

# Lesson 15

## Version Control with Git

September 16, 2015

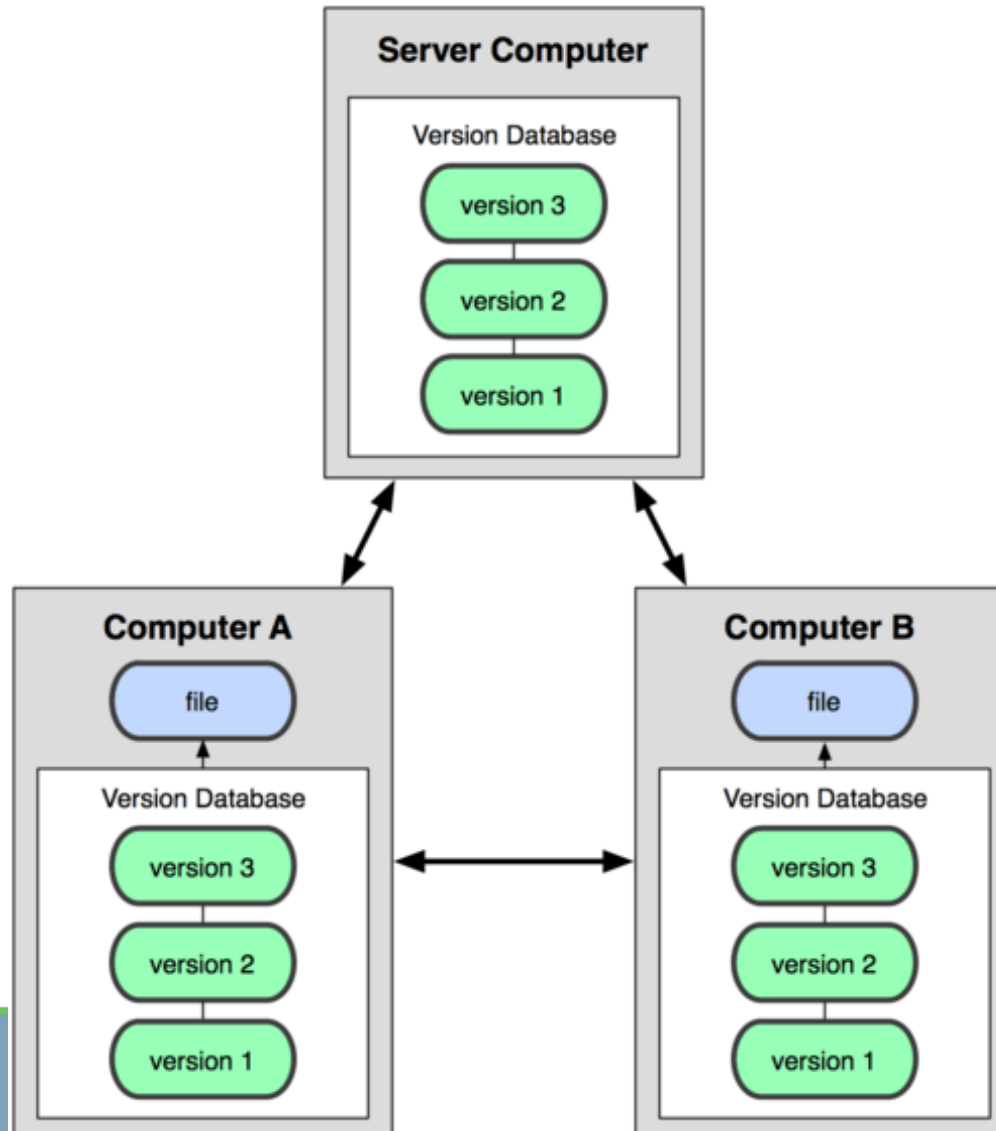
# A Short History of Git

- In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down.
- Linus Torvalds, the creator of Linux, developed their own tool, with the following characteristics:
  - Speed
  - Simple design
  - Strong support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like the Linux kernel efficiently (speed and data size)
- Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities.
- It's fast, very efficient with large projects, and it has branching system for non-linear development.

# Distributed Version Control Systems

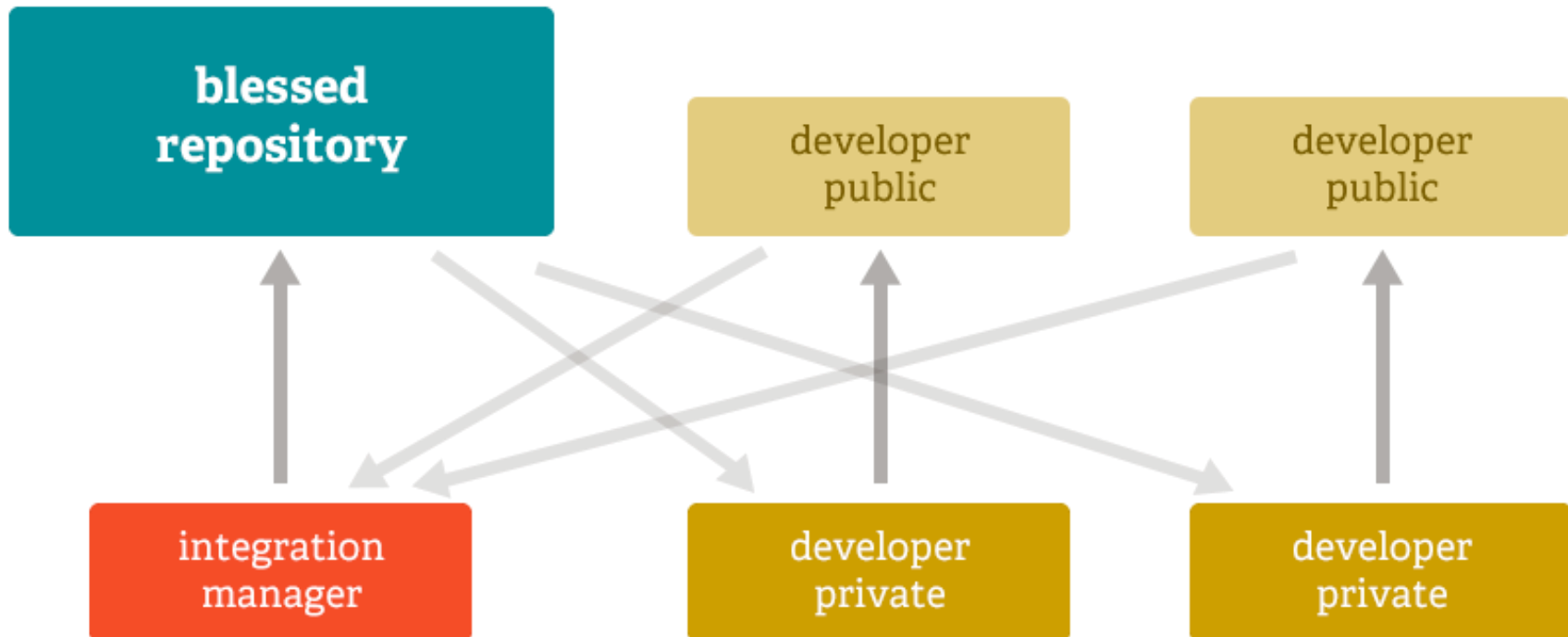
- Clients don't just check out the latest snapshot of the files: they fully mirror the repository.
- Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it.
- Every checkout is really a full backup of all the data
- DVCS: Git, Mercurial, Bazaar or Darcs

