ASSIGNMENT NO. 01 - Software Engineering – BSCS V,– FALL 2017

MARKS: 100 ***DUE DATE: SEP, 18TH, 2017***

A COMPREHENSIVE HOME TAKS FOR LEARNING AND PRACTICING GIT AND GITHUB

DON'T PANIC JUST FOLLOW THE INSTRUCTIONS

Git vs Github

Git is a distributed revision control system.

GitHub is a web-based Git repository hosting service. GitHub offers both public free accounts and paid plans for private repositories.
GitHub takes version control a step further by facilitating **social coding**. GitHub makes it easy to contribute to other projects or accept contributes to a project you started. GitHub is located on the web at github.com.

If you don't already have an account at github, create one now. The free account option is sufficient for this exercise.

You will also need a git client installed on your local machine. If you don't already have a git client installed on a local machine, install one now. Github for Windows (http://windows.github.com) is a popular choice but there are other options including command line clients for Mac, Windows and Linux. For this exercise, I used the Windows command line client:

http://git-scm.com/download/win

There are GUI clients for using git, but the advantage of using the command line is:

1. It doesn't change. Different GUI clients fall in and out of favor and how a GUI client works can change from one version to another.
2. All major GUI clients also offer a command line option.
3. To debug a problem from a GUI interface, it helps to know the git command being issued.

Exercise 0 – Putting A C++ Project on GitHub

This exercise will show you how to put a local C++ Project on GitHub.

Step 1. Create an A C++ PROJECT IN DEV OR CODEBLOCKS. I called mine CALCULATOR.

Step 2. Create a repository on GitHub of the same name. If you don't add a Readme file when you create it, GitHub will show you the git commands needed to add your project:

```
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/SOFTWAREENGAUMC/CALCULATOR.git
git push -u origin master
```

After a change to your project:

```
git add .
git commit -m "short description of change"
git push -u origin master
```

Git + Github Exercises

Exercise 1 – Creating and using a local repository

This exercise will give you practice with:

- Creating and using a local repository
- Retrieving old versions of a file.
- Using a .gitignore file to keep certain files from being tracked
- Creating a repository on github.com
- Committing and syncing (or pushing) changes to github

Open one of the git shells (command line tools) you downloaded above. I recommend the git bash shell.

Create a directory somewhere on your file system for practice and make this the current directory in the command shell. It's OK if there are existing files in the directory.

Show your current directory (pwd = print working directory):

$ pwd

Create a directory:

$ mkdir gitpractice

$cd gitpractice

Start tracking files and subfolders with git:

$ git init

Run git status to see current status of the repository:

$ git status

The first time you use the shell, it is a good idea to set up your identify for committing changes.

To set your identity, enter:

$ git config --global user.name "Your Name"

$ git config --global user.email you@example.com

Check values you just set:

$ git config --list

You may also want to change the editor. To change the editor to notepad, enter:

Hopefully before this you have set username and Email through git config command

$ git config --global core.editor notepad

Add a new file to the directory. For example, use notepad to create a file:

$ notepad hello.txt

And add the contents:

```
// Hello world
void main() {
    print "Hello World";
}
```

Run git status:

$ git status

You should see the file as “untracked”.

Add the file with:

$ git add hello.txt

When there are new or changed files in your directory, you have to add them before they will be included in a commit. This is a nice feature because it prevents generated binary files or other new files you don’t want to be tracked from being included in a commit.

Run git status to verify the file was staged:

$ git status

Note, you can use a regular expression to add more than one file. Example: git add *.c

The add command takes the name of a file or directory. If a directory is specified all the contents of the directory and subdirectories are added. For example:

$ git add src

To commit all changes that have been added, enter:

$ git commit –m ‘description of change’

To verify the commit was successful, run git status:

$ git status

Make a change to the file using notepad or other editor.

Run git status:

$ git status

You should see the status of the file as changed but not staged.

Stage it with:

$ git add hello.txt

Run git status:

$ git status

You should see the status of the file as staged but not committed.

Now, commit the changes with:

$ git commit –m 'description of change'

Note, git add stages changes to be committed. When you commit you don't commit the current version of the file, you commit the state of the file at the point when it was lasted added. For example, if you do something like:

Change 1 → *add* → *change 2* → *commit*

Only change 1 will be committed. Try it. Before you commit, do a git status and you will notice the same file name listed as staged but not committed and as changed but not staged.

