Git Source Control
Developing with Git and GitHub

Instructor: Todd Wright
toddw@jtconsult.com

Schedule:

9:00 - 5:00

10:00 AM Morning Break

11:45-1:00 Lunch Break

2:30 PM Afternoon Break

Git Source Control

Upon completion of this course, you will be able to:

- Discuss Git concepts and usage
- Understand differences between Git and other common version control systems
- Create Git repositories
- Recognize tools and technologies used in development using Git

Git Source Control

- Session 1
  - Git Overview and history
  - *Git Repositories*
- Session 2
  - Git versions
  - *Git hashcodes*
  - Core commands

Git Source Control

- Session 3
  - Remote Git repositories
  - Managing Git Histories
  - *Git project management*
- Session 4
  - Cherry picking
  - Rebasing a repository

Git Source Control

Git Source Control

Student Introductions

- Name
- What you work on
- Reason for attending

Facilities and Logistics

- Training Manual
- Start, Stop Time
- Breaks
- Questions and Answers
- Classroom Etiquette


Module 1: Git Version Control
Overview

Git Source Control

Overview

- Define Version Control
- Distributed vs Centralized
- Difficulties in Centralized
- Difficulties in Distributed

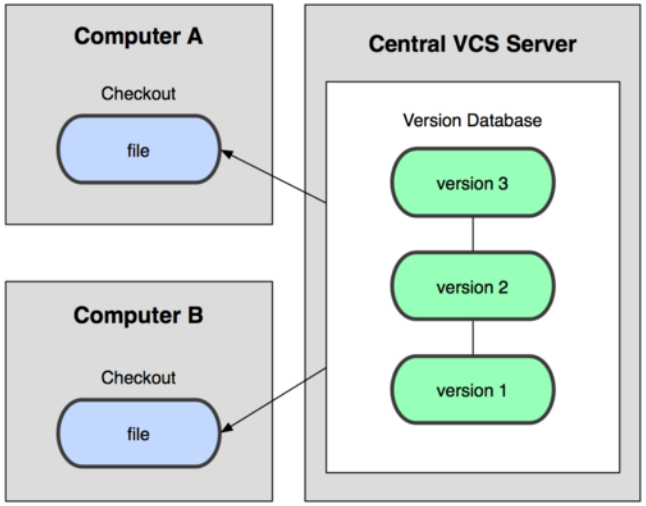
Distributed Version Control

- Authoritative server by convention only
- Every working checkout is a repository
- Get version control even when detached
- Backups are trivial

- Other distributed systems include
  - Mercurial
  - BitKeeper
  - Darcs
  - Bazaar

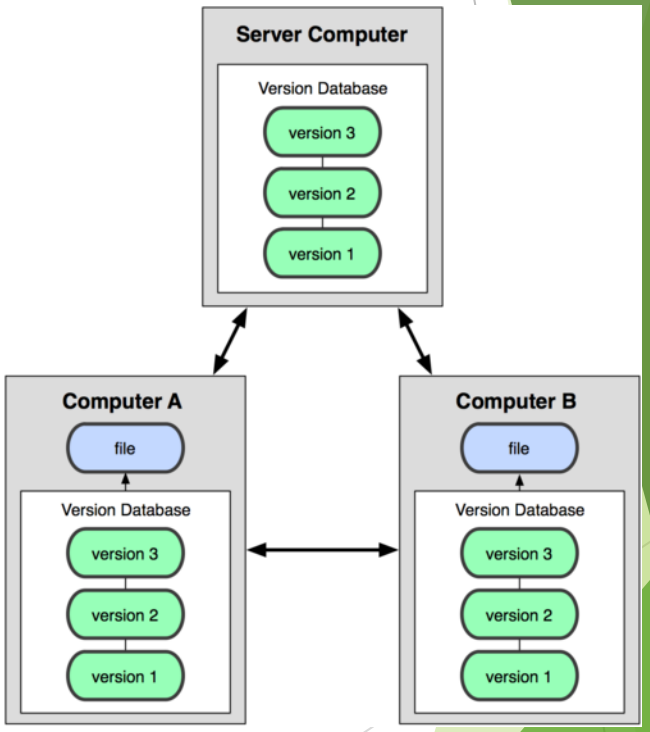
Git uses a distributed model

Centralized Model



(CVS, Subversion, Perforce)

Distributed Model



(Git, Mercurial)

Result: Many operations are local

Git Advantages

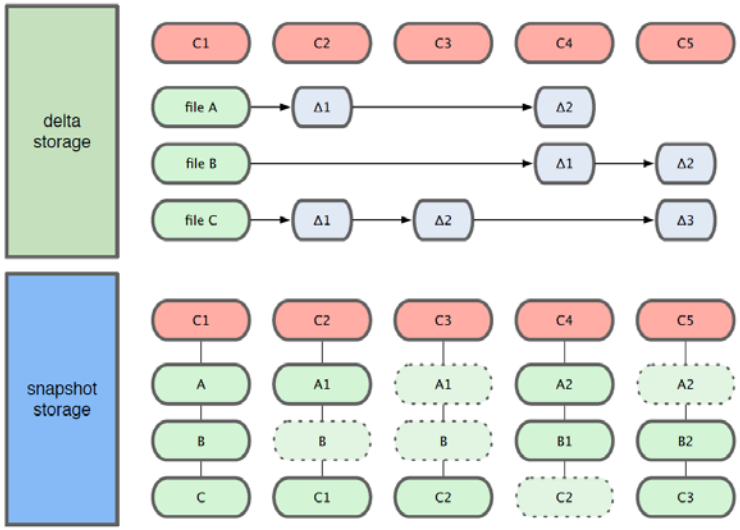
- Resilience
  - No one repository has more data than any other
- Speed
  - Very fast operations compared to other VCS (I'm looking at you CVS and Subversion)
- Space
  - Compression can be done across repository not just per file
  - Minimizes local size as well as push/pull data transfers
- Simplicity
  - Object model is very simple
- Large user base with robust tools

Some GIT Disadvantages

- Definite learning curve, especially for those used to centralized systems
  - Can sometimes seem overwhelming to learn
    - Conceptual difference
    - Huge amount of commands

Getting Started

- Users have their own copies of the repository
  - The full repository could get overwhelming large so the differences are maintained
- Git uses "snapshot storage"



Git uses checksums

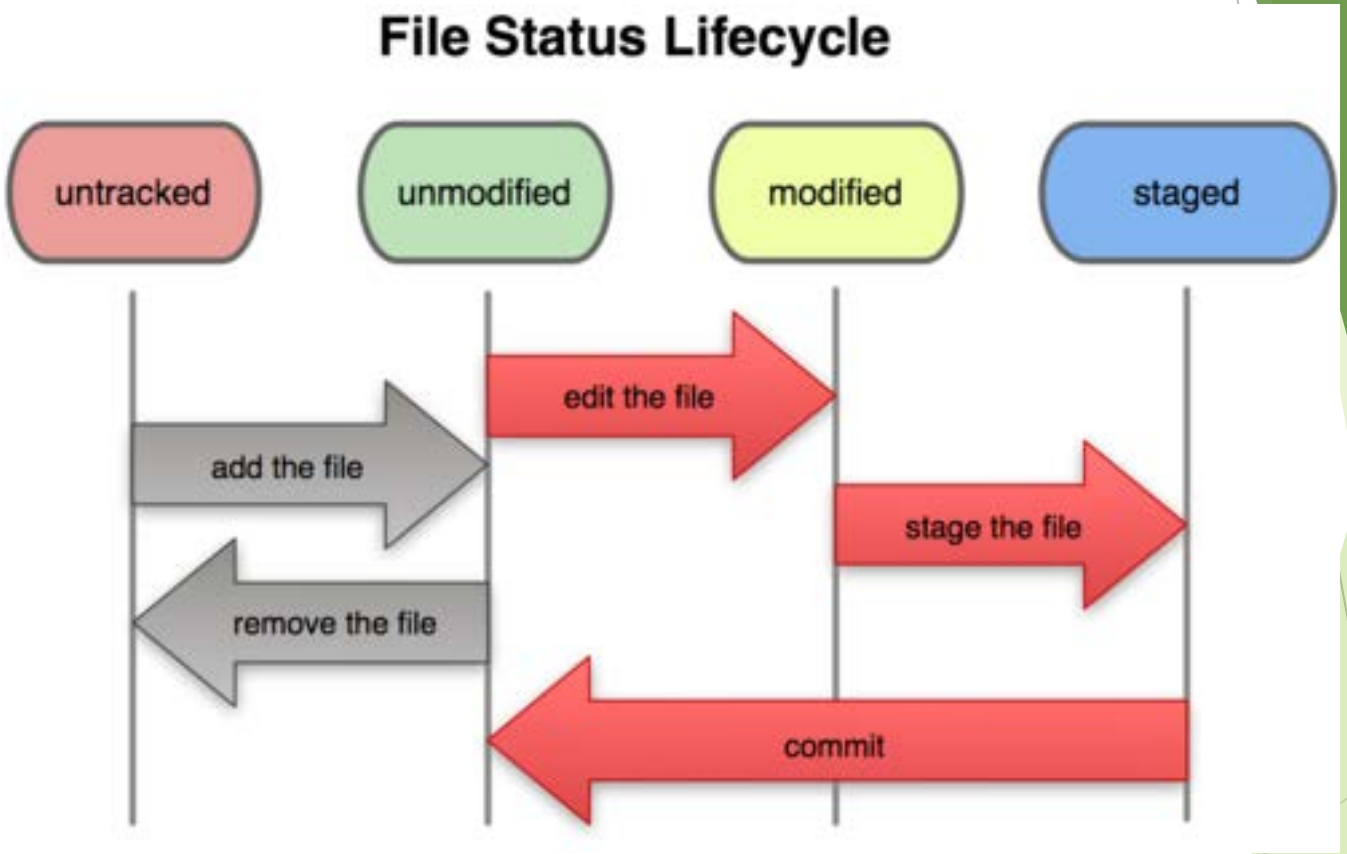
- In Subversion each modification to the **central** repo incremented the version # of the overall repo.
- How will this numbering scheme work **when each user has their own copy of the repo**, and commits changes to their local copy of the repo before pushing to the central server?????
- Git generates a unique SHA-1 hash – 40 character string of hex digits, for every commit. Refer to commits by this ID rather than a version number. Often we only see the first 7 characters:

