

Version Control with Git

BIOS/BIOI/HG 606 Day 4

Hyun Min Kang
University of Michigan

Most of the lecture material was prepared by Barry Grant who is now at UCSD

What is git?

1. An unpleasant or contemptible person. Often incompetent, annoying, senile, elderly or childish in character
2. A modern distributed version control system with an emphasis on speed and data integrity.



What is git?

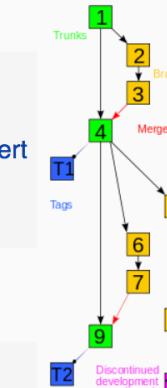
1. An unpleasant or contemptible person. Often incompetent, annoying, senile, elderly or childish in character
2. A modern **distributed version control system** with an emphasis on speed and data integrity.



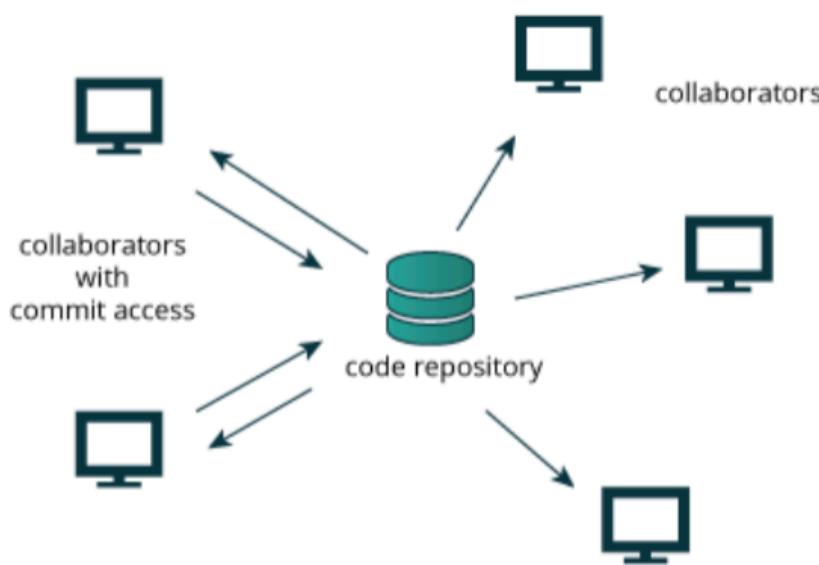
Version Control Systems

- Version control systems (VCS) record changes to a file or set of files over time so that you can recall specific versions later

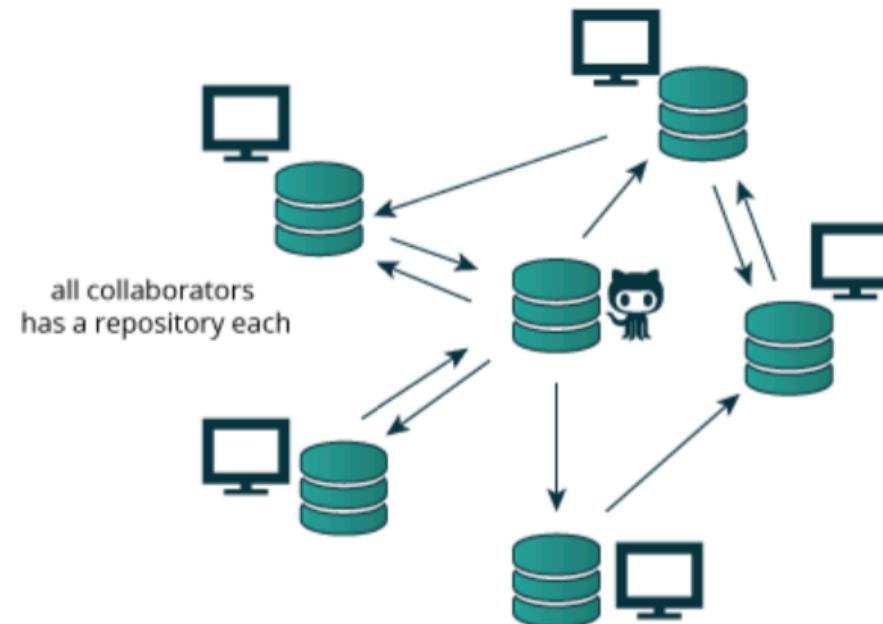
V · T · E		Version control software	[hide]
Years, where available, indicate the date of first stable release. Systems with names <i>in italics</i> are no longer maintained or have planned end-of-life dates.			
Local only	Free/open-source	RCS (1982) · SCCS (1972)	
	Proprietary	PVCS (1985) · <i>QVCS</i> (1991)	
Client–server	Free/open-source	CVS (1986, 1990 in C) · CVSNT (1998) · <i>QVCS Enterprise</i> (1998) · Subversion (2000)	
	Proprietary	AccuRev SCM (2002) · ClearCase (1992) · CMVC (1994) · Dimensions CM (1980s) · <i>DSEE</i> (1984) · Endevor (1980s) · Integrity (2001) · Panvalet (1970s) · Perforce Helix (1995) · SCLM (1980s?) · Software Change Manager (1970s) · StarTeam (1995) · Surround SCM (2002) · Synergy (1990) · Team Concert (2008) · Team Foundation Server (2005) · Visual Studio Team Services (2014) · Vault (2003) · <i>Visual SourceSafe</i> (1994)	
Distributed	Free/open-source	ArX (2003) · BitKeeper (2000) · <i>Codeville</i> (2005) · Darcs (2002) · DCVS (2002) · Fossil (2007) · <i>Git</i> (2005) · GNU arch (2001) · GNU Bazaar (2005) · Mercurial (2005) · Monotone (2003) · Veracity (2010)	
	Proprietary	<i>TeamWare</i> (1990s?) · Code Co-op (1997) · Plastic SCM (2006) · Team Foundation Server (via Git) (2013) · Visual Studio Team Services (via Git) (2014)	
Concepts	Baseline · Branch · Changeset · Commit · Data comparison · Delta compression · Fork (Gated commit) · Interleaved deltas · Merge · Monorepo · Repository · Tag · Trunk		
Category · Comparison · List			



