Source Code Management with Git[1]

Benjamin S. Skrainka

University of Chicago
The Harris School of Public Policy
skrainka@uchicago.edu

September 6, 2019

[1]The nice figures are copyright by Mark Lodato under the Creative Commons 3.0 License.

Objectives

This talk's objectives:

- State benefits of using a source code management (SCM) system
- Use core git commands to manage your workflow for both solo and collaborative projects
- Describe concepts needed to use git effectively

Plan for this Talk

Getting Started

Key Concepts

Single Developer

Distributed Development

Advanced

Benefits of Source Code Management

SCM increases your productivity by allowing you to:

- Track every version of your work (source code, LaTeX, documentation, etc.)
- Coordinate work across different computers
- Collaborate with colleagues on different computers in different locations
- Undo mistakes!
- SCM is also known as version control
- Popular tools include: git, hg (Mercurial), svn (Subversion), CVS, Perforce, ...
- You should be sued for professional negligence if you are not using SCM.

Git vs. Subversion

Git is currently the best SCM tool:

- Blazingly fast performance
- Cheap branches and merges which facilitate experimentation and collaboration
- Simple architecture (once you understand it!)
- Distributed
- Secure
- SVN is none of these things and much more painful to use

Getting Started

First Steps

Let's start by discussing:

- Installation
- Configuration
- Getting Help

Installation: Linux

To get started, install git:

- If you are root:

```
$ sudo apt-get install libcurl4-gnutls-dev \
    libexpat1-dev gettext libz-dev libssl-dev
$ sudo apt-get install git-core
```

- Otherwise,
  - Harass your system administrator
  - Download and build from source

Installation: OS/X

You have several options:

- Use git bundled with latest Xcode
- Use graphical installer:
  http://code.google.com/p/git-osx-installer
- Install your own copy:

```
$ sudo port -v install git-core +svn +doc \
                    +bash_completion +gitweb

Options besides git-core are optional...
```

- Get GitX, SourceTree, etc. to help visualize

Installation: Windows

On Windows,

- Install Git GUI:
  - http://code.google.com/p/msysgit
  - Includes SSH client
- You can probably also use the cygwin package manager...

Visualizer

Install a visualizer:

- Will help you see how commands affect directed graph of changes to your code
- Will help you learn git:
  - View state of repository
  - Predict how it will change
  - Apply git command
  - Check your understanding...
- Some popular visualizers: gitk, SourceTree, GitBox, GitHub, ...

Configuration

Git is highly configurable:

- Configuration is stored in ~/.gitconfig
- Set the options using git:

```
$ git config --global user.name “Dr. Billy Horrible”
$ git config --global user.email “drh@horrible.com”
$ git config --global core.editor vim
$ git config color.ui true
```

- You can edit ~/.gitconfig to set options directly once you know what you are doing. . .

Help

There are many resources for help:

- Git provides three equivalent ways to get help:
  - man git-***command***
  - git help ***command***
  - git ***command*** *–help*
  - To learn about git init, the following all work:

```
man git-init
git help init
git init --help
```

- ProGit: an excellent free-ish book
- Other links
  - Git Reference
  - git - The Simple Guide
  - Think Like (a) Git
  - Visual Git

Key Concepts

Create a Repository

To create a repository (aka *history*) for your work:

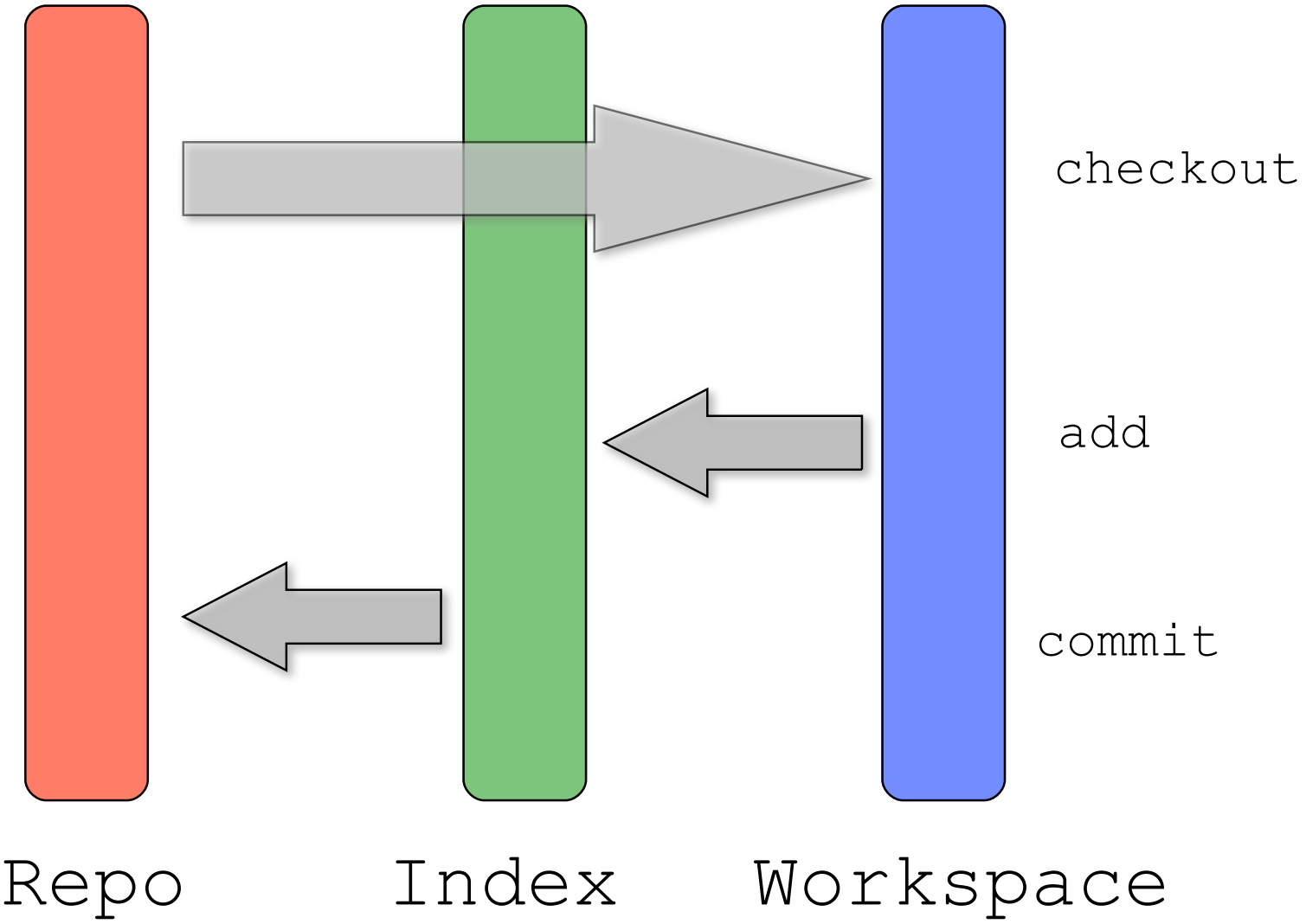
```
$ mkdir orbital   # Create a workspace
$ cd orbital
$ git init
Initialized empty Git repository in \
  /home/horrible/sbox/orbital/.git/
```