  1677b2d Edited first line of readme

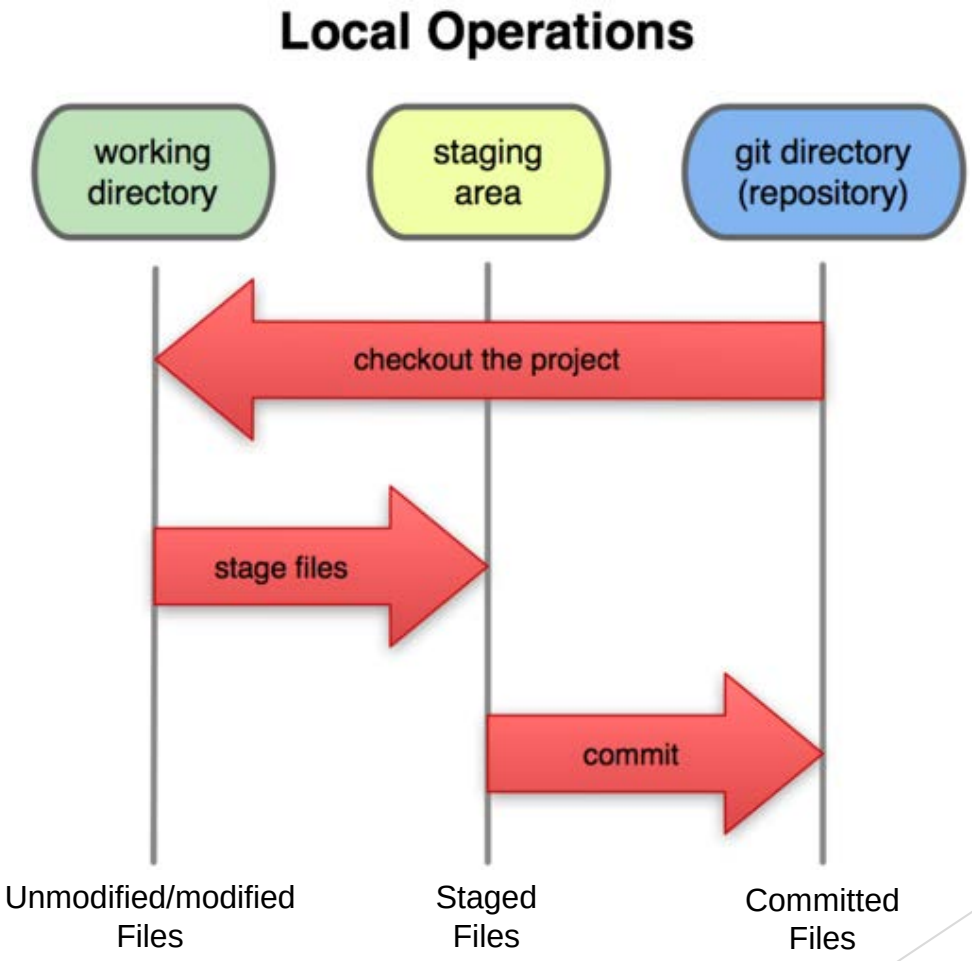
  258efa7 Added line to readme

  0e52da7 Initial commit

Git file lifecycle



A Local Git project has three areas



Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

Basic Git commands

command	description
`git clone` ***`url [dir]`***	copy a git repository so you can add to it
`git add` ***`files`***	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` ***`[command]`***	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

Get ready to use Git!

- Set the name and email for Git to use when you commit:

```
$ git config --global user.name “Bugs Bunny”
$ git config --global user.email bugs@gmail.com
```

- You can call **`git config --list`** to verify these are set.
- These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the **`--global`** flag.
- You can also set the editor that is used for writing commit messages:
  `$ git config --global core.editor emacs` (it is vim by default)

Create a local copy of a repo

Two common scenarios: (only do one of these)

a) To **clone an already existing repo** to your current directory:

```
$ git clone <url> [local dir name]
```

This will create a directory named *local dir name*, containing a working copy of the files from the repo, and a **`.git`** directory (used to hold the staging area and your actual repo)

Create a local copy of a repo

Two common scenarios: (only do one of these)

b) To **create a Git repo** in your current directory:

```
$ git init
```

This will create a .git directory in your current directory.

Then you can commit files in that directory into the repo:

```
$ git add file1.java
$ git commit –m “initial project version”
```

Git Source Control

Exercise 1

- *Refer to Exercise 1* ***in the back of the book***

Module 2: Using Git

Using Git

- A basic workflow
  - Get access to a Repository
  - Make changes to files (or add files) in the repository
  - Push the Git changes to the stage of Git
  - Review changes that will be committed to the repository
  - Commit the stage area to the Git repository

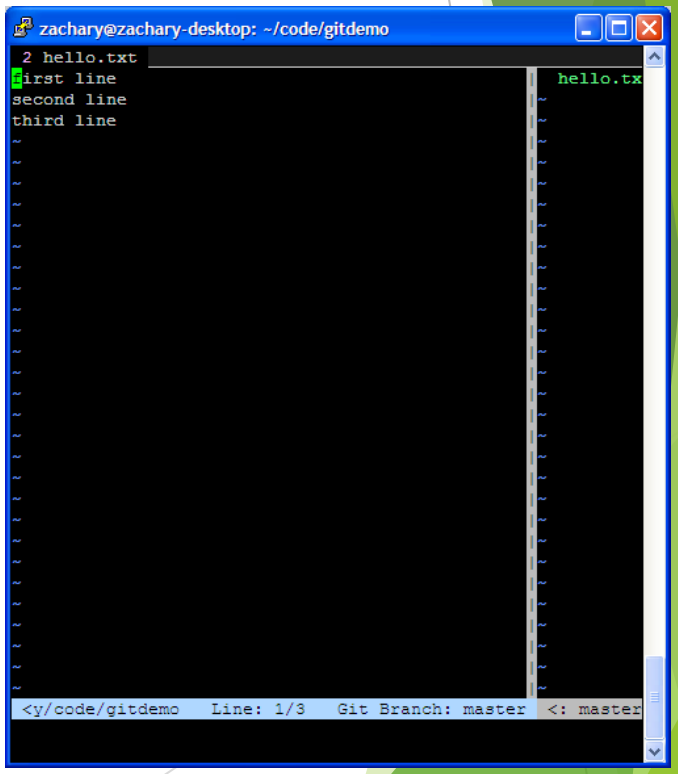
Git access to a repository

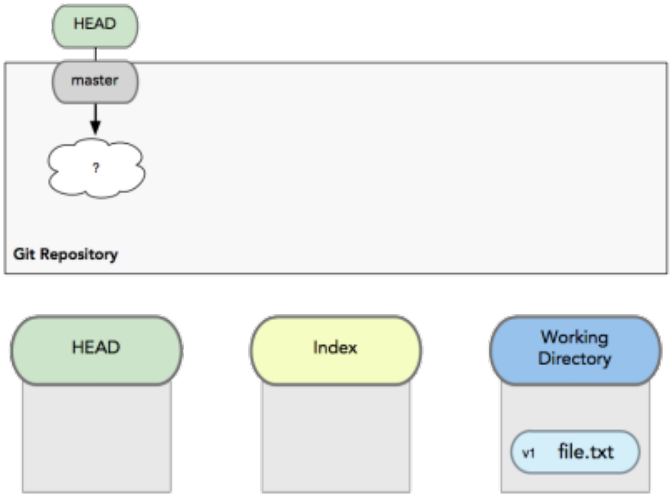
- Init a repository `git init`

```
~/code/gitdemo$ git init
Initialized empty Git repository in /home/student/code/gitdemo/.git/


~/code/gitdemo$ ls -l .git/
total 32
drwxr-xr-x 2 student student 4096 2011-08-28 14:51 branches
-rw-r--r-- 1 student student   92 2011-08-28 14:51 config
-rw-r--r-- 1 student student   73 2011-08-28 14:51 description
-rw-r--r-- 1 student student   23 2011-08-28 14:51 HEAD
drwxr-xr-x 2 student student 4096 2011-08-28 14:51 hooks
drwxr-xr-x 2 student student 4096 2011-08-28 14:51 info
drwxr-xr-x 4 student student 4096 2011-08-28 14:51 objects
drwxr-xr-x 4 student student 4096 2011-08-28 14:51 refs
```

Manipulating Files

- A basic workflow
  - Edit files
  - Stage the changes to the index
  - Review your changes
  - Commit the changes
- Use your favorite editor
- Make changes to your files

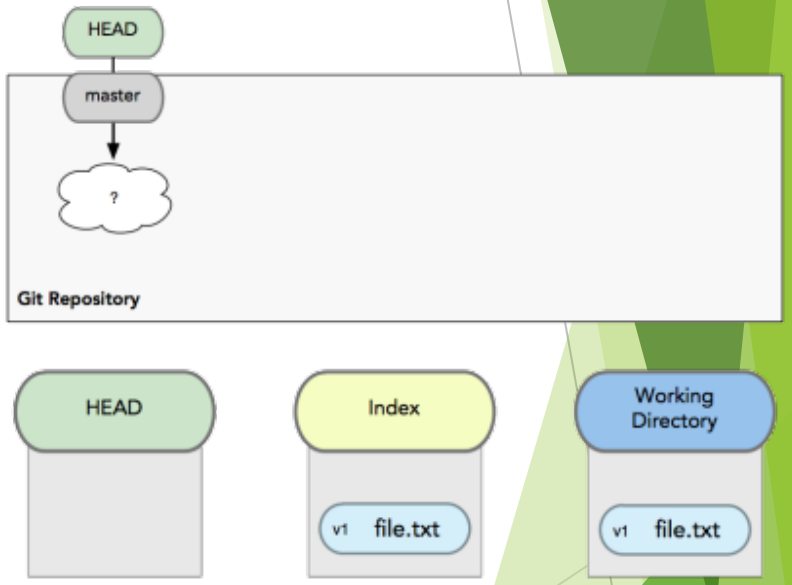




Adding files to your Repository

- A basic workflow
  - Edit files
  - Stage the changes to the index
  - Review your changes
  - Commit the changes

- `git add filename`

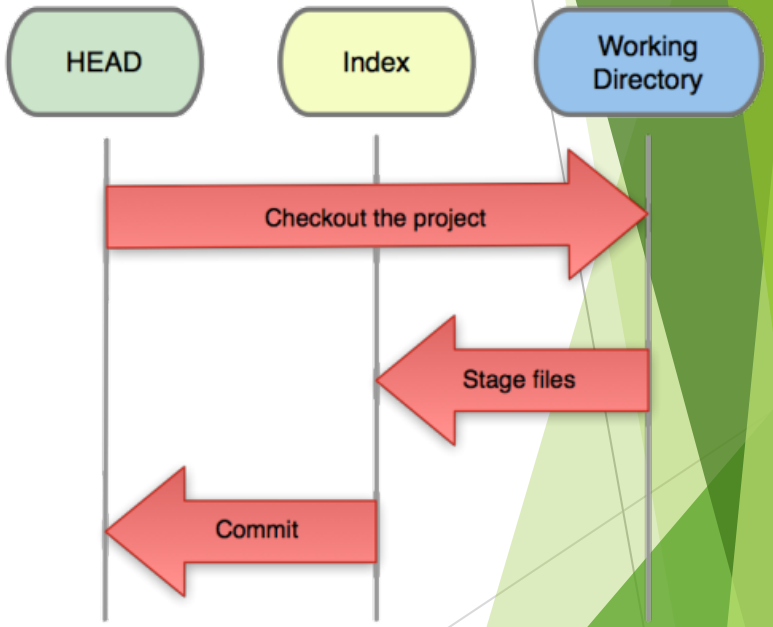



```
~code/gitdemo$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git Commit Flow

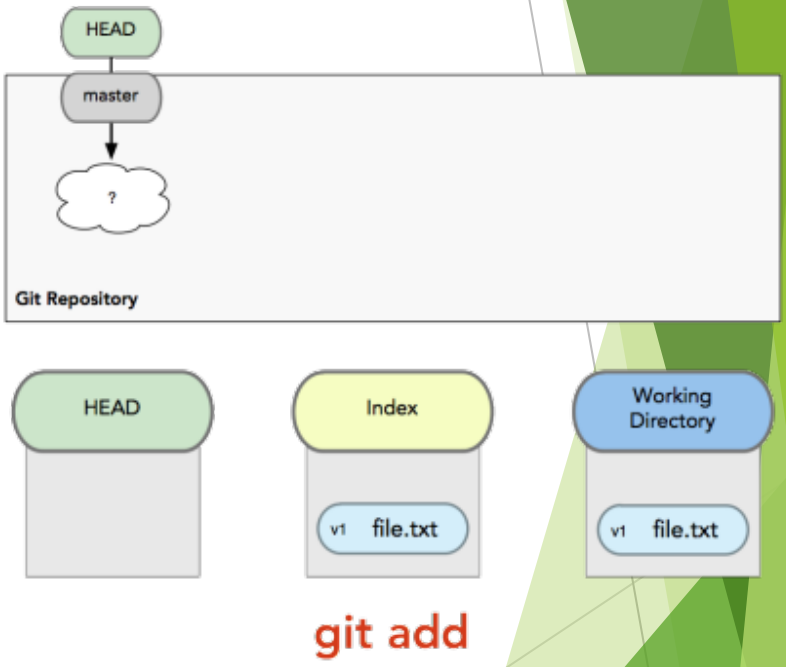
- Changes to the repository go through a series of steps

1. Files are pulled from the HEAD of the repository
    - HEAD is the current active version of the project
2. Changes are made to the working directory
3. Changed files are staged to the index
4. The index is committed to the new updated HEAD of the repository



Verifying what will be committed

- A basic workflow
  - Edit files
  - Stage the changes
  - Review your changes
  - Commit the changes
- `git status`

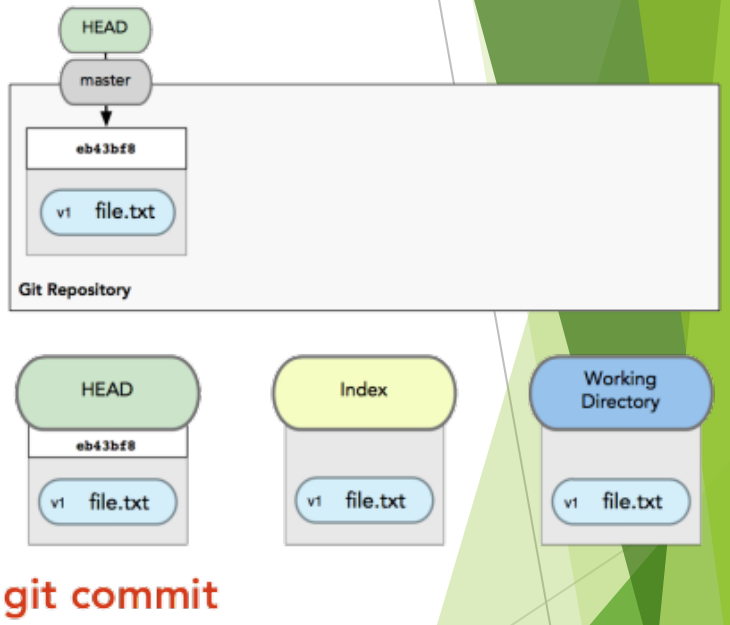



