Neueda Technologies

# Working with Git

---

## Contents

| | |
|---|---|
| Chapter 1 | Introduction to Git |
| Chapter 2 | Branching with Git |
| Chapter 3 | Working with Remote Repositories |
| Chapter 4 | Working with Pull Requests |
| | Lab Exercises |

neueda

# Getting Started with Git

## Contents

1. Introduction to version control systems
2. Overview of Git
3. Using the Git Bash shell
4. Adding new files to a Git repository
5. Modifying, staging, and committing files
6. Additional techniques

## Some Setup

- Open slides pdf on remote desktop so you can follow along
- Download the using_git.zip file
- Extract the contents to C:\
- Verify that you have a folder: C:\UsingGit\

## 1. Introduction to Version Control Systems

- The need for version control systems
- Types of version control systems
- Overview of Git

## The Need For Version Control Systems

- Characteristics of a typical software development project
  - Multiple developers…
  - Potentially working at different locations
  - Working on the same software artifacts at the same time
- To work successfully together, we need to be able to…
  - Share code files
  - Detect conflicting changes introduced by different developers
  - Revert to previous versions of files, on occasion
  - Create new branches of development, e.g. for prototyping or to support variations of a project
- Version control systems provide all these features

## Types of Version Control Systems

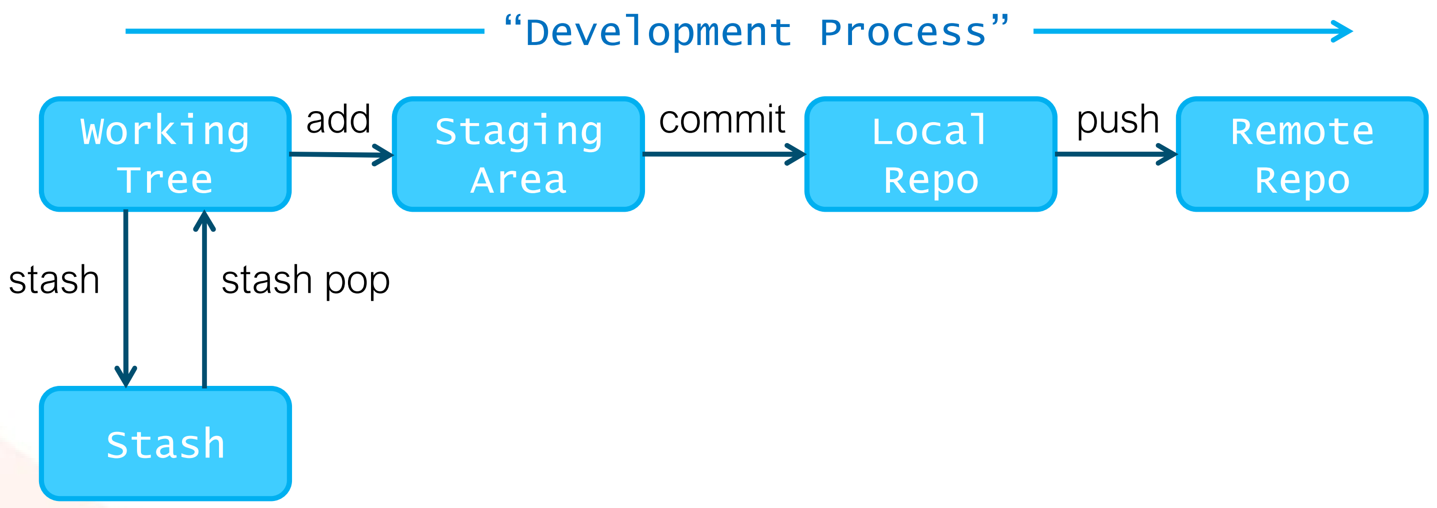
- Local VCS (e.g. Mac OSX res command)
  - Developers have a local database on their machine
  - Pro: Simplicity
  - Cons: No collaboration facilities
- Centralized VCS (e.g. CVS, SVN, Perforce)
  - Developers check in/out files from/to a centralized server
  - Pros: Easy to collaborate with other developers
  - Cons: Single point of failure, plus the need for connectivity
- Distributed VCS (e.g. Git, Mercurial, Bazaar)
  - Developers mirror a complete repository onto their local machine
  - Pros: No need for connectivity after you've cloned a repository
  - Cons: Takes a while to get used to how they work :-}

## 2. Overview of Git

- Git is a distributed VCS
  - Created in 2005 by Linus Torvalds, the creator of Linux
- Characteristics of Git
  - Fully distributed
  - Extremely simple and quick to create new branches
  - Uses file snapshots rather than file deltas (see following slides)
- Key concepts
  - Local and remote repositories
  - Cloning repositories
  - Fetching and pushing updates

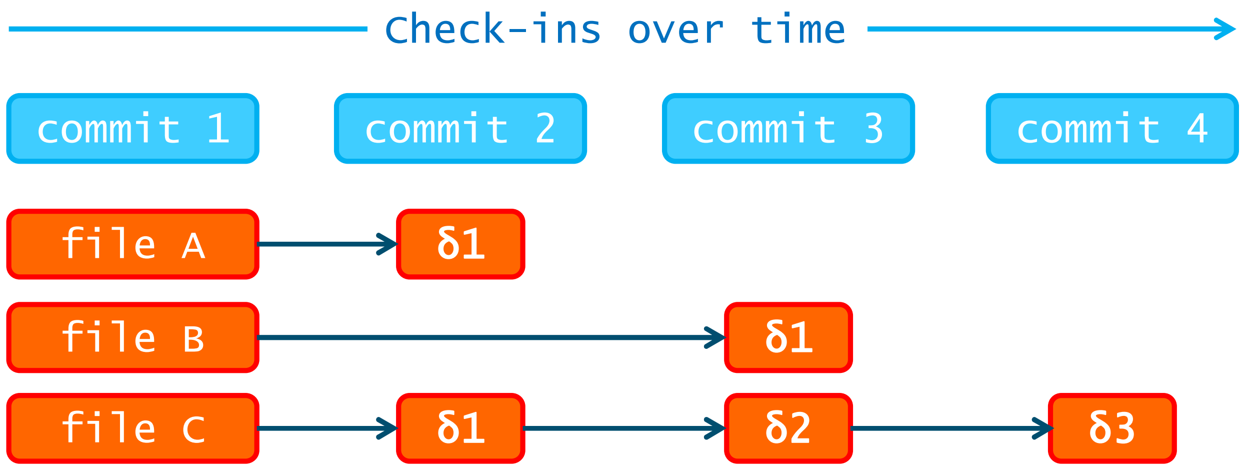
## Git Areas

- As you make changes to files, you use git commands to move those changes to different areas



## Most VCS are based on File Deltas

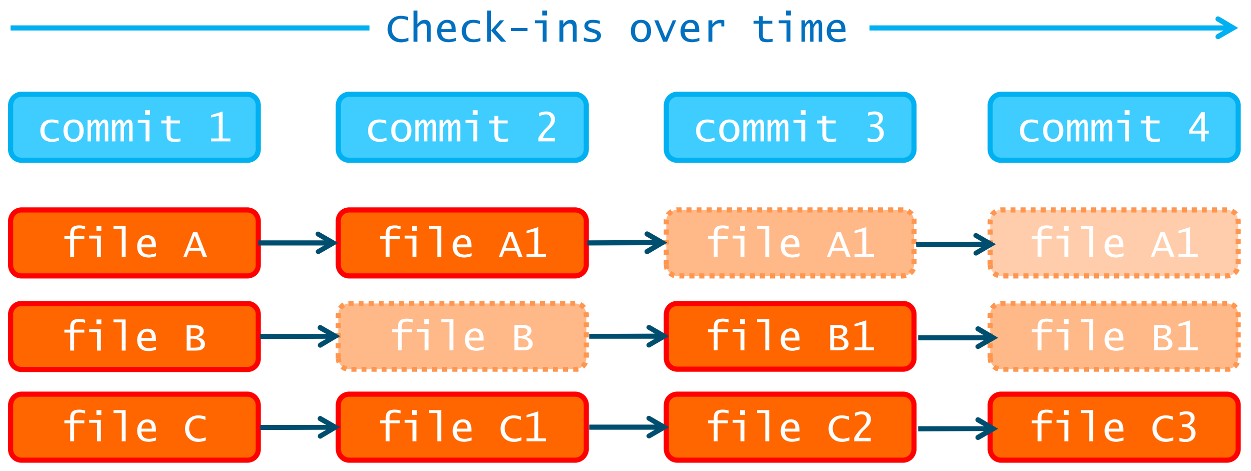
- Most VCS use file deltas to identify file changes across multiple check-ins over time



- On each commit, VCS stores deltas for modified files
  - If a file hasn't changed, the VCS stores nothing for that file

## Git is based on File Snapshots

- Git doesn't use deltas to track file differences - instead, each commit has a snapshot of all the files at that time



- On each commit, Git stores snapshots for all files
  - If a file hasn't changed, Git stores a link to the previous snapshot

## Installing Git

- Installing Git on Linux
- Installing Git on Mac
- Installing Git on Windows

## Installing Git on Linux

- To install Git via a binary installer on Linux…
  - If you're using a system that uses Yum (e.g. Fedora):

```
yum install git
```

  - If you're using a system that uses Debian (e.g. Ubuntu):

```
apt-get install git
```

## Installing Git on Mac

- To install Git via a binary installer on Mac…
  - You can download a graphical installer from here:

```
http://sourceforge.net/projects/git-osx-installer/
```

  - Or if you have MacPorts installed, you can install Git as follows:

```
sudo port install git +svn +doc +bash_completion +gitweb
```

  - Or if you have Homebrew installed, you can install Git as follows:

```
brew install git
```

## Installing Git on Windows

- To install Git via a binary installer on Windows…
  - You can download an installer program from here:

```
https://gitforwindows.org/
```

- Run the installer

## 3. Using the Git Bash Shell

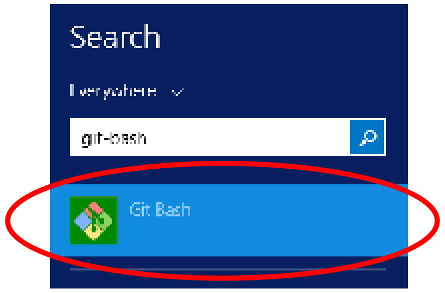
- Overview
- Starting the Git Bash shell
- About the Git Bash shell
- Performing one-off Git configuration
- Checking your configuration
- Getting help

## Overview

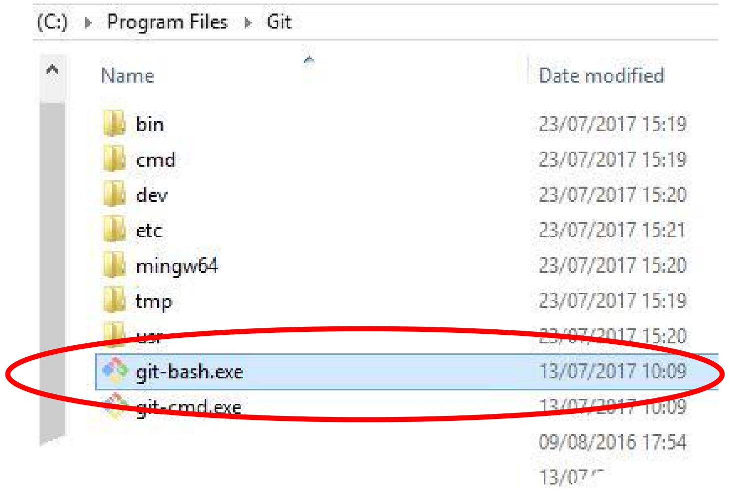
- In this section we'll see how to perform some simple Git operations
  - We'll be using Windows
- You're advised to use the Git Bash shell, which is provided with the Git download and already installed on your VMs
  - Better than using the normal Windows Command Prompt
  - The Git Bash shell handles special characters better (e.g. ' " /)
  - Easier for entering complex commands
- See following slide for how to start the Git Bash shell

## Starting the Git Bash Shell

- You can start the Git Bash shell via the "Windows" button



- Or you can start it from the Git installation folder

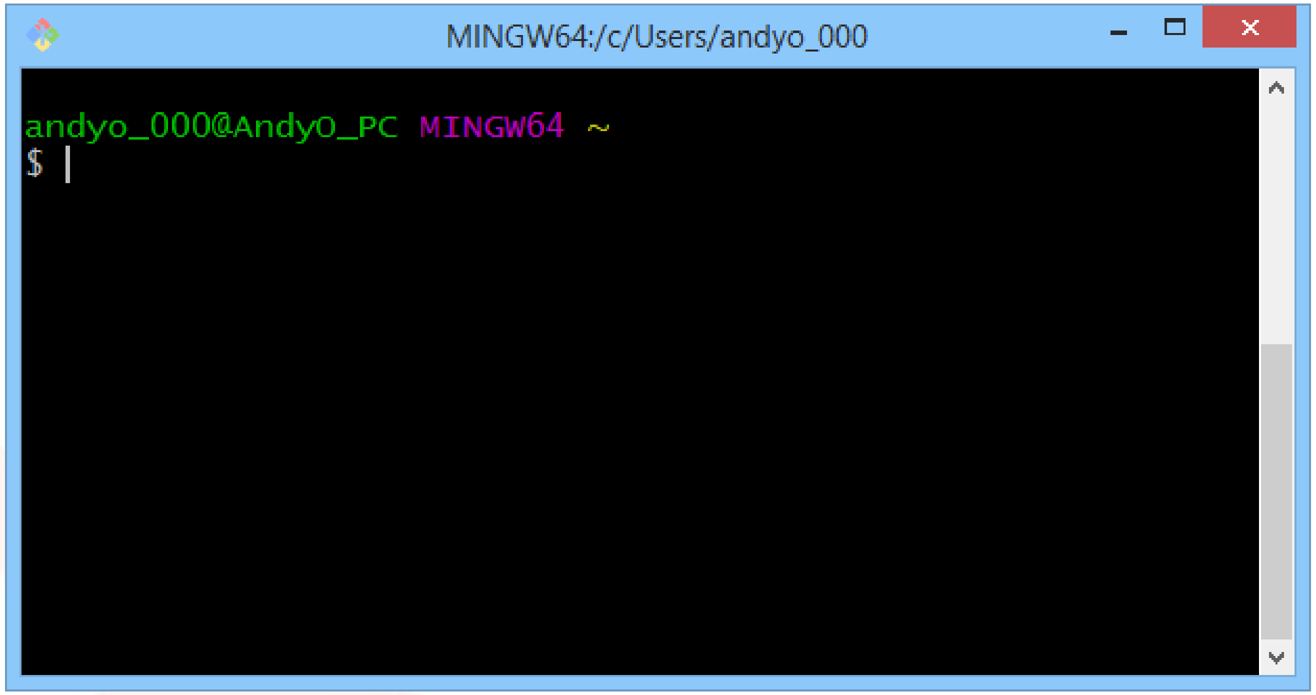


| Name | Date modified |
|---|---|
| bin | 23/07/2017 15:19 |
| cmd | 23/07/2017 15:19 |
| dev | 23/07/2017 15:20 |
| etc | 23/07/2017 15:21 |
| mingw64 | 23/07/2017 15:20 |
| tmp | 23/07/2017 15:19 |
| usr | 23/07/2017 15:20 |
| git-bash.exe | 13/07/2017 10:09 |
| git-cmd.exe | 13/07/2017 10:09 |
| | 09/08/2016 17:54 |
| | 13/07 |

neueda

## About the Git Bash Shell

- The Git Bash shell is where you'll enter your Git commands
  - E.g. clone repository
  - E.g. add files to a repository
  - E.g. commit changes



```
andyo_000@AndyO_PC MINGW64 ~
$ 
```

## Performing One-Off Git Configuration

- When you've installed Git on your system, you should perform some simple one-off configuration:
  - E.g. your user name and email address

- You do this via the Git `config` command
  - Specify the `--global` option, so Git will use these details for all operations you do on your system

```
git config --global user.name "Andy Olsen"
```

```
git config --global user.email andyo@olsensoft.com
```

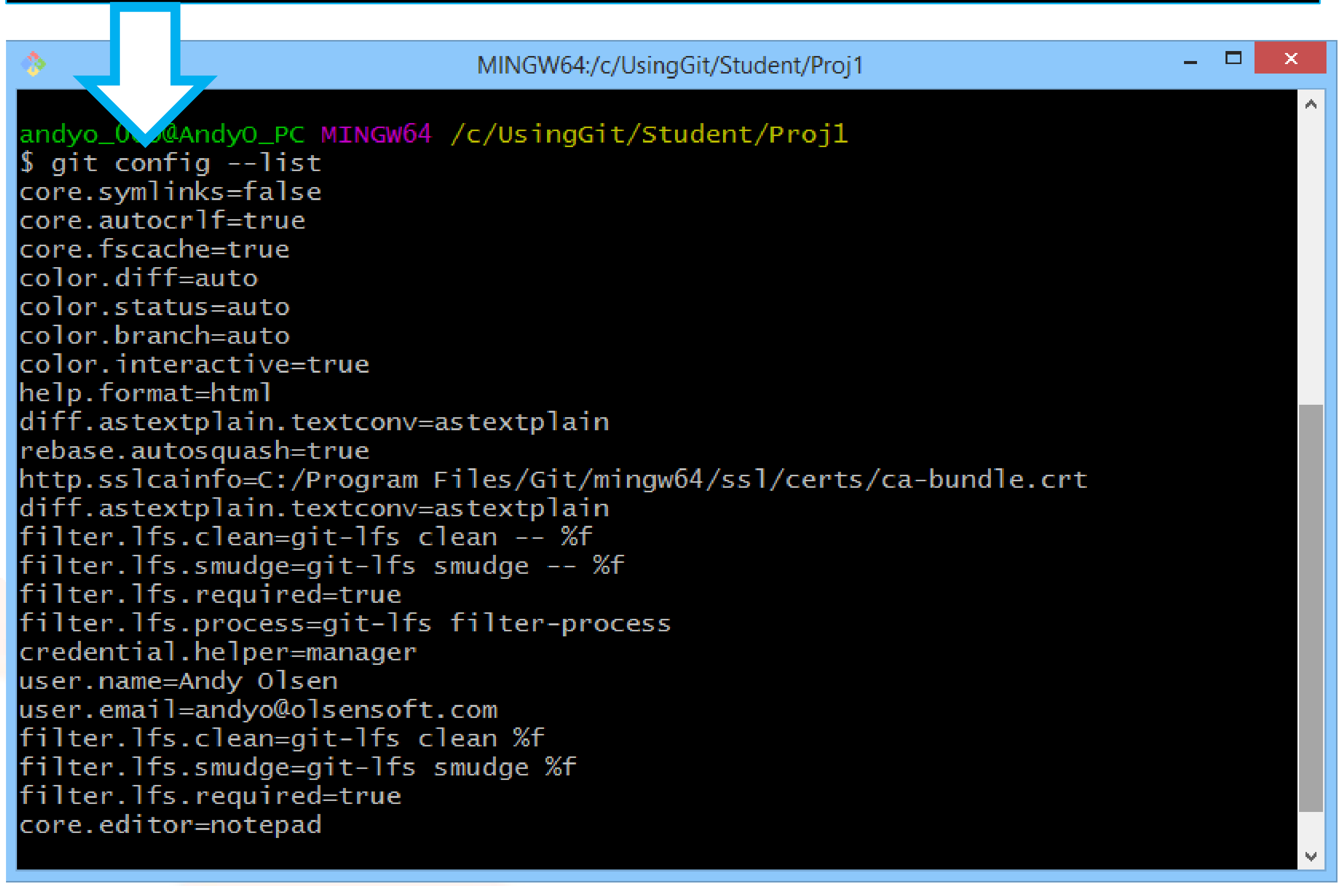
```
git config --global core.editor notepad
```

- On Windows, Git stores your config info in this file:

```
<user-folder>\.gitconfig
```

## Checking your Configuration

```
git config --list
```



```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
diff.astextplain.textconv=astextplain
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.required=true
filter.lfs.process=git-lfs filter-process
credential.helper=manager
user.name=Andy Olsen
user.email=andyo@olsensoft.com
filter.lfs.clean=git-lfs clean %f
filter.lfs.smudge=git-lfs smudge %f
filter.lfs.required=true
core.editor=notepad
```

## Getting Help

- There are a couple of ways to get help for a Git command:

```
git help <command>
```

```
git <command> --help
```

- E.g. to get help on the `config` command:

```
git help config
```

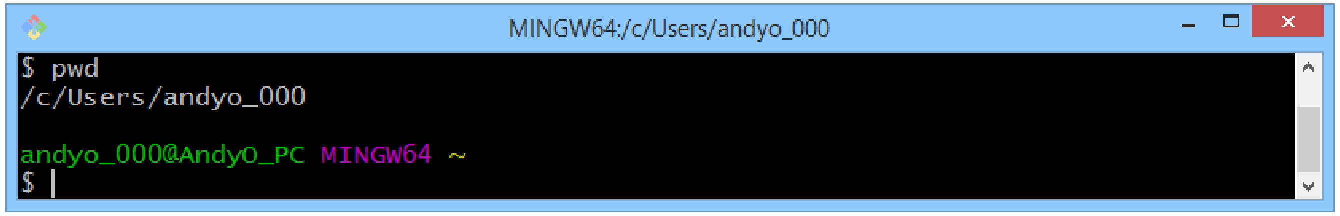
```
git config --help
```

## 4. Adding New Files to a Git Repository

- Navigating the local file system
- Creating a new Git repository
- Viewing the status of files
- Adding a file to the Git repository
- Committing files

## Navigating the Local File System

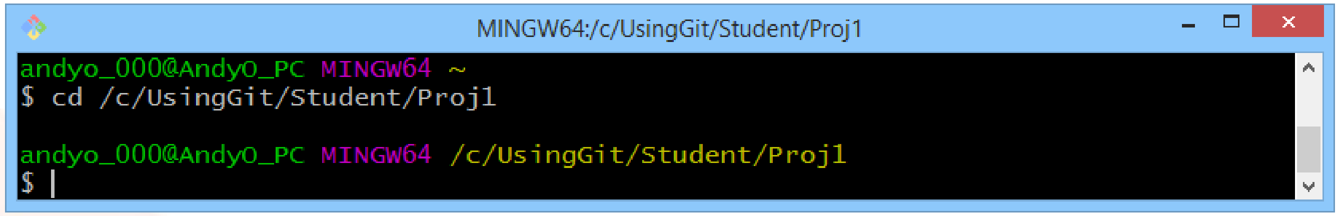
- The Git Bash shell initially puts you in your 'user' directory
  - You can verify this via pwd



MINGW64:/c/Users/andyo_000

```
$ pwd
/c/Users/andyo_000

andyo_000@AndyO_PC MINGW64 ~
$ |
```

- Change directory to `C:\UsingGit\Student\Proj1`
  - This will be the working directory for the demos
  - i.e. the directory where you'll edit files etc.



