GIT SCM*

as a VCS

software version control, git in a nutshell, handling branches and remotes

Can Tunca

*Slides adapted from http://git-scm.org

Courtesy of Gökhan Çapan

The Rationale

- Revision control is essential for projects developed as a team
  - Each member has its working copy
  - A member can create a new revision
  - Other members can apply the revision
  - A new member can get his up-to-date working copy

The Rationale

- An illustration:
    1. Member A gets his working copy
    2. A branches from the main development line (context-switches), and work on a new feature
    3. In the meantime Member B creates a new revision
    4. Member A gets his work done on the branch
    5. Member A updates his working copy and his branch (to make sure that there is no conflict)
    6. Member A merges his work on the branch into the main development line
    7. Member A deletes the feature-branch, and creates a new revision in the main development line

GIT

- Git is an open source version control system, which is really common among organizations creating software (even Linux kernel is developed under git VCS)
- Git is distributed (i.e. every member has its own ‘clone’)
- Git is easy-to-use, and really fast
- Git supports staging
- Branching and working with remotes are extremely easy with Git
- Virtually all types of workflows are possible

Git in Action

- Initialize a project

```
$git init project

Initialized empty Git repository in
/path/to/project/.git/
```

Git in Action

- Observe that a .git directory is created

```
$cd project
$ls -a
```

Git in Action

- Here is the status of your working copy

```
$git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add"
to track)
```

Git in Action

- Let's create a file and commit it, git tells us that there are files that are not tracked (this is a new file), but we can stage them to commit them later

```
$touch first.txt
$git status
On branch master

Initial commit

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

 first.txt
```

Git in Action

- Stage the recently added file

```
$git add first.txt
$git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

 new file:   first.txt
```

Git in Action

- Commit the staged changes

```
$git commit -m "Adding a totally useless file"

[master (root-commit) 5dfba11] Adding a totally useless
file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 first.txt
```

Git in Action

- Do some modifications in the first.txt file

```
$echo “Something useful”>>first.txt
$git status

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

 modified:   first.txt

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

Git in Action

- Commit the file

```
$git commit -m “This will fail"

On branch master
Changes not staged for commit:
 modified:   first.txt

no changes added to commit
```

Git in Action

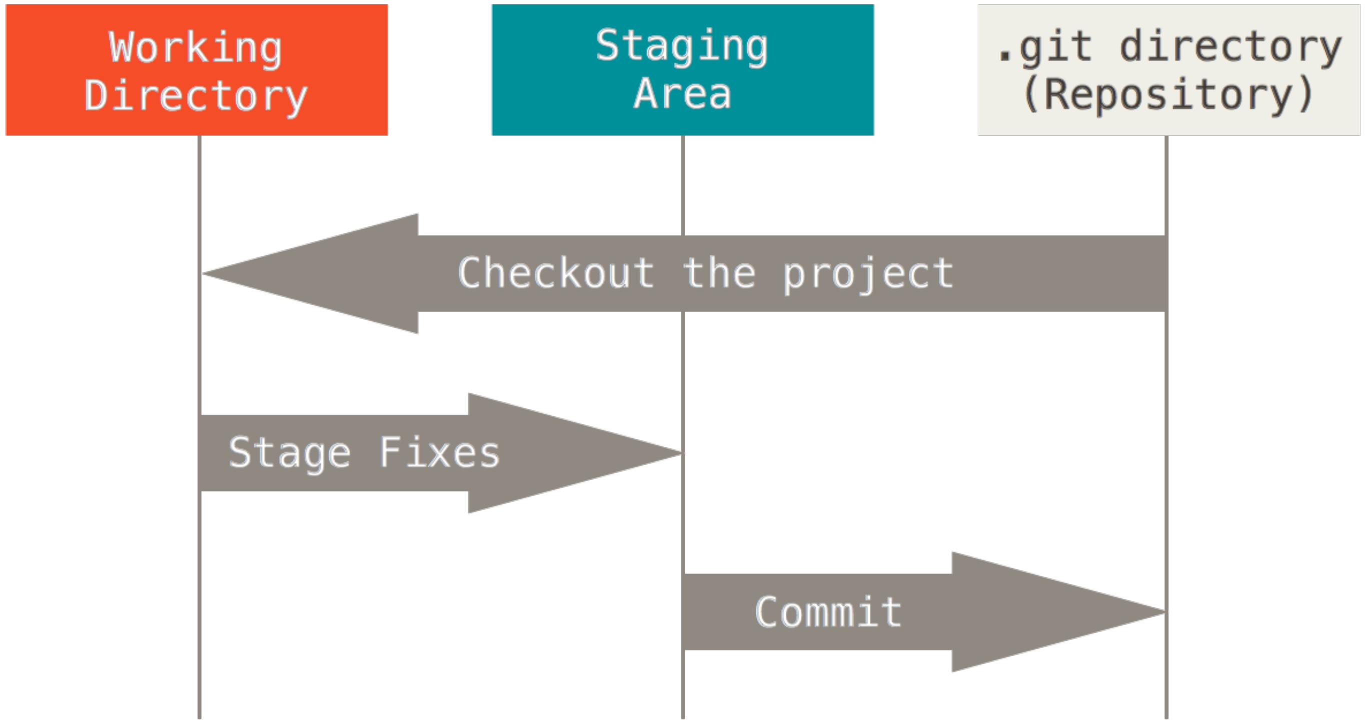
- We need to stage it first by git add <file>, and then commit via git commit -m “Message"
- Alternatively, we can use git commit -a -m “Message”, which is equivalent

Git in Action

- Try git commit -a -m “Message"

```
$git commit -a –m “This will work"

[master 6bfe9c5] This will work
 1 file changed, 1 insertion(+)
```

States of Committing



* Figure copied from http://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Undoing

- Say you want to change your latest commit (you want to change the commit message, or you realize that you this commit should have included another file)

```
#Make your changes and stage them
$git commit --amend -m “A better commit message”

[master 4922193] A better commit message
 1 file changed, 1 insertion(+)
#Observe the commit --amend replaces the previous commit
$git log

commit 492219352622c01c774a7f0ca4b4e0502615b442
Author: Gokhan
    A better commit message

commit 5dfba117353858439a0b5f22d193c4fc1e6b3a0f
Author: Gokhan <gkhncpn@gmail.com>
    Adding a totally useless file
```

Undoing

- Say you want to unstage

```
$ touch second.txt
$ git add second.txt
$ git status

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

 new file:   second.txt

#unstage
$ git reset HEAD second.txt
$ git status

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

 second.txt
```

Undoing

- Say you want to revert your modifications

```
$ echo “An accidental change”>first.txt
$ cat first.txt
An accidental change

$ git status

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

 modified:   first.txt

$ git checkout -- first.txt
$ cat first.txt
Something useful
```

Default Ignore Behavior

- Say you don’t want the files under target/ directory to be committed

```
# Create a file called .gitignore
# Add target/* as a line
$cat .gitignore
target/*

#stage and commit .gitignore
$git add .gitignore
$git commit -m “Adding .gitignore”

#Create target directory and add a file under it
$mkdir target
$touch target/file.txt
$git status
On branch master
nothing to commit, working directory clean
```

Default Ignore Behavior

- Git automatically ignores 'target' directory

```
# Create a file called .gitignore
# Try to add target/ anyway
$ git add target/
$ git commit -m “This will not be committed"

On branch master
nothing to commit, working directory clean
```

Tagging

- It is common behavior to tag milestones
- Say we finally reach one

```
# git tag to see current tags
$ git tag

# tag the current project
$ git tag v0.1 -m "Project is now in version 0.1"
$ git tag
v0.1

$git show v0.1
Tagger: Gokhan

Project is now in version 0.1

commit bd228761a2e09f70966b27a9ffc7e2142d4bc480
Author: Gokhan

    Adding .gitignore
```

Branches

- A branch is simply a pointer to a commit.
- The ‘master’ is just a branch (created by 'git init’)