- This initializes git's internal data structures to track your work
- Git's data is stored in .git:

```
$ ls -a
.git
$ ls .git
HEAD        config        hooks/        objects/
branches/   description   info/         refs/
```

- Edit `.git/description` to describe your project (optional)

Git Workflow



Commited changes (a.k.a. the repository) are referred to as the *History*

Git Internals: Objects

Git's internal structure is very simple and manages only three types of objects:

- blob: a file
- tree: a directory
- commit: a group of blobs and/or trees
  - commit objects are stored via a directed graph (which is not acyclic)
  - Newer commits are children of parent commits
  - A commit with multiple parents is a merge
- ... plus labels for branches

SHA-1 Hash

Commit's ID = hash of its contents + parent

- A very long number like: 833da57ebd73e6f5c35f1753569b112df5364344
- Is cryptographically secure and nigh unique
- Provides a unique ID of a place on the graph of commits
  $\Rightarrow$ Two repos with same IDs are identical from that point backwards in history, even if repos are on different machines!
  - Shortcut: can use 4+ leading characters of hash as an ID
  - Commands show ~7 characters or full hash as ID as needed
- Power, flexibility, and security!
- Can also use HEAD, HEAD^, HEAD^^, HEAD~*n*

Git Internals: SCM Operations

SCM operations manipulate these objects by:

- (Re)labeling commits to handle branching and tagging
- Creating new commits, if necessary, to handle merges
- Moving blobs and trees between:
  - your workspace
  - the staging area (aka the index aka the cache)
  - the history (aka repo)
- Makes git very fast and powerful!

Index File

The Index files sits between your work and the repository:

- Naming of the Index is not standardized:
  - The Index == The Staging Area == The Cache
  - Кто виноват и что делать? ('Who is guilty and what should we do about it?')
- Provides flexibility:
  - Index contains list of files to be committed
  - Allows you to work on several different things at once and commit just subsets of changes
  - In order to commit a change, must add the files or directories to index. Also known as 'staging.'

Single Developer

Basic SCM Operations

Now that you have setup your repository with `git init`, you are ready to get to work:

- Stage changes for commit
- Commit changes
- Branch
- Merge
- Inspect your changes
- Revert changes

Staging Your Work: git add

`git add` registers your file with the index:

```
$ vi README.txt
$ git add README.txt
```

- Work is now staged for committing
- `git add` does more than just adding files – it is the interface to the staging area!
- `git add` *`dir`* will recursively add *`dir`* and its contents.
- Actually copies your work to the repository, but it is unlabeled...

