Git & GitHub

Git

GitHub

In your project

Git != GitHub

- Git is a free and open source distributed version control system
  - Allows working locally
  - Stores your history
- GitHub company that provides hosting for software development version control using Git
  - It uses Git
  - It allows having a remote repository online
  - GitLab, BitBucket are similar services

GIT

GIT

- Distributed Version Control
- Entire Code is locally stored
  - With history
  - Users can make changes without being online

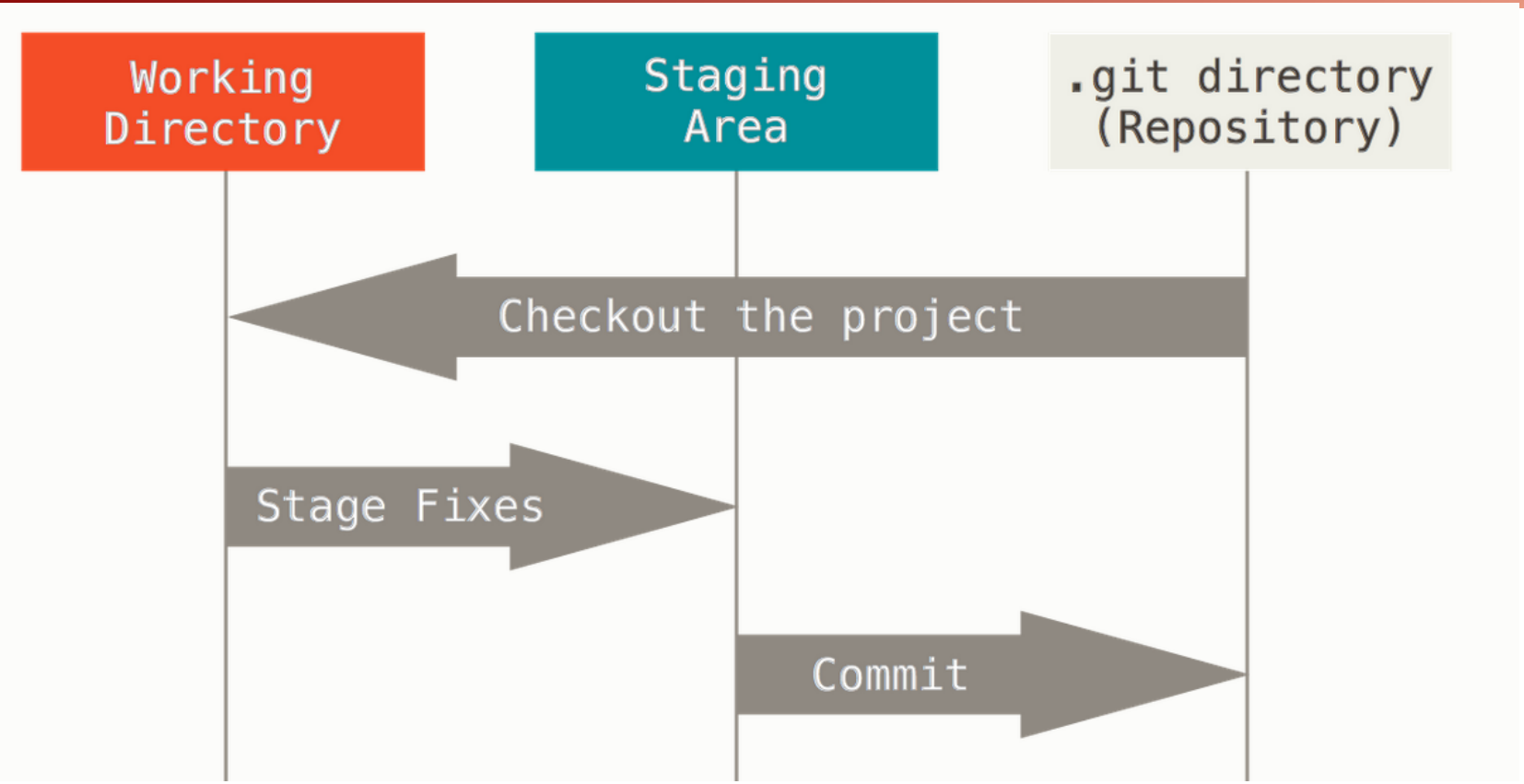
If used correctly you will never lose any of your work (stay tuned)

BASICS

- Git takes Snapshots of your work
  - **You** decide when this Snapshot is taken through **committing** your work
- You can go back and visit all of your Snapshots
  - Your newest Snapshots will still be there
- The Snapshots are all stored locally
  - In .git folder (hidden folder)

DIFFERENT STAGES & COMMITS

THREE STAGES OF GIT DIRECTORY



- Single checkout of one version
- Pulled from database of Git directory
- Local copy to work with

- Stores what will go into next commit
- Also called Index

- Stores metadata and object database
- Most important part

COMMON WORKFLOW

Step 1

- Modify files

- Working on your local files

Step 2

- Stage files
- Adding a snapshot of them to staging area

- **git add file** (stages file)
- **git add .** (stages everything modified or not in index yet)

Step 3

- Commit
- Taking the files as they are in staging areas and stores permanently in Git directory

- git commit (commits staged files)

WHY STAGE AND COMMIT?

To group your Snapshots

- Grouping is very important to maintain a good version control

- Commit messages help with explaining what you did

WHY STAGE?

Group your commits in a way that they make sense

- The changes belong together, e.g. one bug fix, updated README, one feature
- Write good commit messages for each commit
- Split your work into different commits through staging
  - Use ”add <all files belonging together>”
  - Then commit
  - Use “add <all other files belonging together>”
  - Then commit
- Important to maintain your repo
  - Understand what happened
  - Be able to revert changes/browser history

Hidden in video, since explained on the previous slide in video.

BASIC COMMANDS
STAGING AND COMMITTING
BROWSING HISTORY

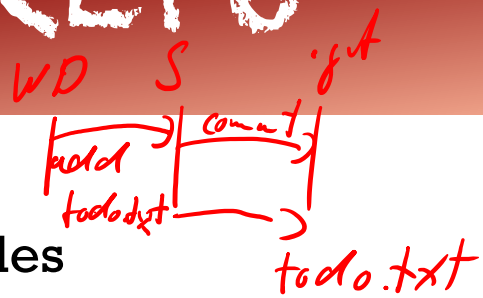
GIT INIT

```
Last login: Tue Aug  6 11:07:06 on ttys003
[12:05:34] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git init
Initialized empty Git repository in /Users/amehlhas/ASU/Alex/2019-Fall/GitTest/.git/
```

- Creates a new subdirectory named .git (hidden folder)
- Nothing is tracked yet (neither staged nor committed)
- .git contains your log, your history, etc. ➔ It is your repository

NEVER EVER DELETE THIS FOLDER

ADD FILES TO YOUR REPO



git add and **git commit** lets you start tracking your files

```
[12:39:05] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git add todo.txt
[12:39:36] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git commit -m "Add   todo list"
[master 91c298d] Added todo list
 1 file changed, 3 insertions(+)
 create mode 100644 todo.txt
```

Explanation

- Add todo.tex file
- Commit all staged files and add a commit message
  - -m states you want to write a commit message for your commit
  - 'Add todo list' is your commit message
- Tells you to which branch the change was made to and if something was added/deleted

SKIPPING THE STAGING

git commit –a stages and commits right away (use when only one like commit message)

```
[12:51:08] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git commit -a -m "Added task to todo list"
[master 9b09c6b] Added task to todo list
 1 file changed, 2 insertions(+), 1 deletion(-)
[12:51:37] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git log
commit 9b09c6b09e04a25cd66e8dfe0da504e7813181cd (HEAD -> master)
Author: amehlhase316 <amehlhase316@asu.edu>
Date:   Tue Aug 6 12:51:37 2019 -0700

    Add   task to todo list
```

Explanation: One file was staged and committed without explicit "add"

- Useful if you only changed files that belong together
- Should not be the default

STAGING ALL FILES

git add . stages all files that were modified

- Not good practice for grouping commits
- Not good practice since temporary files might be staged
  - To avoid this use .gitignore (stay tuned)

git add DirectoryName/

- Stages the directory DirectoryName with all files from that directory

git add Filename

- Stages the file with the name Filename

BROWSING THE HISTORY

git checkout <commit number>

- Lets you checkout a specific commit

git checkout branchname

- Lets you checkout latest branch version

CHECKING THINGS

Hidden in video, since explained on the previous slide in video.

CHECKING CURRENT STATUS

git status lets you check the current state of your repo

```
[12:59:36] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   CoolFile.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   todo.txt

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

        .DS_Store
```

Explanation: lets you see which files changed, which ones are staged and which are not tracked at all.

Hidden in video, since explained on the previous slide in video.

GIT LOG

git log shows you what happened on your branch so far

```
[12:43:09] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git log
commit 91c298d30eac1f43f7db53e969c1775802e7766c (HEAD -> master)
Author: amehlhase316 <amehlhase316@asu.edu>
Date:   Tue Aug 6 12:39:46 2019 -0700

    Add   todo list
```

Explanation: We see one commit with following information

- Each commit has a unique identifier (hash)
  - This id can be used to browse different versions
- Author: who committed the change
  - Traceability
- Date: when was this committed
  - Traceability
- Commit message specifying what changed
  - Traceability

Hidden in video: explained on the previous slide in video.