# Distributed Version Control Systems - schema



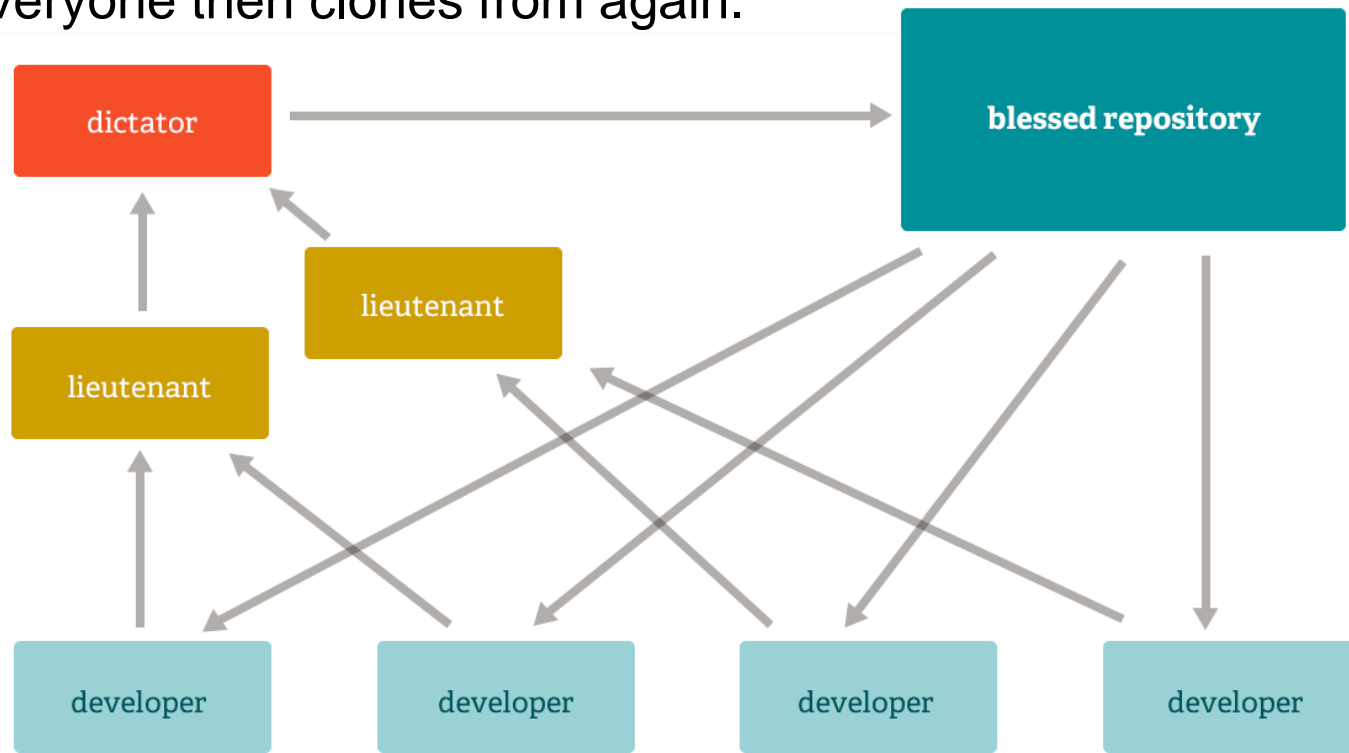
# Integration Manager Workflow

- Integration manager — a single person who commits to the 'blessed' repository.
  - A number of developers then clone from that repository, push to their own independent repositories, and ask the integrator to pull in their changes. → open source or GitHub



# Dictator and Lieutenants Workflow

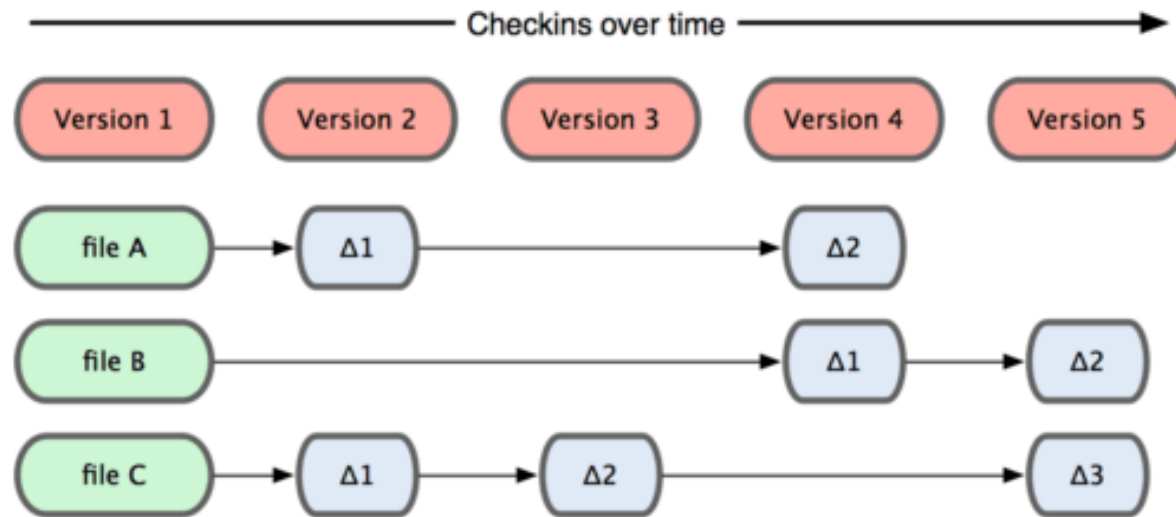
- For more massive projects, a development workflow like that of the Linux kernel is often effective.
  - Senior people ('lieutenants') are in charge of a specific subsystem of the project and they merge in all changes related to that subsystem.
  - Another integrator (the 'dictator') can pull changes from only his/her lieutenants and then push to the 'blessed' repository that everyone then clones from again.