Examining Workspace State: git status

`git status` shows the state of your workspace – i.e. what has been modified and/or staged:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   README.txt
#
```

`git status -s` provides a more concise report

Saving Staged Work: git commit

`git commit` stores the objects which are in the index as a commit:

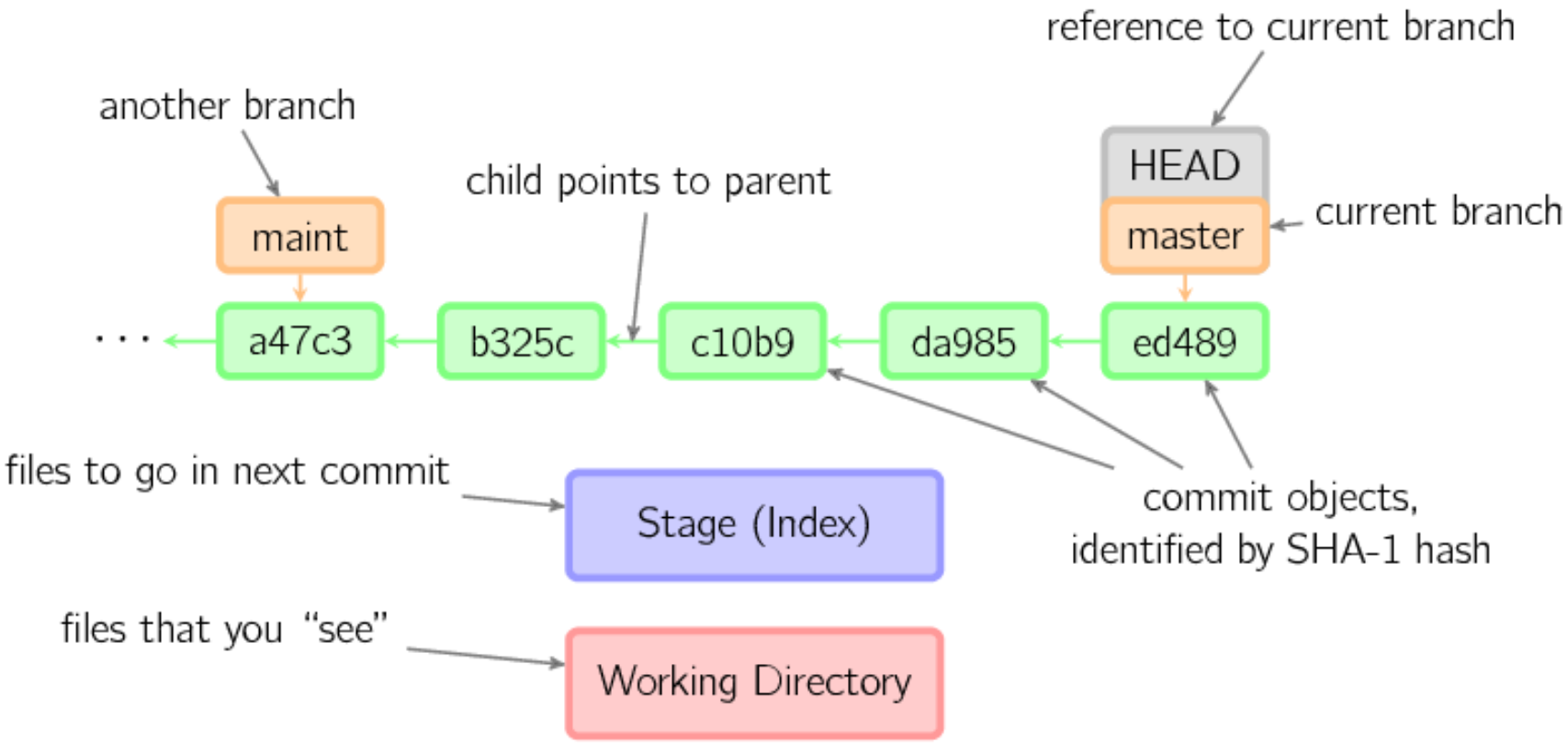
- Each commit has a unique SHA-1 ID
- Should specify a message saying why you did what you did

```
$ git commit -m "Created documentation."
[master (root-commit) 8cddf03] Created documentation.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README.txt
```

- `git commit --amend` adds staged work to previous commit
- `git commit -a` will add and commit modified files
- Only commit objects which are not easily recreated

$\Rightarrow$ commit often to increase granularity of versions

git commit


reference to current branch
another branch
HEAD
child points to parent
maint
master
current branch
· · ·
a47c3
b325c
c10b9
da985
ed489
files to go in next commit
Stage (Index)
commit objects,
identified by SHA-1 hash
files that you "see"
Working Directory

The Commit Message

Provide meaningful comments about why the changes you made:

- Say why you did what you did (git diff will always show what you did)
- Best practice:
  - Do not specify a short message with -m option
  - Then, git will start up your favorite editor so you can write a pithy comment
  - By convention, the comment should be structured as:

```
This is a one line summary of my brilliant changes.

After the blank line, I provide a detailed
explanation of my genius.  Often it is helpful
to include bullet points using *'s or -'s:

* Helps make reason for change clear
* Is nice for your future self and collaborators
```

View History: git log

`git log` displays information about commits (messages, hashes, and committer):

```
$ git log
commit 8cddf03e1a9f62c1f39dc4b47eb40a3bef6080e4
Author: Dr. Billy Horrible <drh@horrible.com>
Date:   Tue Apr 3 14:05:17 2012 -0500
    Created documentation.
```

- There are many options:

```
$ git log -5
$ git log --pretty=oneline -5
$ git log --pretty=format:"%h %ad | %s%d [%an]" \
          --graph --date=short
```

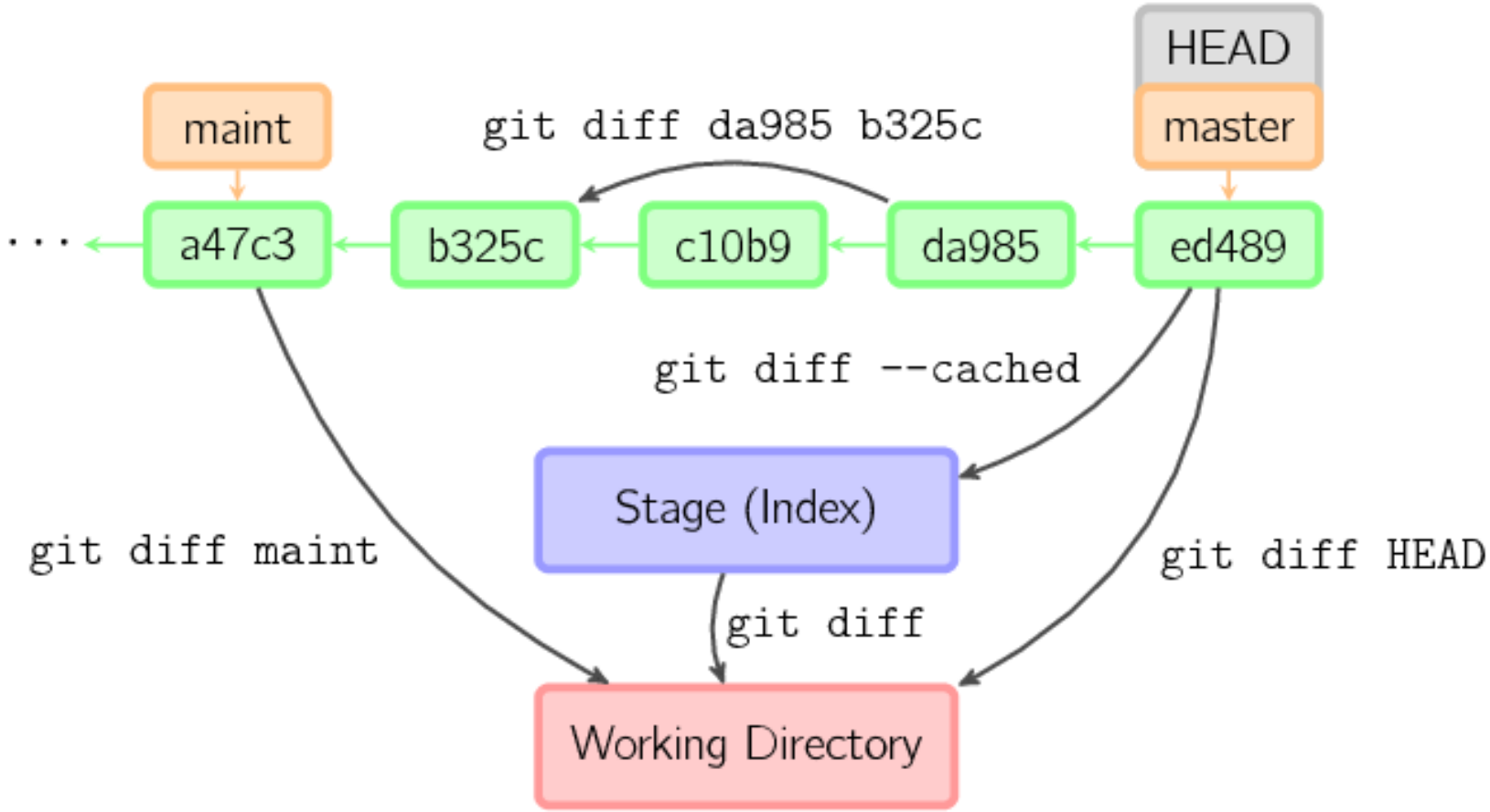
Compare Versions: git diff

You can compare different versions of files as well as different commits with `git commit`:

- `git diff`: compares index vs. working directory
- `git diff --stat`: overview of differences
- `git diff --cached` or `git diff --staged`: compares index vs. HEAD
- `git diff HEAD`: compares working directory with history
- `git diff da985 b325c`: compares two commits
- `git diff master...test`: computes common ancester of the two branches and then shows how test has changed. This is equvalent to:

```
$ git diff --stat $(git merge-base master test) test
```

git diff


HEAD
maint
git diff da985 b325c
master
...
a47c3
b325c
c10b9
da985
ed489
git diff --cached
Stage (Index)
git diff maint
git diff HEAD
git diff
Working Directory

Branching: git branch

git branch:

- Create 'save points'
  - Can experiment safely in a branch...
  - Manage several versions of software: for different projects or feature sets
  - Load work from other repositories into a branch to check before merging

Git is incredibly powerful and flexible because branches and merges are cheap: only 40 char needed to branch!

- This is a PITA with all other SCM tools
- Behind the scenes, git just shuffles labels on the commits which are a directed graph (not acyclic)

git branch + git checkout

Use `git branch` to display or create branches:

```
$ git branch            # shows branches
  maint
* master
$ git branch genius    # create branch
$ git branch -d maint # delete branch
```

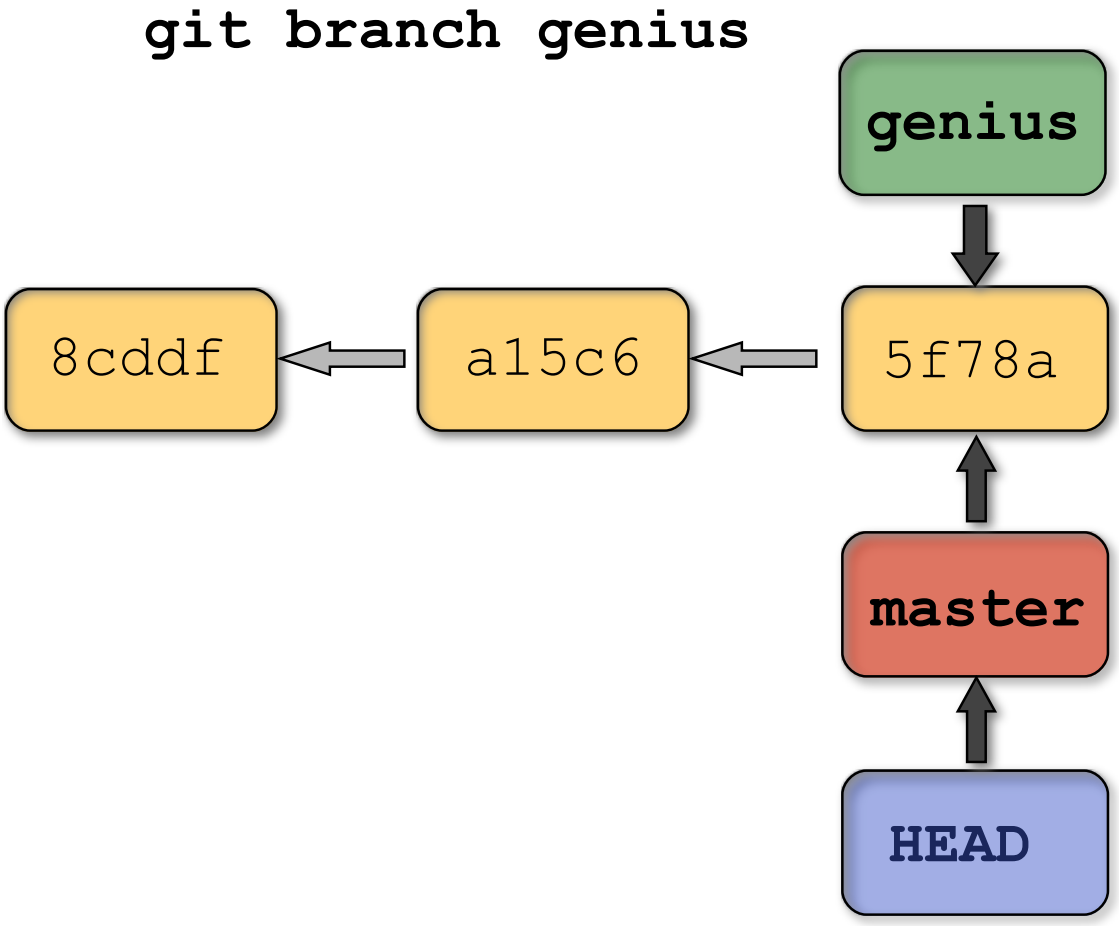
Use `git checkout` to switch branches:

```
$ git checkout genius  # switch to new branch
```

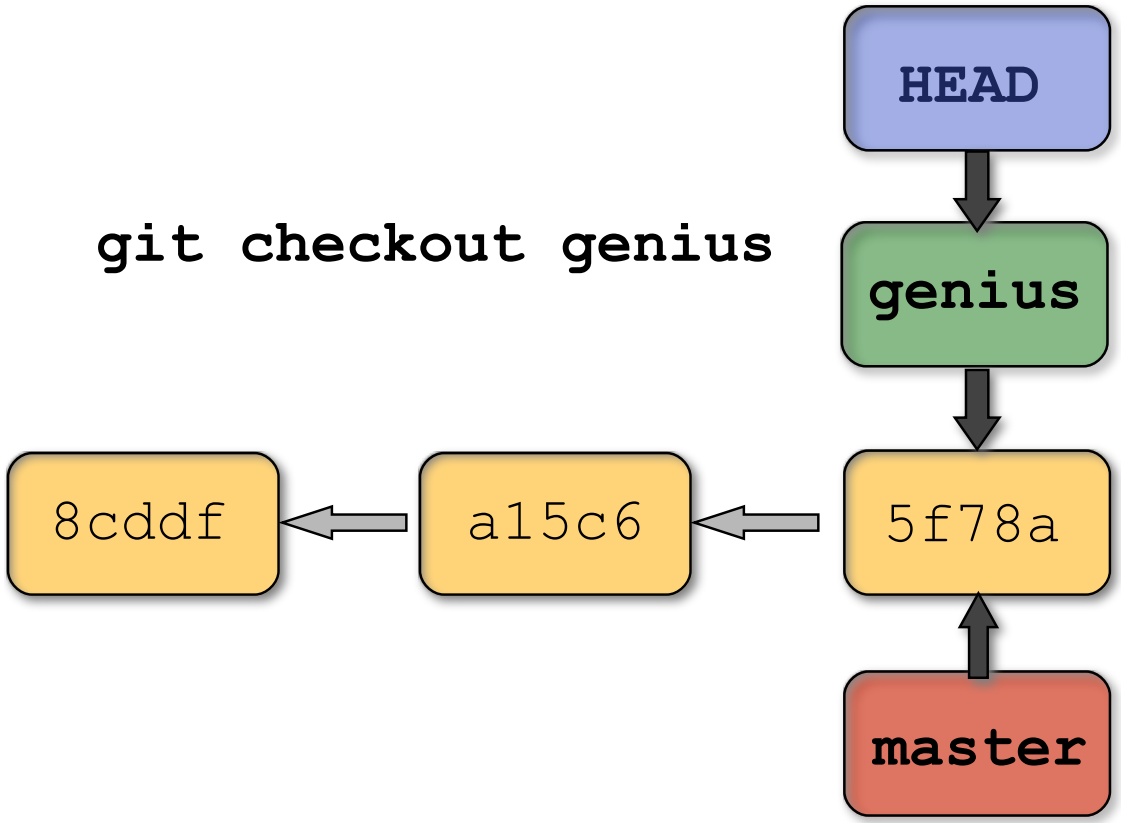
- Now commits affect genius branch
- Git copies appropriate version of files into workspace!
- Can create and switch to a branch with