BASIC COMMANDS
ONLY NEEDED FILES IN REPO

REMOVING FILES

- **git rm [filename]** removes file from local directory and from index
- **git rm –cached [filename]** removes file from index only
- **git rm –r [directory]** removes the directory from local directory and from index recursively

.GITIGNORE

.gitignore file is stored in base directory of repository (where .git folder is located) and specifies which files should not be version controlled

- Can exclude:
  - explicit files (e.g. myFile.txt)
  - Can exclude file extensions (e.g. *.aux)
  - Can exclude whole directories (e.g. build/, bin/)
- Important for SE projects to only include what is needed

```
# Compiled class file
*.class

*.classpath
*.project
local.properties
.settings/
bin/
build/
```

HOUSEKEEPING

GOOD COMMIT MESSAGES

- Well crafted commit message great way to communicate with team
- Some rules:
  - Separate subject from body with a blank line
    - Limit the subject line to 50 characters
    - Capitalize the subject line
    - Do not end the subject line with a period
    - Use the imperative mood in the subject line
  - Wrap the body at 72 characters
    - Use the body to explain *what* and *why* vs. *how*

EXAMPLES FOR COMMIT MESSAGE

A properly formed Git commit subject line should always be able to complete the following sentence:

- If applied, this commit will *your subject line here*

Examples

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*

https://chris.beams.io/posts/git-commit/

FILES IN YOUR REPO

- Only have needed files in your repo (make sure you use a good .gitignore)
- Do not include
  - Temporary files
  - System/Tool specific files
- Take frequent snapshots

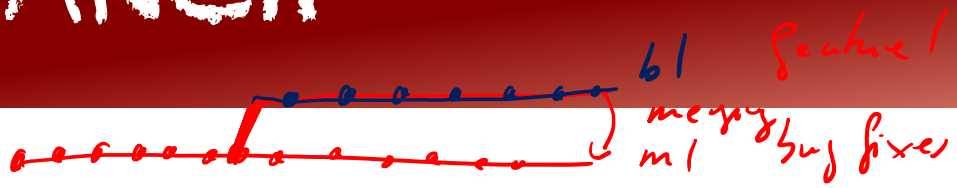
SUMMARY BASIC COMMANDS

- You can create a local repo through **git init**
- You can stage (group) files **git add filename**
- You can remove files through **git rm –cached**
- You can browse history **git checkout <commit number>**
- You can commit changes to your local repo **git commit –m "commit message"**
- You can look at your log **git log**
- Check the current status of your repo **git status**

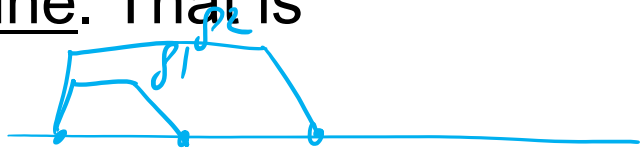
BRANCHES

Some images are taken from: https://git-scm.com/book/en/v2

BRANCH

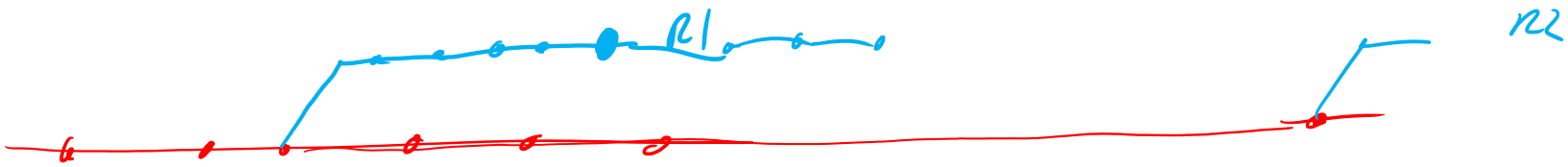


A branch is a named variant of a codeline. That is

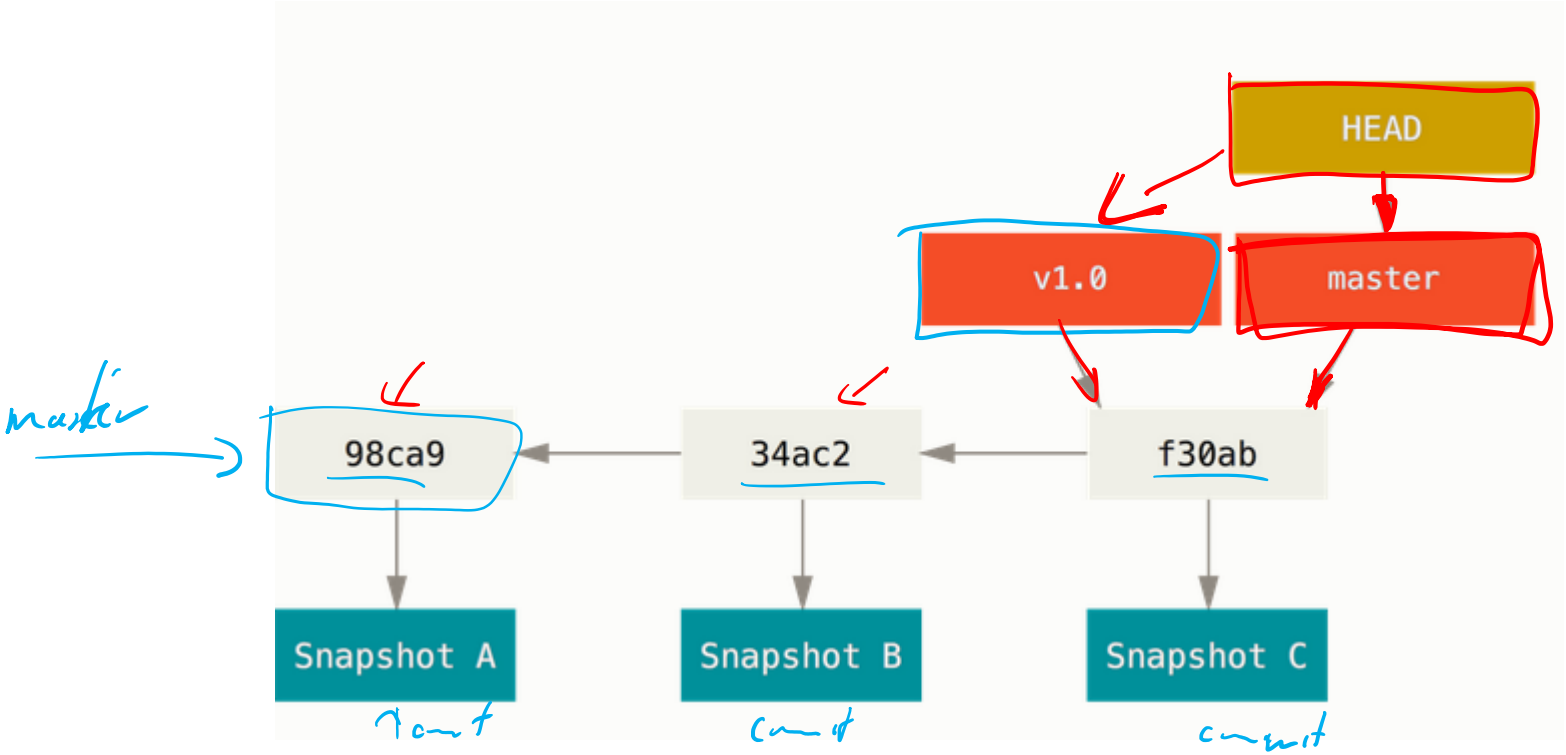


- A collection of software artifacts
- Assigned a logical identifier
- Whose purpose is to be either folded back into the main codeline, or maintained as a release
- Git creates one branch at the beginning: **master**

```
[12:39:05] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git add todo.txt
[12:39:36] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git commit -m "Added todo list"
[master 91c298d] Added todo list
 1 file changed, 3 insertions(+)
 create mode 100644 todo.txt
```



A BRANCH

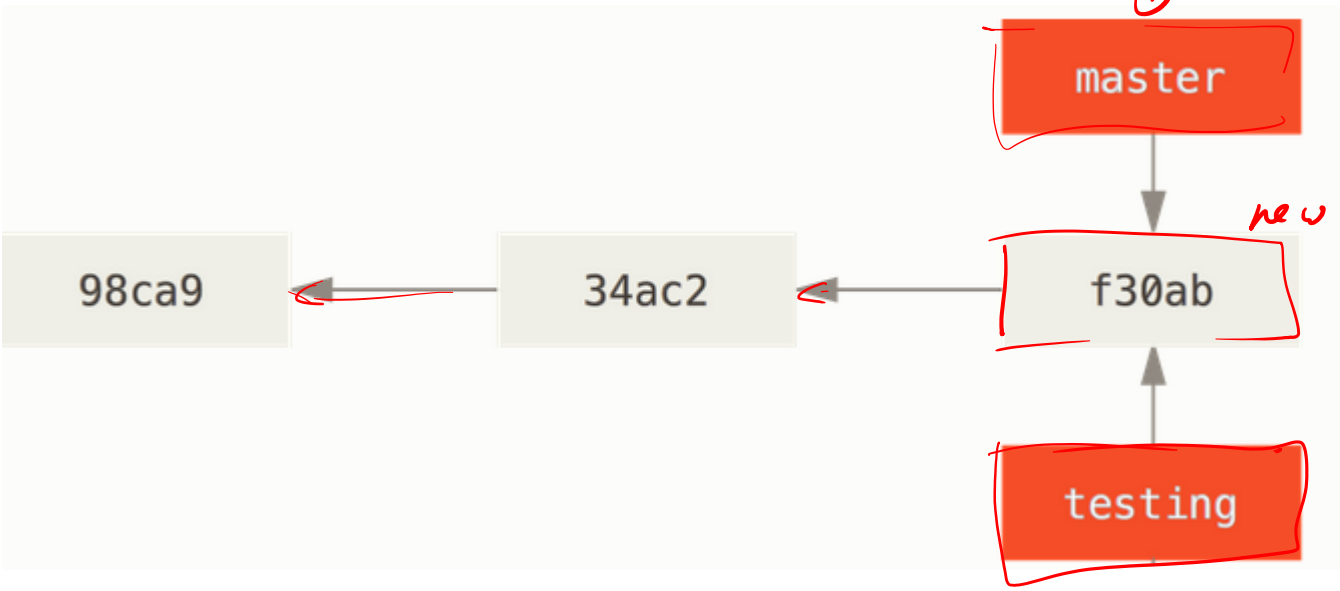


- Branch is pointer to a commit
- Default Branch is master

CREATING A NEW BRANCH

```
[12:59:38] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git branch testing
```