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what's been changed. For example:

```
Make greeting time dependent
- print Good Day if hour is 3am – 5pm
- print Good Evening if hour is 6pm – 2am
```

The best way to enter a multi-line comment, is to leave the –m option off the commit. The following will open the default editor for the commit message:

$ git commit

To see a history of changes:

$ git log

To see an abbreviated history:

$ git log –oneline

If there is more than one page, press space bar to advance to the next page or q to quit.

You can also, limit the number of previous commits to show. The following will show the last 3 commits:

$ git log –n 3

To show the differences introduced in each commit, use the –p option:

$ git log –p

You can of course, combine the two:

$ git log –p –n 3

There are other options for showing magnitude of changes with each commit, author, etc. (See: ***https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log)***

The 40-character string after commit is an SHA-1 checksum of the commit's contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

To get back to a previous state:

$ git checkout <hash from log command or tag>

The hash can be the full hash value from git log or the abbreviated one from git log –oneline.

Try checking out a previous commit. Browse the file to verify it is an older previous version.

To get back to the latest:

$ git checkout master

You can also checkout a single file from a previous commit:

$ git checkout <hash from log command or tag> filename

To get back to the latest version of the file, enter:

$ git checkout HEAD filename

Sometimes there are files in your directory you don't want tracked by git. For example, in most cases there is no need to track files created by the compiler (e.g. .exe, .class, etc.).

git won't track these files unless you add them, but they will show up as untracked when using the status command. If there are files you know will never be tracked, you can tell git to completely ignore them.

If you create a file in your repository named .gitignore, Git uses it to determine which files and directories to ignore.

A .gitignore file should be committed into your repository (add followed by commit), in order to share the ignore rules with any other users that clone the repository.

Use notepad to create a .gitignore file with the following contents:

```
// Ignore .class files and everything
//   in the bin directory
*.class
bin/
```

Add and commit the .gitignore file to your project:

$ git add .gitignore

$ git commit –m 'created .gitignore'

Now, create a bin directory, add a file to it and create a .class file:

$ mkdir bin

$ touch bin/somefile.exe

$ touch file.class

Now, check git status and notice these files are ignored (git doesn't flag these new files as untracked):

$ git status

Viewing staged and unstaged changes. git status gives a high-level picture of what files have changed and need to be added and/or committed. For more details on what has changed, you need git diff. git diff will tell you (1) what is changed but not staged, and (2) what is staged and about to be committed. Unlike git status, git diff shows exact lines changed/added/removed.

The following shows changed but not added:

$ git diff

The following shows exactly what will be added in the next commit (exact lines of each file):

$ git diff –staged

To delete a file from your project and remove it from being tracked, you can't just delete and commit. To delete a file enter:

$ git rm <filename>

$ git commit –m 'message'

git rm <filename> removes the file from the directory and causes git to stop tracking it. If you just want git to stop tracking it, use:

$ git rm --cached <filename>

If you want to rename a file, one option is to rename it outside of git and then delete the old one and add the new one in git:

$ mv oldfile newfile

$ git rm oldfile

$ git add newfile

Or, you can accomplish the same using the git mv command:

$ git mv oldfile newfile

Exercise 2 – Cloning an existing repository

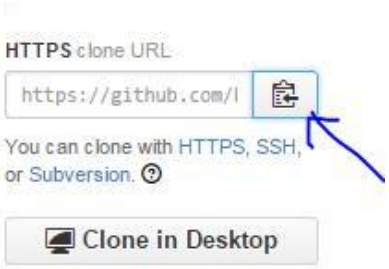
Oftentimes the git repository will already exist on a remote server. The common scenario is you join a project and want to modify existing documents or upload new ones. In cases like this, you (1) create a local copy or clone of the repository, (2) add/modify existing documents. (3) push your changes back to the central or upstream repository.

In this exercise, you will clone a repository, make a change to an existing file and upload or push your changes back to the server.

Open one of the git shells (command line tools) you downloaded above. I recommend the git bash shell.

Get the URL to the remote repository you want to clone.

The clone command will create a new directory for the cloned project. cd to the directory where you want to create the new repository. Clone the remote repository with:



$ git clone https://github.com/burrise/gitpractice.git

The command above will create a new subdirectory gitpractice and initialize it with the remote repository. cd (change directory) into the new directory:

$ cd gitpractice

Verify it the clone command created remote references to the remote repository:

$ git remote –v

Modify one of the downloaded files or add a new file to the directory:

$ notepad <filename>

$ echo “new file contents” > newfile.txt

$ git add <filename>

$ git add newfile.txt

$ git commit –m ‘practice changes’

To push your changes to the remote server:

$ git push

Browse github.com to verify new changes were uploaded correctly.