```
$ git checkout -b genius # create & switch to test
```

git branch



git checkout



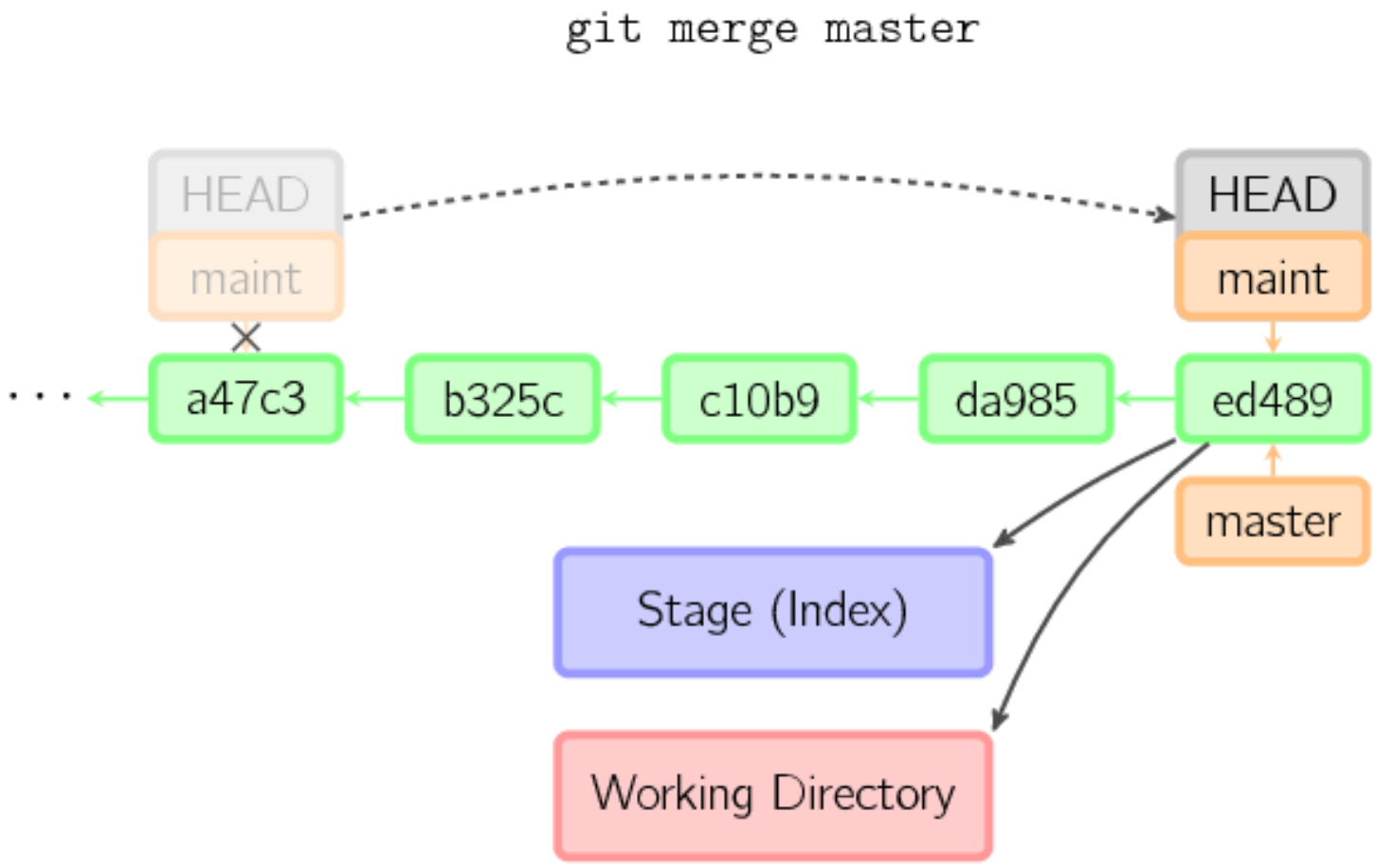
Merging Branches

`git merge` *`branchID`*:

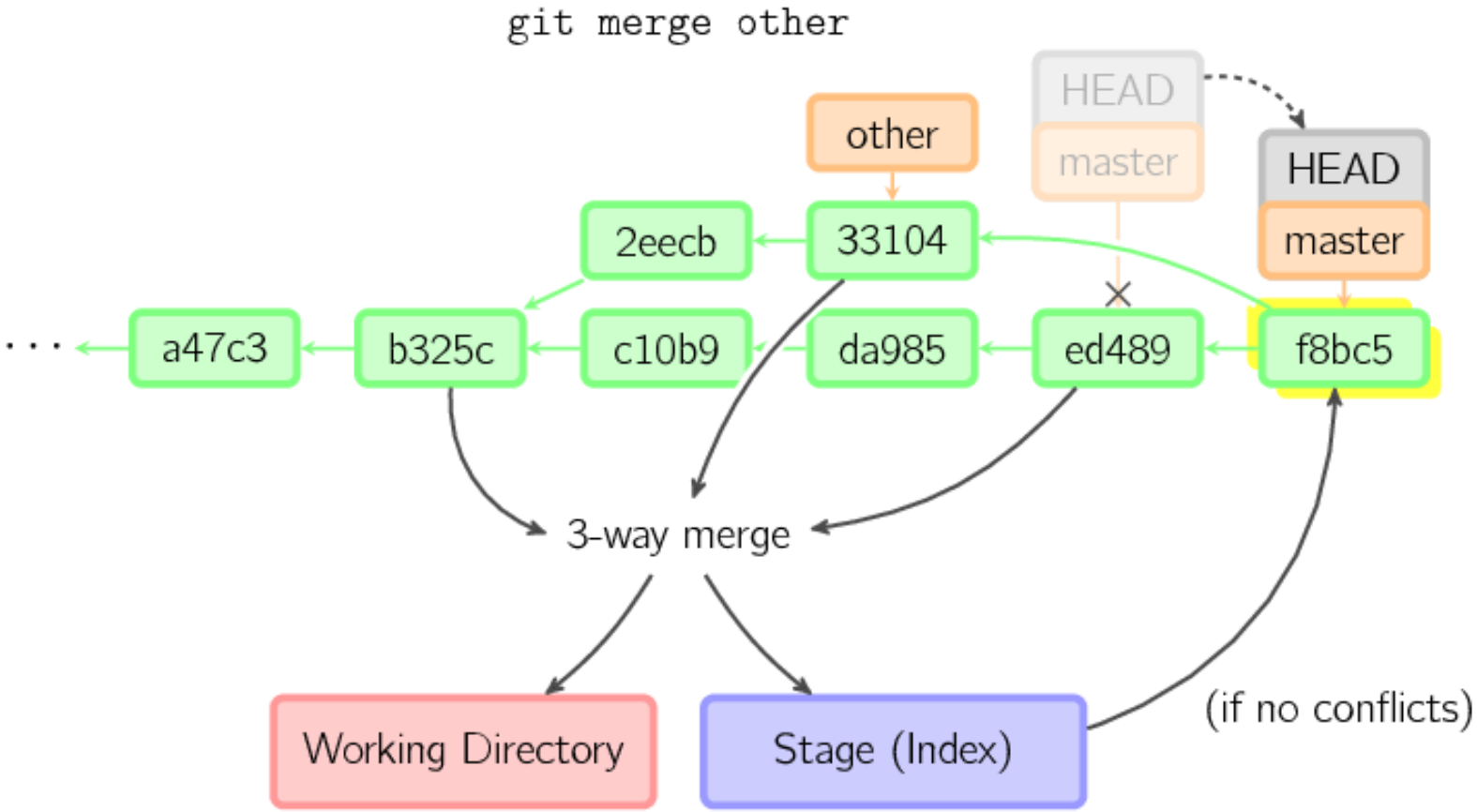
- Fast forward: just move label from ancestor/parent commit to child
- Merge: creates a new commit which contains all the changes which have occurred in both branches to the nearest common ancestor commit
  - Changes are merged automatically
  - . . . unless changes conflict. Then you will need to resolve them manually