Explanation: Creating a new pointer where HEAD is at the moment

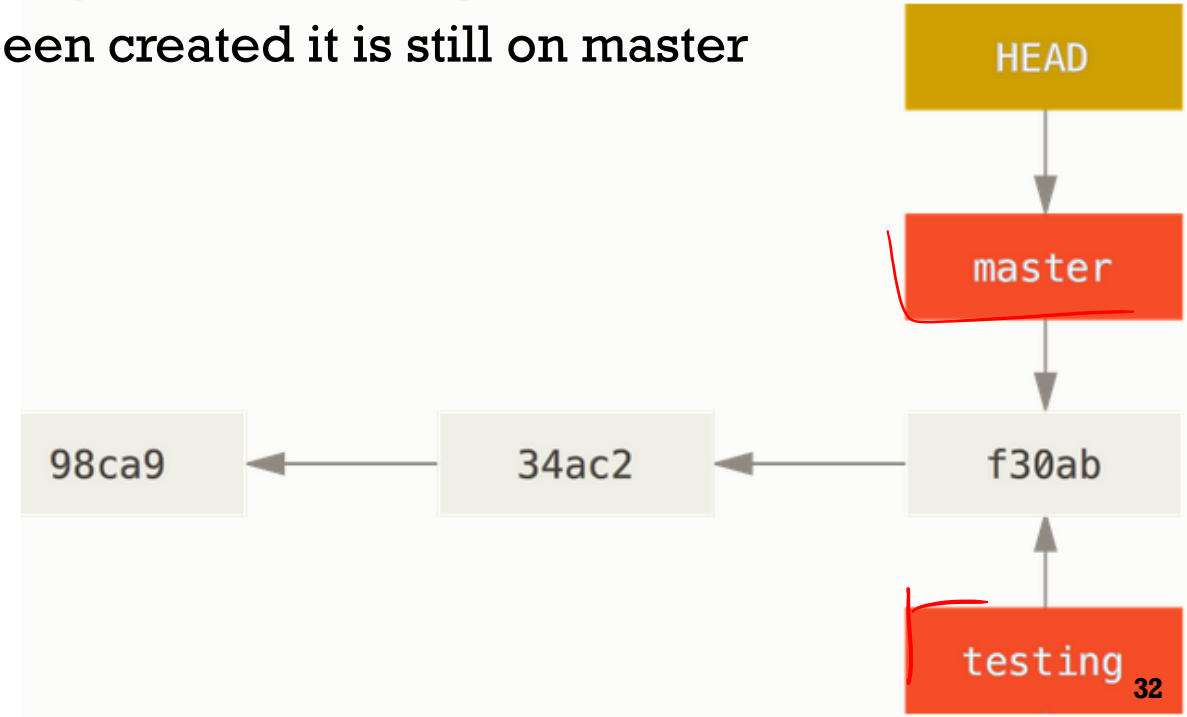


THE HEAD

```
[12:59:38] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git branch testing
[13:12:18] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git branch
* master
  testing
```

Explanation:

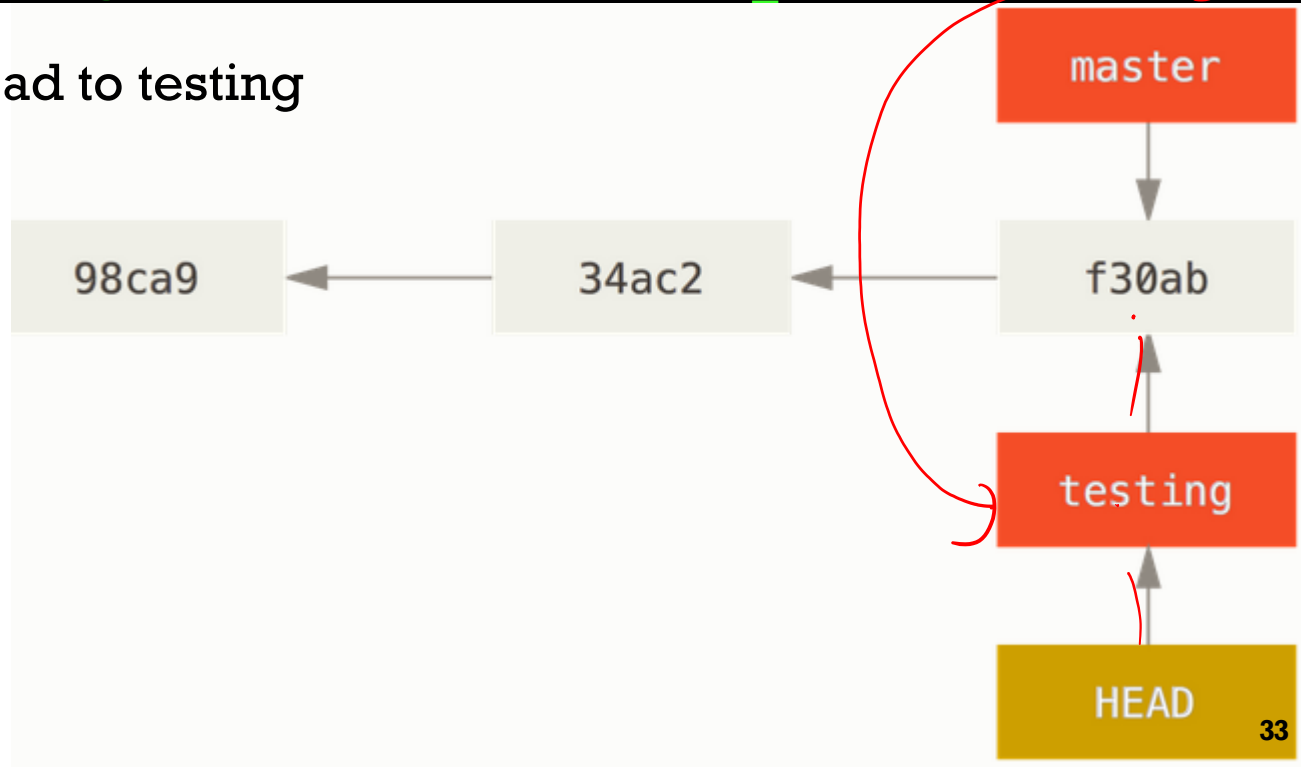
- Head shows which Branch you are currently on and which commit
- Since Branch has only been created it is still on master



SWITCHING BRANCHES

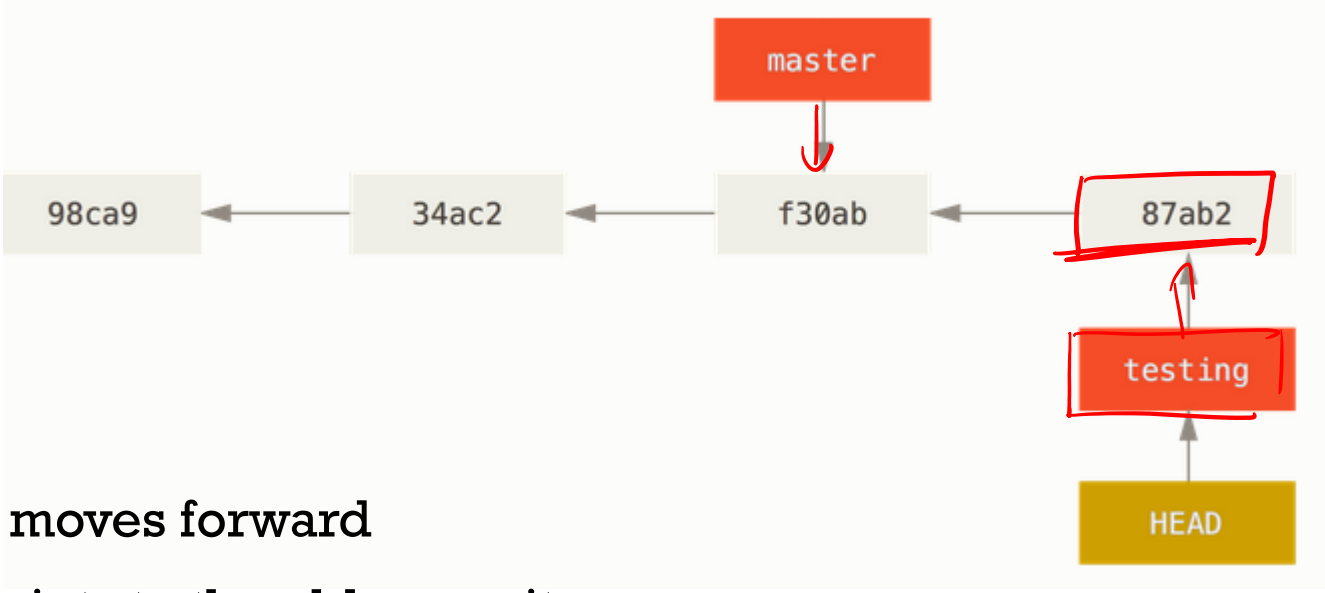
```
[12:59:38] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git branch testing
[13:12:18] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git branch
* master
  testing
[13:13:21] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git checkout testing
M       CoolFile.txt
M       todo.txt
Switched to branch 'testing'
```