```
# Create a branch called newfeature
$ git branch newfeature

# See all branches (* denotes which branch the HEAD
points)
$ git branch -a
* master
  newfeature

#Work on the new branch
$ git checkout newfeature
$ git branch -a
  master
* newfeature
```

Branches

- Do some work on our new branch

```
# Do some work on newfeature
$ touch second.txt
$ git add second.txt
$ git commit -m "Adding second.txt"
$ git log --oneline

7939b73 Adding second.txt
bd22876 Adding .gitignore
4922193 A better commit message
5dfba11 Adding a totally useless file
```

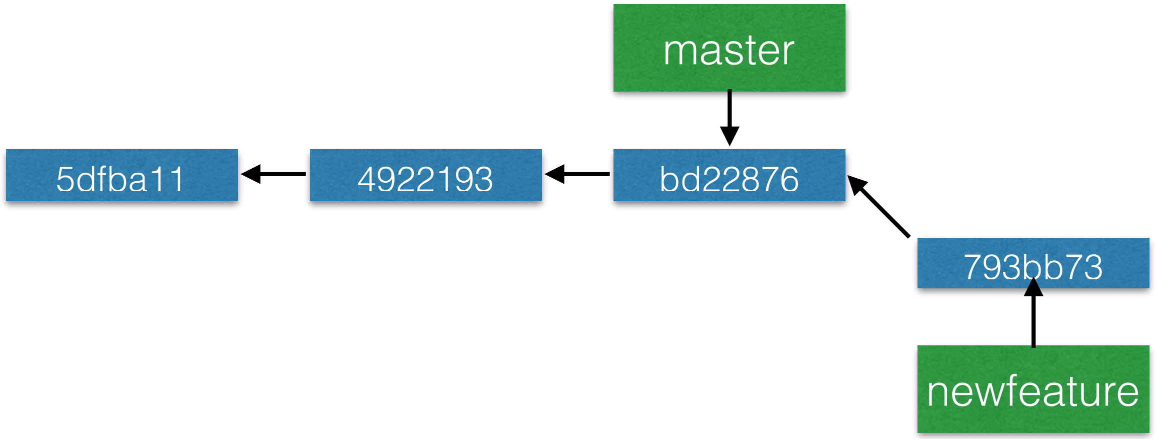
Branches

- Observe master branch

```
# Switch back to the master branch
$ git checkout master
$ git log --oneline

bd22876 Adding .gitignore
4922193 A better commit message
5dfba11 Adding a totally useless file

#Notice that latest commit was not applied to the master
branch
```

Branches

- Current state of the project



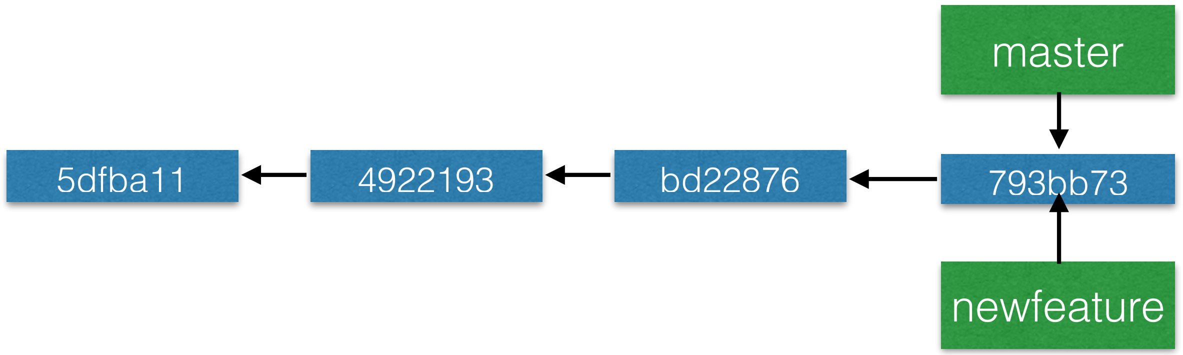
Branches

- Let’s merge our work on ‘newfeature’ into ‘master’

```
# Merge newfeature into master
$ git checkout master
$ git merge newfeature

Updating bd22876..7939b73
Fast-forward
 second.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 second.txt

# Notice bd22876..7939b73
# Notice fast-forward
```

Branches

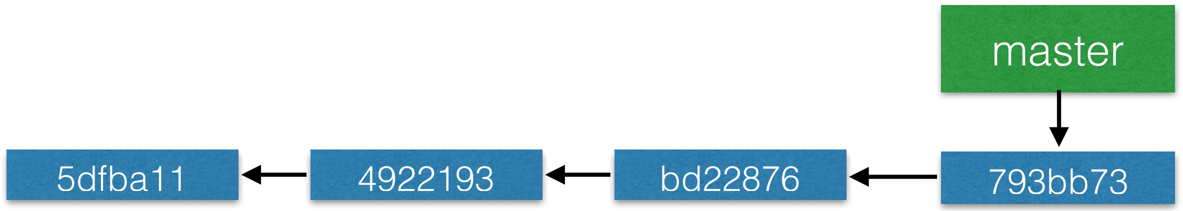
- Current state of the project



Branches

- newfeature is obsolete now, delete it

```
# Delete newfeature since it is fully merged
$ git branch -d newfeature
$ git branch -a
* master
```



Branches

- Note that merge happened in a ‘fast-forward’ way (not even a commit occurred)
- This is because we did not change anything in master after branching from it
- Another scenario: Branch from ‘a’ to ‘b’, make changes in both ‘a’ and ‘b’, then merge ‘b’ into ‘a’

Branches

- Create a new branch from ‘master’ and do some work on it

```
# Create a branch called newfeature
$ git branch newfeature
$ git checkout newfeature
$ touch third.txt
$ git add third.txt
$ git commit -m “Adding third.txt”

$ git log --oneline -n 2
0206eab Adding third.txt
7939b73 Adding second.txt

$ git checkout master
$ git log --oneline -n 2
7939b73 Adding second.txt
bd22876 Adding .gitignore
```

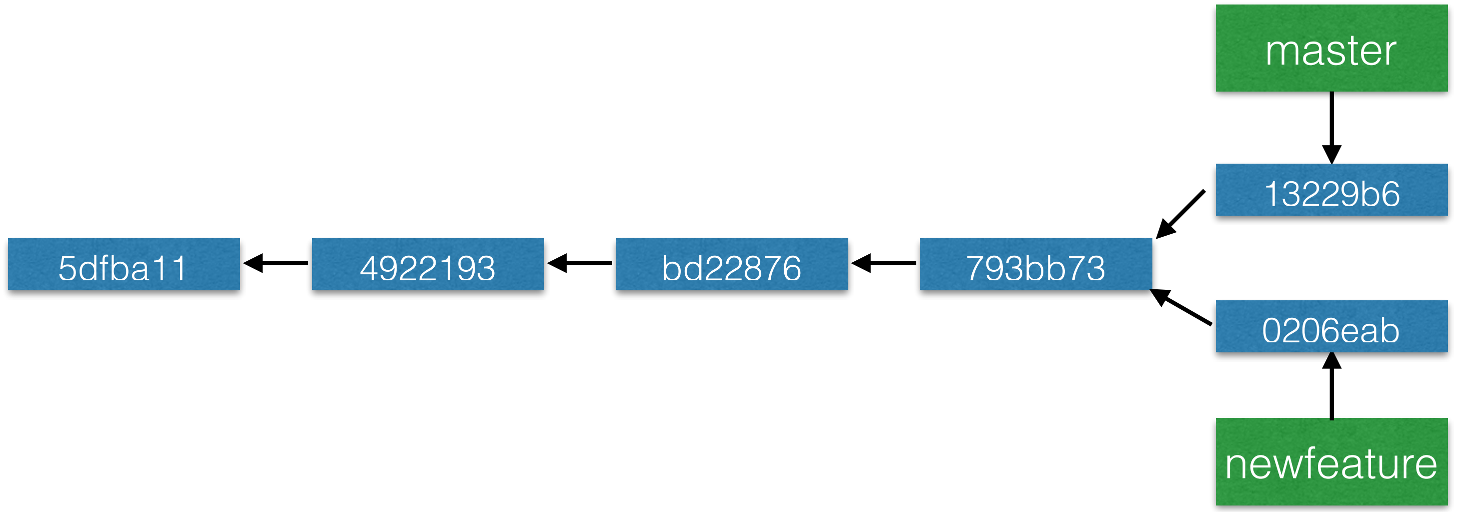
Branches

- Before merging ‘newfeature’ into ‘master’, do some work on ‘master’