# Snapshots, Not Differences - Other

- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.
- Most other systems store information as a list of file-based changes.
- These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time

# Snapshots, Not Differences - Other

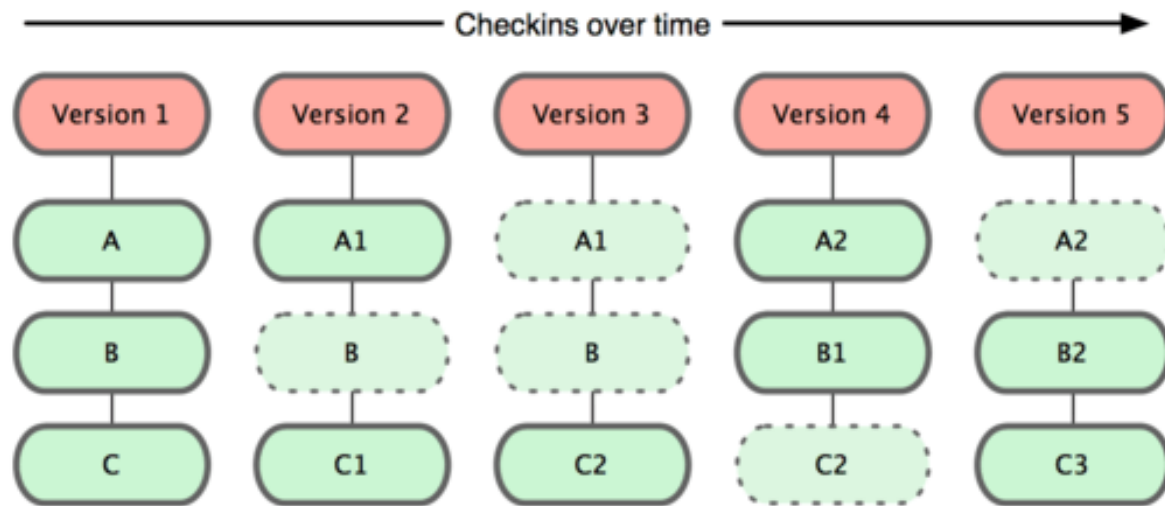




# Snapshots, Not Differences - Git

- Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a mini filesystem.
- Every time you commit, or save the state of your project in Git.
- It takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- If files have not changed, Git stores just a hard link to the previous identical file.

# Snapshots, Not Differences - Git



# Conclusions

- This is an important distinction between Git and nearly all other VCSs.
- It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation.
- This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS.

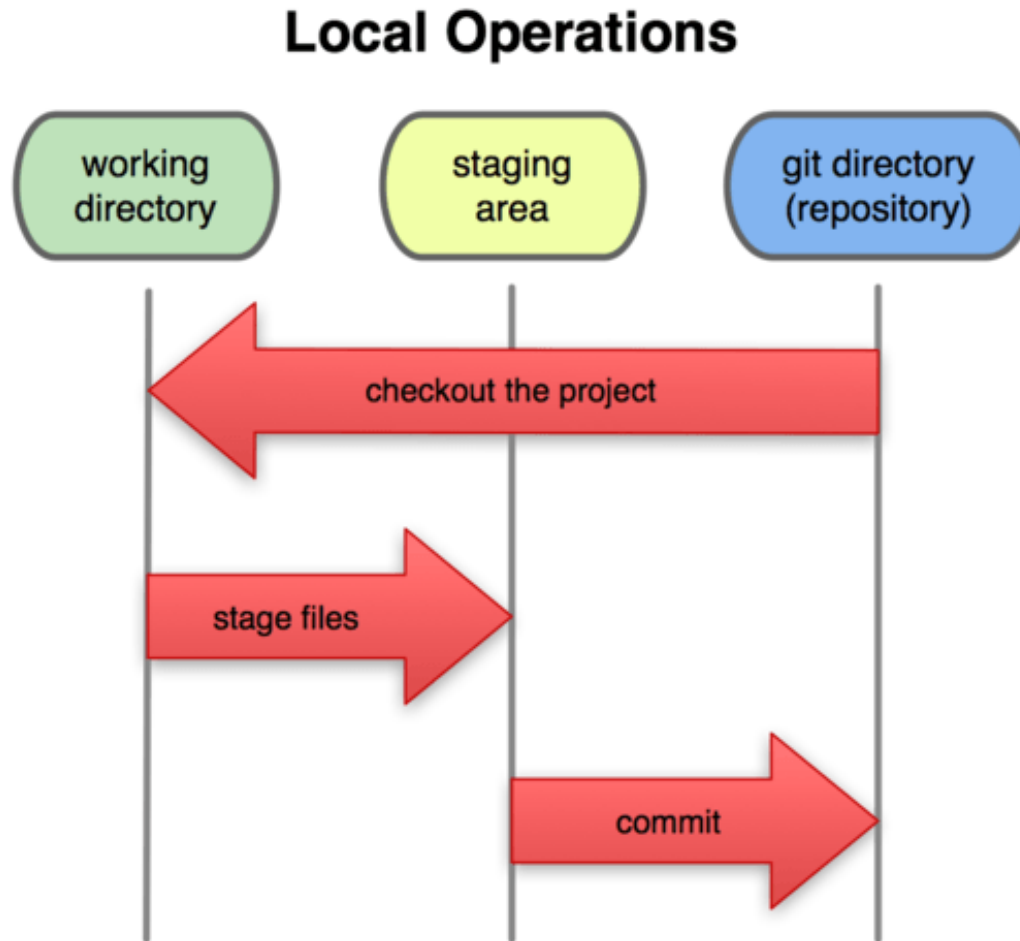
# Nearly Every Operation Is Local

- Most operations in Git only need local files and resources to operate.
- Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.
- If you get on an airplane or a train and want to do a little work, you can commit happily until you get to a network connection to upload

# The Three States

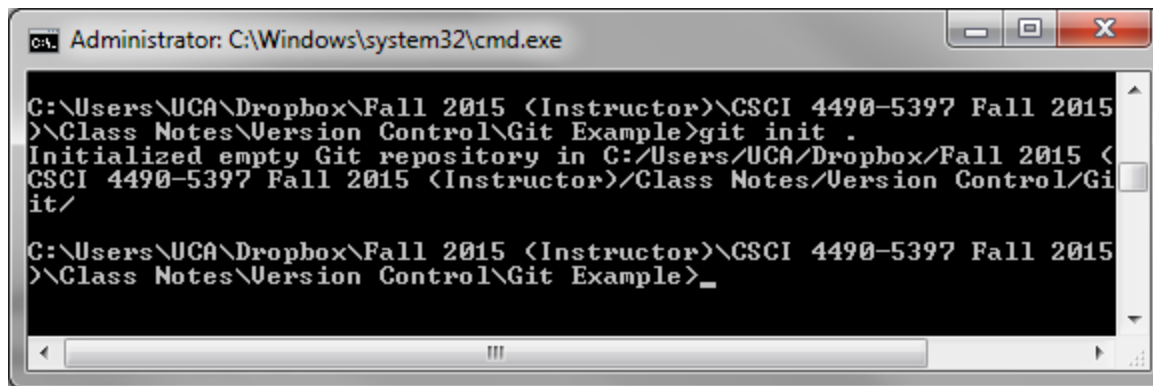
- Git has three main states that your files can reside in: committed, modified, staged.
- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

# The Three States - Local Operations



# Initializing a Git Repository

- Before you can work with Git, you have to initialize a project repository, setting it up so that Git will manage it. Open up your terminal, and in your project directory run the command `git init .` (the period is important) as shown in the screenshot below.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015
>\Class Notes\Version Control\Git Example>git init .
Initialized empty Git repository in C:/Users/UCA/Dropbox/Fall 2015 <
CSCI 4490-5397 Fall 2015 <Instructor>/Class Notes/Version Control/Gi
it/

