

*M. Weintraub*

git

# ACKNOWLEDGMENTS

---

- <https://git-scm.com/documentation/external-links>
- <https://www.atlassian.com/git/tutorials>
- <https://courses.cs.washington.edu/courses/cse403/>
- Kevin Skoglund's lynda.com films

# GIT

---

- Used to track versions of files, especially good for text files
  - Not as good for tracking changes to media or object files
- Main use is to track versions of source code
  - *hence version control and source control will be synonymous.*



# VERSIONS

- What is a version?

A changed file

Example: you edit a word document and save it as *mydocument.html*.

Then you make some changes to it and save it again. *mydocument.html* is now different from the original.



- Why do versioning?

- Sometimes you want to undo.
- It makes sharing files with other people making changes easier. (avoids chaos)
- Sometimes you can learn from history.

- VCS have been around in some form since 1970's...

# GIT IS A DISTRIBUTED VCS

---

- The simplest VCS model is to have a single, central repository that everyone works from
  - Everyone puts changes in; everyone has to decide when and how to stay current.
  - SCCS, CVS, SVN all work by tracking changes to go from one version to another version of each file
- Git uses Distributed Version Control
  - Each user has his/her own repository.
    - *BUT in the real world, you have to work with others and a central repository is standard and very good way to work.*

*We will see later how this is defined by process and supported by git.*  
*This will be the model you will use in this course*
  - Git emphasizes tracking change sets and not individual versions of a file.
    - *If you change three files in a directory, those three changes are marked as a single change set.*

# HOW GIT WORKS CONCEPTUALLY

---

git uses a three part architecture

1. *All work is done in the **working** area.*  
For example, you use eclipse to create a java file.

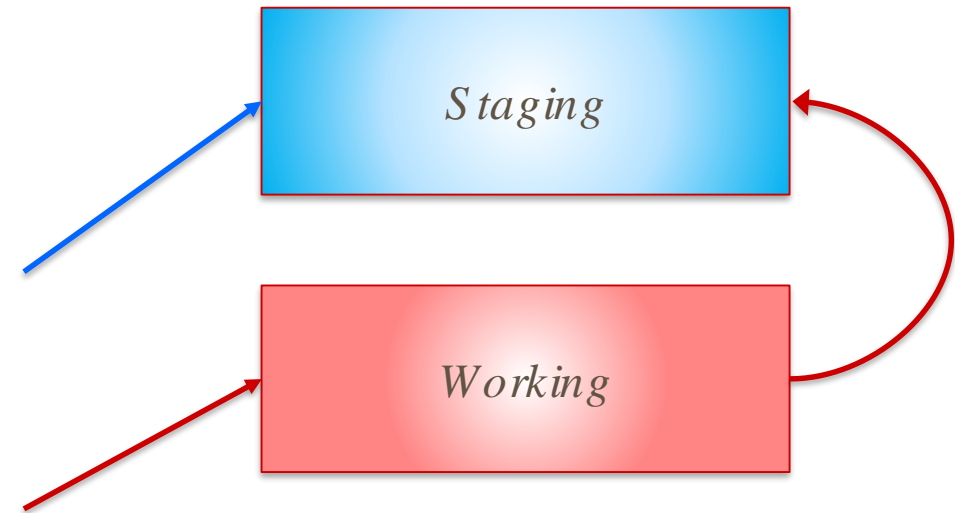


# HOW GIT WORKS CONCEPTUALLY

---

git uses a three part architecture

2. “Finished” work is added to the *staging* area.  
For example, your java module passes the tests.
1. All work is done in the *working* area.  
For example, you use eclipse to create a java file.

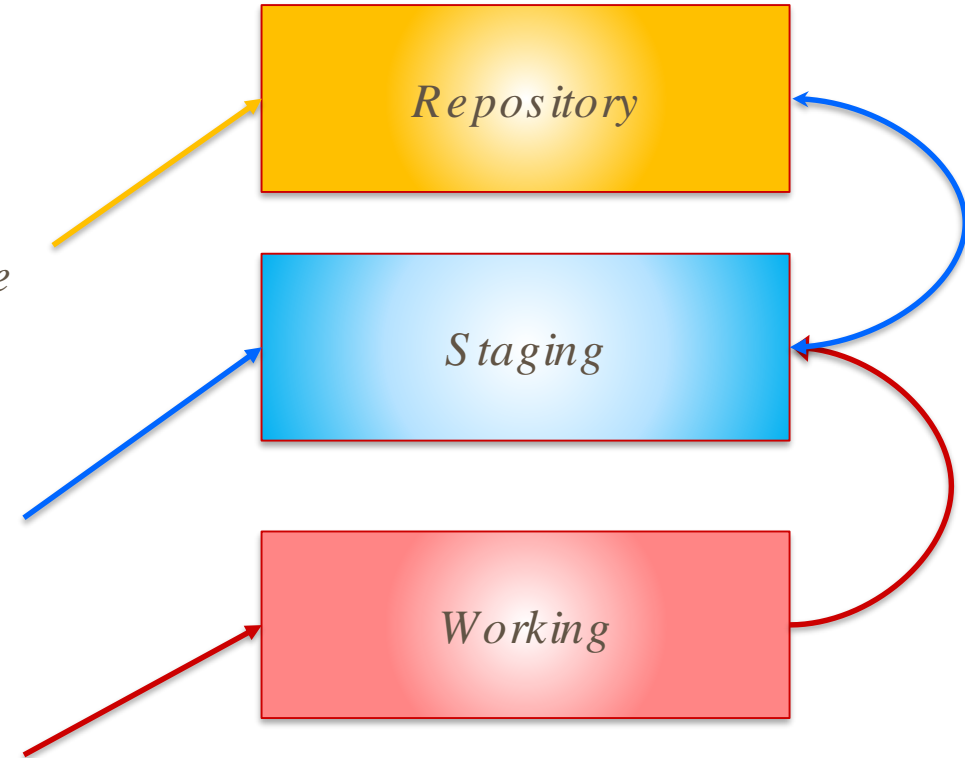


# HOW GIT WORKS CONCEPTUALLY

---

git uses a three part architecture

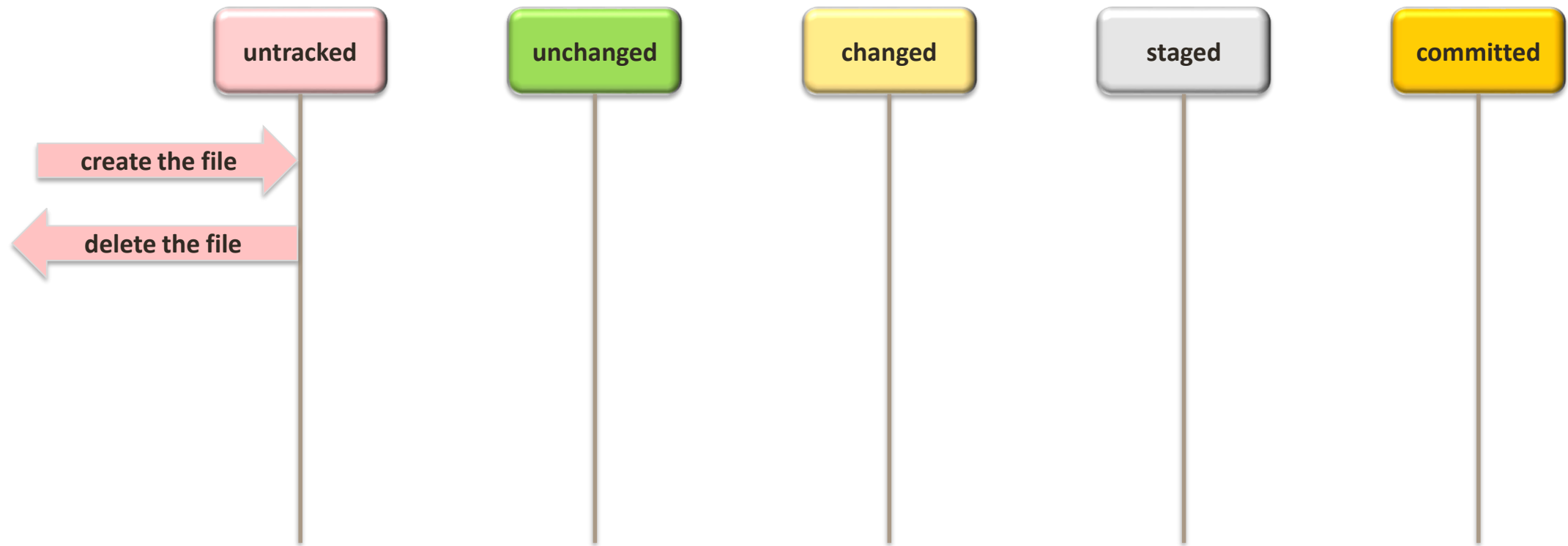
1. All work is done in the **working** area.  
For example, you use eclipse to create a java file.
2. “Finished” work is added to the **staging** area.  
For example, your java module passes the tests.
3. When a batch of work is ready to be committed as complete, the entire batch is committed to the **repository**.





