



Octocat



# Git and GitHub

Dan McDonald, PhD  
Information Systems and Technology  
Utah Valley University



# Agenda

---

- Git Overview
- Git Content Tracking
- Git Branches
- Git Rebasing
- Distributed Version Control



# Git and GitHub



- GitHub is a code hosting platform for version control and collaboration
- Git was created by Linus Torvalds
- A Distributed Revision Control System
- GitHub is a development platform used by 31 million developers





# GitHub Concepts



- Repository - used to organize a single project
  - contains folders, images, videos, spreadsheets, etc
  - recommended to include a README file
- Commit
  - Adds content from staging area to Git database (saved changes)
- Branch
  - A way to work on different versions of a repository at one time
- Open a Pull Request
  - A way to propose your changes to others
- Merge a Pull Request



- Presentation is adapted from a Pluralsight course by Paolo Perrota titled "How Git Works"



Paolo Perrota

@nusco



# Porcelain Commands

---



- git add
- git commit
- git push
- git pull
- git branch
- git checkout
- git merge
- git rebase



# Plumbing Commands

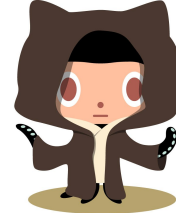
---



- git cat-file
- git hash-object
- git count-objects



# Git is like an onion



- To become a Git master, understand the model, not just the commands
- a Distributed Revision Control System







## Peeling back a layer of the onion

---

- A Revision Control System (forget about Distributed)
  - branches
  - merges
  - rebases





# Peeling back another layer

---

- a Stupid Content Tracker
  - tracks content files or directories
  - still has versioning and commit





# Git Core

---

- At the core, Git is a persistent map (Stupid content)
  - No versioning or commit
  - A simple persistent map that maps keys to values





# Git is a Map with Values and Keys

---

- A table with values and keys
- Values are sequences of bytes
- Given a value, Git creates a key (hash)
- Git uses SHA1 algorithm
  - 20 bytes in hexadecimal format
  - 40 hex digits

Any sequence of bytes



SHA1 hash



# SHA1

- Every piece of content/directory has its own SHA1
- There is only 1 hash for string "Apple Pie"

Command Prompt

```
C:\Users\10623312>echo "apple pie" | git hash-object --stdin  
bc1feb0777d240f29d15028256afe8672c6ec96f  
C:\Users\10623312>
```

"Apple Pie"



23991897e13e47ed0adb91a0082c31c82fe0cbe5



# Keys are SHA1s and Values are content

---

- Every object in Git has its own SHA1
- Putting "Apple Pie" in a file and store the file in Git, then the SHA1 we just generated will identify the file
- Directories and commits have their own SHA1
- What happens if the SHA1 collide?
  - Not likely (chance of winning lottery 6 consecutive times)
  - SHA1 are unique in the universe
  - You could put all the data you ever wrote in your life in the same Git repository and Git would assign a unique SHA1 to each version of each file and each folder



## Git as a persistent map

---

- Not enough just to create the SHA1, we must store it
- It gets stored in a repository
- create a repository in a folder

Command Prompt

```
C:\Users\10623312\gitdemo>git init
Initialized empty Git repository in C:/Users/10623312/gitdemo/.git/
C:\Users\10623312\gitdemo>
```

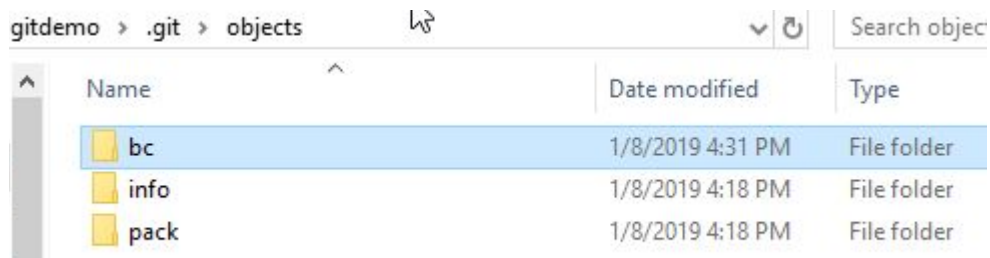
- now save the hash

```
C:\Users\10623312\gitdemo>echo "apple pie" | git hash-object --stdin -w
bc1feb0777d240f29d15028256afe8672c6ec96f
```



# Git as a persistent map

- Where did the SHA1 go?
  - The folder name is the first 2 characters of the SHA1



Name	Date modified	Type
bc	1/8/2019 4:31 PM	File folder
info	1/8/2019 4:18 PM	File folder
pack	1/8/2019 4:18 PM	File folder

- The file name is the remaining digits of the SHA1
- This is a trick to avoid files piling up inside the same directory
- Inside the file is the original "apple pie" data
- This is what Git calls a blob of data (a generic piece of content)





# Git is a Persistent Map

---

- The content in the file name has been mangled a bit
  - Git added a small header
  - Git compressed the content
  - We can't just open the file
- We can run a low-level command to see the content

Command Prompt

```
C:\Users\10623312\gitdemo>git cat-file bc1feb0777d240f29d15028256afe8672c6ec96f -t  
blob
```

```
C:\Users\10623312\gitdemo>git cat-file bc1feb0777d240f29d15028256afe8672c6ec96f -p  
"apple pie"
```



# Git as a Stupid Content Tracker

---

- Create a directory called `cookbook`
- Inside the directory, create a file called `menu.txt` and a folder called `recipes`
  - `menu.txt` should just contain the text "Apple Pie"
- Inside the `recipes` folder, create a file called `README.txt` and a file called `apple_pie.txt`
- `README.txt` should contain the text "Put your recipes in this directory, one recipe per file"
- Inside the `apple_pie.txt` file, put the text "Apple Pie"



# Git as a Stupid Content Tracker

---

```
mkdir cookbook
cd cookbook
echo Apple Pie>menu.txt
mkdir recipes
cd recipes
echo Apple Pie>apple_pie.txt
echo Put your recipes in this directory, one recipe
per file.>README.txt
cd ..
tree .
git init
```



# Run `git status`

- The object database is empty

```
C:\Users\10623312\cookbook>git status
On branch master

No commits yet

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

        menu.txt
        recipes/

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

- To commit a file, it has to be in the staging area first
  - Staging area is like a launch pad
  - Whatever is in the staging area will get in the next commit



# Run git add

- Using git add

Command Prompt

```
C:\Users\10623312\cookbook>git add menu.txt
warning: CRLF will be replaced by LF in menu.txt.
The file will have its original line endings in your working directory.

C:\Users\10623312\cookbook>git add recipes/
warning: CRLF will be replaced by LF in recipes/README.txt.
The file will have its original line endings in your working directory.
warning: CRLF will be replaced by LF in recipes/apple_pie.txt.
The file will have its original line endings in your working directory.

C:\Users\10623312\cookbook>git status
On branch master

No commits yet

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

        new file:   menu.txt
        new file:   recipes/README.txt
        new file:   recipes/apple_pie.txt
```

- Using git commit

```
C:\Users\10623312\cookbook>git commit -m "First commit!"
[master (root-commit) 45e4ccf] First commit!
Committer: Daniel McDonald <10623312@uvu.edu>
```

```
3 files changed, 3 insertions(+)
create mode 100644 menu.txt
create mode 100644 recipes/README.txt
create mode 100644 recipes/apple_pie.txt
```



# Examine the commit

---

- Staging area is now clean

```
C:\Users\10623312\cookbook>git status
On branch master
nothing to commit, working tree clean
```

- Run `git log` to see existing commits

```
C:\Users\10623312\cookbook>git log
commit 45e4ccfc1b137043184d233733a62cd231f5cb3 (HEAD -> master)
Author: Daniel McDonald <10623312@uvu.edu>
Date:   Wed Jan 9 12:41:34 2019 -0700

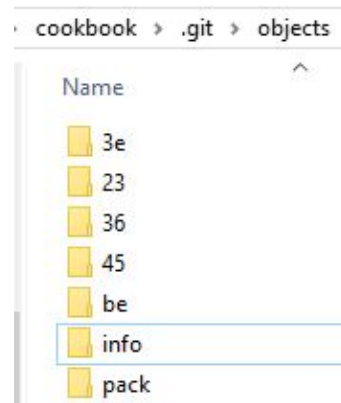
    First commit!
```

- Note the start of the commit hash (i.e. 45)



## Digging deeper

- Go into the objects database



- The commit is compressed just like a blob
- Run `git cat-file`

```
C:\Users\10623312\cookbook>git cat-file -p 45e4ccfc1b1352043184d233733a62cd231f5cb3
tree be4d5bfce489a2591e7fed5c672f9e52cd695a43
author Daniel McDonald <10623312@uvu.edu> 1547062894 -0700
committer Daniel McDonald <10623312@uvu.edu> 1547062894 -0700

First commit!
```



# What is a commit?

---

- It is a simple short piece of text
- Git generates the text and stores it like a blob
  - Git generates its SHA1
  - Git adds a small header to the text to identify it as a commit
  - Git compresses the text
  - Git stores it as a file in the object database
- The commit text contains metadata about the commit
  - author, committer, date
- The commit also contains the SHA1 of a tree





# What is a tree?

---

- A tree is a directory stored in Git
- The commit is pointing at the root directory of the project
- What does the tree look like?
  - A list of SHA1
  - In our case, a blob and another tree
  - shows the names
  - shows access permissions

```
C:\Users\10623312\cookbook>git cat-file -p be4d5bfce489a2591e7fed5c672f9e52cd695a43
100644 blob 23991897e13e47ed0adb91a0082c31c82fe0cbe5    menu.txt
040000 tree 3ee76fde69b730530f1682f1f51789e89cf30500    recipes
```



# What is a tree?

---

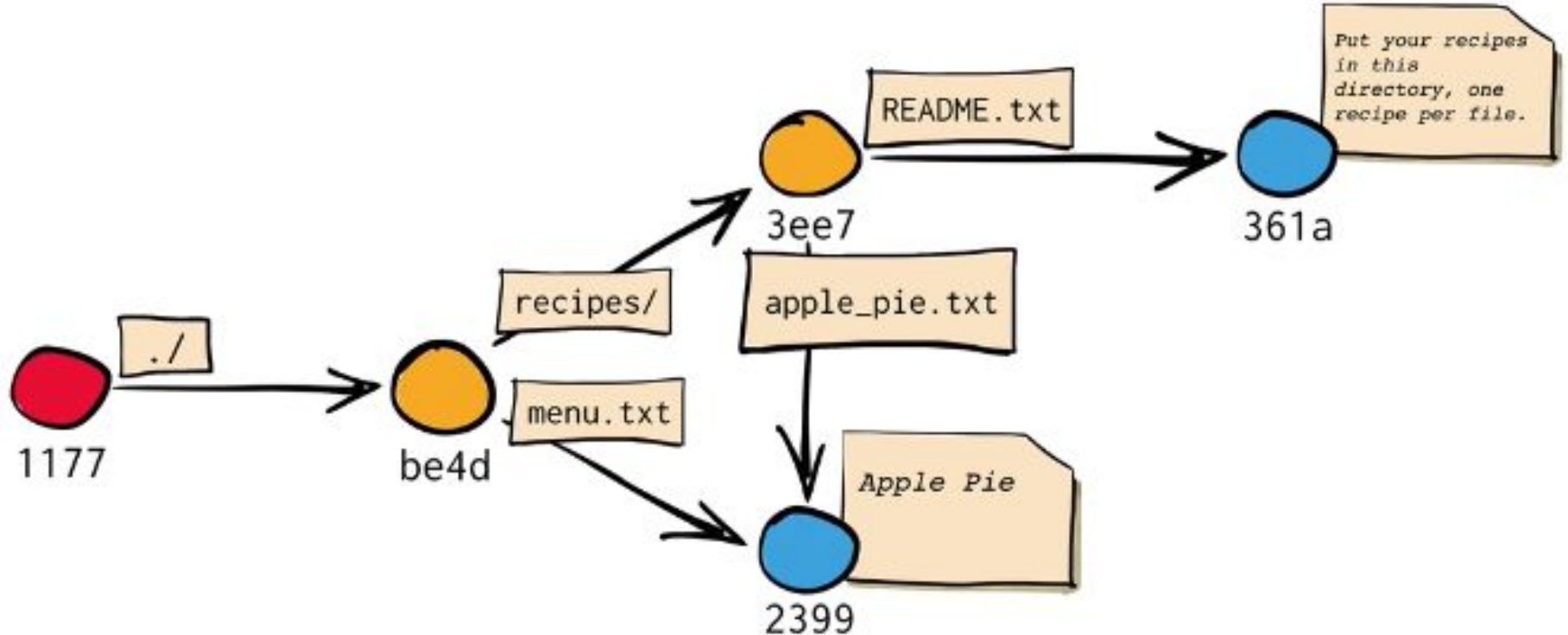
- Now `git cat-file` the recipes directory