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015
>\Class Notes\Version Control\Git Example>_
```

- A new hidden directory called `.git` will now be present in your project directory. This is where Git stores its database and configuration information, so that it can track your project.

# Cloning a Git Repository

---

- There's another way to access a repository, which is cloning. Similar to checking out a repository in other systems, running `git clone <repository URL>` will pull in a complete copy of the remote repository to your local system. Now you can work away on it, making changes, staging them, committing them, and pushing the changes back.



# Adding a New File to Git

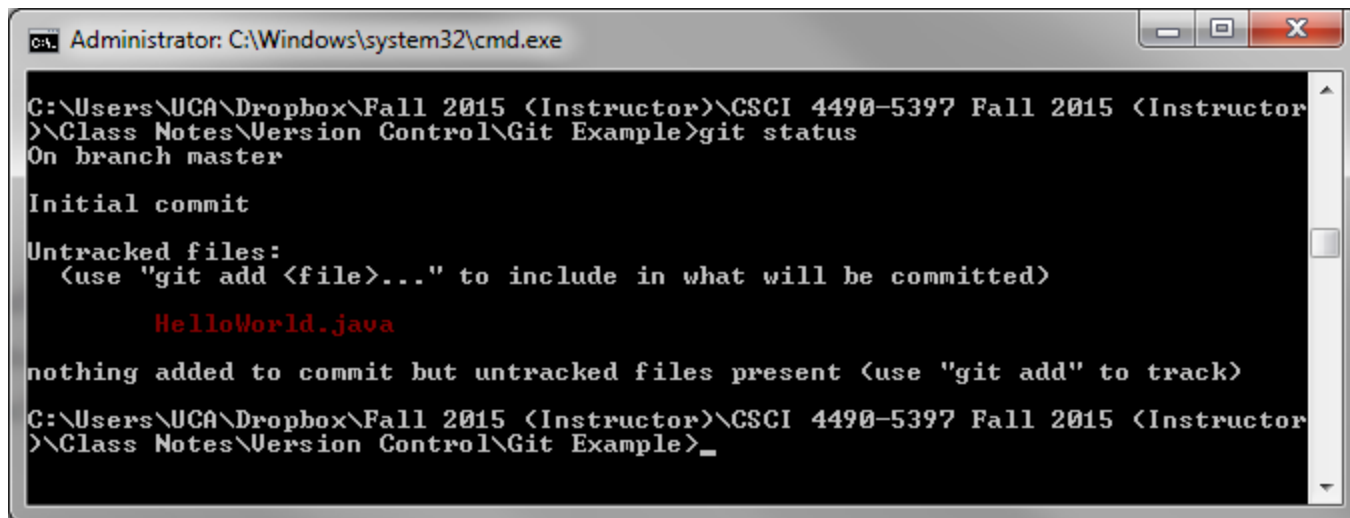
- Create a new file, called HelloWorld.java, in your project directory, and in it, add the following code:

```
// HelloWorld.java

/*
 * An application to print a welcome message.
 */
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Welcome to CSCI4490:
Software Engineering");
    }
}
```

# Adding a New File to Git (cont.)

- After saving the file, from the terminal run the command `git status`. This will show you the current status of your working repository. It should look similar to the screenshot below, with `HelloWorld.java` listed as a new, untracked file.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git status
On branch master

Initial commit

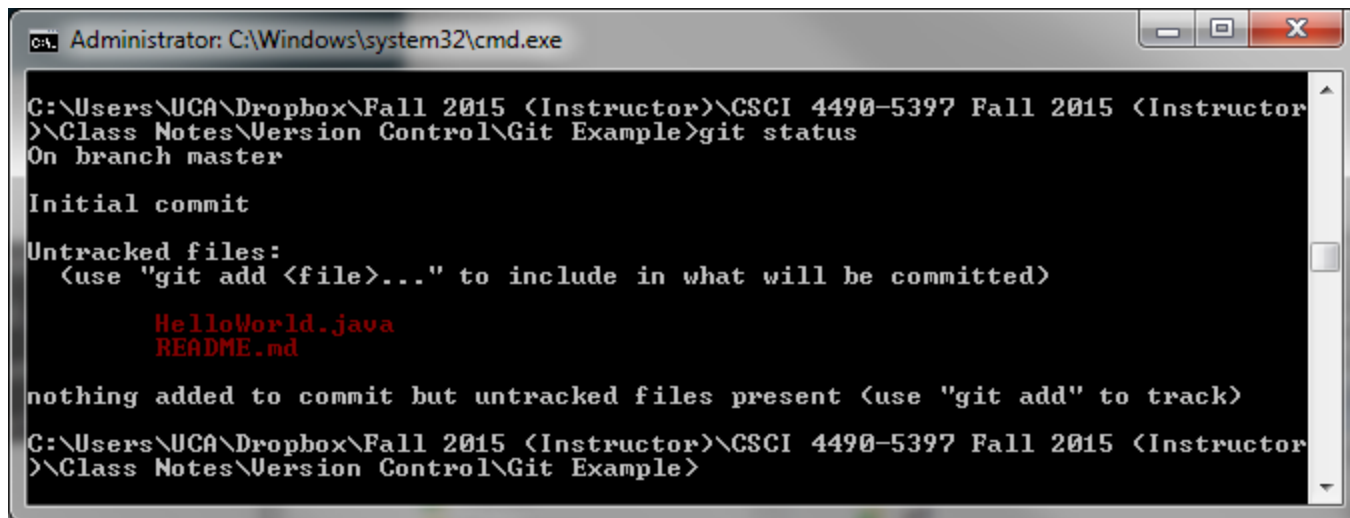
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        HelloWorld.java

nothing added to commit but untracked files present (use "git add" to track)
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>_
```

# Adding a New File to Git (cont.)

- Now let's see how you can work on multiple files, without having to commit all of them. Create a second file, called `README.md`. In that, add a few details, such as the project name, your name, and your email address. Run `git status` again, and you'll now see the two files listed as untracked, as shown below.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git status