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. You can verify this with the following command:

$ git remote -v

Pro tip! As time goes on others may push their own changes to the central repository. On a regular basis you should update your local copy with changes made on the central repository:

$ git pull origin master

Exercise 3 – Contributing to an existing github project

Reference for this section: http://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project

This exercise will give you practice with contributing to an existing github.com project. There are two options for contributing to an existing github project. First, the owner of the project could give you read/write access (not very likely), or you could use the fork & pull request method. This exercise uses the fork & pull request method.

In this exercise you will fork a repository, commit and sync changes to your forked copy on github and then issuing a pull request to the owner of the original project to have your changes merged with the original.

First, fork the following repository: https://github.com/burrisetest/BestTechReferences.git. You will find it under the user ID: burrise.

(Hint: log on to github.com, find the repository, press “Fork”.)

When you fork a github repository, a copy of the repository appears under your account at github.

Clone the forked repository to a local computer. (See exercise #2 above.) (Note, you will have to change burrisetest to your github user ID.)

$ git clone https://github.com/burrisetest/BestTechReferences.git

$ cd BestTechReferences

Create a topic branch for performing work.

$ git checkout -b EddiesContribution

(Or:

$ git branch EddiesContribution

$ git checkout EddiesContribution

)

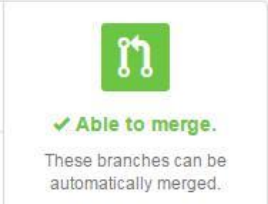
Open the local file References.txt and add a reference. Be sure to include your name so we know who to credit.

Once you are done with all edits and commits, commit your changes to your topic branch.

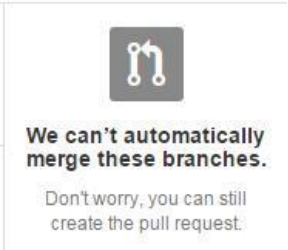
You could do a git push origin EddiesContribution at this point:

$ git push origin EddiesContribution

If the repository you forked hasn't changed since you forked it or changes can be automatically merged, you will be able to issue a pull request that can be merged automatically:



However, if changes were made to the forked repository that conflict with changes you made, you will get a message like the following when you try and issue a pull request:



Either way, it is good practice to pull in any upstream changes before issuing a pull request.

To make it easier for the owner to merge your changes, you should add an upstream remote and pull in any upstream changes there are.

The following command will add an upstream remote:

$ git remote add upstream https://github.com/burrise/BestTechReferences

Now, pull in upstream changes:

Option 1:

$ git fetch upstream

$ git merge upstream/master

(Note, you may want to use the --no-ff flag. It will always generate a merge commit: git merge --no-ff upstream/master.)

Option 2:

$ git pull upstream master

If there are no upstream changes since you forked, there will be nothing to merge.

If there are upstream changes but they don't overlap with the changes you made, the changes will be merged automatically.

If there are upstream changes that can't be merged automatically, you will have to manually merge the changes. If there are merge conflicts, you will get a message something to the effect "Automatic Merge Failed". To see which files need to be manually merged, use the git status command:

$ git status

Look for the section "Unmerged paths".

To manually, resolve the conflict, open the file or files with the merge conflicts and look for lines of the form:

```
<<<<<<< HEAD
<Changes you made>
=======
<Changes on the server that conflict with your changes
>>>>>>> 7bcfe2880234048e64254dbc35da1176ecc6f839
```

Manually resolve the conflict. Be sure to remove the surrounding << .. >>> lines. Once done editing, commit changes:

$ git add <filename>

Staging the file marks it as resolved in Git.

$ git commit –m 'resolved conflicts'

$ git push origin EddiesContribution

The above will fail if your topic branch is behind the current contents of the remote repository.

Now, send the original author of the repository you forked a pull request.

(Hint: go to the repository on github.com, click on pull request, click on create pull request, click on send pull request)

(To accept the pull request, the author will click on "Pull Requests", open the pull request, review it and then click on "Merge Pull Request".)

Exercise 4 – Adding a remote

This exercise walks you through the scenario where you have an existing set of documents you want to place under version control.