```
C:\Users\10623312\cookbook>git cat-file -p 3ee76fde69b730530f1682f1f51789e89cf30500
100644 blob 361af858632ee7d8d8f9c4022ccaf61fc8d4799c    README.txt
100644 blob 23991897e13e47e10adb91a0082c31c82fe0cbe5    apple_pie.txt
```

- So, a blob is not really a file, it is the CONTENT of a file
- The filename and file permissions are not stored in the blob, but rather in the tree that points to the blob
- If a blob is the same, then it is reused



# The Object Database





# Git Versioning

- First, add another food to the `menu.txt` file
- Run `git status` and see the changed file
- Run `git add`
- Run `git status`

```
C:\Users\10623312\cookbook>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:   menu.txt

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

C:\Users\10623312\cookbook>git add menu.txt
warning: CRLF will be replaced by LF in menu.txt.
The file will have its original line endings in your working directory.

C:\Users\10623312\cookbook>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   menu.txt
```



# Git Versioning

---

- Run `git commit -m "Add cake"`
- Run `git status`
- Run `git log`
- Run `cat-file`

```
C:\Users\10623312\cookbook>git status
On branch master
nothing to commit, working tree clean

C:\Users\10623312\cookbook>git log
commit b5a21a1dd9a3ee9161ce4b3b6f3ef1473d662172 (HEAD -> master)
Author: Daniel McDonald <10623312@uvu.edu>
Date:   Wed Jan 9 14:22:13 2019 -0700

    Add cake

commit 45e4ccfc1b1352043184d233733a62cd231f5cb3
Author: Daniel McDonald <10623312@uvu.edu>
Date:   Wed Jan 9 12:41:34 2019 -0700

    First commit!
```



## The second commit

---

- The new commit has a parent
- The parent is the first commit

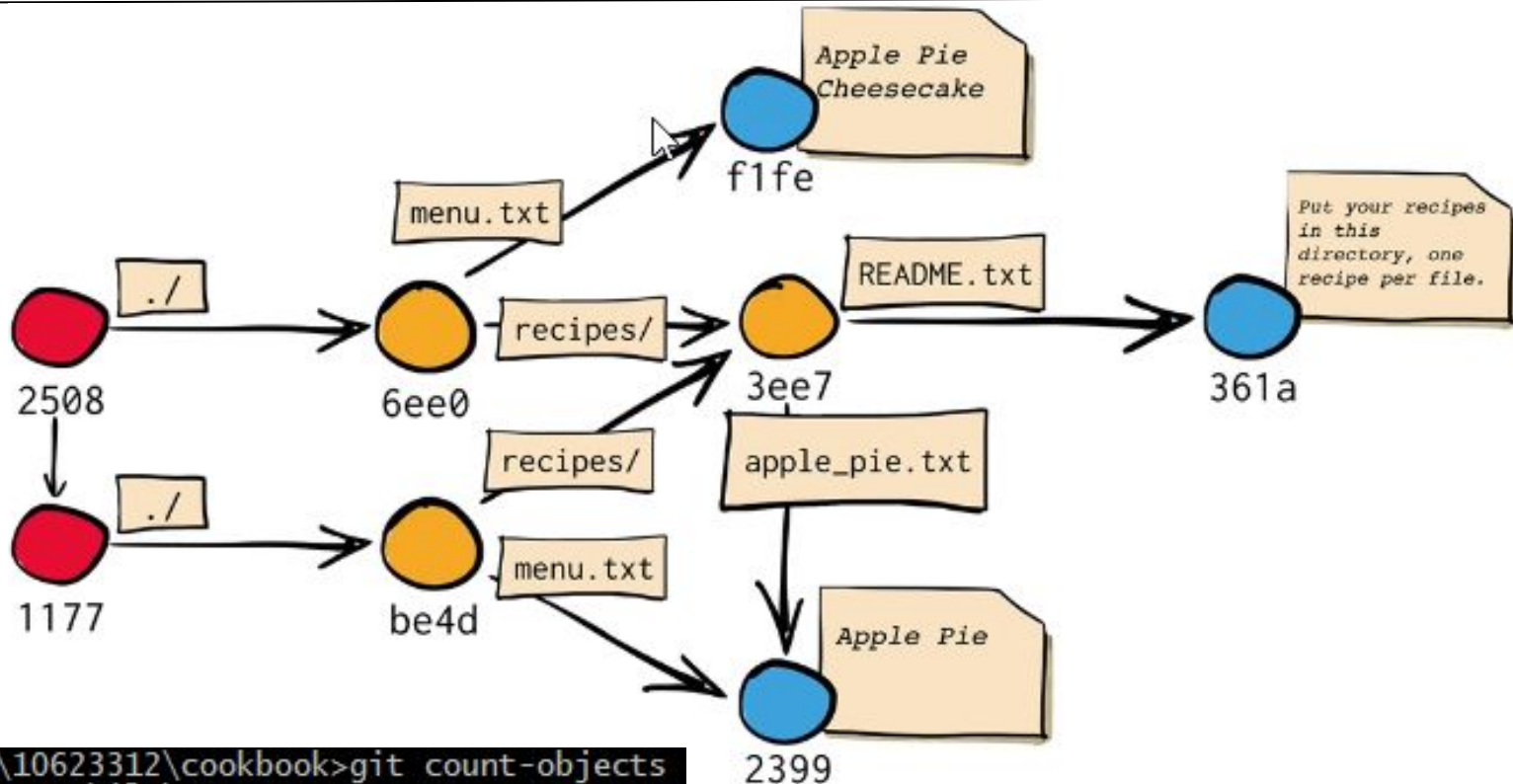
```
C:\Users\10623312\cookbook>git cat-file -p b5a21a1dd9a3ee9161ce4b3b6f3ef1473d662172
tree 8b8ed28c7f86b14c39d1e7befc07030ce692d963
parent 45e4ccfc1b1352043184d233733a62cd231f5cb3
author Daniel McDonald <10623312@uvu.edu> 1547068933 -0700
committer Daniel McDonald <10623312@uvu.edu> 1547068933 -0700

Add cake
```

- commits are linked, most commits have a parent
- The tree in the second commit is a brand new tree



## The second commit



```
C:\Users\10623312\cookbook>git count-objects
8 objects, 0 kilobytes
```



## Changes to big files

---

- Git creates a new hash when file content changes
- What if you have a very large file with a small change?
- Git does do a level of optimizations
  - Sometimes only the difference between files is stored
  - Compress multiple objects in the same physical file
  - The optimizations produce the `info` and `pack` folders
- However, what is most important is that each `commit`, `blob`, or `tree` are hashed and put into the database as separate separate files





# Git tags

---

- A tag is like a label for the current state of the project
- Regular tags
- Annotated tags
  - Come with a message
  - command: `git tag -a mytag -m "I love cheesecake"`
  - Also an object in git's object database

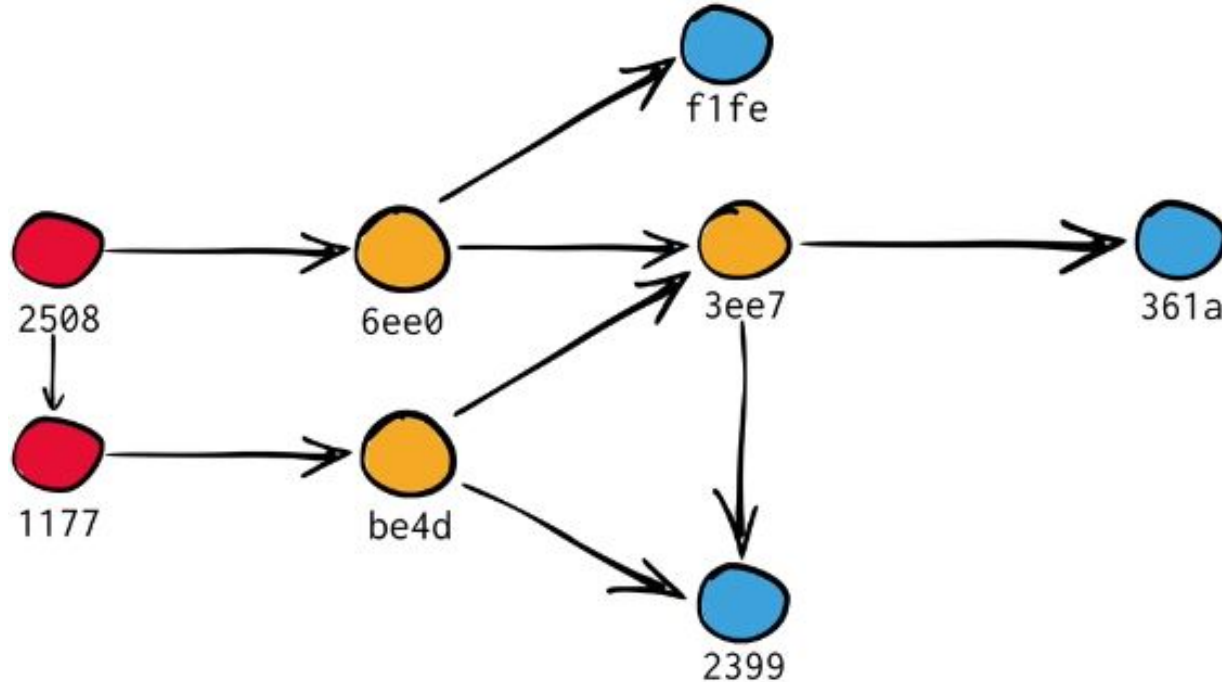
```
C:\Users\10623312\cookbook>git tag -a mytag -m "I love cheesecake"
C:\Users\10623312\cookbook>git tag
mytag
C:\Users\10623312\cookbook>git cat-file -p mytag
object b5a21a1dd9a3ee9161ce4b3b6f3ef1473d662172
type commit
tag mytag
tagger Daniel McDonald <10623312@uvu.edu> 1547072024 -0700
I love cheesecake
```



# Git Objects

---

- Blobs - arbitrary content
- Trees - the equivalent of directories
- Commits - references to blobs, trees, and meta-data
- Annotated Tags



- Git is like a filesystem with commits which give it versioning



- Git is a Revision Control System
  - Branches - Git creates a branch (`master`) when we do our first commit
  - Merges

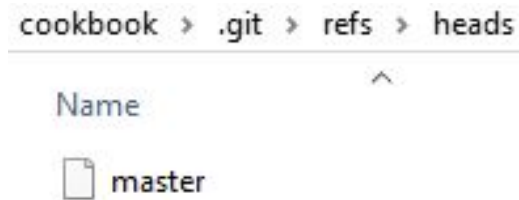
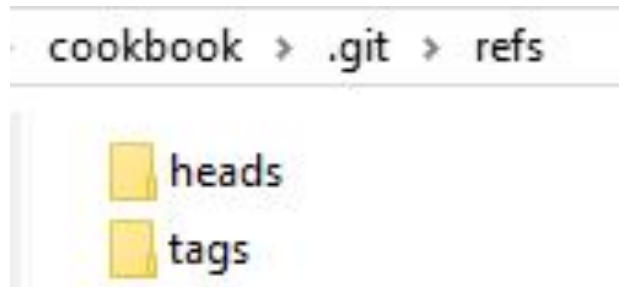
```
C:\Users\10623312\cookbook>git branch
* master
```





## Looking for Branches

- Git puts branches in directory refs/heads



