SLS Lecture 5b : Version Control and GIT: Part II"

UC-SLS: (re)Version Control Part II

Slides for UC:SLS Lecture 6

https://git-scm.com/

- "Pro Git book", written by Scott Chacon and Ben Straub
  - Much of the material and images for this lecture are based on or from this book
    - license: https://creativecommons.org/licenses/by-nc-sa/3.0/
- Reference Manual (man pages) https://git-scm.com/docs

Jonathan Appavoo

1

Git Branches (Pro Git 3)

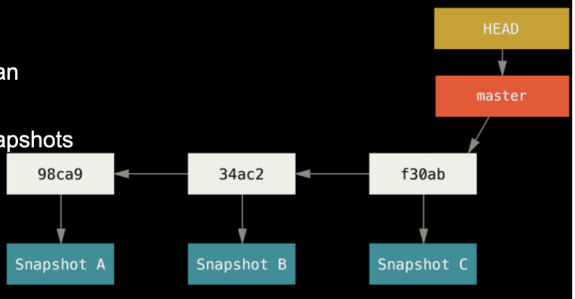
- Branching : What they are and how to use them
- Merging
- Branch Management
- Branching Workflows
- Remote Branches
- Rebasing

"Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop."

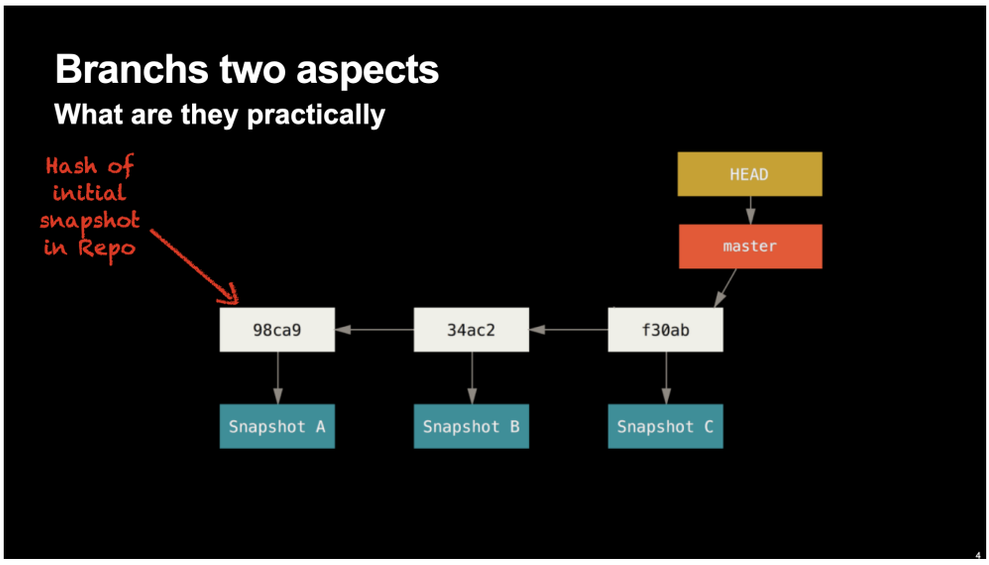
2

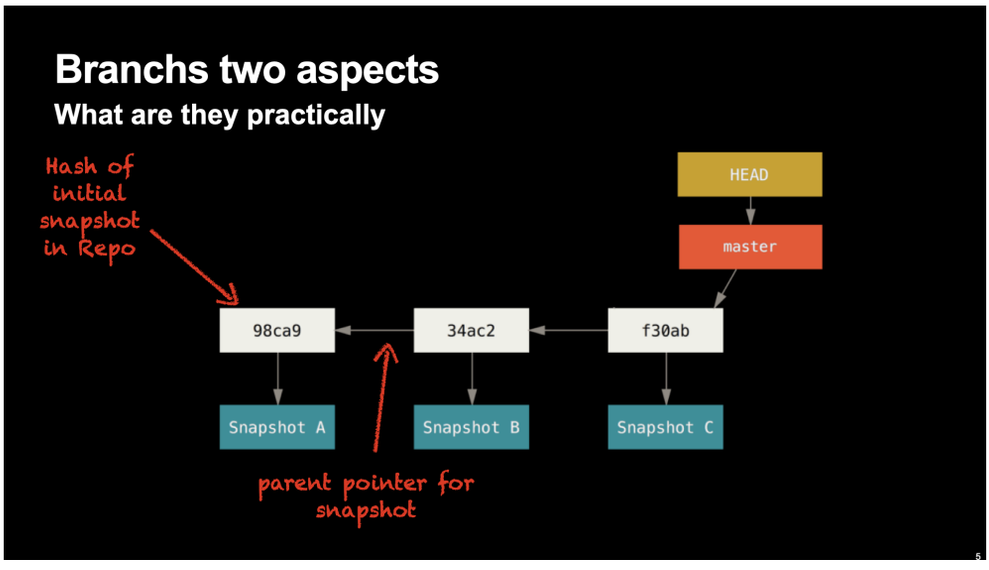
Branchs two aspects

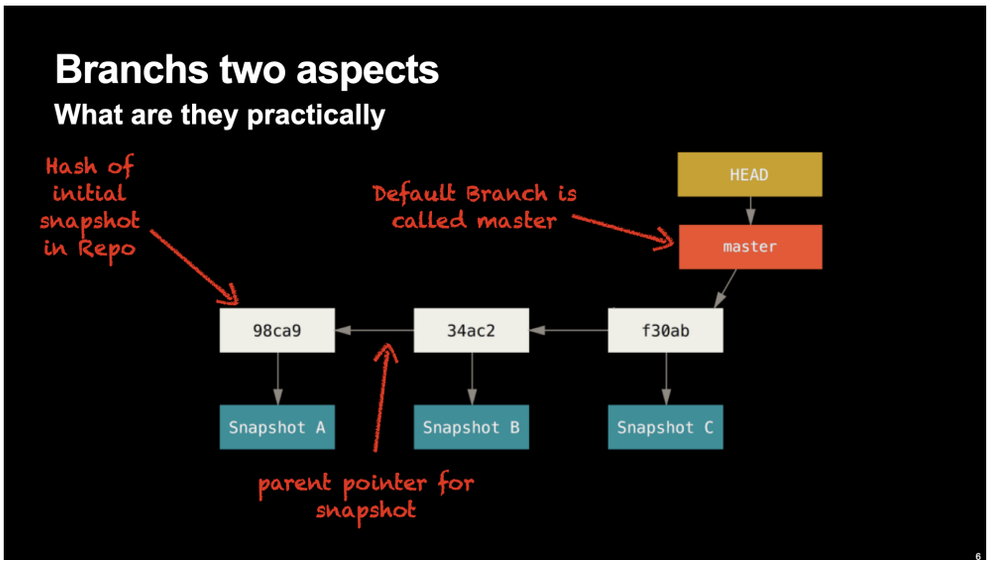
- What they are practically
  - A pointer to a snapshot that you can move around
  - use it to add a new "branch" of snapshots
- How to use them effectively
  - Powerful and defacto git way to communicate and organize work (changes)
  - short term: Fix bugs, add features
  - long term: alternative versions, etc.