Client-Server vs. Distributed VCS



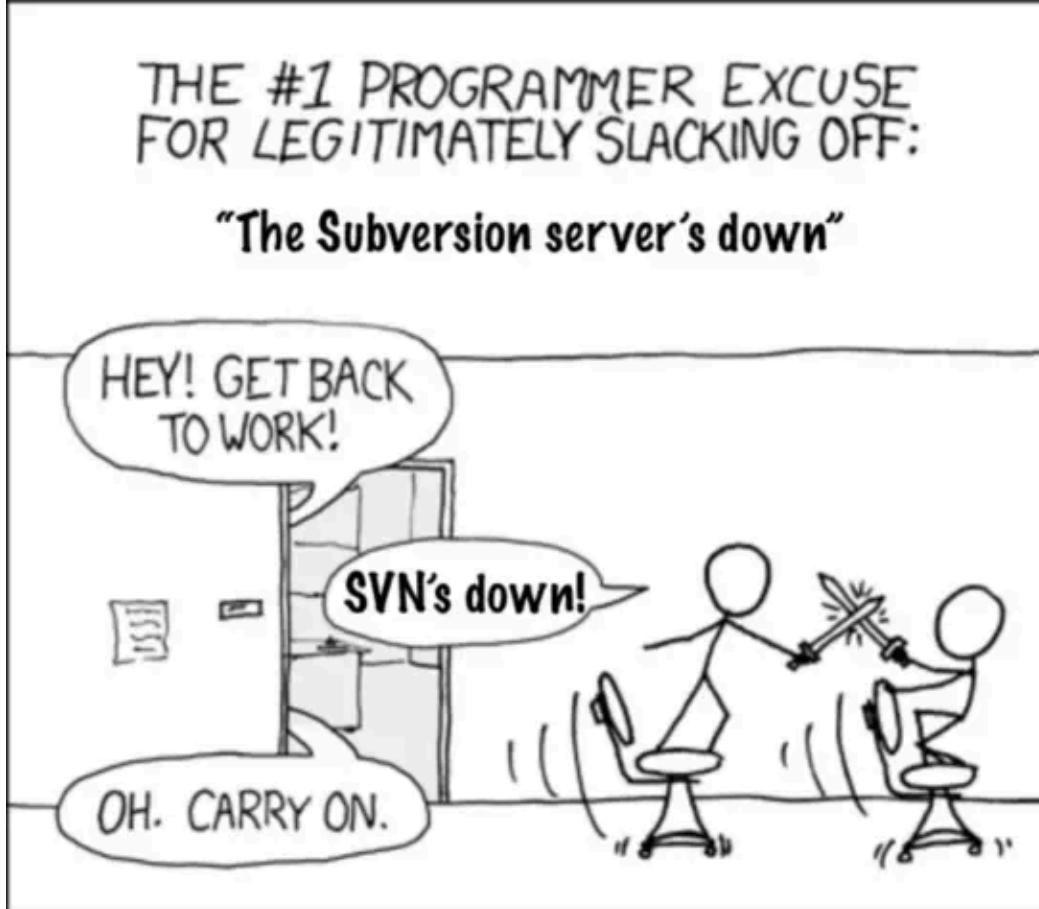
Client-server approach



Distributed approach

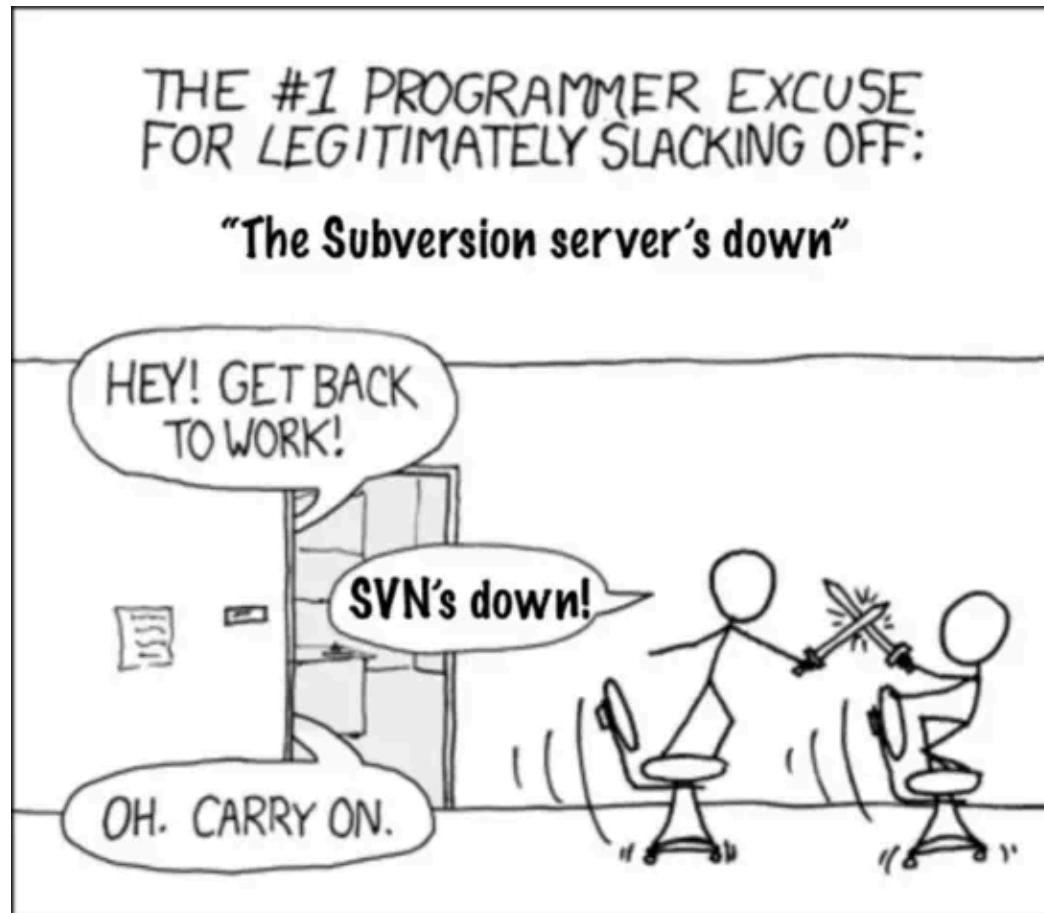
Distributed version control systems (DCVS) allows multiple people to work on a given project without requiring them to share a common network.

Subversion (**SVN**): once the most popular client-server VCS



<http://tinyurl.com/distributed-advantages>

Git is now the most popular and free VCS!

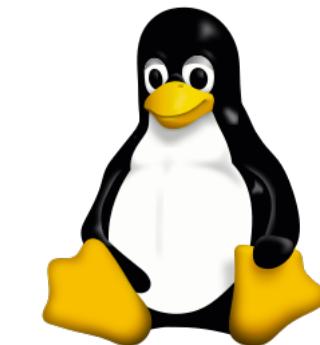


- **Git offers:**

- Speed
- Backups
- Offline access
- Small footprint
- Simplicity
- Social coding

Where did Git come from?

- Written initially by **Linus Torvalds** to support Linux kernel and OS development.
- Meant to be distributed, fast, and more natural.
- Capable of handling large projects
- Now the most popular free VCS!



Images from Wikipedia

Why use Git?

Q. Would you write your lab book in pencil, then erase and overwrite it every day with new content?

Q. Would you write your lab book in pencil, then erase and overwrite it every day with new content?

Version control is the **lab notebook** of the **digital world**: it's what professionals use to keep track of what they've done and to collaborate with others.