```
$ git checkout master
$ git merge maint
$ git merge c24e74
```

Fast Forward Merge



Merge

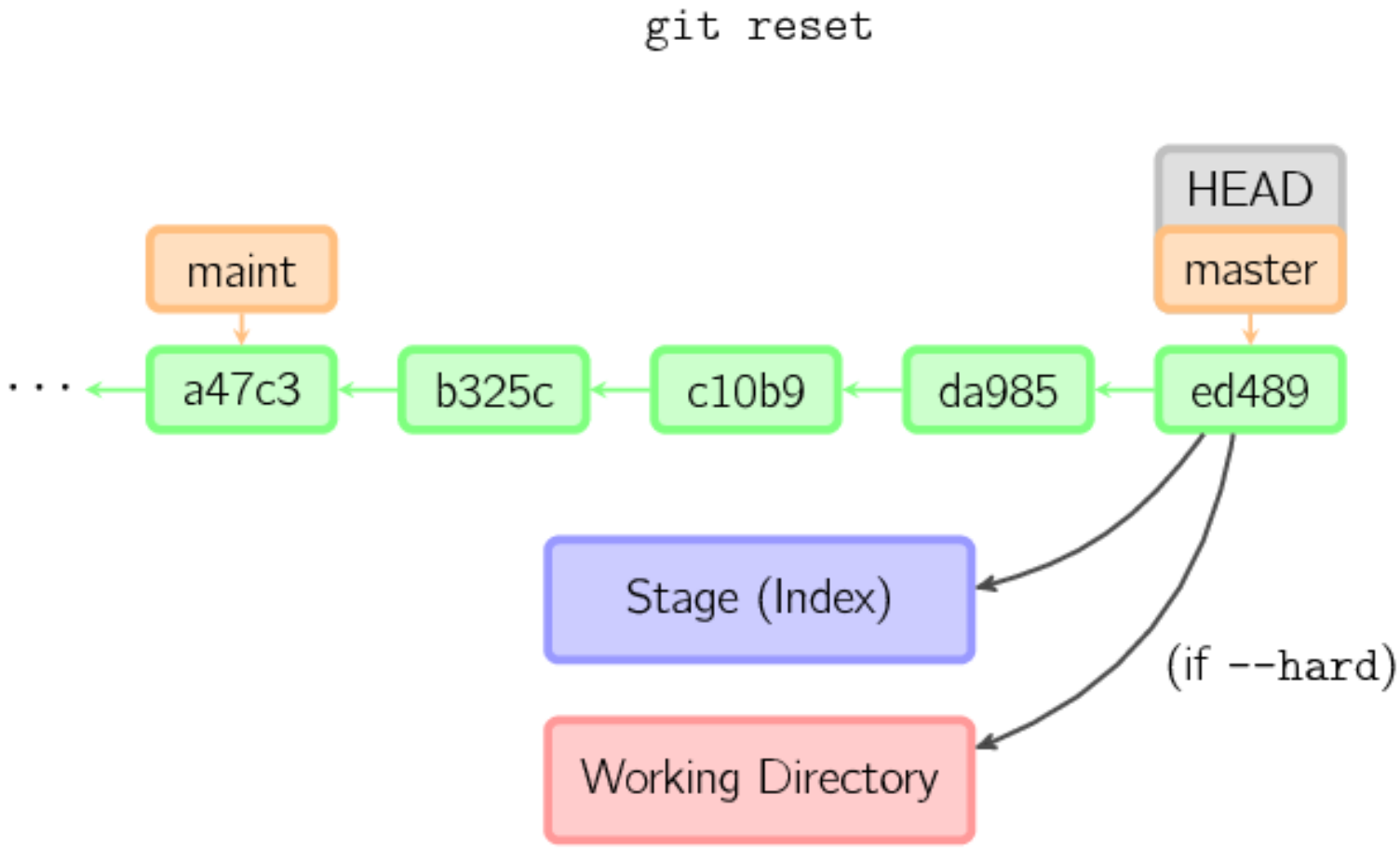

git merge other
HEAD
master
other
HEAD
master
2eecb
33104
...
a47c3
b325c
c10b9
da985
ed489
f8bc5
3-way merge
Working Directory
Stage (Index)
(if no conflicts)

Resolving Conflicts

If you change the same place in a file in two different branches, there will be a conflict when you merge:

- Must resolve manually by editing the file
- The graphical merge took, `git mergetool` may help
- Use `git diff -cc` to verify changes after resolving conflict
- Then stage and commit the changes

Reverting Changes



Use `git reset HEAD --` *file* to unstage a file

Other Commands

There are a few other basic commands which are helpful:

- `git tag -a` *`ver1.0`* `-m 'First release.'`: to mark known point, such as submitting a paper or a release of your software
- `git show-branch`: to display ancestry graph of commits

.gitignore

To ignore certain files, create a `.gitignore` file:

- Prevents output about derived objects
- Avoids accidentally adding files which should not be under SCM
- Can use wild cards and comments

Example .gitignore