```
~/code/gitdemo$ git add hello.txt
~/code/gitdemo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   hello.txt
#
```

Committing to the repo

- A basic workflow
  - Edit files
  - Stage the changes
  - Review your changes
  - Commit the changes

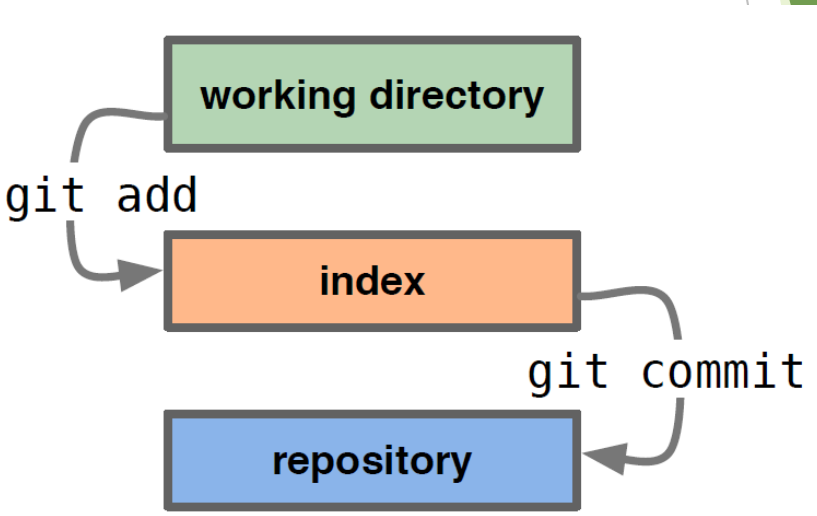
- `git commit`
- `git commit -m 'My Message'`




```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hello.txt
#
```

Core Git process

- A basic workflow
  - Edit files
  - Stage the changes
  - Review your changes
  - Commit the changes



Git Source Control

Exercise 2

Refer to Exercise 2 in the back of the book

Module 3: Git Branching and Merging

Git Source Control

Overview

- Commands to view changes
- Branching the flow
- Merging multiple flows

Comparing versions of files

- View changes
- `git diff`
  - Show the difference between working directory and staged
- `git diff --cached`
  - Show the difference between staged and the HEAD

- View history
- `git log`

```
~/code/gitdemo$ git log
commit efb3aeae66029474e28273536a8f52969d705d04
Author: Student <student@gmail.com>
Date:   Sun Aug 28 15:02:08 2011 +0800

    Add second line

commit 453914143eae3fc5a57b9504343e2595365a7357
Author: Student <student@gmail.com>
Date:   Sun Aug 28 14:59:13 2011 +0800

    Initial commit
```

Controlling changed files

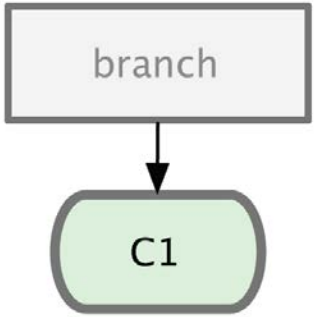
- Revert changes (Get back to a previous version)
  - `git checkout commit_hash`

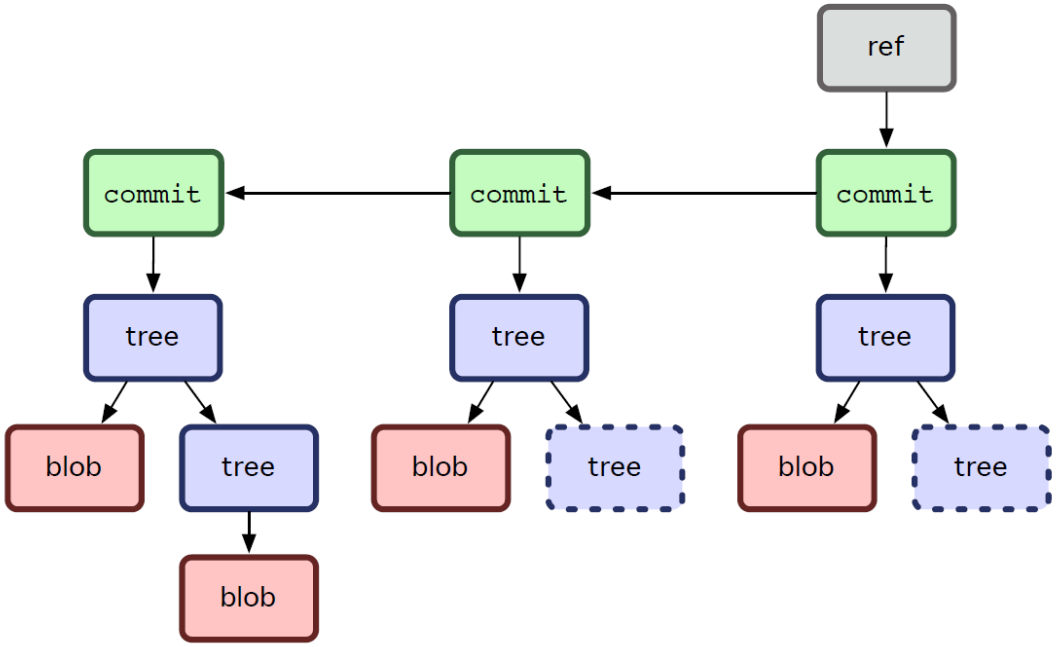
```
~/code/gitdemo$ git log
commit efb3aeae66029474e28273536a8f52969d705d04
Author: Student<student@gmail.com>
Date:   Sun Aug 28 15:02:08 2011 +0800

    Add second line

commit 453914143eae3fc5a57b9504343e2595365a7357
Author: Student<student@gmail.com>
Date:   Sun Aug 28 14:59:13 2011 +0800

    Initial commit
~/code/gitdemo$ git checkout 4539
Note: checking out '4539'.

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 4539141... Initial commit
```

Branching

- Git sees commit this way…
- Branch annotates which commit we are working on

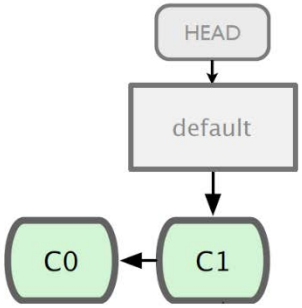
lightweight, movable pointers to a commit





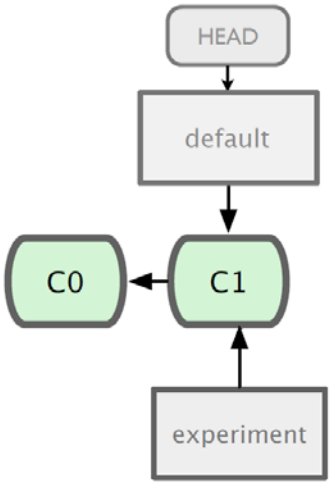
Branching

- A repository is a sequence of commits
  - C0 and C1
  - The HEAD is a pointer to the current active version of the active branch



Branching

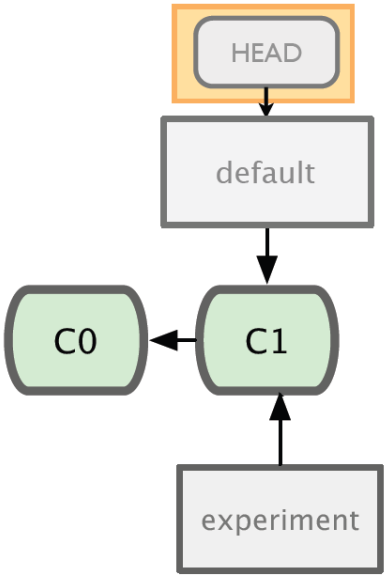
- Alternate branches of commits are created with the branch command:



git branch experiment

Viewing Branches

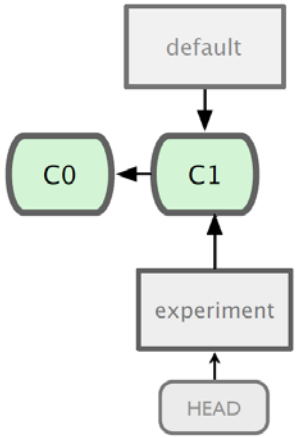
- At anytime you can view your branches:




```
$ git branch
* default
  experiment
```

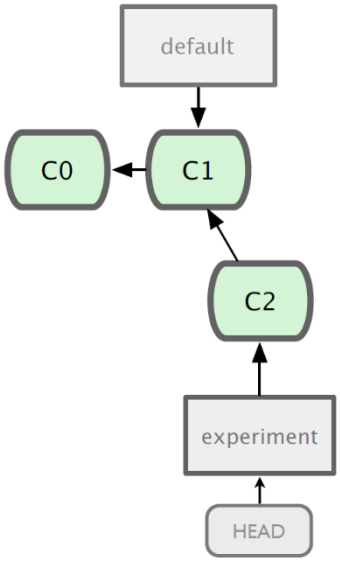
Changing the active branch

- One branch can be active at a time
- `git checkout` *`branchname`*

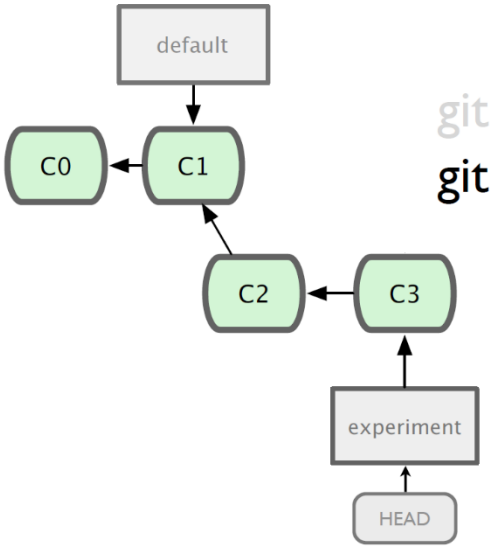


git checkout experiment

Creating a commit on a branch



git commit

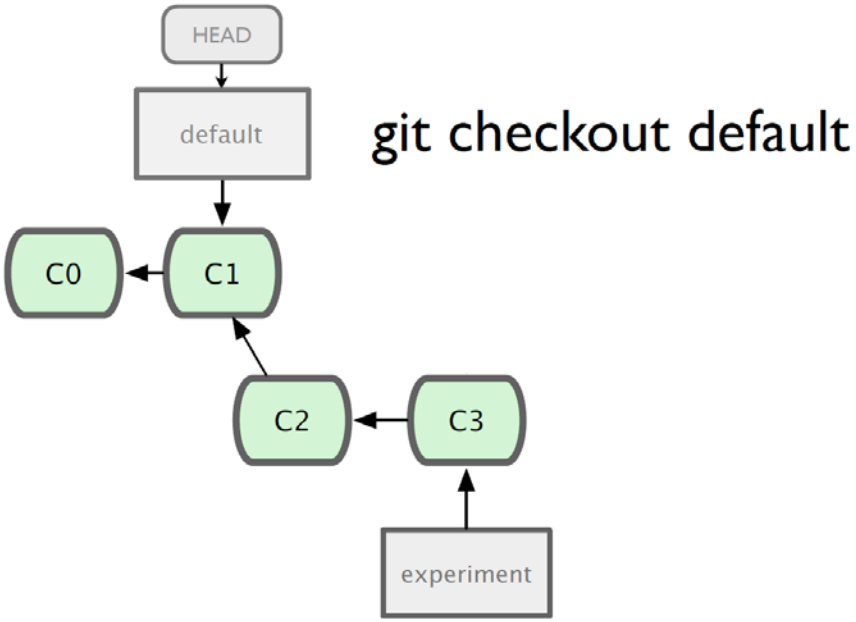


git commit

git commit

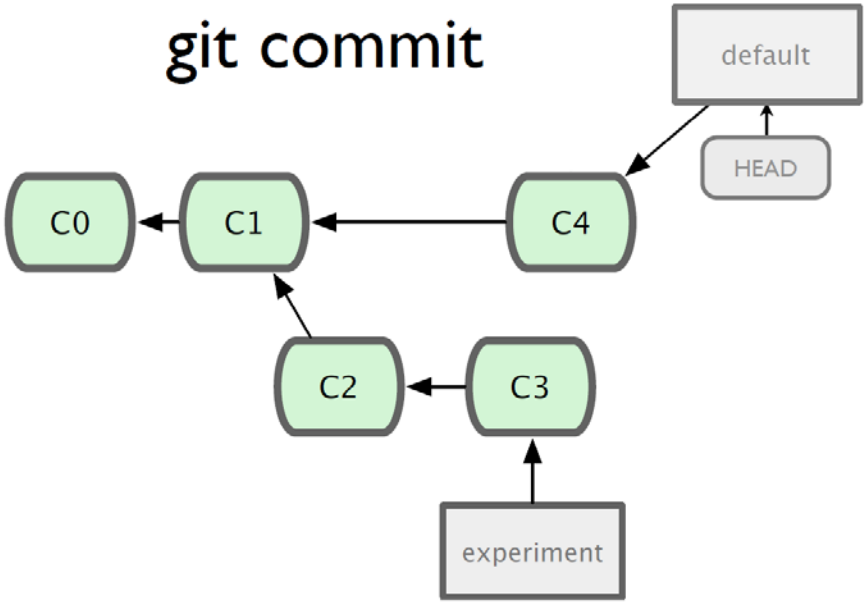
Multiple Branches

- At any time the active branch can be changed



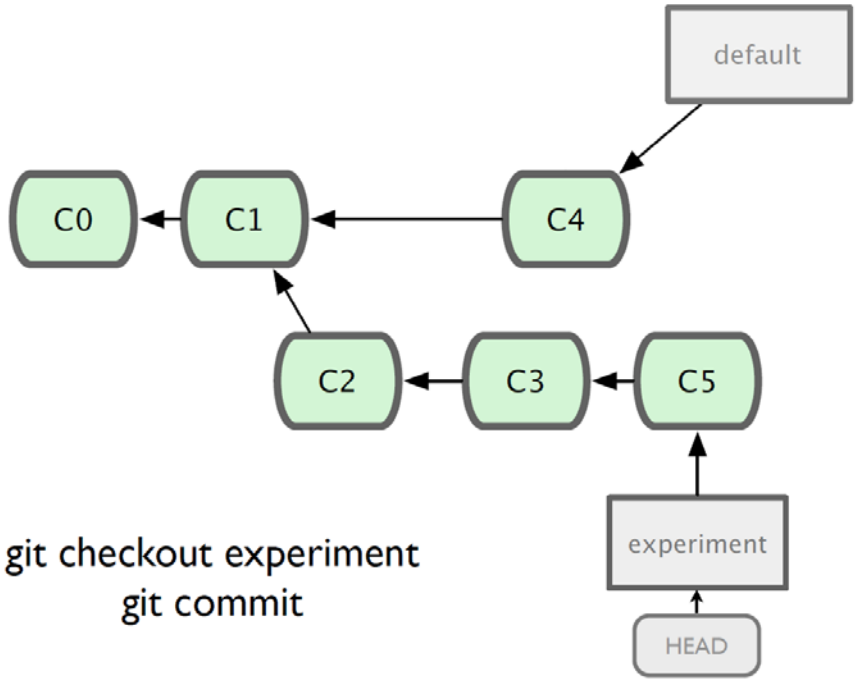
Committing in multiple branches

- Branches allow multiple change flows to be executed simultaneously



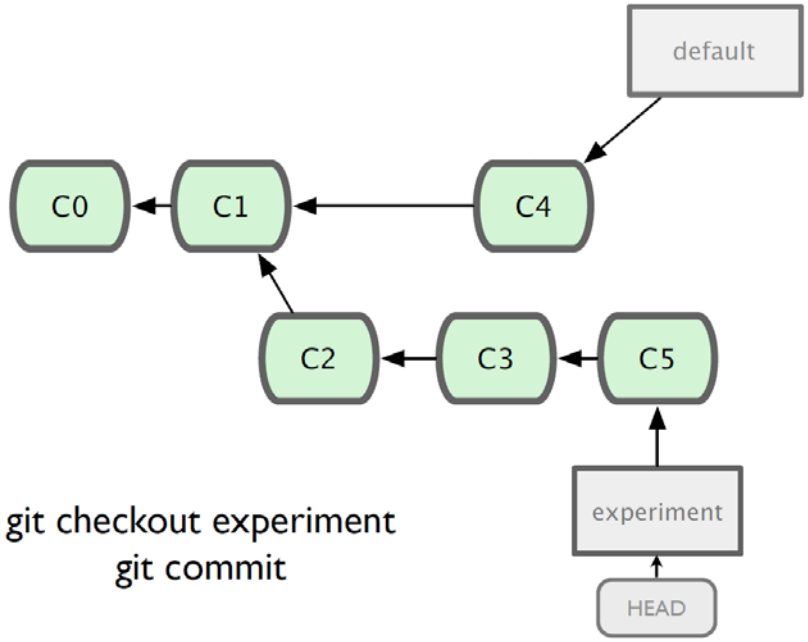
Developing alternate branches

- Switching to the active branch allows commits to occur on other work flows.



Combining multiple branches

- After developing multiple flows differences need to be aggregated
- What do we do with this mess?
  - Merge them



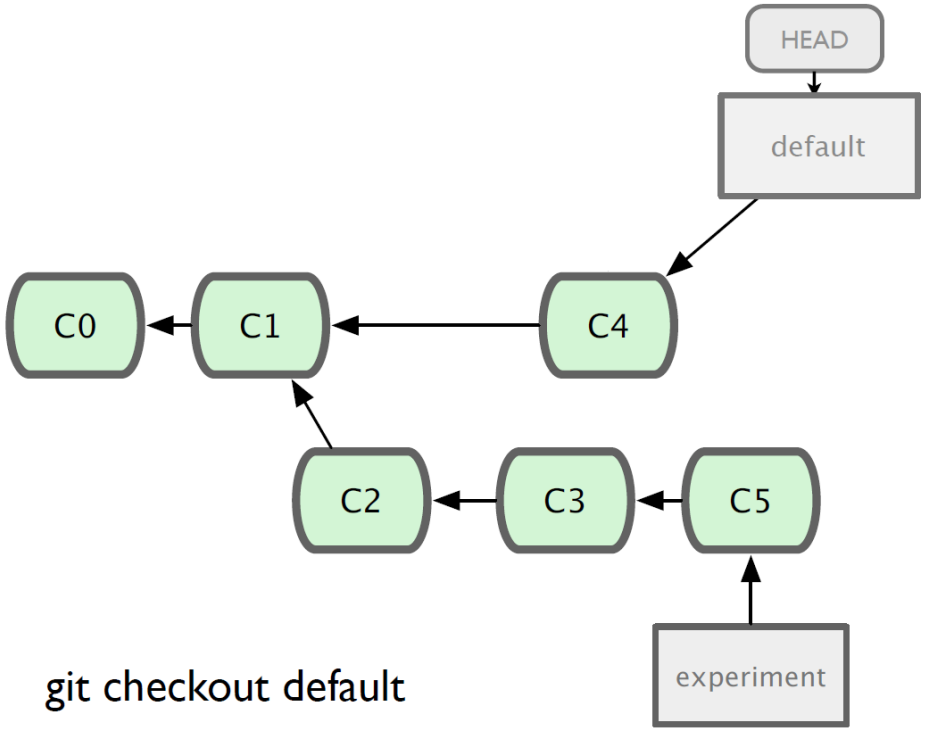
Merging multiple branches

- Steps to merge two branches
  - Checkout the branch you want to merge into
  - Merge the branch you want to merge

    ```
    merge otherbranch
    ```

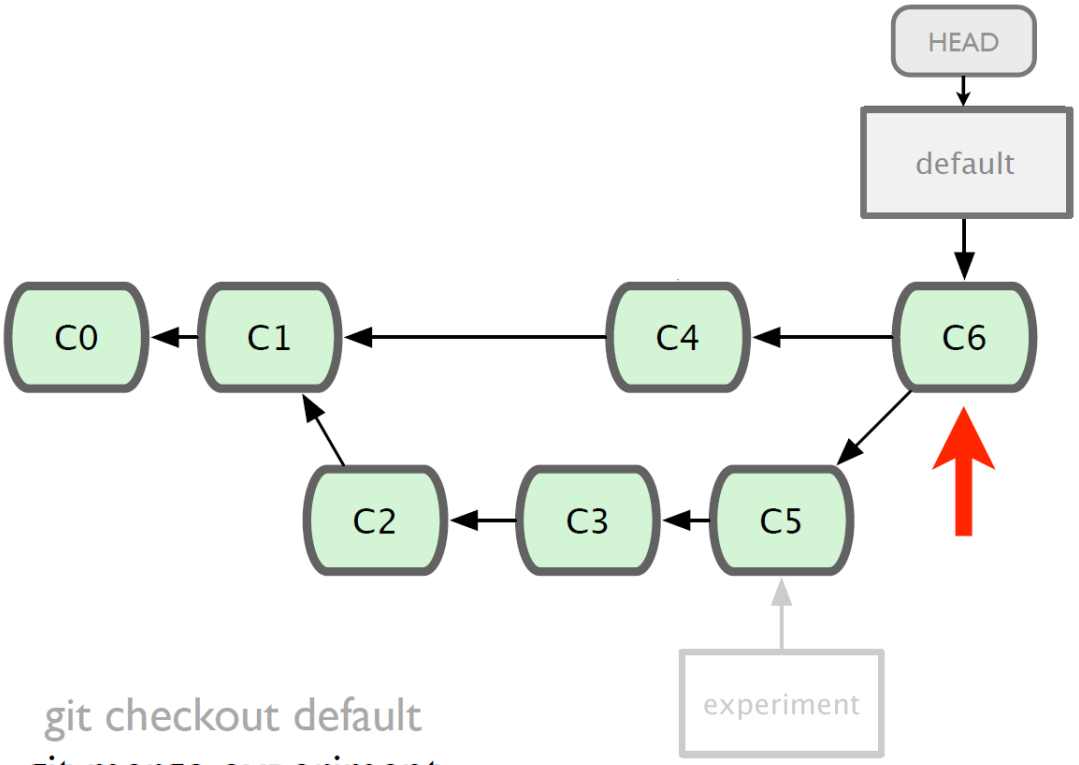
  - Brings the changes from the "`otherbranch`" into the current branch

Merging Step 1



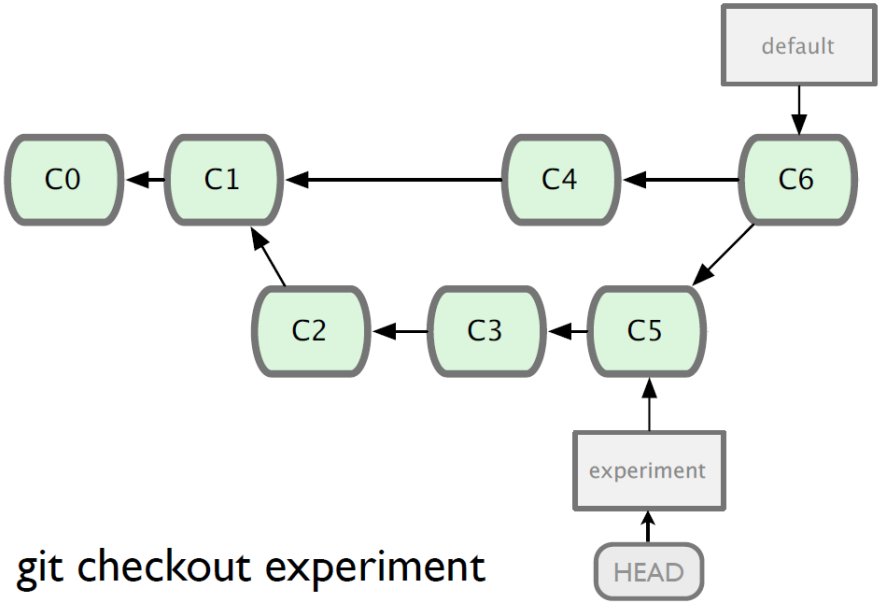
git checkout default

Merging Step 2



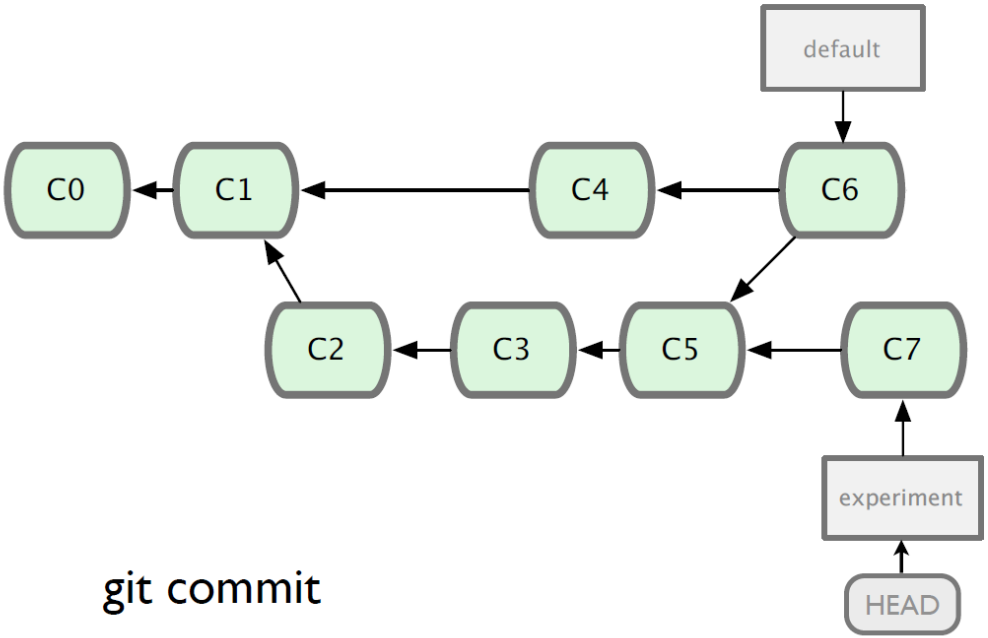
git checkout default
git merge experiment

Merging Step 3



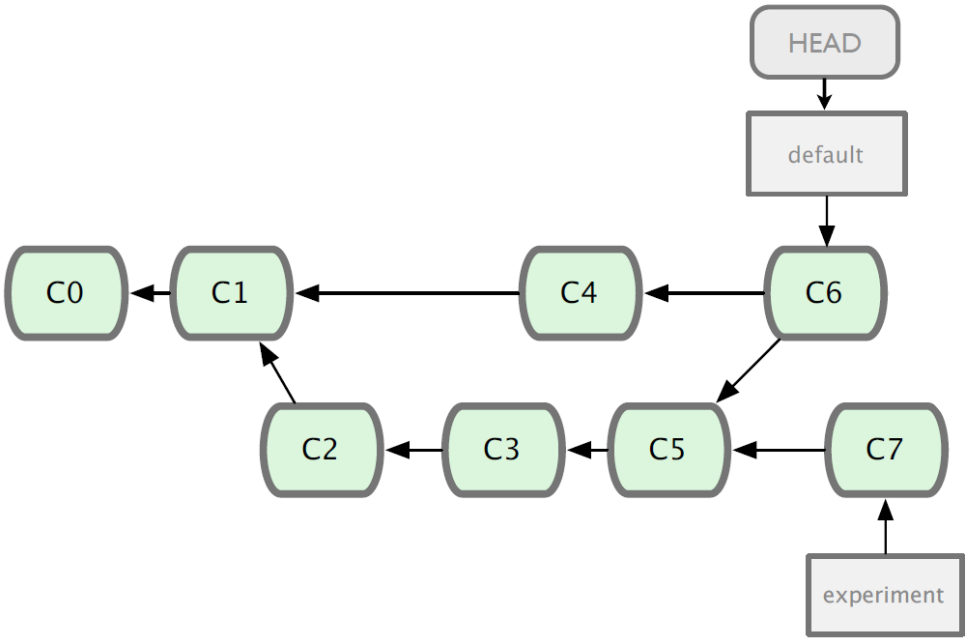
git checkout experiment

Merging Step 4



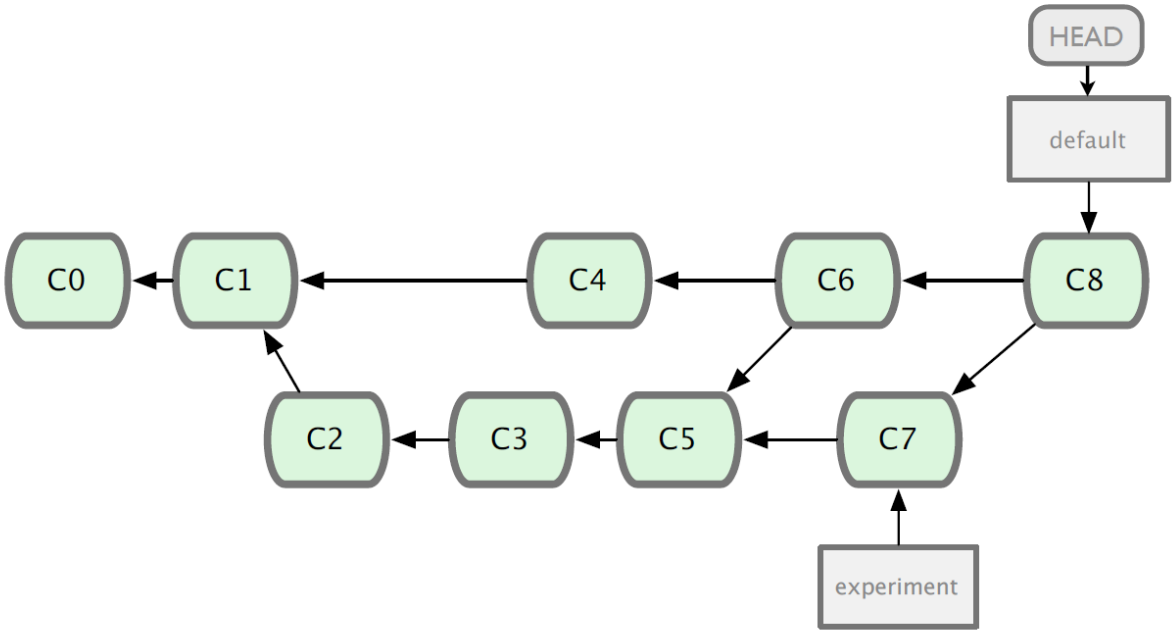
git commit

Merging Step 5



git checkout default

Merging Step 6



git merge experiment

Branching and Merging

- Why this is cool?
  - Non-linear development