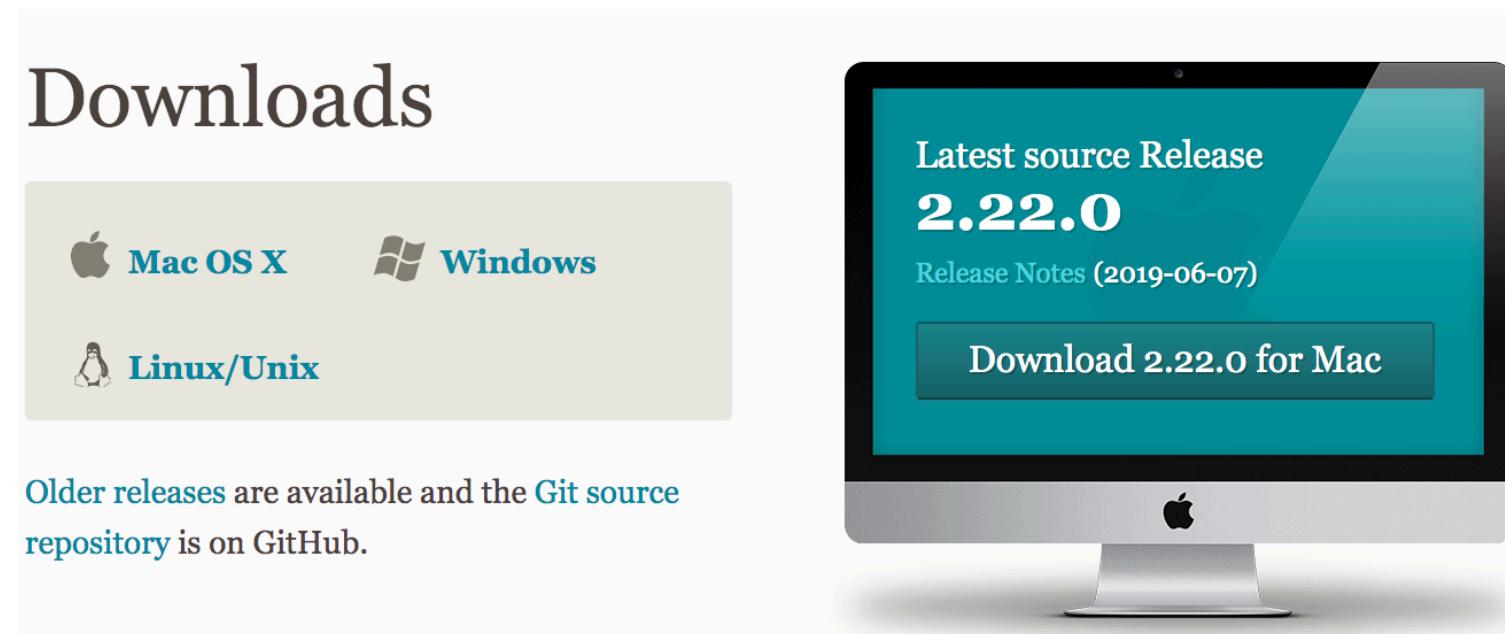
Why use Git?

- Provides ‘**snapshots**’ of your project during development and provides a full record of project **history**.
- Allows you to easily **reproduce** and **rollback** to past versions of analysis and compare differences.
(Note: Helps fix software regression bugs!)
- Keeps **track of changes** to code you use from others such as fixed bugs & new features
- Provides a mechanism for sharing, updating and collaborating (like a social network)
- Helps keep your work and software organized and available.

Obtaining Git

Obtaining Git

- Check if git is already installed by typing ‘**git**’ in your Terminal
- If absent, download and install the latest version of Git at
<https://git-scm.com/downloads>



Configuring Git

Configuring Git

- *First, tell git you you are*

```
$ git config --global user.name "Hyun Min Kang"
```

```
$ git config --global user.email "hmkang@umich.edu"
```

- Optionally, enable terminal colors

```
$ git config --global color.ui true
```

Using Git

Getting started with Git

1. **Initiate** a Git repository
2. **Edit** content (i.e. change some files)
3. Store a '**snapshot**' of the current file state



Initiate a Git repository

```
~$ cd ~/  
      # choose a directory to start from  
  
~$ mkdir git_class  
      # make a new directory  
  
~$ cd git_class  
      # change to the new directory  
  
~/git_class$ git init  
      # Our first Git command!  
  
Initialized empty Git repository in  
/Users/hmkang/git_class/.git/  
  
~/git_class$ ls -a  
      # What happened?  
.  
..  
.git
```

Side-Note: The `.git/` directory

- Git created a ‘hidden’ `.git/` directory inside your current working directory
- You can use the `‘ls -a’` command to list (i.e. see) this directory and its contents.
- This is where Git stores all its goodies – **this is Git!**
- You should not need to edit the contents of the `.git/` directory for now but do feel free to poke around.

Important Git commands

```
$ git status # report on content changes
```

```
$ git add <filename> # stage/track a file  
$ git commit -m "message" # snapshot
```

Important Git commands

```
$ git status          # report on content changes
```

```
$ git add <filename>      # stage/track a file  
$ git commit -m "message" # snapshot
```

You will use these three commands over and over in your Git workflow!

Git TRACKs your directory content

- To get a report of changes (since last commit), use:

```
$ git status
```

- You tell Git which files to track with:

```
$ git add <filename>
```

This adds files to so-called STAGING AREA
(akin to “shopping cart” before purchasing)

- You tell Git when to take a historical SNAPSHOT of your staged files (i.e. record their current state) with:

```
$ git commit -m "message"
```

Example Git workflow



Create a README text file
(this starts as untracked)



Add file to STAGING AREA
(tracked and ready to take a snapshot)



Commit changes
(record snapshot of staged files!)

Example Git workflow



Create a README text file



Add file to STAGING AREA



Commit changes



Modify README and add a ToDo text file



Add both files to STAGING AREA



Commit changes



1. Create a README file

```
[hmkang:~$ cd ~/git_class/  
[hmkang:git_class$ echo "This is the first line of text" > README  
[hmkang:git_class$ cat README  
This is the first line of text  
[hmkang:git_class$ git status  
On branch master
```

Check the file

Create a new file

Report on changes

No commits yet

Untracked files:

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

README

nothing added to commit but untracked files present (use "git add" to track)
hmkang:git_class\$



2. Add README file to ‘staging area’

```
[hmkang:git_class$ git add README
```

Add README file to the staging area

```
[hmkang:git_class$ git status
```

On branch master

No commits yet

Report on changes

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README

```
hmkang:git_class$
```

DO IT
YOURSELF



3. Commit changes

```
[hmkang:git_class$ git commit -m "Create a README file"
[master (root-commit) ed1ec4e] Create a README file
 1 file changed, 1 insertion(+)
   create mode 100644 README
[hmkang:git_class$ git status
On branch master
nothing to commit, working tree clean
hmkang:git_class$
```

Take a snapshot

Report on changes



4. Modify README and add ToDo file

```
[hmkang:git_class$ echo "This is a 2nd line of text" >> README          Add one more line to README
[hmkang:git_class$ cat README                                         Check the contents
This is the first line of text
This is a 2nd line of text
[hmkang:git_class$ echo "Learn git basics" >> ToDo                Create another file named ToDo
[hmkang:git_class$ cat ToDo                                         Check the contents
Learn git basics
[hmkang:git_class$ git status                                       Report on changes
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:   README

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

    ToDo

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



5. Add both files to ‘staging area’

```
[hmkang:git_class$ git add README ToDo
```

Add both files in one command

```
[hmkang:git_class$ git status
```

On branch master

Report on changes

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

new file: ToDo