Explanation: Moves Head to testing



MAKING COMMITS ON BRANCH

```
[13:17:16] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git commit -a -m "Cool feature added"
[testing 86d2f42] Cool feature added
 2 files changed, 5 insertions(+), 1 deletion(-)
```

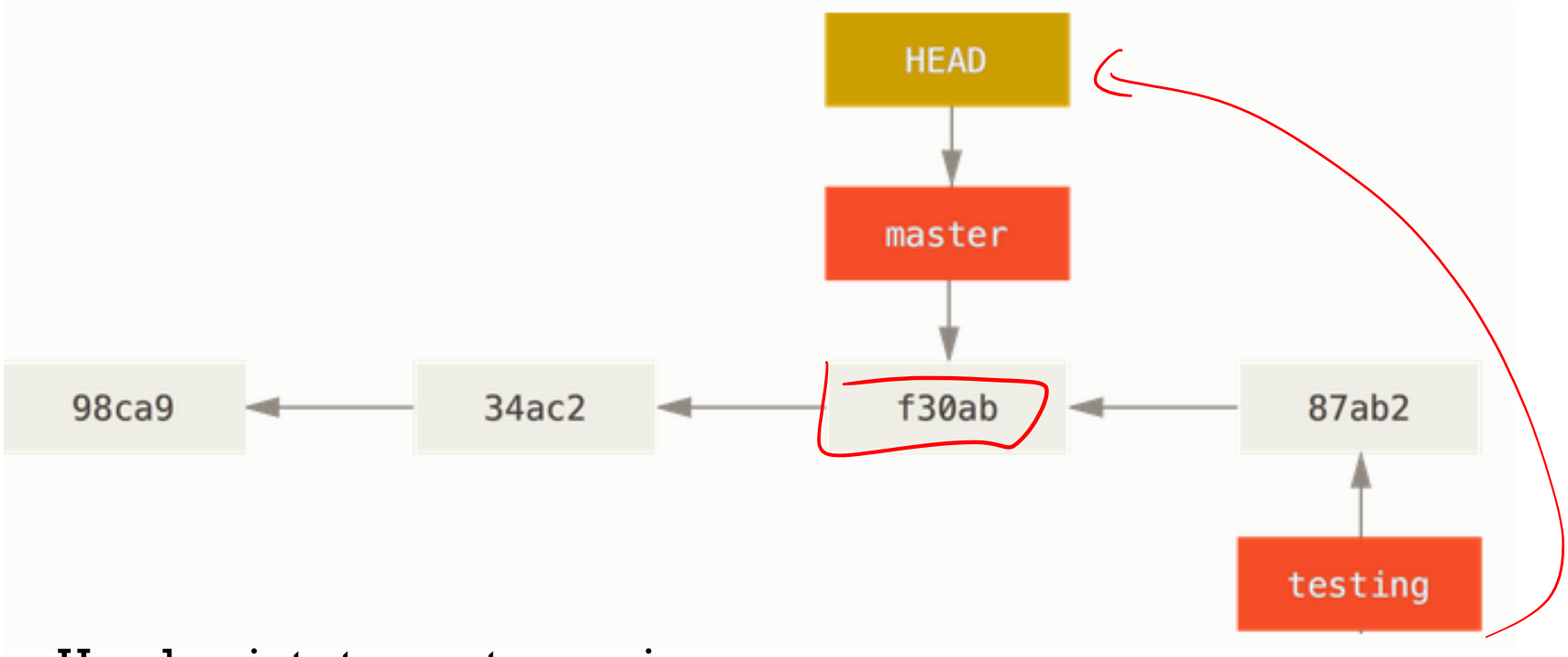


Explanation:

- Head Branch moves forward
- Master still points to the old commit
- You commit on the branch you are currently on (check through **git branch**)

SWITCHING BACK TO MASTER

```
[13:17:50] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git checkout master
Switched to branch 'master'
```

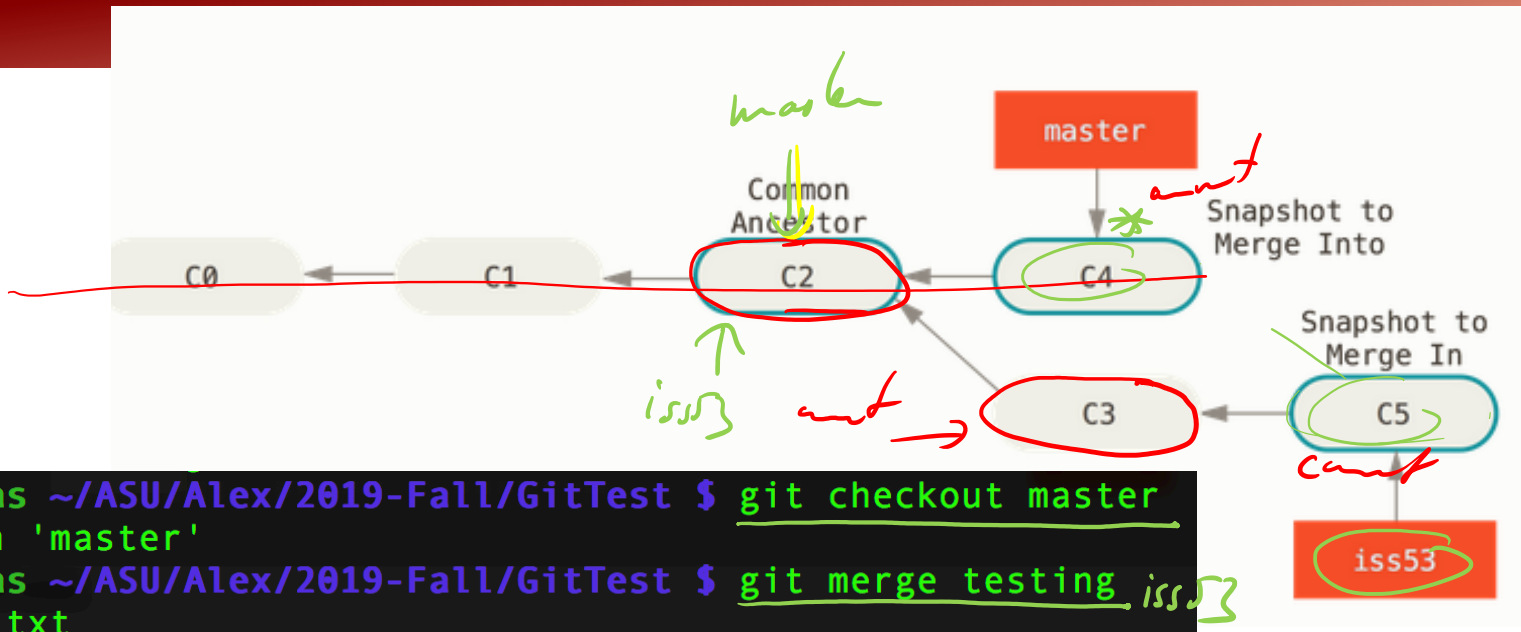


- Head points to master again
- Files in your working directory back to master snapshot

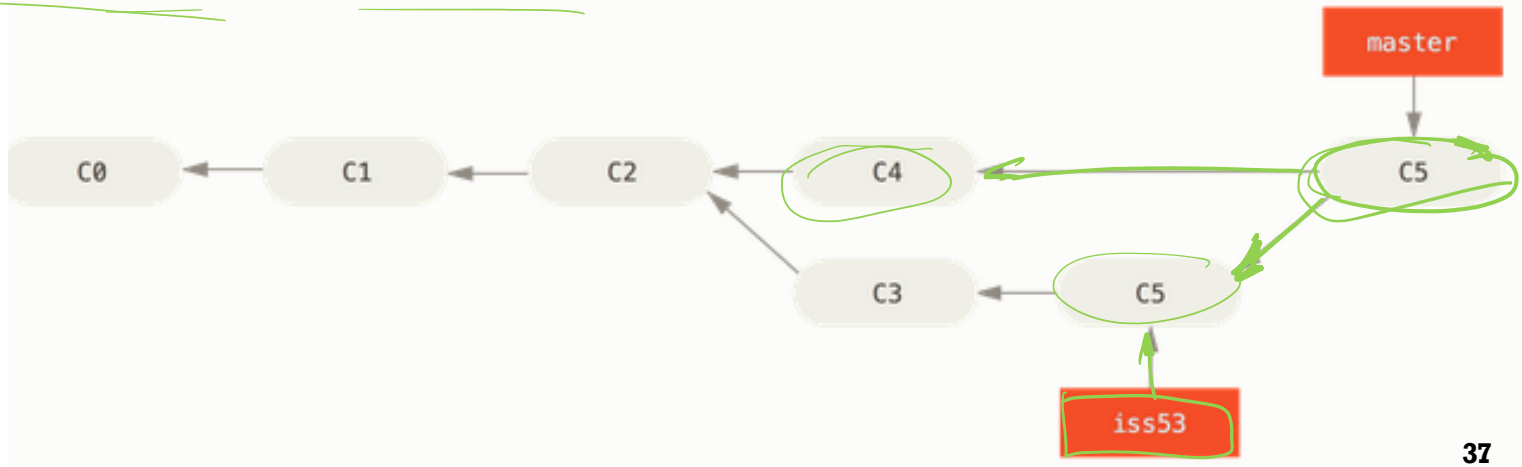
Will include a demo in command line

MERGING BRANCHES

BASIC MERGING




```
[13:26:20] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git checkout master
Switched to branch 'master'
[13:26:24] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git merge testing
Auto-merging todo.txt
Merge made by the 'recursive' strategy.
 todo.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```



CONFLICTS

- If Git cannot merge you will get a conflict