```
# Do some work on master
$ git checkout master
$ touch fourth.txt
$ git add fourth.txt
$ git commit -m “Adding fourth.txt”

$ git log --oneline -n 2
13229b6 Adding fourth.txt
7939b73 Adding second.txt
```

Branches

- Current state of the project



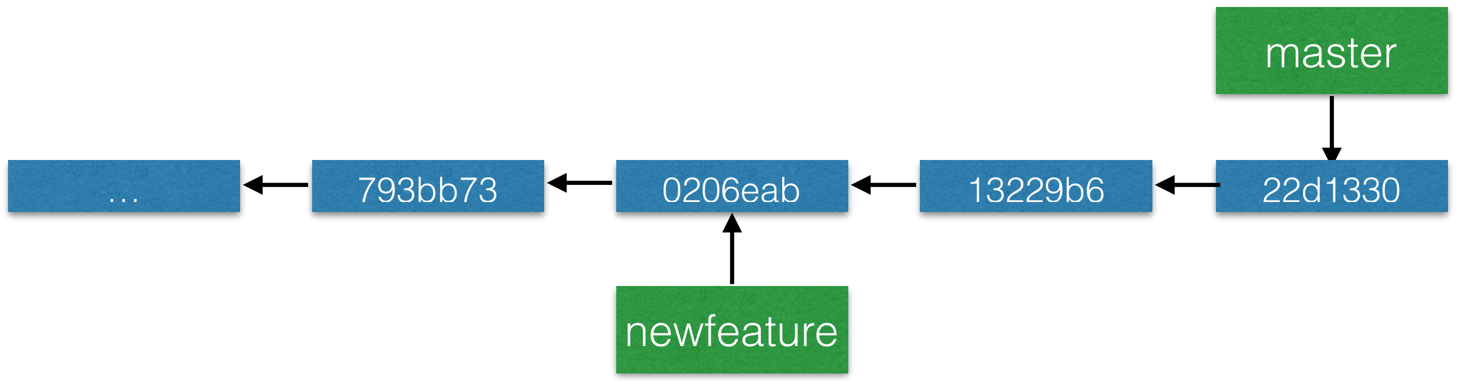
Branches

- Let’s merge ‘newfeature’ into ‘master’

```
# Merge newfeature into master
# This results in a new commit in master
$ git checkout master
$ git merge newfeature
$ git log --oneline -n 4

22d1330 Merge branch 'newfeature'
13229b6 Adding fourth.txt
0206eab Adding third.txt
7939b73 Adding second.txt

$git checkout newfeature
$ git log --oneline -n 2
0206eab Adding third.txt
7939b73 Adding second.txt
```

Branches

- Current state of the project



Branches

- ‘newfeature’ is obsolete now, delete it

```
#Delete newfeature since it is fully merged into master
$ git checkout master
$ git branch -d newfeature
$ git branch -a

* master
```

Branches

- Another scenario is when the same file is modified in both branches

Branches

- Create a branch and modify first.txt

```
$ git branch newfeature
$ git checkout newfeature
$ echo ‘Modifications for new feature’>> first.txt
$ git commit -a -m “cool feature is applied”
$ git log --oneline -n 3

fee3b9b cool feature is applied
22d1330 Merge branch 'newfeature'
13229b6 Adding fourth.txt

#have a look at the content of first.txt
$cat first.txt
Something useful
Modifications for new feature
```

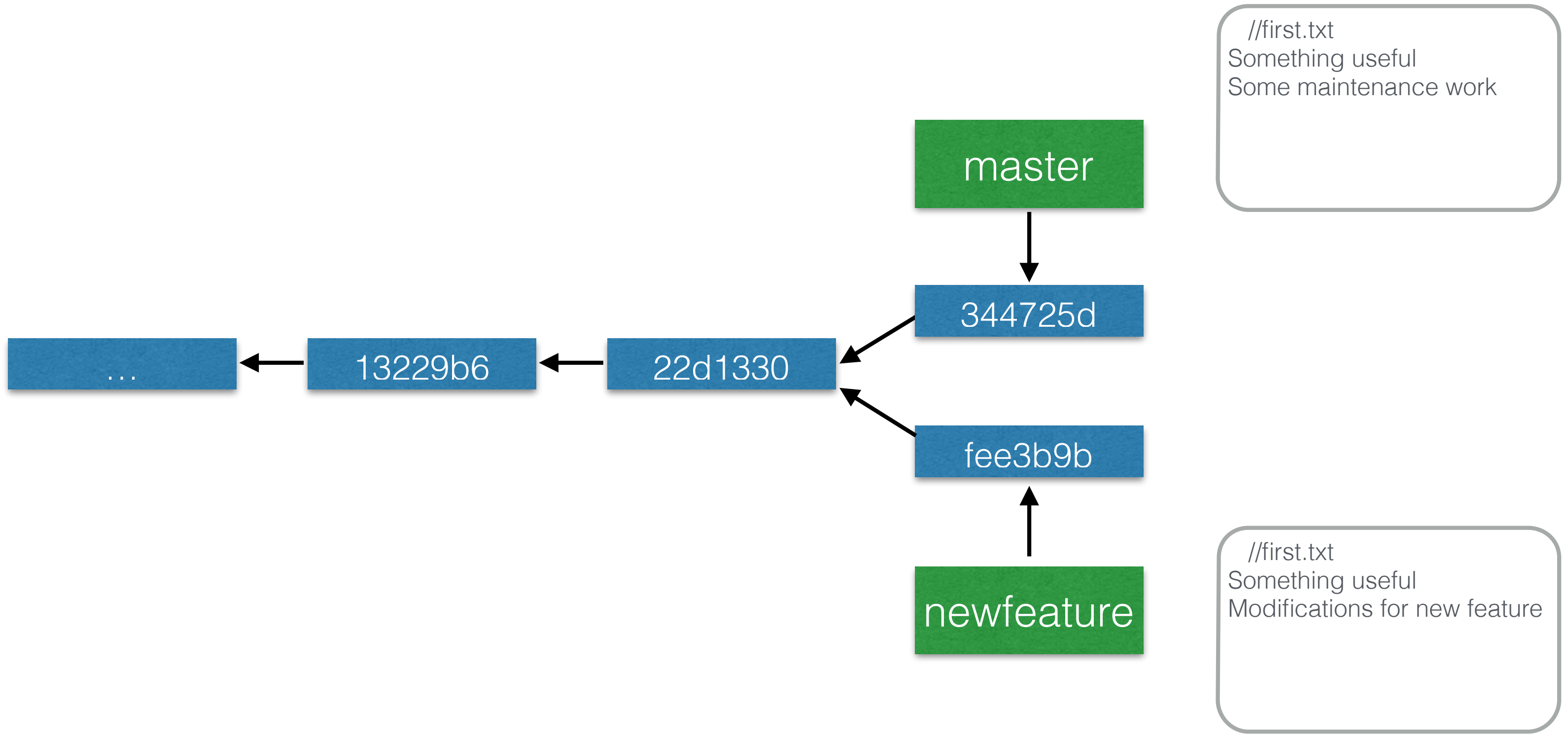
Branches

- Switch back to ‘master’ and modify first.txt before merging ‘newfeature’ into ‘master’