```
hmkang:git_class$
```



6. Commit changes

```
[hmkang:git_class$ git commit -m "Add ToDo and modify README"
[master c147d0c] Add ToDo and modify README
 2 files changed, 2 insertions(+)
  create mode 100644 ToDo
[hmkang:git_class$ git status
On branch master
nothing to commit, working tree clean
hmkang:git_class$
```

Take a snapshot

Report on changes

Example Git workflow



Create a README text file



Add file to STAGING AREA



Commit changes



Modify README and add a ToDo text file



Add both files to STAGING AREA



Commit changes

... But, how do we see the history of our project changes?



git log : Timeline history of snapshots

```
[hmkang:git_class$ git log
commit c147d0c20b77aa279959772387d6ee08c1832187 (HEAD -> master)
Author: Hyun Min Kang <hmkang@umich.edu>
Date:   Fri Aug 9 06:52:33 2019 -0400
```

Add ToDo and modify README

```
commit ed1ec4e0983bd59c77cd7b3cce8b0ccfd501a726
Author: Hyun Min Kang <hmkang@umich.edu>
Date:   Fri Aug 9 06:44:54 2019 -0400
```

Create a README file

```
hmkang:git_class$
```



git log : Timeline history of snapshots

```
[hmkang:git_class$ git log
commit c147d0c20b77aa279959772387d6ee08c1832187 (HEAD -> master)
Author: Hyun Min Kang <hmkang@umich.edu>
Date:   Fri Aug 9 06:52:33 2019 -0400

    Add ToDo and modify README

commit ed1ec4e0983bd59c77cd7b3cce8b0ccfd501a726
Author: Hyun Min Kang <hmkang@umich.edu>
Date:   Fri Aug 9 06:44:54 2019 -0400

    Create a README file
hmkang:git_class$ ]
```

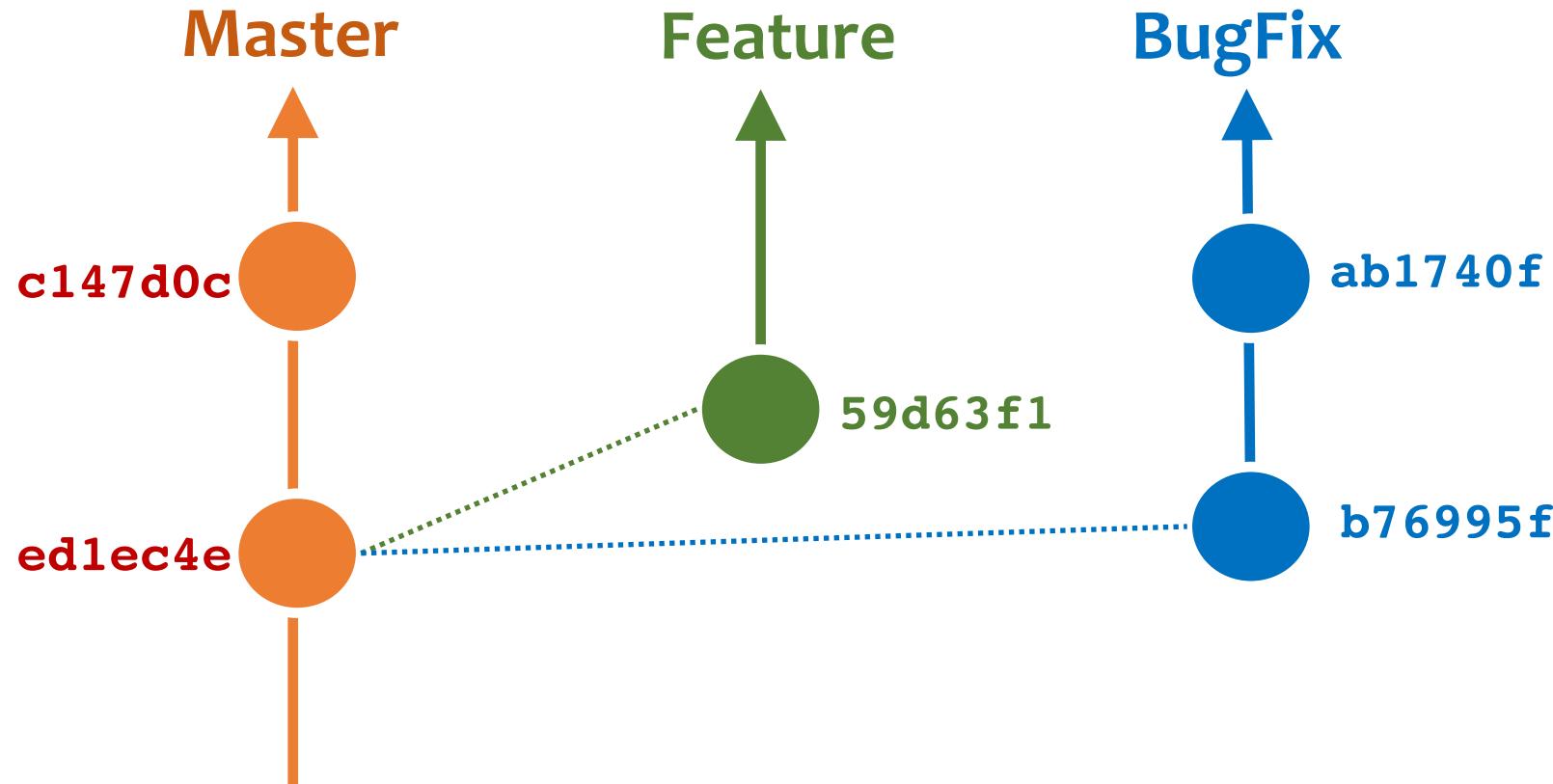


Side-note: Git history is akin to a graph



- Nodes are **commits** labeled by their unique '**commit ID**'.
(This is a **CHECKSUM** of the commits author, time, commit msg, commit content and previous commit ID).
- **HEAD** is a reference (or '**pointer**') to the currently checked out commit (typically the most recent commit).

Branching can complicate project graphs



Branches allow you to work independently of other lines of development
we will talk more about these later!

Key points

- You explicitly and iteratively tell git what files to track (“**git add**”) and snapshot (“**git commit**”).
- Git keeps an historical log (“**git log**”) of the content changes (and your comments on these changes) at each past commit.
- It is a good practice to regularly check the status of your working directory, staging arena repo (“**git status**”)

Break

Important Git commands

```
$ git status # report on content changes
```

```
$ git add <filename> # stage/track a file  
$ git commit -m "message" # snapshot
```

Summary of key Git commands