```
[13:21:13] amehlhas ~/ASU/Alex/2019-Fall/GitTest $ git merge testing
Auto-merging todo.txt
CONFLICT (content): Merge conflict in todo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

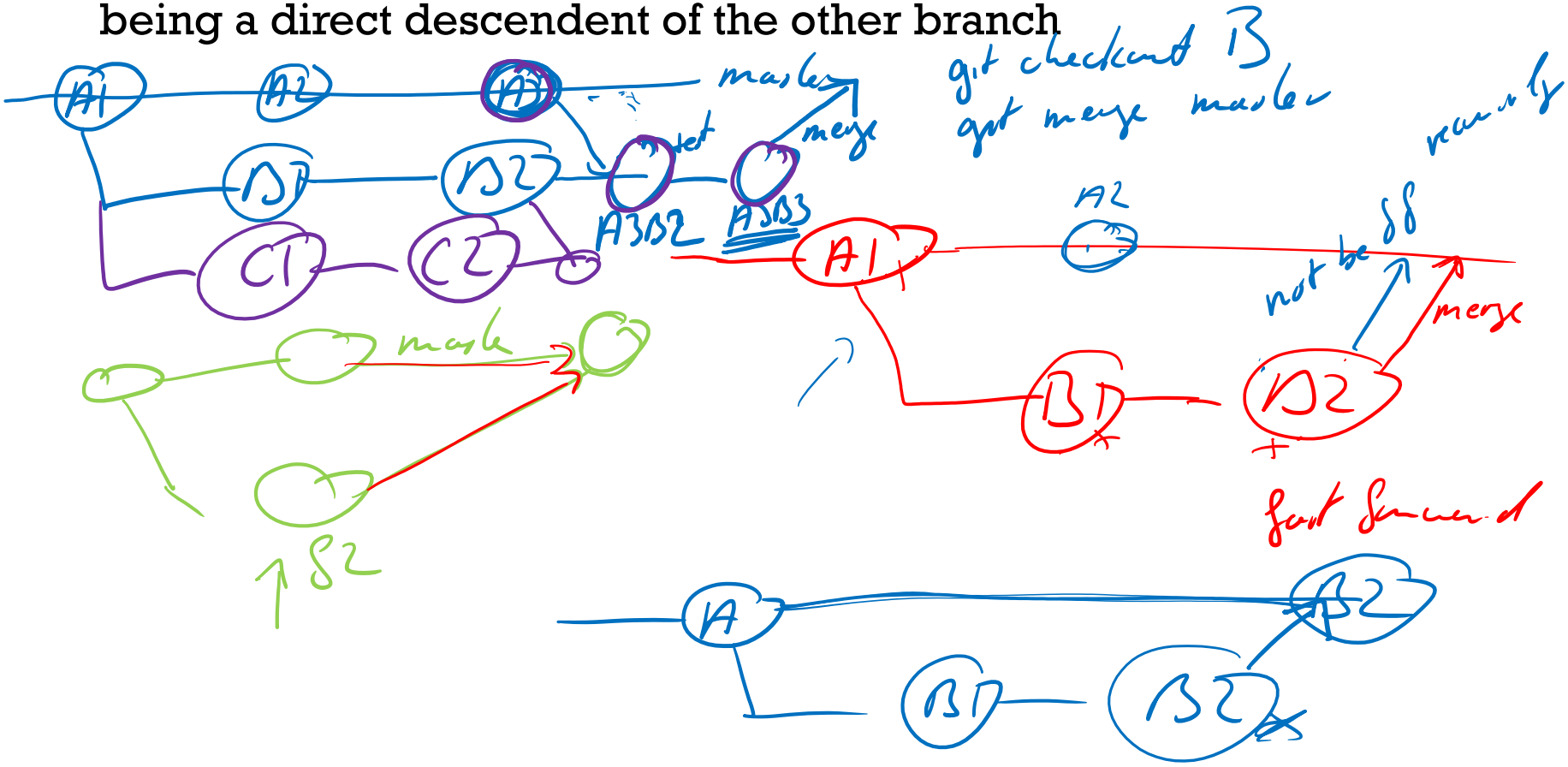
Explanation: Git tells you that it cannot figure out how to merge your files

- Options:
  - Abort **git merge –abort**
  - Go into the files and manually fix the file(s) – see demo

Will include a demo in command line

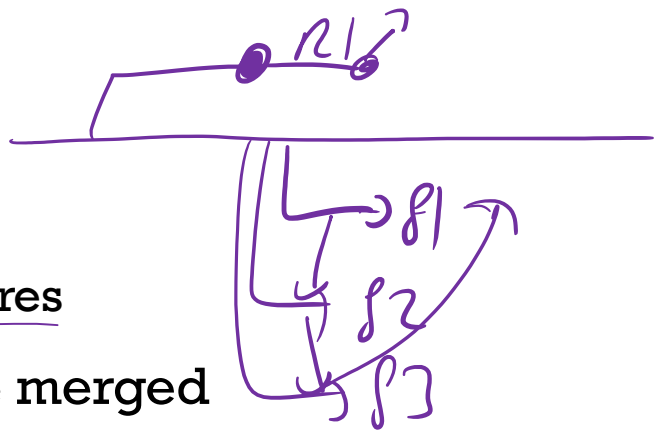
FAST FORWARD

- In case a merge can be done without conflict due to the branch being a direct descendent of the other branch



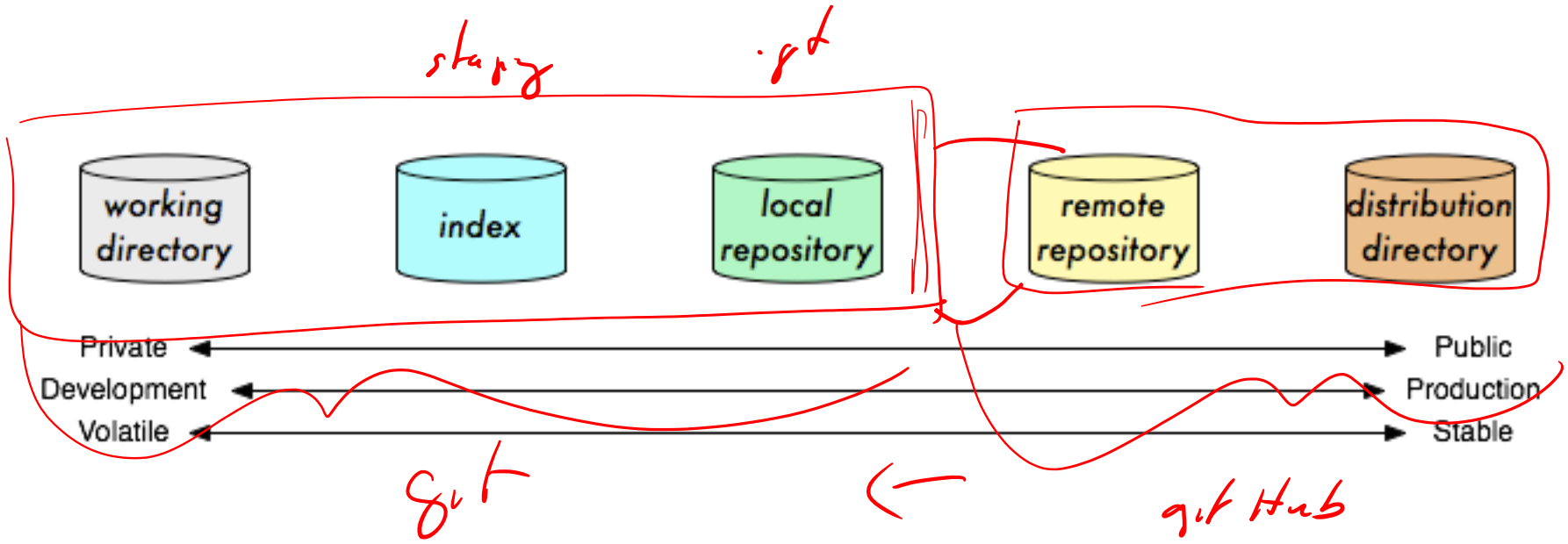
SUMMARY

- Default branch is master
- Branches are a separate Code line
- Important for
  - teams
  - releases
  - to try out new ideas
  - to work on separate features
- Branches are meant to be merged