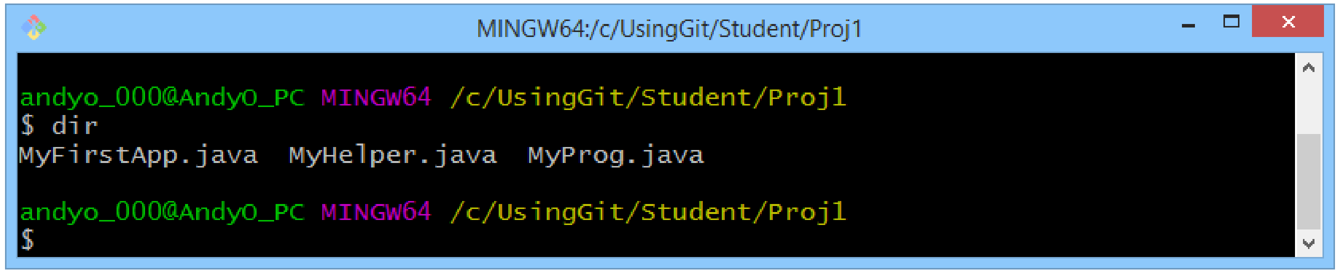
MINGW64:/c/UsingGit/Student/Proj1

```
andyo_000@AndyO_PC MINGW64 ~
$ cd /c/UsingGit/Student/Proj1

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1
$ |
```

neueda

## Creating a New Git Repository (1 of 2)

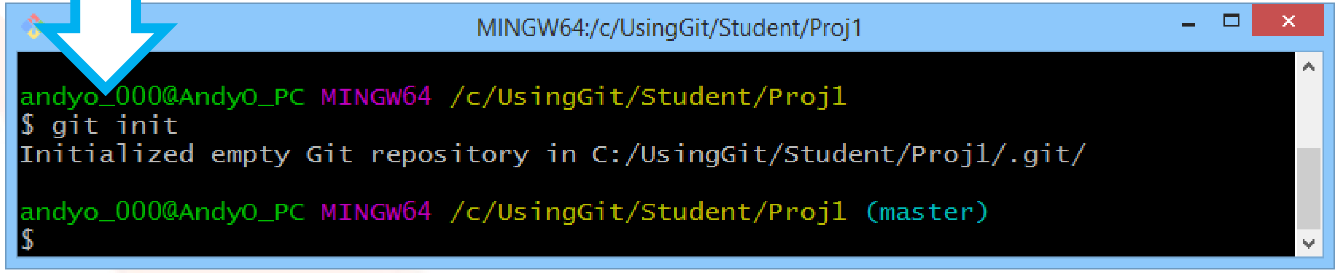
- Git knows nothing about files in the working directory yet
  - The working directory just contains some simple Java files



```
MINGW64:/c/UsingGit/Student/Proj1

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1
$ dir
MyFirstApp.java  MyHelper.java  MyProg.java

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1
$
```

- To tell Git to start tracking the working directory…

```
git init
```

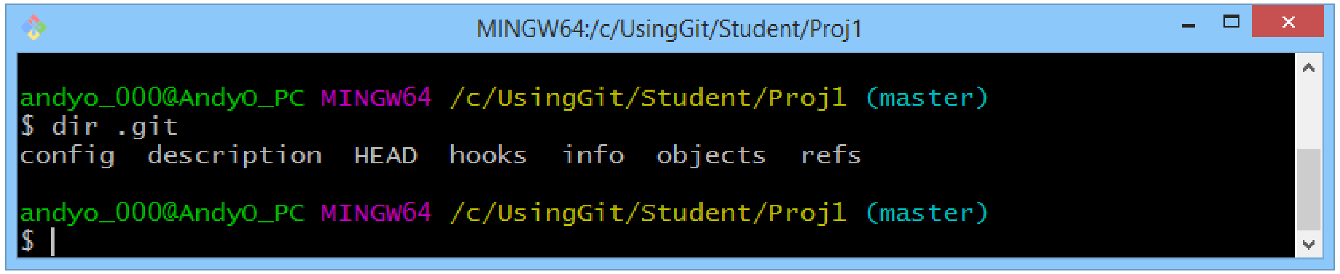


```
MINGW64:/c/UsingGit/Student/Proj1

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1
$ git init
Initialized empty Git repository in C:/UsingGit/Student/Proj1/.git/

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$
```

neueda

## Creating a New Git Repository (2 of 2)

- When you run `git init`:
  - Git creates a new subdirectory named `.git`
  - This is a Git repository (a.k.a. Git directory)
  - It's where Git stores metadata about version history and file status for all the tracked files in the working directory



MINGW64:/c/UsingGit/Student/Proj1

```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ dir .git
config  description  HEAD  hooks  info  objects  refs

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ |
```

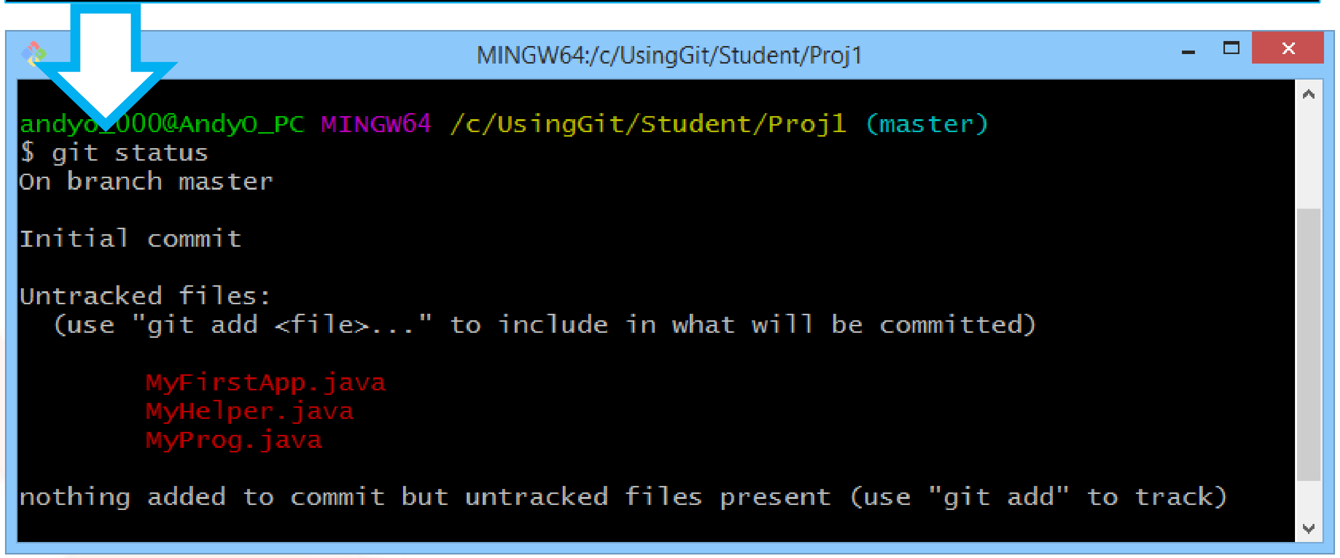
neueda

## Viewing the Status of Files

- Files in your working directory can be in one of two states:
  - Tracked (i.e. Git knows about them)
  - Untracked (i.e. Git doesn't know about them)

- To view the Git status of files in your working directory:

```
git status
```



```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git status
On branch master

Initial commit

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

        MyFirstApp.java
        MyHelper.java
        MyProg.java

nothing added to commit but untracked files present (use "git add" to track)
```

## Adding a File to the Git Repository (1 of 3)

- To tell Git to start tracking a file:

```
git add MyFirstApp.java
```

- A tracked file is in one of the following 3 states:

**Unmodified**

You haven't modified the file since you originally downloading it from the server

→

**Modified**

You've modified the file since downloading it, but you plan to make some additional changes too

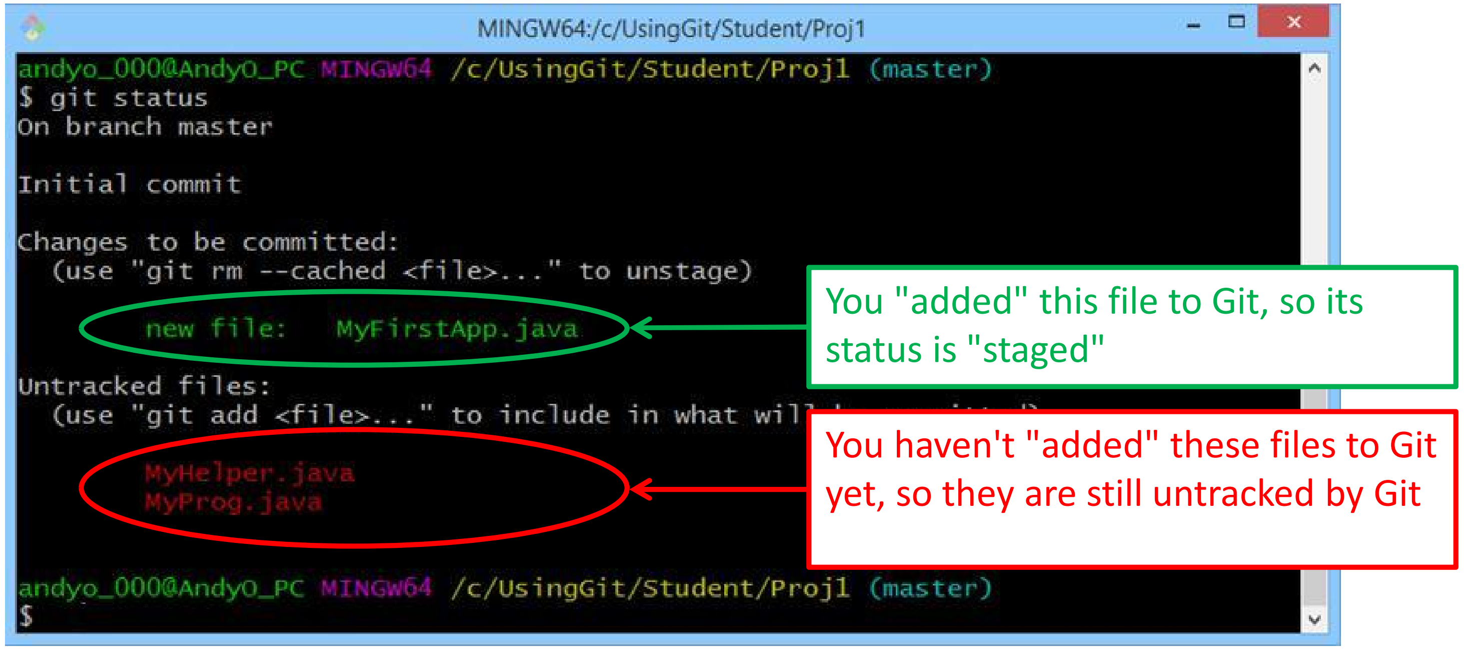
→

**Staged**

You've finished making all the changes you plan to make, and you're ready to commit the file

## Adding a File to the Git Repository (2 of 3)

- Here's the current status of files in the working directory



MINGW64:/c/UsingGit/Student/Proj1

```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git status
On branch master

Initial commit

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

        new file:   MyFirstApp.java

Untracked files:
  (use "git add <file>..." to include in what wil

        MyHelper.java
        MyProg.java

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$
```

You "added" this file to Git, so its status is "staged"

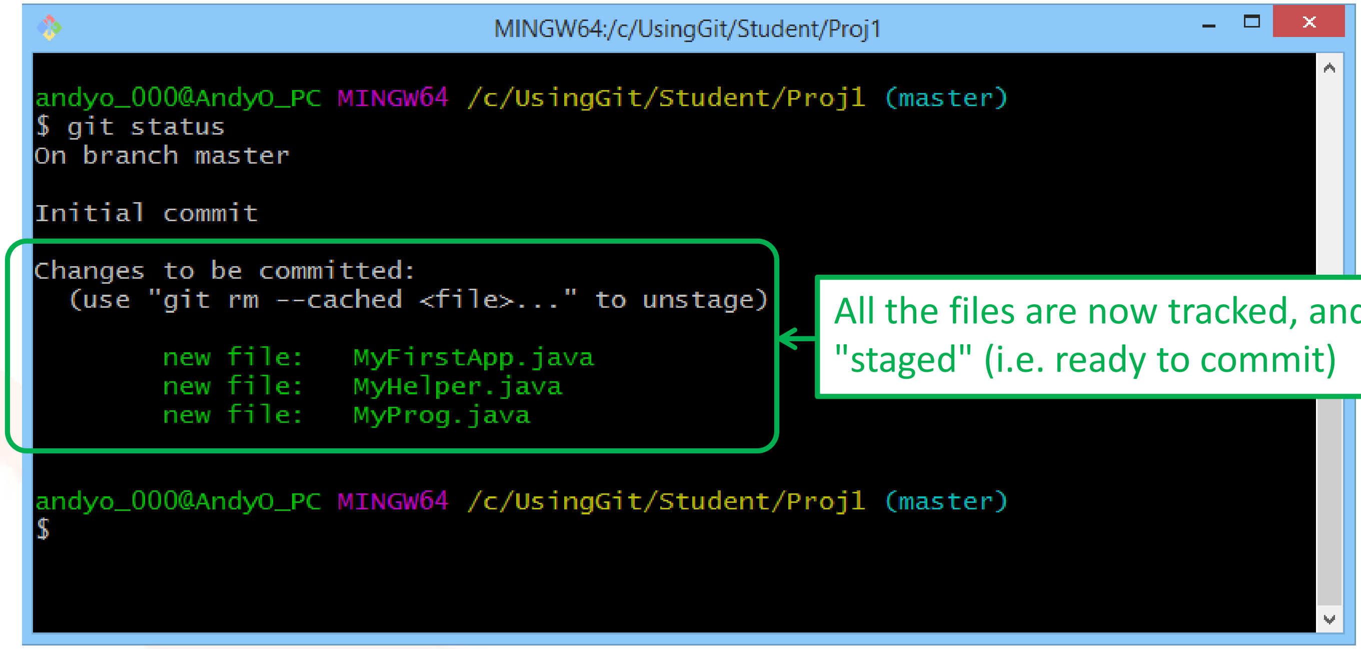
You haven't "added" these files to Git yet, so they are still untracked by Git

## Adding a File to the Git Repository (3 of 3)

- Let's add the other files to the Git repository now

```
git add MyHelper.java MyProg.java
```

- Here's the status of files in the working directory now:



```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git status
On branch master

Initial commit

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

        new file:   MyFirstApp.java
        new file:   MyHelper.java
        new file:   MyProg.java


andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$
```

All the files are now tracked, and "staged" (i.e. ready to commit)

## Committing Files (1 of 3)

- The "staging area" specifies all the files that are ready to be committed to the Git repository

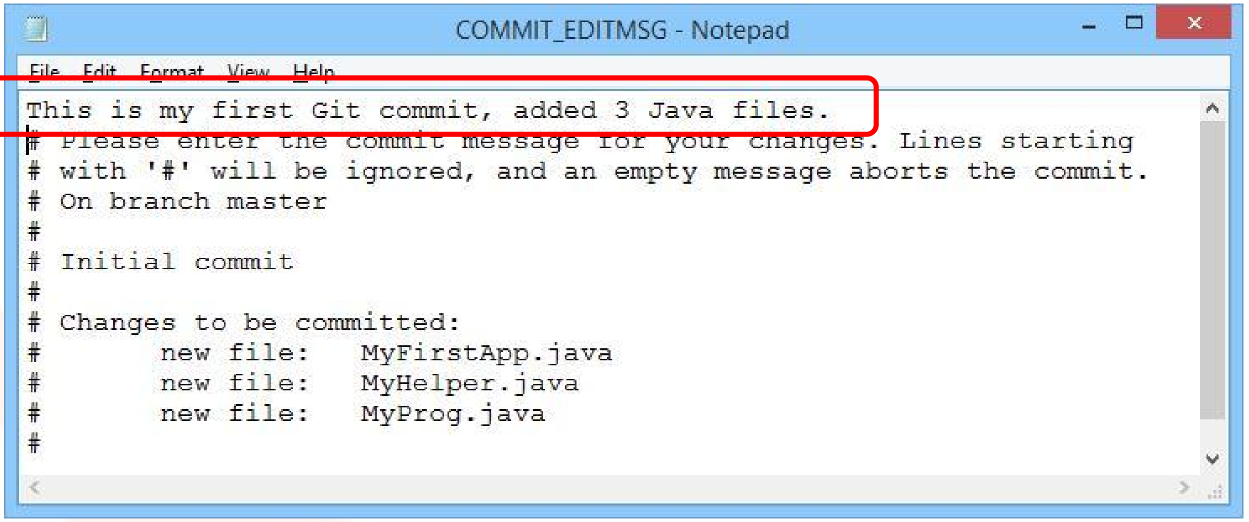
- When you're ready to commit these changes to Git, this is how you do it:

```
git commit
```

## Committing Files (2 of 3)

- When you do a commit, Git opens the registered editor so you can enter a 'commit message'
  - Recall you set the Git editor to be Notepad

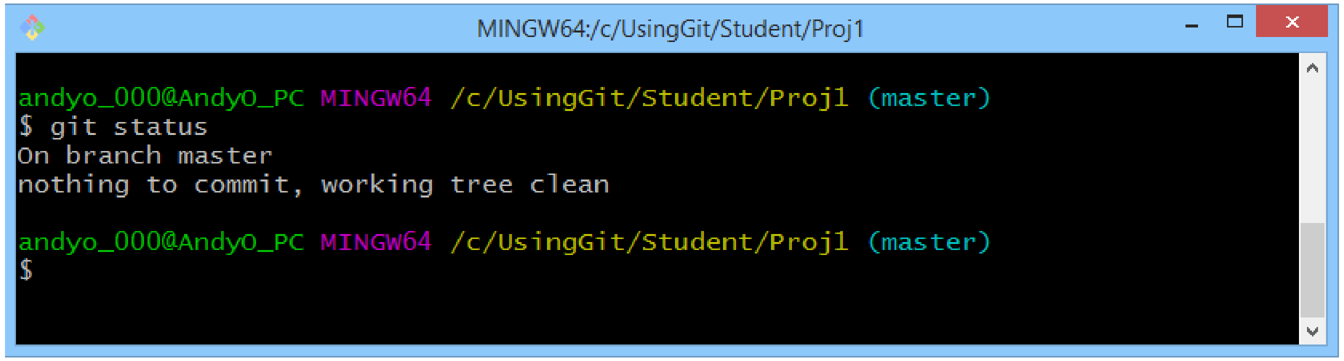
- For example, enter the following commit message
  - Then save the file and close Notepad, so the Git commit proceeds



```
This is my first Git commit, added 3 Java files.
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   MyFirstApp.java
#       new file:   MyHelper.java
#       new file:   MyProg.java
#
```

## Committing Files (3 of 3)

- Here's the current status of files in the working directory
  - There's nothing to commit (i.e. the staging area is empty)
  - There are no modified files either (i.e. there's nothing to stage)



```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git status
On branch master
nothing to commit, working tree clean

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$
```

neueda

## Summary

- git init  : start tracking this directory
- git status : display the current state
- git add : move changes from working tree to staging area
- git commit : move changes from staging area to local repo

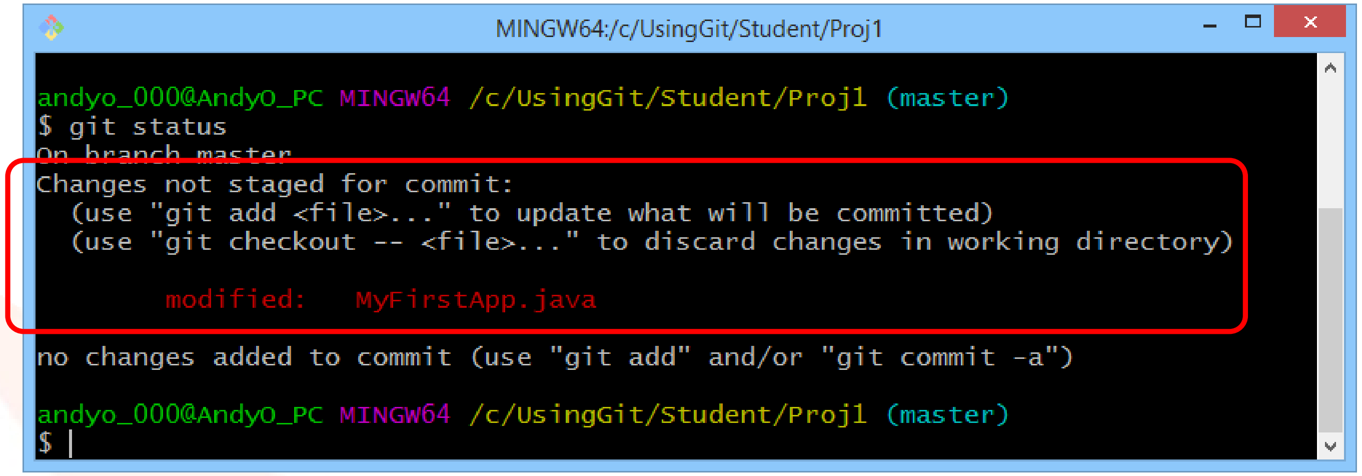
## 5. Modifying, Staging, and Committing Files

- Modifying a file
- Seeing what you've changed
- Staging a file
- Seeing what you've staged
- Committing a file
- Viewing the commit history

# Modifying a File

- When you modify a tracked file, Git knows about it
  - E.g. edit one of the Java files and save it

- When you do a Git "status":
  - Git says the changes are not yet staged for commit
  - i.e. you haven't said you're ready to commit it

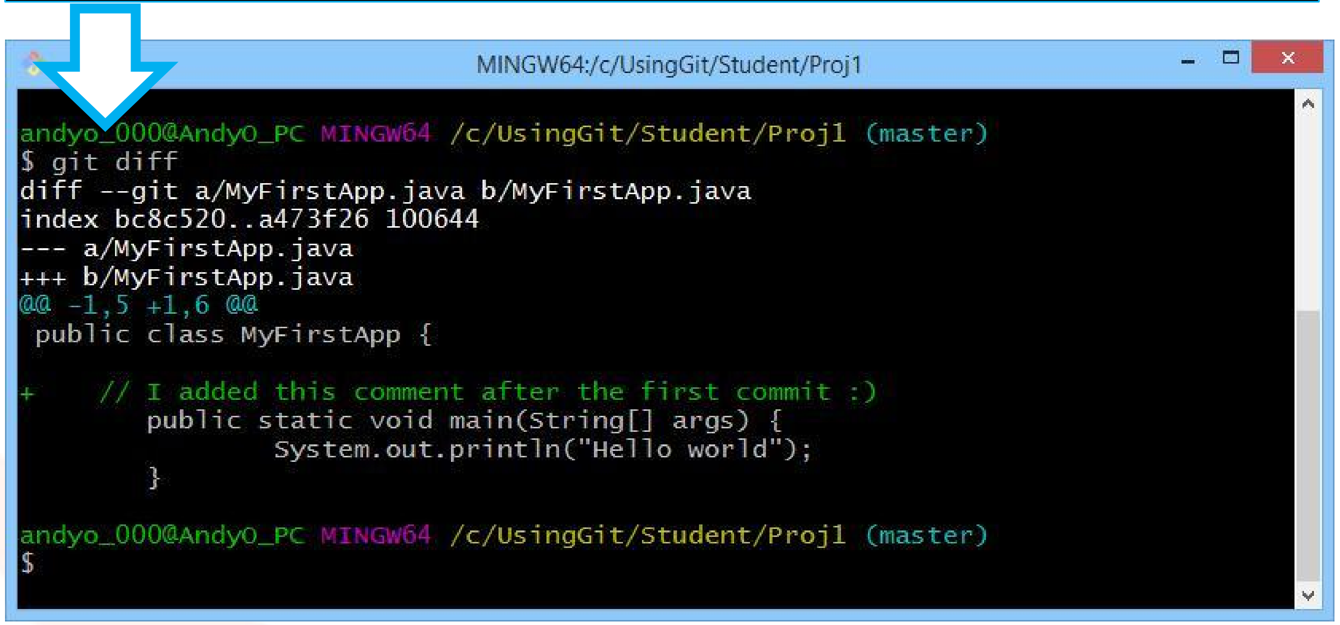


```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   MyFirstApp.java

no changes added to commit (use "git add" and/or "git commit -a")

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ |
```

## Seeing What You've Changed

- To see what you've modified (but not yet staged), use the `git diff` command
  - Compares your working directory with your staging area

```
git diff
```



```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git diff
diff --git a/MyFirstApp.java b/MyFirstApp.java
index bc8c520..a473f26 100644
--- a/MyFirstApp.java
+++ b/MyFirstApp.java
@@ -1,5 +1,6 @@
 public class MyFirstApp {

+    // I added this comment after the first commit :)
        public static void main(String[] args) {
                System.out.println("Hello world");
        }

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$
```

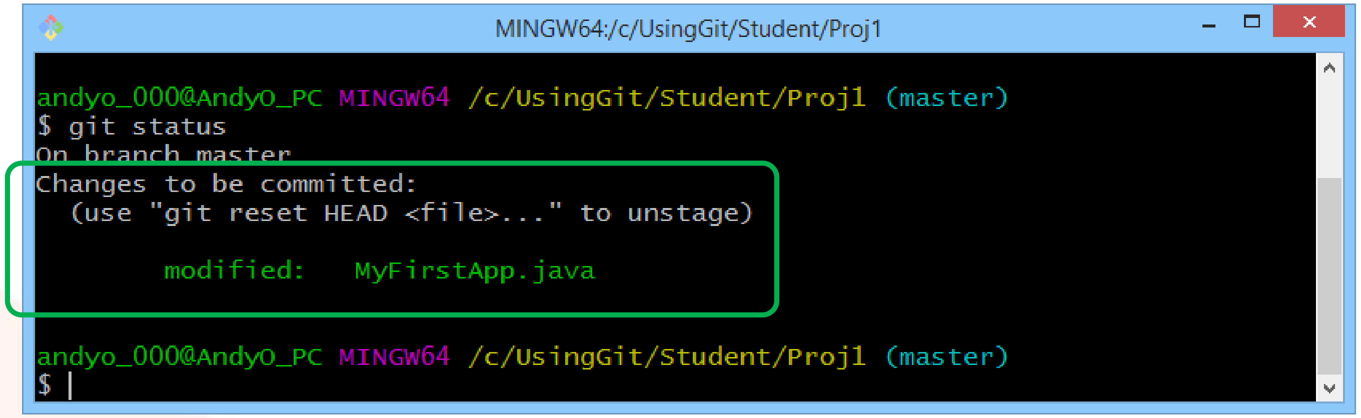
neueda 36

## Staging a File

- When you've done all the changes you plan to do on a file, the next step is to stage it
  - You stage a file via the `git add` command (again)
  - Indicates the file is now ready to be committed

```
git add MyFirstApp.java
```

- When you do a Git "status":
  - Git now says the changes are staged for commit



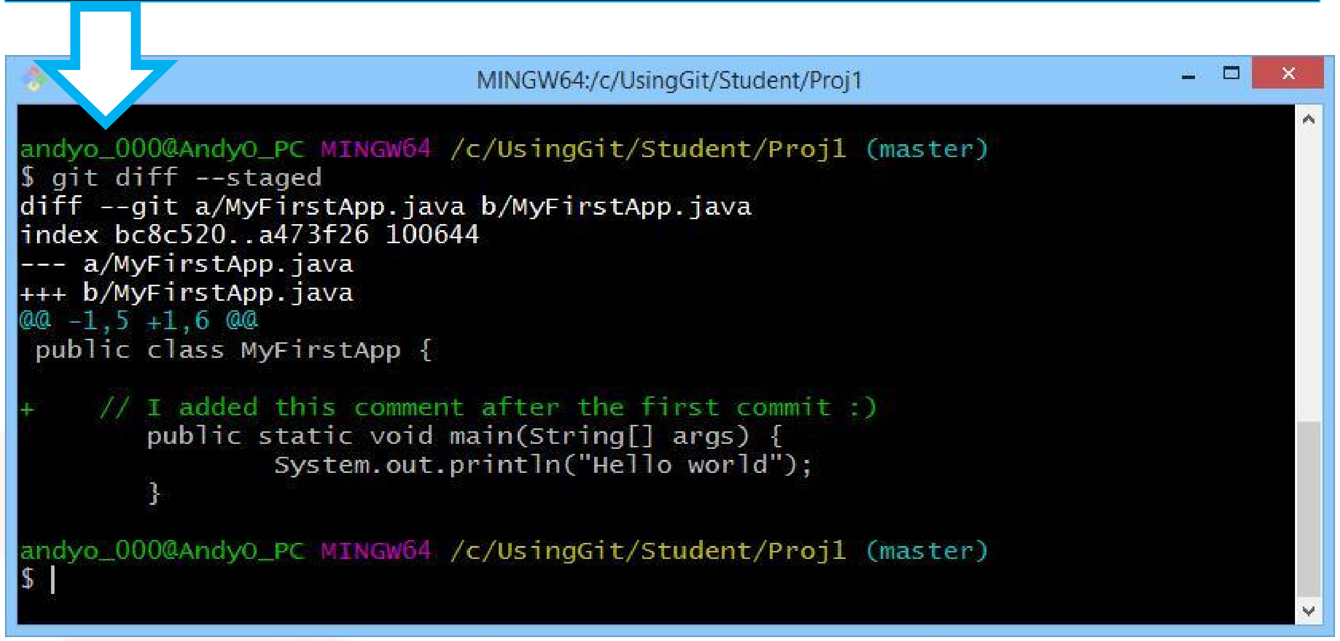
```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   MyFirstApp.java

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ |
```

neueda

## Seeing What You've Staged

- To see what you've staged (but not yet committed), use the `git diff --staged` command (or `--cached`)
  - Compares staged changed with your last commit

```
git diff --staged
```