```
$ git status # Get a status report of changes since last commit
```

```
$ git add <filename> # Tell Git which files to track/stage
```

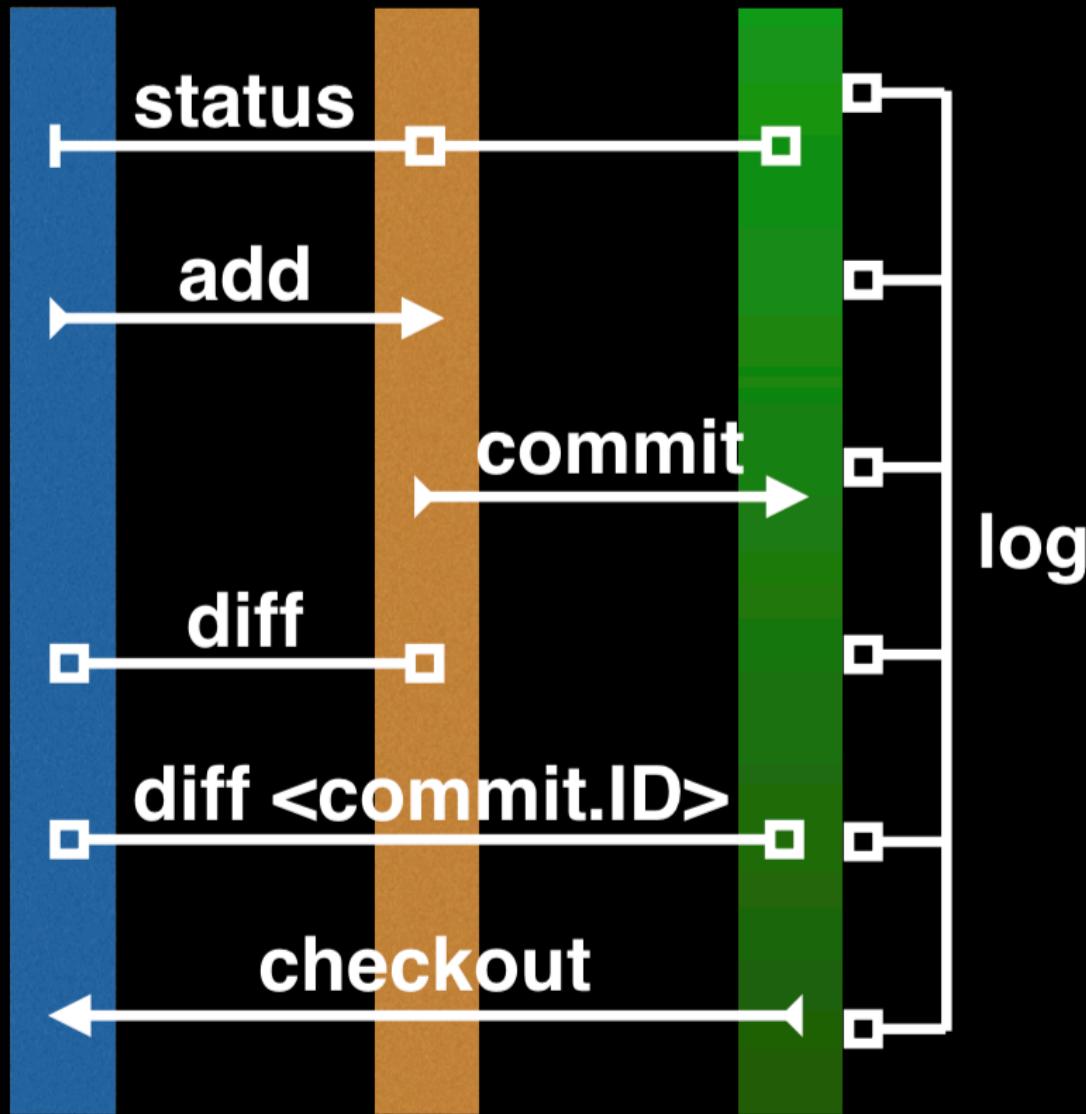
```
$ git commit -m "message" # Take a content snapshot
```

```
$ git log # Review your commit history
```

```
$ git diff <commit.ID> <commit.ID> # Inspect content differences
```

```
$ git checkout <commit.ID> # Navigate through the commit history
```

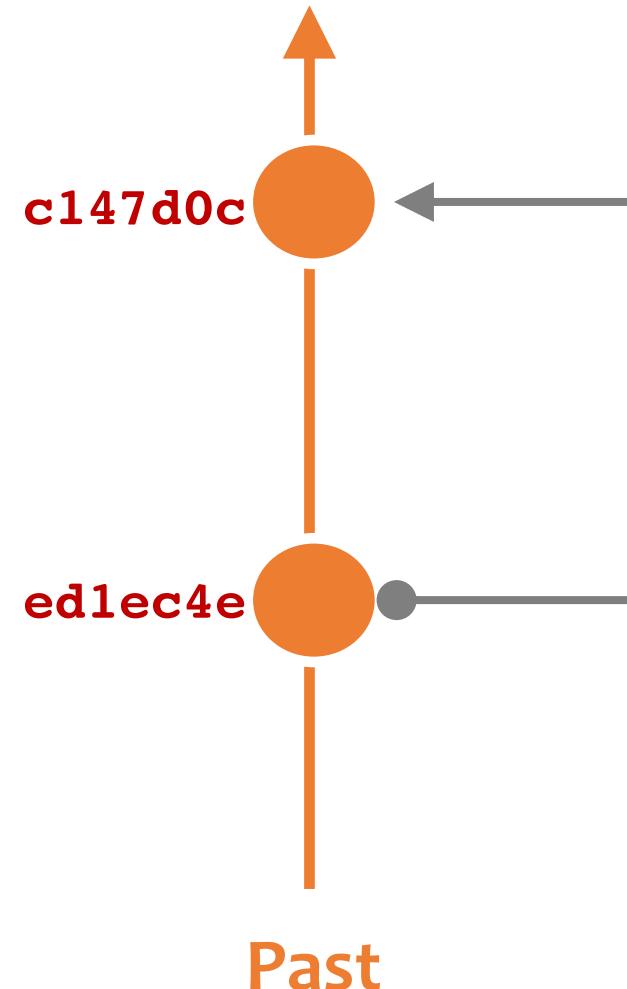
Your 'Staging Local
Directory Area' Repository





git diff: Show changes between commits

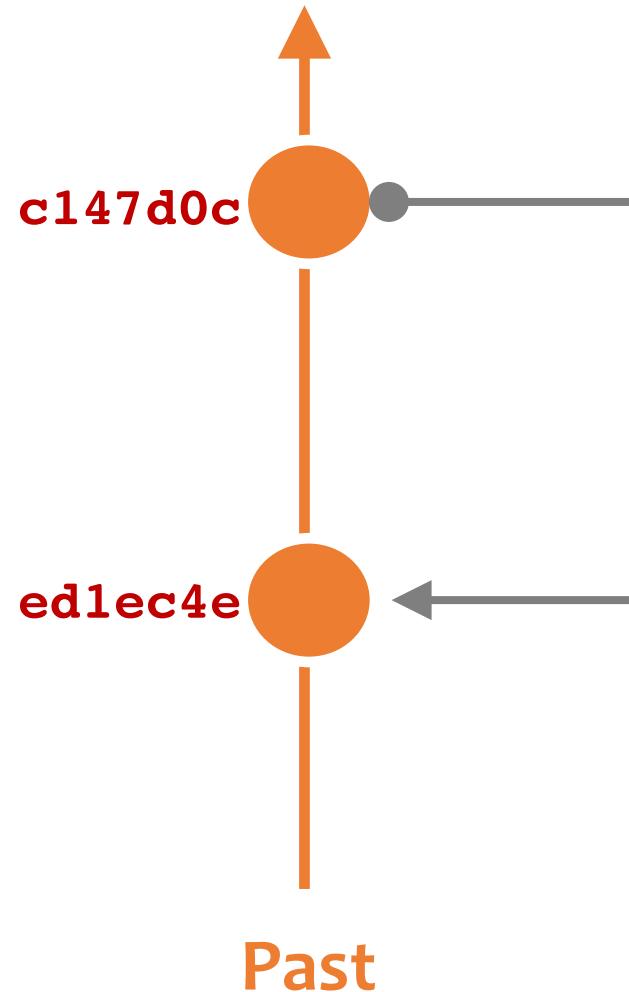
```
[hmkang:git_class$ git diff ed1e c147
diff --git a/README b/README
index 18299f9..af0ddb2 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
    This is the first line of text
+This is a 2nd line of text
diff --git a/ToDo b/ToDo
new file mode 100644
index 0000000..14fb56
--- /dev/null
+++ b/ToDo
@@ -0,0 +1 @@
+Learn git basics
hmkang:git_class$ ]
```





git diff: Show changes between commits