```
clone the code that is in production
create a branch for issue #53 (iss53)
work for 10 minutes
someone asks for a hotfix for issue #102
checkout ‘production’
create a branch (iss102)
fix the issue
checkout ‘production’, merge ‘iss102’
push ‘production’
checkout ‘iss53’ and keep working
```

Git Source Control

Exercise 3

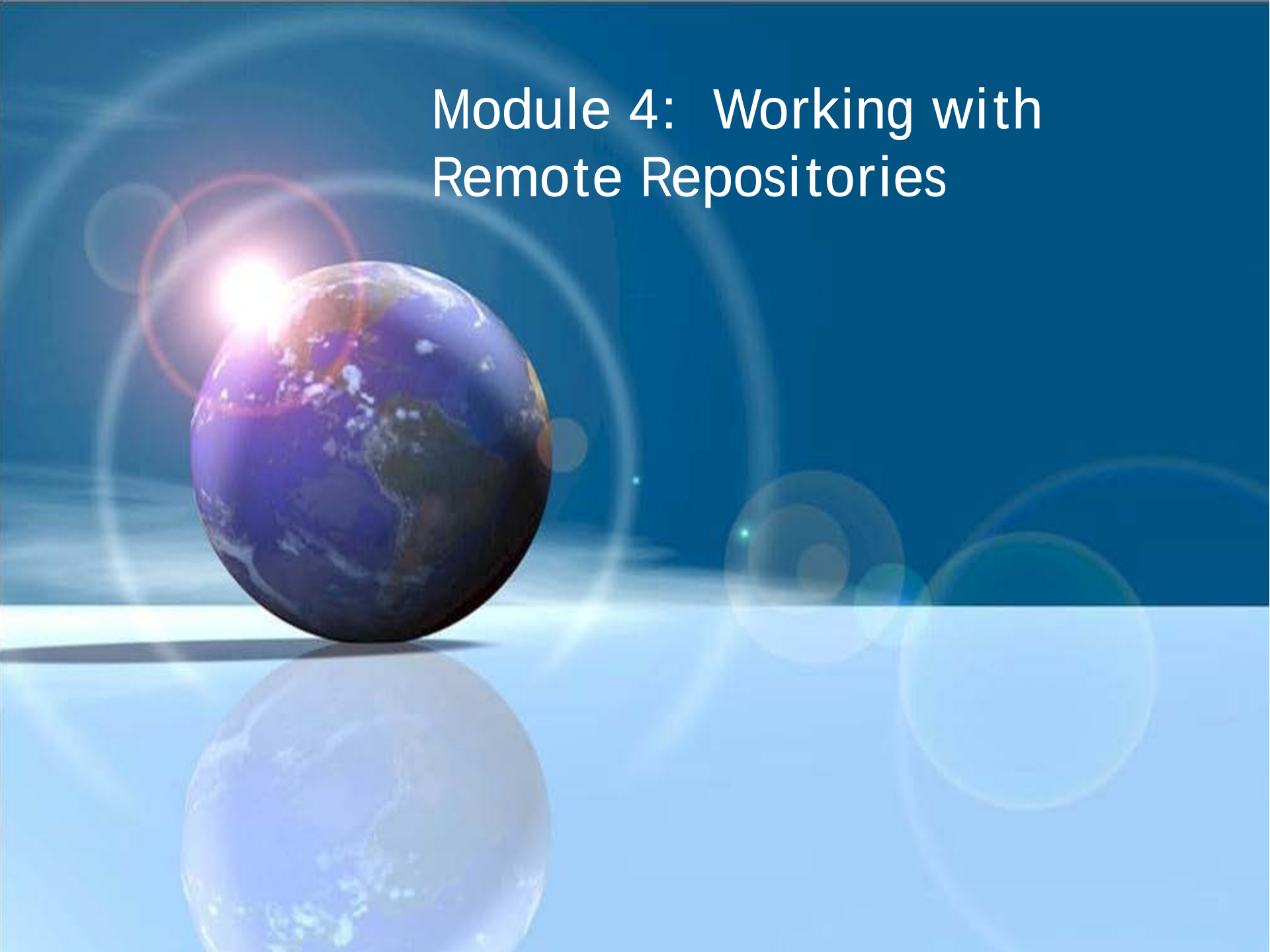
Refer to Exercise 3 in the back of the book

Git Source Control

Summary

- Repositories can have parallel development tracks - Branches
- Branches can have their own commit histories
- Multiple branches can be joined together with the merge command

Module 4: Working with Remote Repositories



Git Source Control

Overview

- Remote Repositories
- Replicating remote repositories
  - GitHub
- Synchronizing to and from remotes

Working with remote

- Use git clone to replicate repository
- Get changes with
  - `git fetch`
  - `git pull` (fetches and merges)
- Propagate changes with
  - `git push`
- Protocols
  - Local filesystem (file://)
  - SSH (ssh://)
  - HTTP (http:// https://)
  - Git protocol (git://)

Working with remote Local filesystem

- Pros
  - Simple
  - Support existing access control
  - NFS enabled
- Cons
  - Public share is difficult to set up
  - Slow on top of NFS

Working with remote SSH

- Pros
  - Support authenticated write access
  - Easy to set up as most system provide ssh toolsets
  - Fast
    - Compression before transfer
- Cons
  - No anonymous access
    - Not even for read access

Working with remote GIT

- Pros
  - Fastest protocol
  - Allow public anonymous access
- Cons
  - Lack of authentication
  - Difficult to set up
  - Use port 9418
    - Not standard port
    - Can be blocked

Working with remote HTTP/HTTPS

- Pros
  - Very easy to set up
  - Unlikely to be blocked
    - Using standard port
- Cons
  - Inefficient

Working with remote

- One person project
  - Local repo is enough
  - No need to bother with remote
- Small team project
  - SSH write access for a few core developers
  - GIT public read access

Aside: So what is github?

- GitHub.com is a site for online storage of Git repositories.
- Many open source projects use it, such as the Linux kernel.
- You can get free space for open source projects or you can pay for private projects.

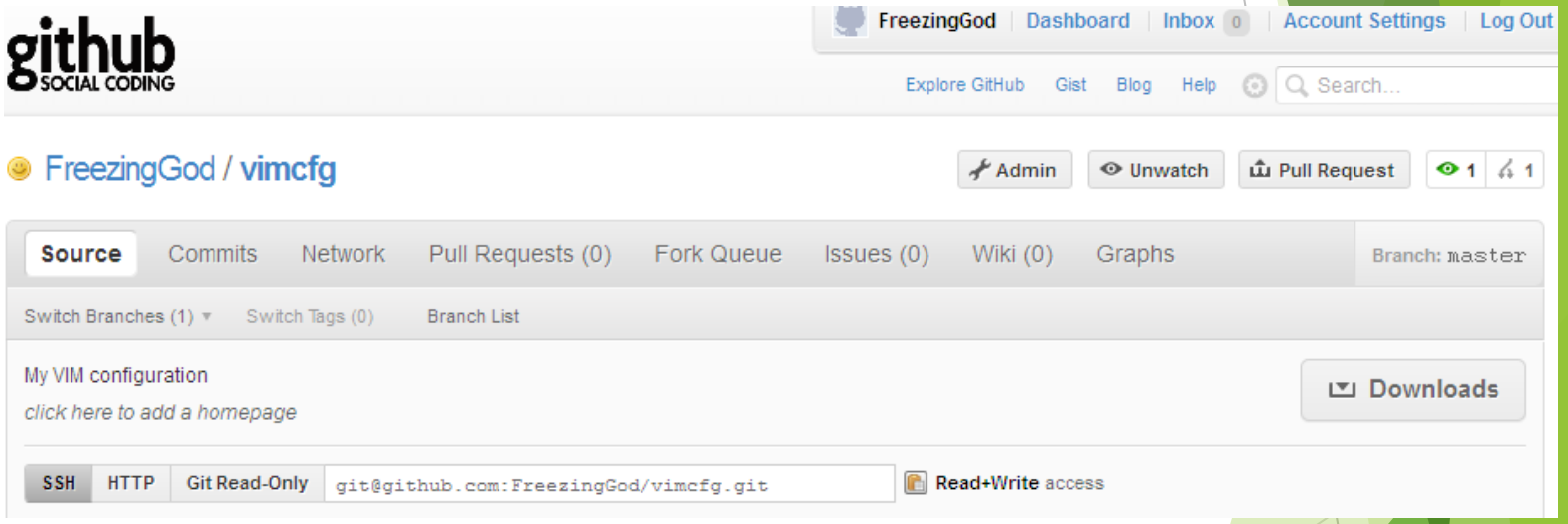
Question: Do I have to use github to use Git?

Answer: No!

- you can use Git completely locally for your own purposes, or
- you or someone else could set up a server to share files, or
- you could share a repo with users on the same file system(as long everyone has the needed file permissions).

Working with remote

- Use git remote add to add an remote repository