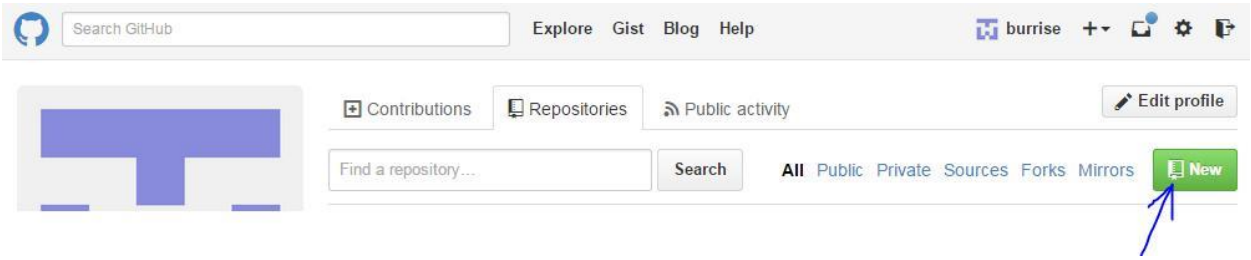
This exercise will give you practice with:

- Creating a repository on github.com
- Committing and syncing (or pushing) changes to github

Local version control is useful, but a more common scenario is to have a centralized repository shared by multiple developers. In this environment developers collaborate by downloading changes from others (pull) and uploading their own changes (push).

In this exercise, you will push a local git repository to a remote server. First, create a local git repository (git init). The repository you created in the first exercise will do just fine. Add files and do some commits.

To create a remote shared repository, go to github.com and create a new repository.
I recommend including a readme file.



Add it as a remote:

$ git remote add origin https://github.com/burrise/gitpractice.git

Issue the remote command to verify it was added successfully:

$ git remote -v

Your local repository is out of sync with the remote that was just created. They have to be in sync before you can push your local changes. The following command will merge the contents of the remote repository (just a readme file) with your existing local repository:

$ git pull origin master

Push your changes with:

$ git push -u origin master

(You will need to enter your github user ID and password.)

The command above pushes to the master branch on the remote server. In a later exercise, you will see how to push to a different branch. Browse the repository on github.com to verify it was sent correctly. Notice, all local commits were preserved when you pushed to the remote server.

Exercise 5 – Branching and merging

This exercise will give you some practice with branching and merging. New feature development and bug fixes are often performed on a branch. Once the work is complete and verified, it is merged back to the main line. This helps ensure there is one branch (usually master or a development branch) that is stable and deployable at all times with the evolution of the product conducted on separate branches.

Let's start with a simple example of branching and merging with a local repository.

Create a new directory and init a git repository:

$ mkdir website

$ cd website

$ git init

Add the file, index.html:

```
<html>
  <body>
    <h1>We are open:</h1>
    <ol>
      <li>Monday</li>
      <li>Wednesday</li>
      <li>Friday</li>
    </ol>
  </body>
</html>
```

$ notepad index.html

$ git add index.html
$ git commit –m 'initial checkin'

This section is about branching, and you actually have a branch at this point. When you do git init, the default branch master is created for you. To see, run the command that lists all branches:

$ git branch

We want to make changes to the index.html file. We will be committing versions that aren't deployable (viewable in a browser), so we want to work on a new branch. In this example, by convention everything in the master branch is stable and deployable. To create a branch for the new feature we will be adding:

$ git branch contactinfo

List the branches again:

$ git branch

You have a new branch but are still on the master branch. To switch to the contactinfo branch:

$ git checkout contactinfo

Now, edit the file to add contact info:

$ notepad index.html

```
<html>
  <body>
    <h1>We are open:</h1>
    <ol>
       <li>Monday</li>
       <li>Wednesday</li>
       <li>Friday</li>
    </ol>
    <p>Phone: 555-1212</p>
    <p>email: ???</p>

  </body>
</html>
```

$ git add index.html
$ git commit –m ‘add phone contact’

It is important to recognize that the files in your directory can change based on the branch you are on. So far, we have made one change since branching off master. If you switch back to master, the files in your directory will appear as they were before you created a branch. Try it:

$ git checkout master

$ cat index.html

Notice, the contents of the file index.html is what it was before you created the feature branch.