```
C:\Users\10623312\cookbook\.git\refs\heads>type master  
b5a21a1dd9a3ee9161ce4b3b6f3ef1473d662172
```

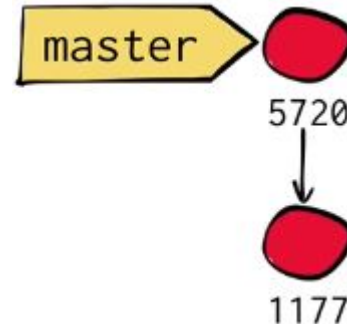
- The contents of master is the SHA1 of the commit



# What is a branch

- We have 2 linked commits in our project
- We have a master branch
- The branch is a simple reference to a commit
  - That is why the directory is called refs
  - You could actually rename the branch
- Create a new branch

```
C:\Users\10623312\cookbook>git branch lisa  
C:\Users\10623312\cookbook>git branch  
lisa  
* master
```





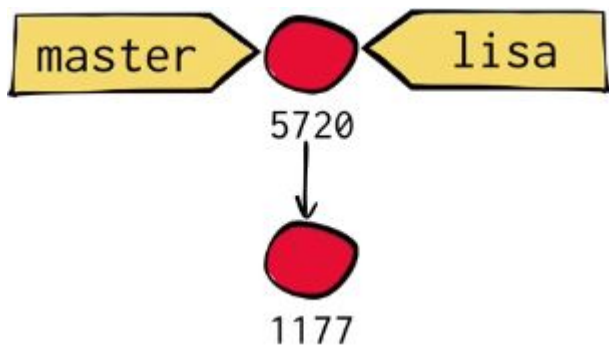
# The New Branch

- The new branch "lisa" has the same SHA1 as master

```
C:\Users\10623312\cookbook\.git\refs\heads>type lisa
b5a21a1dd9a3ee9161ce4b3b6f3ef1473d662172

C:\Users\10623312\cookbook\.git\refs\heads>type master
b5a21a1dd9a3ee9161ce4b3b6f3ef1473d662172
```

- We have 2 branches pointing at the same commit

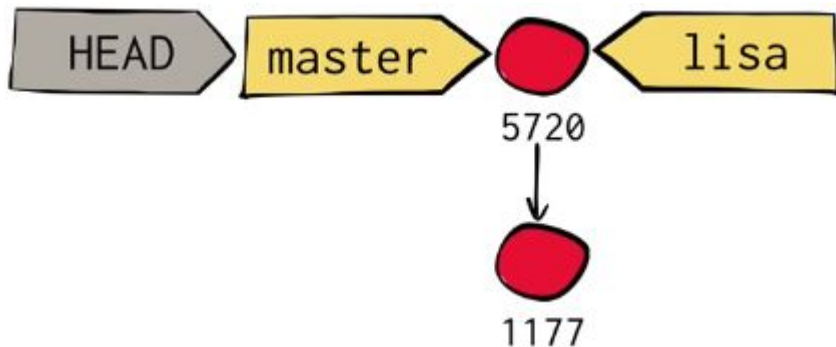




## The current branch

- The branch with an asterisk (\*) is the current branch
- There is only one current branch
- The current branch is recorded by the HEAD file

```
C:\Users\10623312\cookbook\.git>type HEAD  
ref: refs/heads/master
```







# Change some files



- Update the `recipes/apple_pie.txt` file with some ingredients

Apple Pie

pre-made pastry

1/2 cup butter

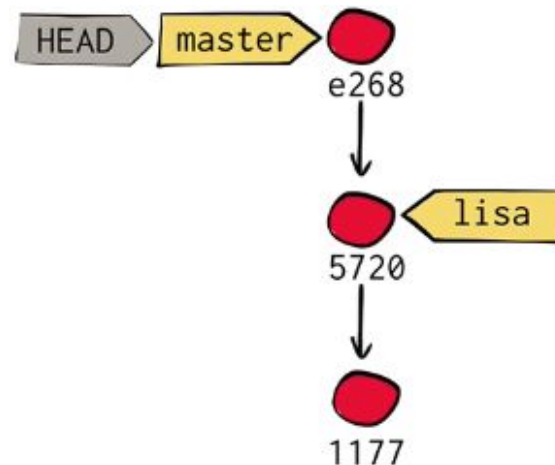
3 tablespoons flour

1 cup sugar

8 Granny Smith apples

```
>git status
>git add recipes/apple_pie.txt
>git commit -m "Add recipe"
>git log
```

- The master file now points to a new commit
- The HEAD did not change
- lisa is still pointing at other commit





## Changing the current branch

---

- command: `git checkout lisa`

- Our working area changed to the content of the commit pointed at by lisa

- Checkout means

move HEAD and update WORKING AREA

```
C:\Users\10623312\cookbook>git checkout lisa
Switched to branch 'lisa'

C:\Users\10623312\cookbook>git branch
* lisa
  master

C:\Users\10623312\cookbook>type .git\HEAD
ref: refs/heads/lisa
```



# Make a Recipe into the lisa version



- Update the `lisa` branch (which is the current branch) file of `recipes/apple_pie.txt`

Apple Pie

pre-made pastry

1/2 cup butter

3 tablespoons flour

1 cup sugar

1 tbsp cinnamon

10 Granny Smith apples

```
>git status
```

```
>git add recipes/apple_pie.txt
```

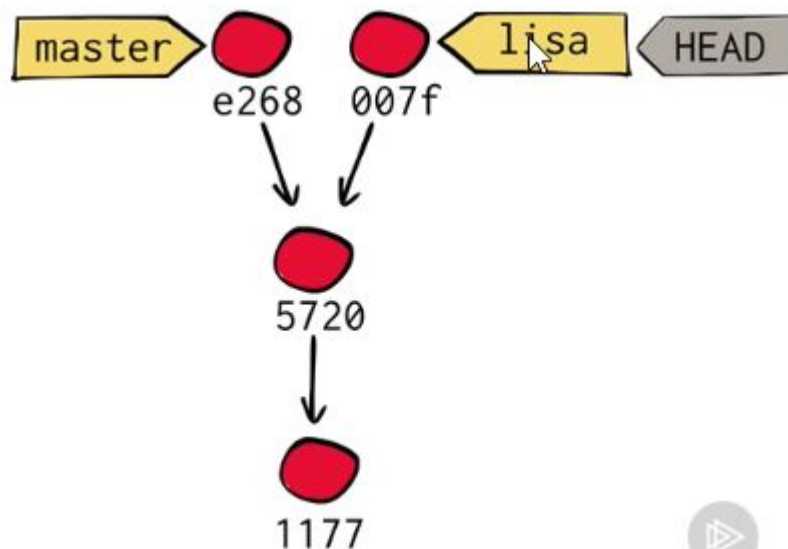
```
>git commit -m "Add Lisa's version of the pie"
```

```
>git log
```



## What changed

- HEAD did NOT change
- Master did NOT change
- lisa did change so that it points to the new commit
- Branches are just references to commits that is all they are





## Move back to Master Branch

---

- Command: `git checkout master`
- The branches did not move, but HEAD did move
- The content has been switched back to our version of the recipes
- Let's say we like Lisa's version of the apple pie better than our own
- Time to merge

```
C:\Users\10623312\cookbook>git checkout master
Switched to branch 'master'

C:\Users\10623312\cookbook>git branch
  lisa
* master
```



## Merging the branches

- Command: `git merge lisa`

- We need to open `apple_pie.txt` and fixed the conflicts

```
C:\Users\10623312\cookbook>git merge lisa
Auto-merging recipes/apple_pie.txt
CONFLICT (content): Merge conflict in recipes/apple_pie.txt
Automatic merge failed; fix conflicts and then commit the result.

C:\Users\10623312\cookbook>git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

        both modified:   recipes/apple_pie.txt

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



## Resolving conflict

- Once conflicts have been resolved, we must `git add`
- `git add apple_pie.txt`
- `git status`
- `git commit`
- The commit tells git conflicts resolved
- Git automatically creates the commit message for us

```
C:\Users\10623312\cookbook>git add recipes\apple_pie.txt
C:\Users\10623312\cookbook>git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
  modified:   recipes/apple_pie.txt

C:\Users\10623312\cookbook>git commit
[master 15b8b62] Merge branch 'lisa'
Committer: Daniel McDonald <10623312@uvu.edu>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author
```