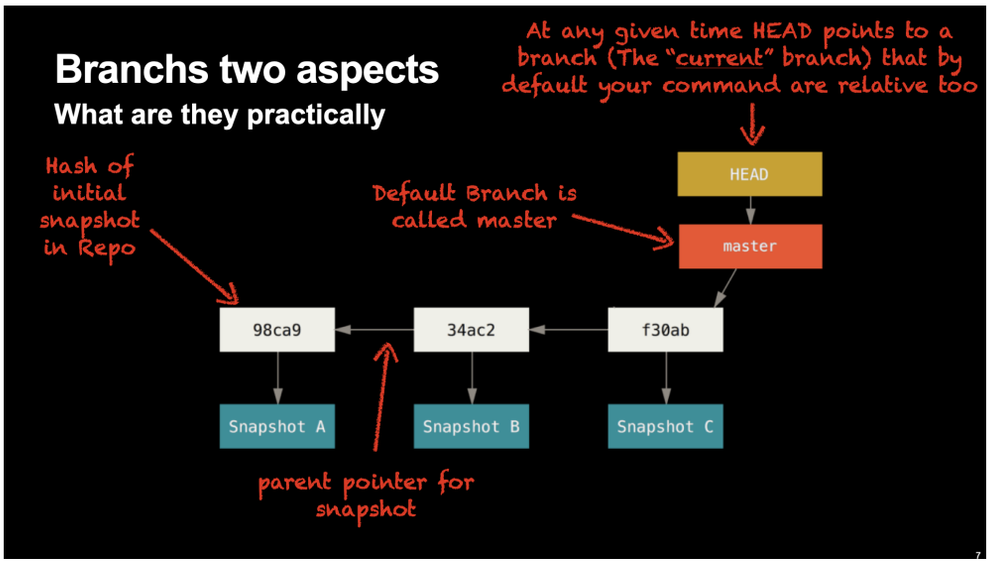


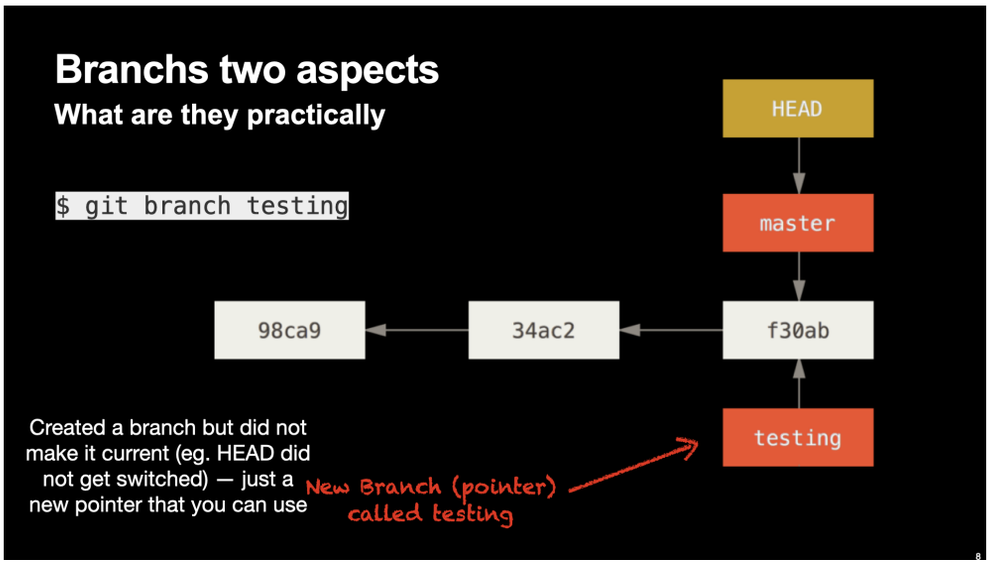
3

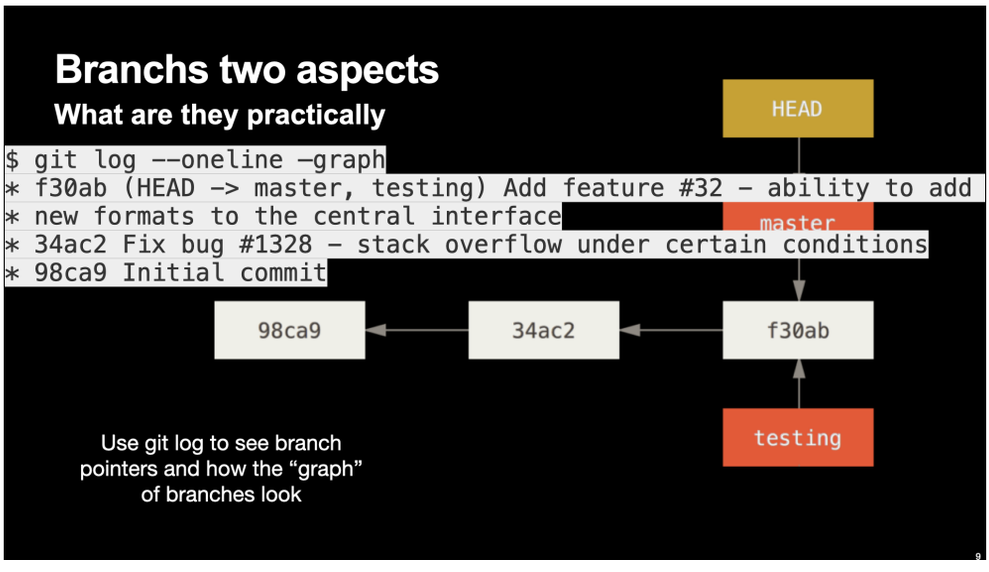

Branchs two aspects
What are they practically
Hash of initial snapshot in Repo
HEAD
master
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C
4

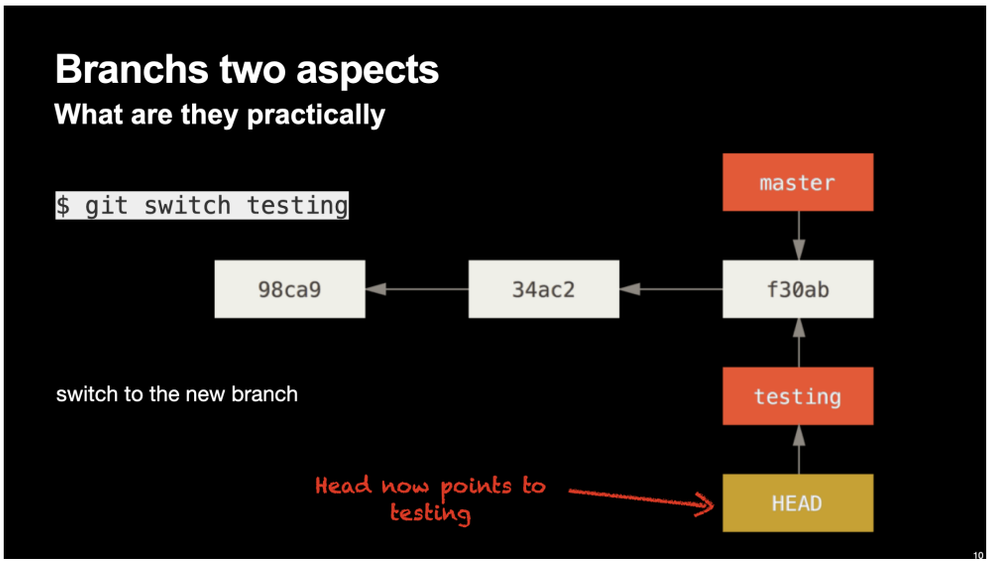

Branchs two aspects
What are they practically
Hash of initial snapshot in Repo
HEAD
master
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C
parent pointer for snapshot
5

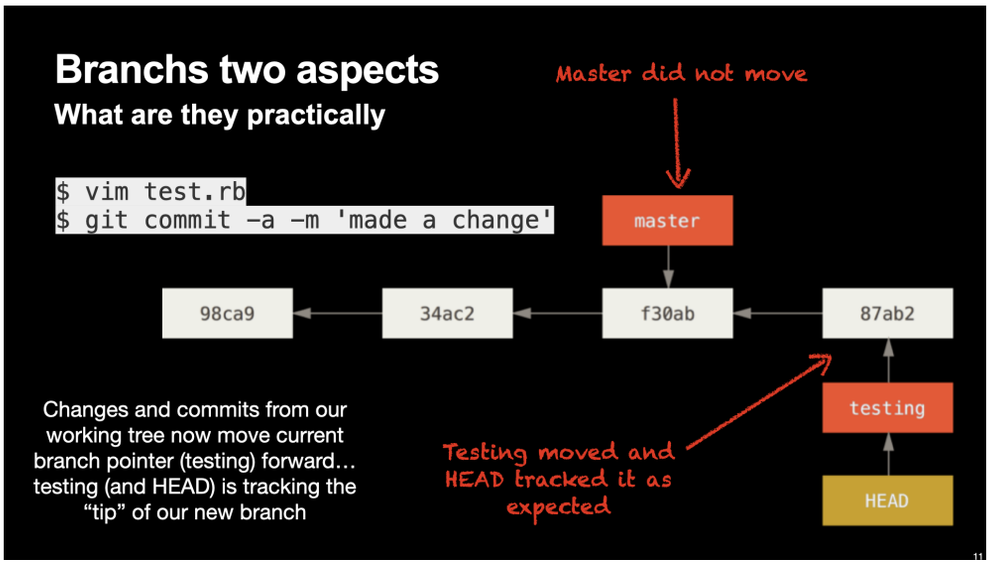

Branchs two aspects
What are they practically
Hash of initial snapshot in Repo
Default Branch is called master
HEAD
master
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C
parent pointer for snapshot
6

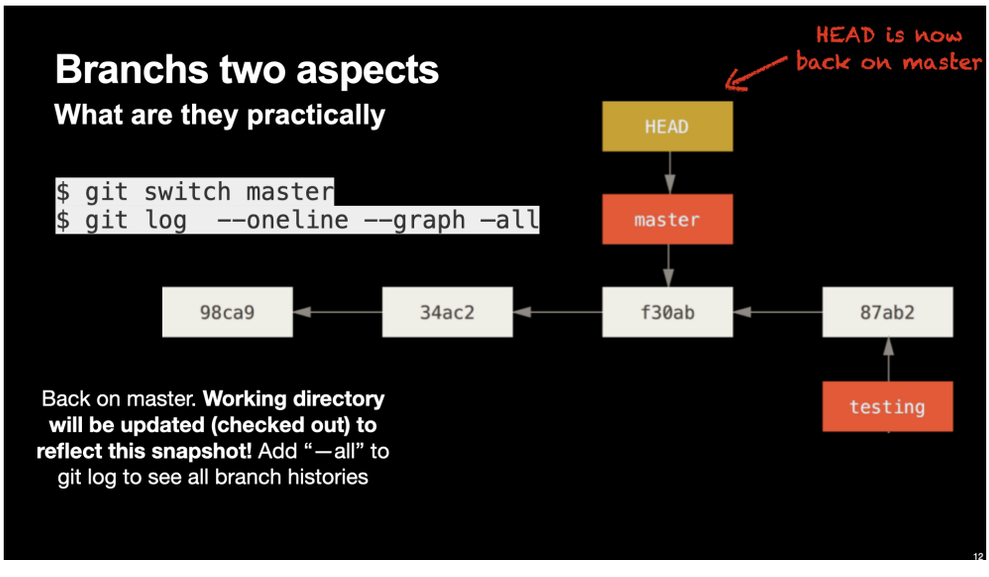

Branchs two aspects
What are they practically
At any given time HEAD points to a branch (The "current" branch) that by default your command are relative too
HEAD
master
Hash of initial snapshot in Repo
Default Branch is called master
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C
parent pointer for snapshot
7