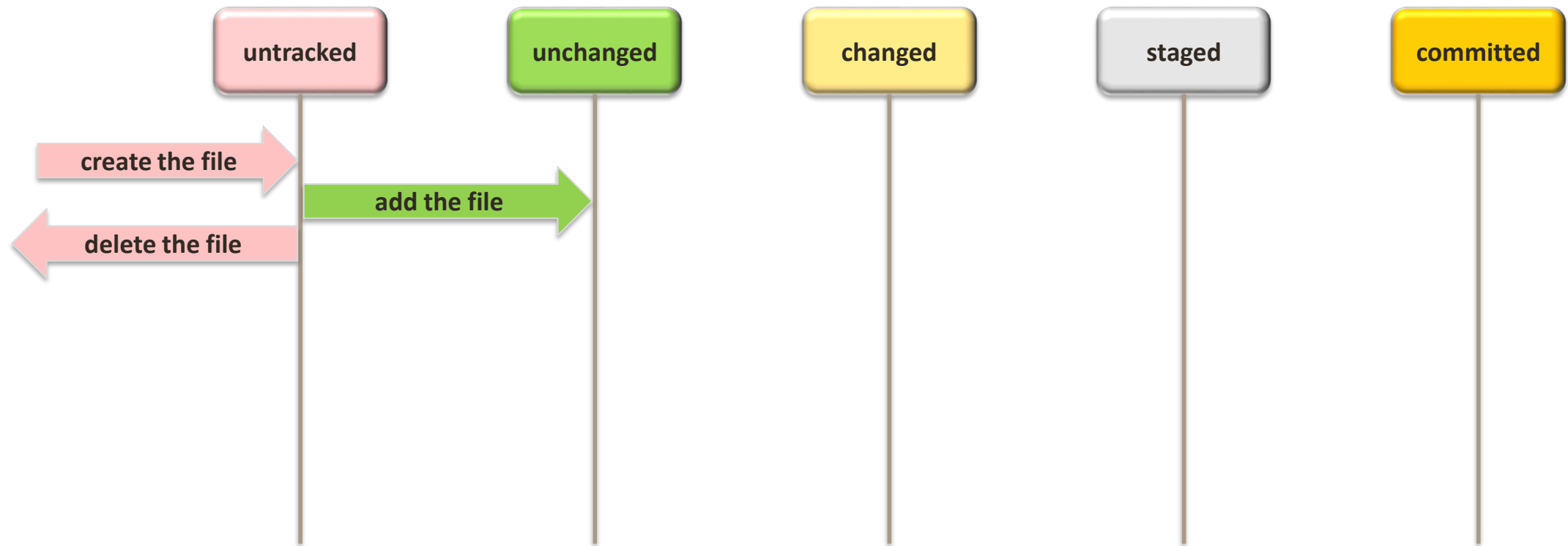
# FROM A FILE'S PERSPECTIVE

---



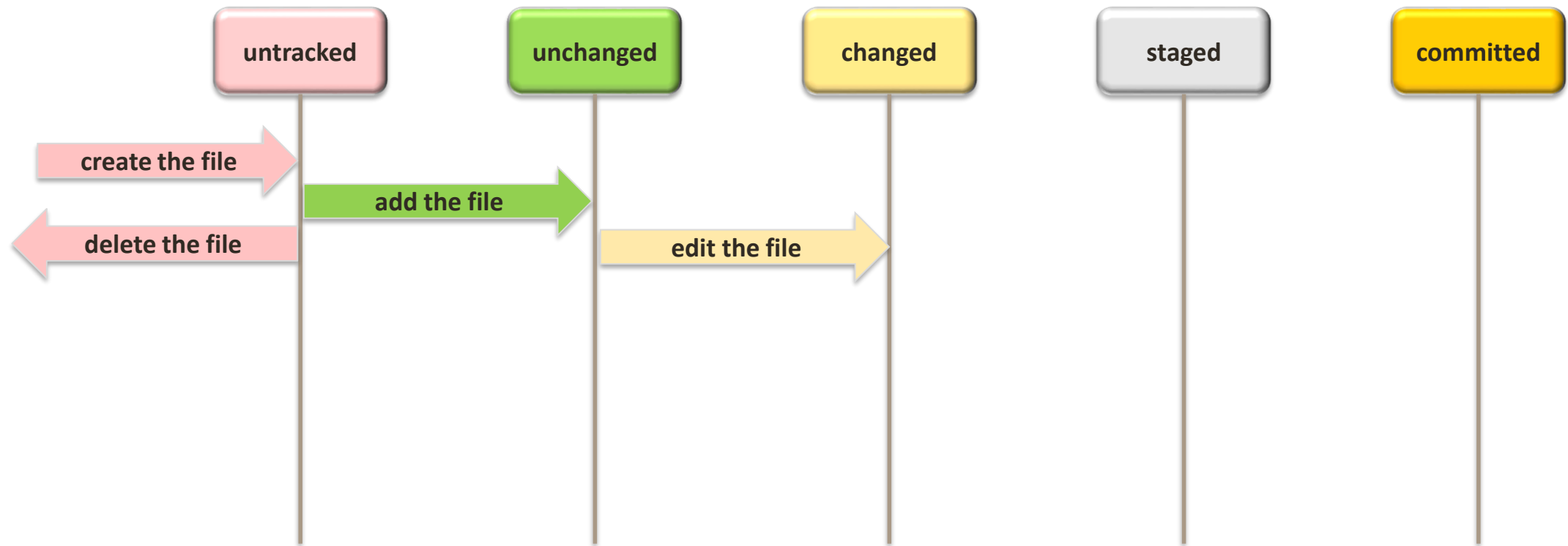
# FROM A FILE'S PERSPECTIVE

---



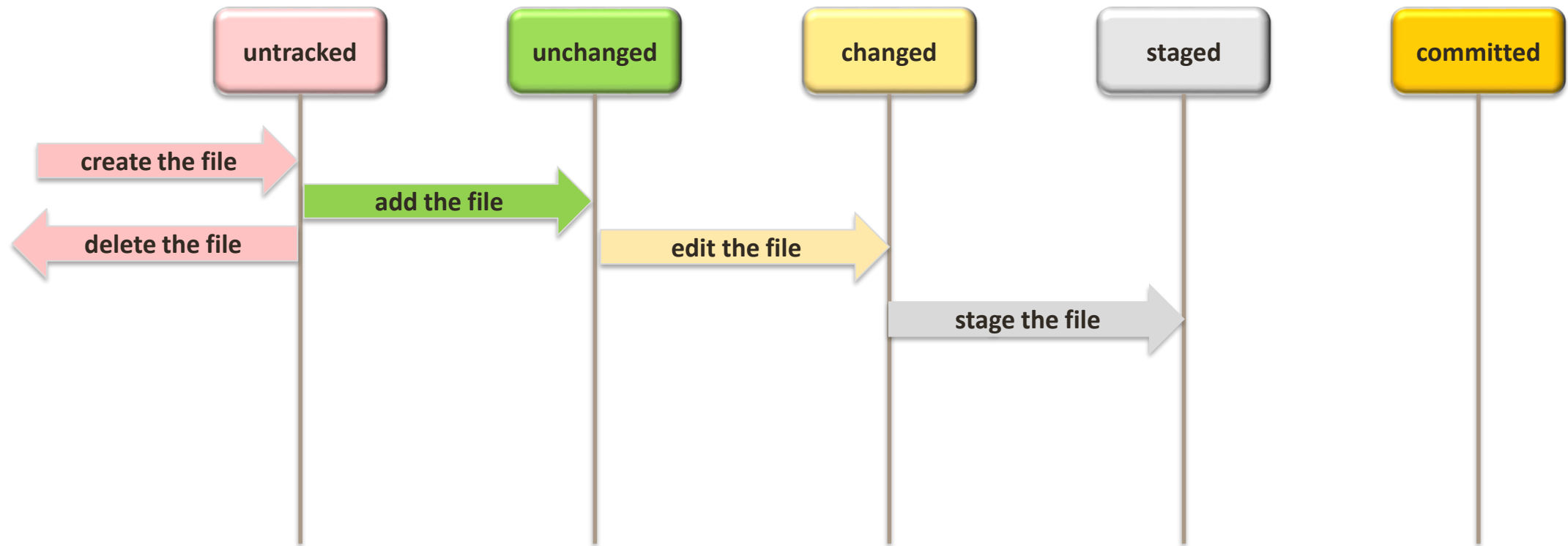
# FROM A FILE'S PERSPECTIVE

---



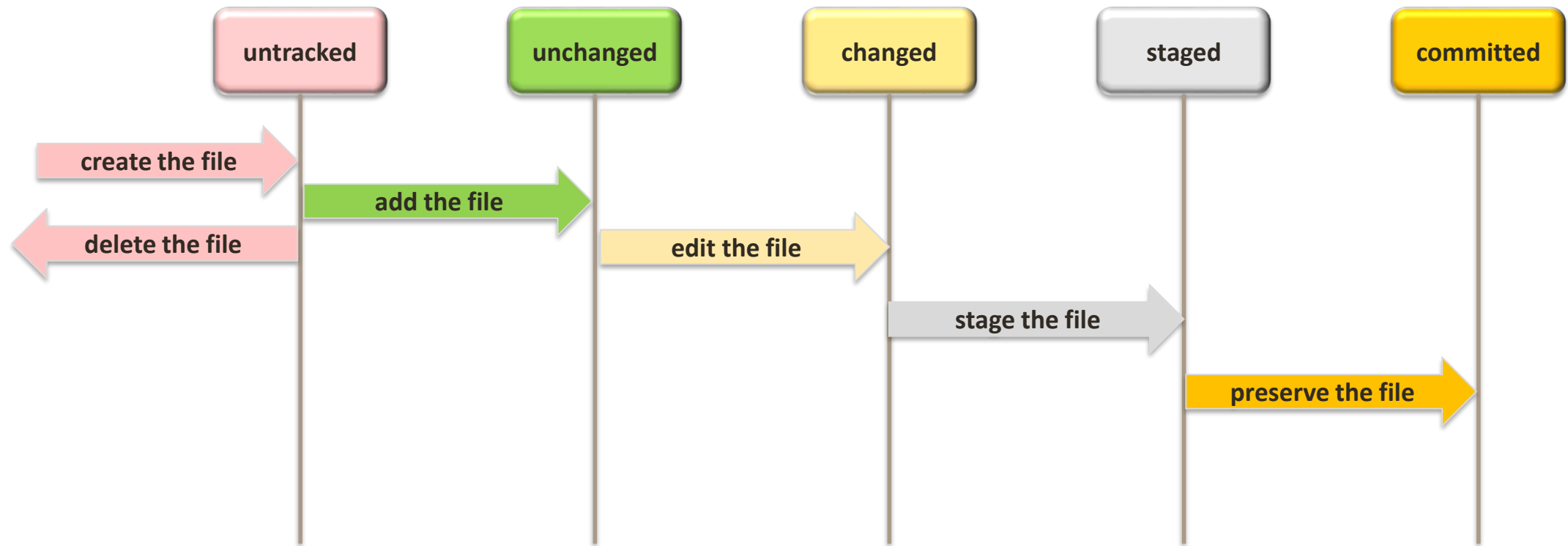
# FROM A FILE'S PERSPECTIVE

---



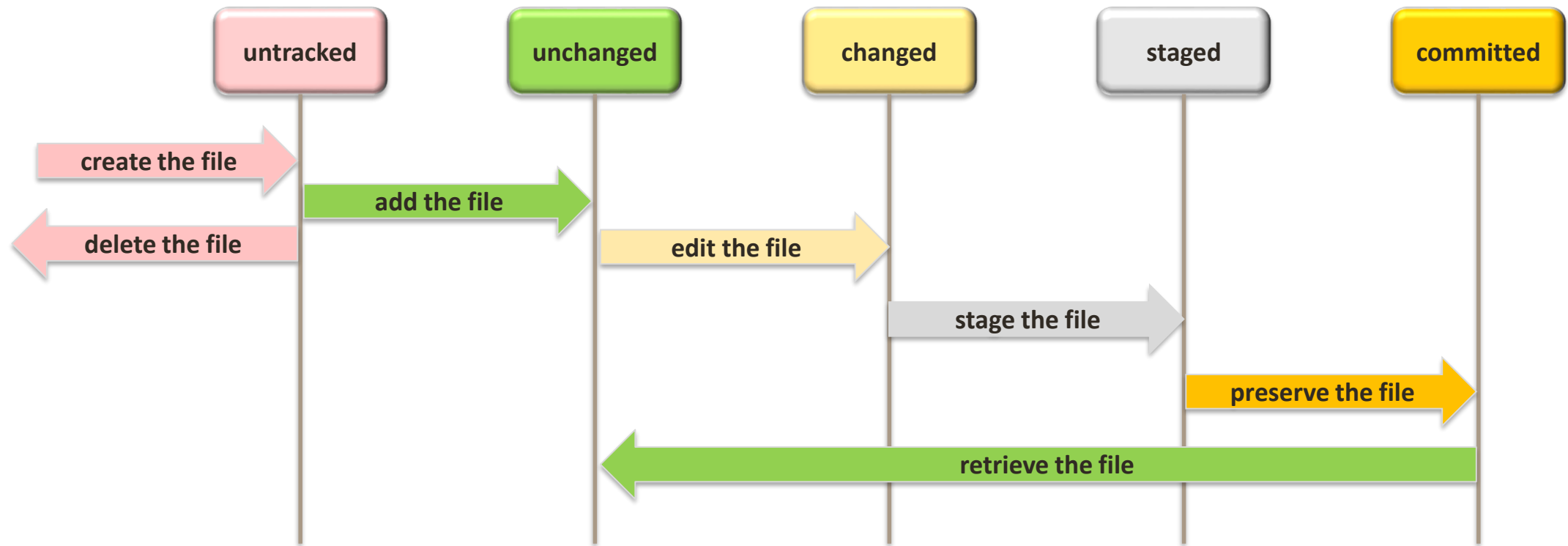
# FROM A FILE'S PERSPECTIVE

---



# FROM A FILE'S PERSPECTIVE

---



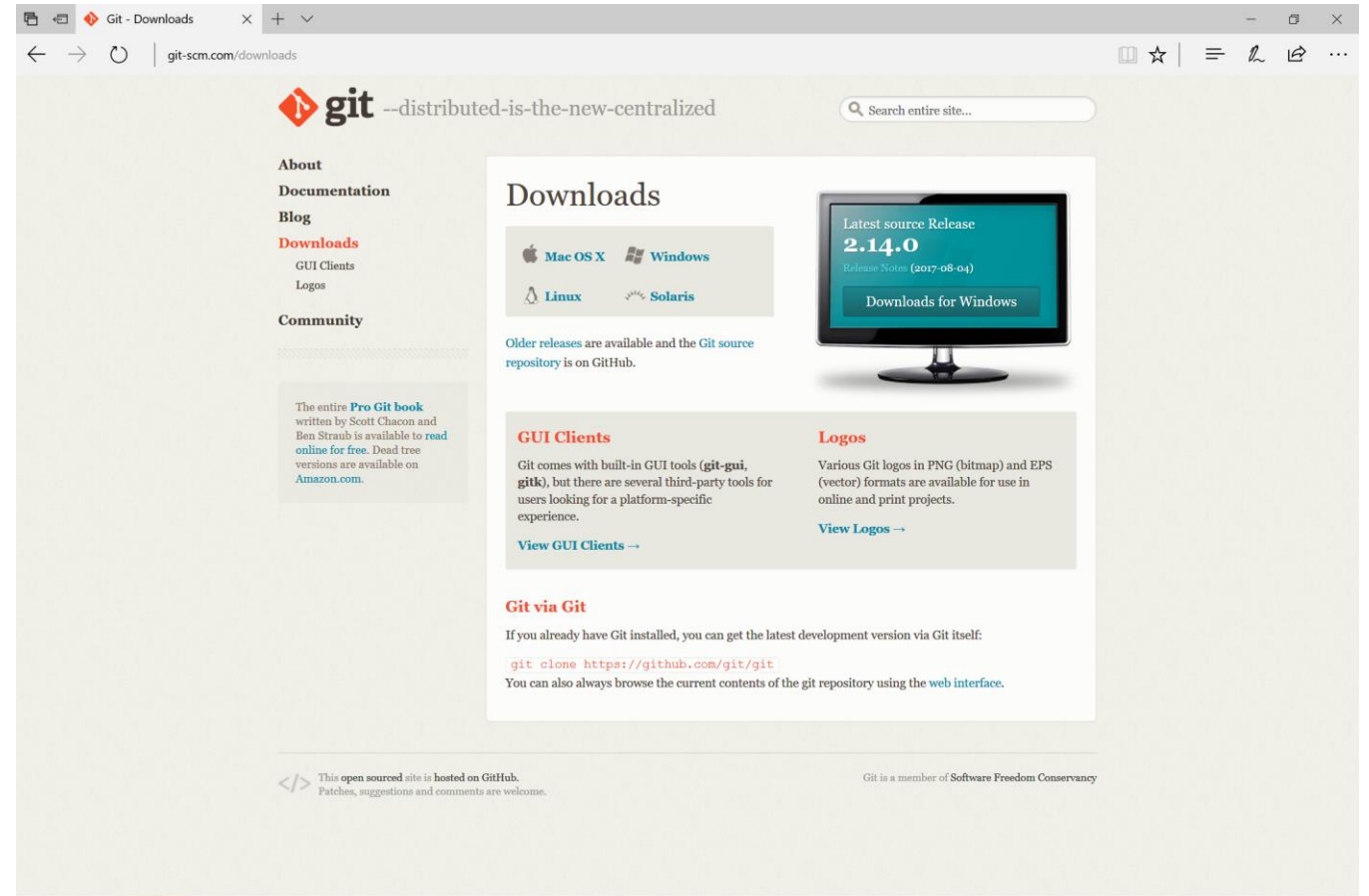
# GETTING GIT

<https://git-scm.com/downloads>

Suggest accepting the defaults

All our work will be through the command line.

Windows users will use `git bash`.



# CONFIGURING GIT

---

## .gitconfig

user specific config's

- *On linux/OSX: look in ~*
- *On windows: look in \$HOME*

## .git/config – project specific config's

git provides commands to set config's:

**\$ git config <level> <config>**

where

- system** (system-wide)
- global** (global to the user)
- nothing:** (project level)

```
Michael@berber ~ $ git config --list
```

```
core.symlinks=false
```

```
core.autocrlf=true
```

```
core.fscache=true
```

```
color.diff=auto
```

```
[...]
```

```
user.name=maw
```

```
user.email=maw@ccs.neu.edu
```

```
Michael@berber ~ $ git config --global core.editor "emacs -w"
```

```
# tells git which editor to open when it needs you to do something
```

```
# You can put vim, nano, emacs, notepad.exe – whatever you want.
```

```
# the -w option tells git to wait until the editor finishes.
```



# FIRST STEP: CREATE A REPO

---

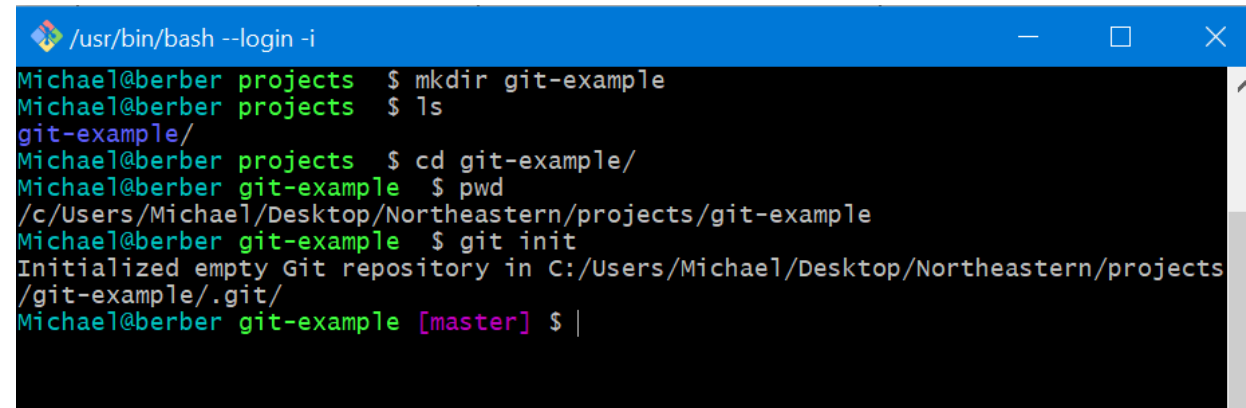
```
$ git init
```

Fine Print:

You should be in the top directory of the project.

Here, I am going to work on a project called *git-example*. I chose to put my project off `Desktop>Northeastern>projects`.

git will treat this directory as a git repo and to track all changes made inside this directory, no matter how deeply nested they are

A terminal window with a blue title bar containing the text "/usr/bin/bash --login -i". The terminal shows a series of commands and their outputs. The user "Michael" is at the "berber" machine in the "projects" directory. They run "mkdir git-example", then "ls" to confirm the directory exists. They then run "cd git-example/" to move into the new directory. Next, they run "pwd" which outputs "/c/Users/Michael/Desktop/Northeastern/projects/git-example". Finally, they run "git init", which outputs "Initialized empty Git repository in C:/Users/Michael/Desktop/Northeastern/projects/git-example/.git/". The prompt then changes to "Michael@berber git-example [master] \$ |".

```
/usr/bin/bash --login -i
Michael@berber projects $ mkdir git-example
Michael@berber projects $ ls
git-example/
Michael@berber projects $ cd git-example/
Michael@berber git-example $ pwd
/c/Users/Michael/Desktop/Northeastern/projects/git-example
Michael@berber git-example $ git init
Initialized empty Git repository in C:/Users/Michael/Desktop/Northeastern/projects
/git-example/.git/
Michael@berber git-example [master] $ |
```

# WHERE GIT LURKS

---

\$ `git init` creates `$CURRENT/.git`

Everything git keeps about the project is in here, even for nested folders.