```
git remote add origin git@github.com:GitStudent/vimcfg.git
~/.vim_runtime$ git remote
origin
```

Pulling and Pushing

Good practice:

1. **Add** and **Commit** your changes to your local repo
2. **Pull** from remote repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
3. **Push** your changes to the remote repo

To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:

```
$ git pull origin master
```

To push your changes from your local repo to the remote repo:

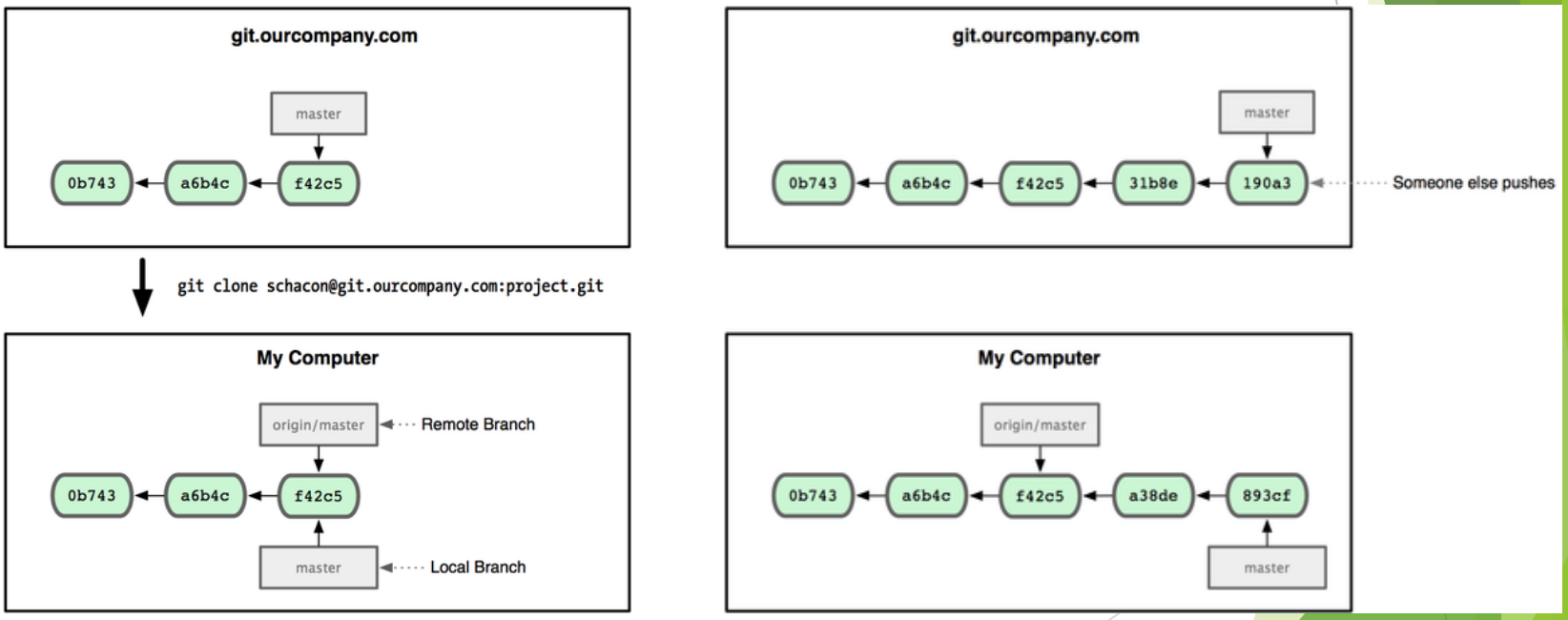
```
$ git push origin master
```

Notes: **`origin`** = an alias for the URL you cloned from

`master` = the remote branch you are pulling from/pushing to,
(the local branch you are pulling to/pushing from is your current branch)

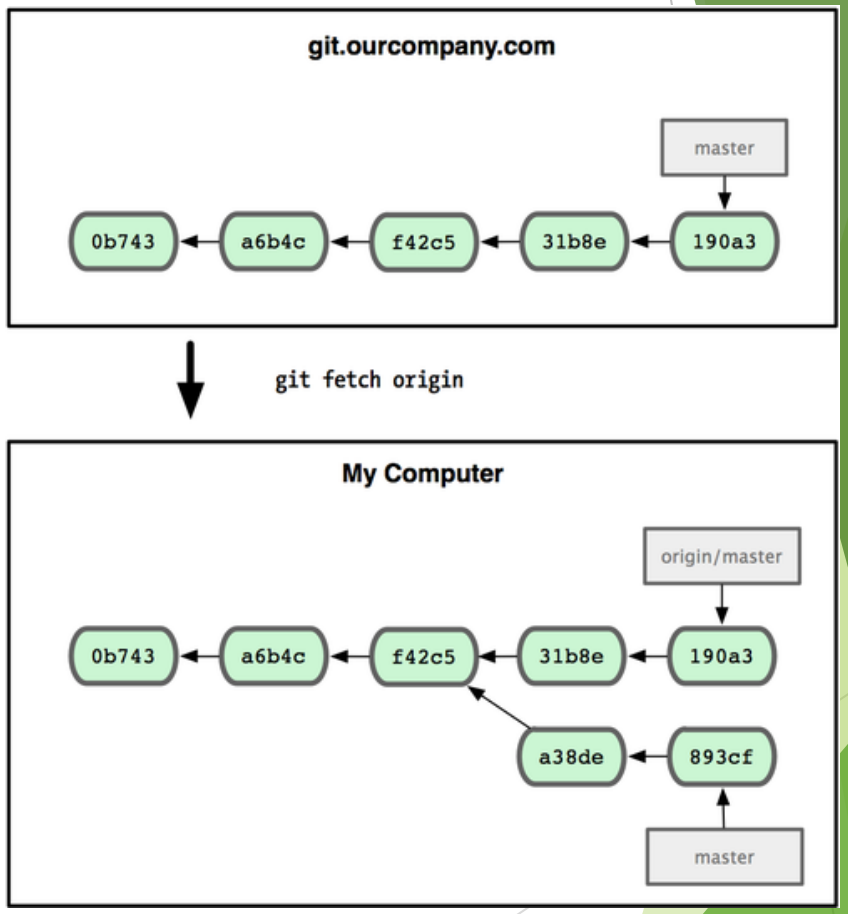
Working with remote

- Remote branching
  - Branch on remote are different from local branch



Working with remote repositories

- Remote branching
  - Branch on remote are different from local branch
  - git fetch origin to get remote changes
  - git pull origin try to fetch remote changes and merge it onto current branch



Git Source Control

Exercise 4

Refer to Exercise 4 in last section of book

Module 5: Git Internals

Git Objects

- Git is a simple key-value data store
  - You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time
- Git stores objects in the objects directory
- The key for the stored objects is the object hash
- The plumbing command `hash-object`, takes some data, stores it in your .git directory, and gives you back the key the data is stored as

hash-objects

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Extracting Objects

- You can pull the content of an object back out of Git with the `cat-file` command.
  - This command is a utility for inspecting Git objects.
- Passing -p to it instructs the cat-file command to figure out the type of content