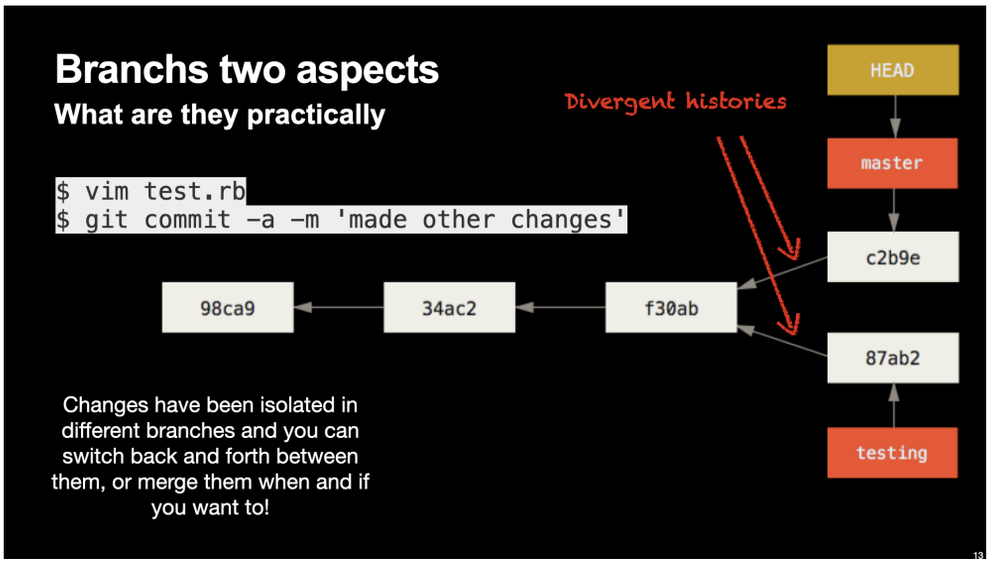

Branchs two aspects
What are they practically
$ git branch testing
HEAD
master
98ca9
34ac2
f30ab
testing
Created a branch but did not make it current (eg. HEAD did not get switched) — just a new pointer that you can use
New Branch (pointer) called testing
8

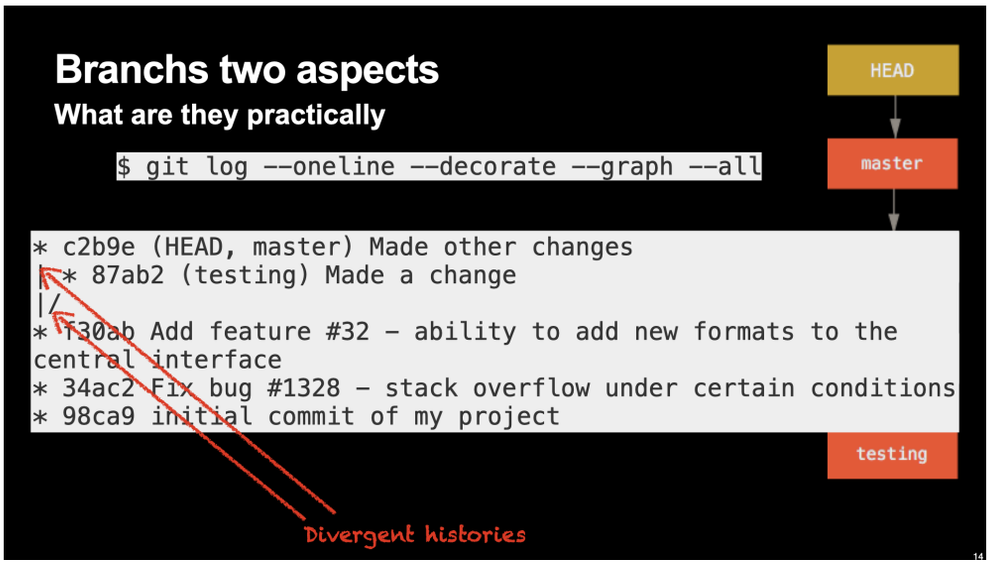

Branchs two aspects
What are they practically
HEAD
$ git log --oneline –graph
* f30ab (HEAD -> master, testing) Add feature #32 - ability to add
* new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 Initial commit
master
98ca9
34ac2
f30ab
testing
Use git log to see branch pointers and how the "graph" of branches look
9

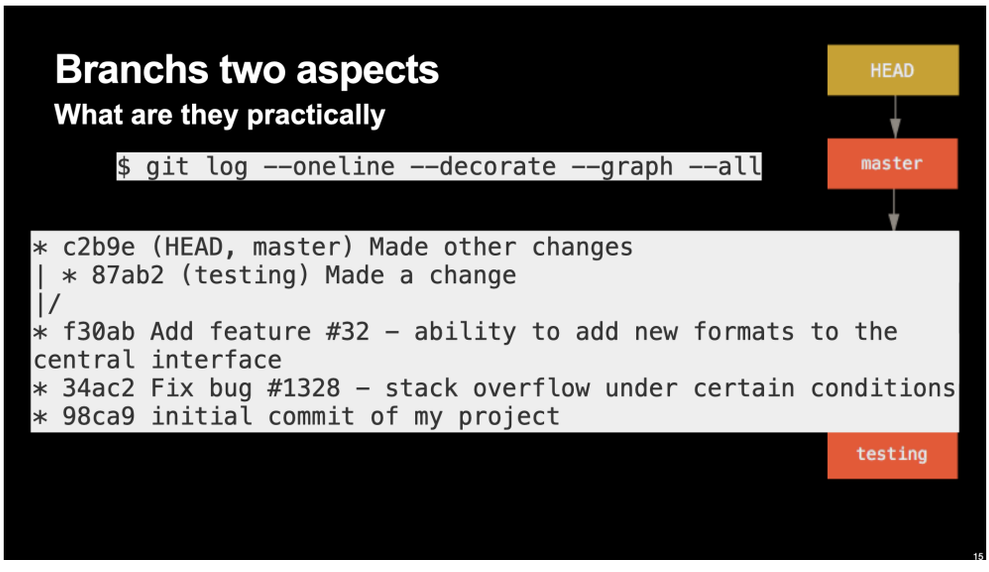

Branchs two aspects
What are they practically
$ git switch testing
master
98ca9
34ac2
f30ab
switch to the new branch
testing
Head now points to testing
HEAD
10


Branchs two aspects
What are they practically
Master did not move
$ vim test.rb
$ git commit -a -m 'made a change'
master
98ca9
34ac2
f30ab
87ab2
Changes and commits from our working tree now move current branch pointer (testing) forward… testing (and HEAD) is tracking the "tip" of our new branch
testing
Testing moved and HEAD tracked it as expected
HEAD
11


Branchs two aspects
What are they practically
HEAD is now back on master
HEAD
$ git switch master
$ git log --oneline --graph -all
master
98ca9
34ac2
f30ab
87ab2
Back on master. Working directory will be updated (checked out) to reflect this snapshot! Add "—all" to git log to see all branch histories
testing
12

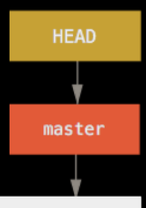

Branchs two aspects
What are they practically
$ vim test.rb
$ git commit -a -m 'made other changes'
Divergent histories
HEAD
master
c2b9e
98ca9
34ac2
f30ab
87ab2
testing
Changes have been isolated in different branches and you can switch back and forth between them, or merge them when and if you want to!
13


Branchs two aspects
What are they practically
$ 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 central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
HEAD
master
testing
Divergent histories
14


Branchs two aspects
What are they practically
$ 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 central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
HEAD
master
testing
15