# What is a merge?

---

- A merge is just a commit with 2 parents
- Run `git log` to get the SHA1 of the merge
- Command: `git cat-file -p 15b8b6`
- Note the 2 parents

```
C:\Users\10623312\cookbook>git cat-file -p 15b8b6
tree d975e854a4935db4225e5c5e86f8ba000f5effbc
parent 95bc7c8ef7bbb36ee55f809d80ac713aa74b620b
parent 1c2b51fe9cc287a58adcb34db9f7d1d969150d25
author Daniel McDonald <10623312@uvu.edu> 1547240306 -0700
committer Daniel McDonald <10623312@uvu.edu> 1547240306 -0700

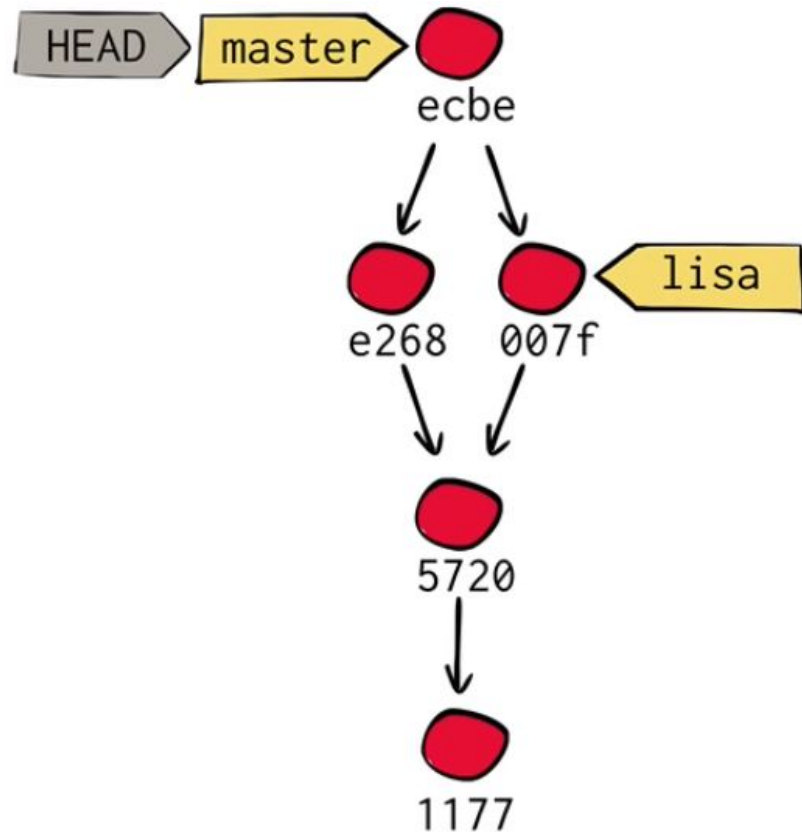
Merge branch 'lisa'
```





# Merge is just a commit with 2 parents

- Git created a new commit with 2 parents to represent the merge
- Git moved master to point to the new commit





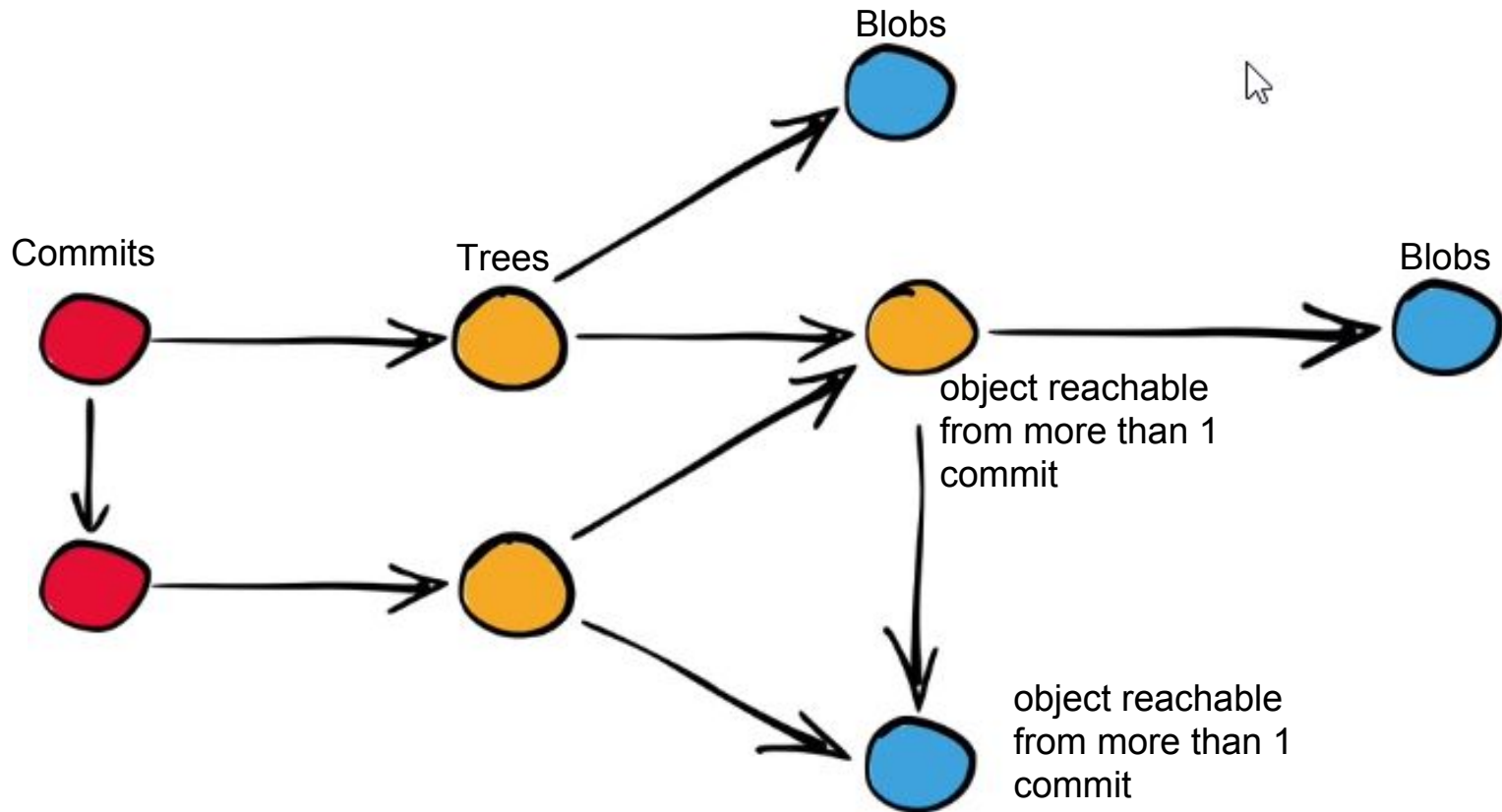
# Adding trees and blobs to manage history

---

- Objects in git db are commits, trees, blobs, and tags
- They are arranged in a graph with references to each other
- References
  - between COMMITS track HISTORY
  - between OTHER OBJECTS track CONTENT



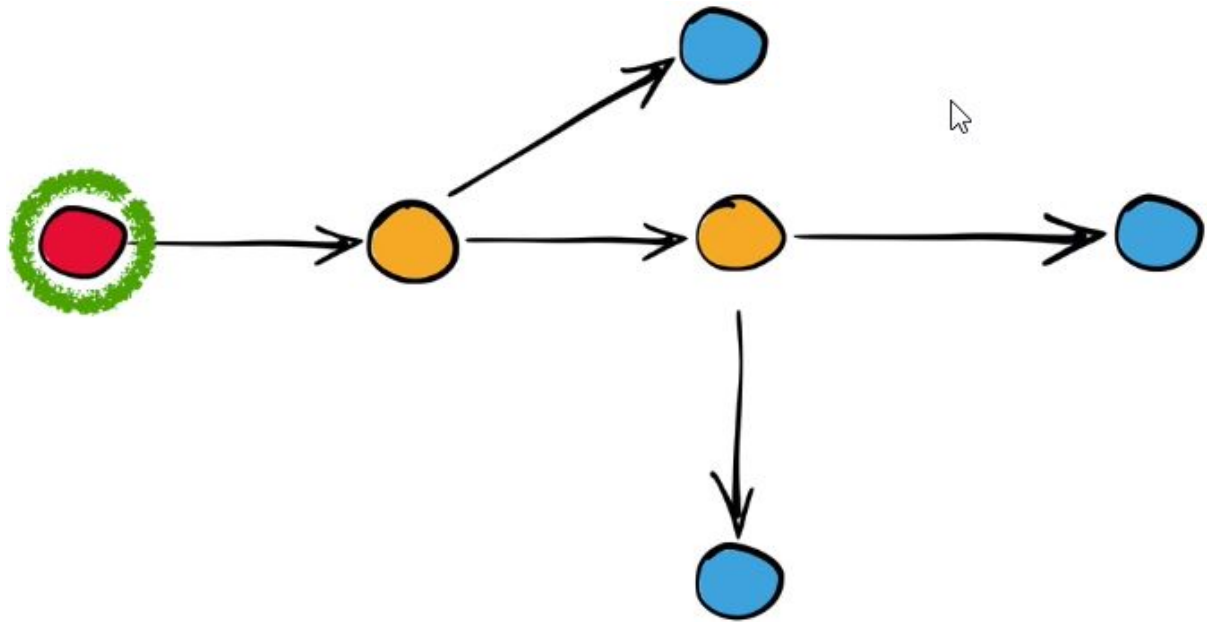
# History and Content





# History and Content

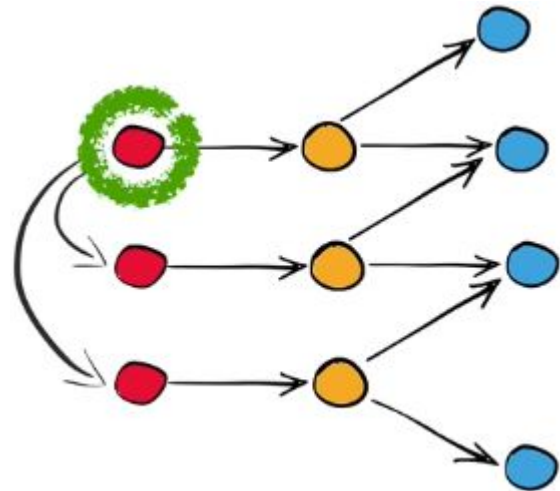
- When you checkout something, it doesn't care about history, just trees and blobs
- Just the tree in the commit and all the objects that can be reached from there
- This info replaces content of working directory





# Merge commits

- Are NOT really more complicated
- They have multiple parents = the definition of a merge
- If you check out, git does NOT really care how many parents
- With a checkout, git just goes in and gets the current tree
- Git does NOT care in which commit the object was created
- Git reuses objects that are already there and creates the objects that are not there





## Git working area

---

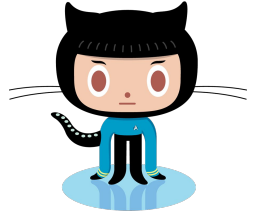
- We should focus on history (how commits connect) and let Git do the right thing with trees and blobs
- Git does not really care too much about our working area
- When you checkout, Git just replaces the object
- Git cares about the objects in the database
- Objects in the database are immutable and persistent
- Working directory files are transient
- Git will give you a warning before overwriting files that have not been committed.