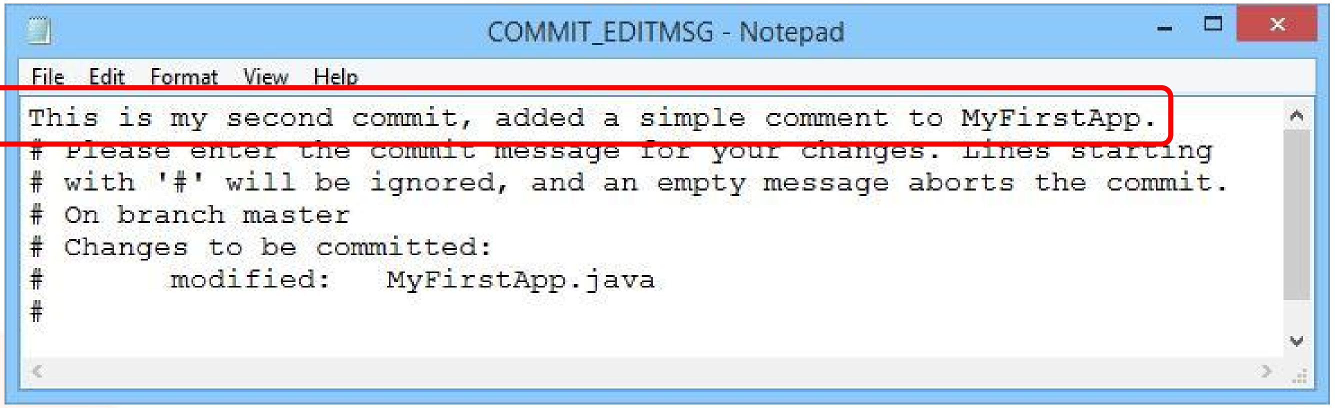


```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git diff --staged
diff --git a/MyFirstApp.java b/MyFirstApp.java
index bc8c520..a473f26 100644
--- a/MyFirstApp.java
+++ b/MyFirstApp.java
@@ -1,5 +1,6 @@
 public class MyFirstApp {

+    // I added this comment after the first commit :)
        public static void main(String[] args) {
                System.out.println("Hello world");
        }

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ |
```

## Committing a File

- When you're ready, commit your staged changes, use the `git commit` command
  - And enter a suitable "commit" message in the text editor

```
git commit
```

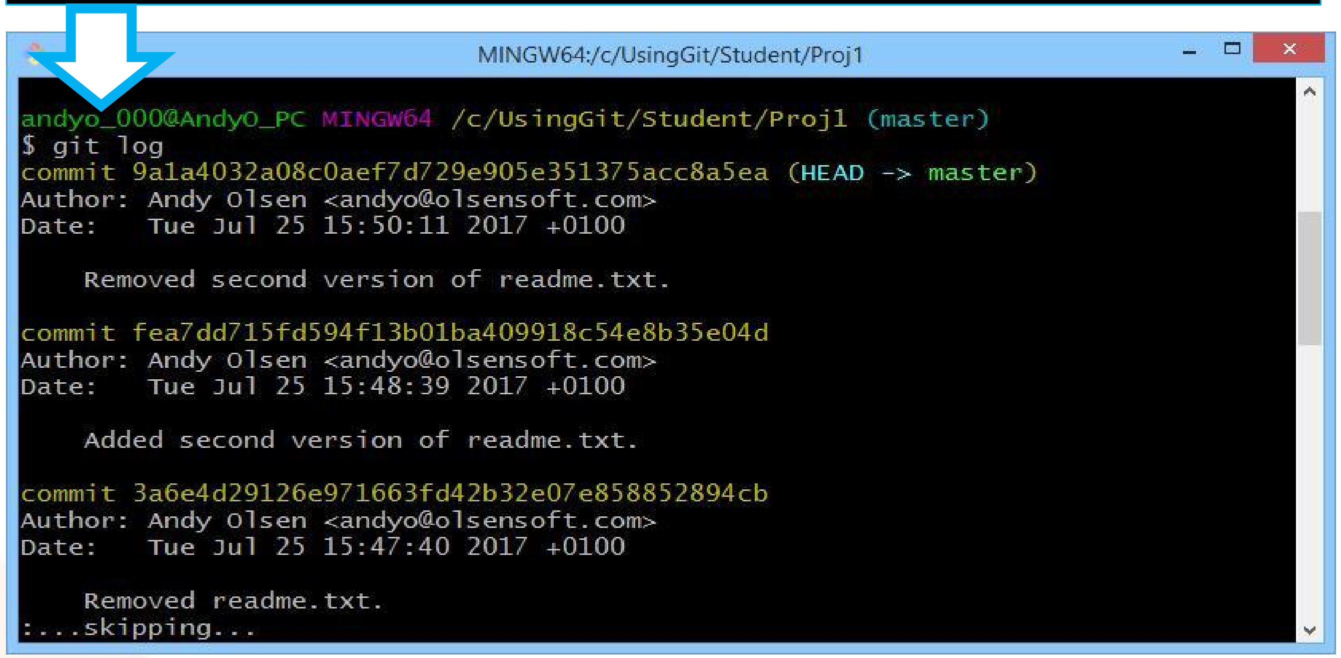


```
This is my second commit, added a simple comment to MyFirstApp.
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   MyFirstApp.java
#
```

# Viewing the Commit History (1 of 2)

- To view the commit history, use the `git log` command

```
git log
```



```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj1 (master)
$ git log
commit 9a1a4032a08c0aef7d729e905e351375acc8a5ea (HEAD -> master)
Author: Andy Olsen <andyo@olsensoft.com>
Date:   Tue Jul 25 15:50:11 2017 +0100

    Removed second version of readme.txt.

commit fea7dd715fd594f13b01ba409918c54e8b35e04d
Author: Andy Olsen <andyo@olsensoft.com>
Date:   Tue Jul 25 15:48:39 2017 +0100

    Added second version of readme.txt.

commit 3a6e4d29126e971663fd42b32e07e858852894cb
Author: Andy Olsen <andyo@olsensoft.com>
Date:   Tue Jul 25 15:47:40 2017 +0100

    Removed readme.txt.
:...skipping...
```

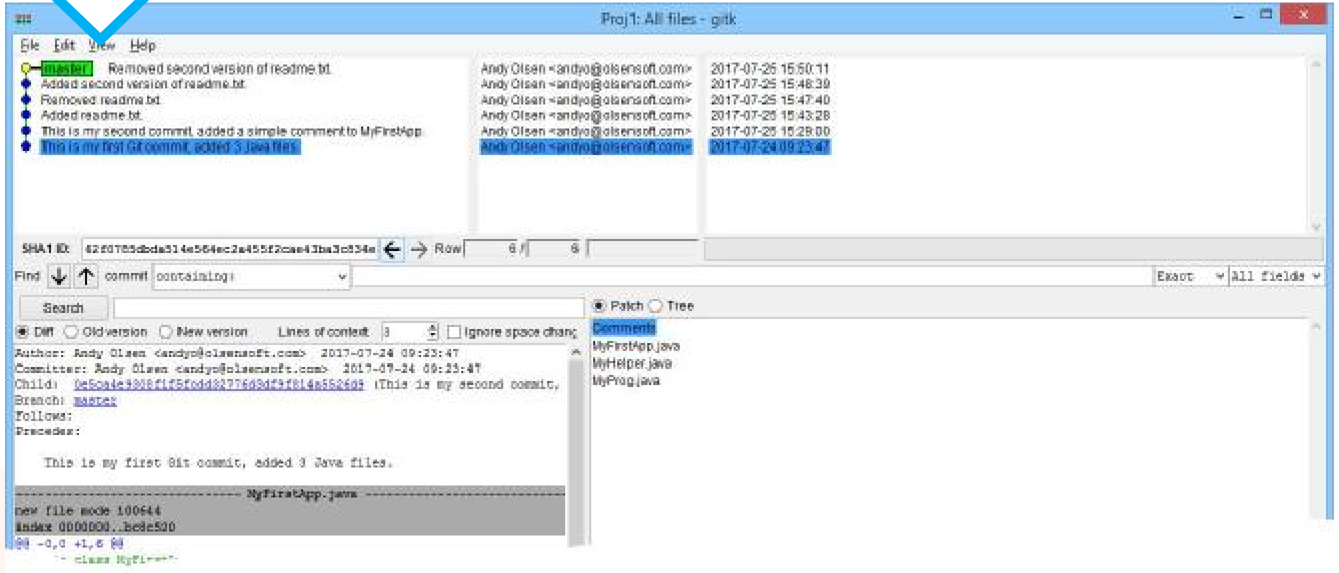
- For details, see:
  - https://git-scm.com/docs/git-log

neueda

## Viewing the Commit History (2 of 2)

- You can view the commit history graphically, via `gitk`

```
gitk
```



```
Removed second version of readme.txt.                                Andy Olsen <andyo@olsensoft.com>  2017-07-25 15:50:11
Added second version of readme.txt.                                  Andy Olsen <andyo@olsensoft.com>  2017-07-25 15:48:39
Removed readme.txt.                                                  Andy Olsen <andyo@olsensoft.com>  2017-07-25 15:47:40
Added readme.txt.                                                    Andy Olsen <andyo@olsensoft.com>  2017-07-25 15:43:28
This is my second commit, added a simple comment to MyFirstApp.      Andy Olsen <andyo@olsensoft.com>  2017-07-25 15:29:00
This is my first Git commit, added 3 Java files.                     Andy Olsen <andyo@olsensoft.com>  2017-07-24 09:23:47
```

SHA1 ID: 62f0785dbda514e564ec2a455f2cae43ba3c534e   Row 6 / 6

```
Author: Andy Olsen <andyo@olsensoft.com>  2017-07-24 09:23:47
Committer: Andy Olsen <andyo@olsensoft.com>  2017-07-24 09:23:47
Child:  0e5ca4e9308f1f5fcdd8277683df9f814a55260f (This is my second commit,
Branch: master
Follows:
Precedes:

    This is my first Git commit, added 3 Java files.

-------------------------------- MyFirstApp.java --------------------------------
new file mode 100644
index 0000000..be8e520
@@ -0,0 +1,6 @@
+ class MyFi
```

## 6. Additional Techniques

- Undoing changes
- Bypassing the staging area
- Removing files
- Telling Git to ignore files

## Undoing Changes

- If you want to discard changes to modified files…
  - Revert the files status from "modified" to "unmodified" as follows:

```
git checkout -- SomeFiles
```

- If you want to unstage staged files (e.g. because you accidentally staged files you're not ready to commit yet)…
  - Remove the files from the Git staging area as follows:

```
git reset HEAD SomeFiles
```

- If you want to change your last commit (e.g. because you forgot to include some files in your previous commit)…
  - Run the commit again with the --amend option
  - Git merges the current staging area into the previous commit

```
git commit --amend
```

## Bypassing the Staging Area

- In the examples we've seen so far, we've run two separate commands to stage and then commit files…
  - First we stage modified files:

```
git add SomeFiles
```

  - Then we commit staged files:

```
git commit
```

- You can bypass the staging area if you like
  - Run `git commit` with the `–a` option
  - Git automatically stages all tracked files, then it does the commit

```
git commit -a
```

## Removing Files from Git

- To remove a file from Git, and physcially delete the file:
  - Use the `git rm` command

```
git rm readme.txt
```

- If you want to remove a file from Git, but not delete it:
  - Use the `git rm --cached` command

```
git rm --cached readme.txt
```

- Note:
  - When you run `git rm`, it just tells Git to "stage" the file removal
  - You have to run `git commit` to actually "do" the file removal

```
git commit
```

## Telling Git to Ignore Files

- Often you have files that you don't want to store in Git
  - E.g. `.class` files in a Java project
  - E.g. the contents of the `obj` or `bin` folder in a .NET project

- You can tell Git to ignore these files, i.e. not to track them
  - Create a file named `.gitignore` in your working directory
  - Define file patterns to be ignored

```
# Ignore .class files
*.class

# Ignore the obj and bin folders, and all their contents
obj/
bin/
```

- For details, see:
  - https://git-scm.com/docs/gitignore

## Exercise

- In the Proj1 directory
- Create a text file called not_to_be_added.txt
  - It doesn’t matter what it’s contents are
- Do a git status, see how the new text file is listed in red
- Create a file called .gitignore, containing a single line:
  - not_to_be_added.txt
- Now do a git status and see the text file is no longer shown
- Add and commit the .gitignore file to your repository

## Summary

- Introduction to version control systems
- Overview of Git
- Using the Git Bash shell
- Adding new files to a Git repository
- Modifying, staging, and committing files
- Additional techniques

## Branching

## Contents

1. Branching essential concepts
2. Merging branches
3. Resolving merge conflicts

## 1. Branching Essential Concepts

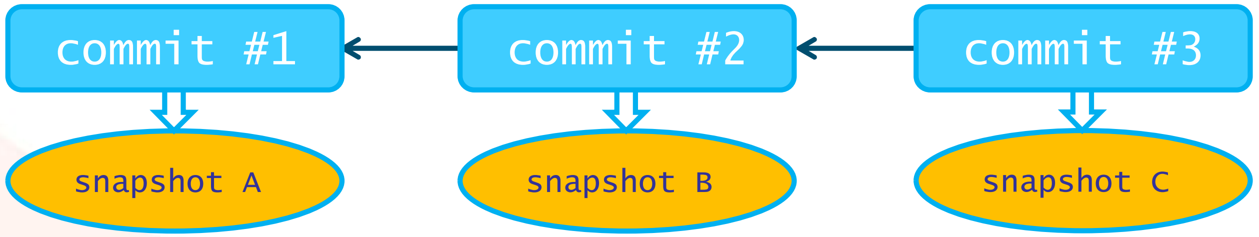
- Recap: Commit history
- What is a branch?
- Creating a new branch
- Changing the current branch
- Aside: Shortcut syntax
- Making subsequent commits
- Switching back to the master branch
- Making subsequent commits (again)
- Viewing branches
- Exercise

## Recap: Commit History

- When you run `git commit` for the 1st time on a project…
  - Git creates a <u>commit object</u> (snapshot of files initially)

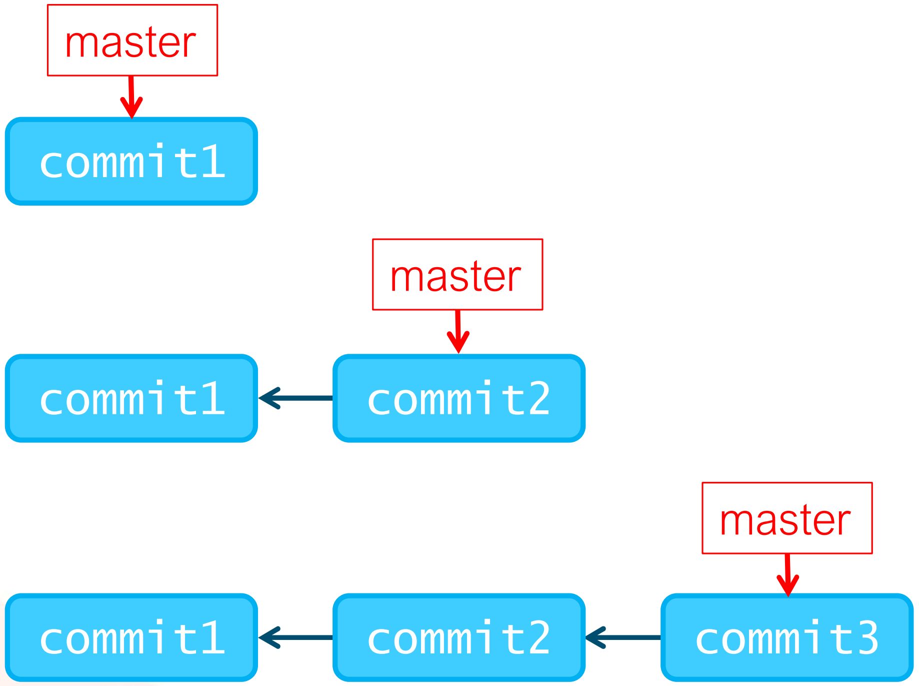
- When you run `git commit` the next time…
  - Git creates another commit object (snapshot of files now)
  - The new commit object points to the previous commit object in the commit history

- As subsequent commits occur, you end up with a chain of commit objects



## What is a Branch?

- A branch is a pointer to a commit in the commit history
  - The default branch name in Git is called master
  - The master branch points to your most recent commit

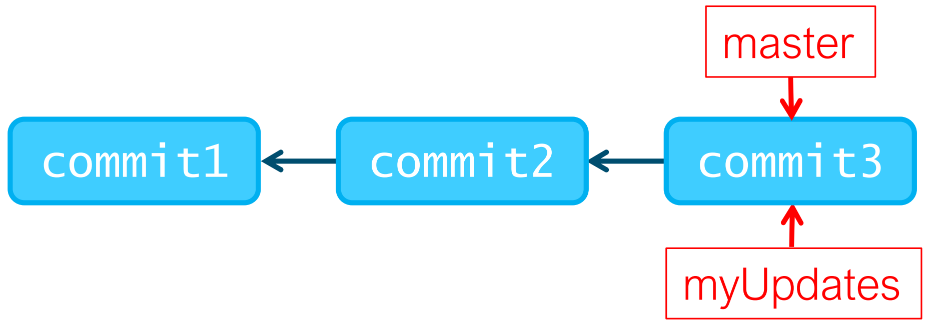


## Creating a New Branch

- You can create a new branch easily and cheaply, via the `git branch` command

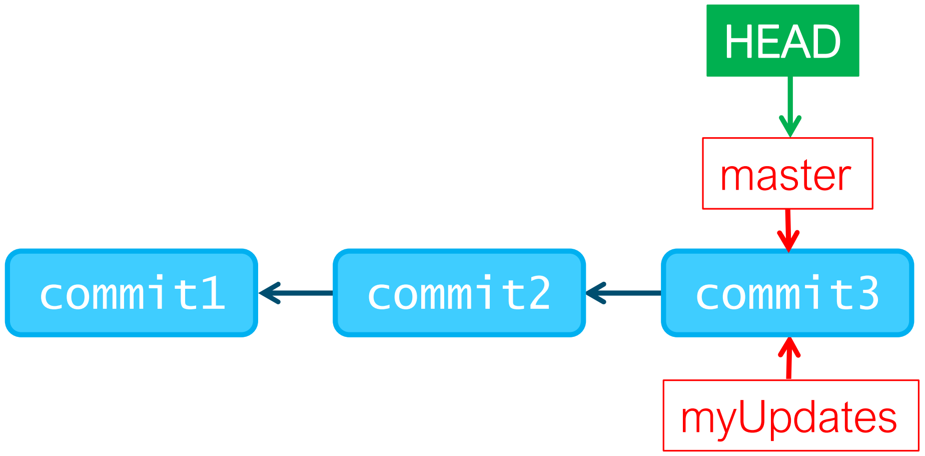
```
git branch myUpdates
```

- This creates a new pointer at the same commit you're currently on



## Changing the Current Branch (1 of 2)

- Git has a HEAD pointer, indicating your current branch
  - Initially this points to the master branch, and…
  - It still points to the master branch even if you create a new branch

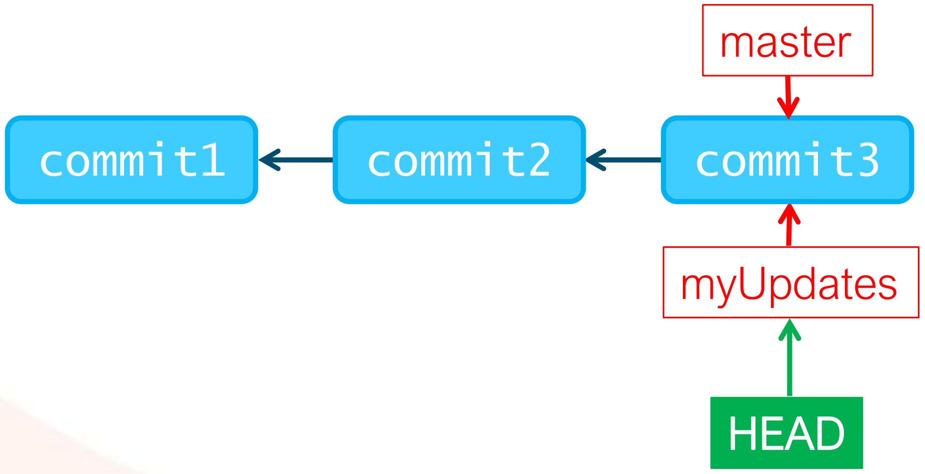


## Changing the Current Branch (2 of 2)

- To tell Git you want to change the current branch, use the `git checkout` command

```
git checkout myUpdates
```

- The `HEAD` pointer now points to the `myUpdates` branch



## Aside: Shortcut Syntax

- If you want, you can create a new branch AND switch to make it the current branch immediately
  - Use the `-b` switch on the `git checkout` command

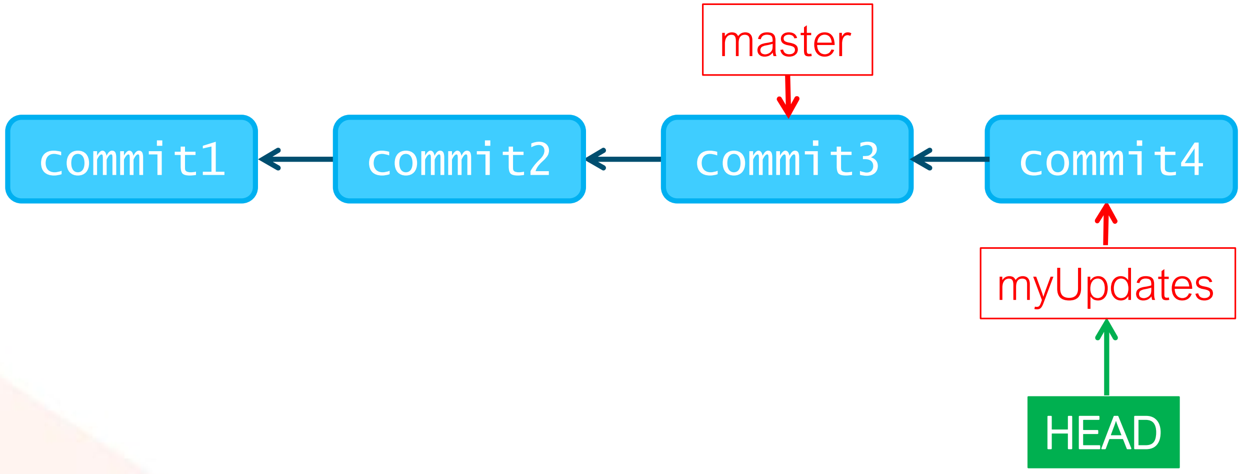
```
git checkout -b myUpdates
```

## Making Subsequent Commits

- If you make subsequent commits, Git will move your current branch pointer (i.e. `myUpdates` now)

```
git commit -a -m 'Made some changes, this is commit #4'
```

- Git moves the `myUpdates` branch and the `HEAD` pointer

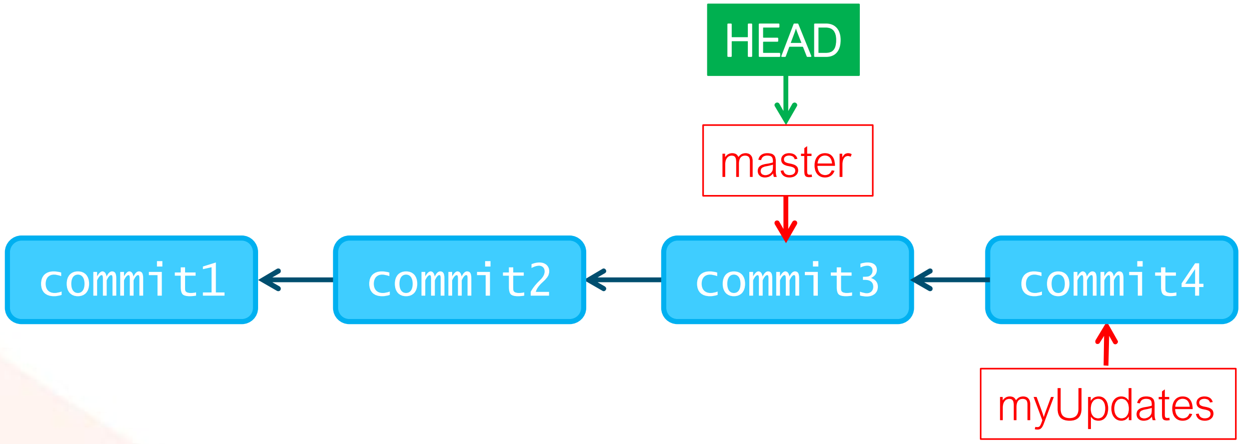


## Switching Back to the Master Branch

- If you want to switch back to the master branch, use the `git` checkout command again

```
git checkout master
```

- Git resets the HEAD pointer, and also reverts files in your working directory to the snapshot that master points to