Now, switch back to the contactinfo branch and finish the feature:

$ git checkout contactinfo
$ notepad index.html

Notice, the contents of index.html changed after switching back to the contactinfo branch.

Add the email address and commit the change:

```
<html>
  <body>
    <h1>We are open:</h1>
    <ol>
      <li>Monday</li>
      <li>Wednesday</li>
      <li>Friday</li>
    </ol>
    <p>Phone: 555-1212</p>
    <p>email: customerservice@acme.com</p>

  </body>
</html>
```

```
$ git add index.html
$ git commit –m ‘add phone contact’
```

We are done with the feature we wanted to add on this branch and are ready to merge our changes back into master. Merging at this point would be trivial because master hasn’t
changed. A more common scenario is for master to change after a branch and the merge would involve merging changes on two branches.

For a more realistic scenario, let’s switch back to the master branch and make some changes:

```
$ git checkout master
```

```
$ notepad index.html
```

Add a title and commit the changes:

```
<html>
<head>
  <title>Achme Inc.</title>
</head>
  <body>
    <h1>We are open for business:</h1>
    <ol>
      <li>Monday</li>
      <li>Wednesday</li>
      <li>Friday</li>
    </ol>
  </body>
</html>
```

```
$ git add index.html
$ git commit –m ‘add title’
```

The master branch has a copy of index.html with a title but no contact information and the contactinfo branch has the contact information but no title (the title was added after the branch was created).

Let's merge the two.

You always merge into a branch. In other words, you start from the branch you want to merge into. Check to make sure you are on the master branch:

$ git branch

To merge the contactinfo branch:

$ git merge contactinfo

```
Auto-merging index.html
Merge made by the 'recursive' strategy.
 index.html | 3 +++
 1 file changed, 3 insertions(+)
```

List the contents of index.html to verify the merge was performed correctly:

$ cat index.html

Notice, we get a three-way merge and not a fast-forward merge. If there had been no changes on master, we would have had a fast-forward merge: the master pointer would have been updated to point to the latest commit on the contactinfo branch.

Use the following command to see a visual representation of the commits:

$ git log --oneline --decorate --graph --all

```
*   5e1e15a (HEAD, master) Merge branch 'contactinfo'
|\
| * 811ab23 (contactinfo) add email contact
| * f5c475c add phone contact
* | 6ae7233 add title
|/
* 4be8655 initial checkin
```

The above shows two commits on master (initial checkin and add title), two on contactinfo and one for the merge of master and contactinfo.

We are done with the contactinfo branch so it can be deleted:

$ git branch -d contactinfo

Exercise 5 – Branching and merging with conflicts

This exercise is similar to the previous except all changes to master will be done through branching and merging and there will be a merge conflict that must be resolved manually.

Create a new directory and init a git repository:

```
$ mkdir website
$ cd website
$ git init
```

Add and commit the file, index.html:

```
<html>
  <body>
    <p>This web page has moved to: http://achme.com/retail/
  </p> </body>
</html>
```

```
$ notepad index.html
```

```
$ git add index.html
$ git commit –m ‘initial checkin’
```

Imaging you have been asked to add a new feature. The client wants the address on the home page to be a hyperlink.

Create a branch for this new feature. You can create and switch to a branch with one command by using the –b option with checkout:

```
$ git checkout -b hyperlink
```

Now, edit the file to make the link a hyperlink:

```
$ notepad index.html
```

```
<html>
  <body>
    <p>This web page has moved to: <a
href=”http://achme.com/retail/”>http://achme.com/retail/ </a> </p>
  </body>
</html>
```

```
$ git add index.html
$ git commit –m ‘make hyperlink’
```

Now, imagine a severity 1 defect comes in. The URL should be https not http (secure not insecure).

To make the fix, we switch back to the master branch, create a branch for the fix and merge it with master:

(Note, before you can switch back to master you need to commit or stash current changes. We committed above.)