On branch master

Initial commit

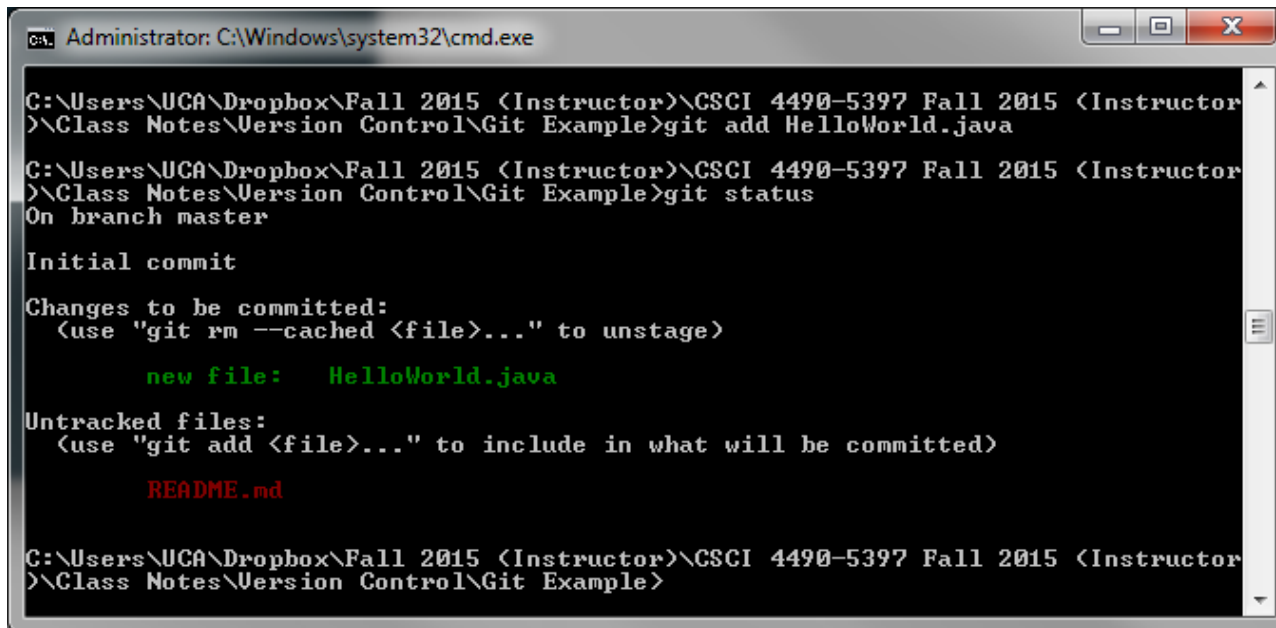
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        HelloWorld.java
        README.md

nothing added to commit but untracked files present (use "git add" to track)
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>
```

# Adding a New File to Git (cont.)

- Now let's stage HelloWorld.java, because we're not interested in README.md just for the moment. To do that, run `git add HelloWorld.java`. Now run `git status` again, and you'll see HelloWorld.java listed as a new file under "Changes to be committed," and README.md left in the "Untracked files" area.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git add HelloWorld.java

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git status
On branch master

Initial commit

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

        new file:   HelloWorld.java

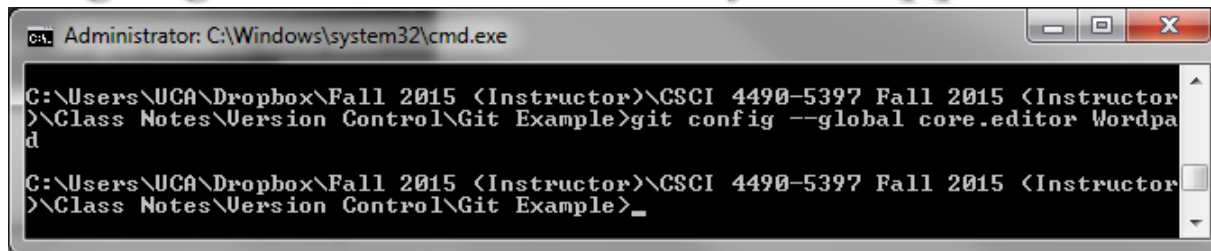
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>
```

# Updating Your Git Configuration

- Now you're ready to commit `HelloWorld.java`. But before you do, I want to show you how to configure the editor, which Git will use when you write commit messages.
- By default, Git uses the program specified in the environment variables `$VISUAL` or `$EDITOR`, which on Linux systems is normally `pico`, `vi`, `vim`, or `emacs`. If these are new to you, you might want to change it to an application you're more familiar with, perhaps `Notepad`, `TextEdit`, or `Gedit`. To do that, run the following command from your terminal:
- `git config --global core.editor <your app's name>`



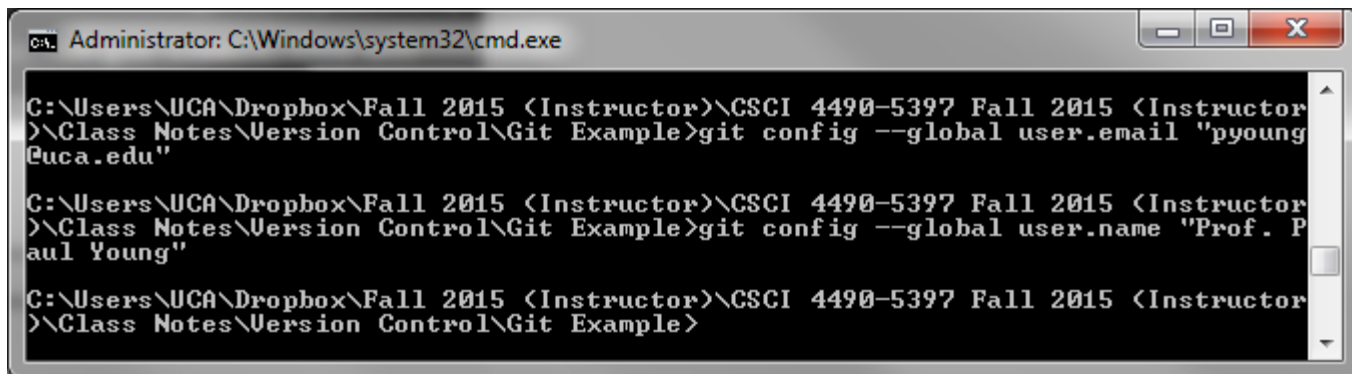
```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>git config --global core.editor Wordpad

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>_
```