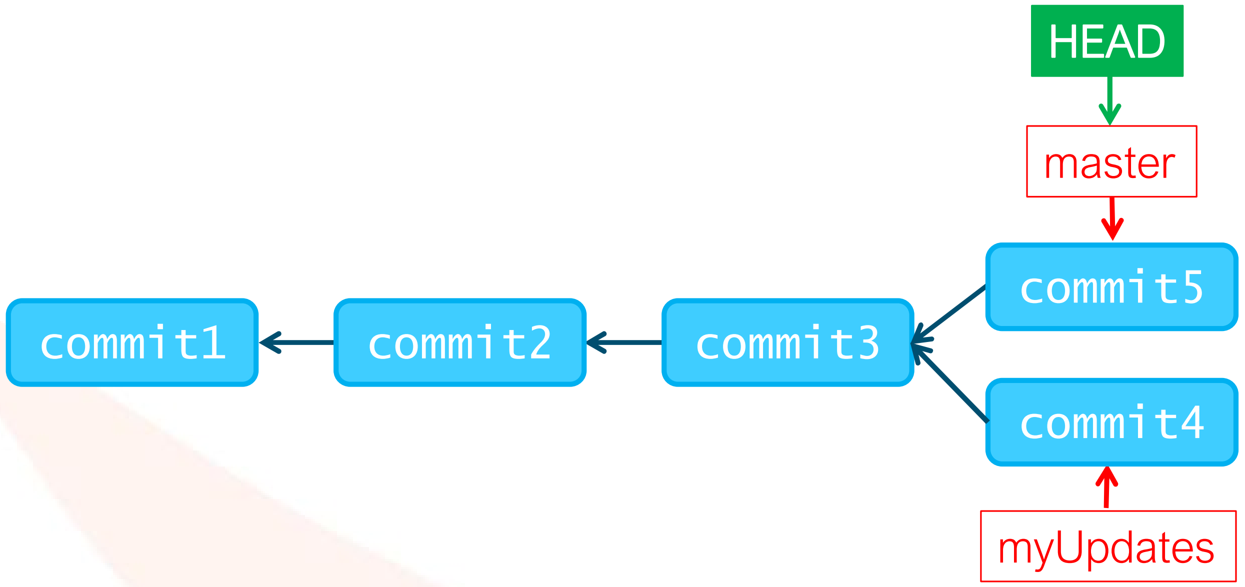

$$\begin{tikzcd}
 & & \text{HEAD} \arrow[d] & \\
 & & \text{master} \arrow[d] & \\
\text{commit1} & \text{commit2} \arrow[l] & \text{commit3} \arrow[l] & \text{commit4} \arrow[l] \\
 & & & \text{myUpdates} \arrow[u]
\end{tikzcd}$$

## Making Subsequent Commits (Again)

- If you now make subsequent commits, Git will move your current branch pointer (i.e. master now)

```
git commit -a -m 'Made some changes, this is commit #5'
```

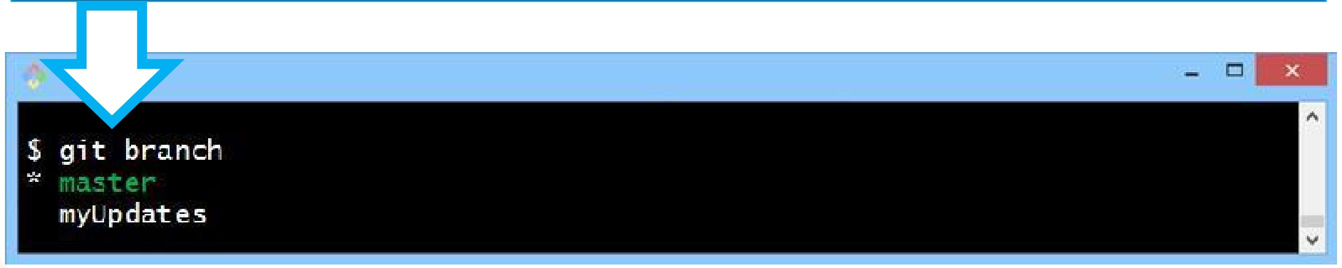
- The project has now diverged into two separate strands of development, hence the term 'branch'



## Viewing Branches

- To view all your current branches, use the `git branch` command
  - Git will list all the branches and will highlight the current branch you have checked out, i.e. where HEAD currently points

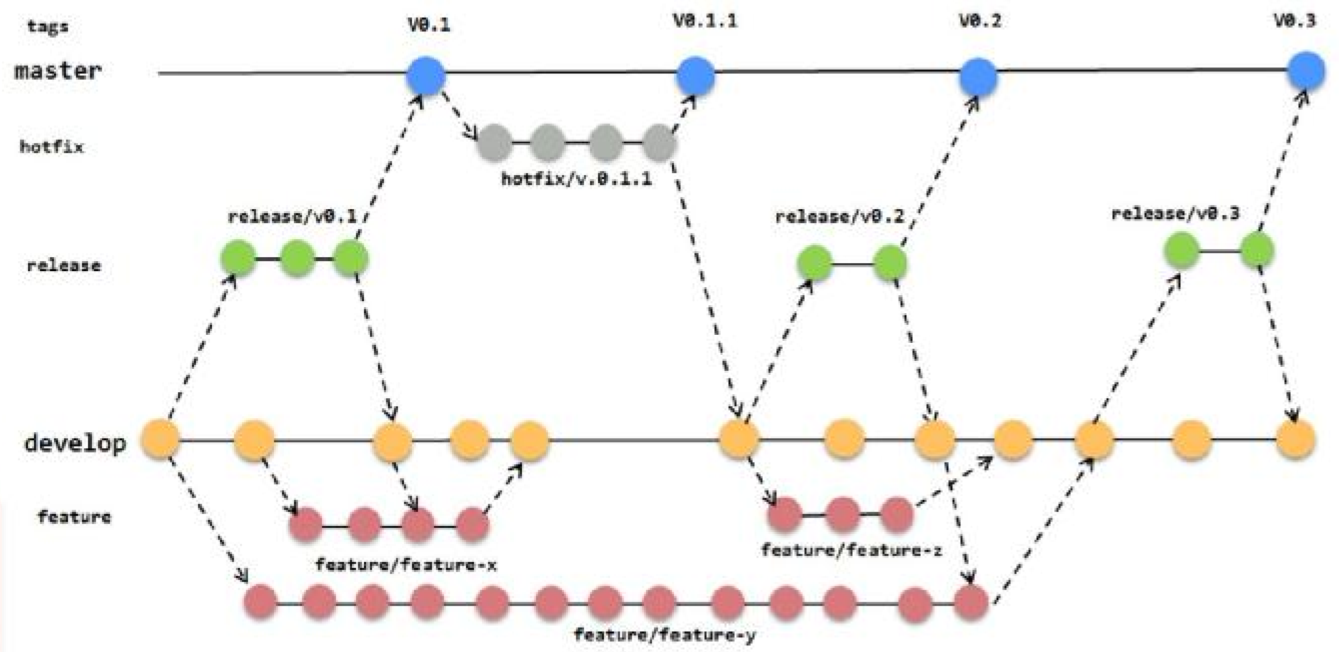
```
git branch
```



```
$ git branch
* master
  myUpdates
```

# Example Branch Structure

- There are a number of commonly used branching strategies
  - https://www.atlassian.com/git/tutorials/comparing-workflows



## 2. Merging Branches

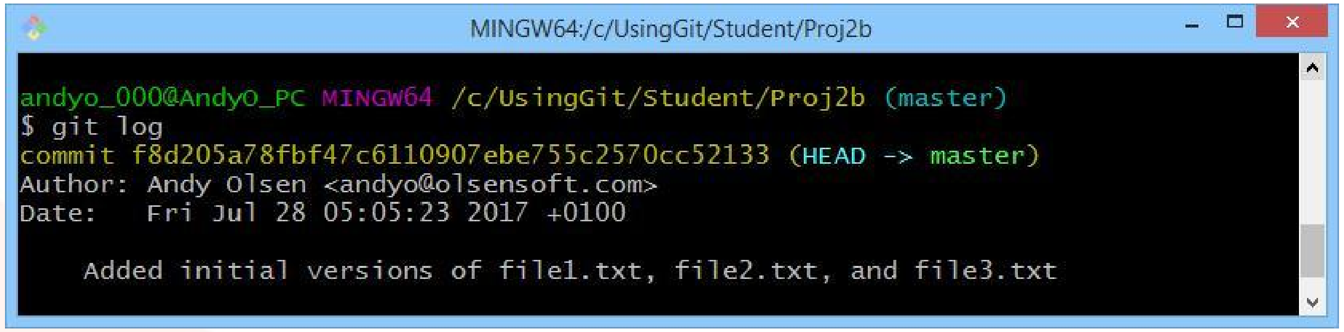
- Overview
- Creating a branch for "req1"
- Making and committing changes for "req1"
- Creating a branch for "bugfix007"
- Making and committing changes for "bugfix007"
- Merging "bugfix007" into the master branch
- Making and committing further changes for “req1”
- Merging “req1” into the master branch and resolve conflicts

## Overview (1 of 2)

- So far we've seen how to create separate branches
  - … and how to commit changes on these branches
- At some point, you'll probably want to merge your branch back into the master branch
  - … so that your changes are folded into the "main" project
- In Git, you do this via the `git merge` command
  - We'll walk through an example of merging in this section
  - We'll show how to deal with merge conflicts in the next section

## Overview (2 of 2)

- We'll work in the following directory:
  - C:\UsingGit\Student\Proj2b
- This working directory contains 3 simple files
  - file1.txt, file2.txt, file3.txt
- We've added these files to the Git repository, and done an initial commit
  - On the master branch



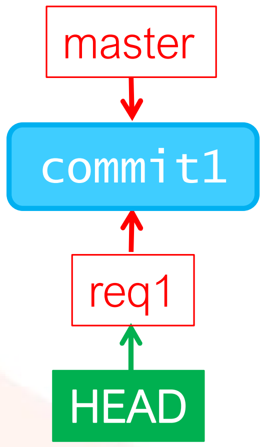
```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (master)
$ git log
commit f8d205a78fbf47c6110907ebe755c2570cc52133 (HEAD -> master)
Author: Andy Olsen <andyo@olsensoft.com>
Date:   Fri Jul 28 05:05:23 2017 +0100

    Added initial versions of file1.txt, file2.txt, and file3.txt
```

## Creating a Branch for "Req1"

- Let's say we want to start working on "Requirement 1" in a hypothetical requirements document
  - This will entail making changes to various files
  - So let's create a new branch called "req1", to house these changes
  - Let's also make this the current branch

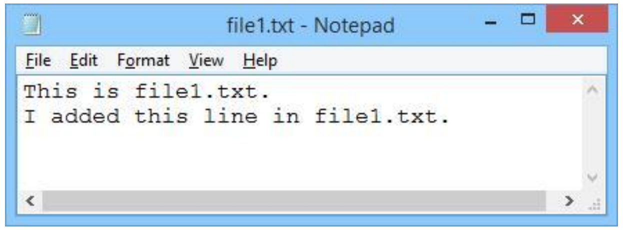
```
git checkout -b req1
```

- Current situation:

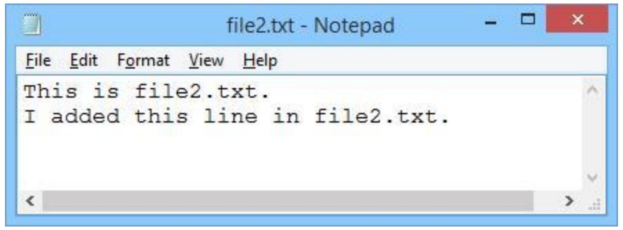


## Making and Committing Changes for "Req1"

- Make some changes to file1.txt and file2.txt, and commit

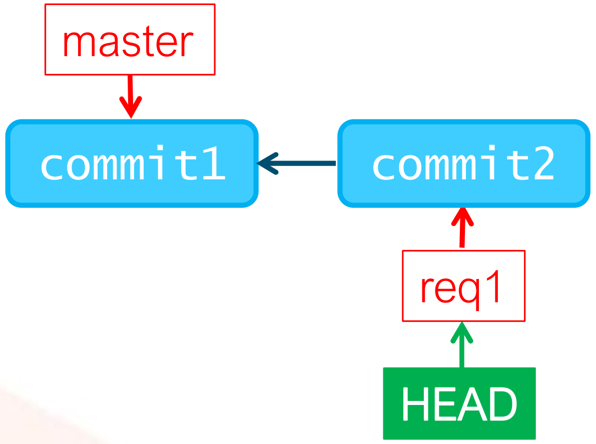


```
This is file1.txt.
I added this line in file1.txt.
```



```
This is file2.txt.
I added this line in file2.txt.
```

```
git commit -a -m 'Made changes for Req 1, lets call this commit #2'
```



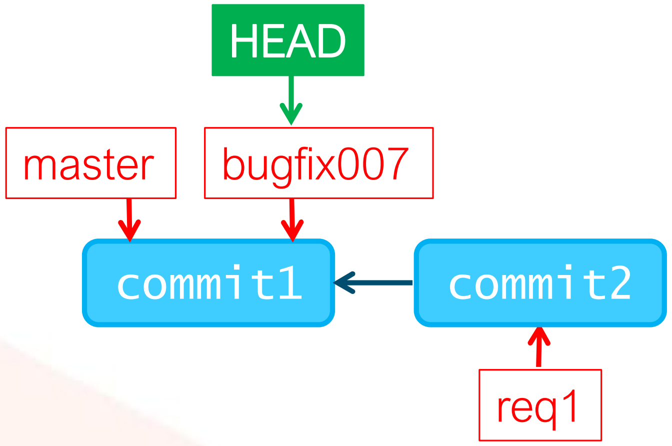
neueda

## Creating a Branch for "Bugfix007"

- Let's say someone has reported a bug "007" in the original codebase, which we need to fix urgently
  - So revert to the master branch
  - Then create a new branch called "bugfix007", and switch to it

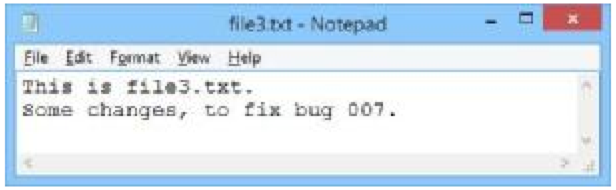
```
git checkout master
```

```
git checkout -b bugfix007
```



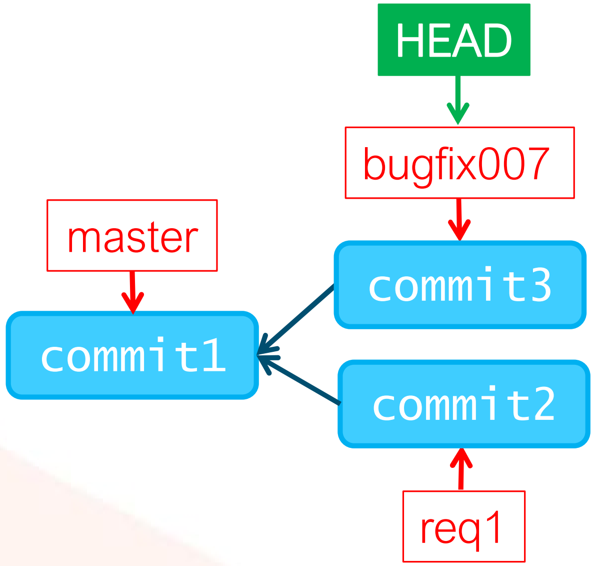
## Making and Committing Changes for "Bugfix007"

- Let's say "bugfix007" only entails changes to file3.txt
  - i.e. no conflict with the changes we made on the "req1" branch



```
This is file3.txt.
Some changes, to fix bug 007.
```

```
git commit -a -m 'Made changes for bug 007, lets call this commit #3'
```



## How and When to Merge Branches

- When you've finished making all the changes you intend to make on a branch (let's call it Branch B) …
  - Merge Branch B into the main branch (let's call it Branch A)
  - Delete Branch B, you're done with it now

- To merge Branch B into Branch A:
  - Switch to Branch A, via the `git checkout` command

```
git checkout branchA
```

  - Merge-in Branch B, via the `git merge` command

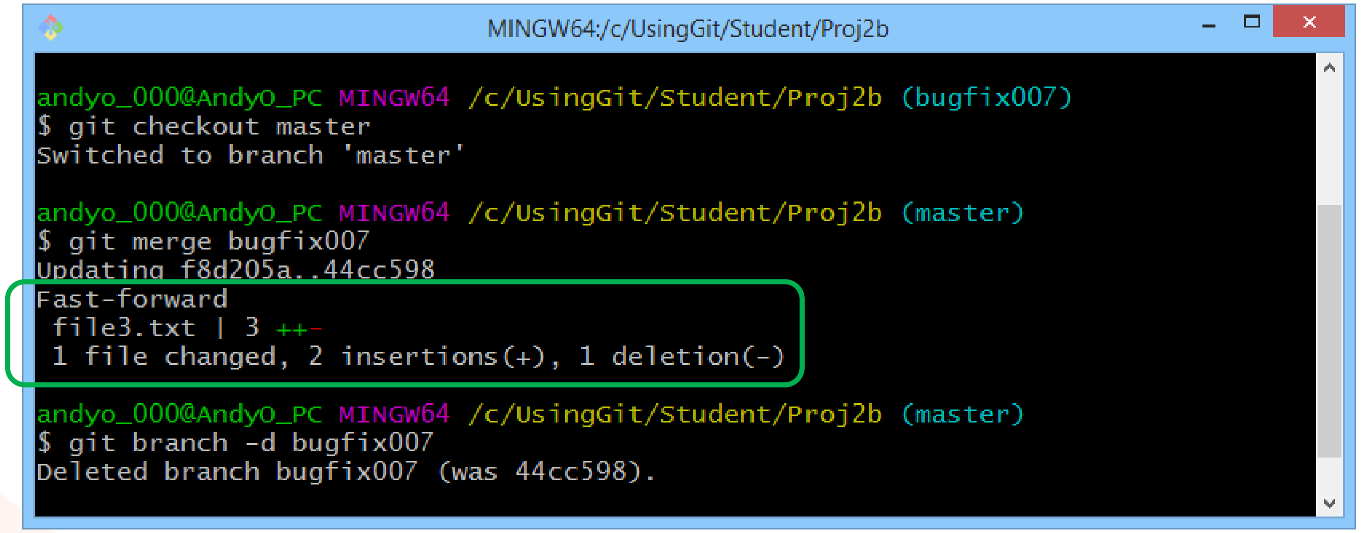
```
git merge branchB
```

  - Delete Branch B, via the `git branch -d` command

```
git branch -d branchB
```

## Merging "Bugfix007" into the Master Branch

- Let's merge the "bugfix007" branch into the master branch

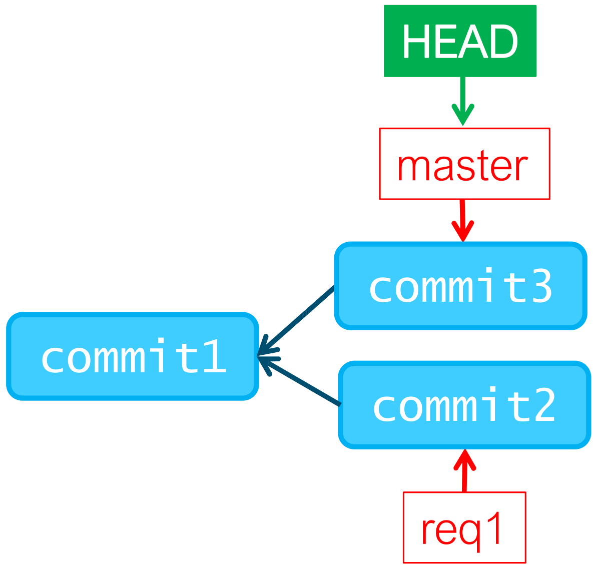


MINGW64:/c/UsingGit/Student/Proj2b

```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (bugfix007)
$ git checkout master
Switched to branch 'master'

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (master)
$ git merge bugfix007
Updating f8d205a..44cc598
Fast-forward
 file3.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (master)
$ git branch -d bugfix007
Deleted branch bugfix007 (was 44cc598).
```

# Reviewing the Current State of Play

- Here's our current state of play, pictorially:



- Notes:
  - The "bugfix007" changes are merged into the master branch
  - The "req1" changes aren't merged in yet, because we're not ready

## 3. Resolving Merge Conflicts

- Overview
- Switching to the "req1" branch
- Reviewing the current state of play
- Making and committing changes for "Req1"
- Trying to merge "Req1" into the master branch
- Viewing merge conflicts
- Resolving merge conflicts
- Indicating you've resolved merge conflicts
- Verifying you've resolved merge conflicts
- Committing the merge
- Reviewing the current state of play

## Overview

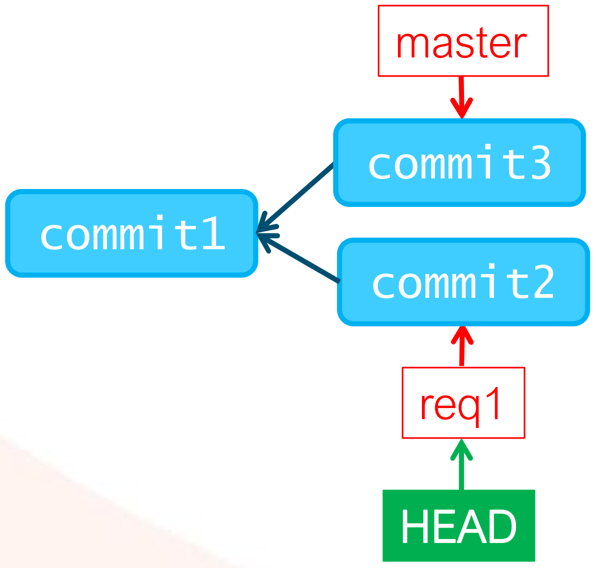
- In the previous section, we merged the "bugfix007" branch into the master branch
  - There were no conflicts between the branches, so Git was able to simply fast-forward the master branch pointer

- In this section, we're going to make some conflicting changes to the "req1" branch
  - Then we'll try to merge the "req1" branch into the master branch…
  - Git will realise our changes conflict with the master branch
  - Git will force us to intervene manually, to resolve the conflict(s)

## Switching to the "Req1" Branch

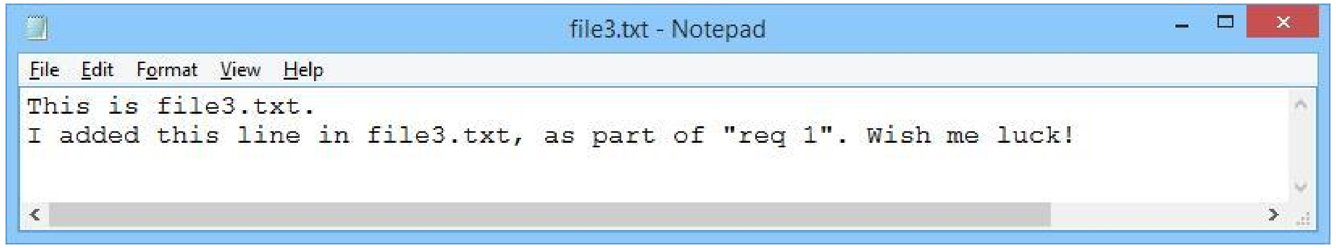
- Let's switch to the "req1" branch, so we can continue working on the files in this branch

```
git checkout req1
```



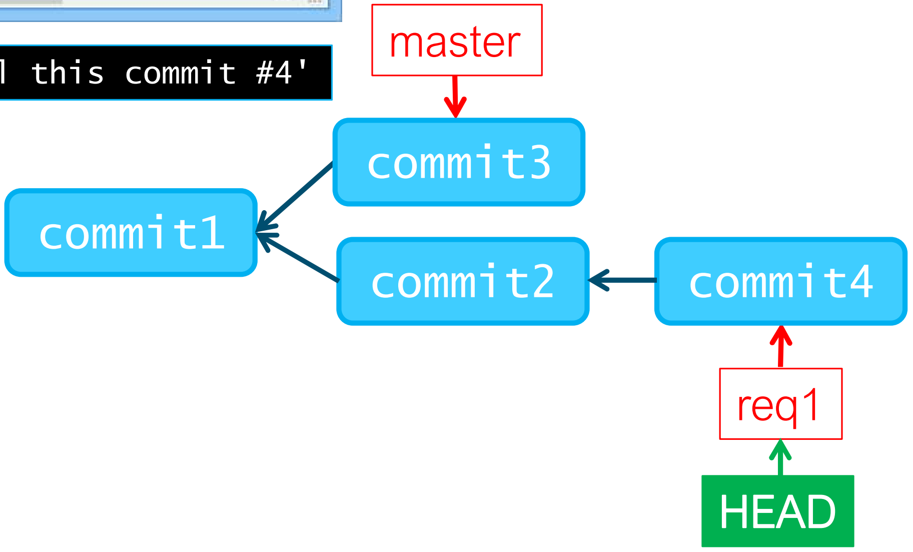
## Making and Committing Changes for "Req1"

- Let's make conflicting changes to file3.txt, and commit



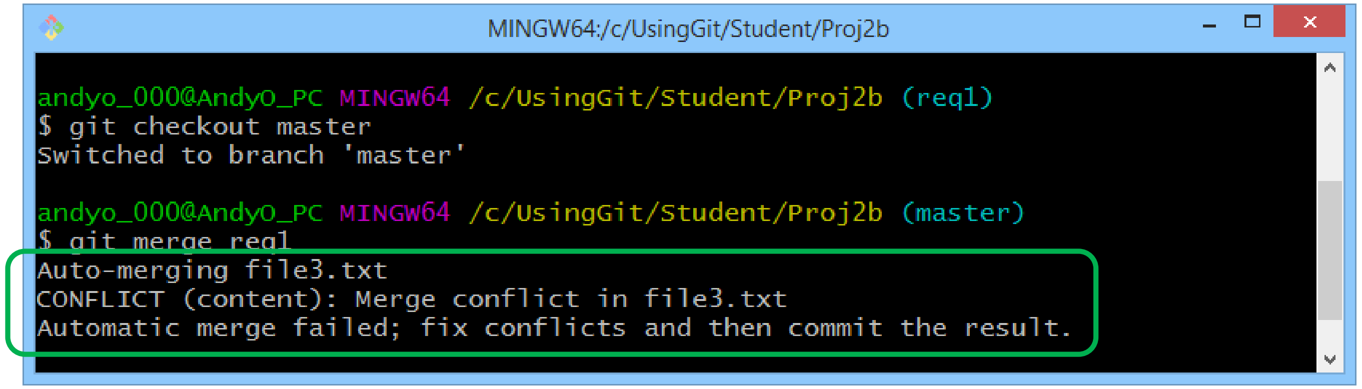
```
This is file3.txt.
I added this line in file3.txt, as part of "req 1". Wish me luck!
```

```
git commit -a -m 'Final changes for Req 1, lets call this commit #4'
```