```
$ git checkout master
#have a look at the content of first.txt
$ cat first.txt
Something useful
#modify first.txt
$ echo “Some maintenance work”>>first.txt
$ git commit -a -m “Maintenance”
$ git log --oneline -n 3
344725d Maintenance
22d1330 Merge branch 'newfeature'
13229b6 Adding fourth.txt

$ cat first.txt
Something useful
Some maintenance work
```

Branches

- Current state of the project



//first.txt
Something useful
Some maintenance work

//first.txt
Something useful
Modifications for new feature

Branches

- Let’s try to merge ‘newfeature’ into ‘master’

```
# Merge newfeature into master
$ git checkout master
# This fails due to merge conflict, you need to resolve
it
$ git merge newfeature
CONFLICT (content): Merge conflict in first.txt
Automatic merge failed; fix conflicts and then commit
the result.
```

Branches

- But git gives you some hint, have a look at first.txt

```
$ cat first.txt

Something useful
<<<<<<< HEAD
Some maintenance work
=======
Modifications for new feature
>>>>>>> newfeature


#Remember Modifications for new feature line was added
in the ‘newfeature’ branch, which git tells us
```

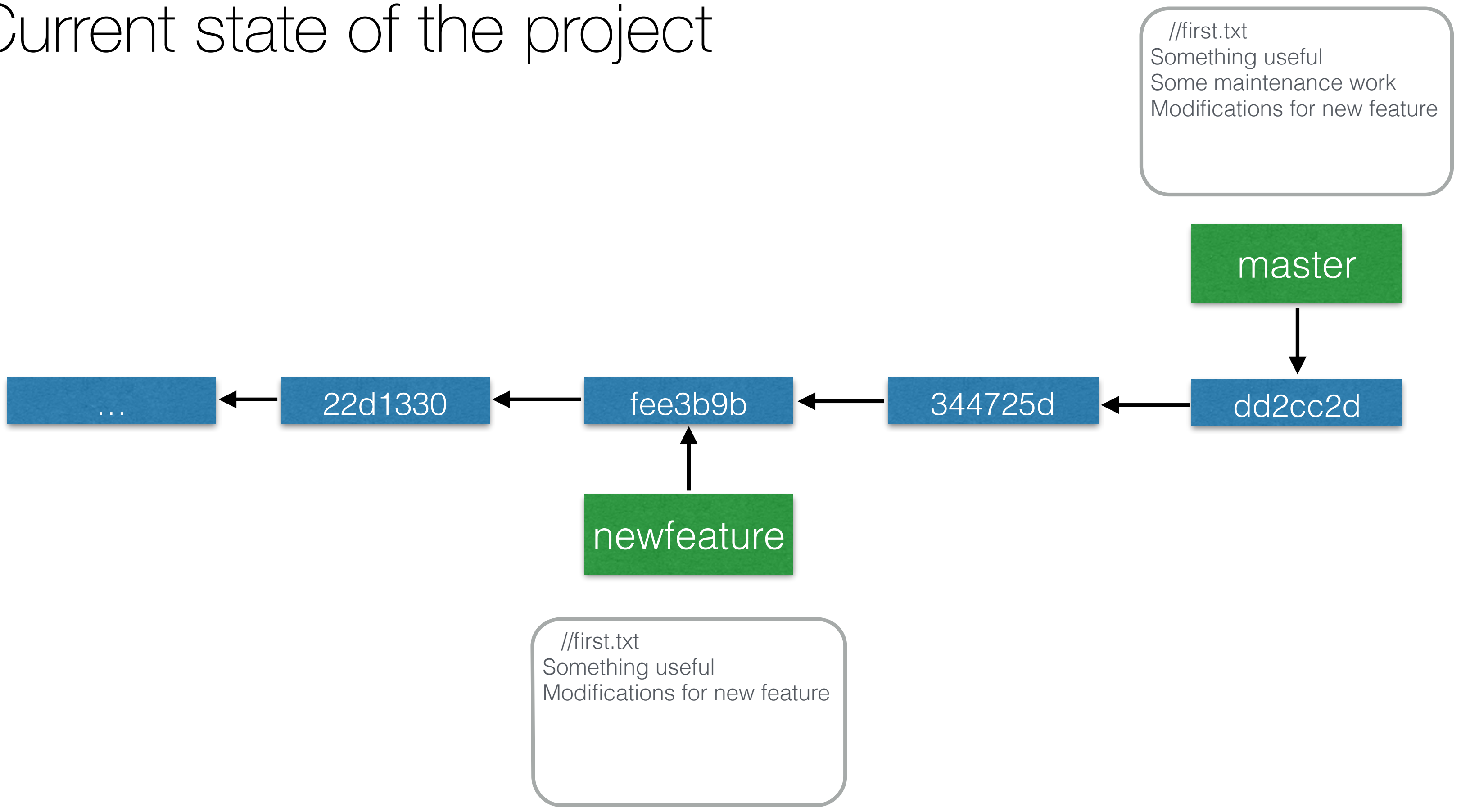
Branches

- Manually resolve the conflict (i.e. edit the file using your editor), and then merge

```
#Manually resolve conflict using your editor
$ cat first.txt
Something useful
Some maintenance work
Modifications for new feature

#Commit your edits
$ git commit -a -m “Conflicts resolved and merge of
newfeature performed”

$ git log --oneline -n 3

dd2cc2d Conflicts resolved and merge of newfeature
performed
344725d Maintenance
fee3b9b cool feature is applied
```

Branches

- Current state of the project



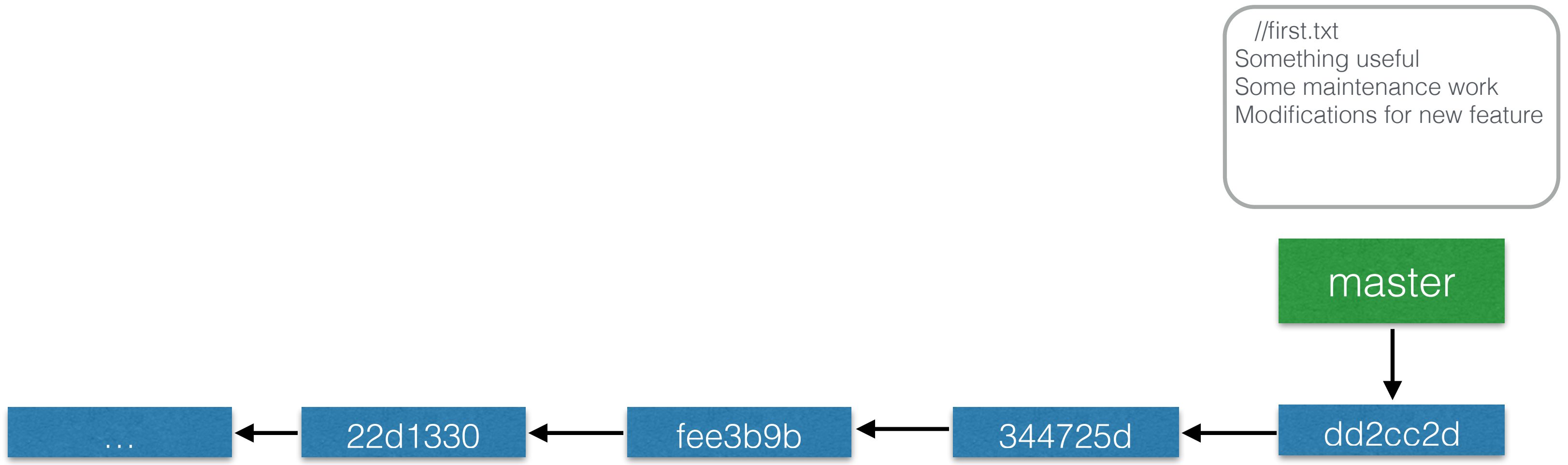
Branches

- ‘newfeature’ is obsolete now, delete it

```
#Delete newfeature since it is fully merged into master
$ git checkout master
$ git branch -d newfeature
$ git branch -a