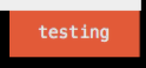
Branchs two aspects

What are they practically




```
$ 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
central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```



As always there are all kinds of shortcuts adding -c to switch will create a new branch and switch in one command and - alone with switch to previous branch you were on.

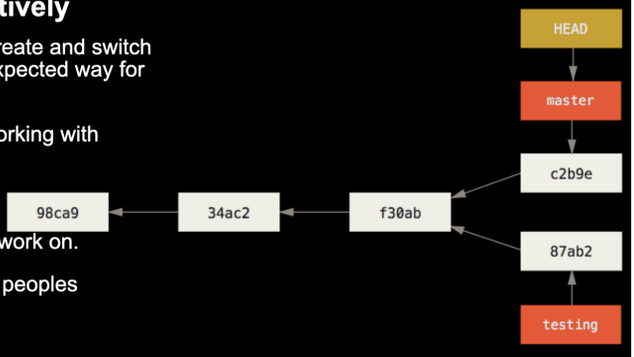
```
$ git switch -c  <branchname>
$ git switch -
```

16

Branchs two aspects

How to use them effectively

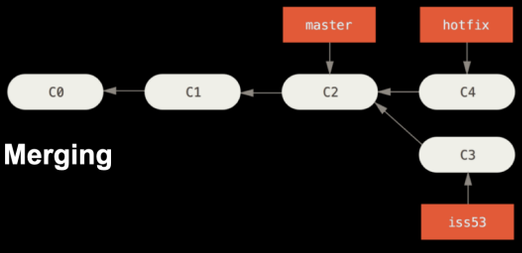
- Given how easy it is in git to create and switch branches it has become the expected way for you to organize you work
- This is especially true when working with others
- It is always a good idea to
  - create branches to do your work on.
  - use branches to track other peoples versions
  - to compare versions
  - to "merge" or "rebase" versions of code



17

Branching and Merging (Pro Git 3.2)

Purposefully Diverging and Merging



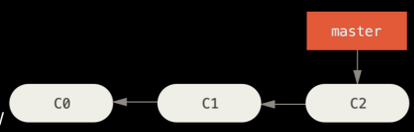
Using branches to carefully ensure that what is stable and in production versus fixes and features are carefully managed

18

Branching and Merging

An example workflow

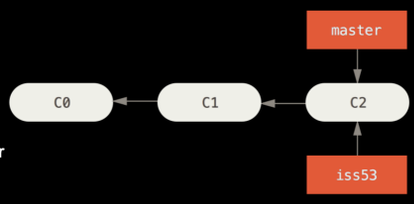
A



- Project is at state A
- Your project uses some form of "issue" tracking / management system. Such a system tracks work to be done and assigns work a unique "issue identifier"
  - Eg. "iss53: Add a footer to the front page".
    - Git services like GitHub and GitLab often provide issue tacking
- In our case we decide to work on "iss53"
  - we update the issue tracking system to let people know
  - and create a branch and switch to it to do our work

```
$ git switch -c iss53
```

B

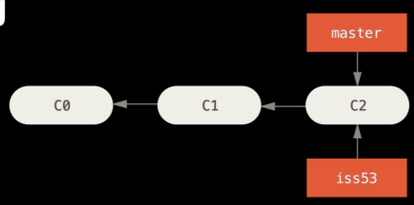


19

Branching and Merging

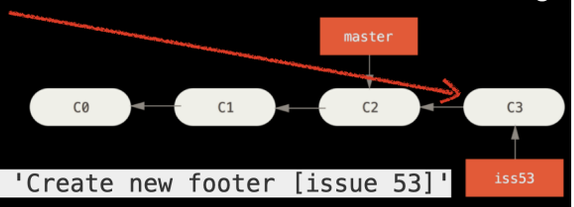
An example workflow

B



- You start working on the changes necessary and do a commit to track your first steps
  - Your commit results in the project being in state C with your issue branch 1 commit "a head" of the master branch

C




```
$ vim index.html
$ git commit -a -m 'Create new footer [issue 53]'
```

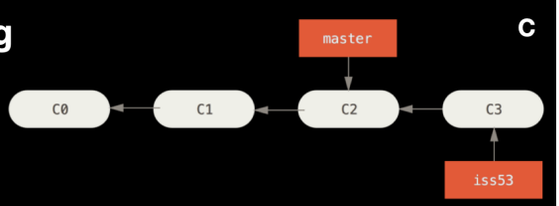
20

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

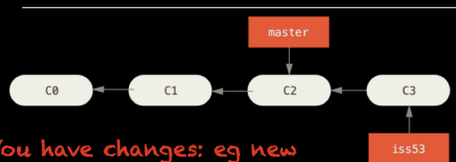
C



- At this point your repository is in state C and your keep working
  - modify files, etc
  - eg.

```
$ vim index.html
```

D



- While you are doing your work you get an urgent message to work on a fix to the production version
  - You are in state D
    - Your working tree is DIRTY
    - What do you do?

You have changes: eg new files, modified files, etc that are not committed and might "conflict" with the master

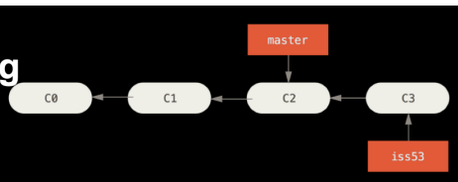
21

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

- We would like to switch over to the master branch and start working on the fix
  - switching to the master means letting GIT change out Working Tree to get it back to the version in master
  - While in some cases you can do it is best practice to only switch when your working tree is "CLEAN"



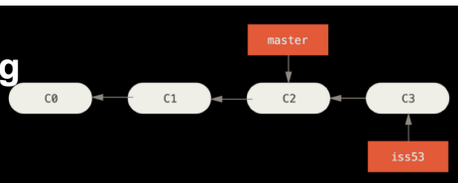
22

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

- So how do "Clean" things in a way that lets us get back to our work later?
  1. Commit the partial work they way it is (maybe on yet another branch)
  2. Get to a point where we "happy" committing it
  3. "Stash" it away — make copy of the working tree and then revert the working tree to your last commit



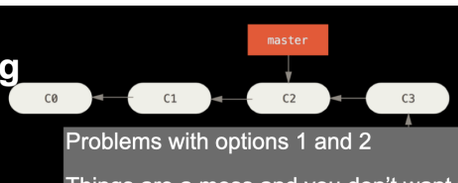
23

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

- So how do "Clean" things in a way that lets us get back to our work later?
  1. Commit the partial work they way it is (maybe on yet another branch)
  2. Get to a point where we "happy" committing it
  3. "Stash" it away — make copy of the working tree and then revert the working tree to your last commit



Problems with options 1 and 2

Things are a mess and you don't want to commit something not working

You are under a time crunch and have to get to the fix and don't have time to get things to a point that make you "happy"

Git has formal support for option 3 "The Stash"

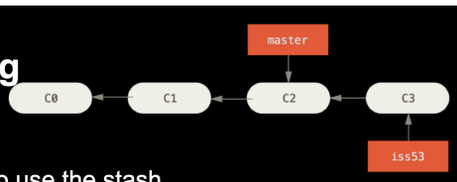
Git has a stack of where you can "stash" changes to your working tree and retrieve them — note the stash does not get pushed so it is only accessible and visible in the copy of the repo where you did the stash

24

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

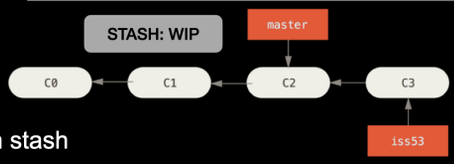


D

- See Pro Git 7.3 for details on how to use the stash.
- Lets assume we use the stash as follows:

```
$ git stash push --all -m "issue 53: index.html broken"
```

- At this point we are in State E
  - Working tree is clean
  - and Work InProgress (WIP) is in stash



E

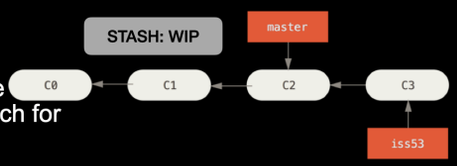
25

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

E



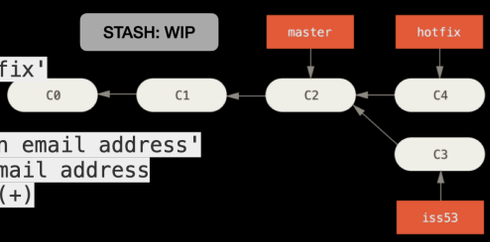
- Now with our working tree clean we switch to master create a new branch for our fix eg.