## Special case of a merge

---

- Check out lisa branch: `git checkout lisa`
- We previously merged `lisa` into `master`
- Now we want to merge `master` into `lisa`
- We could create a new commit with the two parents, like other merges and have `lisa` point at the new commit
- However, the conflicts between `master` and `lisa` have already been resolved
- Git just moves `lisa` branch to point at `master` commit
- This is called a Fast Forward



## Called a Fast Forward

- Fast forward is Git bragging about keeping the number of objects in the objects database under control and keeping your project history cleaner

```
C:\Users\10623312\cookbook>git merge master
Updating 1c2b51f..15b8b62
Fast-forward
 recipes/apple_pie.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```





## Detached Head

---

- Command: `git checkout master`
- HEAD is a reference to a `branch` which is a reference to a `commit`
- When you checkout a `branch`, you are changing HEAD
- You can directly checkout a `commit` instead of a `branch`



# Detached HEAD

```
C:\Users\10623312\cookbook>git checkout 15b8b6  
Note: checking out '15b8b6'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 15b8b62 Merge branch 'lisa'
```

```
C:\Users\10623312\cookbook>git branch  
* (HEAD detached at 15b8b62)  
  lisa  
  master
```

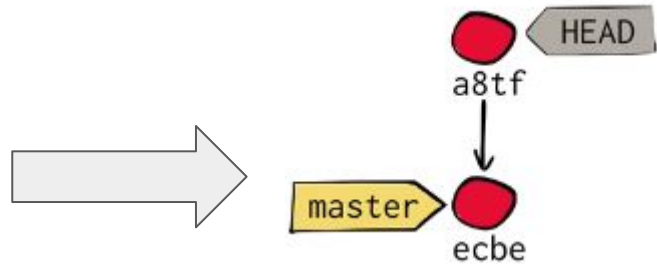




# Update Apple Pie Recipe

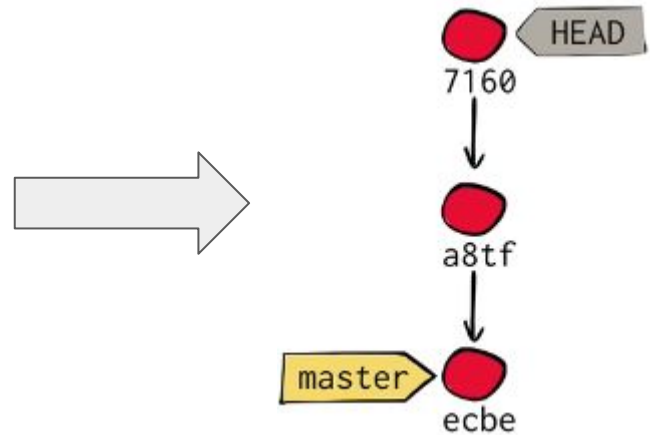
- Change 9 apples to 20 apples

```
>git status  
>git add recipes/apple_pie.txt  
>git commit -m "Add more apples"  
>git log
```



- Now remove sugar ingredient

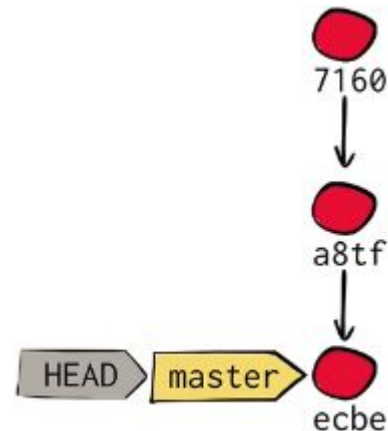
```
>git status  
>git add recipes/apple_pie.txt  
>git commit -m "Remove sugar"  
>git log
```





## Abandon experiment (rollback commits)

- Command: `git checkout master`
- HEAD is back where it belongs
- Commits are still in object database
- The commits are isolated
  - No branch
  - Can only be reached by SHA1
- The object will be garbage collected eventually





# Record experiment

- Navigate to the commit now via the SHA1
- Checkout the commit using the SHA1

```
C:\Users\10623312\cookbook>git checkout 5c01386
Note: checking out '5c01386'.

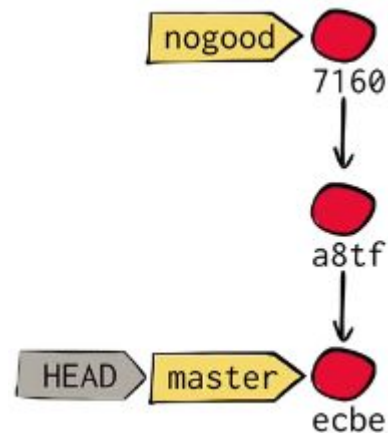
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 5c01386 Remove sugar
```

- Now put a branch on it: `git branch nogood`





## Detached HEAD as a useful tool

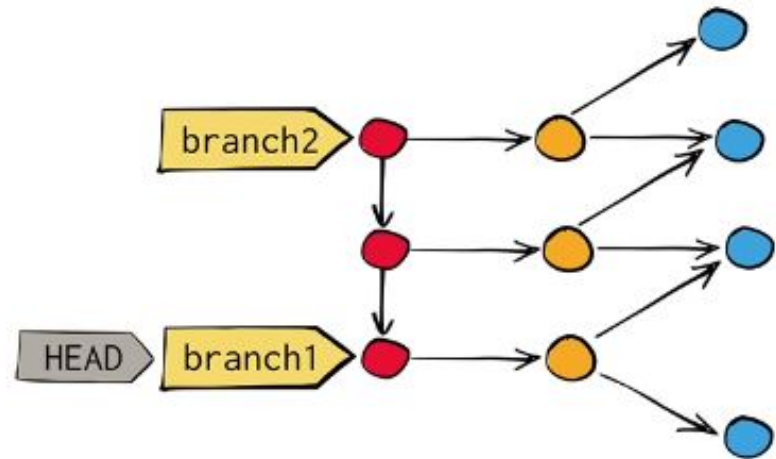
---

- This is a common way to use a detached head
- Do your experiment
- Commit your experiment as much as you wish
- Decide whether to keep the experiment or not
- Just put a branch on what you want to keep



# Git Object Model

- Branch is a reference to a commit
- HEAD is a reference to a branch
- Three Git Rules
  - The current branch tracks new commits
  - When you move to another commit, Git updates your working directory
  - Unreachable objects are garbage collected





# Rebasing

---



- Not a common feature among versioning systems
- Git's signature feature



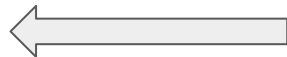
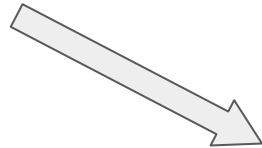
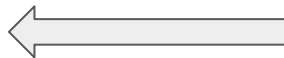
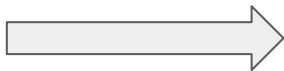


# Make some new commits to master

```
>git checkout master
```

```
>git add apple_pie.txt  
>git commit -m "More sugar"
```

```
>git add apple_pie.txt  
>git commit -m "More Cinnamon"
```



```
#file apple_pie.txt  
Apple Pie
```

```
pre-made pastry  
1/2 cup butter  
3 tablespoons flour  
2 cup sugar  
1 tbsp cinnamon  
9 Granny Smith apples
```

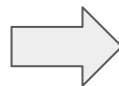
```
#file apple_pie.txt  
Apple Pie
```

```
pre-made pastry  
1/2 cup butter  
3 tablespoons flour  
2 cup sugar  
2 tbsp cinnamon  
9 Granny Smith apples
```



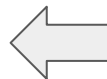
# Create a new branch "spaghetti"

```
>git branch spaghetti  
>git checkout spaghetti
```



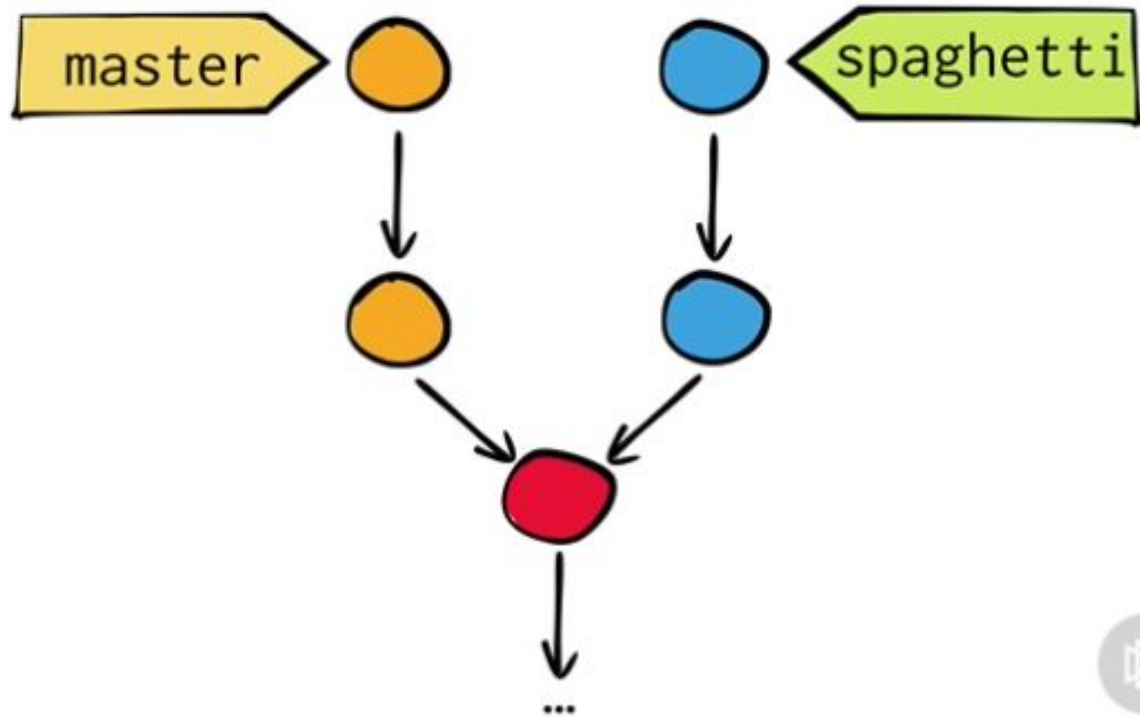
```
#file menu.txt  
Spaghetti alla Carbonara  
Apple Pie  
Cheesecake
```

```
>git add menu.txt  
>git add recipes/spaghetti_alla_carbonara.txt  
>git commit -m "New Spaghetti Recipe"  
>git log
```

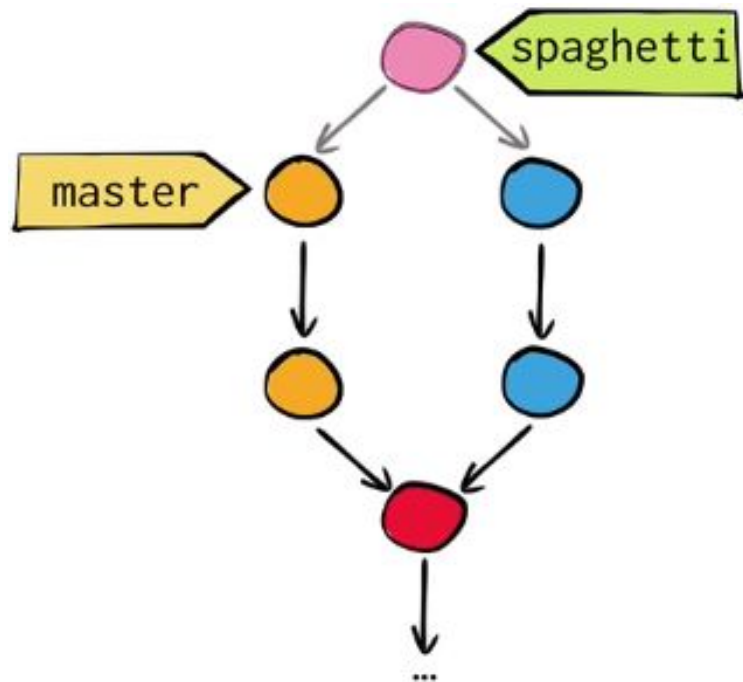