If you \$ `rm -rf $CURRENT/.git`, git would no longer track your project.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ pwd
/c/Users/Michael/Desktop/Northeastern/projects/git-example
Michael@berber git-example [master] $ ls -a
./ ../ .git/
Michael@berber git-example [master] $ ls .git
config description HEAD hooks/ info/ objects/ refs/
Michael@berber git-example [master] $ |
```

# ADDING A FILE TO GIT

**\$ git add <file>**

Puts the file under git control.

Moves a file from the working directory into staging.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
Michael@berber git-example [master] $ echo "My first file" > file1.txt
Michael@berber git-example [master] $ ls
file1.txt
Michael@berber git-example [master] $ cat file1.txt
My first file
Michael@berber git-example [master] $ git status
On branch master

Initial commit

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

        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [master] $ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git status
On branch master

Initial commit

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

        new file:   file1.txt
Michael@berber git-example [master] $ |
```

# LEARNING WHAT'S CHANGED

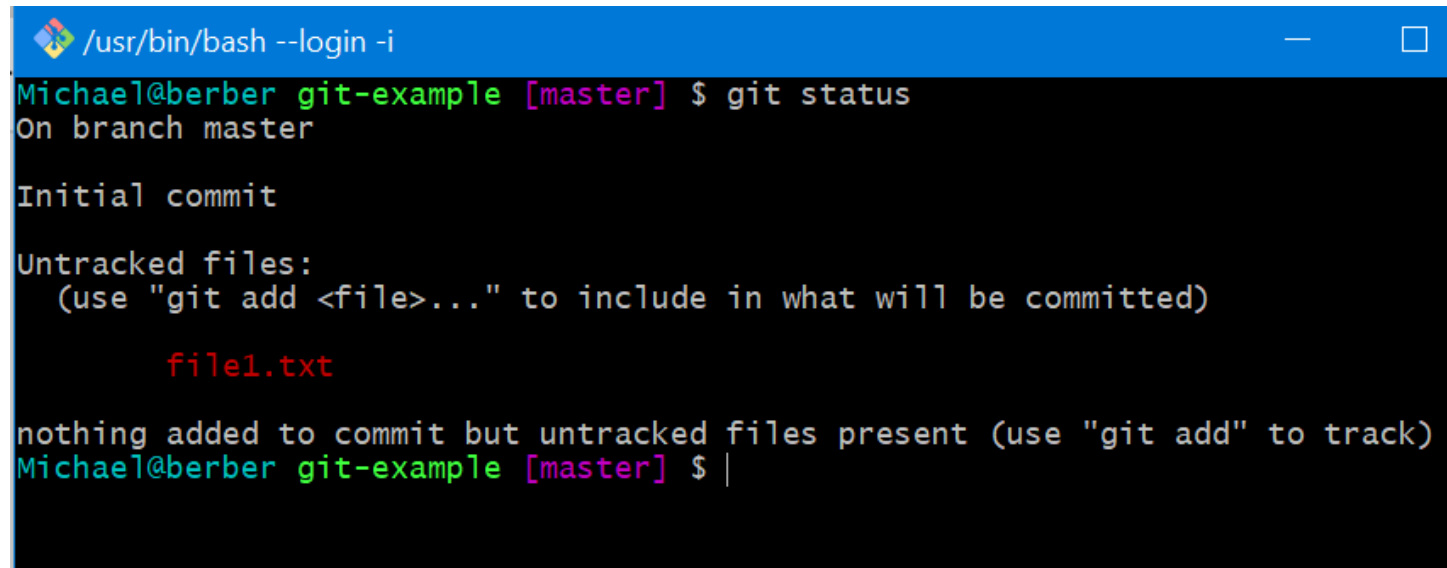
---

**\$ git status**

*Reports the differences between the tiers*

*Here, git status tells us:*

- 1. We are on a branch called **master***
- 2. There is nothing staged*
- 3. There is a file in the working directory that git is not tracking*

A terminal window with a blue title bar containing the text "/usr/bin/bash --login -i". The terminal output shows the command "git status" being executed. The output indicates the user is on the "master" branch, it's an initial commit, and there are untracked files, specifically "file1.txt". It also provides instructions on how to use "git add" to track files.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git status
On branch master

Initial commit

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

        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [master] $ |
```

# CLEANING OUT UNTRACKED FILES

\$ git clean <option>

clean options

- n list what would be cleaned, but don't actually remove anything
- f delete anything that is not being tracked (be careful using this)

Removes untracked files from WORKING

i.e., anything that hasn't been **git added**

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ echo "a temp file" > temp2
Michael@berber git-example [master] $ echo "a temp file" > temp1
Michael@berber git-example [master] $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    temp1
    temp2

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [master] $ git add temp1
warning: LF will be replaced by CRLF in temp1.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   temp1

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

    temp2

Michael@berber git-example [master] $ git clean -n
Would remove temp2
Michael@berber git-example [master] $ git clean -f
Removing temp2
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   temp1

Michael@berber git-example [master] $ |
```

# NOT EVERYTHING SHOULD BE TRACKED

---

You may create all kinds of files in a project, but not every file is important to preserve

- Files that change constantly
  - *E.g. log files, databases*
- Temporary files
- Files that will be created by builds
  - *E.g. object files, compressed files*
- Instance data
  - *E.g. user-generated content (should be stored in a database)*

Typically you want to track all source files (code, documents, templates,...) and configuration files

# TELLING GIT TO IGNORE ITEMS

---

**.gitignore** tells git to ignore specific files or types of files

It's placed in the project's root directory

[Common practice is to include .gitignore in your repo]

The community has built up recommendations for what should go into a .gitignore based on environment

See <https://github.com/github/gitignore>

```
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt

# Mobile Tools for Java (J2ME)
.mtj.tmp/
```

*Excerpt from the java .gitignore template from github.com*

*<https://github.com/github/gitignore/blob/master/Java.gitignore>*

# YOU CAN HAVE MANY `.gitignore` FILES

---

Git ignore rules are usually defined in a `.gitignore` file at the root of your repository.

- This is the convention and simplest approach

You may have different `.gitignore` files in different directories in your repository.

Each pattern in a particular `.gitignore` file is tested relative to the directory tree containing that `.gitignore`

- It is best to use `.gitignores` in sub-directories to augment the rules of `.gitignores` in the parent directories.
- Typically you should only include patterns in `.gitignore` that will benefit other users of the repository.



# .GITIGNORE EXAMPLE

.gitignore lines may identify specific files or may be simple regular expressions

- ? [char\_set] [range] !

*[ note: one may use / to mean files in subdirectories]*

\*.foo

! myReallyImportant.foo

This would ignore all **foo** files but not ignore *myReallyImportant.foo*

PS: git does not track empty directories!!

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file-one file-two
Michael@berber git-example [master] $ echo "a temp file" > temp1
Michael@berber git-example [master] $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    temp1

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [master] $ echo "temp1" > .gitignore
Michael@berber git-example [master] $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [master] $ ls
file-one file-two temp1
Michael@berber git-example [master] $ |
```

# MOVING FROM STAGING TO THE REPO

---

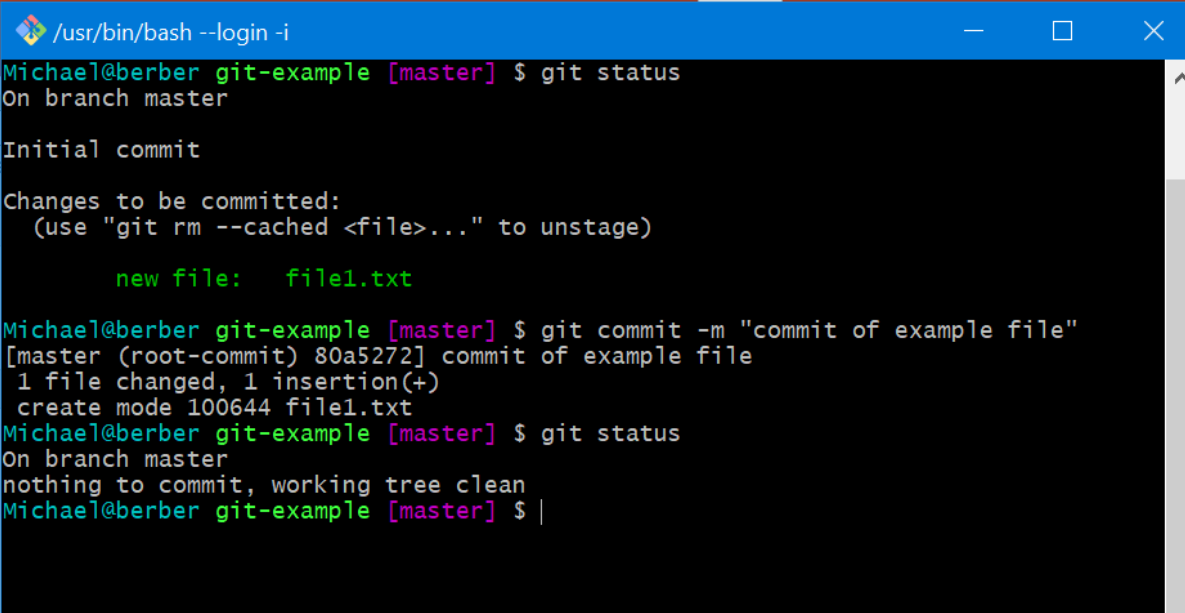
```
$ git commit -m "message"
```

Moves a change set from staging to the repository.

Important note about commit messages

This is a label describing what is contained in this change set so when anyone looks at it in the future, they do not have to look at the actual changes to know what the change set contains.

Good, descriptive commit messages are vital.

A terminal window with a blue title bar containing the text "/usr/bin/bash --login -i". The terminal shows the following commands and output:

```
Michael@berber git-example [master] $ git status
On branch master

Initial commit

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

    new file:   file1.txt

Michael@berber git-example [master] $ git commit -m "commit of example file"
[master (root-commit) 80a5272] commit of example file
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ |
```

# COMMIT MESSAGES MATTER

---

You are writing these messages for other people to read and use later on –  
and you may not be available

In the future, you may not remember what the commit entailed either.

# BEST PRACTICES

---

Re-establishing the context of a piece of code is wasteful. We can't avoid it completely, so our efforts should go to reducing it to as small as possible. Commit messages can do exactly that and as a result, *a commit message shows whether a developer is a good collaborator.*

A good commit message should answer three questions:

- 1. Why is it necessary?** It may fix a bug, it may add a feature, it may improve performance, reliability, stability, or just be a change for the sake of correctness.
- 2. How does it address the issue?** For short obvious patches this part can be omitted, but it should be a high level description of what the approach was.
- 3. What effects does the patch have?** (In addition to the obvious ones, this may include benchmarks, side effects, etc.)

# BEST PRACTICES

---

- Most importantly, please format your commit messages in the following way:
  - Short (50 chars or less) summary of changes  
More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.
  - Further paragraphs come after blank lines.
    - *Bullet points are okay, too*
    - *Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here*
- Use imperative statements in the subject line, e.g. "Fix broken Javadoc link"
- Begin the subject line sentence with a capitalized verb, e.g. "Add, Prune, Fix, Introduce, Avoid, etc"
- Do not end the subject line with a period
- Keep the subject line to 50 characters or less if possible
- Wrap lines in the body at 72 characters or less
- Mention associated jira issue(s) at the end of the commit comment, prefixed with "Issue: " as above
- In the body of the commit message, explain how things worked before this commit, what has changed, and how things work now

# HOW GIT IDENTIFIES COMMITS

git uses a hash, SHA-1, against all the changes in the change set to create a unique id

- 40 digit hex number

Here, the first commit has SHA-1: 80a527... and covers file1.txt.

The second commit has SHA-1: 439a3... and covers the updated file1.txt and the new file2.txt

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git log
commit 80a5272837c9fa3d682d8040d2885cf3a950a249 (HEAD -> master)
Author: maw <maw@ccs.neu.edu>
Date:   Wed Aug 9 19:03:52 2017 -0400

    commit of example file
Michael@berber git-example [master] $ vi file1.txt
Michael@berber git-example [master] $ cat file1.txt
My first file that I changed
Michael@berber git-example [master] $ 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:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ echo "A second file" > file2.txt
Michael@berber git-example [master] $ 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:   file1.txt

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

        file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ git commit -a -m "amended first file and added second file per tkt 7"
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory.
[master 439a3a0] amended first file and added second file per tkt 7
 1 file changed, 1 insertion(+), 1 deletion(-)
Michael@berber git-example [master] $ git log
commit 439a3a0de5ada98a87dbe961256684bdd456e0d7 (HEAD -> master)
Author: maw <maw@ccs.neu.edu>
Date:   Thu Aug 10 17:24:01 2017 -0400

    amended first file and added second file per tkt 7

commit 80a5272837c9fa3d682d8040d2885cf3a950a249
Author: maw <maw@ccs.neu.edu>
Date:   Wed Aug 9 19:03:52 2017 -0400

    commit of example file
Michael@berber git-example [master] $ |
```