```
[hmkang:git_class$ git diff c147 ed1e
diff --git a/README b/README
index af0ddb2..18299f9 100644
--- a/README
+++ b/README
@@ -1,2 +1 @@
 This is the first line of text
-This is a 2nd line of text
diff --git a/ToDo b/ToDo
deleted file mode 100644
index 14fb5d6..0000000
--- a/ToDo
+++ /dev/null
@@ -1 +0,0 @@
-Learn git basics
hmkang:git_class$
```



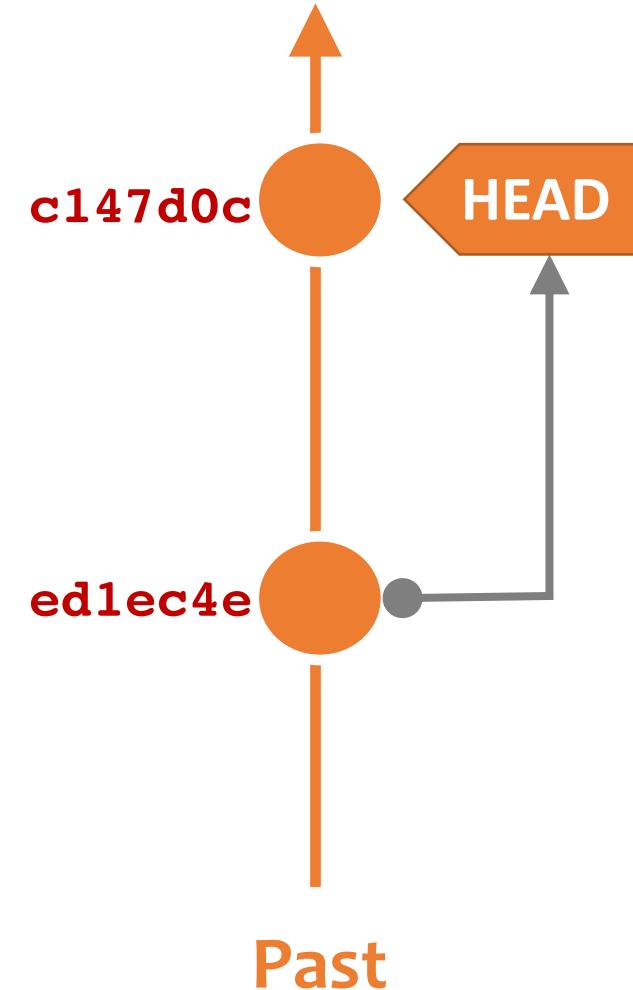


git diff: Show changes between commits

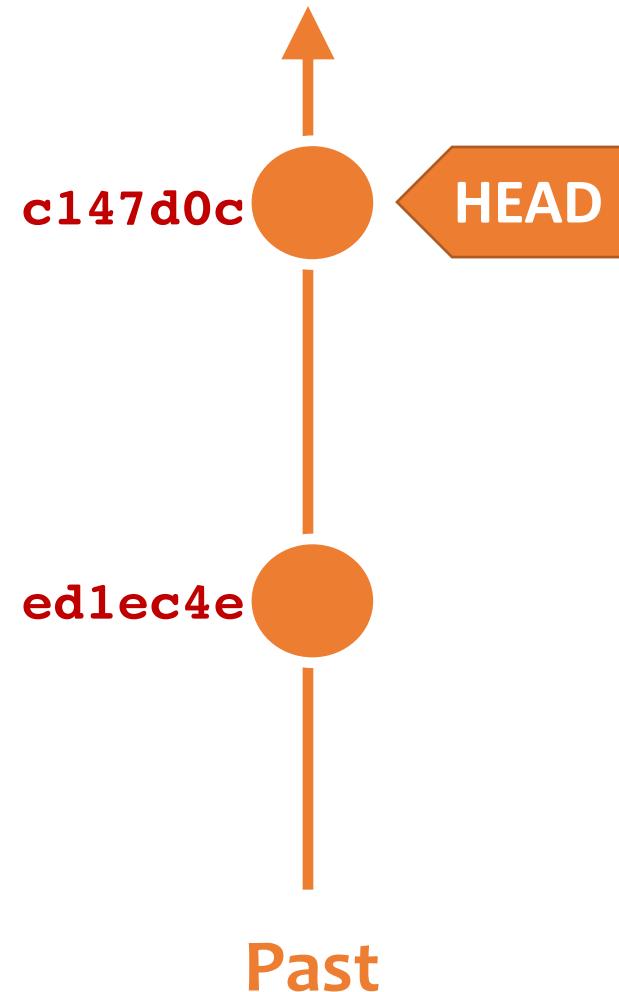
```
[hmkang:git_class$ git diff ed1e
diff --git a/README b/README
index 18299f9..af0ddb2 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
    This is the first line of text
+This is a 2nd line of text
diff --git a/ToDo b/ToDo
new file mode 100644
index 0000000..14fb0d56
--- /dev/null
+++ b/ToDo
@@ -0,0 +1 @@
+Learn git basics
hmkang:git_class$
```

Omitted second argument implies current “HEAD”

If first argument is omitted, it implies the last commit (i.e. git diff with no argument shows uncommitted changes since last commit)



HEAD advances automatically with changes

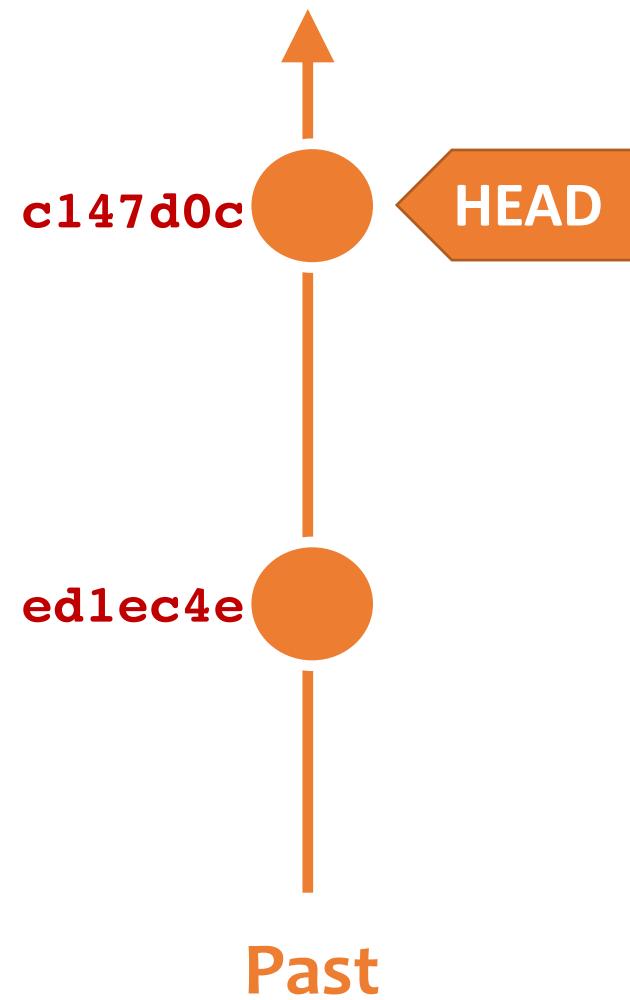


- To move **HEAD** (back or forward) on the Git graph (and retrieve the associated snapshot content) we can use the command:
 > **git checkout <commit.ID>**



git checkout : moves HEAD