# Trying to Merge "Req1" into the Master Branch

- Let's try to merge the "req1" branch into master branch

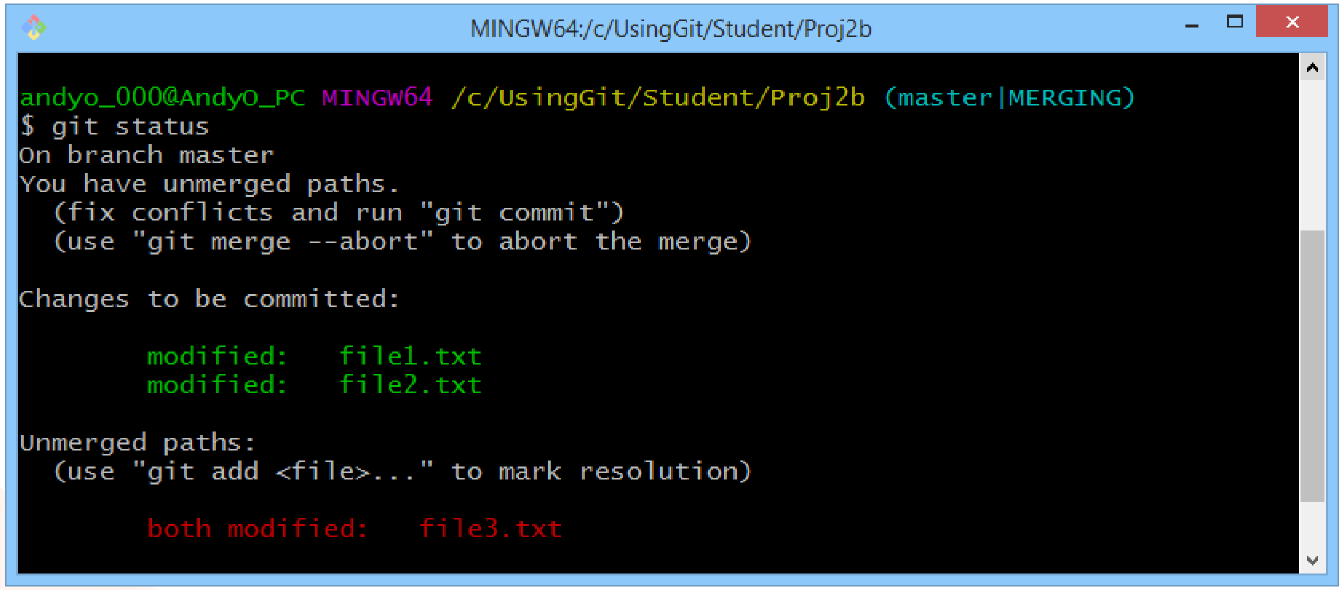


```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (req1)
$ git checkout master
Switched to branch 'master'

andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (master)
$ git merge req1
Auto-merging file3.txt
CONFLICT (content): Merge conflict in file3.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- Git detects conflicts when we try to do the merge
  - Git realises our changes to file3.txt conflict with the master branch
  - Get tells us we need to resolve the conflict first

## Viewing Merge Conflicts

- Run `git status` to see which files merged successfully and which files are in conflict

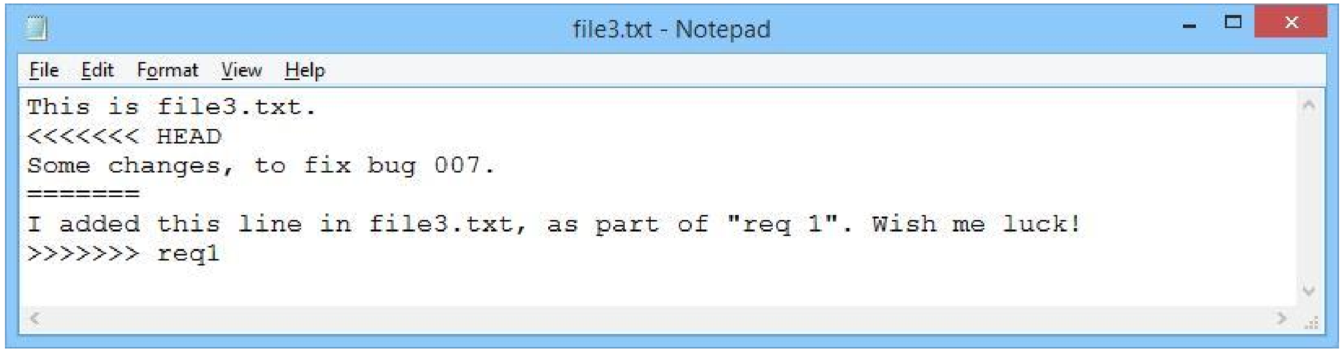


MINGW64:/c/UsingGit/Student/Proj2b

```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (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)

Changes to be committed:

        modified:   file1.txt
        modified:   file2.txt

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

        both modified:   file3.txt
```

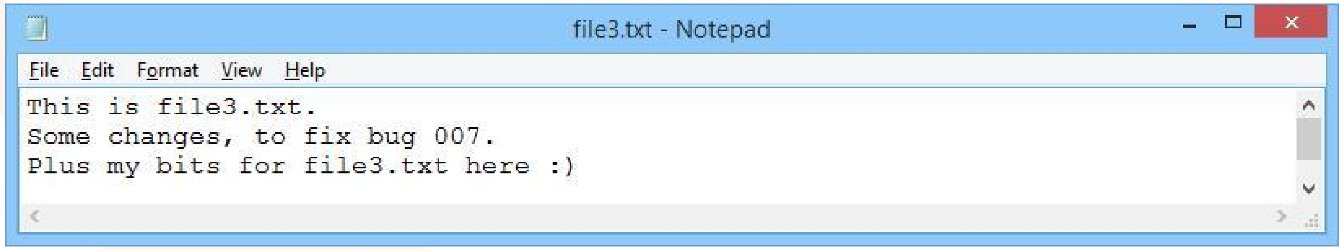
## Resolving Merge Conflicts

- Git adds conflict-resolution markers in the conflicting file(s)
  - Indicates conflict between your branch and the branch you're trying to merge into



```
This is file3.txt.
<<<<<<< HEAD
Some changes, to fix bug 007.
=======
I added this line in file3.txt, as part of "req 1". Wish me luck!
>>>>>>> req1
```

- We need to resolve this conflict, anyway that make sense



```
This is file3.txt.
Some changes, to fix bug 007.
Plus my bits for file3.txt here :)
```

neueda

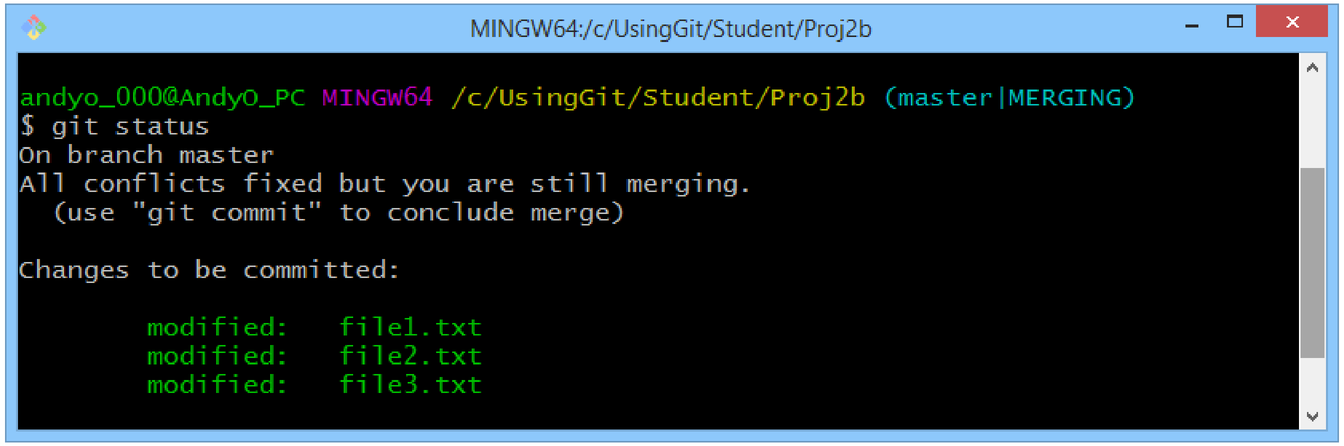
## Indicating You've Resolved Merge Conflicts

- You must tell Git you've resolved merge conflicts for a file, via the `git` add command

```
git add file3.txt
```

## Verifying You've Resolved Merge Conflicts

- Run `git status` again, to verify you've successfully resolved all merge conflicts



MINGW64:/c/UsingGit/Student/Proj2b

```
andyo_000@AndyO_PC MINGW64 /c/UsingGit/Student/Proj2b (master|MERGING)
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   file1.txt
        modified:   file2.txt
        modified:   file3.txt
```

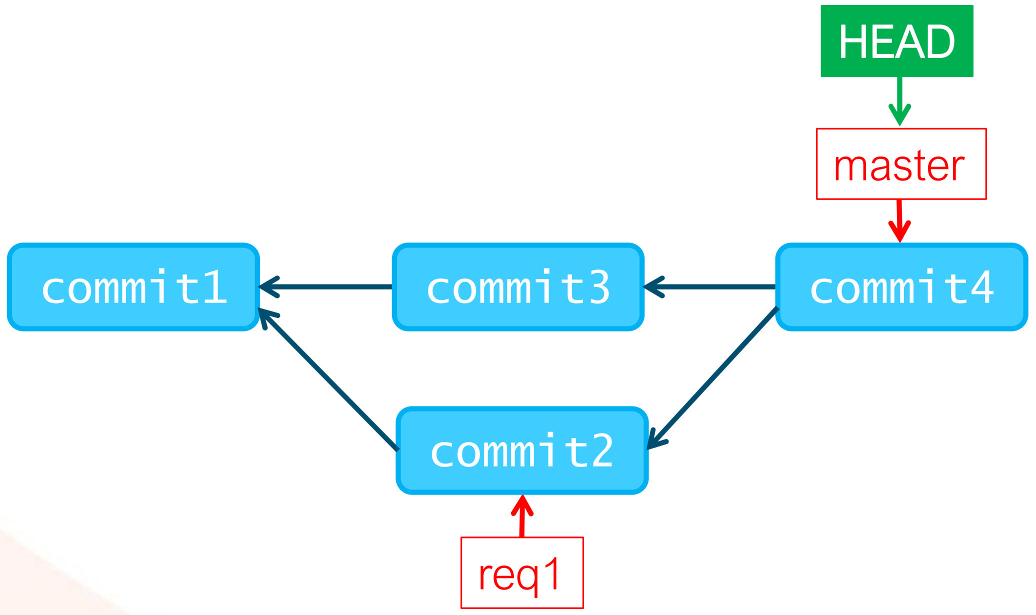
## Committing the Merge

- When you're sure all your changes have been resolved, proceed with the `git commit`
  - This is finishing off the `git commit` we tried a few slides ago

```
git commit
```

## Reviewing the Current State of Play

- Here's our current state of play, pictorially:



## Summary

- Branching essential concepts
- Merging branches
- Resolving merge conflicts

## Working with Remote Repositories

## Contents

1. Cloning a remote repository
2. Understanding remote branches

## 1. Cloning a Remote Repository

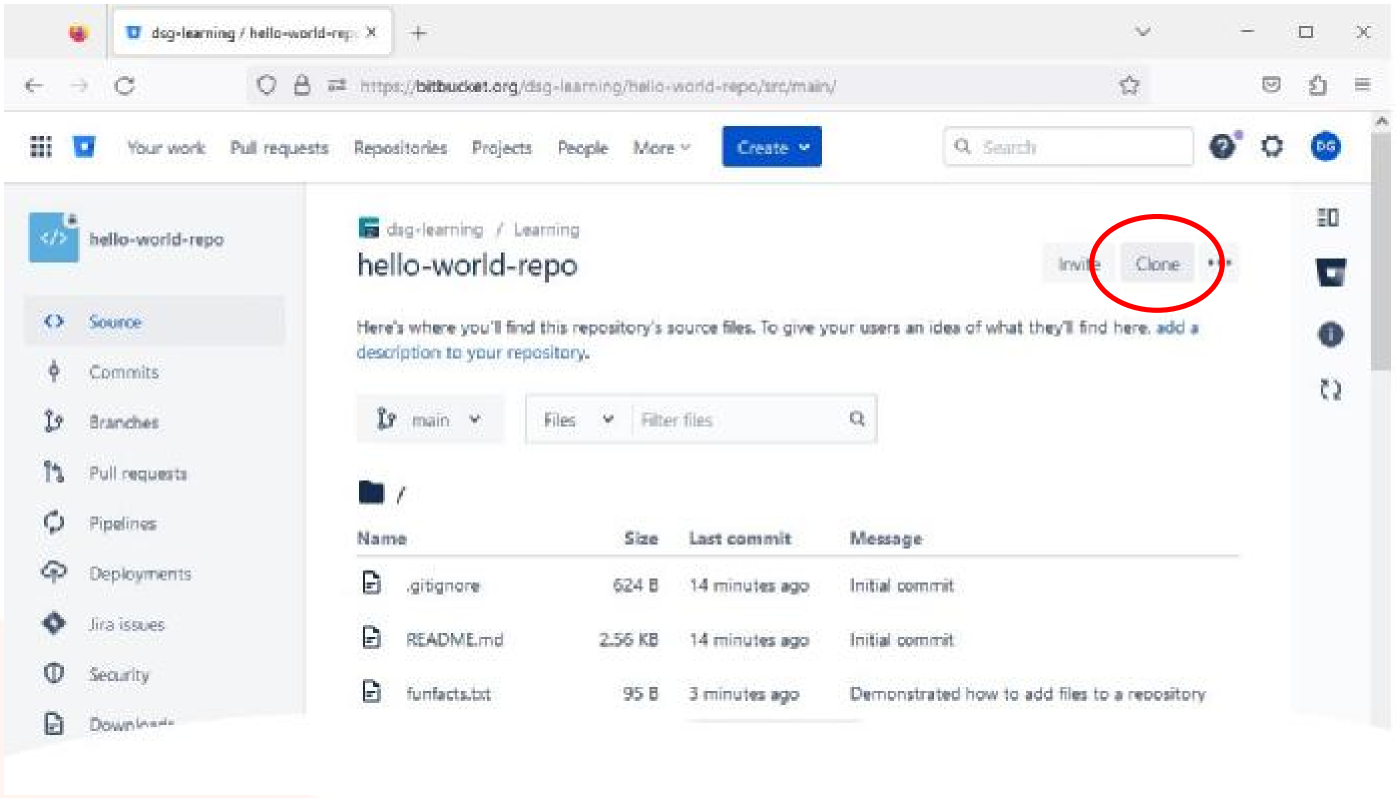
- Overview
- Cloning a repository via the Git command line
- Understanding Git cloning

## Overview

- Git platforms like BitBucket and GitHub hold your files in a remote repository
  - Held remotely on a server on the Internet
- To work on these files locally, you can clone the remote repository onto your own local machine
  - Git copies all the project files onto your local machine
  - You can then view and edit these files locally, without the need for any Internet connectivity
- When cloning a remote repository you can get the correct command from the web UI

## Copy the command from the Bitbucket Web UI

- The Bitbucket Web UI has a "Clone" button
  - Gives you the git command to clone this repository

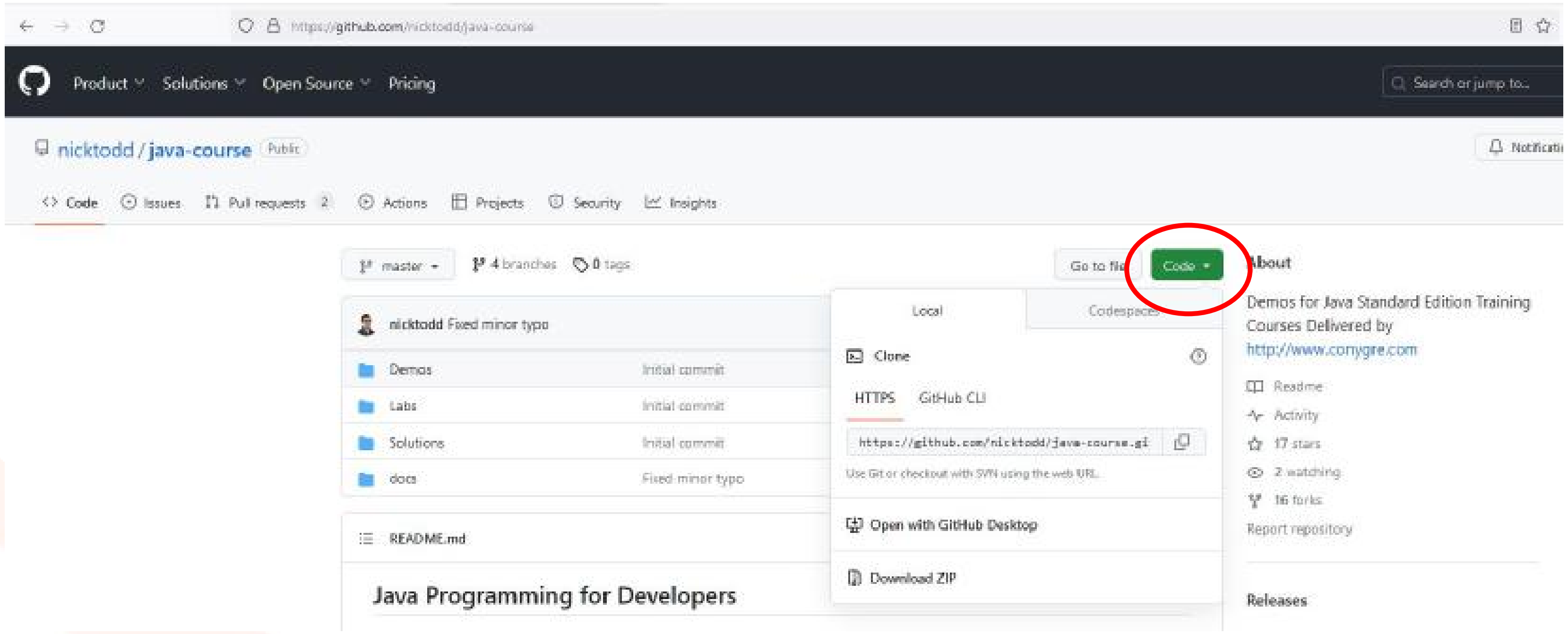


Here's where you'll find this repository's source files. To give your users an idea of what they'll find here, add a description to your repository.

| Name | Size | Last commit | Message |
|---|---|---|---|
| .gitignore | 624 B | 14 minutes ago | Initial commit |
| README.md | 2.56 KB | 14 minutes ago | Initial commit |
| funfacts.txt | 95 B | 3 minutes ago | Demonstrated how to add files to a repository |

# Copy the command from the GitHub Web UI

- The Github Web UI has a “Code" button
  - Gives you the git URL to clone the repository



```
https://github.com/nicktodd/java-course.gi
```

Use Git or checkout with SVN using the web URL.

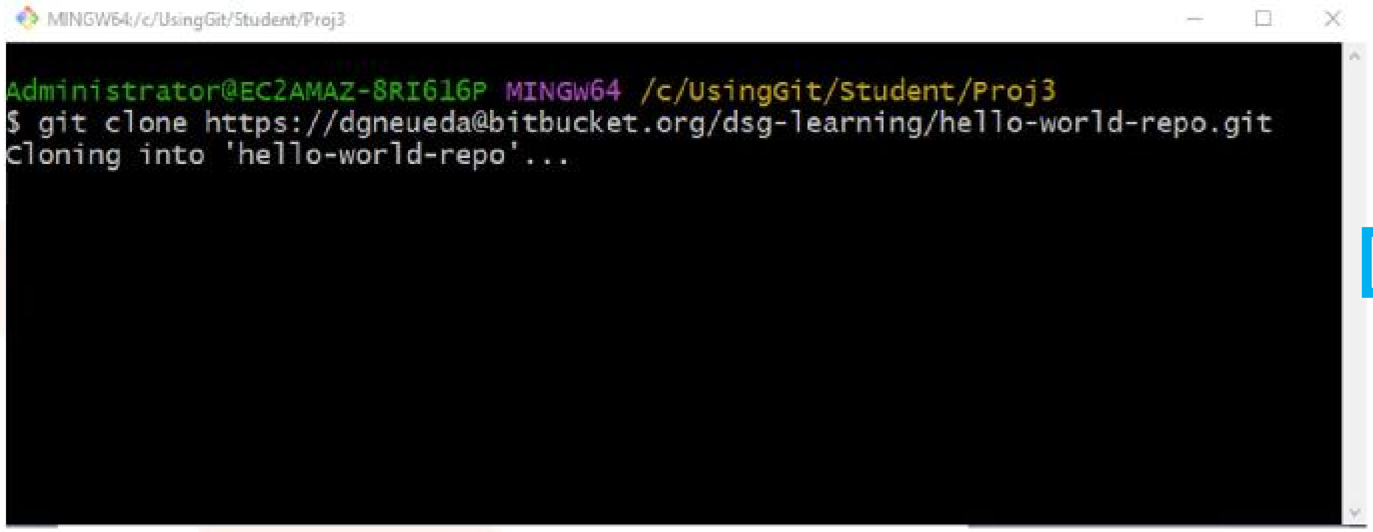
Demos for Java Standard Edition Training Courses Delivered by http://www.conygre.com

Java Programming for Developers

## Cloning a Repository via the Git Command Line (1 of 2)

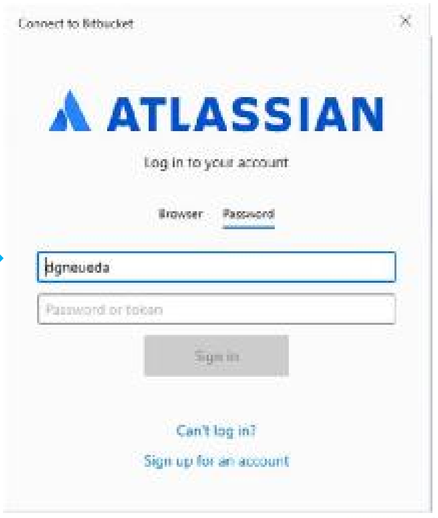
- Move into the Proj3 folder and clone the remote repository via the `git clone` command
  - Specify the URL of the repository you want to clone

```
git clone https://dgneueda@bitbucket.org/dsg-learning/hello-world-repo.git
```



MINGW64:/c/UsingGit/Student/Proj3

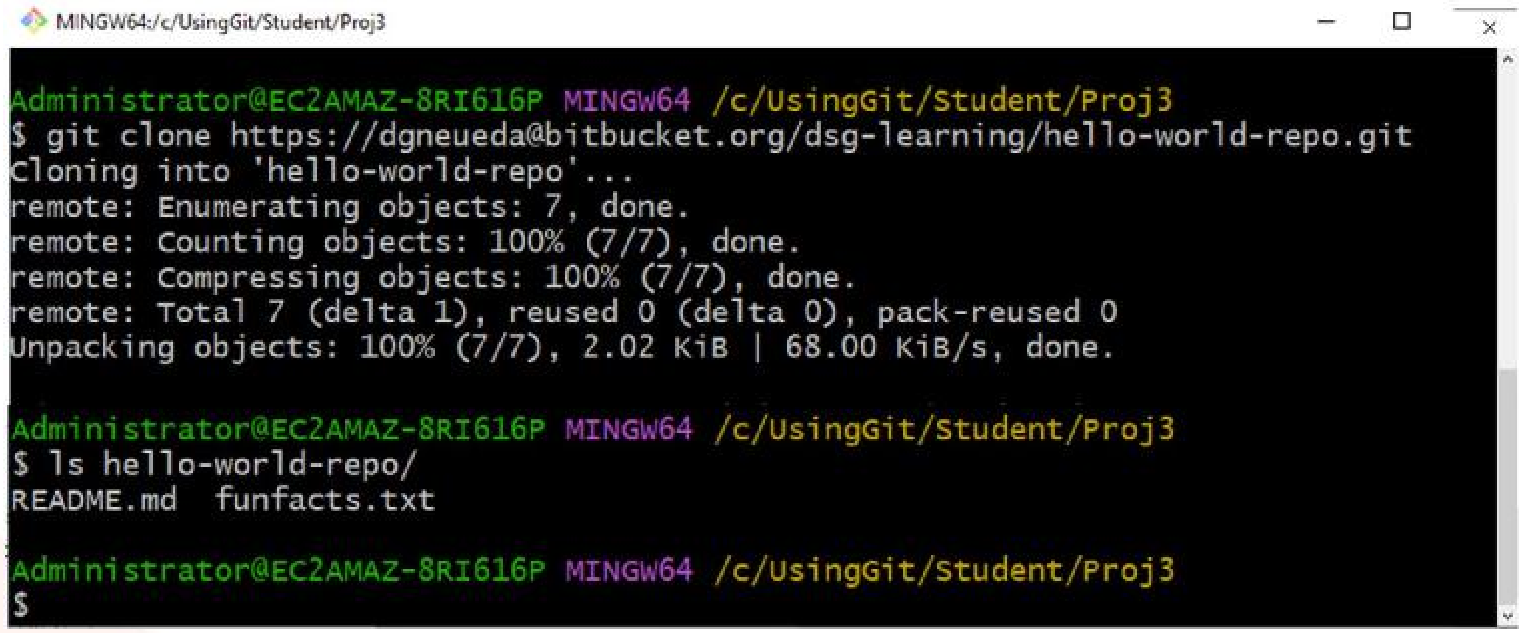
```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3
$ git clone https://dgneueda@bitbucket.org/dsg-learning/hello-world-repo.git
Cloning into 'hello-world-repo'...
```



Connect to Bitbucket

# Cloning a Repository via the Git Command Line (2 of 2)

- The authentication should not need to be repeated for this account
- The repo downloads into your local directory

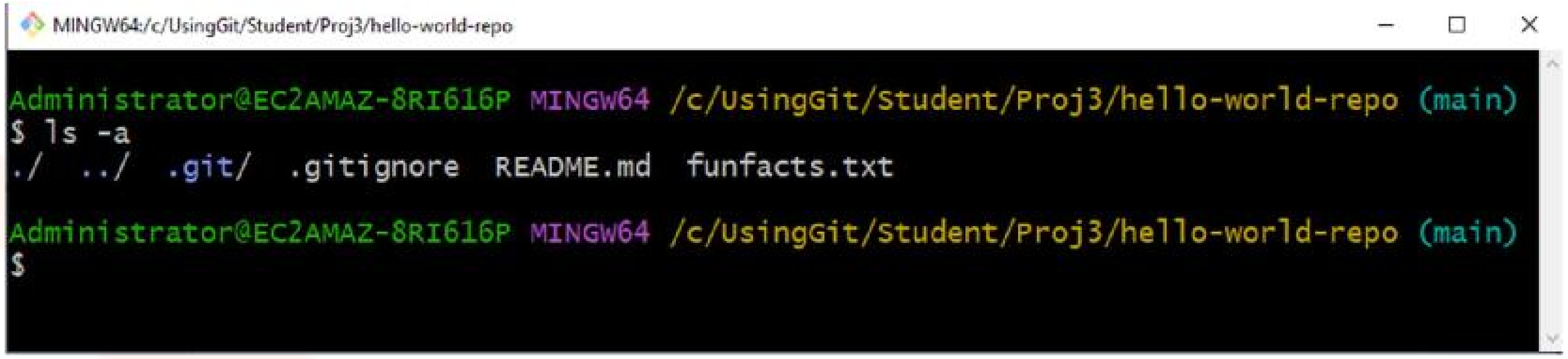


MINGW64:/c/UsingGit/Student/Proj3

```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3
$ git clone https://dgneueda@bitbucket.org/dsg-learning/hello-world-repo.git
Cloning into 'hello-world-repo'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (7/7), 2.02 KiB | 68.00 KiB/s, done.