GITHUB

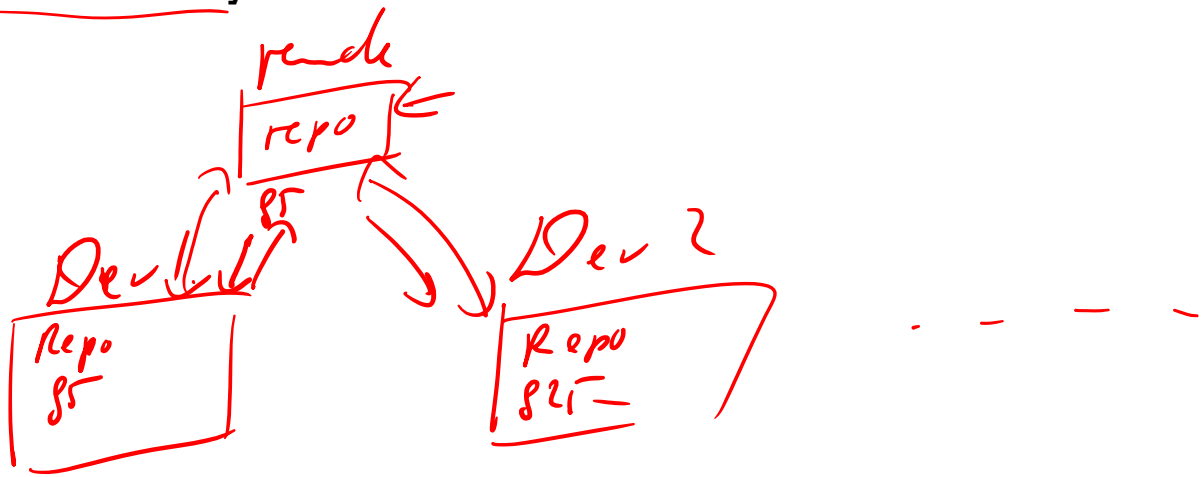
WORKING WITH GIT



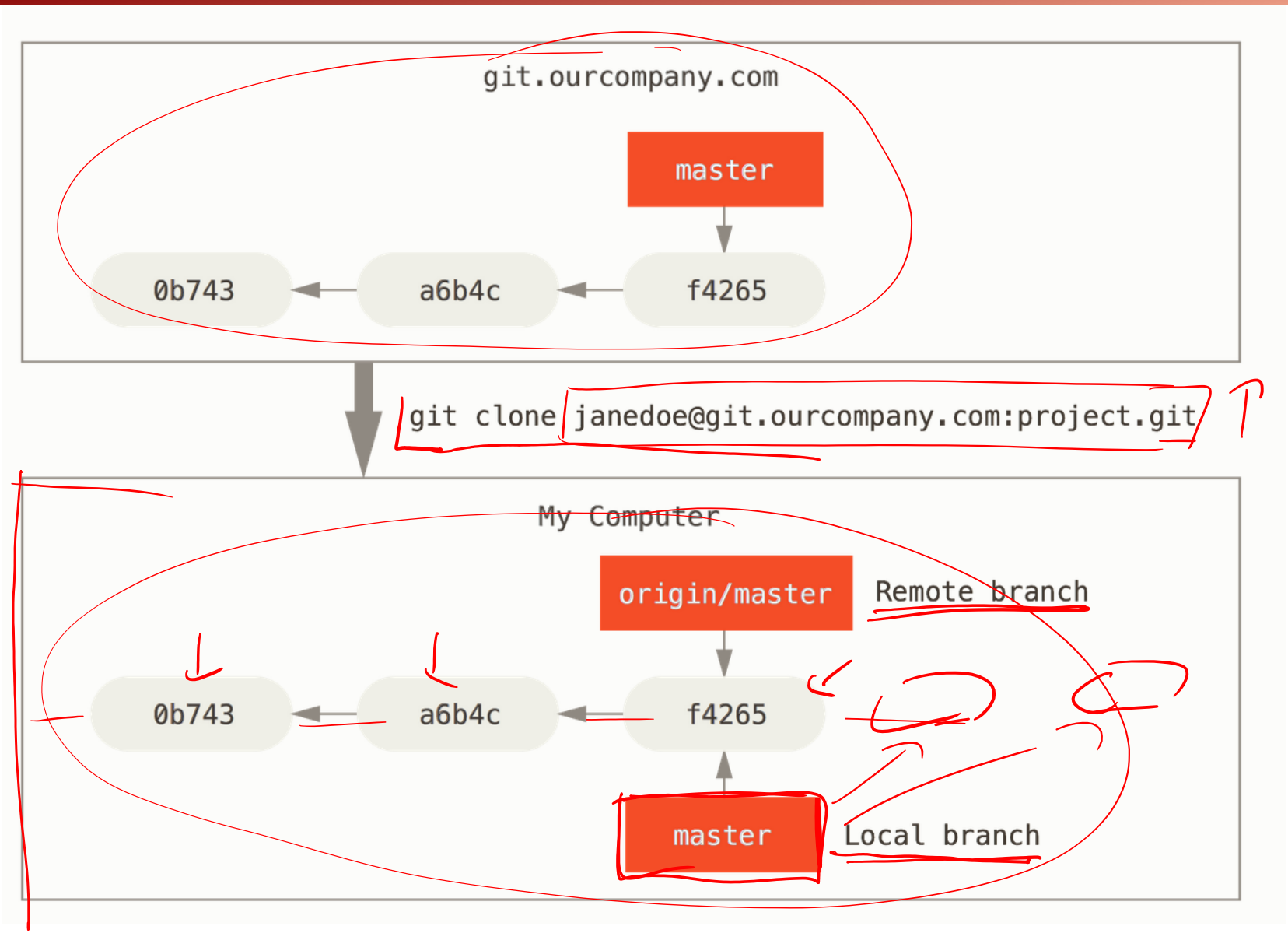
- Git is as effective locally as remotely, perhaps even moreso
- Do not confuse "Git" and "Github"
  - Github is a remote repository in the above diagram, with some value-added tools and services

BASIC IDEA IS

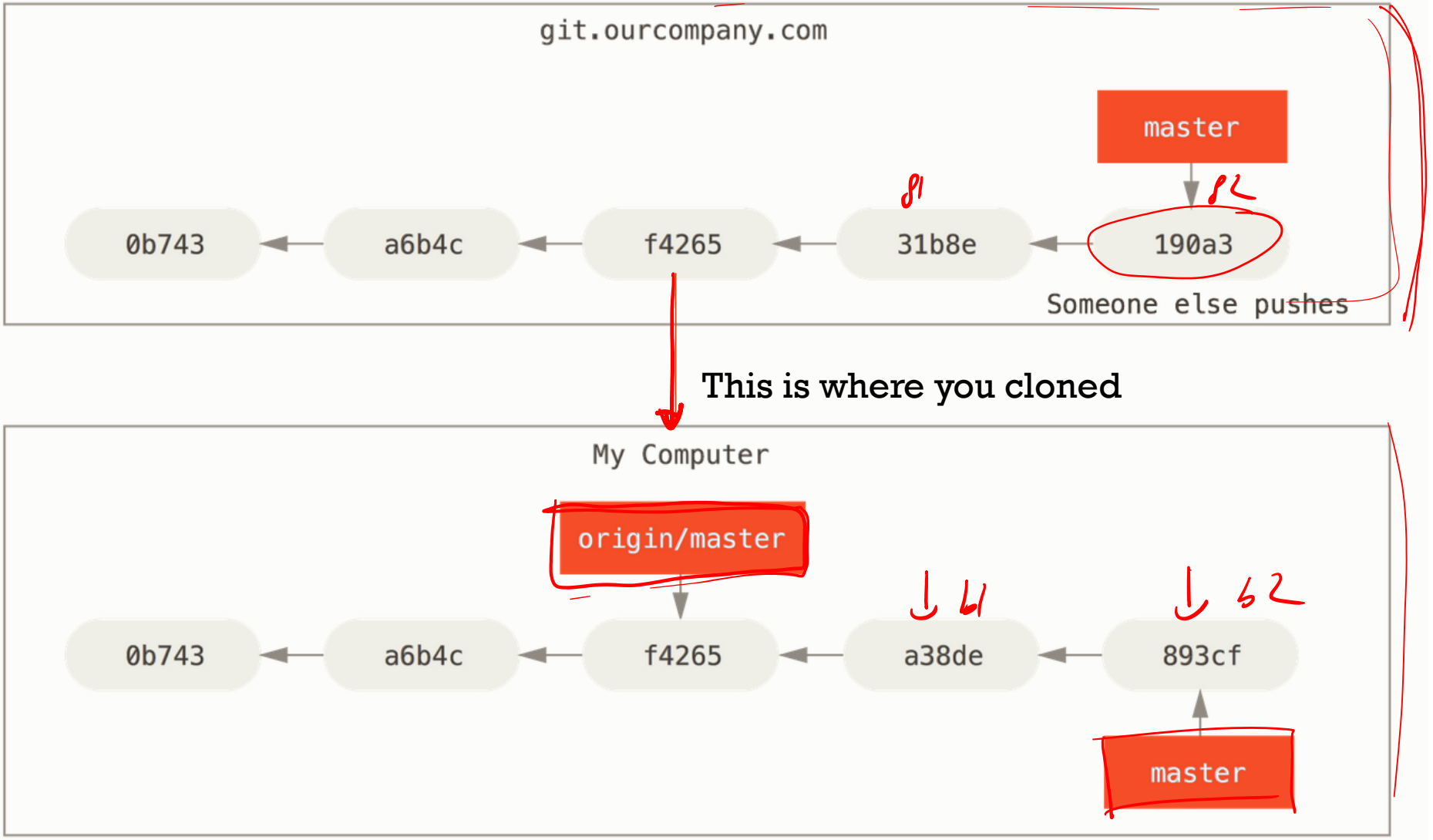
- To work on things locally and push to remote to store remotely
- Work usually happens in branches (not on master)
- Team works together on one Software project on same remote repository
  - Everyone has access to all branches
  - Everyone can test and see coding effort
  - Traceability



WORKING WITH A REMOTE REPO (E.G. GITHUB)