# HEAD

**HEAD** is a pointer to the most recent commit on a branch

You can reference **HEAD** directly in git commands

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```

`$ git init`

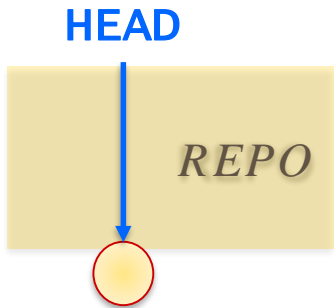
*WORKING*

time 

# HEAD

**HEAD** is a pointer to the most recent commit on a branch

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```



```
$ git init
```

*WORKING*

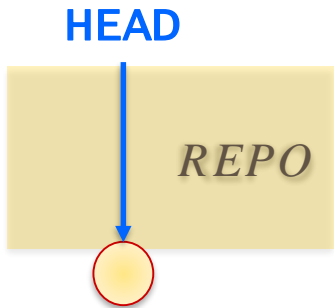
time →



# HEAD

**HEAD** is a pointer to the most recent commit on a branch

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```



```
$ git init
```

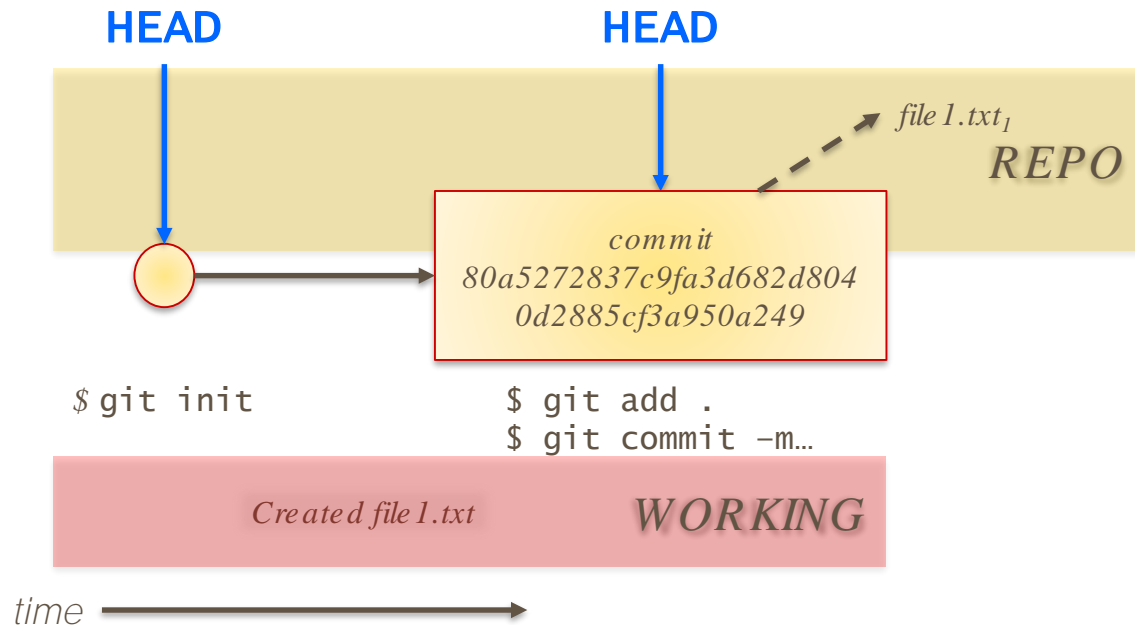


time →

# HEAD

**HEAD** is a pointer to the most recent commit on a branch

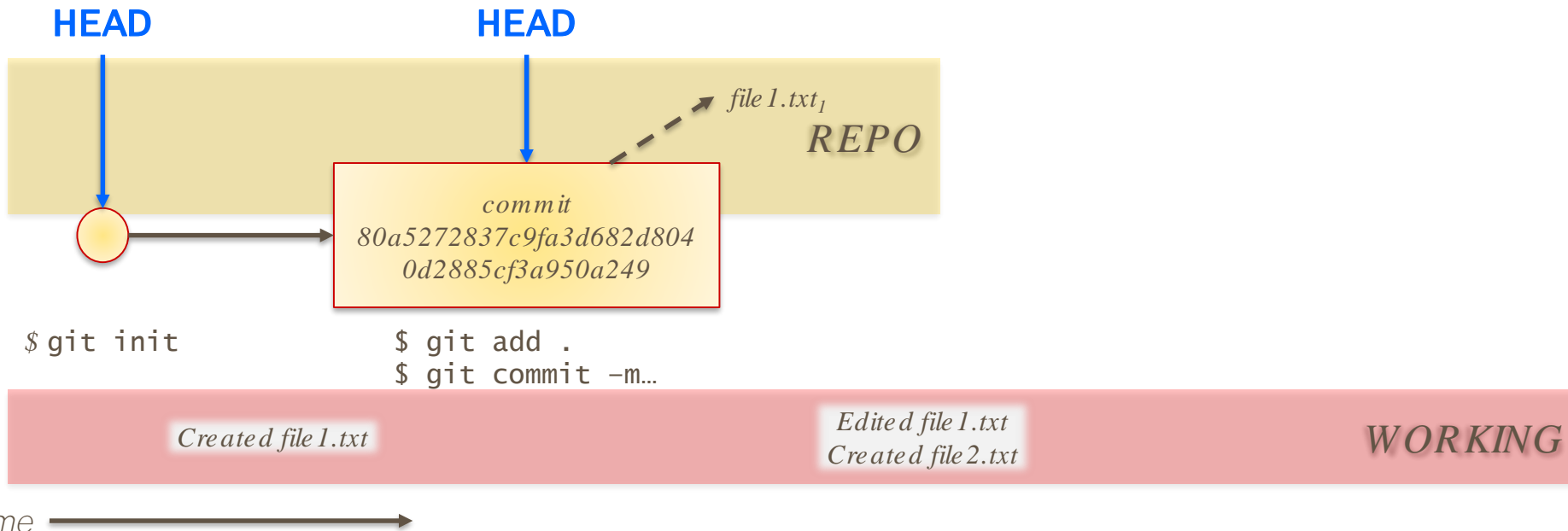
```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```



# HEAD

**HEAD** is a pointer to the most recent commit on a branch

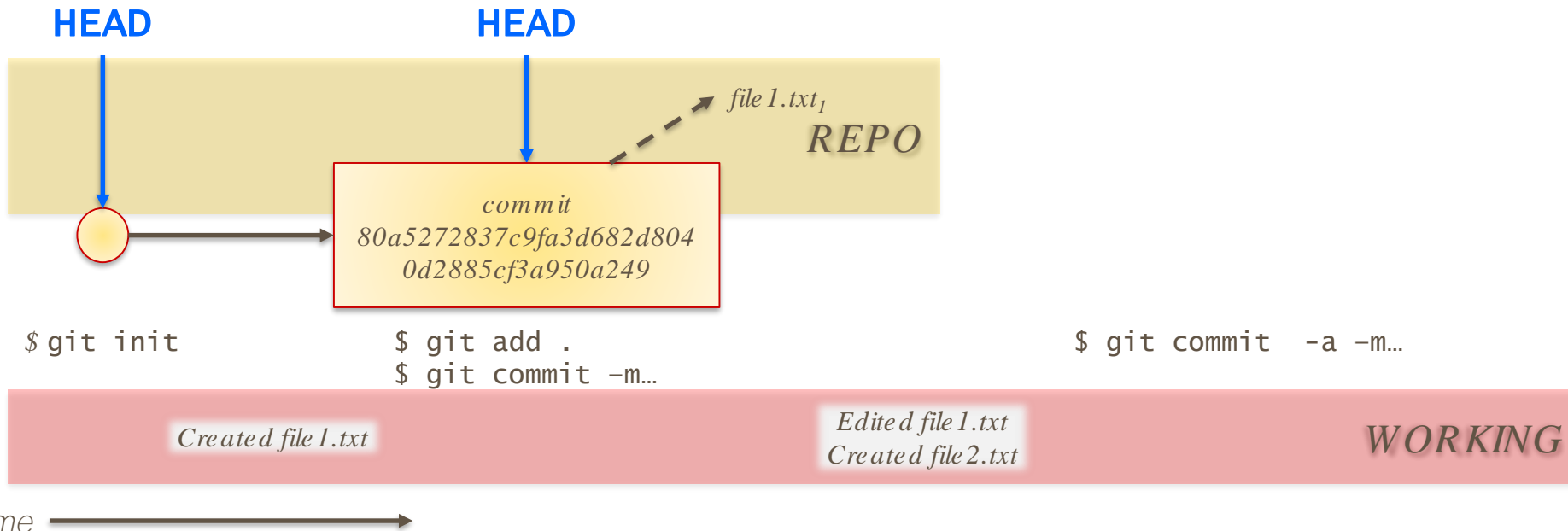
```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```



# HEAD

**HEAD** is a pointer to the most recent commit on a branch

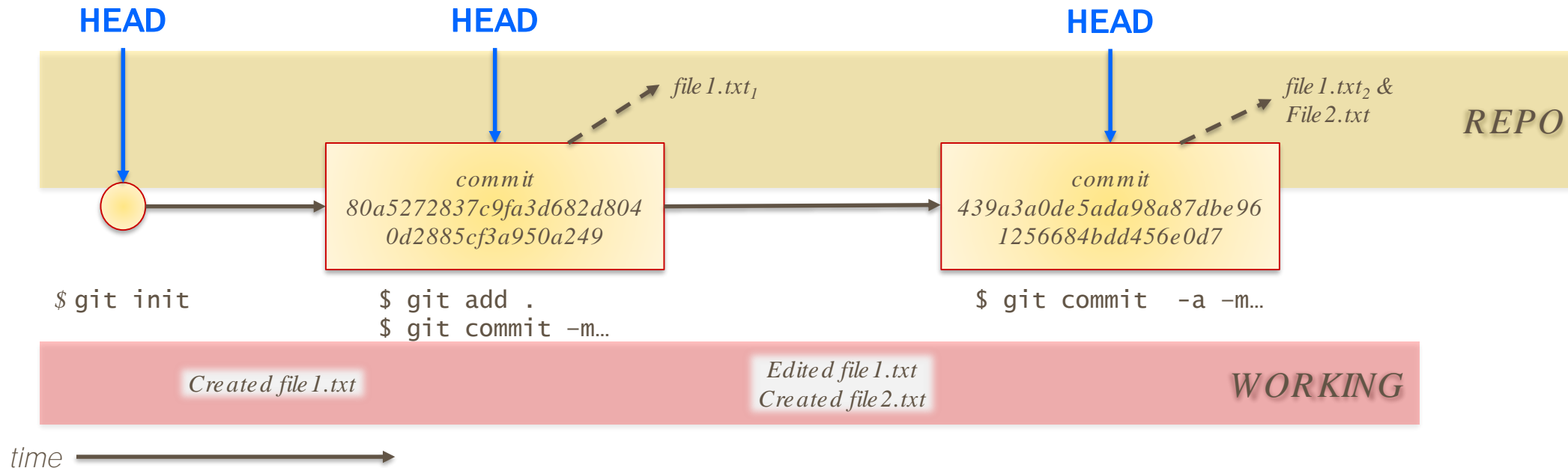
```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```



# HEAD

**HEAD** is a pointer to the most recent commit on a branch

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls -a
./ ../ .git/ file1.txt file2.txt
Michael@berber git-example [master] $ cd .git
Michael@berber .git [GIT_DIR!] $ ls
COMMIT_EDITMSG config description HEAD hooks/ index info/ logs/ objects/ refs/
Michael@berber .git [GIT_DIR!] $ cat HEAD
ref: refs/heads/master
Michael@berber .git [GIT_DIR!] $ cat refs/heads/master
439a3a0de5ada98a87dbe961256684bdd456e0d7
Michael@berber .git [GIT_DIR!] $ |
```



# LISTING COMMITS

\$ git log

Helpful ways to use the log:

\$ git log --oneline

\$ git log --since "2017-9-30"

\$ git log --until "2017-9-30"

\$ git log --since "one month ago" --until "1 day ago"

\$ git log --since=1.month --until=1.day

\$ git log --author="maw"

\$ git log --grep="someKeyword"

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git log
commit 439a3a0de5ada98a87dbe961256684bdd456e0d7 (HEAD -> master)
Author: maw <maw@ccs.neu.edu>
Date: Thu Aug 10 17:24:01 2017 -0400

    amended first file and added second file per tkt 7

commit 80a5272837c9fa3d682d8040d2885cf3a950a249
Author: maw <maw@ccs.neu.edu>
Date: Wed Aug 9 19:03:52 2017 -0400

    commit of example file
Michael@berber git-example [master] $ git log --oneline
439a3a0 (HEAD -> master) amended first file and added second file per tkt 7
80a5272 commit of example file
Michael@berber git-example [master] $ |
```

# LISTING WHAT'S DIFFERENT

`$ git diff <working file>`

Tells us what different between something in **WORKING** and **HEAD** in the repo. Shows us only the lines where something is different. This may include white space.

The **additions** are in green and **deletions** are in red.

`$ git diff -w` ignores whitespace differences

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls
file1.txt file2.txt
Michael@berber git-example [master] $ cat file1.txt
My first file that I changed
Michael@berber git-example [master] $ cat file2.txt
Total change to file 2
Michael@berber git-example [master] $ echo "extension to file 1" >> file1.txt
Michael@berber git-example [master] $ cat file1.txt
My first file that I changed
extension to file 1
Michael@berber git-example [master] $ git diff
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory.
diff --git a/file1.txt b/file1.txt
index 2c67109..f9e10c0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 @@
 My first file that I changed
+extension to file 1
Michael@berber git-example [master] $ echo "a total rewrite" > file2.txt
Michael@berber git-example [master] $ cat file2.txt
a total rewrite
Michael@berber git-example [master] $ git diff
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in file2.txt.
The file will have its original line endings in your working directory.
diff --git a/file1.txt b/file1.txt
index 2c67109..f9e10c0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 @@
 My first file that I changed
+extension to file 1
diff --git a/file2.txt b/file2.txt
index 3ff22b0..f7b9c77 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1,1 @@
-Total change to file 2
+a total rewrite
Michael@berber git-example [master] $ |
```

# LISTING STAGED FILE DIFFERENCES

```
$ git diff --staged
```

Tells us what differences have been **STAGED** and what is in **HEAD** in the repo.

The **additions** are in green and **deletions** are in red.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ 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:   file1.txt
        modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ git add .
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in file2.txt.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file1.txt
        modified:   file2.txt

Michael@berber git-example [master] $ git diff
Michael@berber git-example [master] $ git diff --staged
diff --git a/file1.txt b/file1.txt
index 2c67109..f9e10c0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 @@
 My first file that I changed
+extension to file 1
diff --git a/file2.txt b/file2.txt
index 3ff22b0..f7b9c77 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1,1 @@
-Total change to file 2
+a total rewrite
Michael@berber git-example [master] $ |
```



# DELETING FILES

## \$ git rm

Git only cares if we delete tracked files

- So until you've done a git add on a file, git won't care if you delete it.

1. Delete the file by sending it to the trash or by the command line
2. Stage the change using `git rm` (as opposed to `git add`)
3. `git commit` the change.

```
Michael@berber git-example [master] $ ls
file1.txt file2.txt
Michael@berber git-example [master] $ echo "a temporary file" > temp1
Michael@berber git-example [master] $ echo "another temporary file" > temp2
Michael@berber git-example [master] $ ls
file1.txt file2.txt temp1 temp2
Michael@berber git-example [master] $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    temp1
    temp2

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [master] $ git add .
warning: LF will be replaced by CRLF in temp1.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in temp2.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git commit -m "temporary files for delete example"
[master 7e7435e] temporary files for delete example
 2 files changed, 2 insertions(+)
 create mode 100644 temp1
 create mode 100644 temp2
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ rm temp1
Michael@berber git-example [master] $ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:    temp1

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ git rm temp1
rm 'temp1'
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    temp1

Michael@berber git-example [master] $ git commit -m "example of removing file from repo"
[master 2c0550b] example of removing file from repo
 1 file changed, 1 deletion(-)
 delete mode 100644 temp1
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ |
```

# MOVES / RENAMING

## (TWO STEPS)

You can mv files on the command line. git will notice the change, but treat it as two actions.

The git workflow is:

1. Stage the changes
  - a) `git add` the “new” file.
  - b) `git rm` the “old” file”
2. git will then notice the renaming
3. Commit the change
  - a) `git commit -m “commit not”`

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls
file1.txt file2.txt
Michael@berber git-example [master] $ mv file1.txt file-one
Michael@berber git-example [master] $ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    file1.txt

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

        file-one

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ git add file-one
warning: LF will be replaced by CRLF in file-one.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git rm file1.txt
rm 'file1.txt'
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    file1.txt -> file-one

Michael@berber git-example [master] $ git commit -m "example of moving files"
[master c61ed88] example of moving files
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename file1.txt => file-one (100%)
Michael@berber git-example [master] $ |
```

# MOVES / RENAMING

## (ONE STEP)

---

```
$ git mv <current-file-name> <future-file-name>
```

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ ls
file2.txt  file-one
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ git mv file2.txt file-two
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    file2.txt -> file-two

Michael@berber git-example [master] $ git commit -m "second way to move files"
[master 5f43149] second way to move files
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename file2.txt => file-two (100%)
Michael@berber git-example [master] $ |
```

# CTRL-Z (UNDOING WORK)

---

Sometimes you want to restore what's in WORKING from the repo.

Restoring a file:

```
$ git checkout <file-name>
```

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ ls
file-one file-two
Michael@berber git-example [master] $ cat file-one
My first file that I changed
extension to file 1
Michael@berber git-example [master] $ echo "this will be a mistake" >> file-one
Michael@berber git-example [master] $ cat file-one
My first file that I changed
extension to file 1
this will be a mistake
Michael@berber git-example [master] $ git checkout file-one
Michael@berber git-example [master] $ cat file-one
My first file that I changed
extension to file 1
Michael@berber git-example [master] $ |
```

# UNSTAGING FILES

```
$ git reset HEAD <file-name>
```

This works when:

1. You accidentally add a file.
2. You decide a particular batch of changes to a file shouldn't be in a commit

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ echo "this will be a mistake" >> file-one
Michael@berber git-example [master] $ echo "this will also be a mistake" >> file-two
Michael@berber git-example [master] $ 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:   file-one
        modified:   file-two

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ git add .
warning: LF will be replaced by CRLF in file-one.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in file-two.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file-one
        modified:   file-two

Michael@berber git-example [master] $ git reset HEAD file-one
Unstaged changes after reset:
M       file-one
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file-two

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:   file-one

Michael@berber git-example [master] $ |
```

# TOUCHING COMMITS

`$ git commit --amend -m "msg"`

Sometimes you want to make a small adjustment to the last commit and you don't want to trigger a whole new commit

*more on this later*

You should not adjust prior commits because it will affect all the ensuing commits (badly)

Or, sometimes you want to adjust a commit message (because it may not be very good or no longer appropriate)

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ 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:   file-one

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master] $ git log --oneline --since="10:00am"
ac59460 (HEAD -> master) adjusted file-two per ticket 7
Michael@berber git-example [master] $ git add file-one
warning: LF will be replaced by CRLF in file-one.
The file will have its original line endings in your working directory.
Michael@berber git-example [master] $ git commit --amend
[master 0c42661] adjusted file-two per ticket 7
  Date: Tue Aug 15 10:50:43 2017 -0400
  2 files changed, 2 insertions(+)
Michael@berber git-example [master] $ git log --oneline --since="10:00am"
0c42661 (HEAD -> master) adjusted file-two per ticket 7
Michael@berber git-example [master] $ git commit --amend -m "this is a much better message"
[master 8ea3fc0] this is a much better message
  Date: Tue Aug 15 10:50:43 2017 -0400
  2 files changed, 2 insertions(+)
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ git log --oneline --since="10:00am"
8ea3fc0 (HEAD -> master) this is a much better message
Michael@berber git-example [master] $ |
```

# REVERTING A FILE FROM A PREVIOUS COMMIT

Suppose you want to reset a file back to a previous commit, the best course of action is to check it out and then commit the change.