```
[hmkang:git_class$ cat README
This is the first line of text
This is a 2nd line of text
[hmkang:git_class$ git log --oneline
c147d0c (HEAD -> master) Add ToDo and modify README
ed1ec4e Create a README file
hmkang:git_class$ ]
```



DO IT
YOURSELF



git checkout : moves HEAD (e.g. back in time)

```
[hmkang:git_class$ git checkout ed1e
Note: checking out 'ed1e'.
```

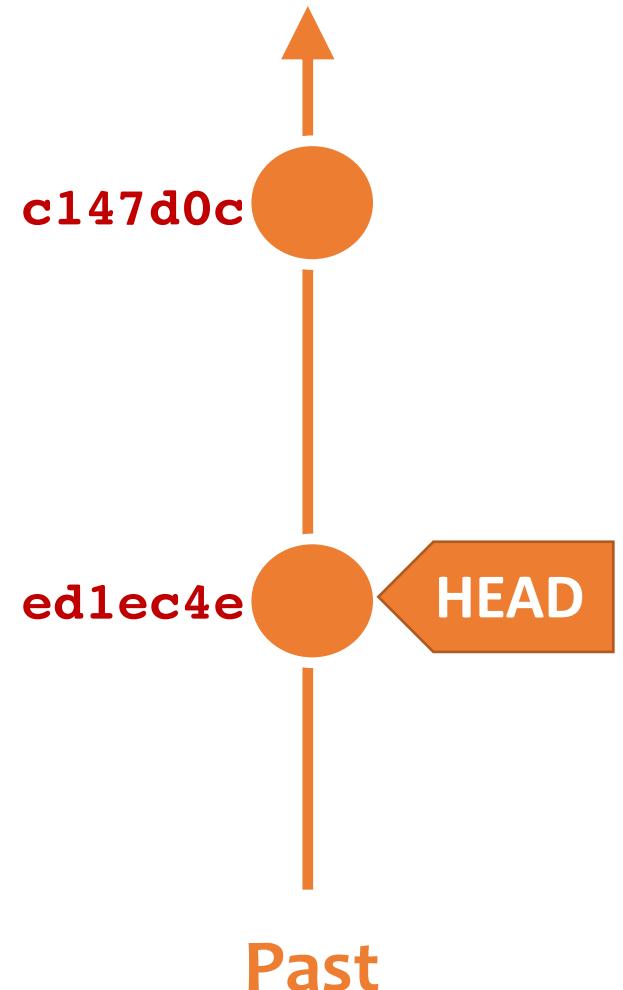
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

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 ed1ec4e Create a README file

```
[hmkang:git_class$ cat README
This is the first line of text
[hmkang:git_class$ git log --oneline
ed1ec4e (HEAD) Create a README file
hmkang:git_class$
```

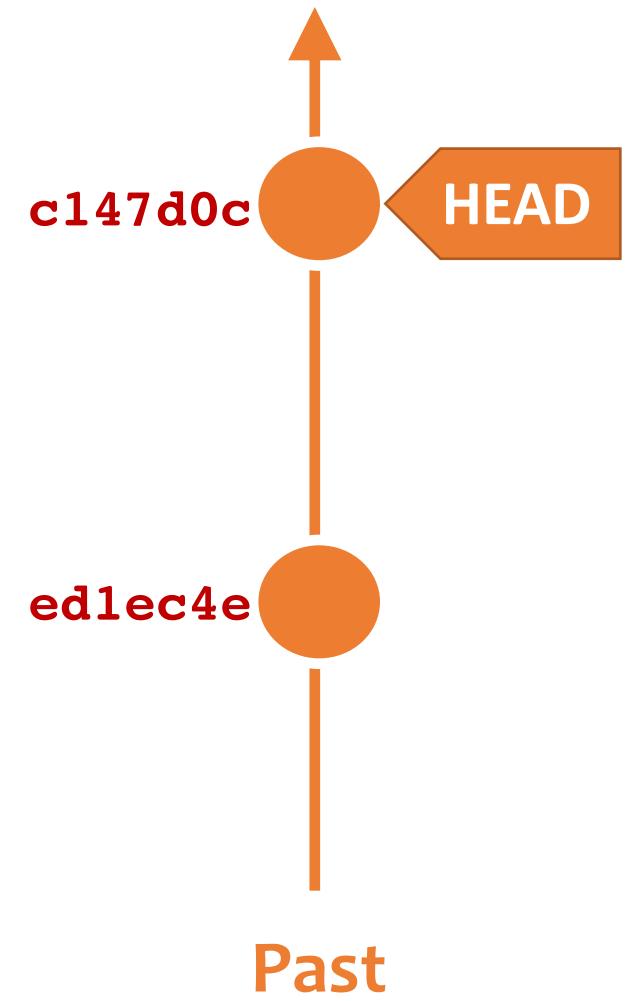


DO IT
YOURSELF



git checkout : moves HEAD (e.g. back to the future!)

```
[hmkang:git_class$ git checkout master
Previous HEAD position was ed1ec4e Create a README file
Switched to branch 'master'
[hmkang:git_class$ git log --oneline
c147d0c (HEAD -> master) Add ToDo and modify README
ed1ec4e Create a README file
[hmkang:git_class$ cat README
This is the first line of text
This is a 2nd line of text
hmkang:git_class$
```



Side-Note : two main ways to use git checkout

- Checking out a **commit** makes the entire working directory match that commit. This can be used to view an old state of your project.

> **git checkout <commit.ID>**

- Checking out a **specific file** lets you see an old version of that particular file, leaving the rest of your working directory untouched.

> **git checkout <commit.ID> <filename>**

You can discard revisions with `git revert`

- The `git revert` command undoes a committed snapshot.
- But, instead of removing the commit from the project history, it figures out how to **undo the changes** introduced by the commit and **appends a new commit** with the resulting content.
 > `git revert <commit.ID>`
- This prevents Git from losing history!

Removing untracked files with `git clean`

- The `git clean` command removes untracked files from your working directory.
- Like an ordinary `rm` command in UNIX, `git clean` is **not undoable**, so make sure you really want to delete the untracked files before you run it.