* master
```

Branches

- Current state of the project



On Resolving Conflicts

- Your IDE almost surely provides more convenient ways (i.e. graphical tools) for resolving conflicts, use it
- There are many options in git merge (such as you can ignore whitespaces and merge smoothly if they are the causes of the conflict), but we can't cover them all
- git merge can be run with -Xours or -Xtheirs to automatically resolve conflict by accepting one particular version
- When in one branch the file is deleted, and in the other it is modified, you can either resolve the conflict by keeping the file, or deleting it

Aside: Undoing a commit

- git revert <commitnumber> allows you to undo a commit
- Git does it by applying a new commit, that undoes to the other

Working as a Team

- There are many workflows possible, but I am going to cover the common workflow* used in the group projects for this particular course

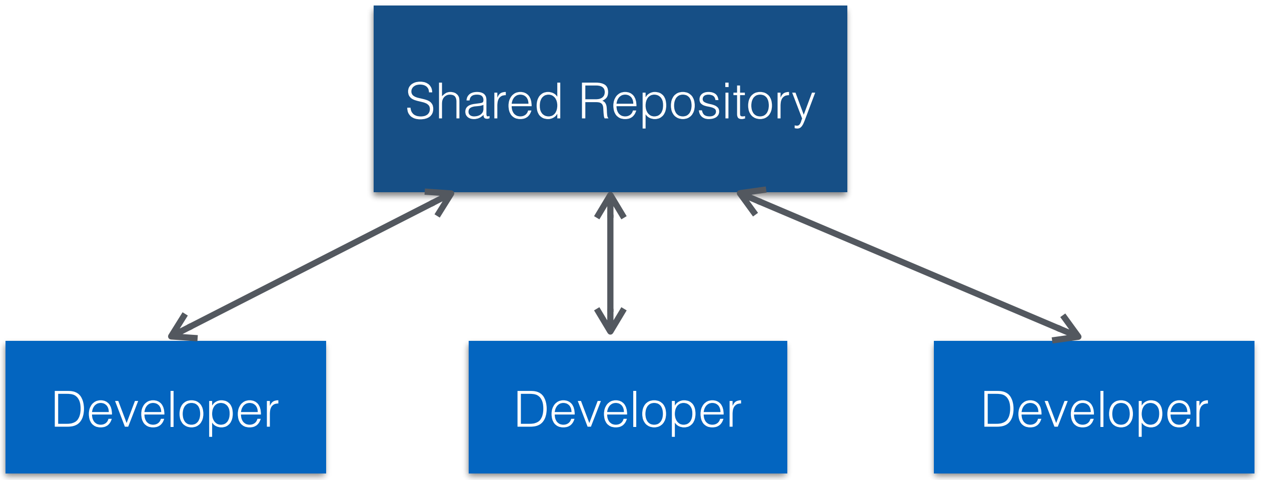


*Figure adapted from http://git-scm.org/about/distributed

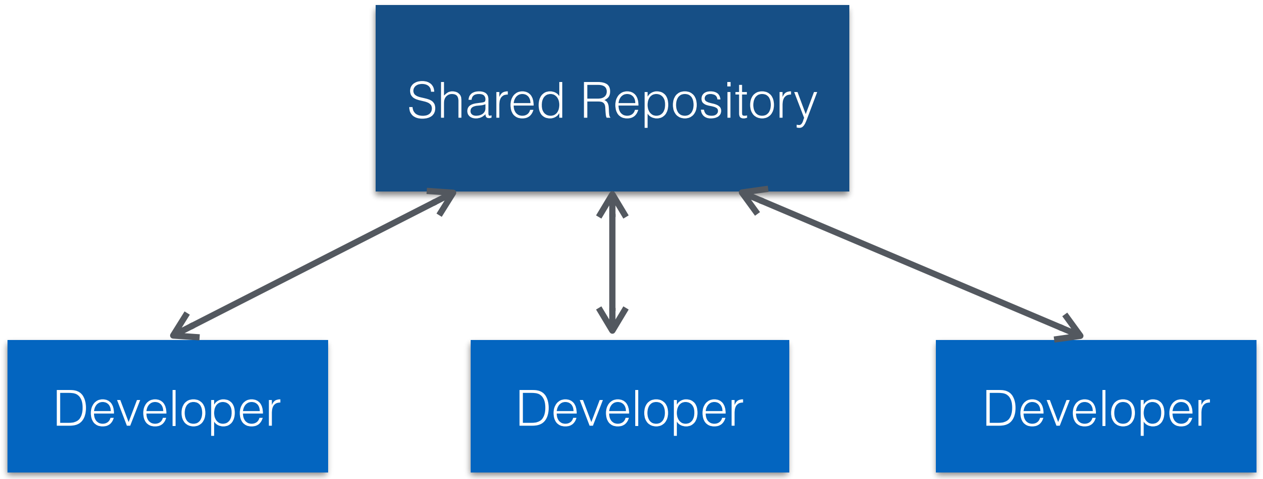
Working as a Team

- Initially, somebody creates a repository for common access at github (or Google code)

Shared Repository

Working as a Team

- Team members ‘clone' the entire repository
- Note that this is not a client-server architecture (unlike SVN-style VCSs)



Working with Remotes

- Often, the running-prototype is maintained in a remote machine (a shared repository) that provides access via HTTP or SSH
- github, Google Code are examples
- Developers generally ‘clone’ a copy to their local workstations, and ‘push’ back their contributions
- This is handled by remotes

Cloning from Shared Repository

- Just clone it

```
#clone from github
$ git clone https://github.com/cmpe451/toy.git

Cloning into 'toy'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.

#git automatically clones the project into the directory
#called 'toy'
```

Cloning from Shared Repository

```
# Observe logs
$ git log --oneline
e8bada4 Initial commit #This is cloned from the remote

# List remotes
$ git remote
origin
# git automatically give the name origin to the remote
# you may have run git clone -o <remotename>

# Observe branches
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master

#1. We are on branch master (git automatically created
this for us)
#2. On OUR computer, a branch called origin/master is
created, and it is a remote-tracking-branch
#3. origin/master is not updated automatically, it is
LOCAL
```

Cloning from Shared Repository

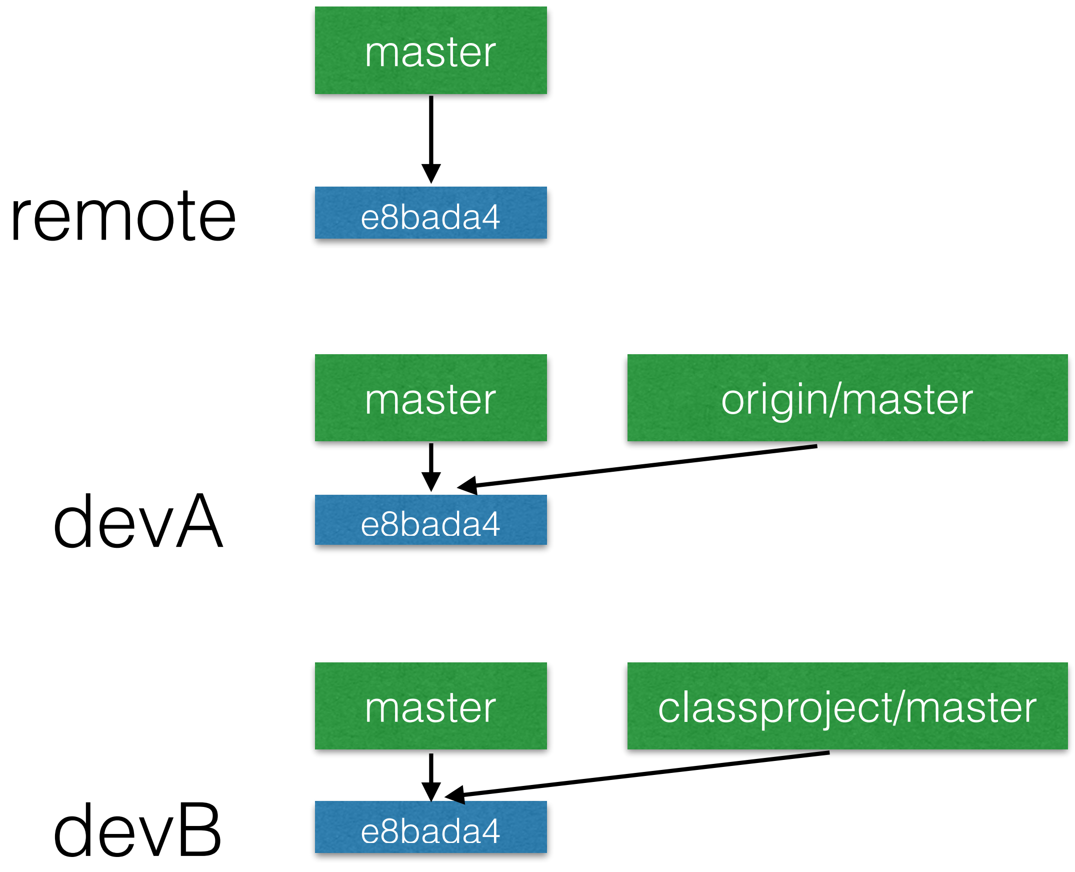
```
# Another developer clones the project
$ git clone -o classproject https://github.com/cmpe451/
toy.git
$ cd toy
$ git remote
classproject