- You should preserve the change long so people can see the project's path and reasons for change
  - *Don't hide mistakes. Accept them and move on.*

**\$** `git checkout <commit> <branch> <file>`

Resets working and stages the change.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ cat file-two
a total rewrite
this will also be a mistake
Michael@berber git-example [master] $ git log --oneline
8ea3fc0 (HEAD -> master) this is a much better message
5f43149 second way to move files
c61ed88 example of moving files
0e62bc6 faster removal example
2c0550b example of removing file from repo
7e7435e temporary files for delete example
f61ec6e simple example used for diff
00330e1 Adding a second file as an example
439a3a0 amended first file and added second file per tkt 7
80a5272 commit of example file
Michael@berber git-example [master] $ git show 5f43149:file-two
a total rewrite
Michael@berber git-example [master] $ git checkout 5f43149 -- file-two
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file-two
Michael@berber git-example [master] $ cat file-two
a total rewrite
Michael@berber git-example [master] $ git commit -m "reverting file-two from commit 5f43149 per requirement 6"
[master 5aae012] reverting file-two from commit 5f43149 per requirement 6
 1 file changed, 1 deletion(-)
Michael@berber git-example [master] $ |
```

# REVERTING A PREVIOUS COMMIT

`$ git revert <commit>`

Notice the change does not delete the previous commit; it tracks the change as simply another change, with the effect of “resetting” working, staging, and the repo.

Again, git keeps around the changes long so people can see the project’s path and reasons for change

- Again, don’t hide mistakes. Accept them and move on.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ cat file-two
a total rewrite
Michael@berber git-example [master] $ git log --oneline --since="9am"
5aae012 (HEAD -> master) reverting file-two from commit 5f43149 per requirement 6
8ea3fc0 this is a much better message
5f43149 second way to move files
c61ed88 example of moving files
Michael@berber git-example [master] $ git revert 5aae012
[master 75d0aac] Revert "reverting file-two from commit 5f43149 per requirement 6"
 1 file changed, 1 insertion(+)
Michael@berber git-example [master] $ git log --oneline --since="9am"
75d0aac (HEAD -> master) Revert "reverting file-two from commit 5f43149 per requirement 6"
5aae012 reverting file-two from commit 5f43149 per requirement 6
8ea3fc0 this is a much better message
5f43149 second way to move files
c61ed88 example of moving files
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ |
```



# CHANGING HEAD

---

```
$ git reset --reset-type <commit>
```

Points HEAD to a previous commit. Any new commits will be from that point.

The “later” commits will be lost.

BE CAREFUL USING THIS COMMAND.

## Reset Types

1. `--soft`  
Moves the repo HEAD, but leaves WORKING and STAGING alone (in their current states).
2. `--mixed`  
Moves the repo HEAD and sets STAGING to match the REPO. WORKING remains in its current state (untouched).
3. `--hard`  
Moves the repo HEAD and aligns both WORKING and STAGING to match the repo.

# BRANCHES

---

A branch is string of commits

In all the examples, we've been working on a branch called MASTER.

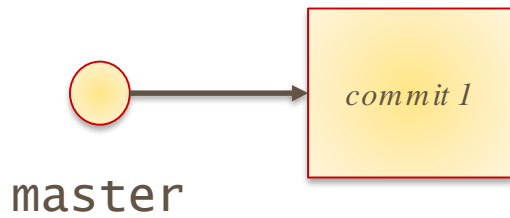
Branching lets you diverge from the main line of development and continue to do work without messing with the main line

- Enables concurrent development
- Enables working per-feature or ticket
- Enables exploration of ideas

Branches are fast, cheap, and easy. You should use branching extensively. You can branch off branches.

# THE BASIC IDEA

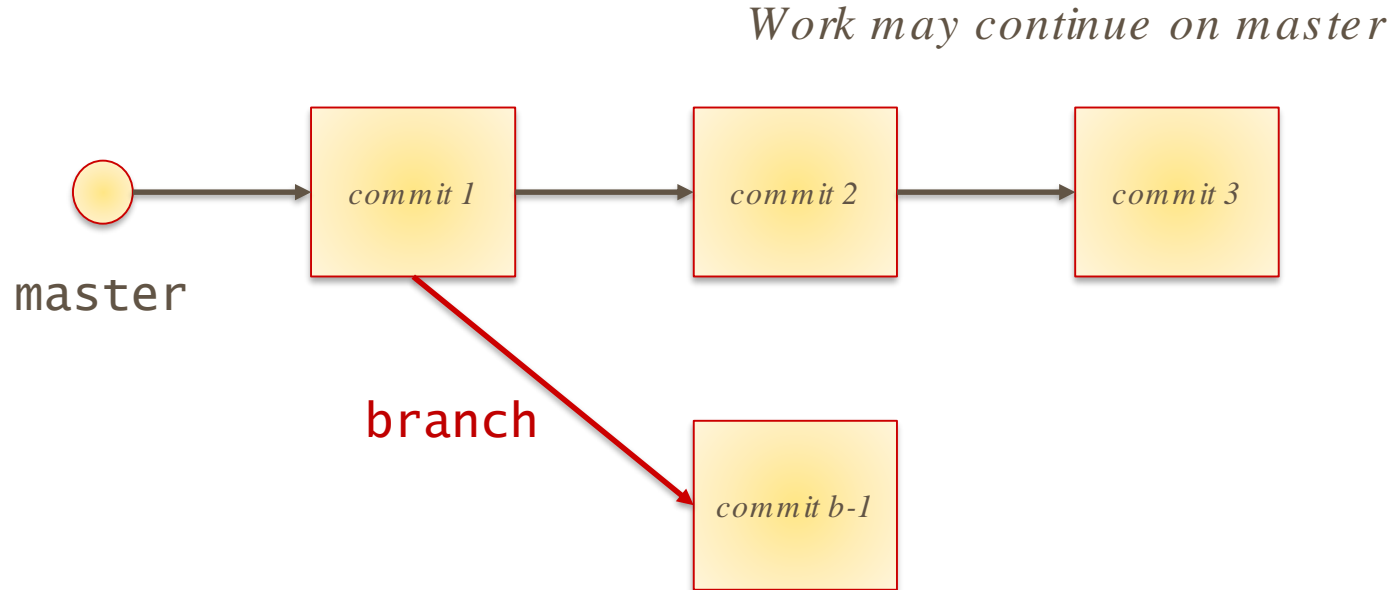
---



*At some point, we want to work on the code as an independent activity – could be a new feature, could be a bug-fix*

# THE BASIC IDEA

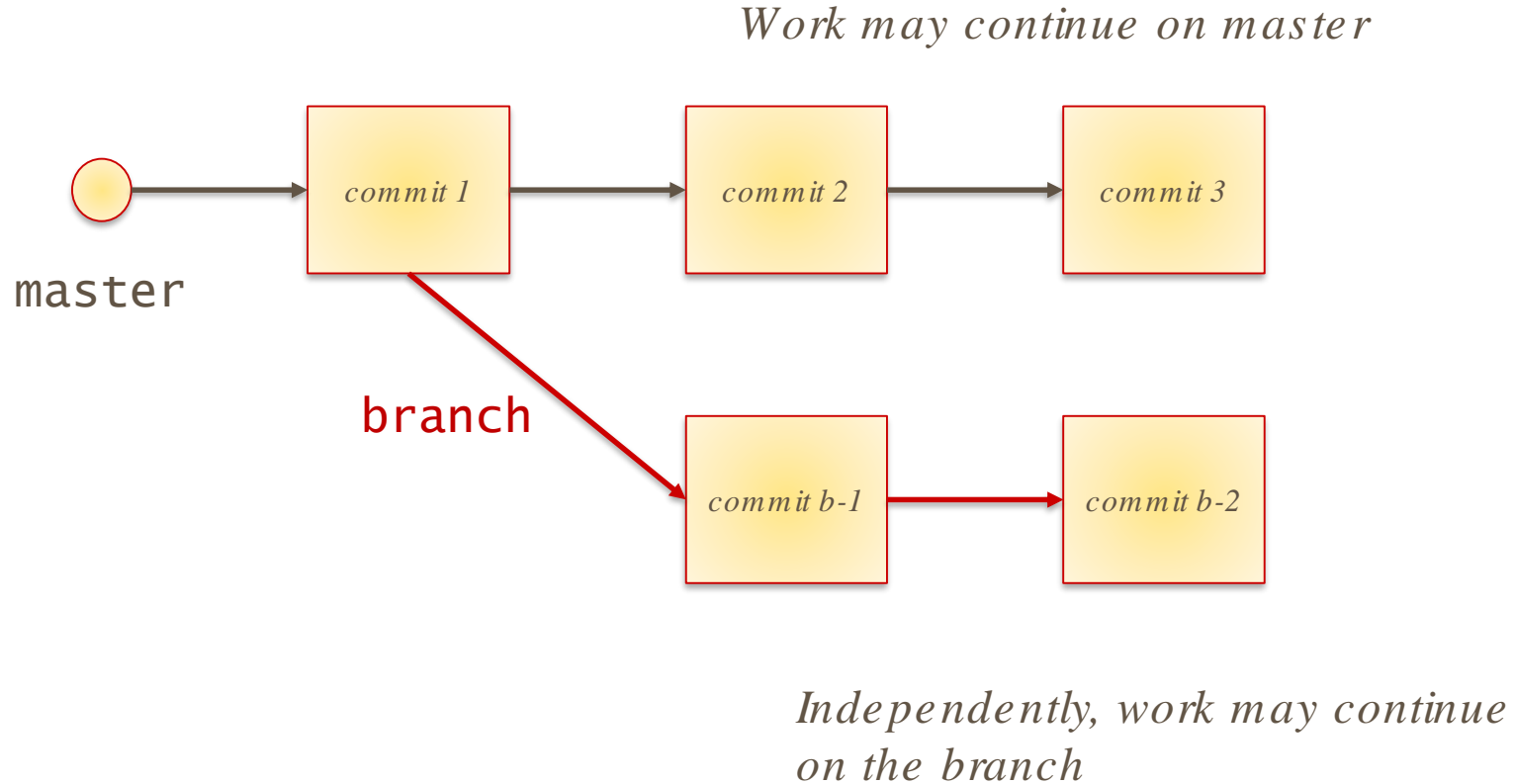
---



*So we create a branch*

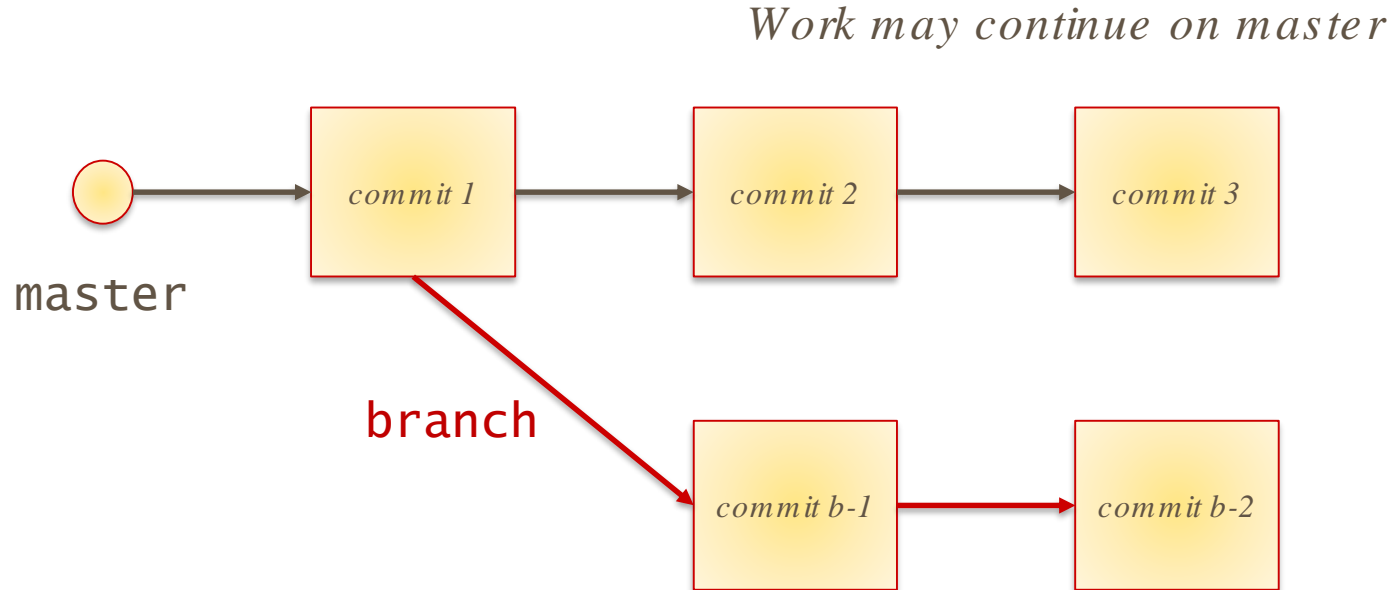
# THE BASIC IDEA

---



# THE BASIC IDEA

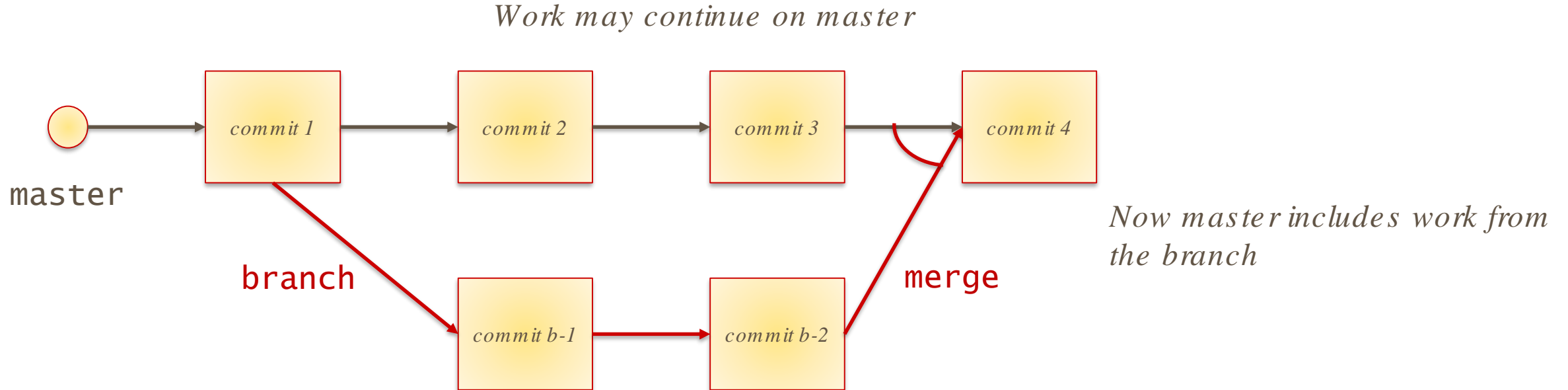
---



*At some point, we've done  
enough on the branch and want to  
merge it with master*

# THE BASIC IDEA

---



# LISTING BRANCHES AND CREATING A BRANCH

---

**\$ git branch**

Lists all branches in a project

**\$ git branch <branch-name>**

Creates a new branch

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
* master
Michael@berber git-example [master] $ git branch demoBranch
Michael@berber git-example [master] $ git branch
demoBranch
* master
Michael@berber git-example [master] $ |
```



# SWITCHING BETWEEN BRANCHES

\$ git checkout <branch-name>

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
demoBranch
* master
Michael@berber git-example [master] $ git checkout demoBranch
Switched to branch 'demoBranch'
Michael@berber git-example [demoBranch] $ echo "a temp file" > temp
Michael@berber git-example [demoBranch] $ git status
On branch demoBranch
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    temp

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber git-example [demoBranch] $ git add .
warning: LF will be replaced by CRLF in temp.
The file will have its original line endings in your working directory.
Michael@berber git-example [demoBranch] $ git status
On branch demoBranch
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   temp

Michael@berber git-example [demoBranch] $ git commit -m "demonstrating branches are separate"
[demoBranch c26d57d] demonstrating branches are separate
1 file changed, 1 insertion(+)
create mode 100644 temp
Michael@berber git-example [demoBranch] $ git log --oneline --since="2pm"
c26d57d (HEAD -> demoBranch) demonstrating branches are separate
db00ba2 (master) adding .gitignore
Michael@berber git-example [demoBranch] $ git checkout master
Switched to branch 'master'
Michael@berber git-example [master] $ git log --oneline --since="9am"
db00ba2 (HEAD -> master) adding .gitignore
75d0aac Revert "reverting file-two from commit 5f43149 per requirement 6"
5a4e012 reverting file-two from commit 5f43149 per requirement 6
8ea3fc0 this is a much better message
5f43149 second way to move files
c61ed88 example of moving files
Michael@berber git-example [master] $ |
```

# SWITCHING BETWEEN BRANCHES

**\$ git checkout <branch-name>**

git will only allow switching branches if there is no conflict between the branches and the current branch's working directory changes are staged!

git tries to keep you from losing work by accident.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
demoBranch
* master
Michael@berber git-example [master] $ git checkout demoBranch
Switched to branch 'demoBranch'
Michael@berber git-example [demoBranch] $ git branch
* demoBranch
master
Michael@berber git-example [demoBranch] $ echo "adding a line" >> temp
Michael@berber git-example [demoBranch] $ git status
On branch demoBranch
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:   temp

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [demoBranch] $ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
temp
Please commit your changes or stash them before you switch branches.
Aborting
Michael@berber git-example [demoBranch] $ |
```

# COMPARING BRANCHES

---

```
$ git diff <branch-1>..<branch-2>
```

Here, git is telling you that **master** does not have file called **temp** and **demoBranch** does.

