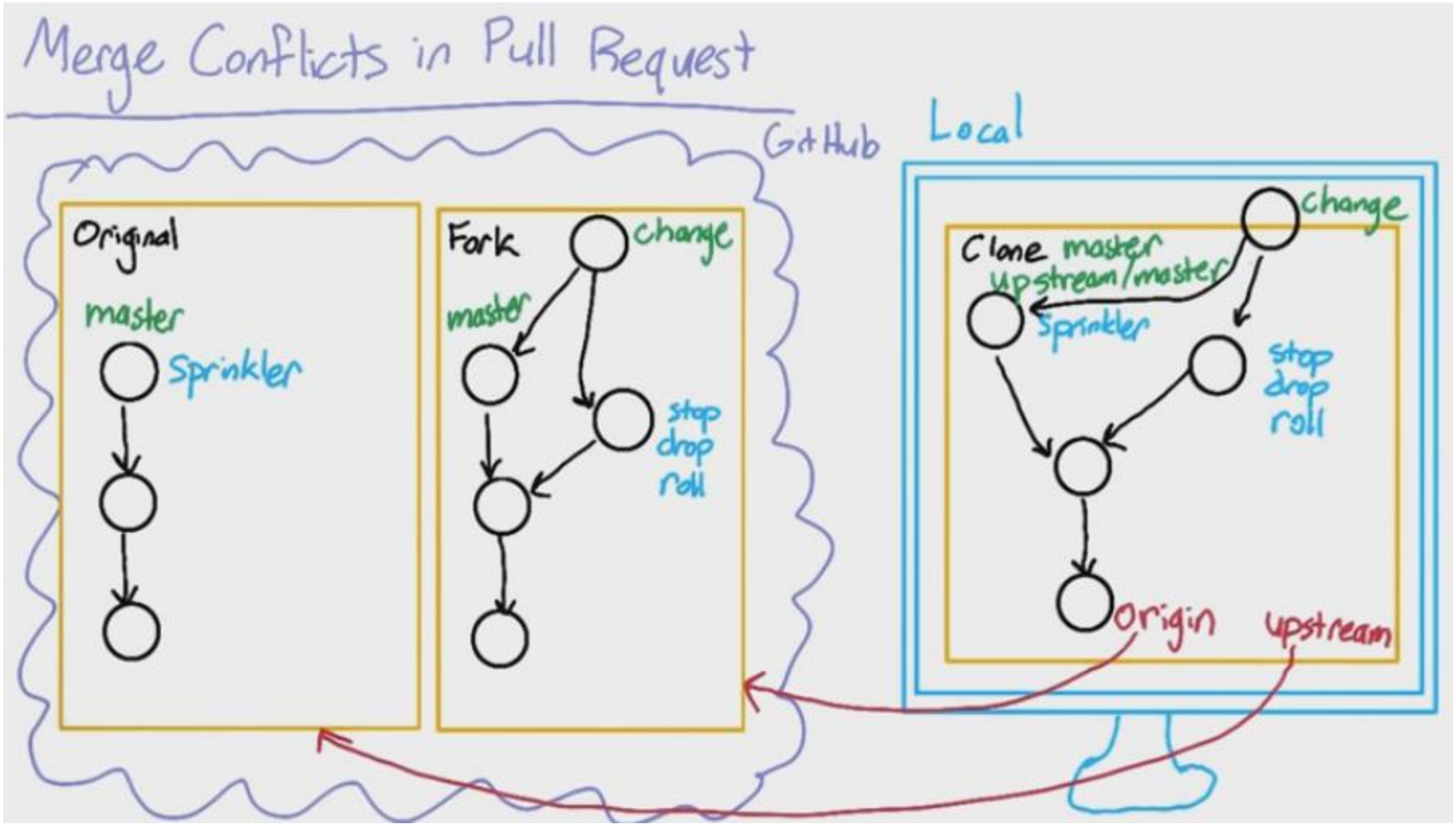


Merge Conflicts in Pull Request



Merge Conflicts in Pull Request




Merge Conflicts in Pull Request
GitHub
Local
Original
master
Sprinkler
Fork
change
master
stop drop roll
Clone
change
master
upstream/master
Sprinkler
stop drop roll
origin
upstream

Updating a Cross-Repository Pull Request

- Add the original repository as a remote in your clone
  - $git remote add upstream https://github/xxx/yyy.git
- Pull the *master* from the original repository
  - $git checkout master
  - $git pull upstream master
- Merge the *master* branch into your *change* branch
  - $git checkout stop-drop-roll
  - $git merge master stop-drop-roll
- Resolve any conflict, git add, then commit
- Push your *change* branch to your fork
  - $git push origin stop-drop-roll