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3
$ ls hello-world-repo/
README.md  funfacts.txt

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3
$
```

## Understanding Git Cloning (1 of 3)

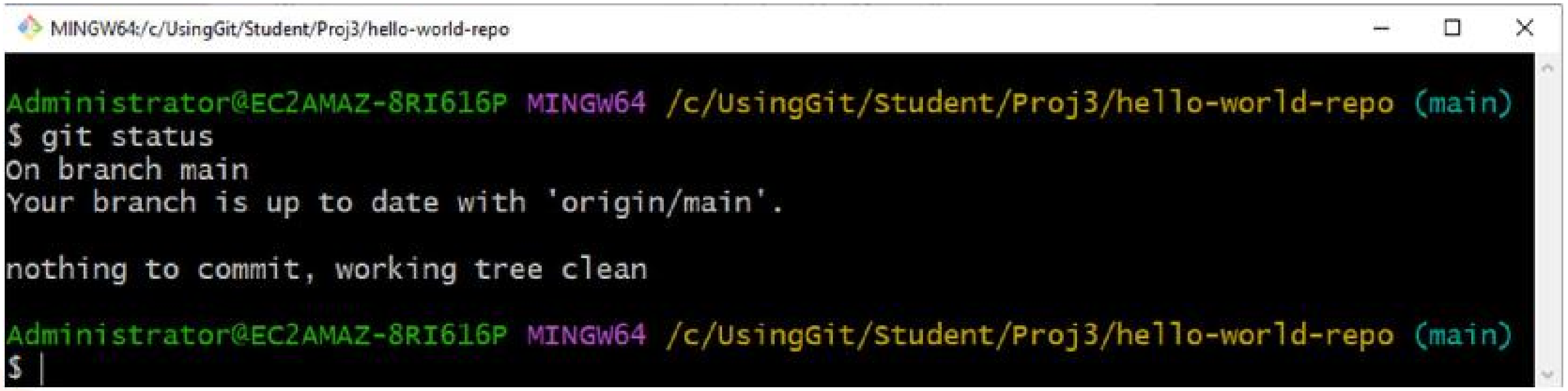
- When you clone a Git repository…
  - Git creates a local folder, with the same name as the repository
  - This is your local working directory, contains the latest committed version of all the files from the remote repository
- The working directory also has a `.git` folder
  - This is your local repository
  - You can work with files locally, and commit to your local repository when you're ready



```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ ls -a
./  ../  .git/  .gitignore  README.md  funfacts.txt

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$
```

## Understanding Git Cloning (2 of 3)

- Take a look at the current status of your local repository
  - It indicates there are no changes yet
  - You haven't made any changes to files in your local repository



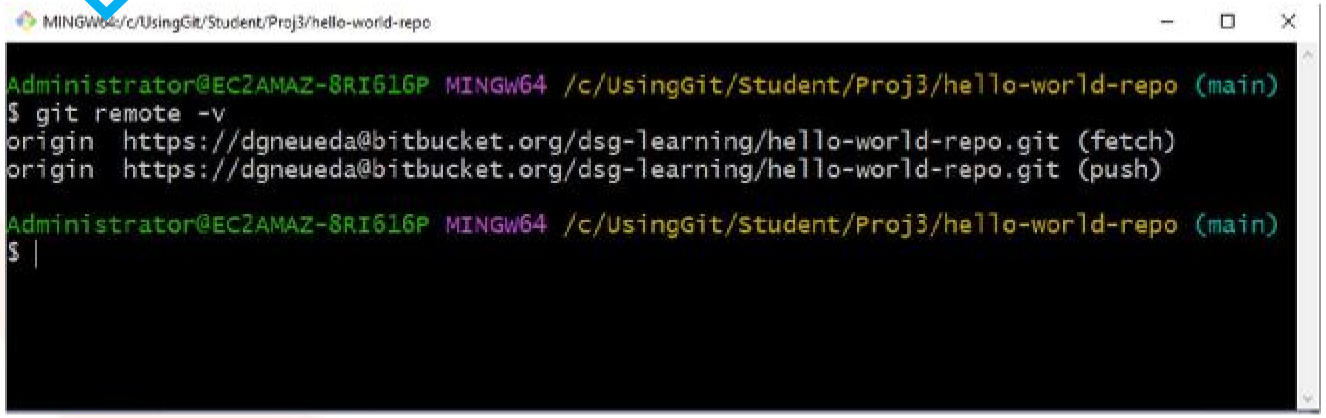
MINGW64:/c/UsingGit/Student/Proj3/hello-world-repo

```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ |
```

## Understanding Git Cloning (3 of 3)

- To see which remote servers you've configured, use the `git remote` command
  - If you've cloned your repository, Git uses the default name `origin` for the remote repository
  - You can use the `-v` option to show the detailed URL

```
git remote -v
```



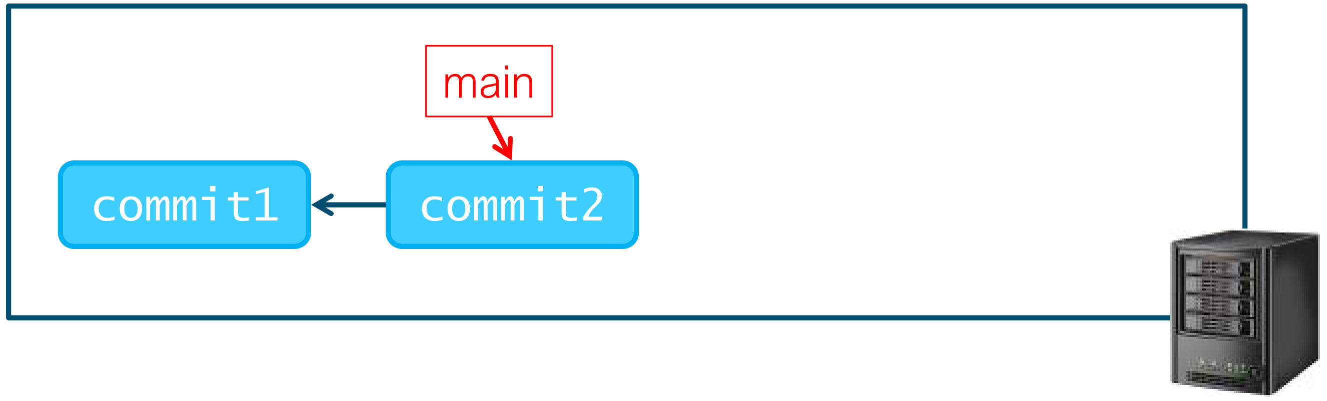
```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git remote -v
origin  https://dgneueda@bitbucket.org/dsg-learning/hello-world-repo.git (fetch)
origin  https://dgneueda@bitbucket.org/dsg-learning/hello-world-repo.git (push)

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ |
```

## 2. Understanding Remote Branches

- Branches on the server
- Branches on your local machine
- Listing all the branches on your local machine
- Making additional commits on the server
- Fetching updates from the server
- Merging updates into your local branch
- Making additional commits locally
- Pushing local commits to the server

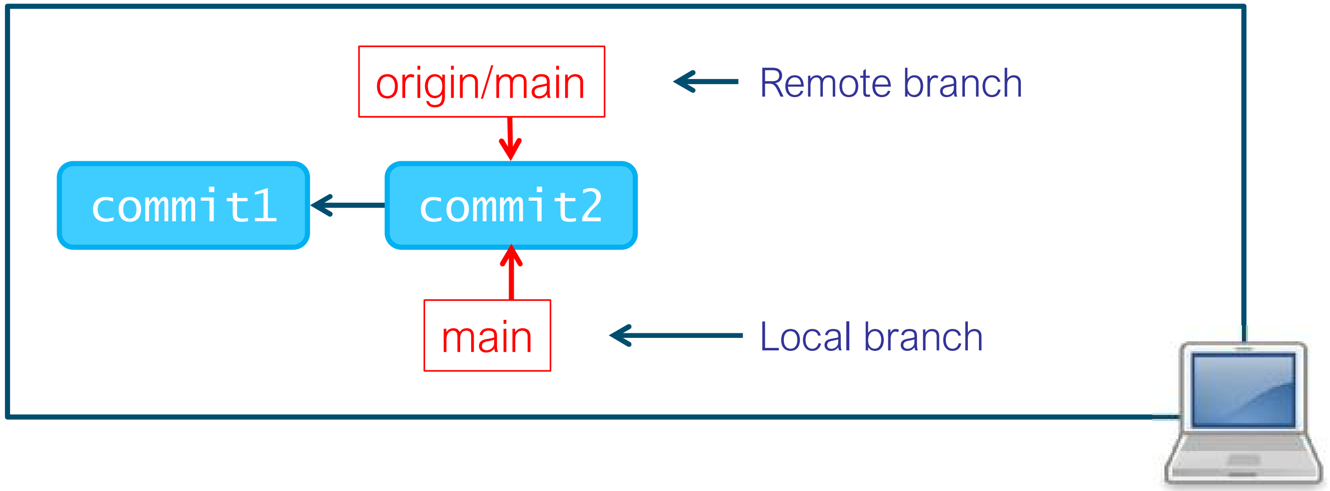
## Branches on the Server

- Let's assume this is the current state of play in our Git repository on the server



## Branches on Your Local Machine

- When you clone the repository onto your local machine, Git creates <u>two</u> branches

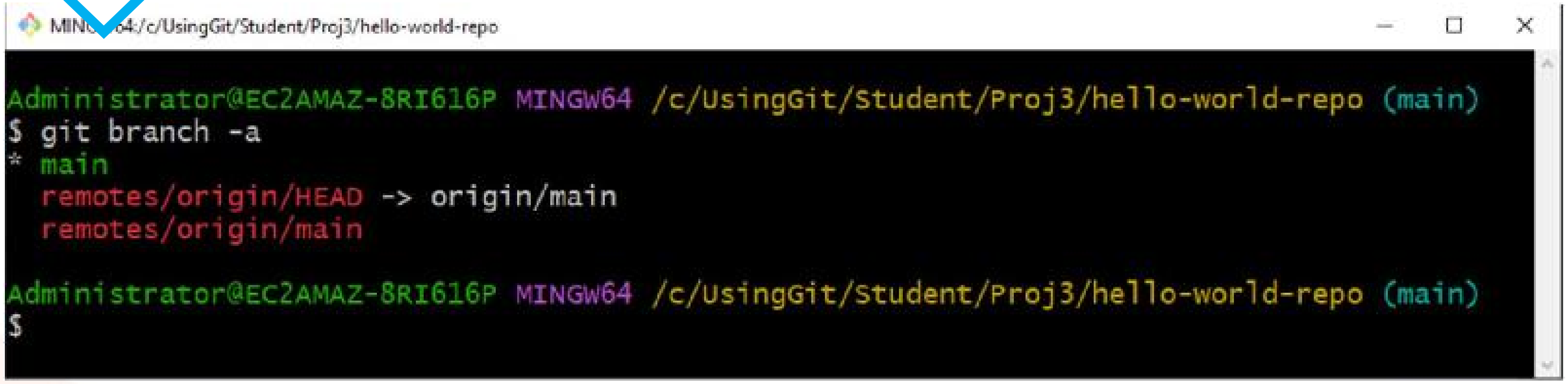


- origin/main is a <u>remote branch</u>
  - It's like a bookmark, indicates where the main branch was on the server the last time you connected to it

## Listing all the Branches on your Local Machine

- You can list all the branches on your local machine
  - i.e. all the remote branches and all the local branches

```
git branch -a
```



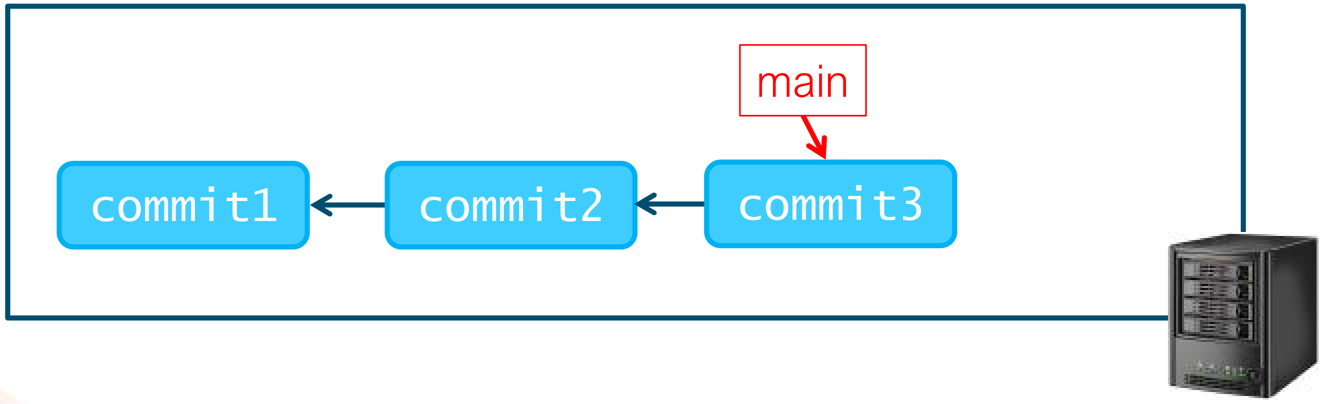
MINGW64:/c/UsingGit/Student/Proj3/hello-world-repo

```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$
```

# Making Additional Commits on the Server (1 of 2)

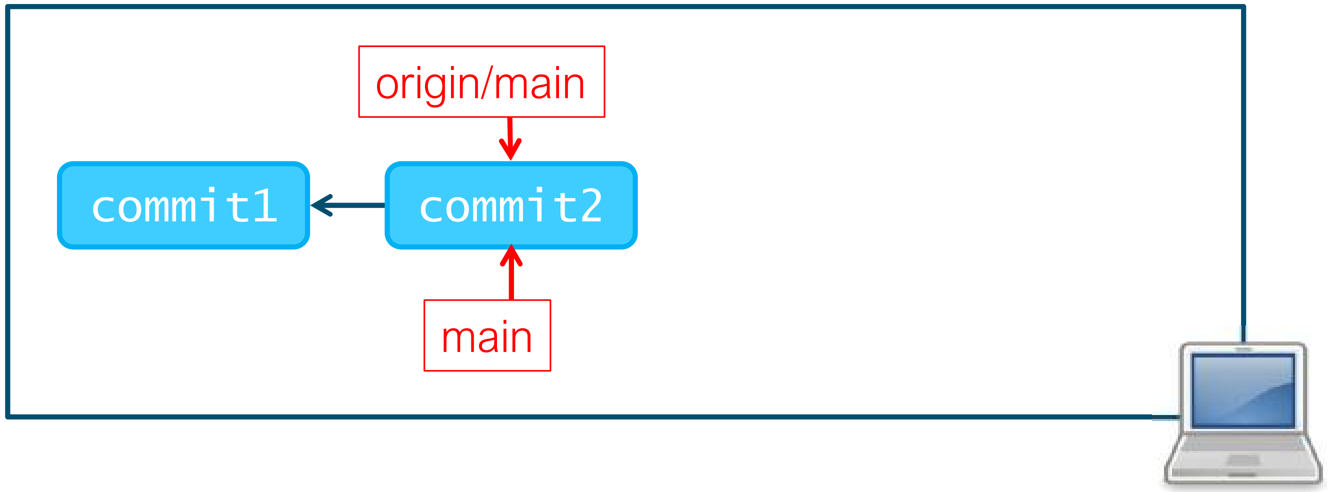
- Let's make some changes to files in the server repository
  - E.g. add a file, modify a file, etc.

- Let's commit these changes in the server repository
  - The main branch advances in the server repository



## Making Additional Commits on the Server (2 of 2)

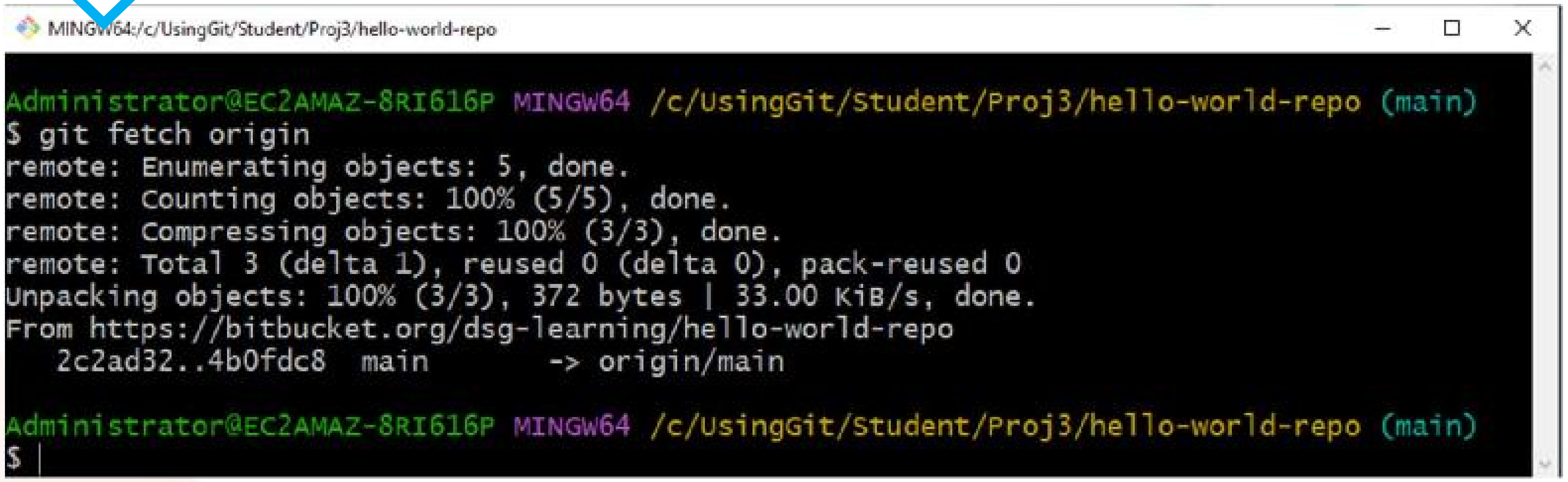
- However, the branches in your local repository are NOT automatically updated



# Fetching Updates from the Server (1 of 2)

- To sync the remote reference on your local machine with the latest commit on the server…
  - Use the `git fetch` command

```
git fetch origin
```

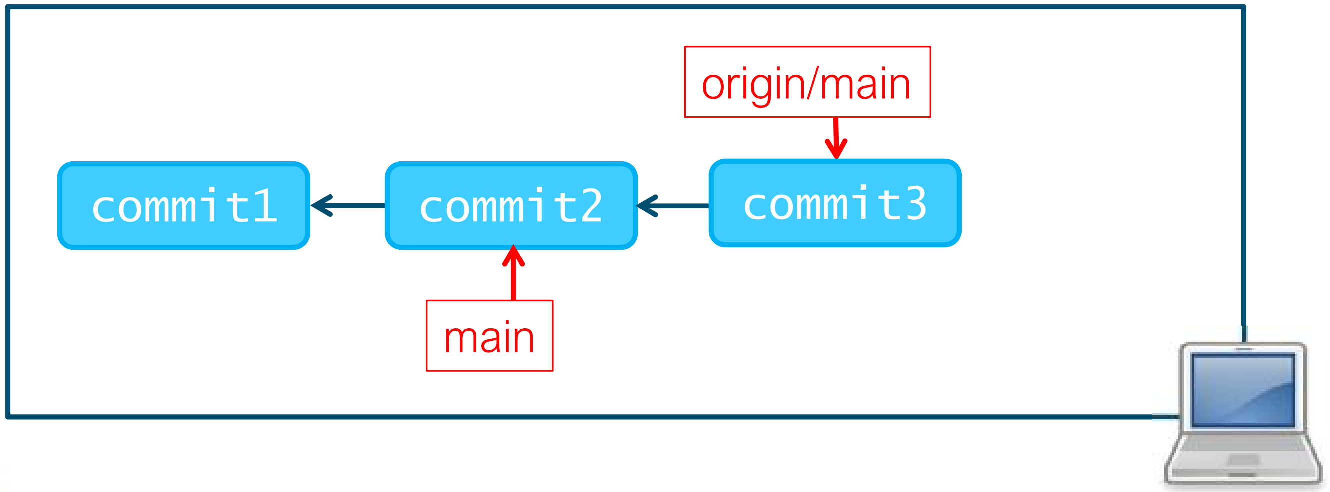


```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git fetch origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 372 bytes | 33.00 KiB/s, done.
From https://bitbucket.org/dsg-learning/hello-world-repo
   2c2ad32..4b0fdc8  main       -> origin/main

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ |
```

neueda

## Fetching Updates from the Server (2 of 2)

- `git fetch` updates your remote reference, so that it's synchronized with the current commit on the server

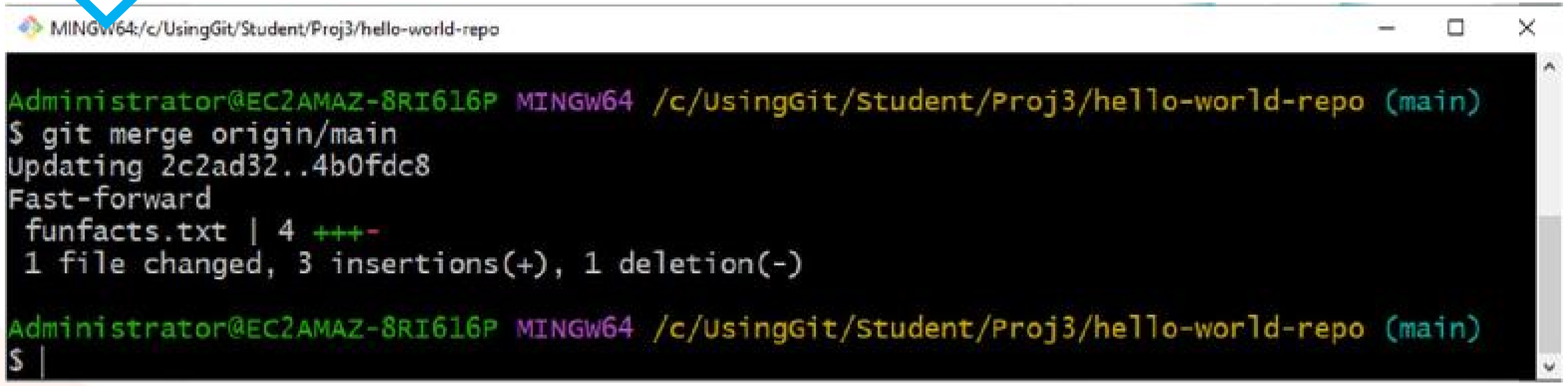


- However, `git fetch` doesn't update your local reference or the contents of your working directory
  - To do this, you must do a Git merge

## Merging Updates into your Local Branch (1 of 2)

- To merge the latest updates from the server into your local branch…
  - Use the `git merge` command

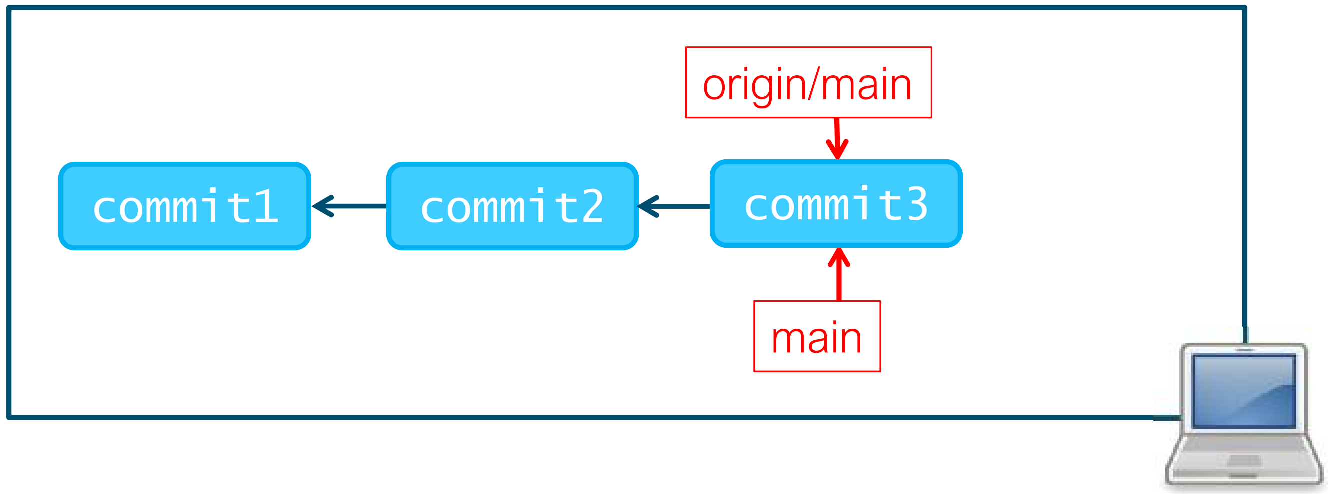
```
git merge origin/main
```