```
/usr/bin/bash --login -i
Michael@berber git-example [demoBranch] $ git log --oneline --since="2pm"
be40b5d (HEAD -> demoBranch) expanding the temp file
c26d57d demonstrating branches are separate
db00ba2 (master) adding .gitignore
Michael@berber git-example [demoBranch] $ git checkout master
Switched to branch 'master'
Michael@berber git-example [master] $ git log --oneline --since="2pm"
db00ba2 (HEAD -> master) adding .gitignore
Michael@berber git-example [master] $ git diff master..demoBranch
diff --git a/temp b/temp
new file mode 100644
index 0000000..3692c2a
--- /dev/null
+++ b/temp
@@ -0,0 +1,2 @@
+a temp file
+adding a line
Michael@berber git-example [master] $ |
```

# RENAMING BRANCHES

---

\$ git branch --move <current-name> <new-name>

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
demoBranch
* master
Michael@berber git-example [master] $ git branch --move demoBranch moreDescriptiveName
Michael@berber git-example [master] $ git branch
* master
  moreDescriptiveName
Michael@berber git-example [master] $ git checkout moreDescriptiveName
Switched to branch 'moreDescriptiveName'
Michael@berber git-example [moreDescriptiveName] $ git branch --move moreDescriptiveName simplerName
Michael@berber git-example [simplerName] $ git branch
master
* simplerName
Michael@berber git-example [simplerName] $ |
```

# DELETING BRANCHES

\$ git branch --delete <branch-name>

But, git tries to prevent you from doing dumb things

1. You cannot be on the branch you are trying to delete. You must be on a different branch.
2. If the branch to delete contains committed work that has not yet been merged, git will ask you to force it to delete the branch (-D option)

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch firstUp
Michael@berber git-example [master] $ git branch
  firstUp
* master
  simplerName
Michael@berber git-example [master] $ git checkout firstUp
Switched to branch 'firstUp'
Michael@berber git-example [firstUp] $ git branch --delete firstUp
error: Cannot delete branch 'firstUp' checked out at 'C:/Users/Michael/Desktop/Northeastern/projects/git-example'
Michael@berber git-example [firstUp] $ git checkout master
Switched to branch 'master'
Michael@berber git-example [master] $ git branch --delete firstUp
Deleted branch firstUp (was db00ba2).
Michael@berber git-example [master] $ git branch nextUp
Michael@berber git-example [master] $ git checkout nextUp
Switched to branch 'nextUp'
Michael@berber git-example [nextUp] $ echo "adding a line" >> temporary
Michael@berber git-example [nextUp] $ git add .
warning: LF will be replaced by CRLF in temporary.
The file will have its original line endings in your working directory.
Michael@berber git-example [nextUp] $ git commit -m "Losing a file"
[nextUp dc50b8f] Losing a file
1 file changed, 2 insertions(+)
create mode 100644 temporary
Michael@berber git-example [nextUp] $ git checkout master
Switched to branch 'master'
Michael@berber git-example [master] $ git branch --delete nextUp
error: The branch 'nextUp' is not fully merged.
If you are sure you want to delete it, run 'git branch -D nextUp'.
Michael@berber git-example [master] $ git branch -D nextUp
Deleted branch nextUp (was dc50b8f).
Michael@berber git-example [master] $ git branch
* master
  simplerName
Michael@berber git-example [master] $ |
```

# MERGING BRANCHES

---

Brings together two branches

One of two things happen:

1. Things go smoothly
2. There are conflicts
  - *These have to be resolved*

Merges can be tricky.

## Best Practices

1. Start with a clean WORKING directory  
*Insure all commits are done in the receiving branch (where you are merging into).*
2. Keep merges small  
*Merges can be messy and confusing. Small merges are easier to work with than large ones.*
3. Merge often

# MERGING BRANCHES

---

`[$ git checkout <branch-to-receive-merge>`

# be sure to be on the receiving branch]

`$ git merge <branch-to-merge>`

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
* master
  simplerName
Michael@berber git-example [master] $ git diff master..simplerName
diff --git a/temp b/temp
new file mode 100644
index 0000000..3692c2a
--- /dev/null
+++ b/temp
@@ -0,0 +1,2 @@
+a temp file
+adding a line
Michael@berber git-example [master] $ git log --oneline --since="2pm"
db00ba2 (HEAD -> master) adding .gitignore
Michael@berber git-example [master] $ git merge simplerName
Updating db00ba2..be40b5d
Fast-forward
 temp | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 temp
Michael@berber git-example [master] $ git log --oneline --since="2pm"
be40b5d (HEAD -> master, simplerName) expanding the temp file
c26d57d demonstrating branches are separate
db00ba2 adding .gitignore
Michael@berber git-example [master] $ git checkout simplerName
```

# RESOLVING CONFLICTS

If git cannot merge the branches automatically, git marks each issue in each file.

You have two choices:

1. Abort the merge  
\$ git merge --abort
2. Resolve each issue in each file  
manually, add the changes to staging,  
and then commit those changes.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ cat file-one
cs is a great major
Michael@berber git-example [master] $ git checkout simplerName
Switched to branch 'simplerName'
Michael@berber git-example [simplerName] $ cat file-one
cs is an awesome major
Michael@berber git-example [simplerName] $ git checkout master
Switched to branch 'master'
Michael@berber git-example [master] $ git merge simplerName
Auto-merging file-one
CONFLICT (content): Merge conflict in file-one
Automatic merge failed; fix conflicts and then commit the result.
Michael@berber git-example [master|MERGING] $ git merge --abort
Michael@berber git-example [master] $
Michael@berber git-example [master] $ git merge simplerName
Auto-merging file-one
CONFLICT (content): Merge conflict in file-one
Automatic merge failed; fix conflicts and then commit the result.
Michael@berber git-example [master|MERGING] $ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

        both modified:   file-one

no changes added to commit (use "git add" and/or "git commit -a")
Michael@berber git-example [master|MERGING] $ cat file-one
<<<<<< HEAD
cs is a great major
=====
cs is an awesome major
>>>>>> simplerName
Michael@berber git-example [master|MERGING] $ vi file-one
Michael@berber git-example [master|MERGING] $ cat file-one
cs is an awesome major
Michael@berber git-example [master|MERGING] $ git add .
Michael@berber git-example [master|MERGING] $ git commit
[master 0b3e39a] Merge branch 'simplerName'
Michael@berber git-example [master] $ git log --oneline -3
0b3e39a (HEAD -> master) Merge branch 'simplerName'
d21db03 (simplerName) setting up a merge conflict 2
f603653 setting up a merge conflict
Michael@berber git-example [master] $ git branch --merged
* master
  simplerName
Michael@berber git-example [master] $ |
```



# RESOLVING CONFLICT

---

Suppose branch<sub>1</sub> has file<sub>1</sub>, which contains the text: **CS is cool**

Suppose branch<sub>2</sub> has file<sub>1</sub> contains on the same line: **CS is neat**

# RESOLVING CONFLICT

---

Suppose branch<sub>1</sub> has file<sub>1</sub>, which contains the text: **CS is cool**

Suppose branch<sub>2</sub> has file<sub>1</sub> contains on the same line: **CS is neat**

When you merge the two branches, there will be a conflict and git will report it.

# RESOLVING CONFLICT

---

Suppose  $\text{branch}_1$  has  $\text{file}_1$ , which contains the text: **CS is cool**

Suppose  $\text{branch}_2$  has  $\text{file}_1$  contains on the same line: **CS is neat**

When you merge the two branches, there will be a conflict and git will report it.

```
[branch1] $ git merge branch2
```

```
Auto-merging file1
```

```
CONFLICT (add/add): Merge conflict in  
file1
```

```
Automatic merge failed; fix conflicts  
and then commit the result.
```

```
[master|MERGING] $
```

# RESOLVING CONFLICT

---

Suppose branch<sub>1</sub> has file<sub>1</sub>, which contains the text: **CS is cool**

Suppose branch<sub>2</sub> has file<sub>1</sub> contains on the same line: **CS is neat**

If you look at branch<sub>1</sub> file<sub>1</sub>, you will see

```
<<<<<< HEAD
```

```
CS is cool
```

```
=====
```

```
CS is neat
```

```
>>>>>> 1
```

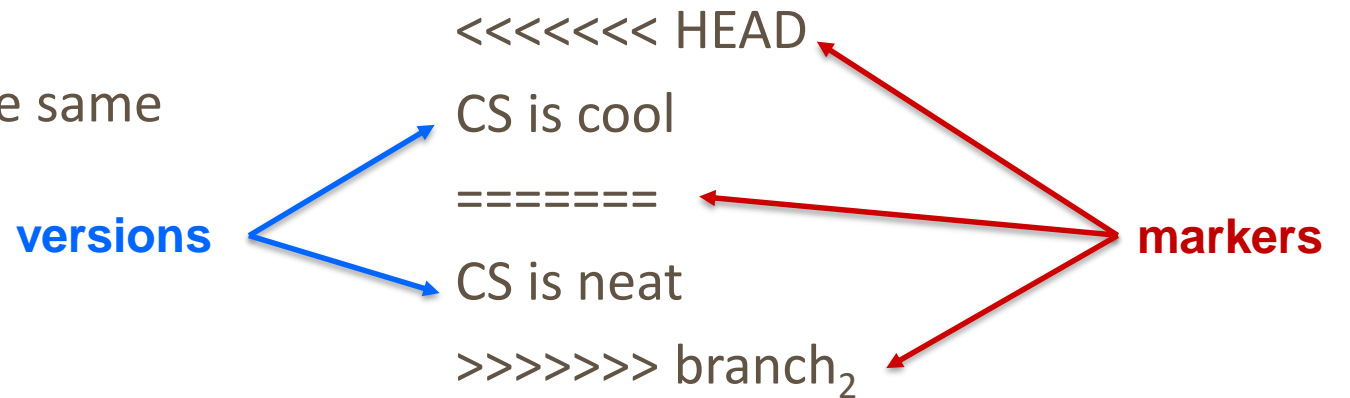
You need to decide which version to keep and get rid of the other.

# RESOLVING CONFLICT

Suppose  $\text{branch}_1$  has  $\text{file}_1$ , which contains the text: **CS is cool**

Suppose  $\text{branch}_2$  has  $\text{file}_1$  contains on the same line: **CS is neat**

If you look at  $\text{branch}_1$   $\text{file}_1$ , you will see



You need to decide which version to keep and get rid of the other.

You edit the file leaving the preferred version and you get rid of the markers.

# RESOLVING CONFLICT

---

Suppose branch<sub>1</sub> has file<sub>1</sub>, which contains the text: **CS is cool**

Suppose branch<sub>2</sub> has file<sub>1</sub> contains on the same line: **CS is neat**

Suppose we prefer the branch<sub>1</sub> version. We edit file<sub>1</sub> and it now looks like:

CS is cool

# RESOLVING CONFLICT

---

Suppose branch<sub>1</sub> has file<sub>1</sub>, which contains the text: **CS is cool**

Suppose branch<sub>2</sub> has file<sub>1</sub> contains on the same line: **CS is neat**

Now we have to finish the merge by staging and committing the final version.

# RESOLVING CONFLICT

---

Suppose  $\text{branch}_1$  has  $\text{file}_1$ , which contains the text: **CS is cool**

Suppose  $\text{branch}_2$  has  $\text{file}_1$  contains on the same line: **CS is neat**

Now we have to finish the merge by staging and committing the final version.

```
[branch1 | MERGING] $ git status
```

On branch  $\text{branch}_1$

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths:

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

**both added:**  $\text{file}_1$

```
[branch1 | MERGING] $ git add .
```

```
[branch1 | MERGING] $ git commit
```

```
[master 7a9074b] Merge branch 'branch2'
```

```
[branch1] $
```



# RESOLVING CONFLICT

---

Suppose  $\text{branch}_1$  has  $\text{file}_1$ , which contains the text: **CS is cool**

Suppose  $\text{branch}_2$  has  $\text{file}_1$  contains on the same line: **CS is neat**

The merge is now complete

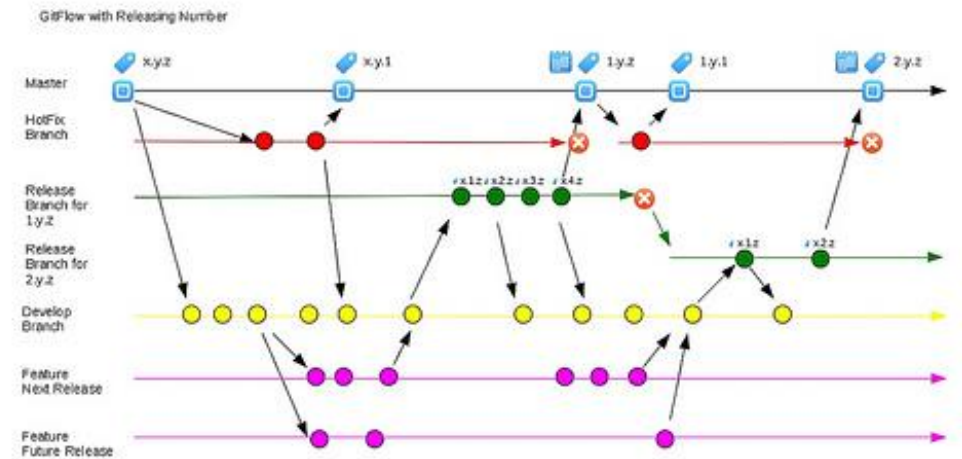
```
[branch1] $ git log --oneline
7a9074b (HEAD -> branch1) Merge branch 'branch2'
[...]
289b7d1 (branch1) first commit
```

# WORKING IN GROUPS

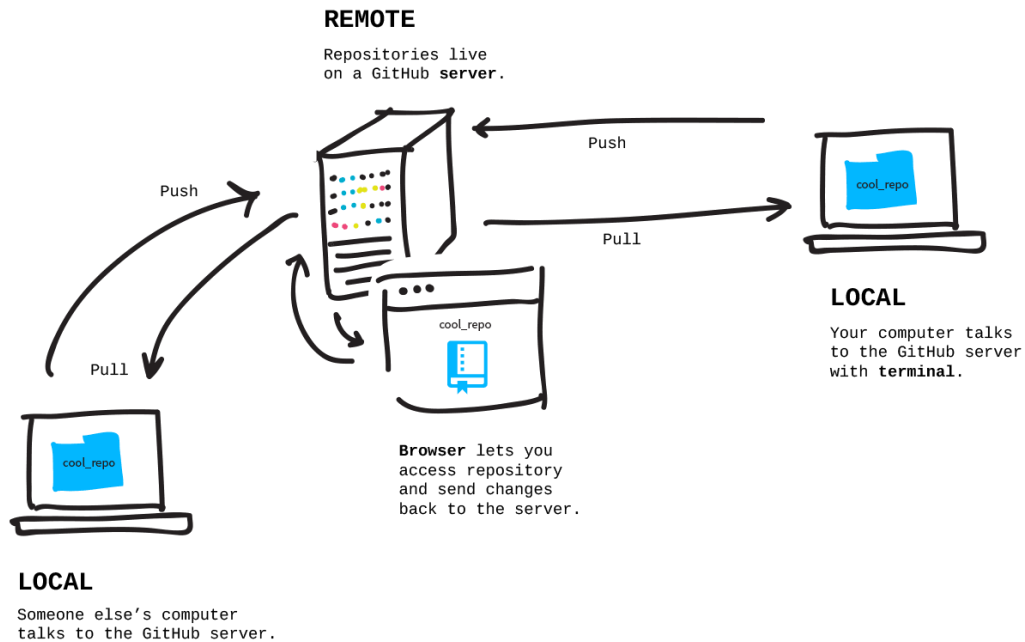
Usually the repository you clone is the center of work in your project.

Typically, a common repository service is used and master at this site (origin) is the gold copy.

Developers work on branches on their own machines. When someone finishes something, the work needs to be incorporated into the broader project.



# STAYING IN SYNC



With all these people working independently, change is happening all the time. The slides so far cover how to do your own work.

What we need now is to share your work and to incorporate other people's work.

In git, sharing means you have to **push** your work so others can get it and **pull** in other people's work.

From: <https://duzun.me/tips/git>

# WORKING WITH REMOTE REPO'S

---

A typical project strategy is to make a central host the “origin” for the project.

Team members:

1. Clone the remote repository
2. Do work in a personal environment, synchronizing with the origin as appropriate
3. Propose updates to the origin
4. Accept or reject these updates (a role usually reserved for a special few or *at least not the proposer*). Accepted changes are merged into the origin

■ Example hosts:

- <https://github.ccs.neu.edu/>  
*(we will work here)*
- <https://github.com/>
- <https://bitbucket.org/>

# CLONING A PROJECT


---

Typically, projects are established at the origin first or you join an established project.

In this case, you clone (copy) the project to your machine.

**\$** `git clone <url>`

Notice in the example that git creates a subdirectory in directory where git clone was run