$ git branch -a
* master
  remotes/classproject/HEAD -> classproject/master
  remotes/classproject/master
```

Fetch-Merge and Push

- To update our *remote tracking branch*, a.k.a. ‘remote/rtb’, we fetch from the remote, and then merge the remote tracking branch into the ‘rtb’
- A shortcut is git pull, which is equivalent to git fetch + git merge
- When we do our work on rtb (*tracking branch*), after committing, we can push it to the ‘rtb’ branch in the remote with git push

Fetch-Merge and Push

- Current state of the project

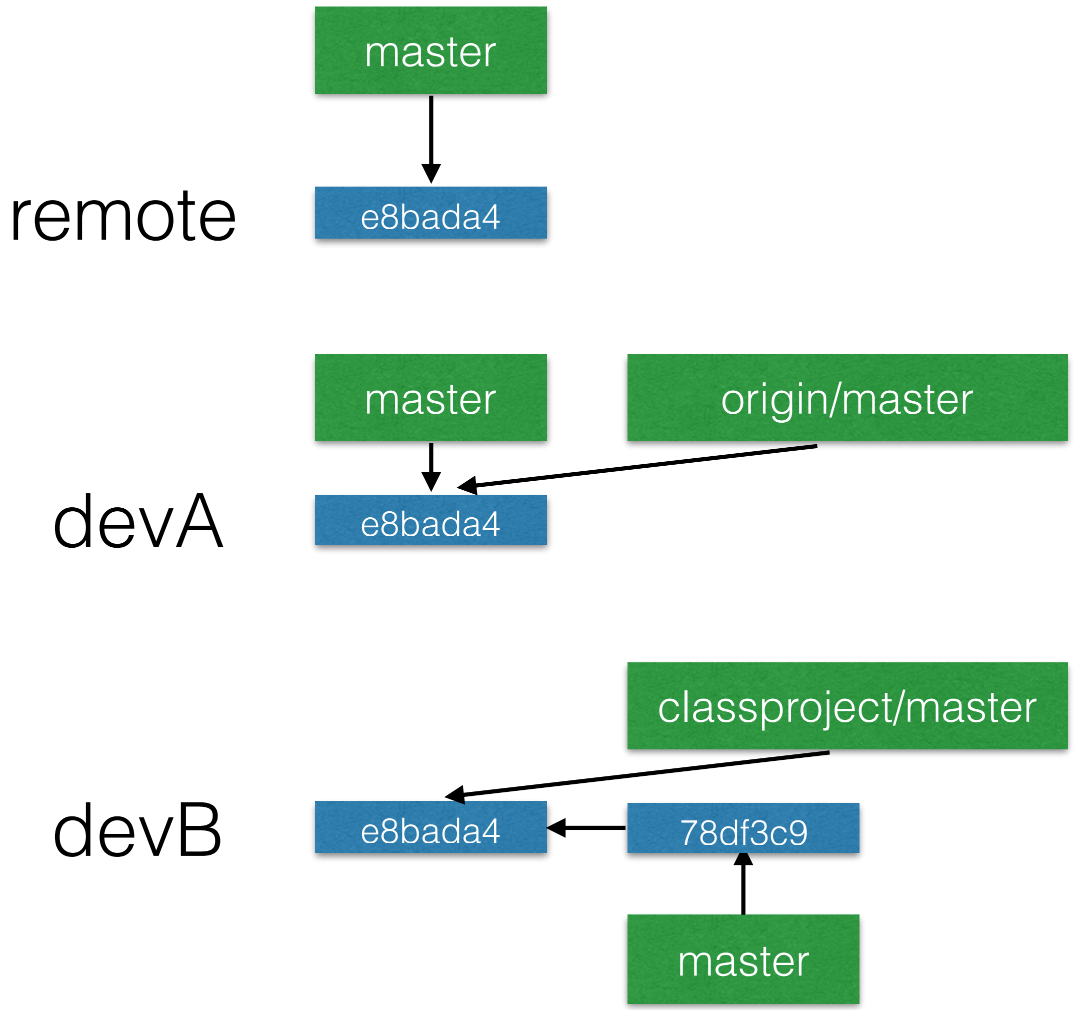


Working as a Team

```
#devB makes some changes and commits
$ touch first.txt
$ git add first.txt
$ git commit -m “Adding first.txt”
$ git log --oneline
78df3c9 Adding first.txt
e8bada4 Initial commit

#have a look at the classproject/master branch
#This is just for illustration purposes! Normally, you
#have nothing to do with the remote tracking branch,
#they are READ-ONLY
$ git checkout classproject/master
$ git log --oneline
e8bada4 Initial commit
```

Working as a Team

- Current state of the project

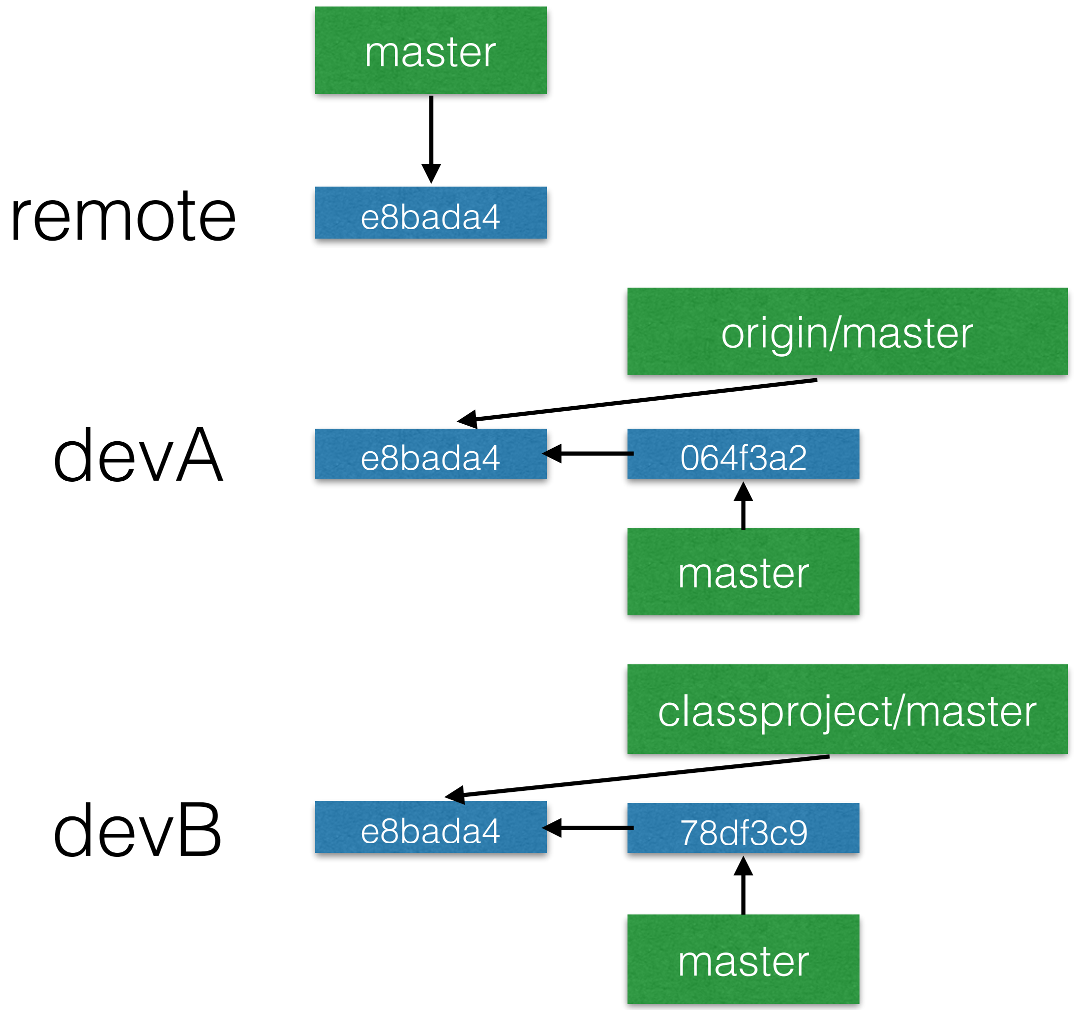


Working as a Team