# Updating Your Git Configuration (cont.)

- There are a number of other configuration changes you can make, such as your name and email address, what the commit message looks like by default, whether to use colors, and so on. For a complete list, check out the [git configuration section](#) of the Git book.



```
Administrator: C:\Windows\system32\cmd.exe

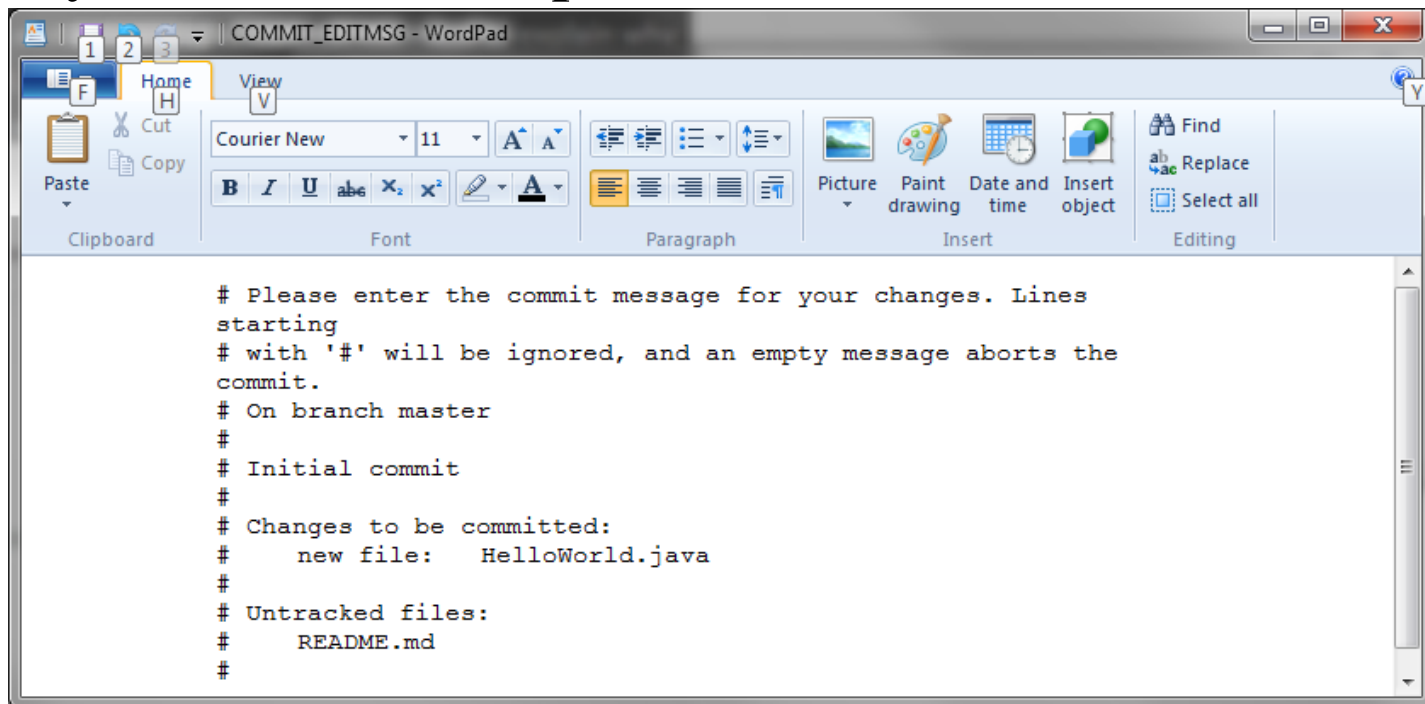
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git config --global user.email "pyoung@uca.edu"

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git config --global user.name "Prof. Paul Young"

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>
```

# Making the First Commit to Git

- Committing in Git is a lot like committing in other version control systems, such as Subversion. You start the process, add a meaningful commit message to explain why the change was made, and that's it, the file's changed. So run `git commit`. This will automatically open up your editor and display the commit template below.



# Git commit Message

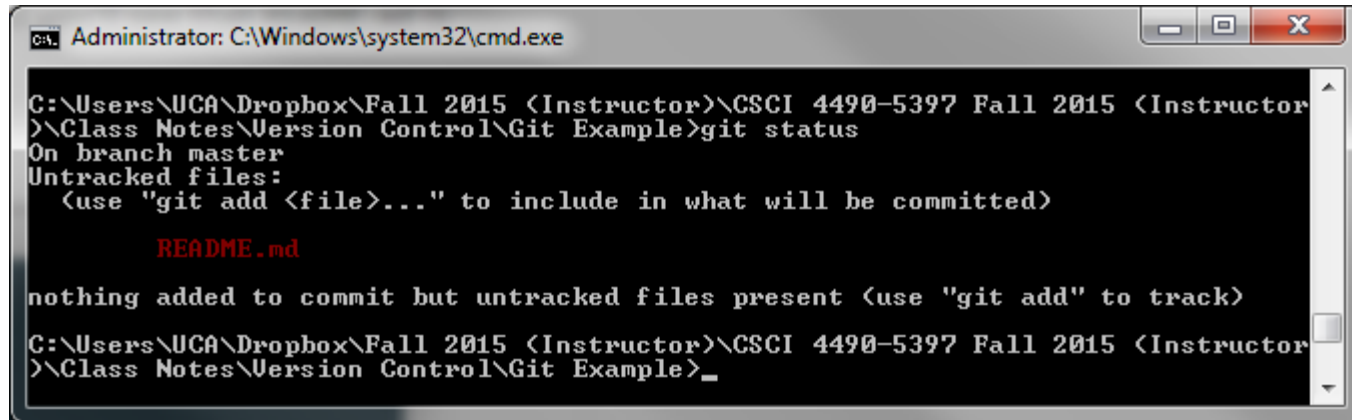
---

- As with the output of `git status`, you see the state of your working repository, which makes it easy to remember what you're committing and what you're not. A good commit message is composed of two parts: a short message, less than 72 characters long, which briefly states (in active voice) the change being made; and a much longer, optional description, which is separated from the brief description by a newline.
- In this case, there's no need to write anything too involved, as we're just adding the file to the repository. But if the change you were making involved a complex algorithm, perhaps in response to a bug filed against the code, you'd want to give your fellow developers a good understanding of why you made the change that you did. So add the following simple message "Adding the HelloWorld.java file to the repository," save it, and exit the editor.



# Making the First Commit to Git (cont.)

- Now that the file's committed, run `git status` again, and you'll see that `README.md` is still listed as untracked.



```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

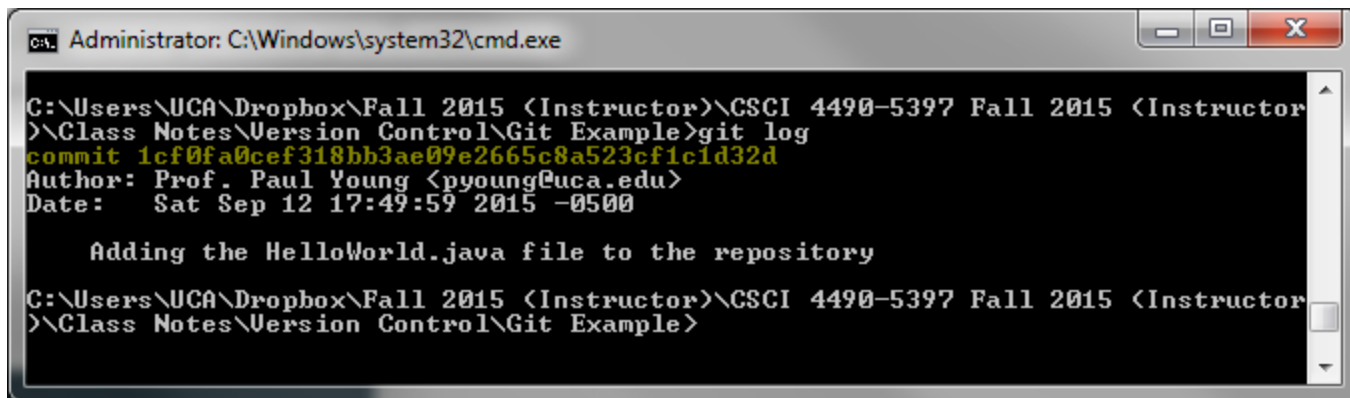
nothing added to commit but untracked files present (use "git add" to track)
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>_
```

# Seeing Differences

---

# Viewing Git Change History

- Now what if you wanted to see your repository or file history over time? To do that, you need to use [git log](#). Just running `git log` in your project repository will show you a list of changes in reverse chronological order. With no further arguments, you'll see the commit hash, the author name and email, a timestamp for the commit, and the commit message.