```
$ git checkout master
Switched to branch 'master'
```

```
$ git switch -c hotfix
Switched to a new branch 'hotfix'
```

```
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```

F



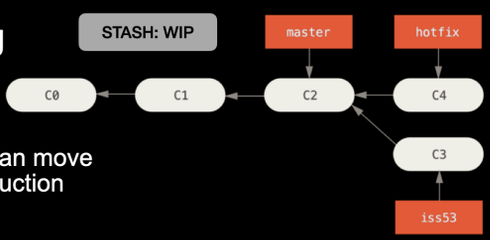
26

Branching and Merging

An example workflow

Mixing in some content from Pro Git 7.3 (stashing)

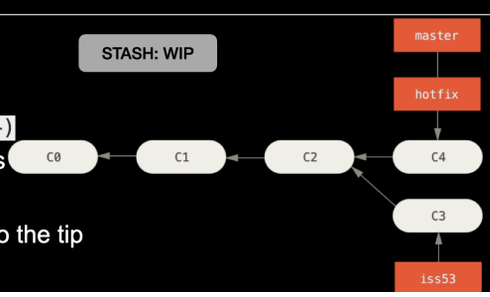
F



- Once we are confident in our fix we can move master "forward" to get the fix in production with the "merge" command

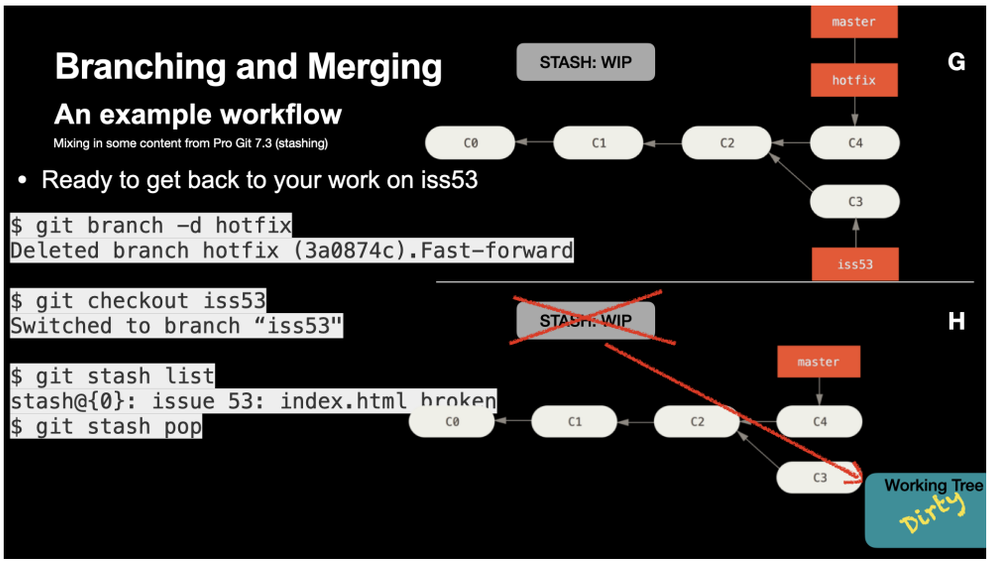
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

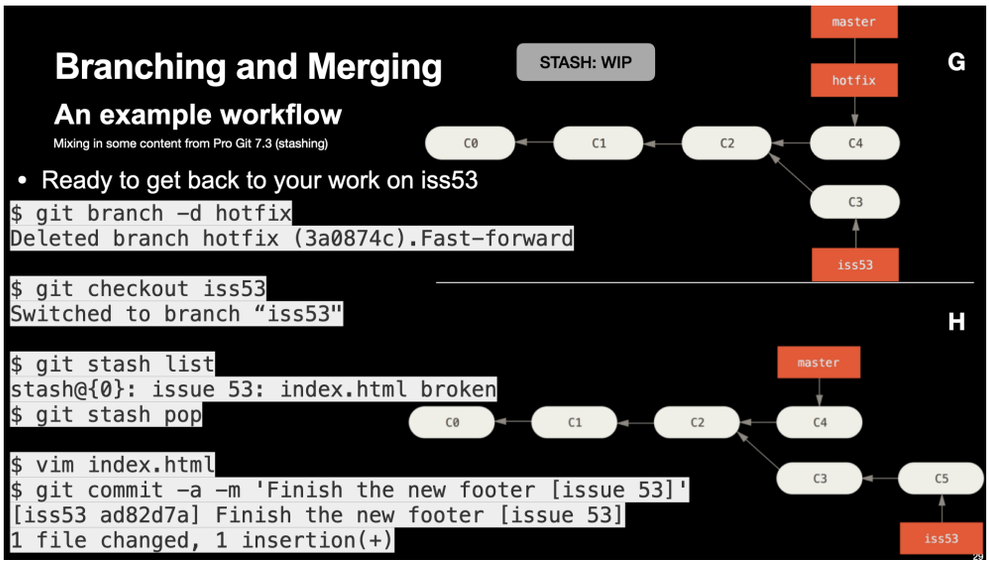
G

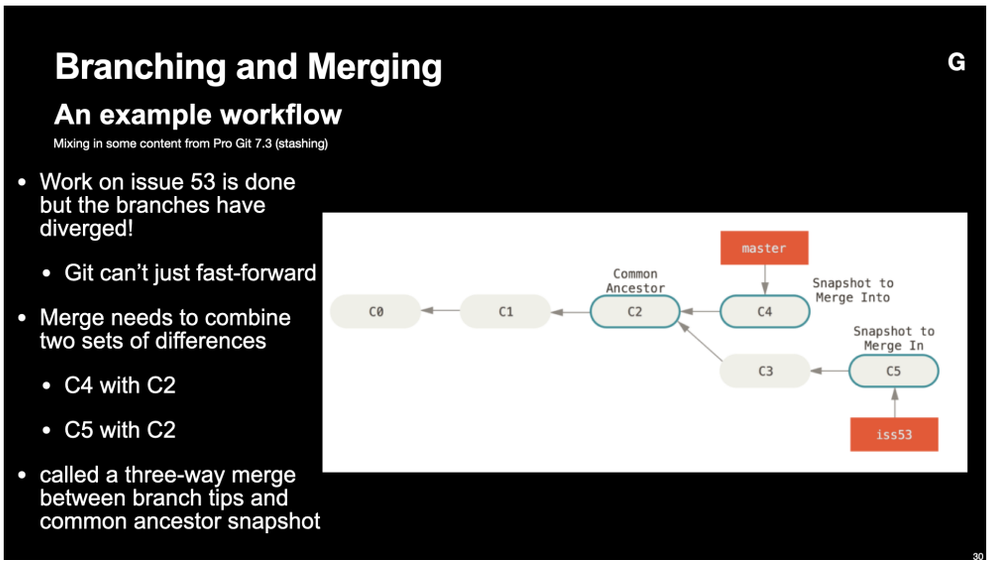


- Given that our hotfix and master does diverge this is a Fast-Forward merge
  - Can just move the master pointer to the tip of hot fix

27


Branching and Merging
An example workflow
Mixing in some content from Pro Git 7.3 (stashing)
STASH: WIP
G
H
master
hotfix
iss53
C0
C1
C2
C3
C4
Working Tree
Dirty
• Ready to get back to your work on iss53
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).Fast-forward
$ git checkout iss53
Switched to branch "iss53"
$ git stash list
stash@{0}: issue 53: index.html broken
$ git stash pop


Branching and Merging
An example workflow
Mixing in some content from Pro Git 7.3 (stashing)
STASH: WIP
G
H
master
hotfix
iss53
C0
C1
C2
C3
C4
C5
• Ready to get back to your work on iss53
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).Fast-forward
$ git checkout iss53
Switched to branch "iss53"
$ git stash list
stash@{0}: issue 53: index.html broken
$ git stash pop
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)

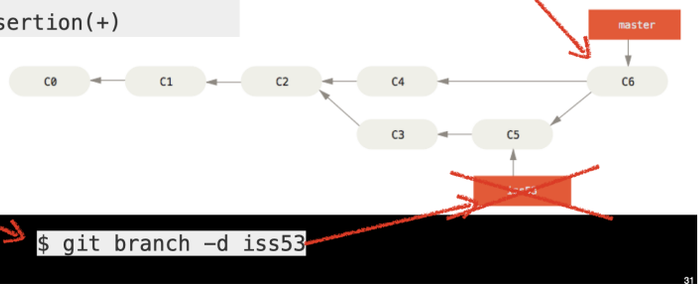

Branching and Merging
An example workflow
Mixing in some content from Pro Git 7.3 (stashing)
G
• Work on issue 53 is done but the branches have diverged!
• Git can't just fast-forward
• Merge needs to combine two sets of differences
• C4 with C2
• C5 with C2
• called a three-way merge between branch tips and common ancestor snapshot
master
Common Ancestor
Snapshot to Merge Into
Snapshot to Merge In
iss53
C0
C1
C2
C3
C4
C5

Branching and Merging

An example workflow

Merge Commit (more than one parent)

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



- No longer need the iss53 pointer so we can delete
  - Note commits are still in repo
    - history reflects everthing that happened an their order

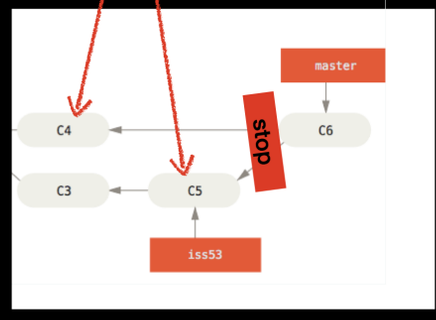
```
$ git branch -d iss53
```

31

Merge Conflicts

Both change the same lines in index.html

- If files in the two branches change the same lines then we have a problem
  - CONFLICTING CHANGES
    - Which one is right?
    - Git cannot automatically figure this out and needs your help to complete creating the merge commit