```
#In the meantime, devA makes some changes and commits
them
$ touch second.txt
$ git add second.txt
$ git commit -m “Adding second.txt”
$ git log --oneline
064f3a2 Adding second
e8bada4 Initial commit
```

Working as a Team

- Current state of the project

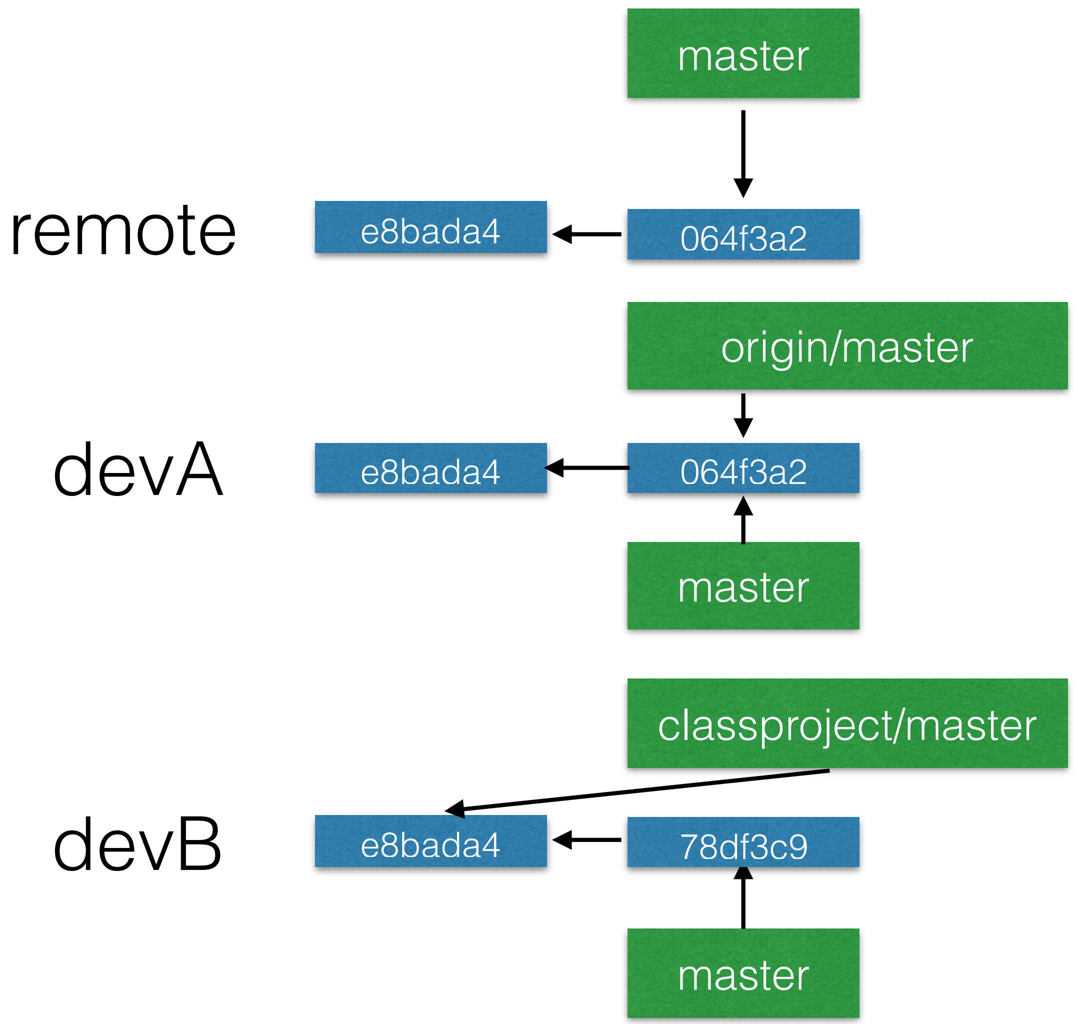


Working as a Team

```
#devA wants to push his changes to the repository
#Ideally he first updates his local version of the
#software with git fetch+merge, and then pushes to the
#remote. Let’s skip this step once (since we know that
devA’s computer is sync with the remote)

$ git push origin

#this pushes the master branch to the master branch in
#the remote origin

#this pushes the LOCAL branch master to the REMOTE
branch master (git push origin <local_branch>)
$ git push origin master

# this pushes the LOCAL branch master to the REMOTE
branch master (git push origin <lbranch>:<rbranch>)
$git push origin master:master