```
#file  
recipes/spaghetti_alla_carbonara.txt  
Spaghetti alla Carbonara  
  
1 point spaghetti  
2 tablespoons oil  
4 ounces diced bacon  
1 onion  
3 eggs  
1 cup parmesan cheese  
1 handfull parsley  
salt and pepper
```

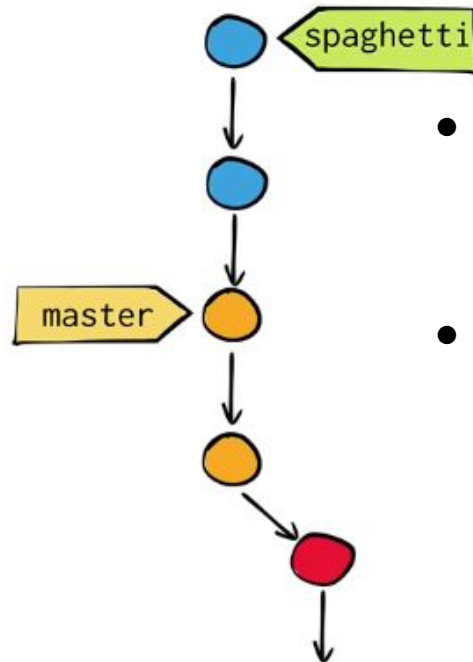
# After changes



## Merge?



## Rebase spaghetti branch



- git looks for the first commit in spaghetti that is also a commit in master
- git then detaches the branch and moves it



## Rebase commands

---

- Currently checkout branch = spaghetti
- There is just one commit that gets applied on top of master (or 2 commits if you did it that way)

```
C:\Users\10623312\cookbook>git branch
  lisa
  master
  nogood
* spaghetti

C:\Users\10623312\cookbook>git rebase master
First, rewinding head to replay your work on top of it...
Applying: New Spaghetti Recipe
```

- We might have to resolve conflicts if there are any



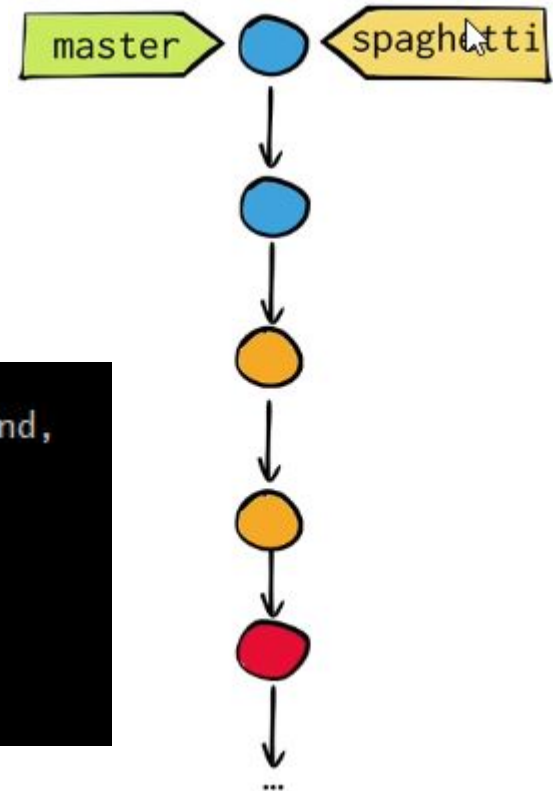
## Rebase Master as well

- Checkout master
- We have all the commits related to spaghetti and master all in the same history

```
C:\Users\10623312\cookbook>checkout master
'checkout' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\10623312\cookbook>git checkout master
Switched to branch 'master'

C:\Users\10623312\cookbook>git rebase spaghetti
First, rewinding head to replay your work on top of it...
Fast-forwarded master to spaghetti.
```





- mutable  
copied (with new  
DT moved  
es new commits
- 
- Diagram illustrating the 'NO' case for a branch merge. A vertical sequence of nodes (blue, blue, orange, orange) leads to a red merge node. A green arrow labeled 'a\_branch' points to the second blue node. The word 'NO' is written in the center.



# Why Rebase over Merge

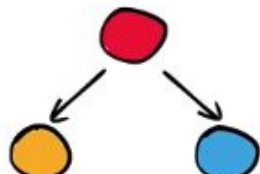
- Merge preserves history
- Merge commits include resolving conflicts
- Merge history never lies
- A project with a lot of rebasing looks clean
- Rebases refactor history
- Rebase history can be deceiving
- When in doubt, just merge

## Rebase



yellow  
commits did  
not take  
place  
before blue

## Merge







## Tags - a part of versioning

---

- One of four git database objects
- Two types of tags
  - Annotated tags (date, author, description)
  - Non-annotated tags or lightweight tags (simple label)

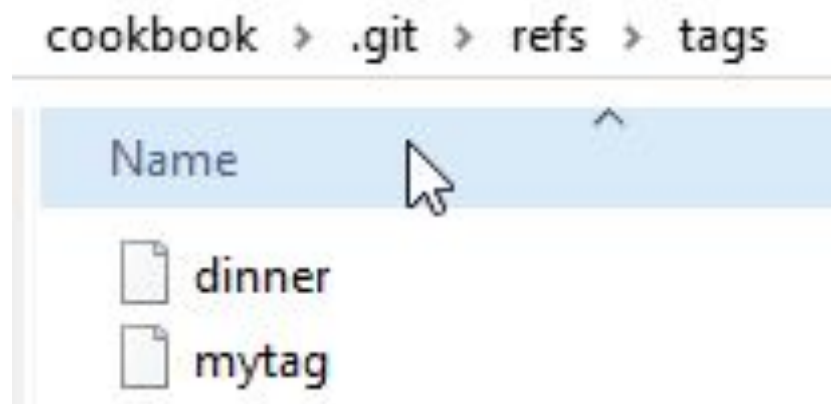
```
C:\Users\10623312\cookbook>git tag dinner
```

```
C:\Users\10623312\cookbook>git tag  
dinner  
mytag
```



## Tags are saved in git database

- Tags look like branches, but unlike branches, they don't move
- Tags are just a SHA1
- Tags reference commits
- Tags stick to the same commit forever



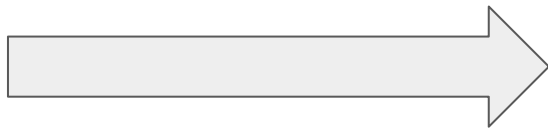
```
C:\Users\10623312\cookbook\.git\refs\tags>type dinner
6c36c2173216a3ed9c5fd7c3944ba4626083789d
```



## Recap

---

- From Stupid Content Tracker to Revision Control System
- Branches, merges, rebases, tags to handle versioning



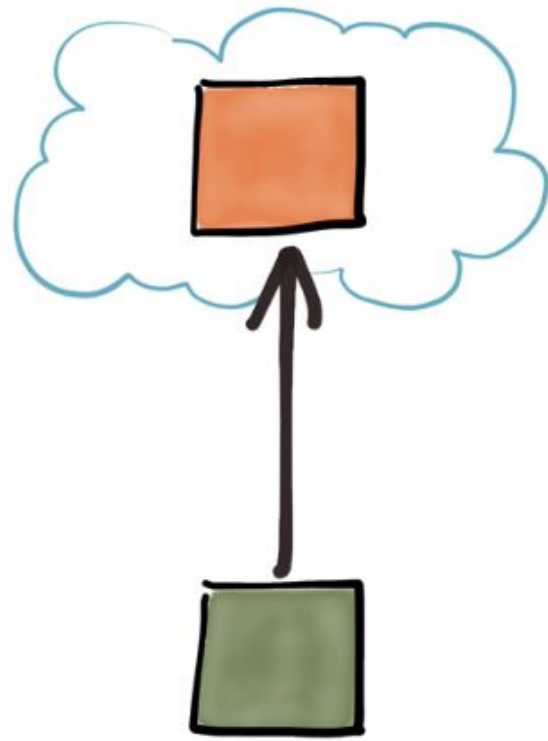


# Distributed Revision Control System

- We now have a repo in the cloud:  
GitHub
- Our local repo is the green square
- Command: `git clone`



Multiple Repos





# Cloning Time

- Time to clone a project

```
C:\Users\10623312>mkdir demo
C:\Users\10623312>cd demo
C:\Users\10623312\demo>git clone https://github.com/nusco/cookbook.git
Cloning into 'cookbook'...
remote: Enumerating objects: 47, done.
remote: Total 47 (delta 0), reused 0 (delta 0), pack-reused 47
Unpacking objects: 100% (47/47), done.
```

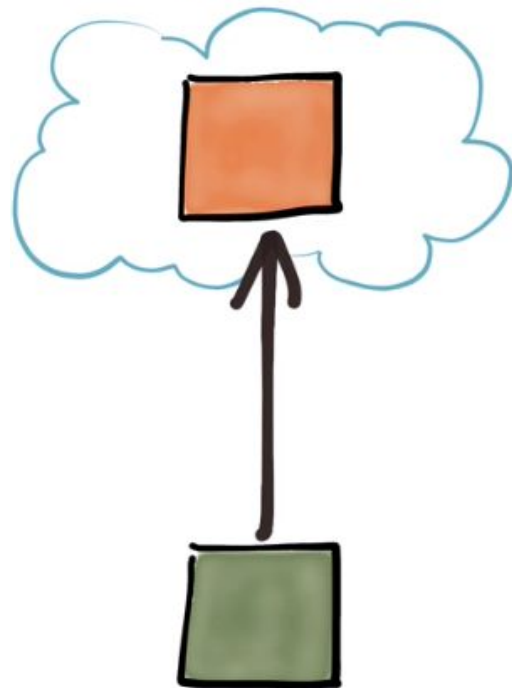
- Git created an empty directory for cookbook
- Git copies the .git directory from GitHub to local directory
- In later GitHub versions, only copies objects from the master branch
- After the git database was downloaded, git checks out the master branch and rebuilds it in the working area
- We now have a copy of the project and its history on local computer



## Multiple Clones

- We now have two clones of the project that are equally good
- Git is not like Subversion that needs a centralized server and everyone must talk to that server

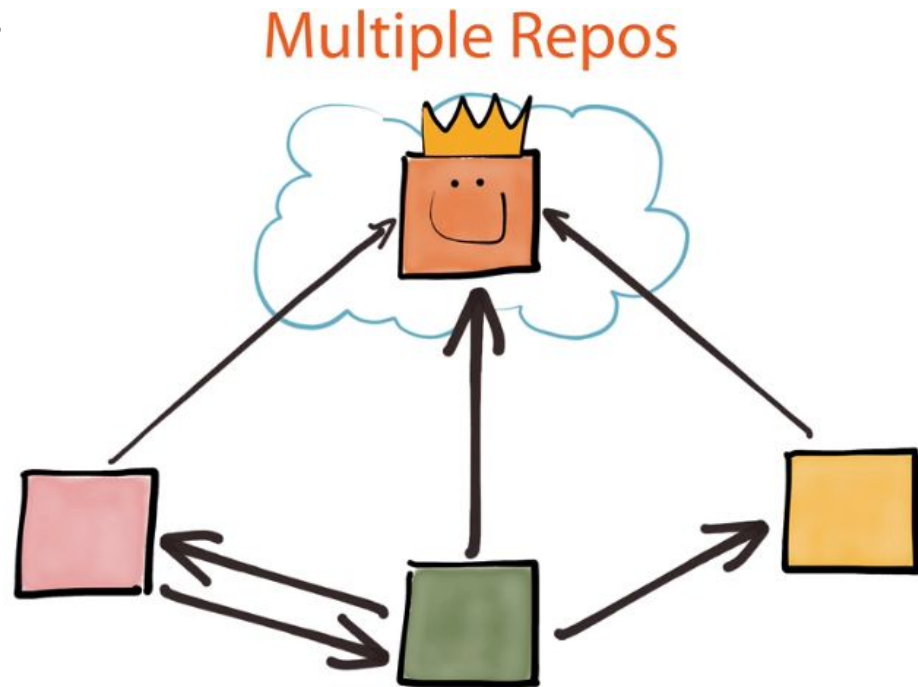
## Multiple Repos





# Clone Wars

- You can have as many clones as you want synchronizing with each other
- One clone can still be the most important one
- It is a good idea to have a well-known reference copy that everyone synchronizes with (social issue)





## .git/configure

- Useful for Git to remember the repo it cloned
- Git added a few configuration lines when we issued the `git clone` command
- `vim .git/configure`
- Other copies of the same repository are called a remote
- There is a default remote called `origin`