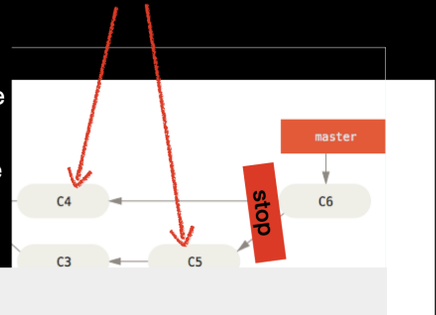


32

Merge Conflicts

Both change the same lines in index.html

- Git stops in the middle of merging
  - Any files it could automatically take care are updated in your working tree
  - the rest have been modified to mark the "conflicts" that you need to take care of




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

33

Merge Conflicts

- git status tells us what we need to do

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

    both modified:      index.html

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

34

Merge Conflicts

- Git contents of the files that have unmerged changes in standard way
- We must open each file with conflicts and "resolve" them (testing of course our fixes)

index.html contents

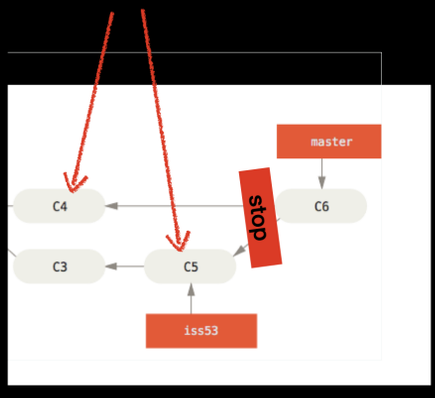
```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

HEAD is master since that is what we had checkout when we started the merge

35

Merge Conflicts

- At this point you must edit the files and remove all the "conflicts" in each (removing all the markers as well)
  - Sometimes it as easy as pick one version or the other
  - But some times it is more complicated you might need to combine things into a new content all together
    - This can be a hard job… its a good idea to get a sense of what you are getting your self into but inspect the branches before you get going



36

Merge Conflicts

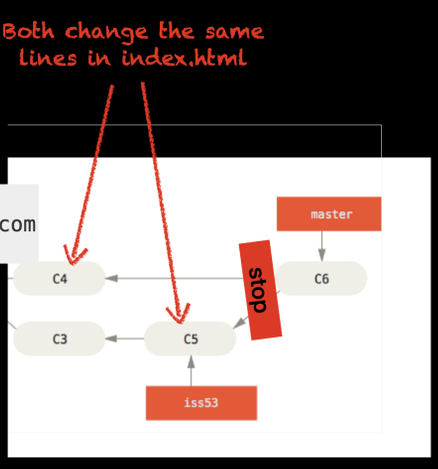
- Once resolve

Resolved

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

- git add each changed file

```
$ git add index.html
```



37

Merge Conflicts

- Once resolve

Resolved

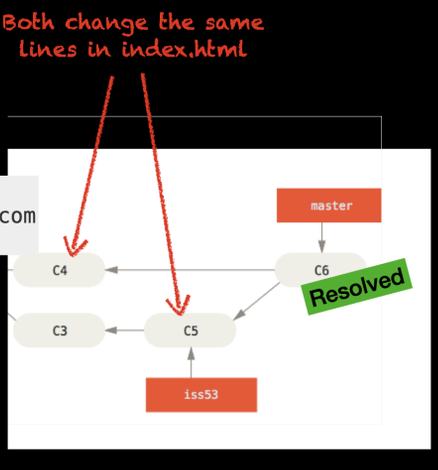
```
<div id="footer">
 please contact us at support@github.com
</div>
```

- git add each changed file

```
$ git add index.html
```

- finalize the merge by running the commit

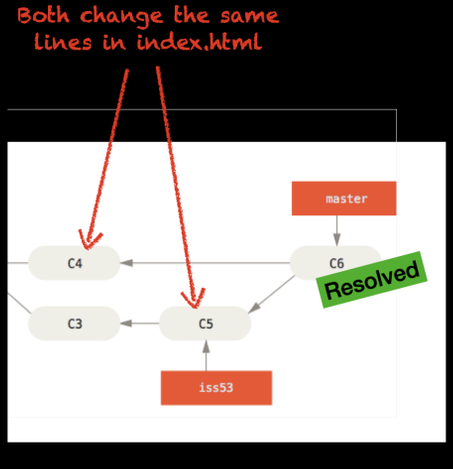
```
$ git commit
```



38

Merge Conflicts

- Some things to note
  - Git will provide a default merged commit message you might want to update to document what you did to resolve the conflicts
  - There are tools that can help you inspect the version of the files in question : see git mergetool
    - eg vimdiff and others
  - For more advanced coverage
    - See ProGit 7.8 Git Tools - Advanced Merging



39

Branches and Remotes

Revisiting Remotes with Branches in mind

- Reminder a remote is just other repository that you have "added" — so that you can exchange information it

```
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Now that we understand branches

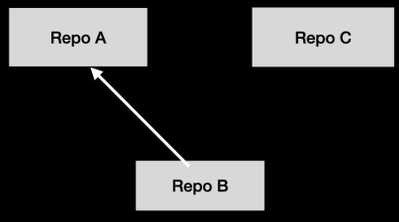
- Let's look at how branches work with respect to remotes

40

Branches and Remotes

Revisiting Remotes with Branches in mind

- Lets think about a scenario involving 3 repos
  - Repo A
  - Repo B cloned from Repo A
    - So starts with one Remote called "origin" that points to Repo A
  - Repo C owned Jill
- As part of their state each Repo has branches as we have looked at
  - all of them have at least one branch which is usually named "master"

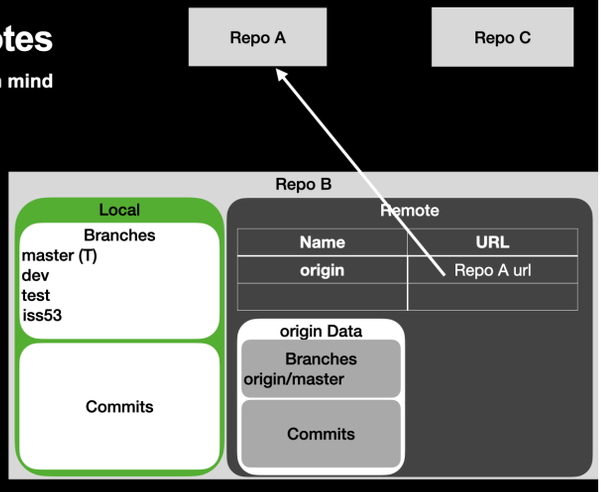


41

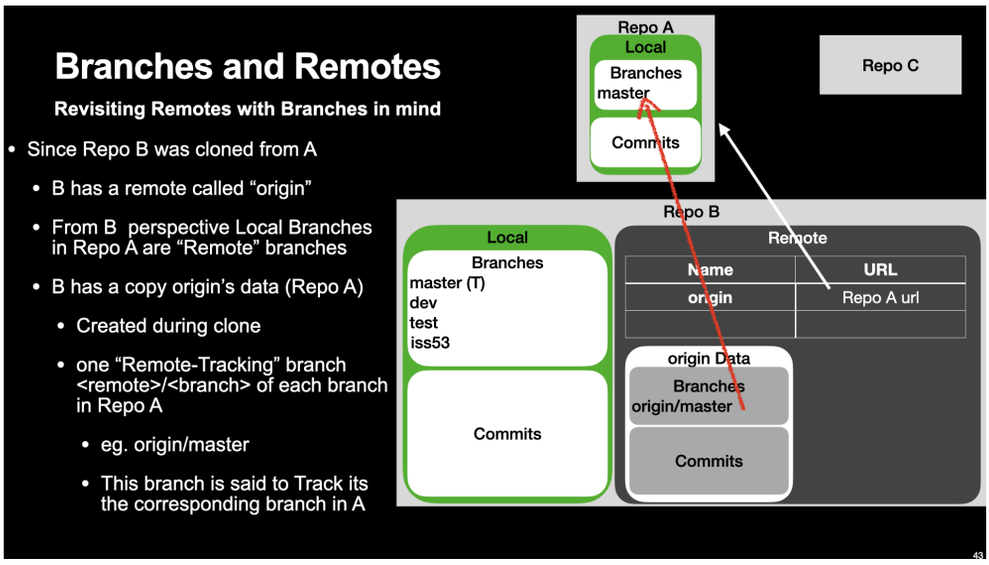
Branches and Remotes

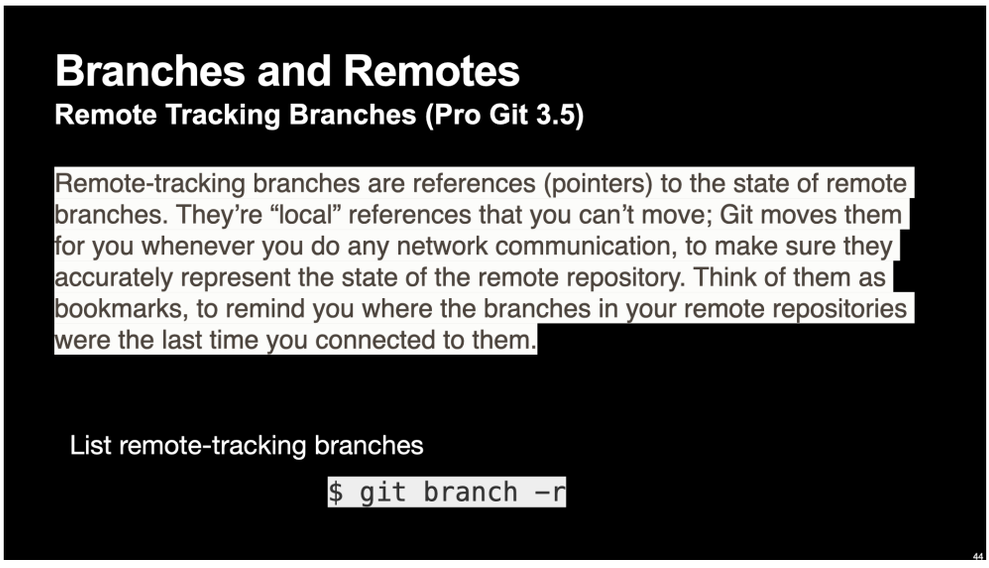
Revisiting Remotes with Branches in mind