# pushing also updates the remote-tracking-branch
```

Working as a Team

- Current state of the project



Working as a Team

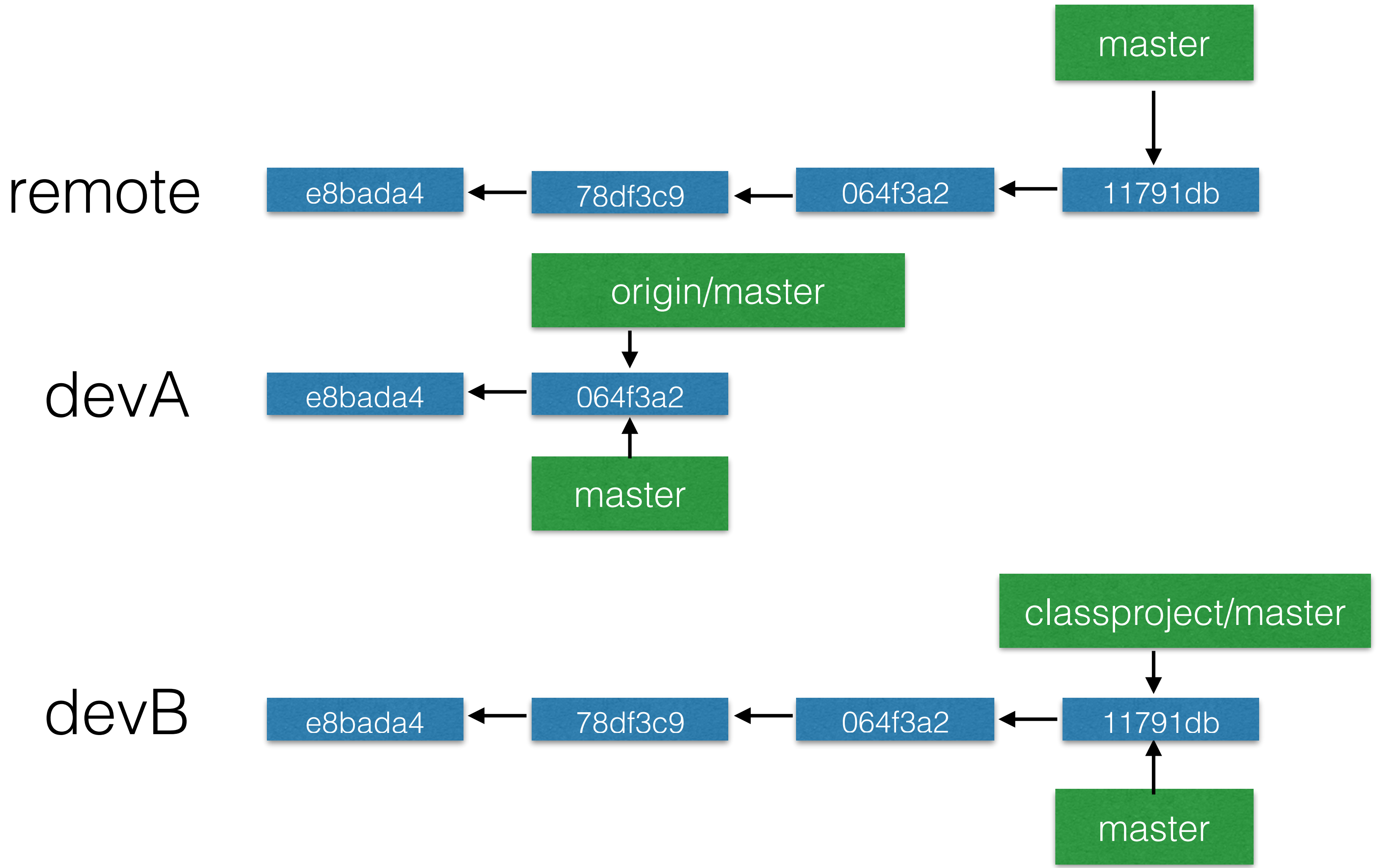
```
#for devB to push his commits, he needs to fetch from
remote first
#git fetch <remote> <rbranch> fetches the rbranch from
the remote, and creates a remote tracking branch called
remote/rbranch if it doesn’t exist, if remote/rbranch
exists, merges the changes in remote into remote/rbranch
#devB then should merge the work he fetched into his
local brach

$git fetch classproject master
$git checkout master
$git merge classproject/master
#This triggered a commit, since devB made additional
#changes before
```

Working as a Team

```
#Current state of the local master branch of devB
$git log --oneline -n 3
11791db Merge remote-tracking branch 'classproject/
master'
064f3a2 Adding second #pushed by devA, remember?
78df3c9 Adding first.txt

#Current state of the remote tracking branch
classproject/master
$git checkout classproject/master
$git log --oneline
064f3a2 Adding second
e8bada4 Initial commit

#We fetched from the remote to the remote tracking
branch, and merged it into our local branch, but remote
tracking branch, and the master branch in the remote are
still unaware of our changes. We need to push our
commits
$ git checkout master
$ git push classproject
```

Working as a Team

- Current state of the project

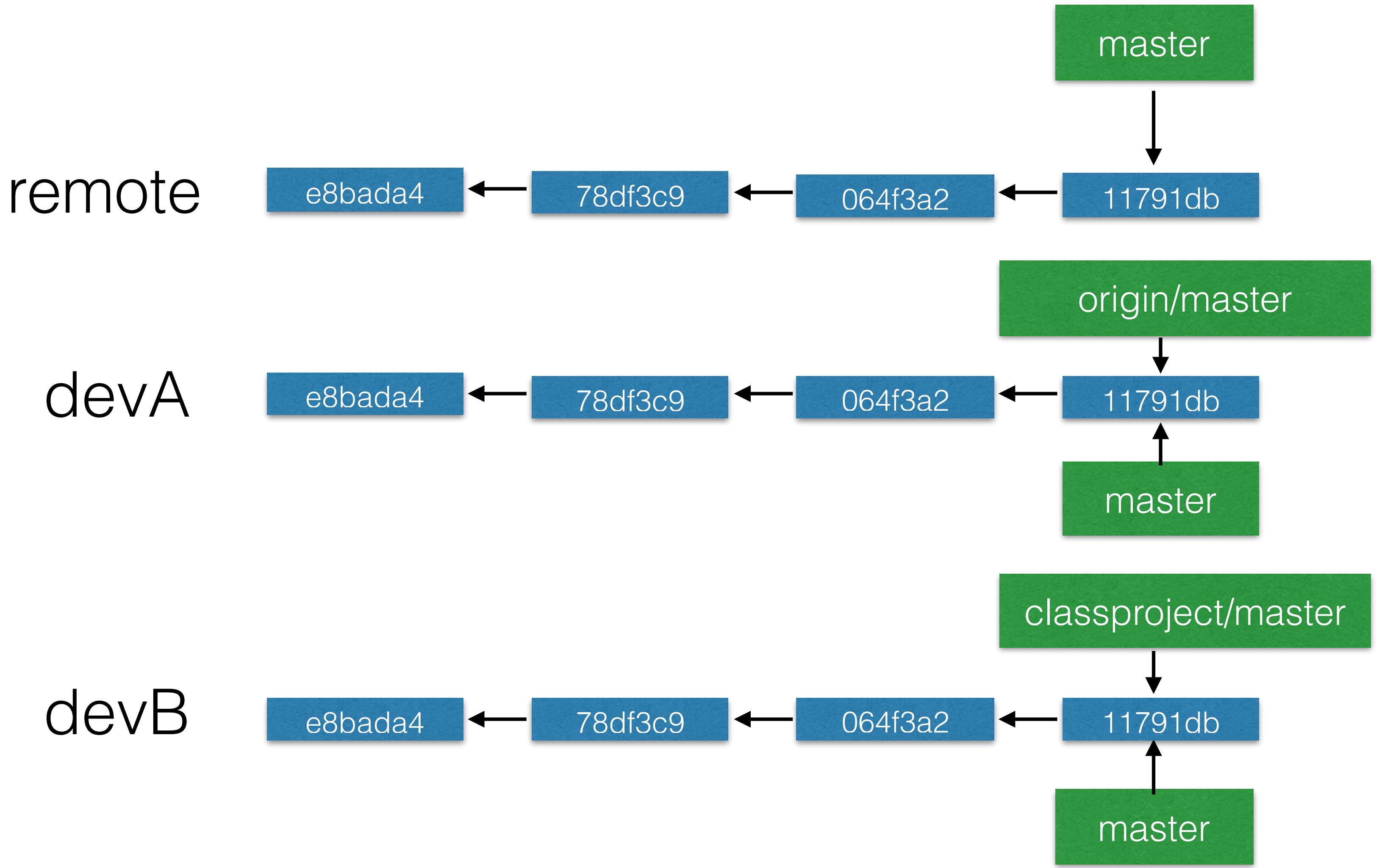


Working as a Team

```
#devA pulls the changes from the remote origin
$ git log --oneline
064f3a2 Adding second
e8bada4 Initial commit

$git pull origin master
11791db Merge remote-tracking branch 'classproject/
master'
064f3a2 Adding second
78df3c9 Adding first.txt
e8bada4 Initial commit
```

Working as a Team

- Current state of the project



Working with Remotes

- You can git push to a non-existing remote branch, it is created on the fly
- You can pull/push tags
- You can pull from remote branches that you’re not tracking, the READ-ONLY remote-tracking branch and the tracking branch are created automatically
- These are the basics, see the documentation for more information
- Again, once you understand the basic concepts, use your IDE’s git integration, they are really handy

Suggestions

- When implementing new features, or fixing issues, always work on new branches
- After you're done, merge it into master, then commit and push
- If any file is irrelevant to the build, don't version it
  - IDE-specific files, OS-specific files
  - 3rd party archives: They are large, they don't change, and cannot properly versioned
  - There are better tools to handle 3rd party dependencies (e.g. maven)