```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git merge origin/main
Updating 2c2ad32..4b0fdc8
Fast-forward
 funfacts.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ |
```

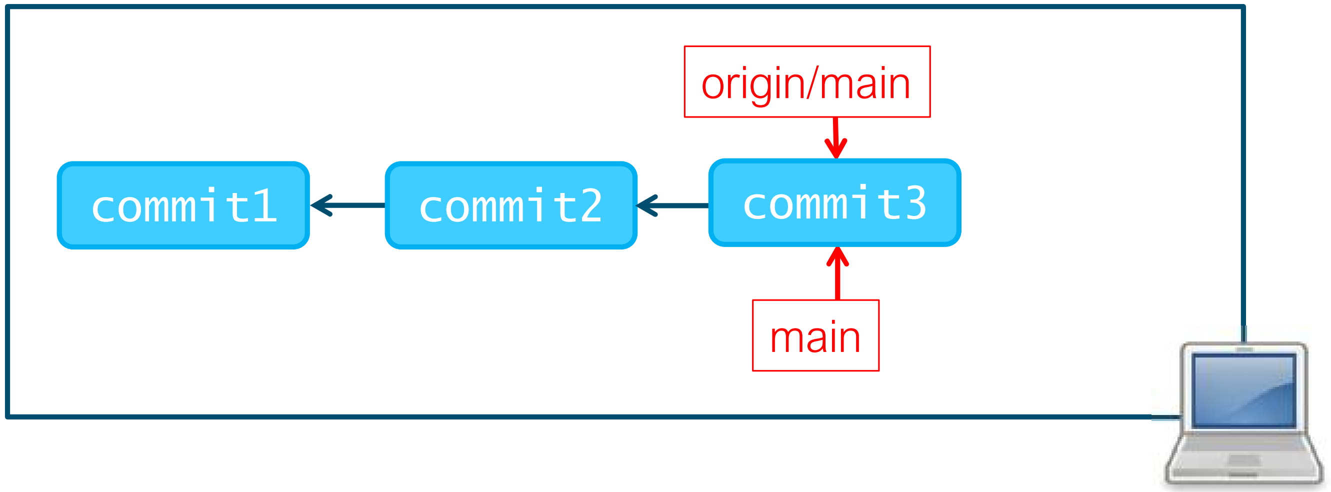
## Merging Updates into your Local Branch (2 of 2)

- `git merge` updates your local branch and the contents of your working directory



## Merging Updates into your Local Branch (2 of 2)

- `git merge` updates your local branch and the contents of your working directory



## git pull - an Alternative way to get Remote Updates

- We just did a “git fetch” AND “git merge” to get remote updates into our working tree
- An more commonly used alternative is “git pull” this does both of the above steps in one go.
- Try making another change directly on bitbucket.
- Do a “git pull” and verify the change appears in your working tree.

## Making Additional Commits Locally (1 of 2)

- Now let's make some changes to files in the local repo and get them into the remote repo
- Create a new local file

```
echo “New stuff” > someNewFile.txt
```

- Add to the staging area and commit to local repo

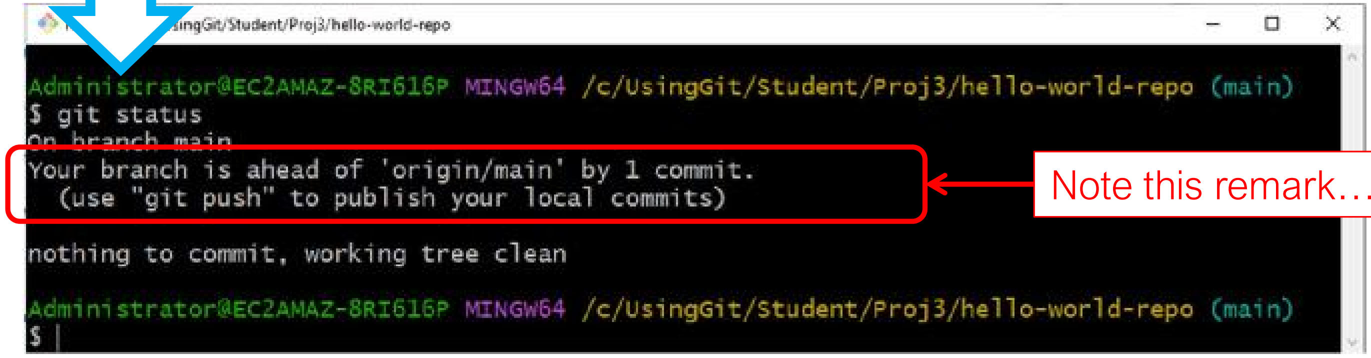
```
git add someNewFile.txt
```

```
git commit -m 'Added someNewFile.txt'
```

## Making Additional Commits Locally (2 of 2)

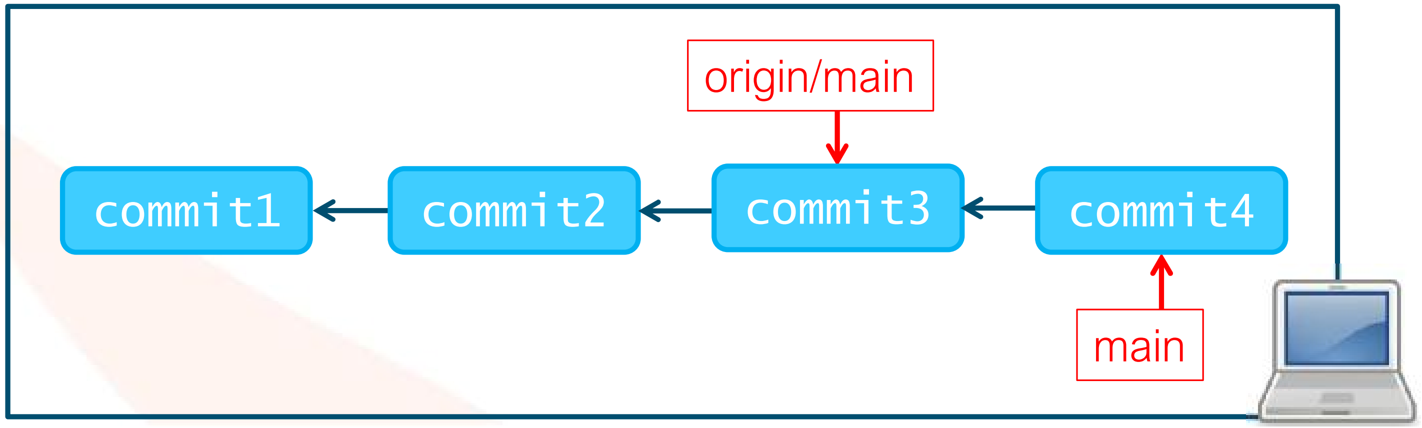
- Run `git status`, to see the status in the local repo now

```
git status
```



```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ |
```

Note this remark…

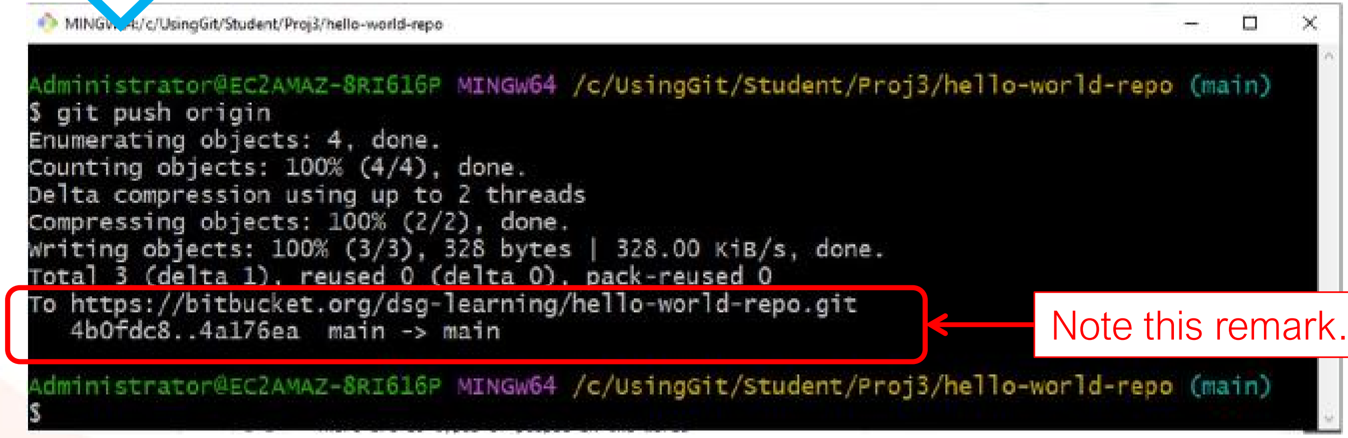


neueda

## Pushing Local Commits to the Server (1 of 3)

- To publish local updates to the server…
  - Use the `git push` command

```
git push origin
```



```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git push origin
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 328.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To https://bitbucket.org/dsg-learning/hello-world-repo.git
   4b0fdc8..4a176ea  main -> main

Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$
```

Note this remark…

## When you make a Mistake

- Git push will push your updates up to the origin, but what if you have made an error in what is committed or what is contained in your commit message?

- In those cases you can use **git push --force**

- The --force flag will take an amended local commit and ***overwrite*** the remote commit with your revised commit

- Using this option comes with a health warning when working with others, because if someone does a pull before you do your --force, they will now have a wrong version of your updated commit

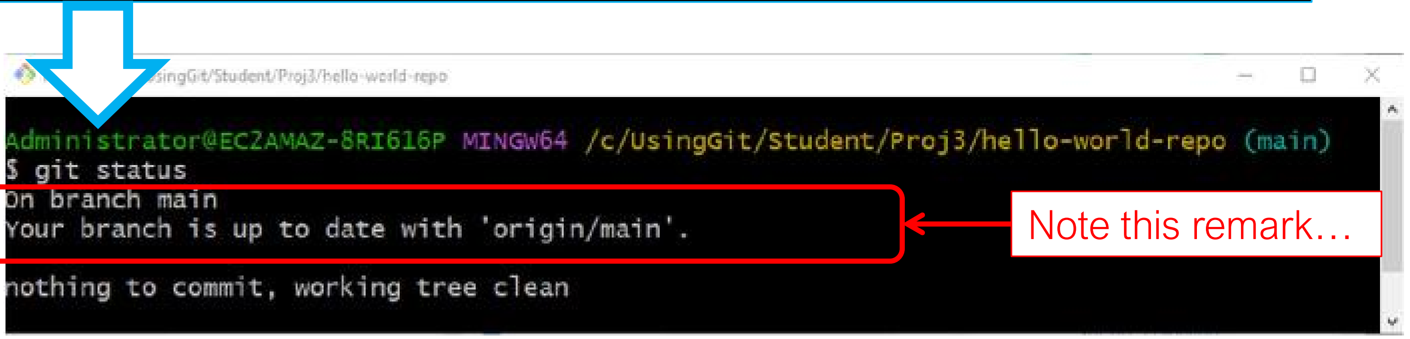
## Use Cases for git push --force

- **Rewriting Local History**
  - If you've made local commits with mistakes, and you want to rewrite your local commit history to correct them, you can use git push --force to update the remote branch with your corrected history
  - However, only do this if you are the sole contributor to the branch, or you have coordinated with other contributors
- **Syncing After Rebasing - (discussed later)**
  - When you rebase a branch, it rewrites commit history. If you've rebased a branch and want to push it to the remote repository, you may need to use git push --force
  - This should only be done if you are certain that nobody else has based their work on the previous version of the branch
- **Removing Sensitive Information**
  - If you accidentally commit sensitive information (like passwords or API keys) and need to remove it from the Git history, you may need to use git push --force to remove the sensitive commits
  - Inform your team about the necessity of this action and make sure everyone updates their local repositories to avoid recommitting the sensitive information.
- **Emergency Recovery**
  - In rare cases where a remote repository is in a severely corrupted state, and regular Git commands don't work, a force push can be used as a last resort to reset the remote repository to a known good state

# Pushing Local Commits to the Server (2 of 3)

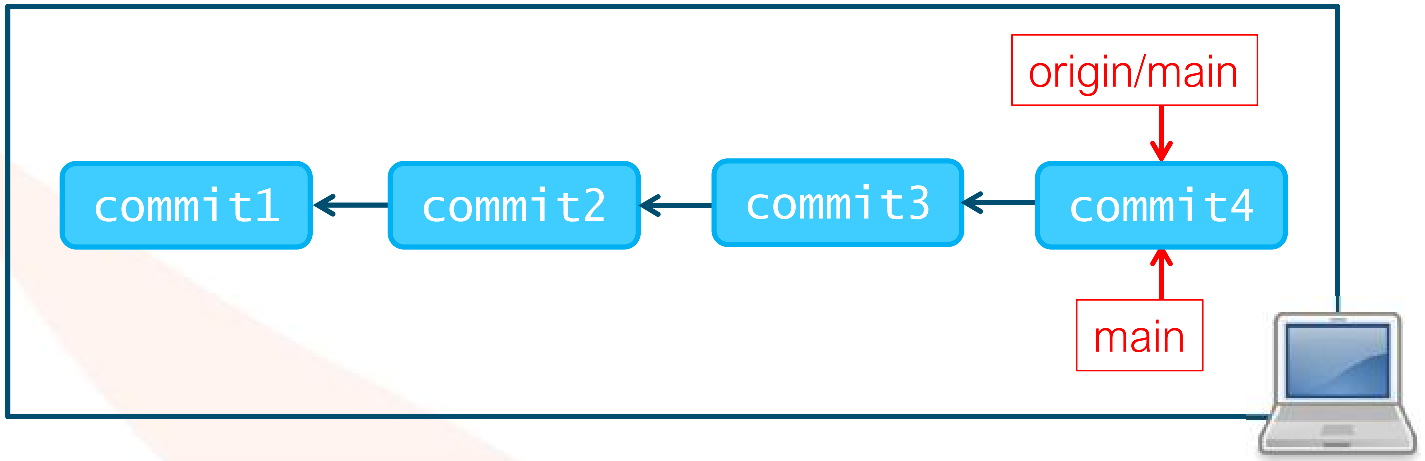
- Run `git status`, to see the status in our local repo new

```
git status
```



```
Administrator@EC2AMAZ-8RI616P MINGW64 /c/UsingGit/Student/Proj3/hello-world-repo (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

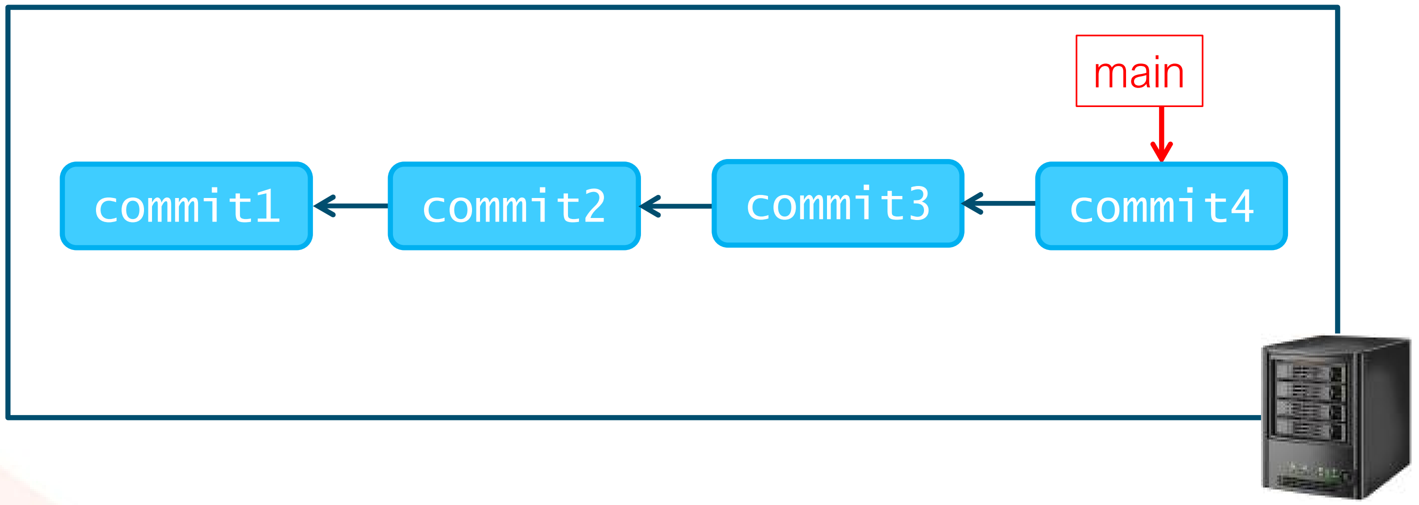
Note this remark…



neueda 29

## Pushing Local Commits to the Server (3 of 3)

- The server branch has also been updated
  - i.e. we have successfully published our local commit to the server



## Summary

- Cloning a remote repository
- Understanding remote branches

neueda

## Pull Requests

## Contents

1. Workflow Management in Git
2. What is a Pull Request
3. The Pull Request Workflow

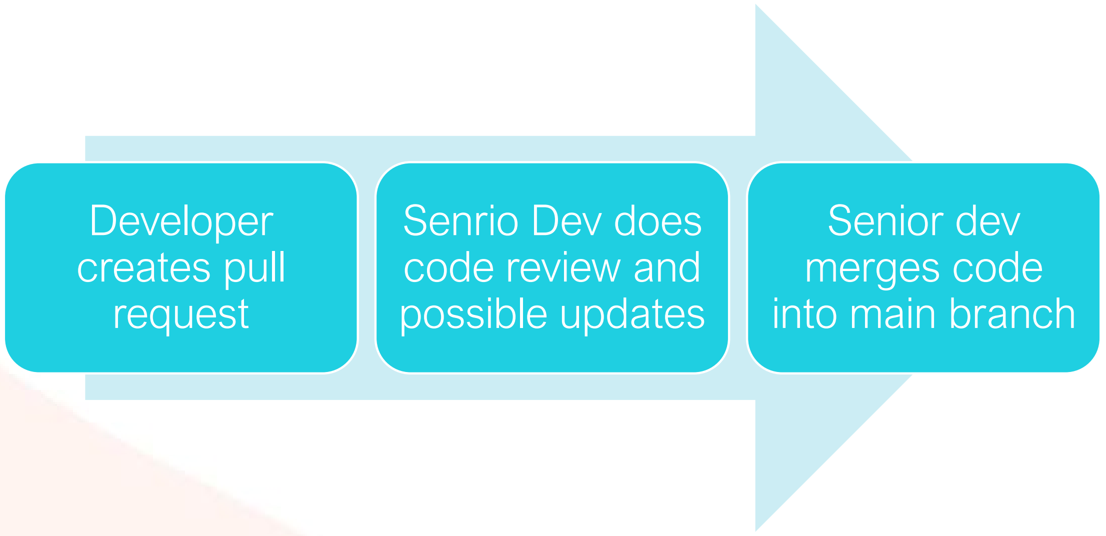
## Workflow Management in Git

- You have seen how you can merge one branch in Git into another branch
  - For example merging a feature branch back into the main branch once the feature is complete
- However, there are some questions such as
  - Who does that merge?
  - Does the merge need approval?
  - Do we want to run some tests on that feature branch code before we merge it into the main branch?
  - Has anyone reviewed the code?

- This is where the **Pull Request** comes in

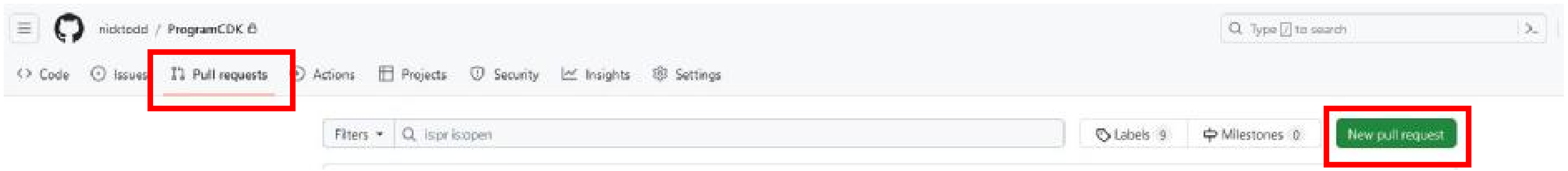
# What is a Pull Request?

- A Pull request is where you tell others about changes you've pushed to a branch in a repository
- Others can then open a pull request and
  - Review your changes
- Once a pull request is opened your changes can be discussed and reviewed
  - Follow-up commits may be added
  - Eventually, all being good, your changes are merged into the main branch

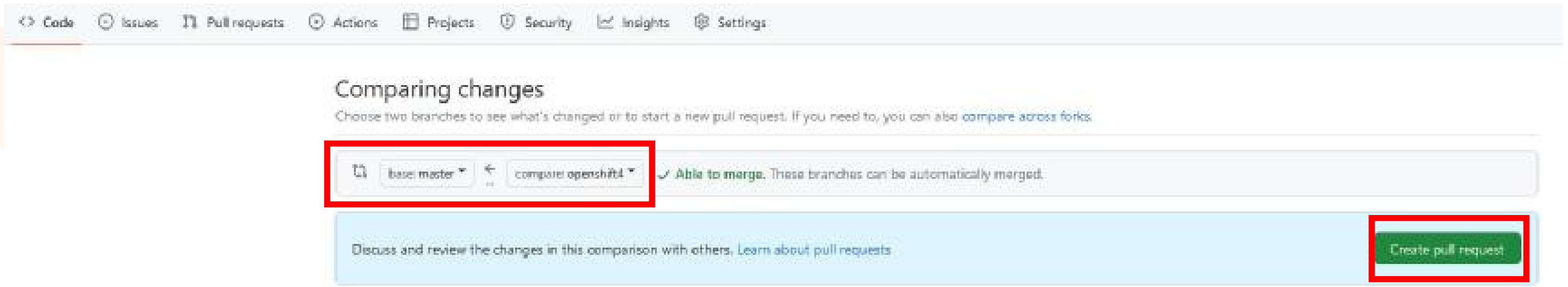

Developer creates pull request
Senrio Dev does code review and possible updates
Senior dev merges code into main branch

## Creating a Pull Request in GitHub

- In GitHub, you can create a pull request from the Pull requests menu option at the top of your repository home page



- You can then select the branches you wish to issue the pull request for



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

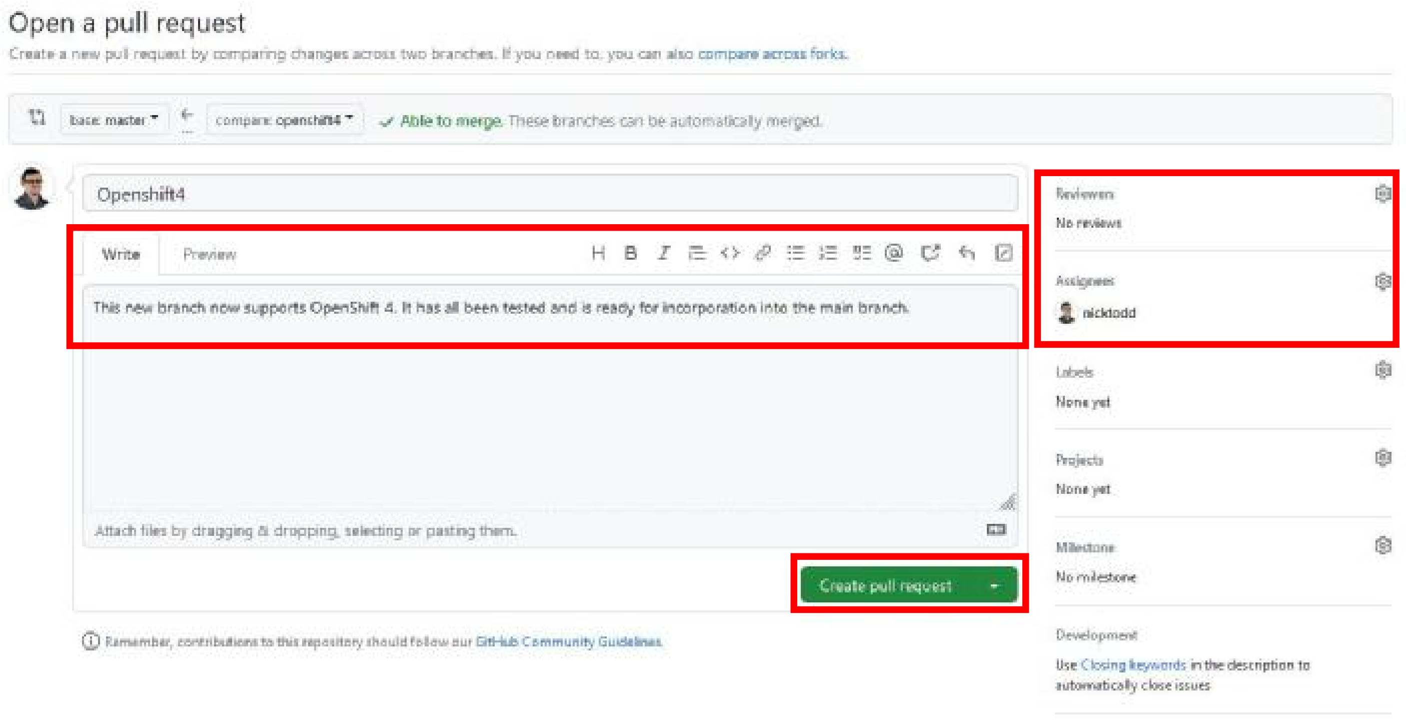
✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others. Learn about pull requests

neueda

## Adding Information to your Pull Request

- You can then add a description to your pull request and assign it to someone to review
- You can also add labels, a milestone and a project to help you track the pull request



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base: master ← compare: openshift4 ✓ Able to merge. These branches can be automatically merged.

Openshift4

This new branch now supports OpenShift 4. It has all been tested and is ready for incorporation into the main branch.

Attach files by dragging & dropping, selecting or pasting them.

Remember, contributions to this repository should follow our GitHub Community Guidelines.