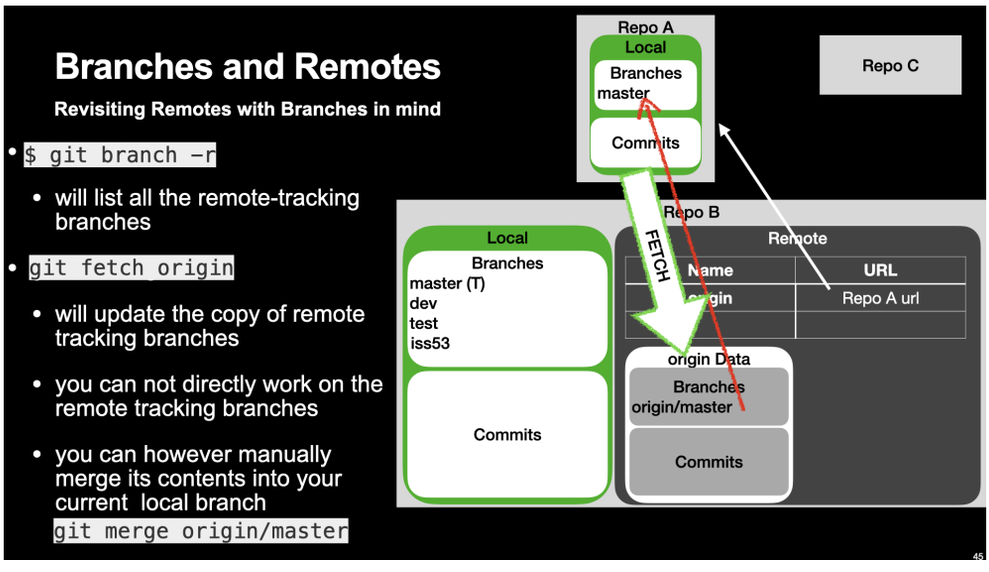
- Need to dig into Repo
- Repo state is organized into
  1. Local Information
     - Branch pointers
     - Commits
  2. Remote Information
     1. Known remote repos
     2. Data for each remote



42


Branches and Remotes
Revisiting Remotes with Branches in mind
Since Repo B was cloned from A
B has a remote called "origin"
From B perspective Local Branches in Repo A are "Remote" branches
B has a copy origin's data (Repo A)
Created during clone
one "Remote-Tracking" branch <remote>/<branch> of each branch in Repo A
eg. origin/master
This branch is said to Track its the corresponding branch in A
Repo A
Local
Branches
master
Commits
Repo C
Repo B
Local
Branches
master (T)
dev
test
iss53
Commits
Remote
Name
URL
origin
Repo A url
origin Data
Branches
origin/master
Commits
43


Branches and Remotes
Remote Tracking Branches (Pro Git 3.5)
Remote-tracking branches are references (pointers) to the state of remote branches. They're "local" references that you can't move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.
List remote-tracking branches
$ git branch -r
44


Branches and Remotes
Revisiting Remotes with Branches in mind
$ git branch -r
will list all the remote-tracking branches
git fetch origin
will update the copy of remote tracking branches
you can not directly work on the remote tracking branches
you can however manually merge its contents into your current local branch
git merge origin/master
Repo A
Local
Branches
master
Commits
Repo C
FETCH
Repo B
Local
Branches
master (T)
dev
test
iss53
Commits
Remote
Name
URL
Repo A url
origin Data
Branches
origin/master
Commits
45

Branches and Remotes

Tracking Branches (Pro Git 3.5)

Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.

List tracking branches

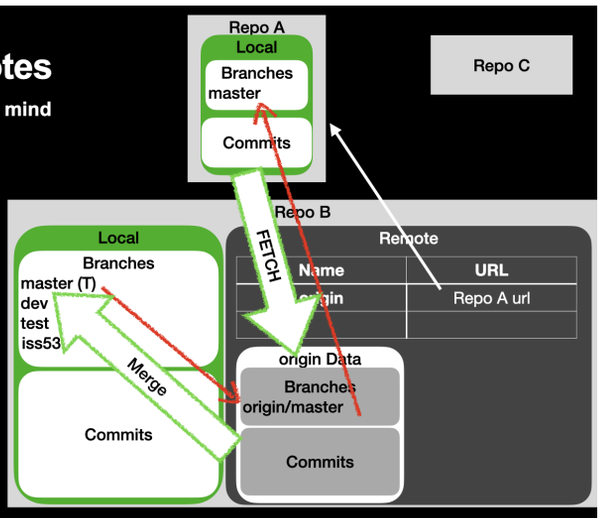
```
$ git branch -vv
```

46

Branches and Remotes

Revising Remotes with Branches in mind

- However you can create a local branch that also tracks one of the remote tracking branches
  - by default the clone creates the local master to track the origin/master
- With a local tracking branch you can do `git pull`
  - automatically **fetch** and a **merge**
  - update your work with remote