```
$ cd objects/d6
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Git Folder Structure

- Within a repository git will create a .git folder
- A .git directory has a structure similar to the following one:
  - COMMIT_EDITMSG
    - This is the last commit message. It's not actually used by Git at all, but it is for your reference after you have made a commit.
      `cat COMMIT_EDITMSG first commit`
  - config
    - This is the main Git configuration file.
  - description
    - Documentation message for gitweb or git instaweb
  - gitweb
    - A folder with the GIT web scripts. They allow you to browse the git repository using a web browser.

Git Folder Structure Cont.

- HEAD
  - This file holds a reference to the branch you are currently on. This tells Git what to use as the parent of your next commit: `cat HEAD ref: refs/heads/master`
- hooks/
  - This directory contains shell scripts that are invoked after the corresponding Git commands. For example, after you run a commit, Git will try to execute the post-commit script.
- index
  - The Git index is used as a staging area between your working directory and your repository.
- info/
  - Contains additional information about the repository.
- logs/
  - Keeps records of changes made to refs.

Git Folder Structure Cont.

- objects/
  - In this directory the data of your Git objects is stored – all the contents of the files you have ever checked in.
- ORIG_HEAD
  - This is the previous state of HEAD.
- packed-refs
  - The file consists of packed heads and tags. It is useful for an efficient repository access.
- refs/
  - This directory normally contains three subfolders – heads, remotes and tags. There you will find the corresponding local branches, remote branches and tags files.

Git Trees

- The git system is composed of 3 "Trees"
  - The HEAD and Working Directory are definitely trees
  - The index is arguably a tree :-)
- The Three Trees
  - HEAD is the last commit, the "saved state"
  - Working directory is your current folder
  - The Index is the proposed commit snapshot
- Commands like checkout and commit manipulate the trees

Git Index

- The git “index” is where you place files you want committed to the git repository.
- Before you “commit” (checkin) files to the git repository, you need to first place the files in the git “index”
- The git index goes by many names. But they all refer to the same thing. Some of the names you may have heard:
  - Index
  - Cache
  - Directory cache
  - Current directory cache
  - Staging area
  - Staged files

File System Check

- Git automatically runs a command called “`auto gc`”.
  - gathers up all the loose objects and places them in packfiles
- With Git being a filesystem there can at times be "issues"
  - The `git fsck` command is able to check Git’s database and verify the validity and reachability of every object that it finds.

```
$ git branch
  feature-branch
* master
$ git rev-parse --short feature-branch
f71bb43
$ git branch -D feature-branch
Deleted branch feature-branch (was f71bb43).
$ git fsck
```

Pruning

- When working with remote repositories it is common the many branches are created
  - Branches "should" be deleted when developers are done with them
  - Branches that are deleted on the remote may still have references on local clones
- Objects which are no longer referenced can be evicted with git prune;

```
$ git checkout develop

$ git fetch

$ git remote prune your_remote # Don't show branches
that have already been deleted

$ git branch -a --merged # This will show all branches
that have been merged into develop
```

filter-branch

- `git rm` will remove a file from the HEAD of the repository
  - The files will still exist in the remaining branches
- `git filter-branch` can walk all the branches of a repository and execute actions on each branch
  - Delete sensitive data files that were accidentally added
  - Rename files

```
$ git filter-branch --tree-filter 'rm -rf
vendor/yourfolder' HEAD

#removing the folder from all branches

$ git filter-branch --tree-filter 'rm -rf
vendor/yourfolder' 7b3072c..HEAD

#removing the folder from a range of commits

$ git filter-branch --tree-filter 'rm -f filename' HEAD
#removing from all snapshots of the current branch
```

Git Source Control

Exercise 5

- Open your lab guide and proceed through Lab Exercise #5

Module 6: Git Configuration

Environment Variables

- Git runs inside a shell (Generally bash)
  - Variables control how Git behaves
    - HOME - where Git looks for the configuration file
    - GIT_EDITOR - The default editor for Git Commits
    - GIT_DIR - The location of the .git file
    - GIT_AUTHOR_NAME- The name of the code author
    - GIT_AUTHOR_EMAIL - The email address of the author
    - GIT_CURL_VERBOSE - Verbose CURL Networking (debugging)

Variable Locations

- Variables will generally be set in one of the following 3 locations:
  - /etc/gitconfig - Values for every user of the system and all of their repositories
  - ~/.gitconfig - Values that are tied to this particular user
  - $GIT_REPO/config - Values that are tied to this particular repository
- Each level in the above list overrides settings in the prior level
  - .gitconfig overrides /etc/gitconfig

Local Variables

- Each repository has a set of local variables
  - The `git config` command sets/unsets these options
- `git config` by itself displays the usage of the command
- `git config --list` displays all your current configuration variables
- `git config` *`variablename value`* assigns a value to that variable

Using git config

- Setting the user.name property for the current repo:

```
$ git config user.name "Delois Price"
# Set a new name
$ git config user.name
# Verify the setting
# Delois Price
```

- Setting the same property globally (For all repo's)

```
$ git config --global user.name "Delois Price"
```

Git Attributes

- Git allows the definition of attributes for files within a repository
  - Attributes can be specified on a file by file or on a directory basis
  - A file (.gitattributes) within a folder of your repository will store the attributes for that folder
- Git Attributes allow individual files to be treated differently than they normally would
  - Leave out particular files from Git
  - treat certain files as binary

Git Source Control

Exercise 6

- Open your lab guide and proceed through Lab Exercise #6

Module 6: Advanced Git Commands

Git Aliases

- Aliases are shortcuts to Git commands
  - Instead of all the crazy typing for commands like checkout we can shortcut it to something like co:

    ```
    git config --global alias.co checkout
    ```

  - Aliases are stored in your .gitconfig under the alias section
    - `co = checkout`
      - (Instead of git checkout my-branch it is now git co my-branch)
    - `ec = config --global -e`
      - (Open the git config with the default editor)

Finding Content

- So.. You have put a resource in the Git repository, but now you can't find it.
- The git describe command creates unique identifiers for the particular version of a file
  - Allows you to retrieve whatever version you want
- The git grep command searches the repository for the pattern you specify
  - Many options are provided to allow very complex searching

Git Describe

- At its simplest git describe will output something that looks similar to this: v1.5.5.1-178-g1f8115b
  - Version 1.5.5.1 is the start point
  - 178th commit past that version
  - g18115b is the hash for this file
- git describe will return the most recent tag that is reachable from a commit

```
git describe
```

Git Grep

- Git grep finds things.
  - We can suggest what to find and where to look
  - We can also ask it to return various information about the finds
- Assume the following code problem:

```
for (i = 0; i < 256; i++) {
                struct object_entry **next = c;;
                while (next < last) {
```

- The `git grep -e  ';;'` will display all the matches
- Common git grep options:

```
# return the line number where the match was found
PS:\> git grep -n monkey
# return just the file names
PS:\> git grep -l monkey
# count the number of matches in each file
PS:\> git grep -c monkey
```

Debugging Git

- An issue occurs when a bug is introduced into your codebase
  - The bug may be significant enough to prevent future commits until the bug is cleaned up
  - You could randomly search for prior commits that do not have the bug, but on a large codebase with many developers this would be a nightmare.
  - The blame utility can let you see where a particular change was introduced
  - the bisect utility walks the commit tree displaying the differences from two known commits

Git Blame

- git blame helps you locate changes to a file between commits
- You would need to know what you are looking for
  - Usually a filename
- Changes to a file are listed along with when they edited the file
  - It is a file by file history

Git Bisect

- git bisect is showing you the changes that were made between commits
  - Essentially the deltas from the two commits
- Great when you do not know what the bug is caused by, but you know when the bug was inserted

```
git bisect start # Initialize the bisect
git bisect bad # Set the end commit (or current if ommitted
git bisect good initialcommit # define the start of the bisect
```

Git stash

- Stashing takes the dirty state of your working directory
  - That is, your modified tracked files and staged changes
  - Saves it on a stack of unfinished changes that you can reapply at any time.
- If you are working on a branch and need to switch to another branch for some reason, you will need to do something with the work you have done in your current branch
  - Instead of performing a commit on partially "done" files, stash stores the changes so you can re-enter that point later

```
git stash
```

Git stash cont.

- Users of stash can have multiple stash*ed* segments
  - It is a set of stored changes that can be re-applied
- Once a stash is created you can list available stashes:

```
git stash list
```

- stashes can be applied to a branch

```
git stash apply
```

- If you want to delete a stash:

```
git stash drop
```

Archives (send & receive)

- To create a backup of the repository you can clone the repository and then manually archive it with zip or tar
- The git archive command creates a zip file from the archive
  - git archive will not backup the .git folder which will generally be very large

```
git archive master --format=zip --output=../dev-master.zip
```

Patching

- Generally when a commit is applied, we are considering the commit as a full version of the project
  - A commit is a working copy as of this point in time
- Git provides a few commands that allow you to think of committing changes

```
git format-patch master > newcontents.patch
```

  - This will take all the differences from the current branch (Assumed not to be the master branch) and creates a patch to bring master up to the current branches state
- Patches can be applied to a branch

```
git apply --stat newcontents.patch
```

  - Show the differences that will happen if applied

```
git check --stat newcontents.patch
```

  - Confirm that the patch will work

```
git am --signoff <newcontents.patch
```

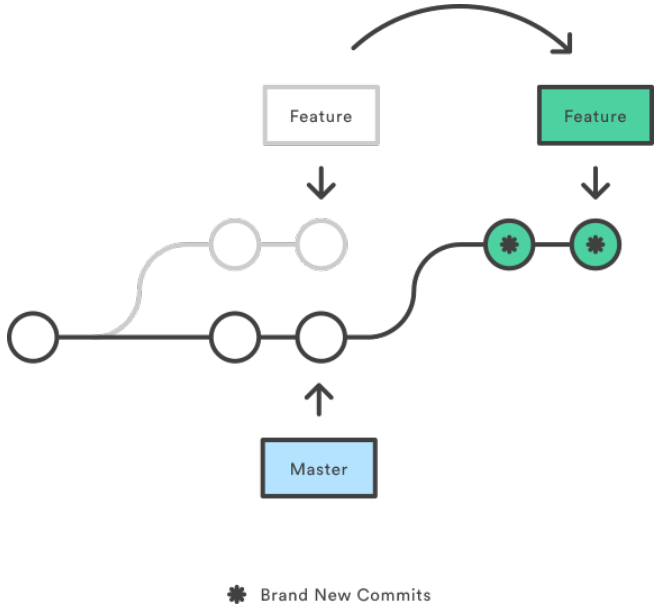
  - Put the patch into the master branch