```
git clean -n # dry run display of files to be 'cleaned'  
git clean -f # remove untracked files
```

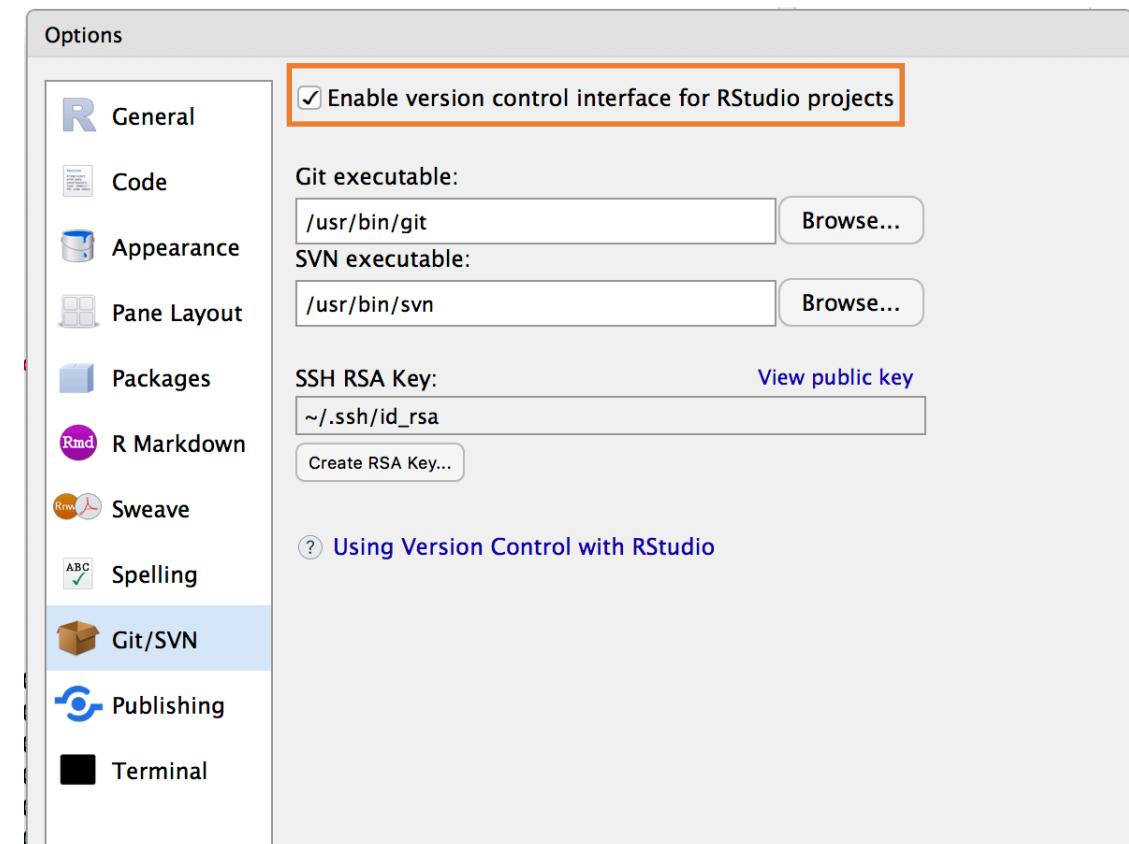
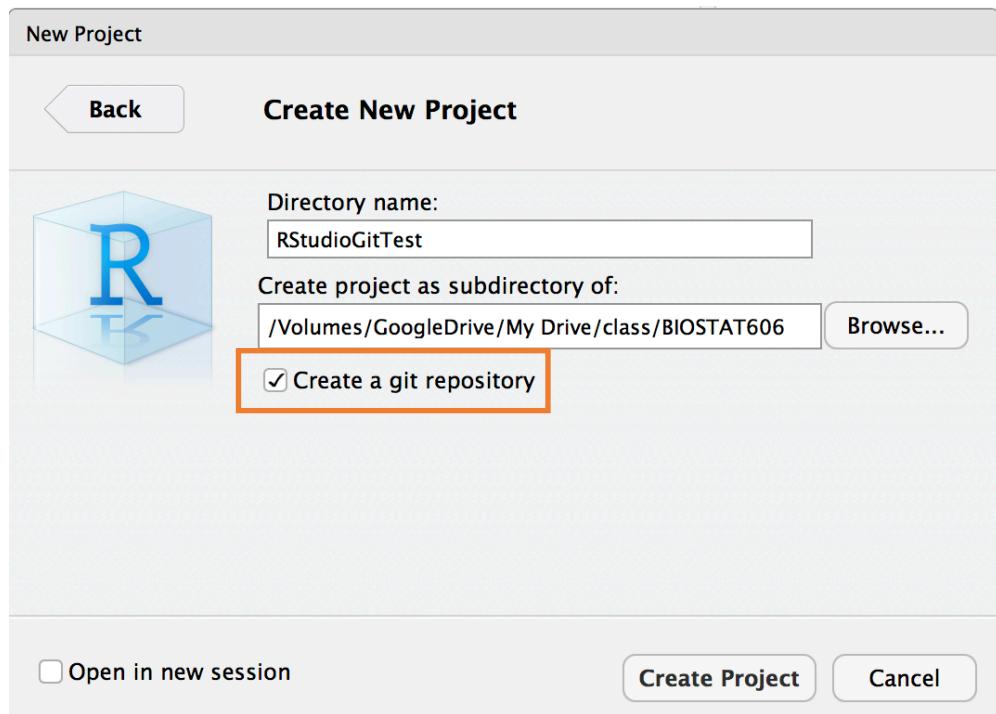
GUIs

- Tower (Mac, Windows)
- GitHub_Desktop (Mac, Windows)
- SourceTree (Mac, Windows)
- SmartGit (Linux)
- RStudio

See <https://git-scm.com/downloads/guis> more details

Side-Note : Using Git with RStudio

- Two initial steps within Rstudio
 - Tools > Global Options > Git/SVN
 - File > New Project > New Directory > Empty Project



Summary

- **Git** is a popular ‘distributed’ version control system that is lightweight and free
- Introduced **basic** git usage and encouraged you to adopt these ‘best practices’ for your future projects
- Next lecture we will cover **GitHub** and **BitBucket**, two popular hosting services for git repositories that have changed the way people contribute to open source projects

Learning Resources

- **Try Git.** Overrated hands-on git tutorial in your browser.
<https://try.github.io/levels/1/challenges/1>
- **Set up Git.** If you will be using Git mostly or entirely via GitHub, look at these how-tos.
<https://help.github.com/categories/bootcamp>
- **Getting Git Right.** Excellent Bitbucket git tutorials
<https://www.atlassian.com/git/>
- **Pro Git.** A complete, book-length guide and reference to Git, by Scott Chacon and Ben Straub
<http://git-scm.com/book/en/v2>