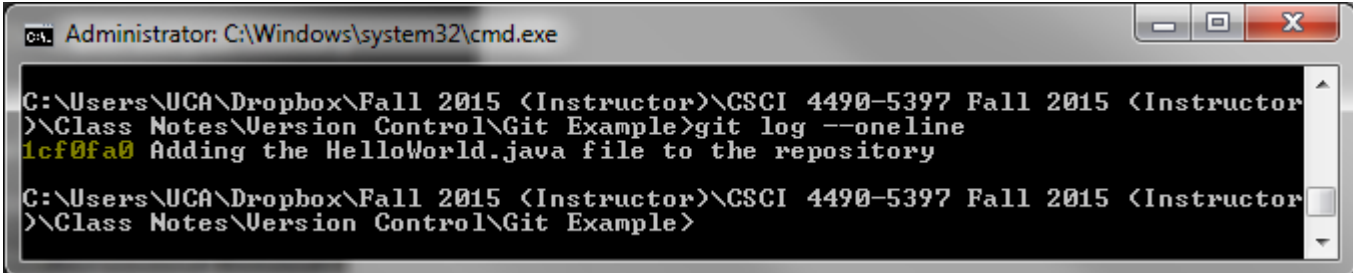
```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git log
commit 1cf0fa0cef318bb3ae09e2665c8a523cf1c1d32d
Author: Prof. Paul Young <pyoung@uca.edu>
Date: Sat Sep 12 17:49:59 2015 -0500

    Adding the HelloWorld.java file to the repository

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>
```

# Viewing Git Change History (cont.)

- Now this is fine, but what if you want to customize what you see? What if you just want to view the commit hash and the commit message? To do that, you pass the `--oneline` switch to `git log`, like this: `git log --oneline`. This will output history information, although we don't yet have enough history with our project. `--oneline` is a shortcut for `--pretty=oneline`. Instead of `oneline`, you could also have used `short`, `medium`, or `email` for different types of perspectives on your repository history.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>git log --oneline
1cf0fa0 Adding the HelloWorld.java file to the repository

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>
>\Class Notes\Version Control\Git Example>
```

# Git Branching

---

- Branches are essential for being able to safely experiment with concepts and ideas. Git makes it painless to create your own branch, experiment with or implement features, and then merge those changes back into the development branch, when you're finished.

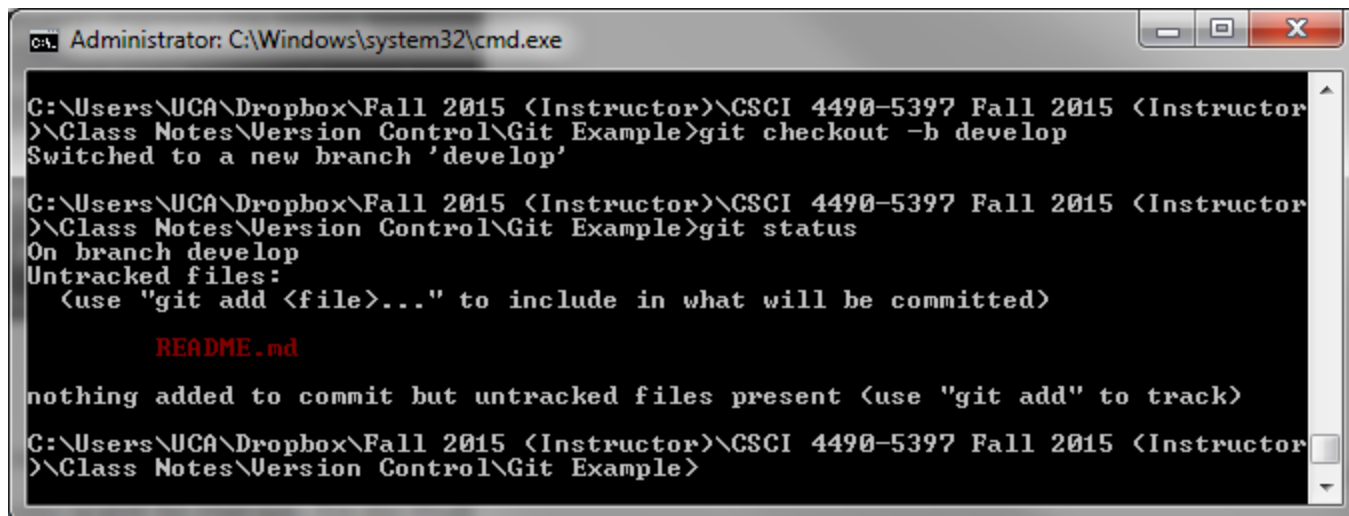
# Git Branching (cont.)

---

- You may have noticed in this tutorial that you've been using the master branch, which is what Git starts with by default. Now we'll create the development branch.
- From your terminal, run `git checkout -b develop` to create a new branch called develop.
  - Running this command will both create and check out the new branch, which at first is simply a copy of the master branch.
  - If you run `git status`, you'll still see the two separate changes to README.md. Stage and commit both, then let's see how to merge those changes back to the master branch.

# Git Branching (cont.)

- If you run `git status`, you'll still see the two separate changes to `README.md`. Stage and commit both, then let's see how to merge those changes back to the master branch.



```
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>git checkout -b develop
Switched to a new branch 'develop'

C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>git status
On branch develop
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