Reviewers: No reviews

Assignees: nicktodd

Labels: None yet

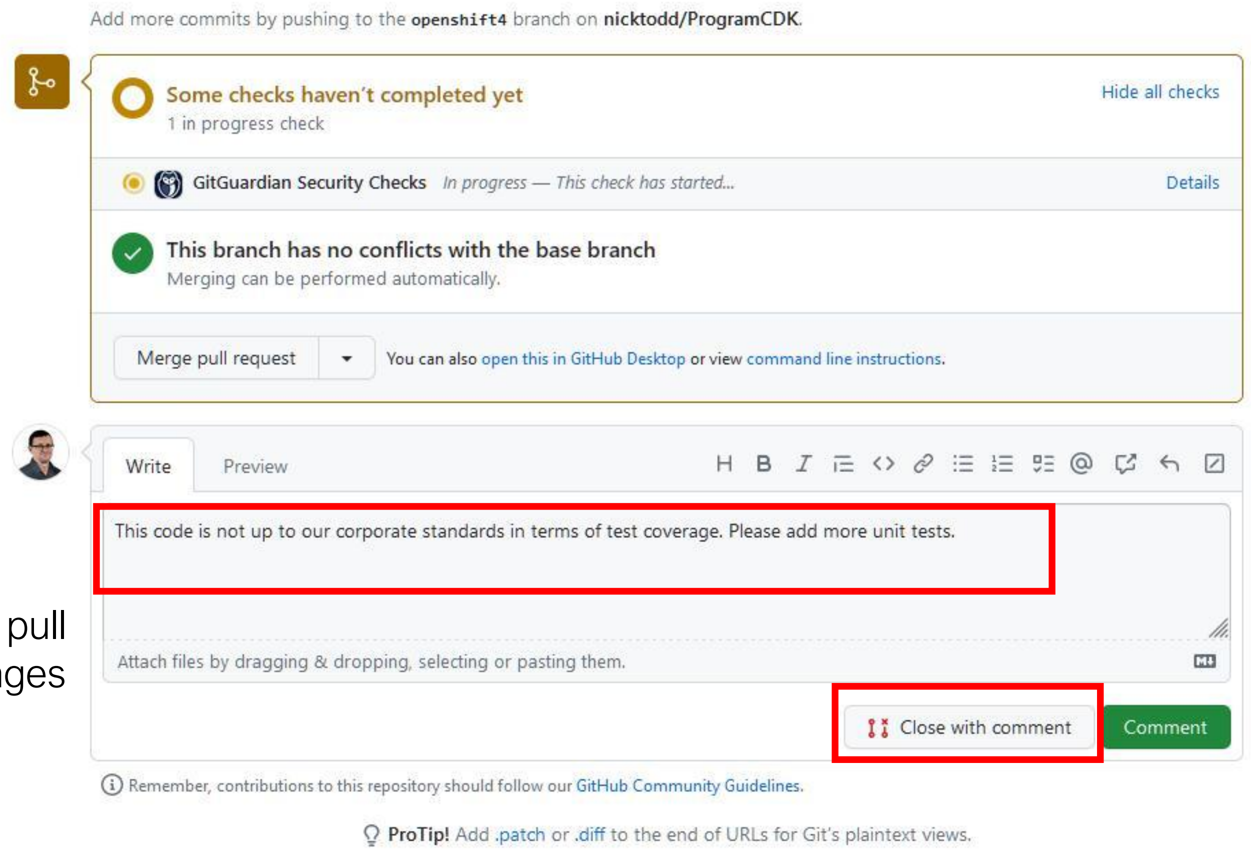
Projects: None yet

Milestone: No milestone

Development: Use Closing keywords in the description to automatically close issues

## Reviewing and Rejecting a Pull Request

- Once a Pull Request has been submitted, the reviewers can now review and accept / reject the request



Add more commits by pushing to the **openshift4** branch on **nicktodd/ProgramCDK**.

Some checks haven't completed yet
1 in progress check

GitGuardian Security Checks *In progress — This check has started...*

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

This code is not up to our corporate standards in terms of test coverage. Please add more unit tests.

Attach files by dragging & dropping, selecting or pasting them.

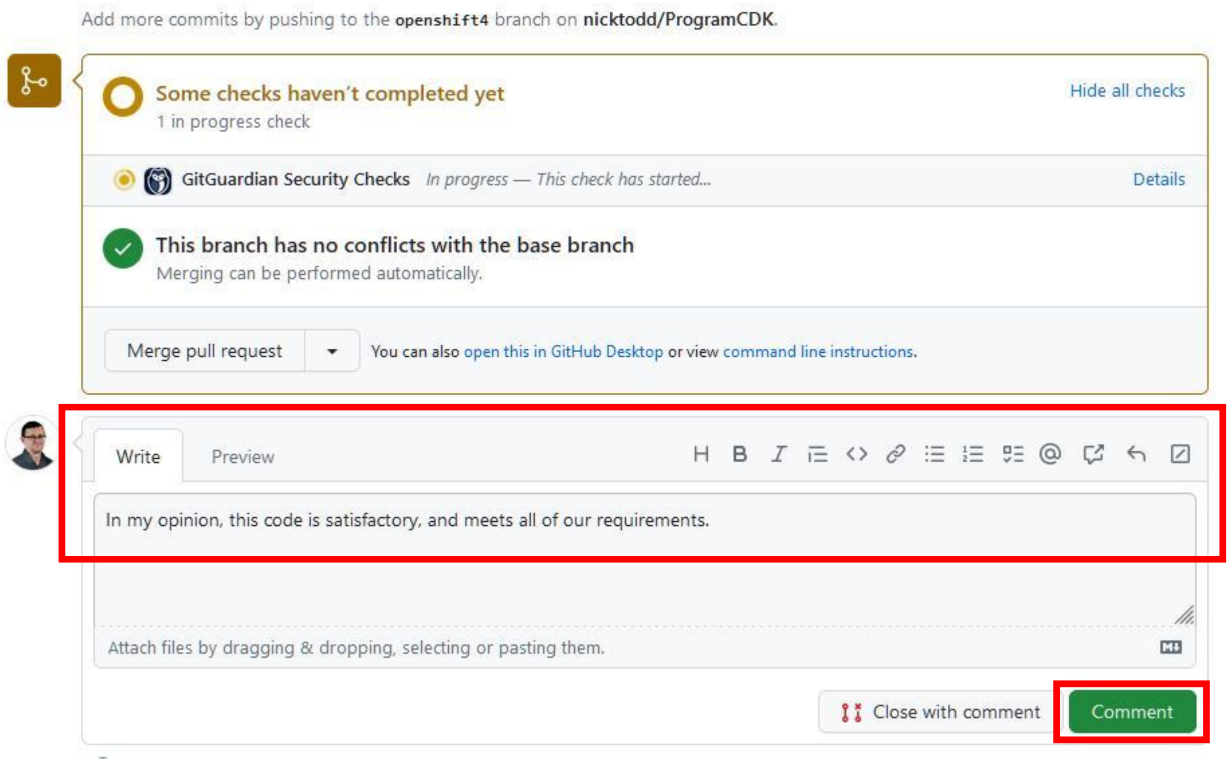
Remember, contributions to this repository should follow our GitHub Community Guidelines.

**ProTip!** Add .patch or .diff to the end of URLs for Git's plaintext views.

The reviewer could close the pull request without merging the changes

# Reviewing and Commenting on a Pull Request

- A code reviewer could make a comment without closing the PR or pulling anything



Add more commits by pushing to the `openshift4` branch on nicktodd/ProgramCDK.

Some checks haven't completed yet
1 in progress check

GitGuardian Security Checks *In progress — This check has started...*

This branch has no conflicts with the base branch
Merging can be performed automatically.

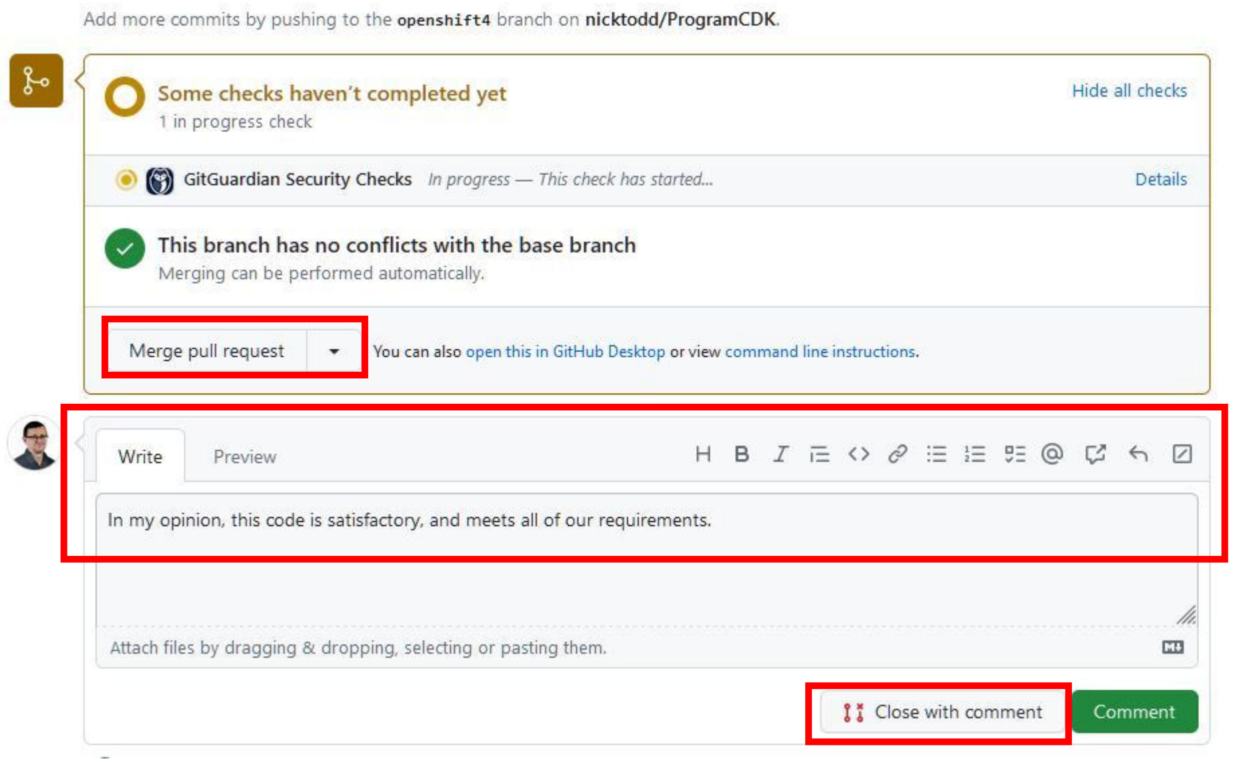
You can also open this in GitHub Desktop or view command line instructions.

In my opinion, this code is satisfactory, and meets all of our requirements.

Attach files by dragging & dropping, selecting or pasting them.

# Accepting a Pull Request

- Alternatively a reviewer could accept the code changes and complete the merge



Add more commits by pushing to the openshift4 branch on nicktodd/ProgramCDK.

Some checks haven't completed yet
1 in progress check

GitGuardian Security Checks *In progress — This check has started...*

This branch has no conflicts with the base branch
Merging can be performed automatically.

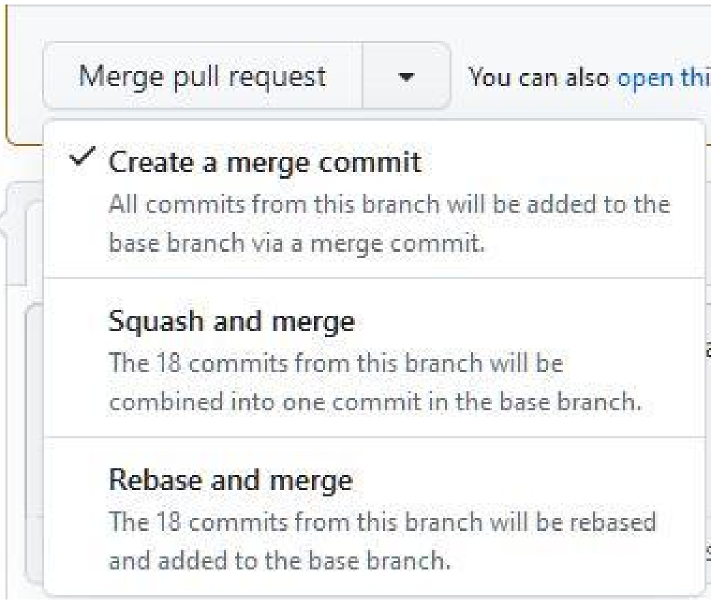
You can also open this in GitHub Desktop or view command line instructions.

In my opinion, this code is satisfactory, and meets all of our requirements.

Attach files by dragging & dropping, selecting or pasting them.

## Accepting the Pull Request - A Closer Look

- If you look at the options for accepting the Pull Request, there are more choices than simply **Merge pull request**
- The full set of options are
  - Create a merge commit
  - Squash and merge
  - Rebase and merge



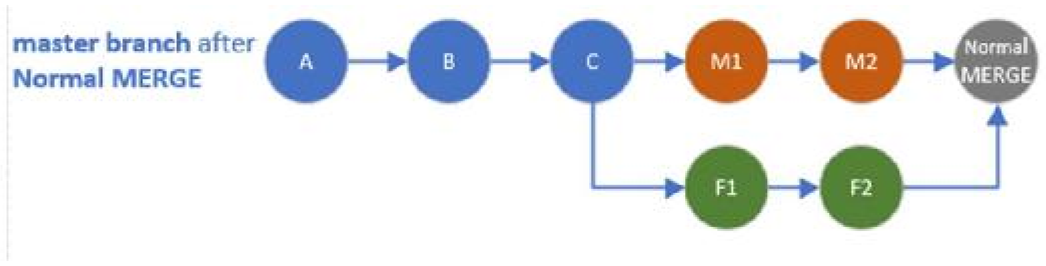
✓ Create a merge commit
All commits from this branch will be added to the base branch via a merge commit.

Squash and merge
The 18 commits from this branch will be combined into one commit in the base branch.

Rebase and merge
The 18 commits from this branch will be rebased and added to the base branch.

## Accepting the PR - A Merge Commit

- A Merge Commit is the most commonly applied option
- A merge commit is essentially just a simple merge as discussed before where the merged changes are then committed into the main branch



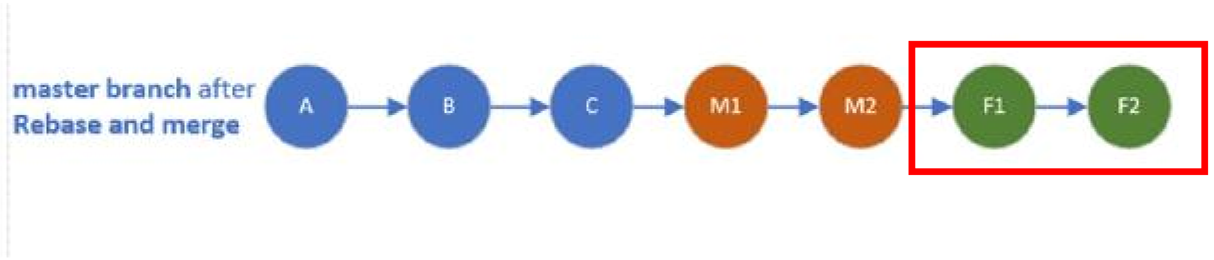
https://i.stack.imgur.com/hUtiB.png

## When to Use Merge Commit

- Merge Commit makes sense with feature branches when you want to maintain the history of the commits and the branch itself
- Merge Commit can also make sense when completing bug fixes so you can see the history of the commits and branches

# Accepting the PR - a Rebase and Merge

- A Rebase and Merge is where the feature branch commits are moved to the end of the main branch
  - The feature branch then disappears
- When you rebase a branch, it rewrites commit history. If you've rebased a branch and want to push it to the remote repository, you may need to use git push --force
  - This should only be done if you are certain that nobody else has based their work on the previous version of the branch



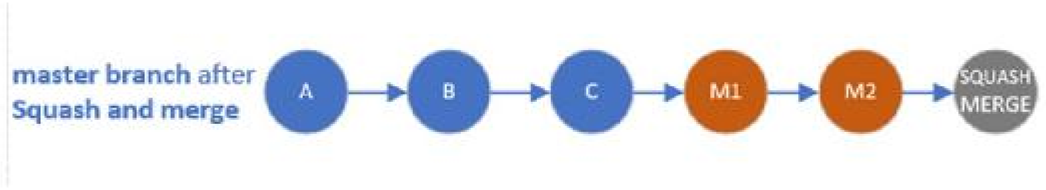
https://i.stack.imgur.com/hUtiB.png

## Rebase and Merge - When to Use

- Rebase and Merge is good for the situations where you want to maintain a clear linear history of commits
- When using rebase and merge, conflicts are addressed in smaller chunks as you merge changes from the original head into the new commits that are coming after

# Accepting a PR - a Squash and Merge

- The feature branch may have many commits in it, in a squash and merge, the feature branch commits compressed into a single commit that is then added to the main branch
  - Note that all the original feature commits have effectively disappeared



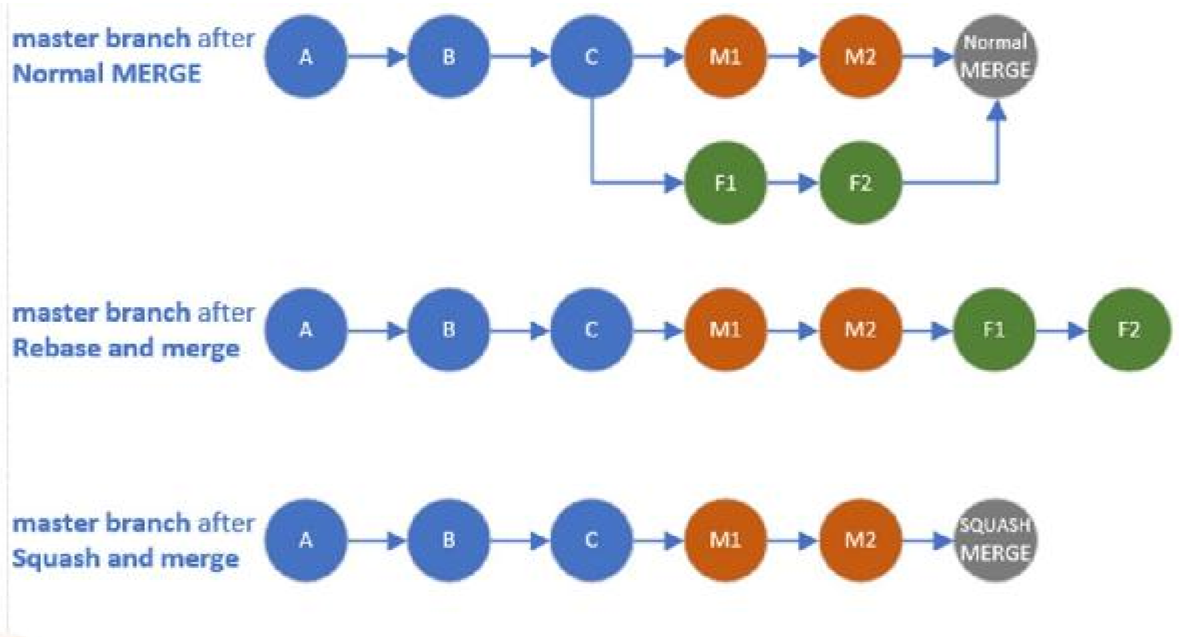
https://i.stack.imgur.com/hUtiB.png

## When to Use - Squash and Merge

- The benefits are similar to the rebase and merge, with the included benefit of removing lots of smaller commits from the history to keep it simple and concise

# Accepting the PR Request Summary

- Below are the three options presented as a single diagram


master branch after Normal MERGE
A
B
C
M1
M2
Normal MERGE
F1
F2
master branch after Rebase and merge
A
B
C
M1
M2
F1
F2
master branch after Squash and merge
A
B
C
M1
M2
SQUASH MERGE

https://i.stack.imgur.com/hUtiB.png

## Summary

1. Workflow Management in Git
2. What is a Pull Request
3. The Pull Request Workflow

# Git Lab – Committing Changes

## Objective

The objective for this exercise is to become familiar with basic operations using the git command line tool.

## Steps

1. In Proj1. Make a change to one of the java files. In this case don't worry about correct Java, just treat the file as text.

   e.g. open it in notepad or vs code and add a comment.

2. 'git status' to see what files have changed.

3. 'git diff' to see what lines have changed.

4. 'git add <Filename>' to get the change into the staging area

   (specify which file you're adding)

5. 'git status' to see the state of play

6. 'git commit –m <a good commit message>' to get the file into the local repo

   (include your commit message)

7. 'git status' to see that there are no differences between your working tree and local repo

8. 'git log' to get a list of the commits on this branch

9. Commit two more changes to get comfortable with the operations

# Git Lab – Branching & Merging

## Objective

The objective for this exercise is to become familiar with branching and merging operations with the git command line tool.

## Steps

1. You can start in the directory (it's just an empty directory):

   ```
   UsingGit/Student/Proj2a
   ```

2. Create an empty local Git repository

```
git init
```

3. Create some simple text files (e.g. file1.txt, file2.txt, file3.txt) and commit them to the repository (on the master branch)

   e.g. use notepad or VsCode to create your txt files

   ```
   git status

   git add .

   git status

   git commit -m "your comment"

   git status
   ```

4. Create a new branch named req1, representing changes you need to make to address some imaginary requirements (req1)

   ```
   git checkout -b req1

   git status
   ```

5. Now you should be on the req1 branch, though the files should currently be the same as they were on the master branch.

6. Make some changes to some of the files and commit these changes. These changes will be added to the req1 branch

   e.g. using notepad or VsCode make some changes to your files

   ```
   git add .

   git commit -m "your comment"

   git status
   ```

7. Revert to the master branch, and verify the files in your working directory have reverted back to the original versions prior to your most recent changes.

   i.e. Check that the master branch still has the old versions of the files.

8. Now make some other changes to the files and commit these changes. These changes wil be added to the master branch

   e.g. using notepad or VsCode make some changes to your files

   ```
   git add .

   git commit -m "your comment"

   git status
   ```

9. Now try switching back and forth between the branches with the files open in VsCode.

   Notice how the contents of the files changes to the current branch.

   ```
   git checkout master

   < take a look at your files >

   git checkout req1

   < take a look at your files again >
   ```

10. Now try manually "merging" the req1 branch into the master

    ```
    git checkout master

    git merge req1
    ```

11. If you get any "CONFLICT" messages then open the files in a text editor and fix them.

    To do this, look for any lines containing:

    '<<<<<<<<<<<<<<', '==============' or '>>>>>>>>>>>>'.

    Fix these lines so they contain the content you want.

12. Once you're happy with the contents of your files you need to add and commit them before the merge is complete.

    ```
    git add .

    git commit -m "Merged req1 branch into master"

    git status
    ```

13. Notice how all the commits are all now on the master branch, including those made on the req1 branch.

    ```
    git log
    ```

# Git Lab – Working in Teams

## Objective

The objective of this lab is to become familiar with using git and a remote repository system. The

remote repository system described in the lab is BitBucket. If you wish to use a different one such as GitHub, you may need to modify some of the steps.

In particular this lab covers:

• Creating feature branches for new work

• Creating a pull requests to merge work into "upstream" branches

• If there are merge conflicts you will need to merge manually e.g. using the gitbash command

line

## Steps

1. Note you will work in teams but with a SINGLE repository. You will share access to this repository.

2. One team member create a SINGLE Bitbucket repository, make it public so that your team and the instructors can all browse it and READ from it. Include a README when you create the project. This means the project will have a single file in it when it's created.

3. A public project can be READ by anyone, however your team will also need WRITE or ADMIN access to the project so that they can make code changes.

4. Give your team all have write permission to the repository e.g. On your bitbucket repository web page go to: Repository Settings -> User and group access -> Add members

   You will need to add the email address that each team member uses for bitbucket.

   Each team member will then get an "invitation" email notifying them that they have WRITE access to the repository.

5. **Each** team member will make some changes to the README over the next steps. So **EACH** team member should clone the repository to their own machine using "git clone…" (the bitbucket page will show the clone command so you can copy it).

   For example (your URL will be different):

   git clone https://bitbucket.org/teammember1/exercise-repo.git

6. **One** team member should make a change to the README file with any simple text change e.g. add a message in it with your name. This team member should now make this change available to the rest of the team with "add, commit, push" to get this change to the remote (bitbucket) repository.

7. Verify that you can all see the change in bitbucket, look under the source and commits tabs to see the change.

8. **Each** team member should now run 'git pull' so that they retrieve this new change from the bitbucket repository**.**

9. Now in turn **Each** team member should now make their own unique change or changes to the README file. Add, commit and push your change so that it's available to the rest of your team. AND each other team member should use 'git pull' to bring that change to their local copy of the repository.

10. Repeat step 9 until a change from every team member has been added to the file.

In the above steps we have see every team member add a change **TO THE SAME BRANCH** of a repository. While this is a useful first step in using remote repositories, in production scenarios we would use new branches for different changes (we would not all commit our changes to the "master" branch).

## Optional – Advanced Steps – Using branches

**IF** your team understands the above steps, then try repeating them, however in this case when each team member makes their change in Step 6. They should this time create a new branch first E.g.

1. Create a new branch for your particular change:

   ```
   git checkout -b teammember1_branch
   ```

2. <make your change to the README file>

3. Add your changes to the repository

   ```
   git add .
   ```

4. Commit your changes to the repository

   ```
   git commit -m "Adding a test change to README"
   ```

5. Push your changes to bitbucket BUT now going to the new branch

   ```
   git push -u origin teammember1_branch
   ```

In the commands above, we have created a new branch, made a change on it and pushed it to bitbucket so that new branch is available to the team.

Each time a team member completes the steps above, you should then bring these changes from the new branch over to the master branch. Do this by creating a "Pull Request" on the bitbucket web screen. This pull request will be from "teammember1_branch" to "master" the branch changes are coming from and the branch changes are going to.

When the pull request has been created, you should then “Merge” the pull request to complete the process.

When the pull request has been “Merged” then every team member should run ‘git pull’ on their own machine to take these changes down to their local copy of the master branch.

The next team member can then repeat the process of creating a branch, committing changes to it, pushing it to bitbucket AND merging it by creating a Pull Request.

The above is a more realistic workflow for Teamwork with Git.