Dry-runs

- Often we want to know what a command will do before we actually do the command
  - Dry run is a concept. Many commands have the concept, but use a variety of ways to set it
- Many commands have a dry-run option:
  - `git clean -f` will remove all untracked files from the repository
    - This could be dangerous
  - `git clean -n is a dry run`
  - `git merge --dry-run`
- Many commands have a dry run option. To be safe, just check the help docs for the command you want to dry run

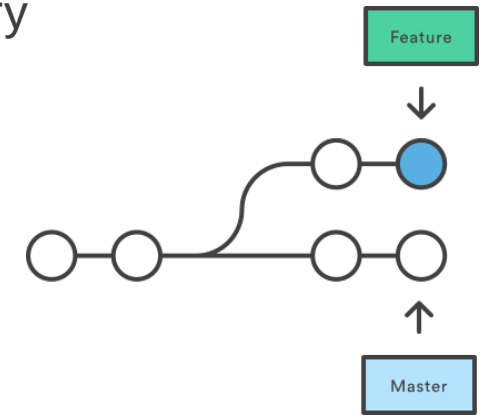
Rebasing

- Rebasing is the process of moving a branch to a new base commit
  - Branches are based on a master version
  - When a branch is created it is created from a particular base
  - rebasing is moving a branch from being based on one commit to being based on another



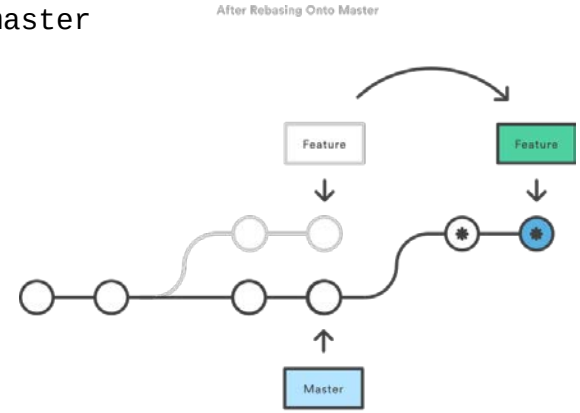
Rebasing cont.

- The primary reason for rebasing is to maintain a linear project history



- The master has progressed since the feature was added

```
git rebase master
```



Cherry-Picking

- The cherry-pick command allows you to merge a single commit from one branch into another
  - cherry-pick when a full branch merge is not possible due to incompatible versions or because you do not want everything in the other branch
    - Git copies changes made in one place, and replays them somewhere else
  - git cherry-pick is a limited form of git rebase
    - Instead of rebasing the entire branch, it allows the individual commit to be moved

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
            \ 76cada - 62ecb3 - b886a0 [feature]
```

```
git checkout master
git cherry-pick 62ecb3
```

Git Source Control

Exercise 7

- Open your lab guide and proceed through Lab Exercise #7

Module 7: Git Collaboration

Working nicely with others

- Repositories are local or remote
  - local are on your box
  - remote are not
- Remote Repositories are intended to be shared
  - github is a very common public site for git repositories
- It is very common for a local repository to be replicated to a remote site
  - Backup
  - Sharing among other developers

Configuring Remotes

- Git repositories are built to support distributed development
  - Repositories can easily be remote
  - A "server" needs to provide the networking
- Centralized, remote repositories enable multiple developers to work concurrently
- Team development has challenges … -- Seriously..
  - Stomping on toes (developers writing the same resource)
  - sharing updates (One developer depends on another developer)
  - re-fixing updates (One developer re-inserting a old bug)

Configuring Remotes

- Local repositories can reference any number of remote servers
- View your remotes with git remote
  - The remote sites are used with commands like git push, git pull, git fetch
- A default remote named origin is usually the default
- Add a remote site with:
  - git remote add origin http://www.yourremote.com/repo.git

Refspecs

- Branches in the local repository can map to branches in the remote repository
  - A refspec is specified as [+]<src>:<dst>.
  - The <src> parameter is the source branch in the local repository
  - The <dst> parameter is the destination branch in the remote repository.
- Local commands can actually be working with a remote repository

```
git push origin master:refs/heads/qa-master
```

The master branch is pushed to the origin

The remote branch name will be qa-master

Sub-Modules

- sub-modules allow you to attach an external repository inside another repository at a specific path
  - allows you to include other projects in your codebase
  - solves the vendor library and dependency problems
- They are *not* automatically updated when the repository specified by the submodule is updated
  - They are updated when the parent project itself is updated
- A set of submodules can be synchronized on demand:

```
git submodule sync
```

Pulling and Pushing

Good practice:

1. **Add** and **Commit** your changes to your local repo
2. **Pull** from remote repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
3. **Push** your changes to the remote repo

To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:

```
$ git pull origin master
```

To push your changes from your local repo to the remote repo:

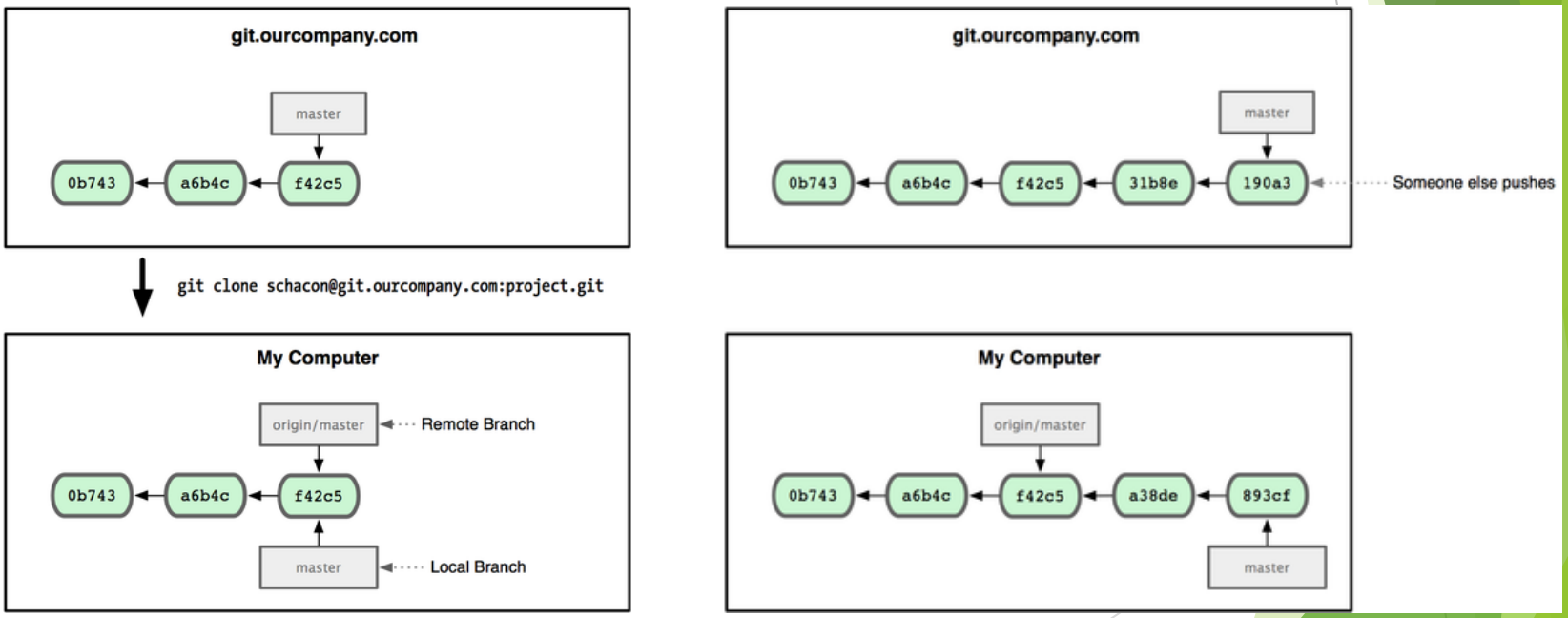
```
$ git push origin master
```

Notes: **`origin`** = an alias for the URL you cloned from

`master` = the remote branch you are pulling from/pushing to,
(the local branch you are pulling to/pushing from is your current branch)

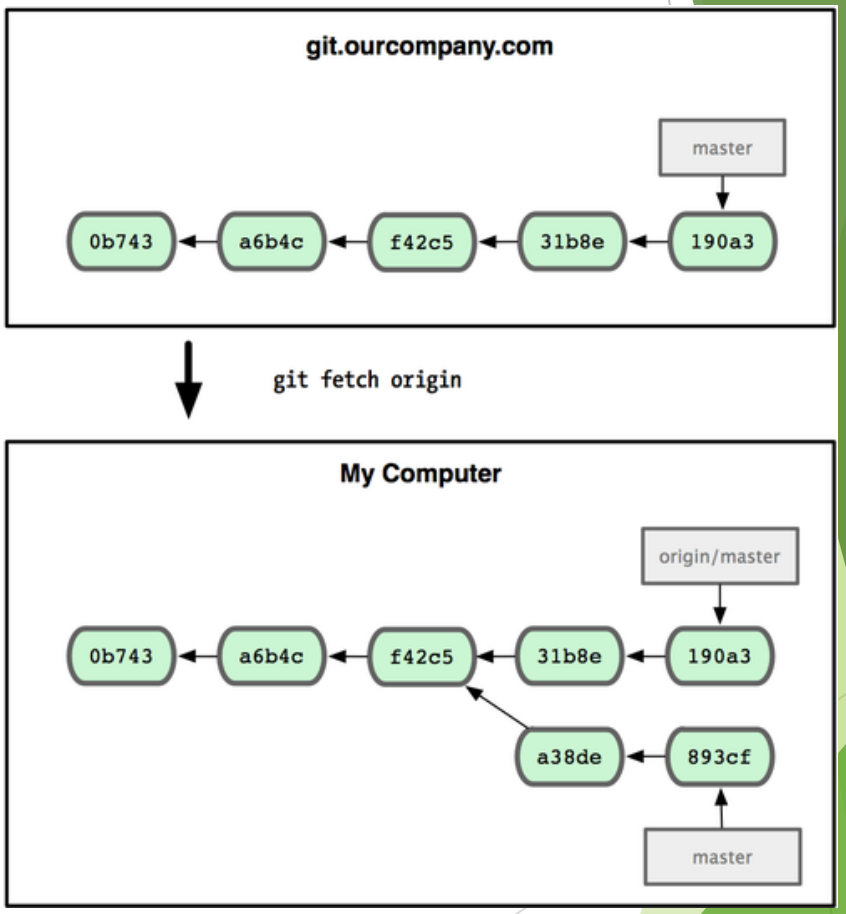
Working with remote

- Remote branching
  - Branch on remote are different from local branch



Working with remote

- Remote branching
  - Branch on remote are different from local branch
  - Git fetch origin to get remote changes
  - Git pull origin try to fetch remote changes and merge it onto current branch




Course Summary

References

- Some of the slides are adopted from “Introduction to Git” available at

http://innovationontherun.com/presentation-files/Introduction%20To%20GIT.ppt

- Some of the figure are adopted from Pro GIT by Chacon, which is available at

http://progit.org/book/

- Some of the slides are adopted from “Git 101” available at http://assets.en.oreilly.com/1/event/45/Git%20101%20Tutorial%20Presentation.pdf

Evaluations

- Please take the time to fill out an evaluation
- All evaluations are read and considered

Questions