```
[remote "origin"]  
  url = https://github.com/nusco/cookbook.git  
  fetch = +refs/heads/*:refs/remotes/origin/*
```







## Local Git remembers

---

- Which other repos (remote) we want to synchronize with
- To synchronize, git needs to know the current state of origin
  - which branches are on the remote
  - which commits the branches are pointing at
- Git does store that information as well

```
$ git branch --all
* master
remotes/origin/HEAD -> origin/master
remotes/origin/lisa
remotes/origin/master
remotes/origin/nogood
remotes/origin/spaghetti
```



## Local Git remembers

---

- Git tracks remote branches exactly like it tracks local branches
- Git writes those branches as references in the refs folder
- The origin folder contains references to branches, tags, and the current HEAD pointer of origin
- Git automatically updates this information when we connect to a remote

```
$ ls -l refs/remotes/ 1
total 0
drwxr-xr-x 1 10623312 1049089 0 Jan 15 13:06 origin/
```



## Local Git

---

- Some of the references are sometimes included in the `packed-refs` file as an optimization
- All branches, local or remote are still *references to a commit*



# Branches are references to commits

- Command: `git show-ref master` shows references to all commits with master in their name

```
$ git branch --all
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/lisa
  remotes/origin/master
  remotes/origin/nogood
  remotes/origin/spaghetti

10623312@601MCDONALD MINGW64 ~/demo/cookbook (master)
$ git show-ref master
704182f5e2925fbdc03f9874d35ce696c21e9a3d refs/heads/master
704182f5e2925fbdc03f9874d35ce696c21e9a3d refs/remotes/origin/master
```

- Note the two master references are still pointed at the same commit, lisa branch is different



## Branch references a commit

---

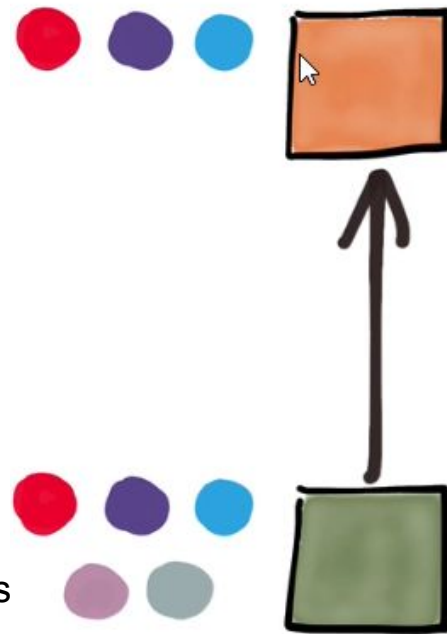
- Local branch in git is a reference to a commit
- Remote branch is the same thing
- Whenever you synchronize with the remote, Git updates remote branches



# Synchronizing Repos

- SHA1 are unique in the universe
- Synchronization is about *getting the same objects on all the clones*
- All objects are immutable and have a unique SHA1
- Git has to also keep the branches synchronized on the clones
  - THIS can be tricky

## Synchronizing Repos



added blobs/trees



## Adding a branch

---

- Command: `vi recipes/apple_pie.txt`
- Change the amount of lemon juice in the recipe

```
>vi recipes/apple_pie.txt
>git add recipes/
>git status
>git commit -m "Add lemon juice to the apple pie"
```

- A few new objects in the database
  - A new blob to represent the file changed
  - A new tree to represent the updated project root folder
  - A new commit



## Local and remote branches differ

- The local and remote master branches are now different

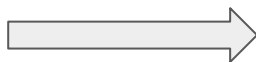
```
$ git show-ref master
943f67fefaa685ff114acf07c92f9ef872b77db2 refs/heads/master
704182f5e2925fbd03f9874d35ce696c21e9a3d refs/remotes/origin/master
```

- Lets send the new objects and updated branch to the origin