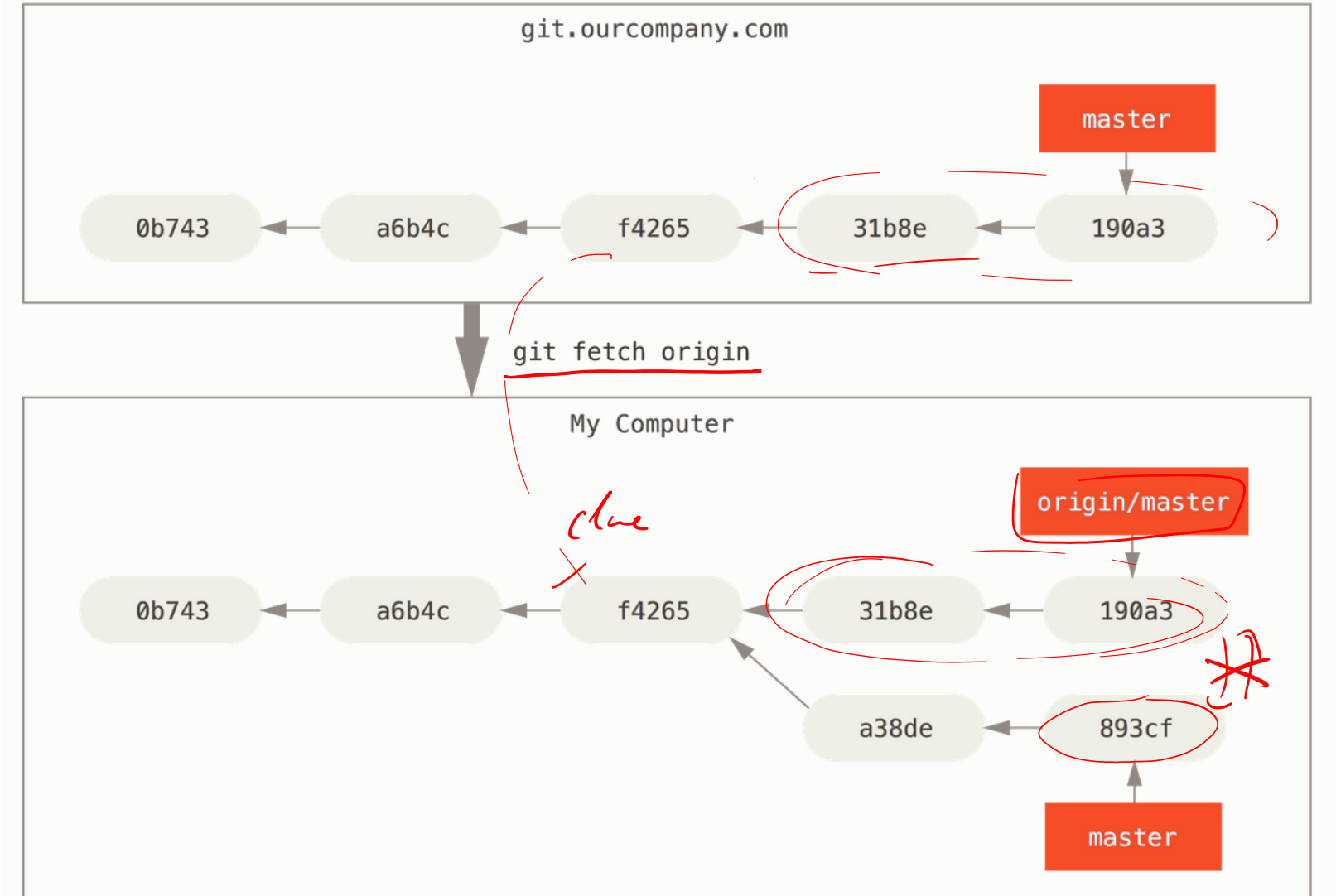


CHANGES IN THE REMOTE AND LOCAL REPO



UPDATING YOUR ORIGIN/MASTER

- You will then have two branches (origin/master and master) in your local Git
- You will need to merge later



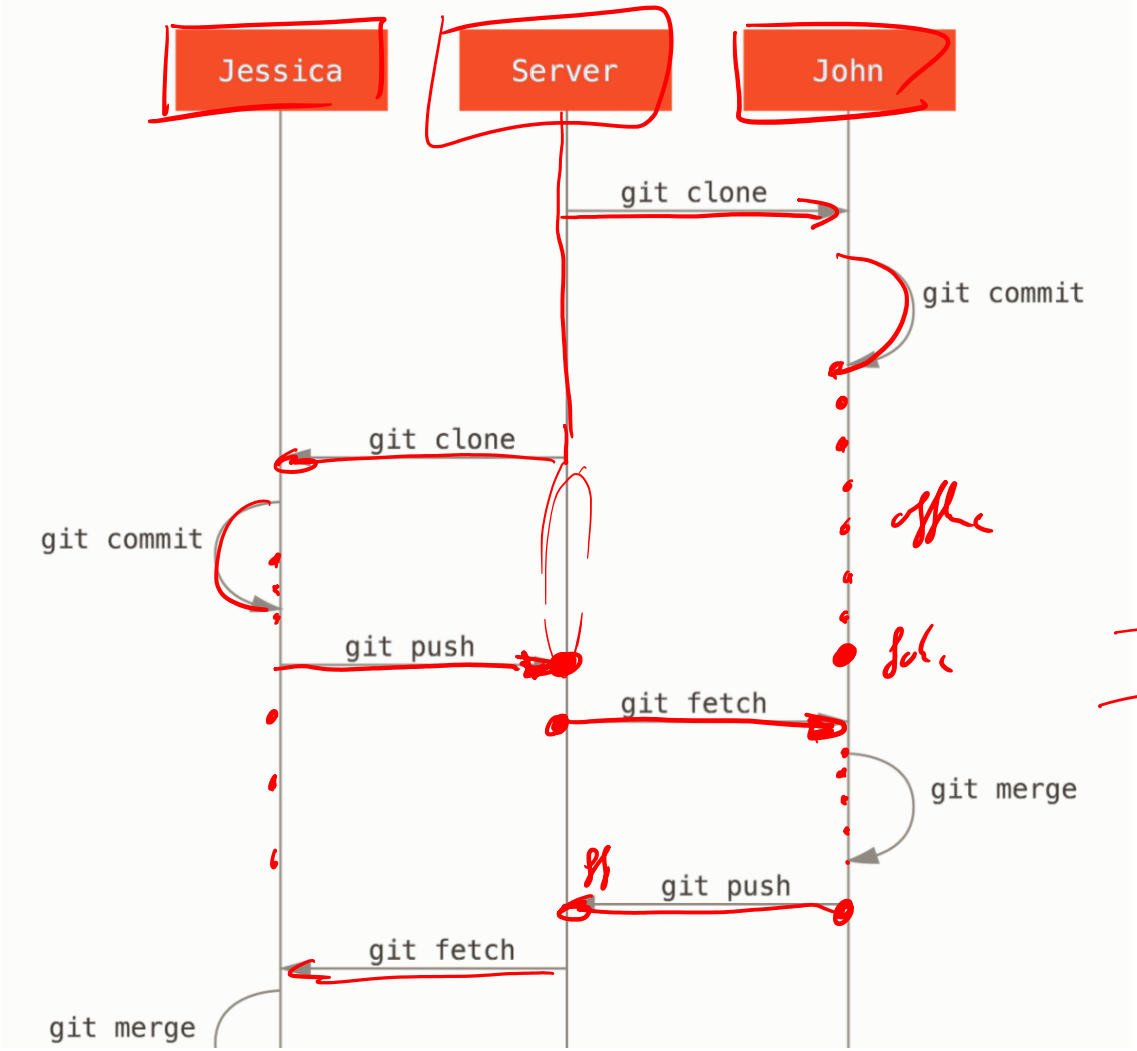
GIT FETCH & PULL

- **Git fetch fetches remote branch into new local branch**
  - New branch is origin/branchname
  - Needs to be merged locally
- **Git pull** fetches and merges remote branch into your local branch
  - Might be without conflicts
  - Might have conflicts which you need to fix manually

BRANCH, FETCH, MERGE ETC.

Read: Contributing to a Project

https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project



SUMMARY

- Remote repo lets team work together
- Usually team has different branches
- Main commands
  - Clone: clones the remote repo (done once when starting to work on repo)
  - Fetch: fetches a remote branch and creates origin/branchname branch locally
  - Pull: fetches and merges remote branch into local branch
  - Push: Pushes changes from local branch to remote branch
    - Make sure you resolve conflicts locally first

SOME EXAMPLES FOR WORKFLOWS

GIT WORKFLOWS

- As a content-addressable filesystem, Git doesn't prescribe the way in which you manage objects
- While others SCMs allow you to vary the workflow, the predominant is branch-&-merge
- Nothing in Git enforces a workflow, but agreeing on a workflow is a best practice, otherwise chaos ensues!