```
 /usr/bin/bash --login -i
Michael@berber projects $ git clone https://github.ccs.neu.edu/software-dev/example.git
Cloning into 'example'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Michael@berber projects $ ls -a
./ ../ example/ git-example/
Michael@berber projects $ cd example
Michael@berber example [master] $ ls -a
./ ../ .git/ README.md
Michael@berber example [master] $ |
```

# MORE REMOTE REPOSITORY BASICS

---

**\$ git remote --verbose**

Lists all remote repo's (the verbose option give the full url. Otherwise, git will report aliases).

**\$ git remote rm <alias>**

Remove a connection to a remote repository

```
Michael@berber course [master] $ git remote --verbose
origin https://github.ccs.neu.edu/cs5500/course.git (fetch)
origin https://github.ccs.neu.edu/cs5500/course.git (push)
Michael@berber course [master] $ git remote
origin
Michael@berber course [master] $ |
```

# PUBLISHING YOUR WORK

```
$ git push -u <alias> <branch>
```

*If you omit the alias or branch,*

```
$ git push
```

*git will push the current  
branch to the remote  
repository.*

```
/usr/bin/bash --login -i
Michael@berber example [master] $ echo "adding a line" >> first-file
Michael@berber example [master] $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    first-file

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber example [master] $ git add .
warning: LF will be replaced by CRLF in first-file.
The file will have its original line endings in your working directory.
Michael@berber example [master] $ git commit -m "Kick off the project"
[master 963cd99] Kick off the project
 1 file changed, 1 insertion(+)
 create mode 100644 first-file
Michael@berber example [master] $ git push -u origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 291 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.ccs.neu.edu/software-dev/example.git
 d770aba..963cd99  master -> master
Branch master set up to track remote branch master from origin.
Michael@berber example [master] $ |
```

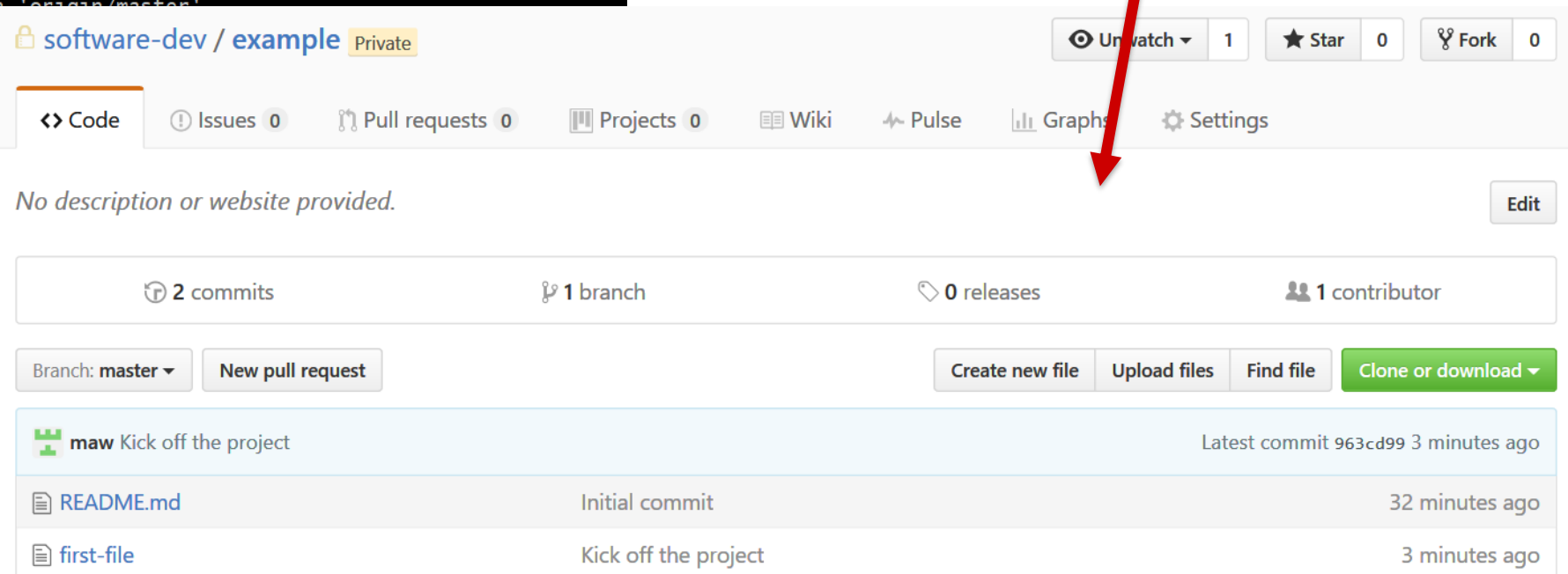
# THE RESULT OF THE PUSH – THE COMMIT IS ON THE REMOTE

```
$ git push -u <alias> <branch>
```

*Remote repository (github.ccs)  
as seen in the browser*

```
/usr/bin/bash --login -i
Michael@berber example [master] $ echo "adding a line" >> first-file
Michael@berber example [master] $ git status
On branch master
Your branch is up-to-date with 'origin/master'
Untracked files:
  (use "git add <file>..." to add to the commit)
        first-file

nothing added to commit but untracked files present (use "git add" to track)
Michael@berber example [master] $ git push -u origin master
warning: LF will be replaced by CRLF in first-file.
The file will have its original line endings.
[master 963cd99] Kick off the project
1 file changed, 1 insertion(+)
 create mode 100644 first-file
Michael@berber example [master] $ git push -u origin master
Counting objects: 3, done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 1.0 KiB | 1.0 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.ccs.neu.edu:~maw/d770aba/.963cd99 master -> 963cd99
Branch master set up to track origin/master.
Michael@berber example [master] $
```





# WHAT HAPPENS IF TWO PEOPLE PUSH TO ORIGIN AT THE SAME TIME

---

If there are no conflicts, it's not a problem

If there are conflicts, it's a problem. git will help and reject the second push.



# WORKING ON SEPARATE BRANCHES AVOIDS THE PROBLEM

---

You push your branch; I push mine. We merge and settle the differences.

Well, it defers the problem to when the branches are merged.

But we know how to handle this (`git merge`).



# STAYING IN SYNC WITH THE REMOTE REPO

---

Conceptually, there are two steps:

1. Get the changes from the remote
2. Integrate these changes into my environment

# STAYING IN SYNC WITH THE REMOTE REPO

---

Conceptually, there are two steps:

1. Get the changes from the remote
2. Integrate these changes into my environment

```
$ git pull
```

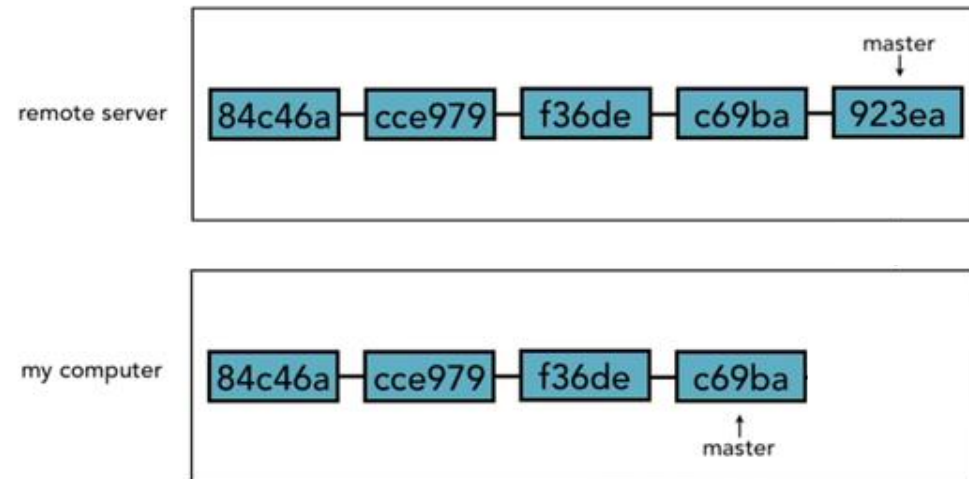
git will fetch the differences and then try to merge them into your branch.

You have to resolve any conflicts .

# PULL MECHANICS ILLUSTRATED

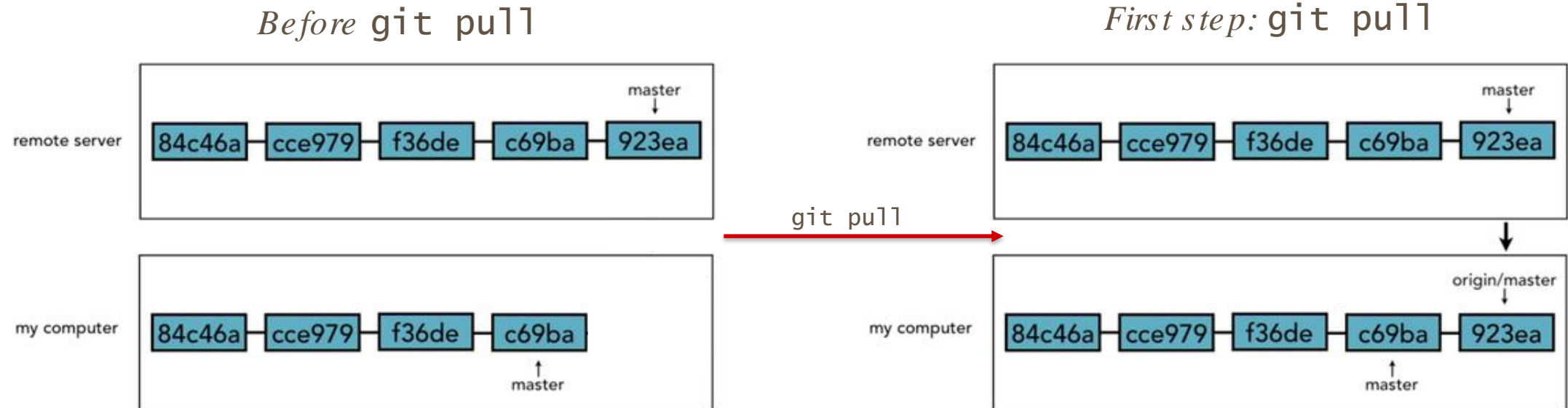
---

*Before git pull*



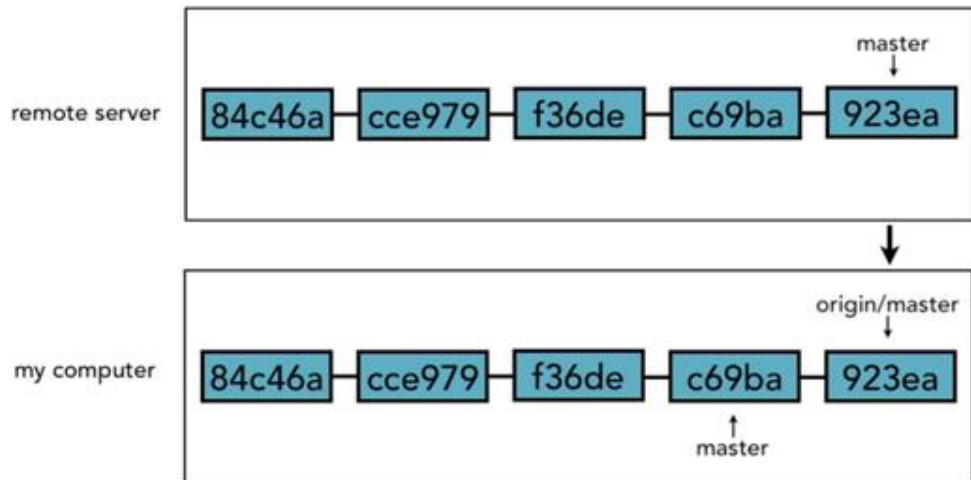
Adapted from <http://archive.fabacademy.org/archives/2016/doc/gitCheatSheet.html>

# PULL MECHANICS ILLUSTRATED

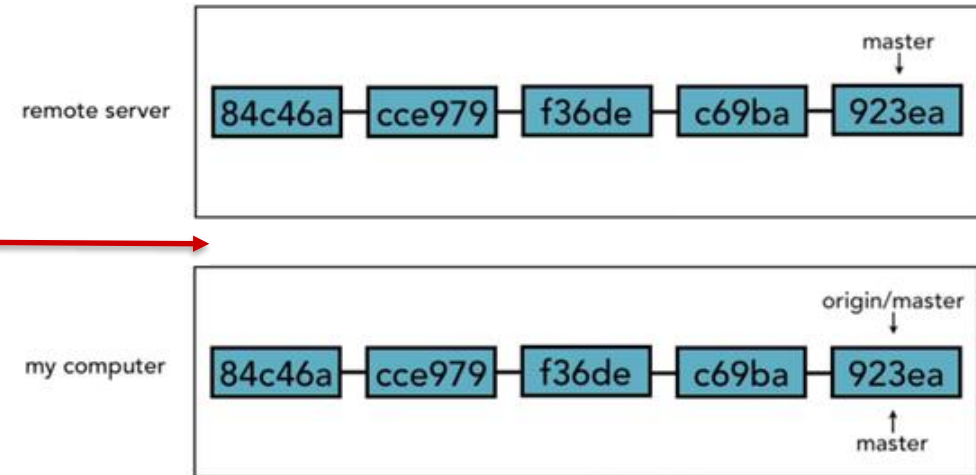


# PULL MECHANICS ILLUSTRATED

*First step: fetch from master*



*Second step: merge the two*



# WORKING AS A GROUP

---

You now have all basics needed to use git in your project.

But, you need a game plan for how to work git together.

Remember, stepping on each other's work  
is a big concern





# RULES OF THE ROAD

---



Rule #1: You never, ever push directly to master

Rule #2: You always push to a working branch

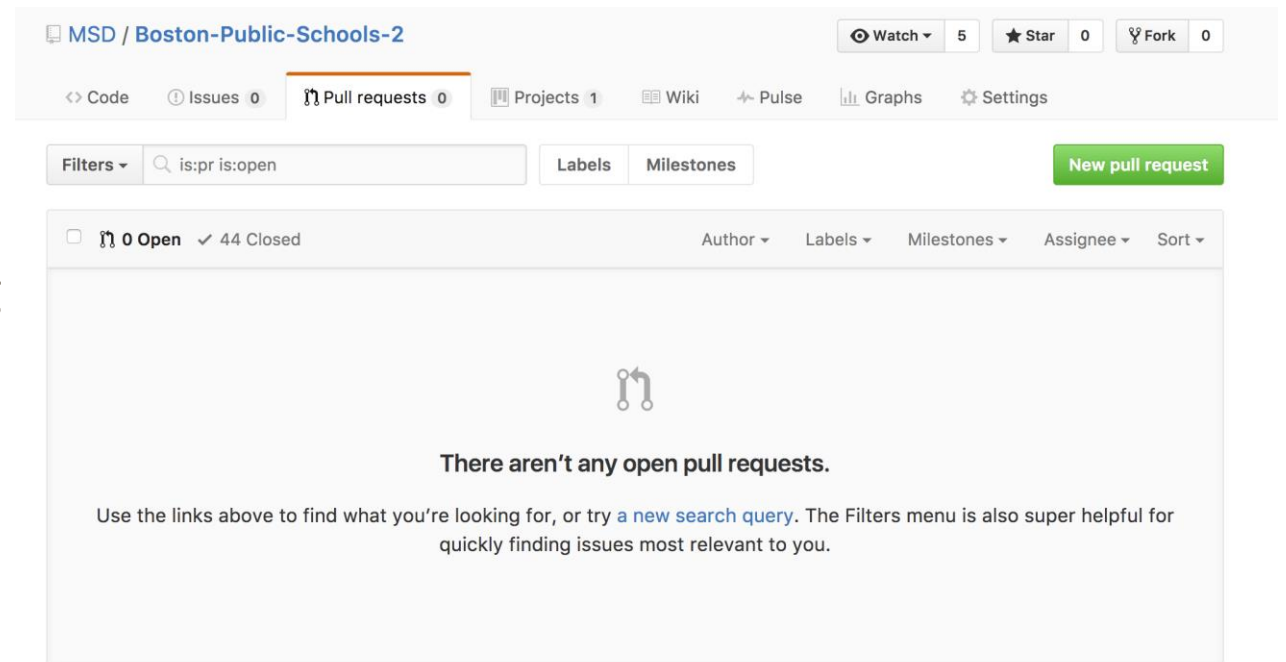
Rule #3: We control merging your branch with master very tightly and require permission to merge into master

Rule #4: Only good code is merged (passes the tests)

# HOW THIS WORKS, REALLY

Assuming you set up git and clone'd the repo

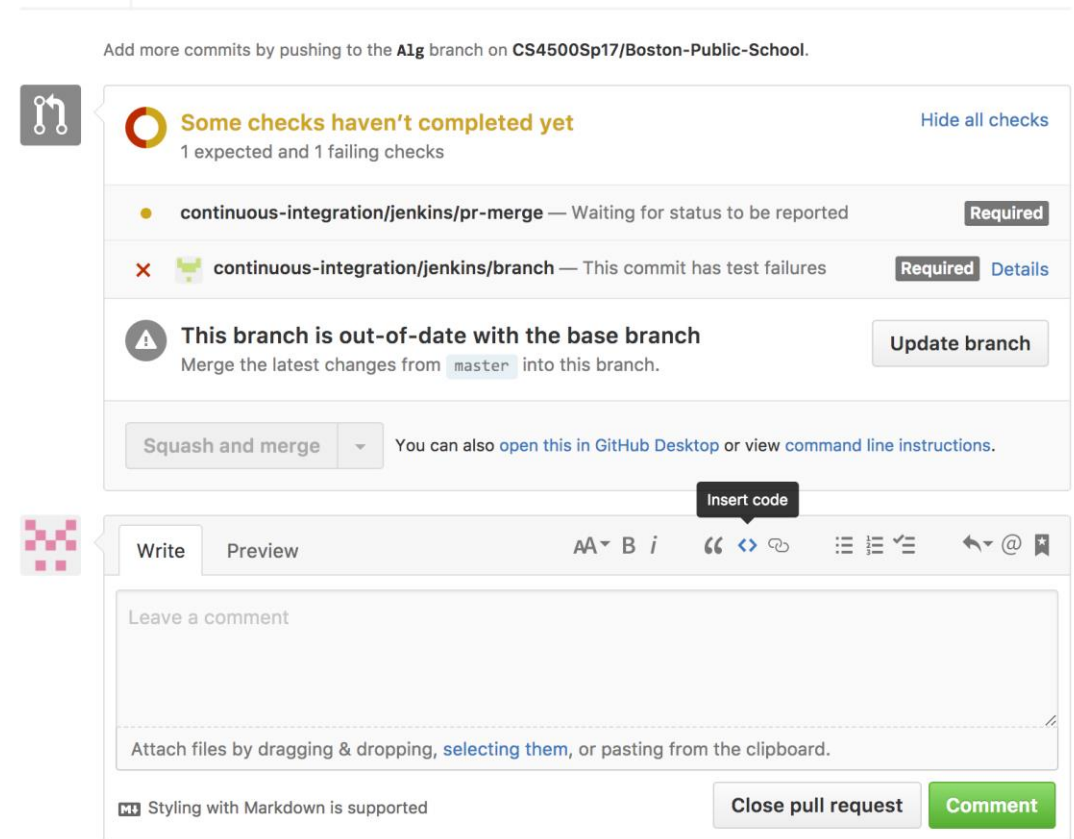
1. `git branch <your-branch>`
2. `git checkout <your-branch>`
3. # your work goes here
4. `git add .` # move your changes to staging
5. `git commit -m "a good message"`
6. `git pull`
  - `git merge` # if any updates, repeat (5)
7. `git push`
8. Open github in a browser and go to the Pull requests tab



💡 ProTip! Ears burning? Get @grobalex mentions with [mentions:grobalex](#).

# EXERCISING THIS WORKFLOW – PULL REQUESTS

8. Create a pull request and select your branch.
9. The pull request will then show up
  - a. Will kick off a continuous integration run
  - b. Requires approval by another team member.  
(in open source, the repo owner;  
in industry, the project lead – often)
10. Two outcomes
  - a. Once all those tests pass: you can squash and merge.
  - b. (tests don't pass): you close the pull request without merging.



## In case of fire



1. `git commit`



2. `git push`



3. leave building

# BACKUP

---

# DELETING FILES

## (IN ONE STEP USING GIT)