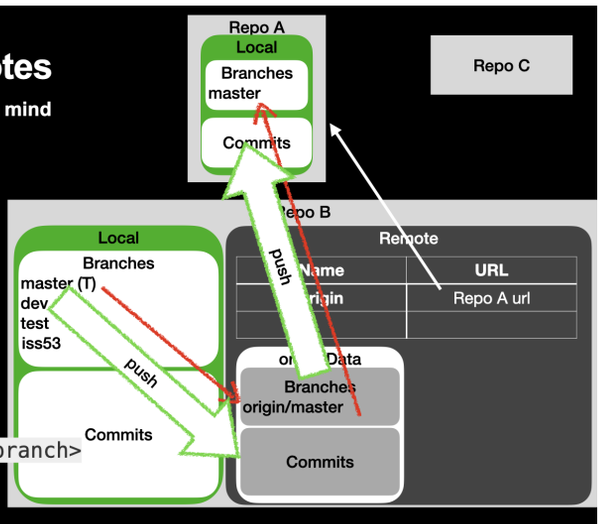
47

Branches and Remotes

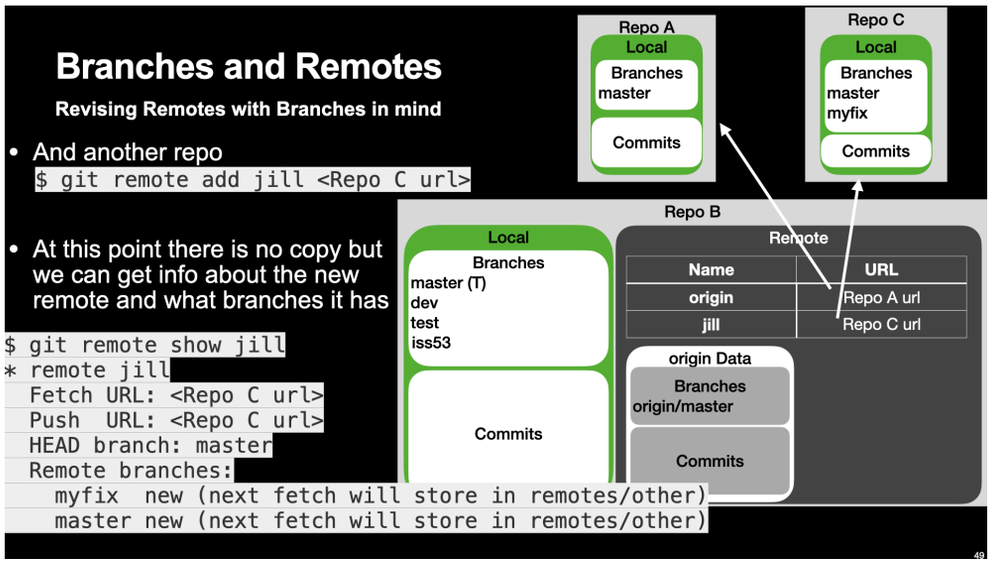
Revising Remotes with Branches in mind

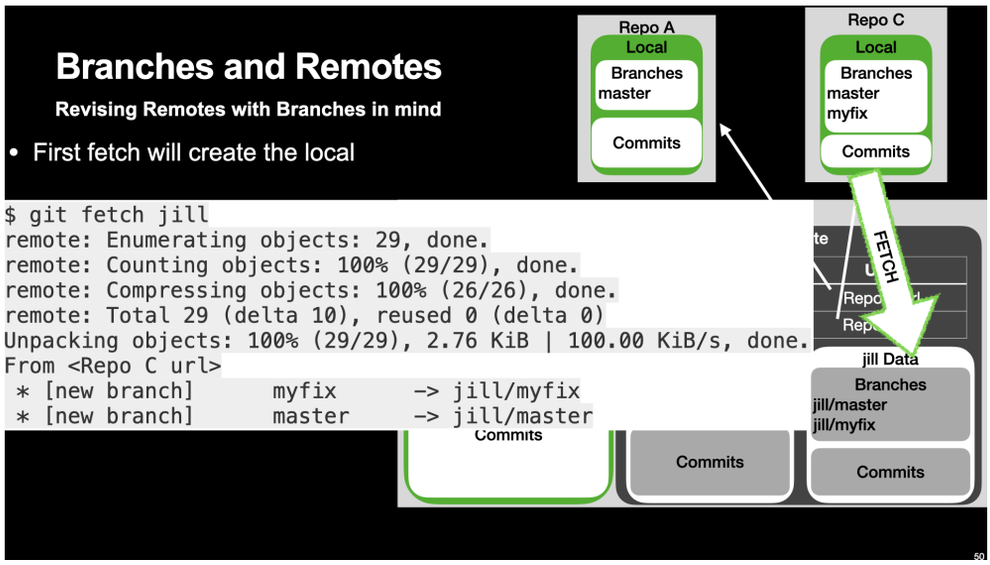
- To send your work to a repo use `git push <remote> <branch>`
- This however will fail if your version of the remote is stale
  - you will first need to do a pull
    - take care of merges if needed

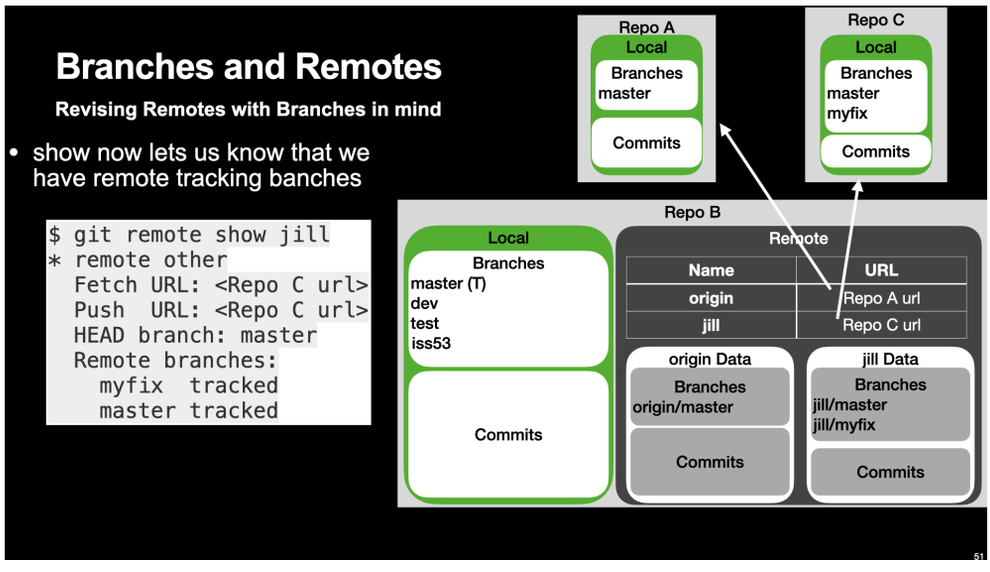
`git push <remote> <branch>:<branch>`



48


Branches and Remotes
Revising Remotes with Branches in mind
And another repo
$ git remote add jill <Repo C url>
At this point there is no copy but we can get info about the new remote and what branches it has
$ git remote show jill
* remote jill
Fetch URL: <Repo C url>
Push URL: <Repo C url>
HEAD branch: master
Remote branches:
myfix new (next fetch will store in remotes/other)
master new (next fetch will store in remotes/other)
Repo A
Local
Branches
master
Commits
Repo C
Local
Branches
master
myfix
Commits
Repo B
Local
Branches
master (T)
dev
test
iss53
Commits
Remote
Name
URL
origin
Repo A url
jill
Repo C url
origin Data
Branches
origin/master
Commits
49


Branches and Remotes
Revising Remotes with Branches in mind
First fetch will create the local
$ git fetch jill
remote: Enumerating objects: 29, done.
remote: Counting objects: 100% (29/29), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 29 (delta 10), reused 0 (delta 0)
Unpacking objects: 100% (29/29), 2.76 KiB | 100.00 KiB/s, done.
From <Repo C url>
* [new branch] myfix -> jill/myfix
* [new branch] master -> jill/master
FETCH
jill Data
Branches
jill/master
jill/myfix
Commits
50

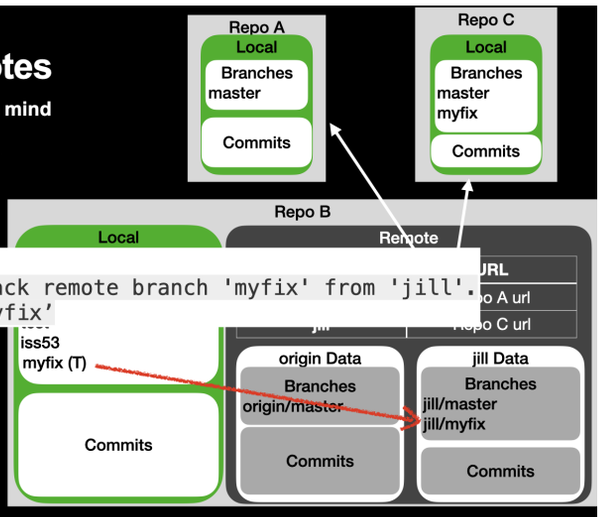

Branches and Remotes
Revising Remotes with Branches in mind
show now lets us know that we have remote tracking banches
$ git remote show jill
* remote other
Fetch URL: <Repo C url>
Push URL: <Repo C url>
HEAD branch: master
Remote branches:
myfix tracked
master tracked
Repo A
Local
Branches
master
Commits
Repo C
Local
Branches
master
myfix
Commits
Repo B
Local
Branches
master (T)
dev
test
iss53
Commits
Remote
Name
URL
origin
Repo A url
jill
Repo C url
origin Data
Branches
origin/master
Commits
jill Data
Branches
jill/master
jill/myfix
Commits
51

Branches and Remotes

Revising Remotes with Branches in mind

- By default switching to a remote-tracking branch creates a corresponding tracking branch for you to work with

```
$ git switch myfix
Branch 'myfix' set up to track remote branch 'myfix' from 'jill'.
Switched to a new branch 'myfix'
```



52

Rebasing vs Merging

ProGit 3.6

- There are two strategies that people take with respect to a projects history
  - Merging —
    - where the history of a project accurately recorded the concurrent branching and merging
      - As one produces with thee merge command
  - Rebasing —
    - Where published versions of a project present and idealized history with no merges of concurrent work.
      - Rather history is a sequestional set of fast-forward commits (a straight line)
      - Here you do not merge branches rather when you are ready to publish you changes you must "rebase" your changes on to the mainline version
- There are advantages and disadvantages to both. My suggestion is you first worry bout using git well in your daily workflow and then worry about how to rewrite history for the sake of how a projects wants to present itself to the world

53

Distributed Git (Pro Git 5.1)

- Now that you know more about how git works you can look at chapter 5.1 to learn about how people use it to manage projects across teams and computers
- There are various ways that people organize their repos and control the flow of changes
  - Forking is a particular model that is very popular model supported by services such as GitHub and GitLab
    - updates to the main repo happen via Pull Requests (PRs) after testing and review.

54

Git Services: eg. GitHub, GitLab, etc

- For a quick overview of GitHub see ProGit chapter 6
- However it is worth noting that there are other service such as GitLab
- Both GitHub and GitLab also allow you to create you own instances of them for you private organization. Either on your own computers or in the cloud
  - This is what cs400-gitlab.bu.edu. —- it is our drives and admins that manage it

55

Git Tools

ProGit Chapter 7

- There are many advanced tools to help you with more complex things you might do.
  - One important one is interactive rebasing.
    - It allows you to interactively rewrite the commit history
      - one common operation you may have to do when contributing to projects is squashing several of your commits into one.

56

GIT Graphical Interfaces and IDE integration

ProGit Appendix A

- Version control is one of those things that can really benefit from a Graphical interface so that
  - you can better visualize your history and branches
  - make merging of multiple versions of a file easier
- Many standalone tools exist
- Many IDE's include Git support — vscode, Juypter, emacs, etc.
  - But remember it pays to understand the underlying operations.

57