```
# Compiled Object files
*.slo
*.lo
*.o
# Compiled Dynamic libraries
*.so
# Compiled Static libraries
*.lai
*.la
*.a
# Edited files
*~
.DS_Store
# Derived files
*.pdf
fort.*
depend.inc
```

Distributed Development

Multiple Repositories

Git makes it easy to collaborate with others or synchronize your work across multiple computers:

- Can manage source code on different computers (laptop, Athens, Blue Gene/Q, Beagle, . . .)
- Can coordinate work with multiple developers
- Git is fast and decentralized (a.k.a. *peer to peer*):
  - No central repository
  - Redundant
    → Can keep working when the Internet is down
    → Doesn't matter if the repository server is vaporized.

Cloning Repositories: git clone

`git clone` *`repo`* creates a local copy of another repository:

```
$ git clone git@github.com:drh/omcl.git
Cloning into omcl...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 24 (delta 7), reused 17 (delta 1)
Receiving objects: 100% (24/24), 74.36 KiB, done.
Resolving deltas: 100% (7/7), done.
```

- Pathname to repository can specify SSH, Git, or local path.
- This repository is now known as ***origin***

Remote Repositories: git remote

Use `git remote` to manage remote repositories:

```
$ git remote        # List remote repos
origin
```

Use `git remote add [shortname] [url]` to associate a name with a remote repository:

```
$ git remote add moon git://moon.com/horrible/omcl
$ git remote -v           # Detailed listing
origin git@github.com:drh/omcl.git (fetch)
origin git@github.com:drh/omcl.git (push)
moon git://moon.com/horrible/omcl.git (fetch)
moon git://moon.com/horrible/omcl.git (push)
```

Update Your Repository

To update your repository with changes from a remote repository, use `git fetch` [remote]:

```
$ git fetch origin
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From git@github.com:drh/omcl
   8cddf03..3415ba4  genius     -> origin/genius
   343c44b..2593ae6  master     -> origin/master
```

- Copies changes from remote repository to a local branch named *RemoteName/RemoteBranch* (info stored in `.git/refs/remotes`)
- Allows you to examine changes with `git diff` and then merge:

```
$ git diff origin/genius
```

One-stop Update & Merge: git pull

If you are feeling lucky, you can use `git pull` to perform a fetch and merge as one operation:

- Fetches changes and merges into current branch (dangerous!)
- Remote repository you clone from is known as ***origin***.

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From git://fortress.com/horrible/orbital
   8cddf03..343c44b  master     -> origin/master
Updating 8cddf03..343c44b
Fast-forward
 README.txt |    5 +++++
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Publishing Your Changes: git push

You can publish your changes to a remote repository:

- `git push` [remote] [branch], e.g.:

  ```
  $ git push origin master
  ```

- Must have write access to remote repository
- Dangerous if two people push to a repository at the same time
- By default tags are not pushed
  → use `git push remote origin --tags`
- To push everything under .git/refs/heads (effectively branches and tags)
  → use `git push origin --all`

Repository Hosting

Because git is distributed, setup a central repository for coordination:

- BitBucket
- GitHub

Provides:

- Hosting
- Issue Tracking
- Coordination – especially useful if you work on a laptop!
- Forking off other repositories
- Wiki
- Security

Remote Repository Setup

To setup remote repository hosting:

1. Create an account on BitBucket or GitHub
2. Generate SSH Public & Private keys:

```
$ ssh-keygen -t rsa -C "drh@horrible.com"
Generating public/private rsa key pair.
Enter file in which to save the key \
  (/home/horrible/.ssh/id_rsa):  <press enter>
...
```

3. Upload your public key (follow the instructions on the website)
4. Create a repository. Typically this is a *bare* repository – i.e. it lacks a working directory
5. Website will provide URL to use
6. Add this URL as origin:

```
git remote add origin git@github.com:horrible/omcl.git
```

Advanced Configuration

Some helpful aliases are:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git config --global alias.unstage 'reset HEAD --'
$ git config --global alias.last 'log -1 HEAD'
$ git config --global alias.hist \
  "log --pretty=format:\"%h %ad | %s%d [%an]\" \
  --graph --date=short"
```

- Can also create by editing `~/.gitconfig` (if you know what you are doing...)

Advanced Commands

See the manual for other commands, including:

- `git cherrypick`: replay one change from one branch on another
- `git rebase`: replay all changes from one branch on another
- `git add --patch`: lets you interactively choose which changes on a file to stage
- `git log`: formatting commands
- `git remote show` *RemoteName*: print info about *RemoteName*
- `git remote rm` *RemoteName*: delete a remote

Conclusion

SCM will increase your productivity and make your research more reproducible:

- Track changes to your software
- Test changes safely in a branch
- Coordinate work with (remote) colleagues
- Synchronize work across multiple machines

Conclusion

Core commands:

- `git init`: create a repository
- `git add`: stage changes
- `git commit`: store changes to repository
- `git branch`: create a branch
- `git checkout`: change workspace to a different branch
- `git merge`: merge two branches
- `git remote`: specify path to repository
- `git fetch`: retrieve changes from remote repository