```
$ git rm <filename>
```

In this case, git will both remove the file and mark it as a change. You then only need to **commit** the change.

### WARNING

**git rm** uses a command line delete, which means the file is not recoverable - i.e., it's really gone and not in a trash folder.

```
Michael@berber git-example [master] $ ls
file1.txt file2.txt temp2
Michael@berber git-example [master] $ git rm temp2
rm 'temp2'
Michael@berber git-example [master] $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    temp2

Michael@berber git-example [master] $ git commit -m "faster removal example"
[master 0e62bc6] faster removal example
 1 file changed, 1 deletion(-)
 delete mode 100644 temp2
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $
```

# CONNECTING TO A REMOTE REPOSITORY

---

**\$ git remote --verbose**

Lists all remote repo's (the verbose option give the full url. Otherwise, git will report aliases.

**\$ git remote add <alias> <url>**

Add a remote repo at location <url> and call it <alias>

**\$ git remote rm <alias>**

Remove a connection to a remote repo's

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git remote
Michael@berber git-example [master] $ git remote add origin https://github.ccs.neu.edu/software-dev/example.git
Michael@berber git-example [master] $ git remote
origin
Michael@berber git-example [master] $ git remote rm origin
Michael@berber git-example [master] $ |
```

# SYNCHRONIZING WITH THE REMOTE REPO

```
$ git fetch <alias>
```

This pulls down the latest version from the remote repo.

It does not integrate it with your local repo, but it looks like a branch (origin/<branch-name>).

Here, a second user to the project fetches the repo.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
* master
  simplerName
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ cat file-one
cs is an awesome major
Michael@berber git-example [master] $ ls
file-one file-two temp templ
Michael@berber git-example [master] $ git remote add origin https://github.ccs.neu.edu/software-dev/example.git
Michael@berber git-example [master] $ git fetch origin
warning: no common commits
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 8 (delta 0), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (8/8), done.
From https://github.ccs.neu.edu/software-dev/example
 * [new branch]   master    -> origin/master
 * [new branch]   model     -> origin/model
Michael@berber git-example [master] $ git branch
* master
  simplerName
Michael@berber git-example [master] $ git branch -r
  origin/master
  origin/model
Michael@berber git-example [master] $ git log --oneline -4 origin/master
963cd99 (origin/master) Kick off the project
d770aba Initial commit
Michael@berber git-example [master] $ git log --oneline -4 master
0b3e39a (HEAD -> master) Merge branch 'simplerName'
d21db03 (simplerName) setting up a merge conflict 2
f603653 setting up a merge conflict
ae40b5d expanding the temp file
Michael@berber git-example [master] $ |
```



# PUSHING CHANGES TO THE REMOTE REPO

---

```
$ git push <alias> <local-branch>
```

This pushes local-branch to the remote repo.

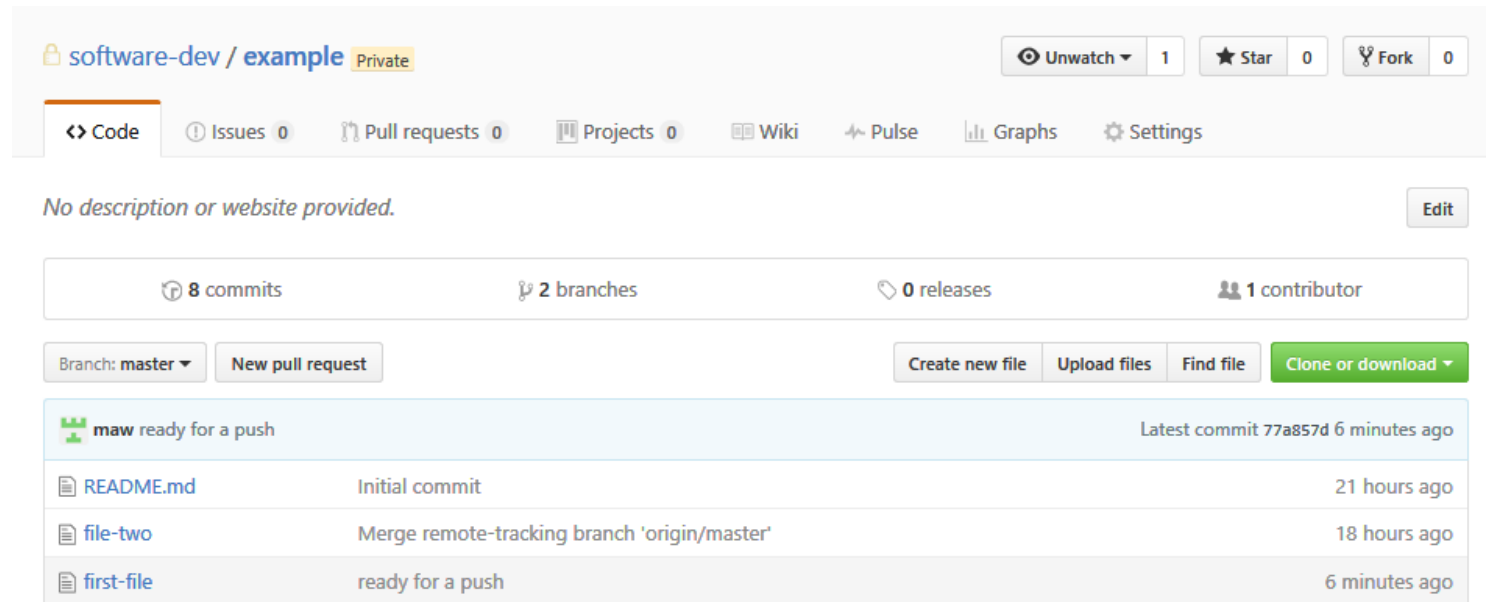
Here, my master pushed commit 77a857d (HEAD) to origin.

```
Michael@berber ~ $ cd Desktop/Northeastern/projects/
Michael@berber projects $ ls
example/  other-guy/
Michael@berber projects $ cd example
Michael@berber example [master] $ ls
file-two  first-file  README.md
Michael@berber example [master] $ git diff master..origin/master
Michael@berber example [master] $ echo "yet another line" >> first-file
Michael@berber example [master] $ git add .
warning: LF will be replaced by CRLF in first-file.
The file will have its original line endings in your working directory.
Michael@berber example [master] $ git commit -m "ready for a push"
[master 77a857d] ready for a push
1 file changed, 1 insertion(+)
Michael@berber example [master] $ git log --oneline -4
77a857d (HEAD -> master) ready for a push
978a79b (origin/master, origin/HEAD) Merge remote-tracking branch 'origin/master'
d8fedd2 yo!
e09966c adding spain into the mix
Michael@berber example [master] $ git log --oneline -3 origin/master
978a79b (origin/master, origin/HEAD) Merge remote-tracking branch 'origin/master'
d8fedd2 yo!
e09966c adding spain into the mix
Michael@berber example [master] $ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 273 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local objects.
To https://github.ccs.neu.edu/software-dev/example.git
   978a79b..77a857d  master -> master
Michael@berber example [master] $ |
```

# PUSHING CHANGES TO THE REMOTE REPO

```
$ git push <alias> <local-branch>
```

This pushes my current branch to the remote repo.



The screenshot shows a GitHub repository page for 'software-dev / example' (Private). The repository has 1 star, 0 forks, and 0 issues. The 'Code' tab is selected, showing a table of commits. The latest commit is '77a857d' from user 'maw' 6 minutes ago, with the message 'ready for a push'. The table lists three files: 'README.md' (Initial commit, 21 hours ago), 'file-two' (Merge remote-tracking branch 'origin/master', 18 hours ago), and 'first-file' (ready for a push, 6 minutes ago).

File	Commit Message	Time
README.md	Initial commit	21 hours ago
file-two	Merge remote-tracking branch 'origin/master'	18 hours ago
first-file	ready for a push	6 minutes ago

# STAYING IN SYNC WITH THE REMOTE REPO

```
$ git fetch <alias>
```

This pulls down the latest version from the remote repo.

It does not integrate it with your local repo, but it looks like a branch (origin/<branch-name>).

Here, a second user to the project fetches the repo.

```
/usr/bin/bash --login -i
Michael@berber git-example [master] $ git branch
* master
  simplerName
Michael@berber git-example [master] $ git status
On branch master
nothing to commit, working tree clean
Michael@berber git-example [master] $ cat file-one
cs is an awesome major
Michael@berber git-example [master] $ ls
file-one file-two temp templ
Michael@berber git-example [master] $ git remote add origin https://github.ccs.neu.edu/software-dev/example.git
Michael@berber git-example [master] $ git fetch origin
warning: no common commits
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 8 (delta 0), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (8/8), done.
From https://github.ccs.neu.edu/software-dev/example
 * [new branch]      master    -> origin/master
 * [new branch]      model    -> origin/model
Michael@berber git-example [master] $ git branch
* master
  simplerName
Michael@berber git-example [master] $ git branch -r
  origin/master
  origin/model
Michael@berber git-example [master] $ git log --oneline -4 origin/master
963cd99 (origin/master) Kick off the project
d770aba Initial commit
Michael@berber git-example [master] $ git log --oneline -4 master
0b3e39a (HEAD -> master) Merge branch 'simplerName'
d21db03 (simplerName) setting up a merge conflict 2
f603653 setting up a merge conflict
0e40b5d expanding the temp file
Michael@berber git-example [master] $ |
```

# INTEGRATION-MANAGER WORKFLOW

---

This scenario includes a canonical repository that represents the “official” project. To contribute to that project, you create your own public clone of the project and push your changes to it. Then, you send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows:

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor’s repo as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.