```
>git push
```

- The remote branch is updated

```
cookbook> git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 469 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To https://github.com/nusco/cookbook.git
   5d4a817..704182f  master -> master
cookbook> git show-ref master
704182f5e2925fbd03f9874d35ce696c21e9a3d refs/heads/master
704182f5e2925fbd03f9874d35ce696c21e9a3d refs/remotes/origin/master
```

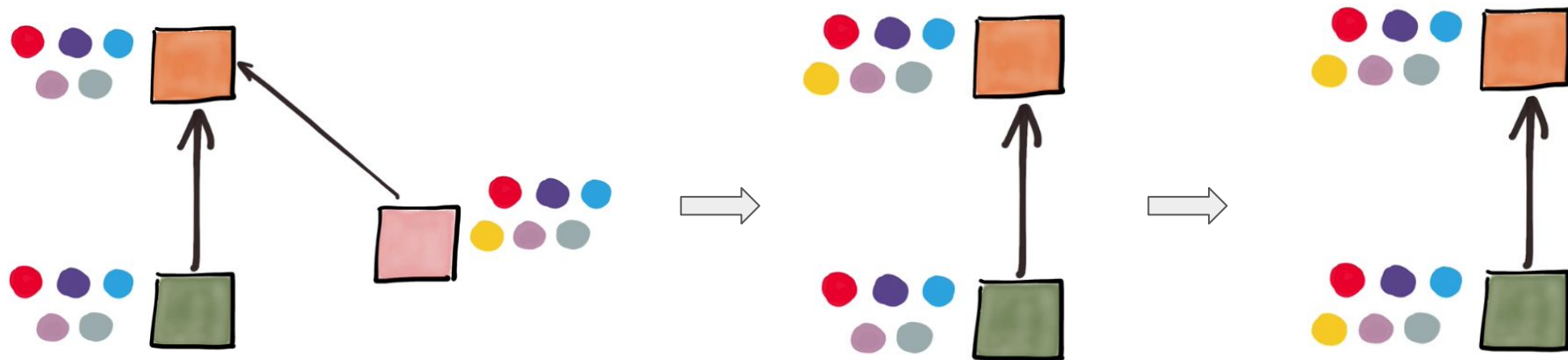






# Synchronizing Repos

- We can't just write changes to the remote, we have to read changes from the remote as well

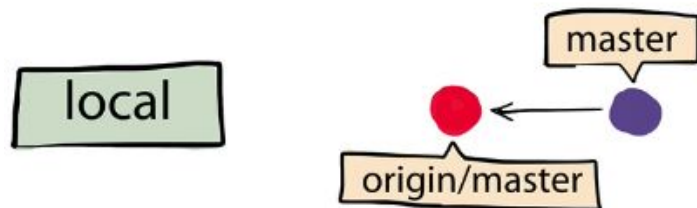


# Synchronizing with a Remote

somebody else pushed a commit



added a commit

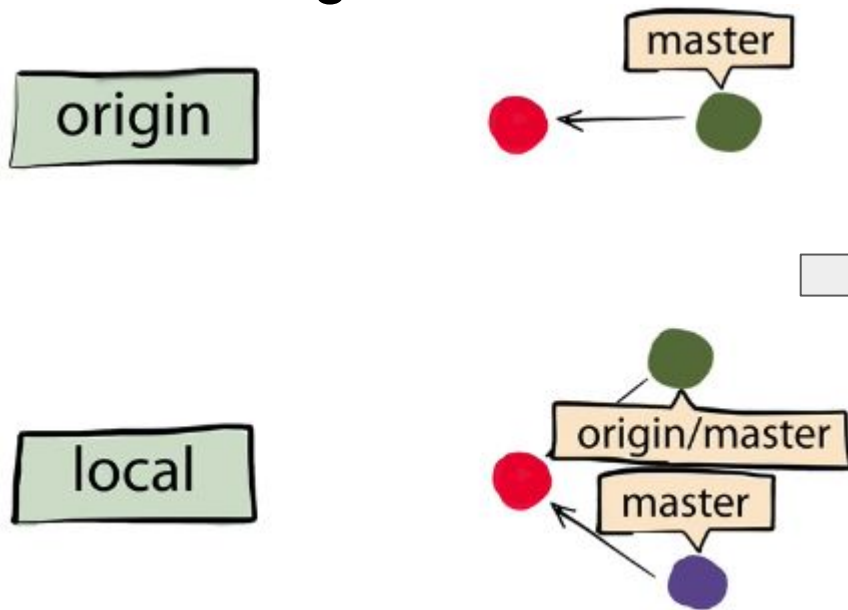


- We pulled from the remote repo
- We made a commit locally, but someone else made a commit remotely
- We need to fix the conflict on our own machine before we push
- `git push -f` (not recommended)
- First we need to fetch the data
  - `git fetch`

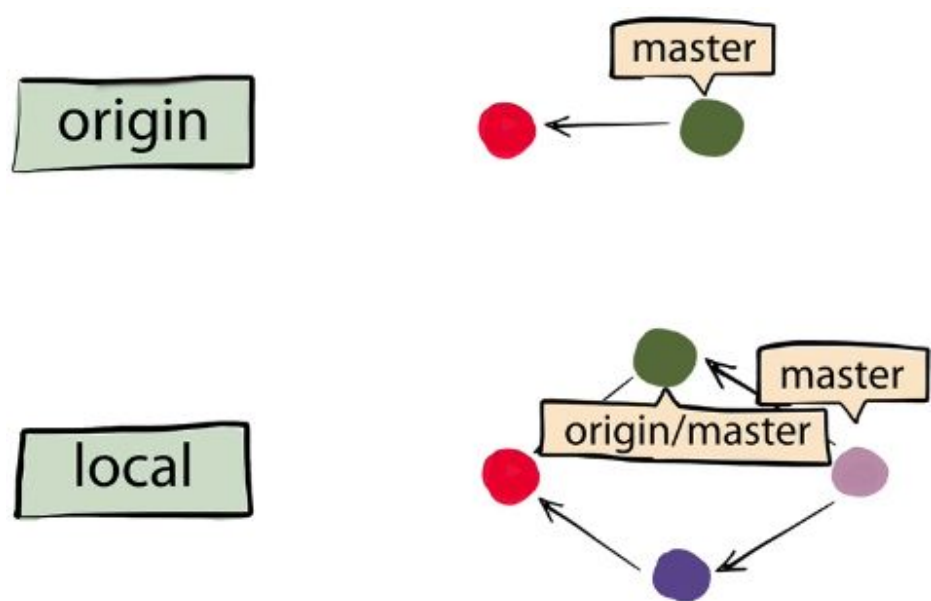


## Better solution: git fetch and git merge

git fetch



git merge



- merges never rewrites history, it only adds commits

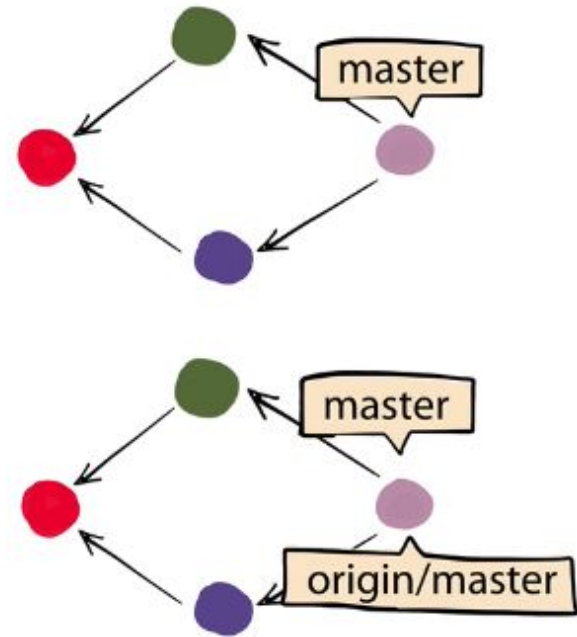


# FETCH + MERGE = PULL

- You "fetch" and "merge", then you "push"
- `git fetch`, followed by `merge` = `git pull`

origin

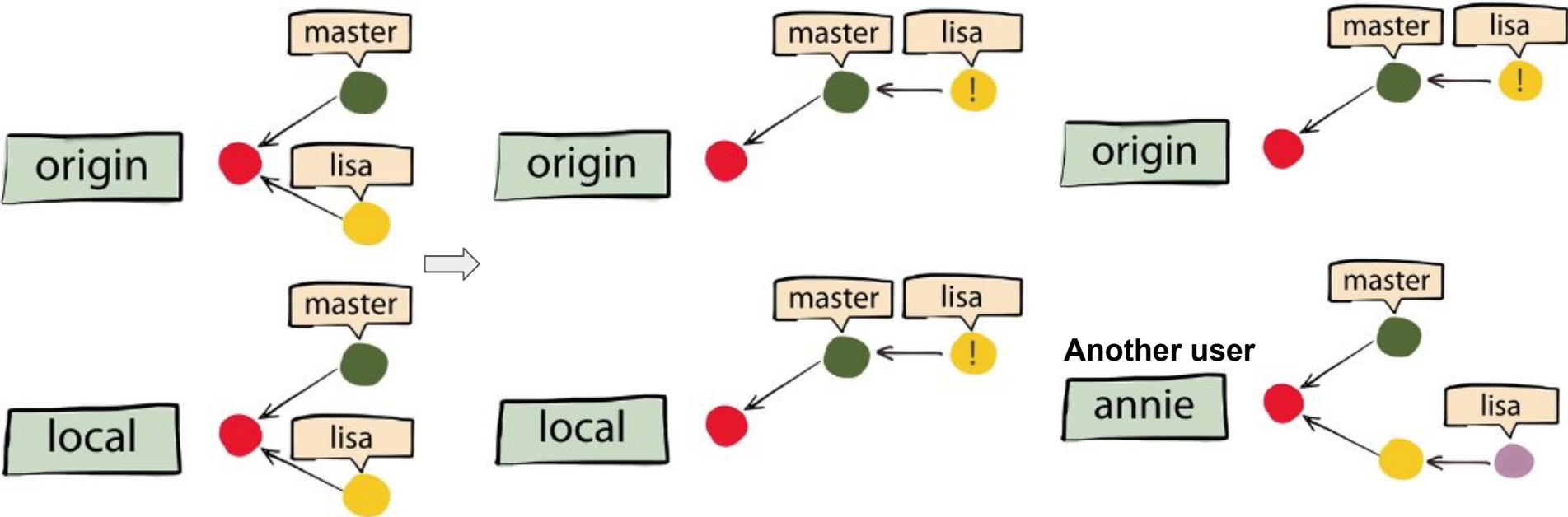
local





# Pushing and Pulling with Rebasing

- We decide to roll changes from master to lisa using rebase





## Bottom line with rebase

---

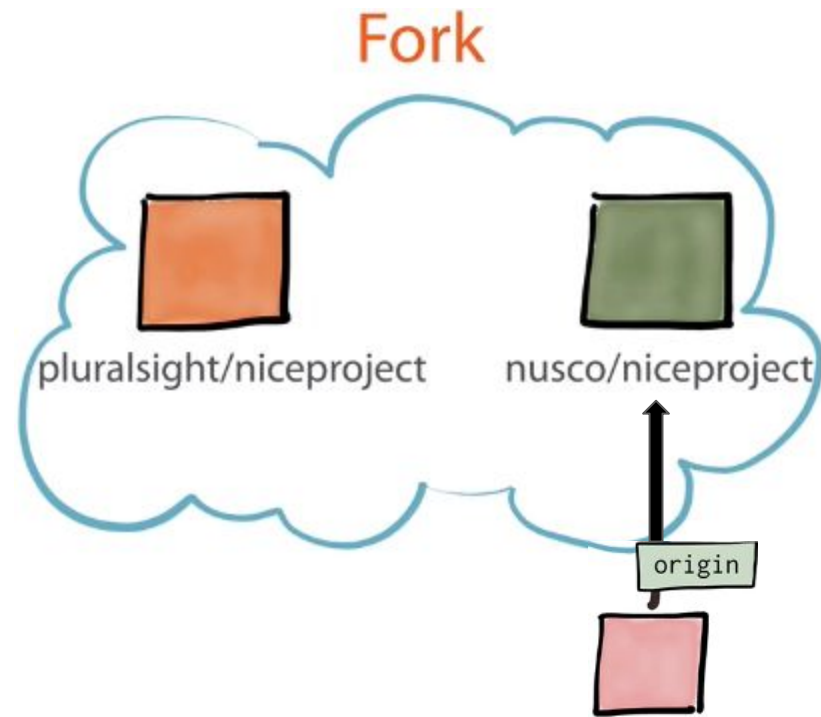
- Never rebase stuff that has been shared with other repository
  - Never rebase shared commits
- It is okay to rebase non-shared commits



# GitHub Features



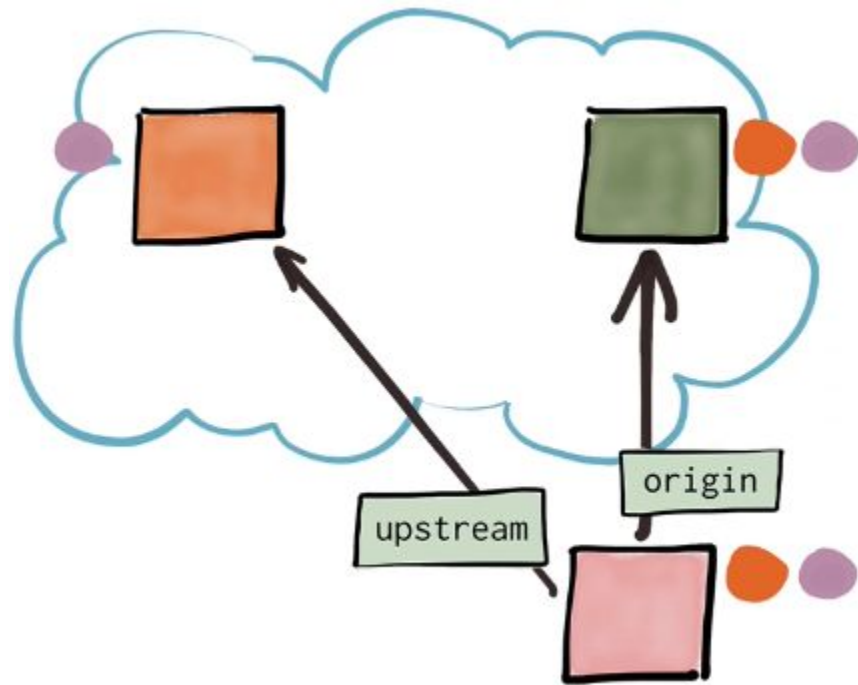
- A fork is like a clone, but it is a remote clone
- We can clone the new cloud project on our local machine so that we can push to it
- No connection from our project to the project we forked





# GitHub Forking

- If we want to track changes to the original project, we explicitly add one (upstream)
- Locally committed changes can be pushed
- Changes on upstream can be pulled to our local and then pushed to origin
- *We cannot push to upstream*

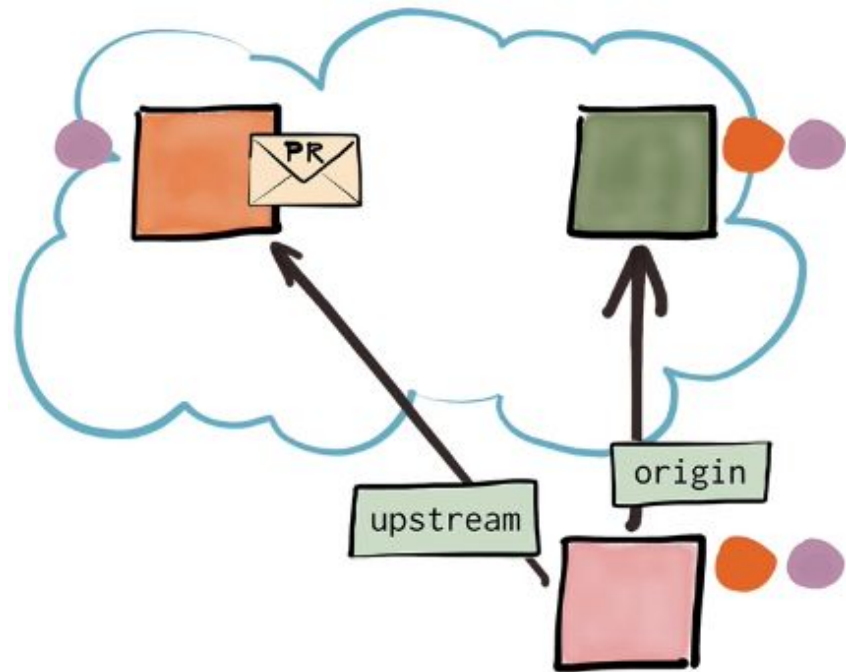






# GitHub Features: Pull Requests

- Not a git feature
- Not even a version control feature
- A social network feature





## Wrap up

---

- A Persistent Map
- A Stupid Content Tracker of changes to content and trees
- A Revision Control System: branches, merges, rebases
- A Distributed Revision Control System: pulling, pushing, forking





# Some Useful Commands

---



#Useful documentation

<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

#to see the last commit on each branch

```
git branch -v
```

#To show branches you have or have not merged

```
git branch --merged
```

```
git branch --nomerged
```

#Branches you have merged in the master branch can be deleted

```
git branch -d
```

```
git branch -D    #This forces deletion
```