ONLY ONE BRANCH

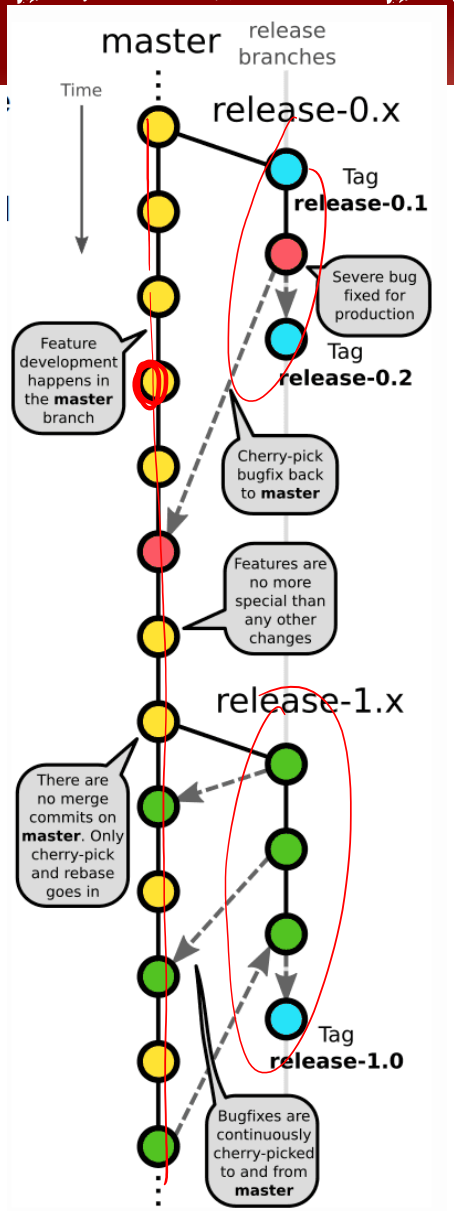
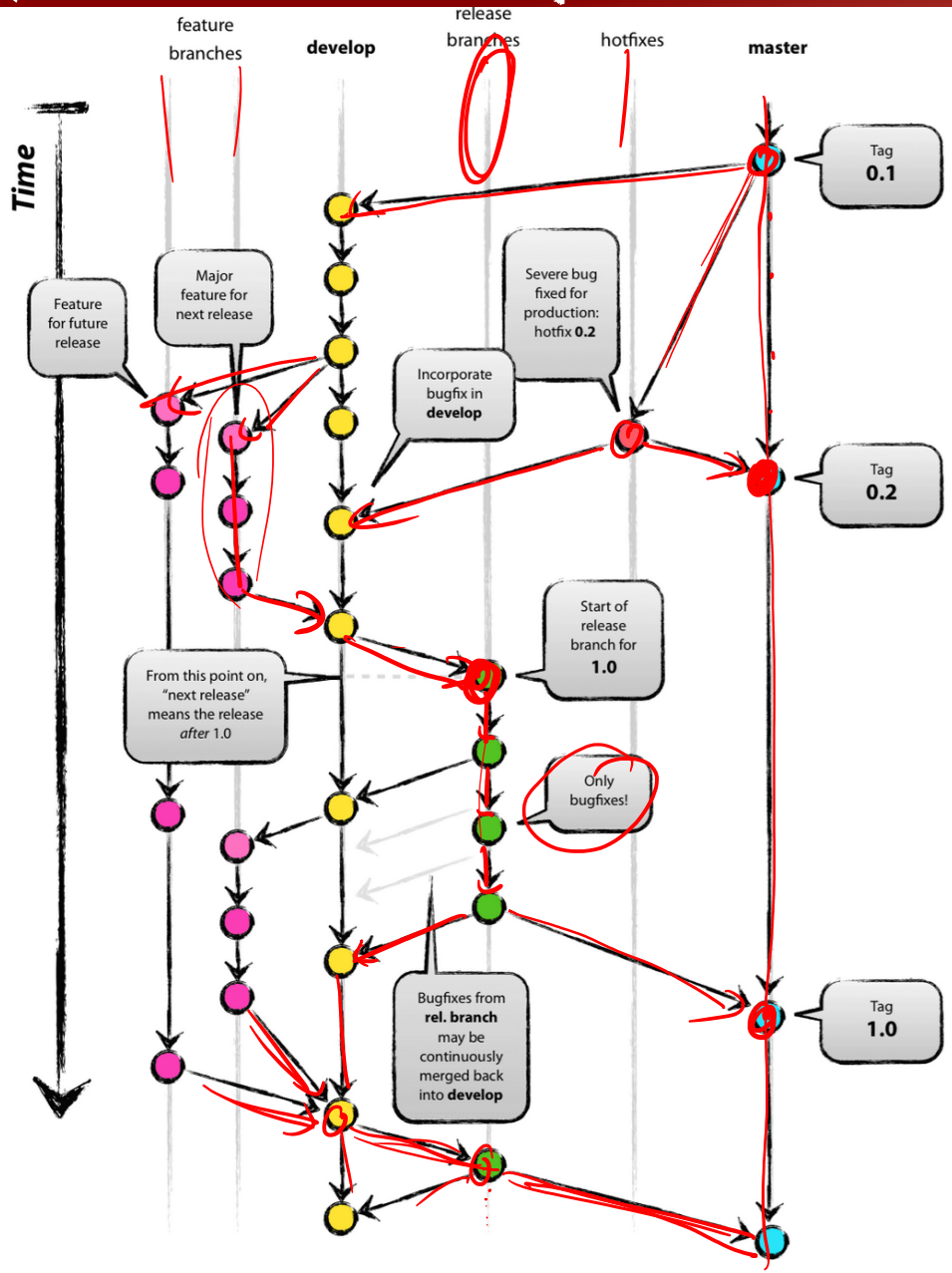


Figure 1: The cactus model

- Everything is directly committed to master
- You can still use branches for features
  - But usually only locally
- Argument here is: that master is for development and should be used a such:
- https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/

MANY BRANCHES ('OPPOSITE')



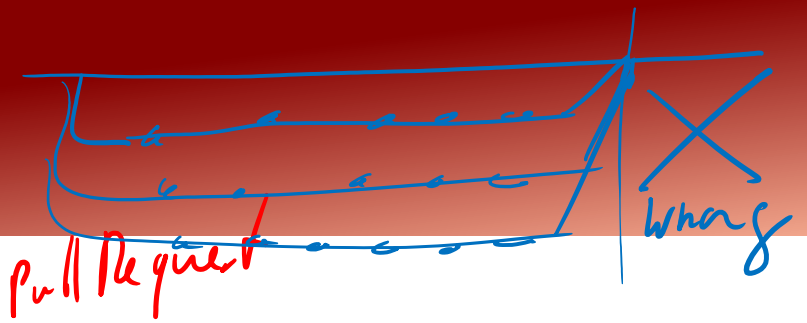
- Much more complicated with many branches
- Two main branches (master, development)
  - Master is always deployable
  - Develop is always a working version
- Other branches are for development, testing, bug fixing etc.
- This model is actually mentioned as bad in the former article (see previous slide)

- More complicated to work with but Master is always stable!

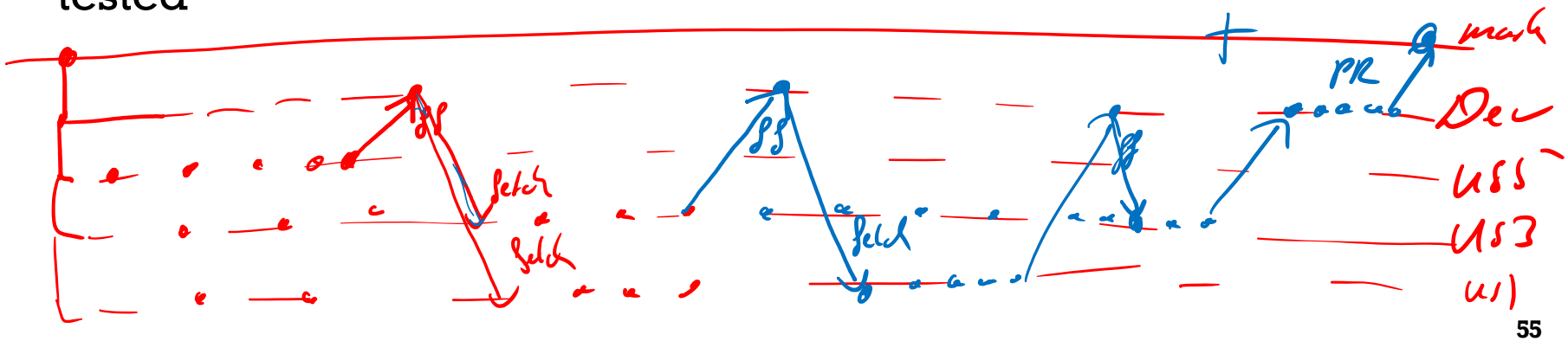
http://nvie.com/posts/a-successful-git-branching-model/

IN YOUR PROJECT

BRANCHES



- Master branch is your stable release branch (I will protect the master)
- Development branch is another stable branch which should always have the newest working/tested copy of your software (I will protect the Dev if you want to)
- One branch per User Story names US#-ShortDescription (e.g. US5-FixPreferences)
  - You can have task branches if you like
- US branches should be merged into DEV when completed and tested



WORKING ON YOUR BRANCH

- Commit often (remember if you do not commit it is not stored on Git)
  - Group your commits so they make sense
  - Write good commit messages
- Push often (so it is also on the remote branch)
- If another person is working on the same branch fetch (if you want to check the differences first) or pull into your local branch before starting to work
- If Dev changes you should pull Dev and merge Dev into your branch (locally and then push to remote) – so you are always working on the newest stable code version.

FINISHING A USER STORY

When done with the US on your branch

1. You or a team member should test the new code
   - Is it stable
   - Are the Acceptance tests fulfilled – is the US really done
   - Is it error free
   - Is it based on the newest Dev branch
2. If tested successfully make PullRequest to Dev (should be a fast forward thus no Merge conflicts)
   - Now a team member should do a detailed code review and determine if code is ready to be merged
   - If so then Pull Request is approved (with a good code review message)
   - Pull Request can then be merged into Dev

THE MASTER

- Only code that is stable and tested from Dev should be going into master
- If Dev is stable then make Pull Request to Master
  - At least 2 Team members should then test the code and review the changes
  - If approved with good review message then the Git Master can merge into master
- Customer should always be able to pull master and have a stable working version of your software.