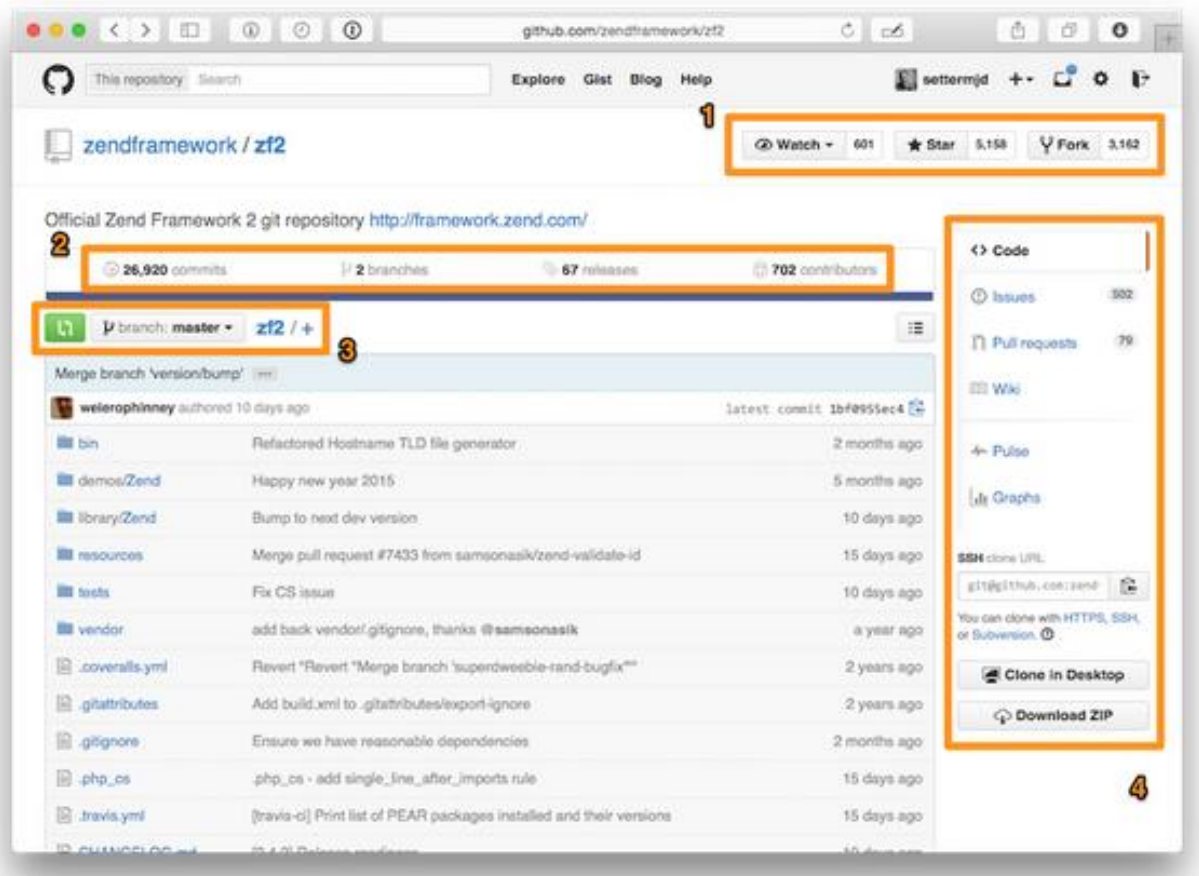
nothing added to commit but untracked files present (use "git add" to track)
C:\Users\UCA\Dropbox\Fall 2015 <Instructor>\CSCI 4490-5397 Fall 2015 <Instructor>\Class Notes\Version Control\Git Example>
```

- With those two changes staged and committed, you're ready to merge them to the master branch. First, you need to check out the branch you want to merge. To do that, run **git checkout master**. Then you need to merge the changes to the current branch from the branch you've worked on. To do that, run **git merge develop**.
- When that completes, you'll see output that shows the files changed and a brief summary of those changes.



# Using GitHub

- GitHub is more than simply a project repository, but that's where you're likely going to spend most of your time on the site.



# GitHub Project Page

---

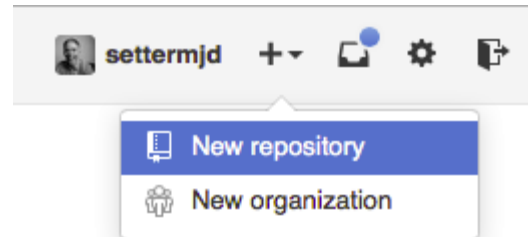
- What you see is a project homepage.
  - Across the top, in point one, are listed the project name, how many people are watching it, how many people have given it a vote of confidence by starring it, and how many people have forked it, perhaps to make changes of their own to it and contribute to it.
  - Then, in point two, there's the number of commits to the current branch, the number of branches, the number of releases, and the number of contributors.
  - Next, in point three, there's the branch picker, then below that there's a listing of the top-level files in the project, and when the last commit was.

# GitHub Project Page (cont.)

- Over on the righthand side, in point four, you have the key navigation options. These are:
  - **Code:** The view you're on by default, showing the files in the project.
  - **Issues:** A simple but effective issue tracker.
  - **Wiki:** A simple but effective wiki for documenting the project in more detail than standard README file allows.
  - **Pulse:** A summary of statistics about the project, including open and closed issues.
  - **Graphs:** A timeline of commits, followed by a breakdown of commits by individual contributor. You can then use the available tabs to look at the project activity in detail.
- Finally, also on the righthand side, there's the link to the repository URL.

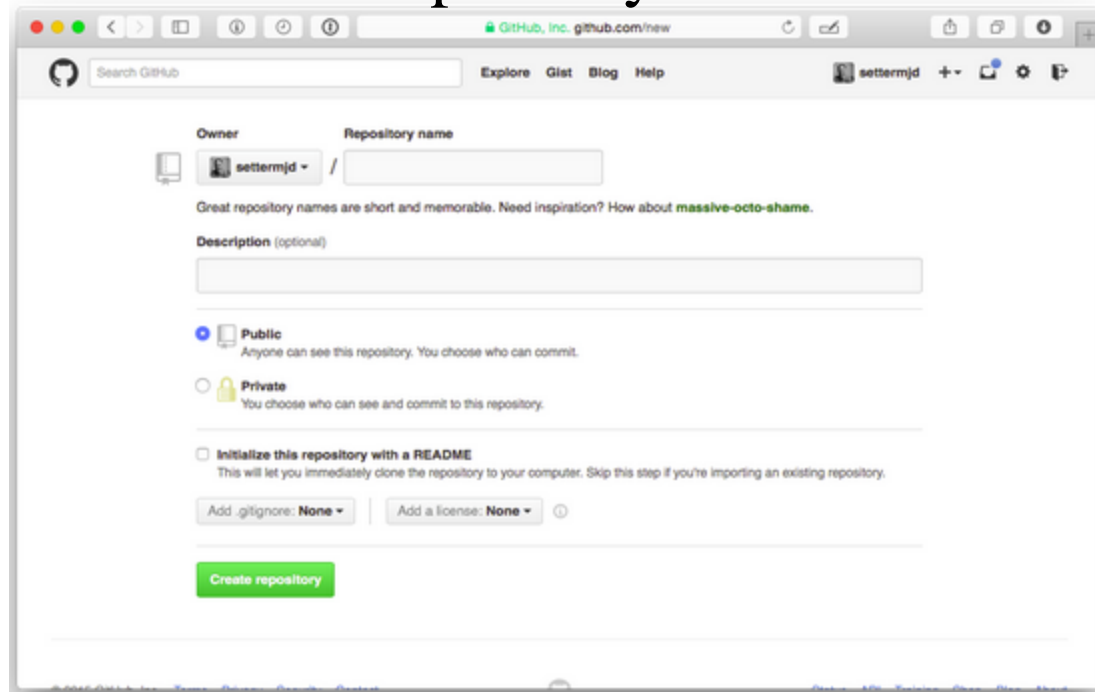
# Adding Our Project to GitHub

- After you've logged in to your account, click the plus symbol in the upper righthand corner, and click New repository from the dropdown. There you'll see the new project creation form.



# Adding Our Project to GitHub (cont.)

- In the Repository name field, add a name. Give it a description if you want. Then, leave the project with the default of public. That way anyone can find it if they search for it. Finally, click the Initialize this repository with a README checkbox, and leave the two select boxes set to None. Now click Create repository.



# Adding Our Project to GitHub (cont.)

---

- Next, the quick setup page gives you a host of post-setup information about integrating your new GitHub project with your existing local repository, which we'll do, or cloning it fresh. We're going to add GitHub as a remote for our project. To do that, copy the first line under ... or push an existing repository from the command line, and paste it in the terminal where you've been working up till now.

# Adding Our Project to GitHub (cont.)