$ git checkout master
$ git checkout -b hotfix123
$ notepad index.html

Change http to https:

```
<html>
  <body>
    <p>This web page has moved to: https://achme.com/retail/ </p>
  </body>
</html>
```

$ git add index.html
$ git commit –m ‘Use secure protocol https’

We are done making the changes for the hot fix, so we can merge it back in with master:

$ git checkout master

$ git merge hotfix123

```
Updating 7aecabd..3a982aa
Fast-forward
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Notice because there were no changes on master, we get a fast-forward merge.

You can view the commit tree with:

$ git log --oneline --decorate --graph --all

```
* 3a982aa (HEAD, master, hotfix123) use secure protocol https
| * c1f80ba (hyperlink) make hyperlink
|/
* 7aecabd initial checkin
```

Notice, by default with a fast-forward merge you don't get a commit for the merge. Many developers find it useful to have a commit for all merges. You can force a commit with a merge by using:

$ git merge --no-ff <branch>

The --no-ff option is useful because it documents the merge. Someday you might want to know that a merge was performed at this point in the project.

We are done with the hotfix branch, so we can delete it:

$ git branch –d hotfix123

At this point we can switch back to the hyperlink branch and finish our changes. To simply the example, we will assume the changes are complete and ready to be merged back in with master. We are on master, so we only need to merge:

$ git merge hyperlink

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

burrise@KC-SCE-3XVQFS1 ~/website (master|MERGING)
```

We get a merge conflict. Changes can't be merged automatically. What was changed on master after we created the hyperlink branch overlaps with changes made on the hyperlink branch.

git status indicates which files have conflicts and must be manually edited and tells us how (add / commit):

$ git status

```
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Files that need changing will we annotated:

$ cat index.html

```
<html>
  <body>
<<<<<<< HEAD
    <p>This web page has moved to: https://achme.com/retail/ </p>
=======
```

```
   <p>This web page has moved to: <a
href="http://achme.com/retail/">http://ac
hme.com/retail/ </a> </p> >>>>>>> hyperlink

  </body>
</html>
```

The standard diff shows the file on master (HEAD) uses https and the file I'm trying to merge from has an anchor tag around the address. In this situation the right thing to do is to combine the two and remove the diff tags:

$ notepad index.html

```
<html>
  <body>
   <p>This web page has moved to: <a
href="https://achme.com/retail/">https://achme.com/retail/ </a> </p>
  </body>
</html>
```

To manually resolve a merge conflict, you make the changes, add and commit the results:

$ git add index.html

$ git commit –m 'make link clickable and https'

Run git log to see the results of the merge:

$ git log --oneline --decorate --graph --all

We are done with the hyperlink branch, so we can delete it:

$ git branch –d hyperlink

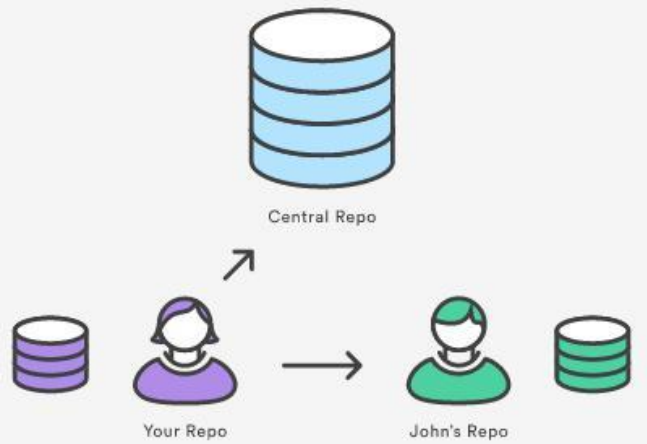
Note, you don't have to delete a branch after merging. It makes sense to delete the branch here because it was create to add a feature and the feature has been added (something called a topic branch). You may have a development branch off master that aggregates all new development. At regular intervals when development work is stable, it can be merged back into master. In this case, development runs parallel to master and is occasionally merged back in.

Exercise 6 – Branching and remotes

References for this section:

- https://www.atlassian.com/git/tutorials/syncing
- http://git-scm.com/book/en/v2/Git-Branching-Remote-Branches

With git you collaborate with others through remote connections:



The git remote command lets you create, view, and delete connections to other repositories. For example, the following command will list the remote connections you have to other repositories:

$ git remote

Use the –v option for more details:

$ git remote –v

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository.

You may want to add others, such as a reference to an upstream remote:

$ git remote add upstream https://github.com/burrise/BestTechReferences

(See example above where you forked a repository and added an upstream remote to keep your forked repository up-to-date.)

When you clone a repository you get local copies of the branches on the remote repository. These are called **remote branches**. (The term remote branch is a little misleading because the copies are local, they just represent the state of the remote repository the last time you connected—usually via the git fetch command.)

To see remote branches, enter:

$ git branch –r

```
  origin/HEAD -> origin/master
  origin/feature123
  origin/master
```

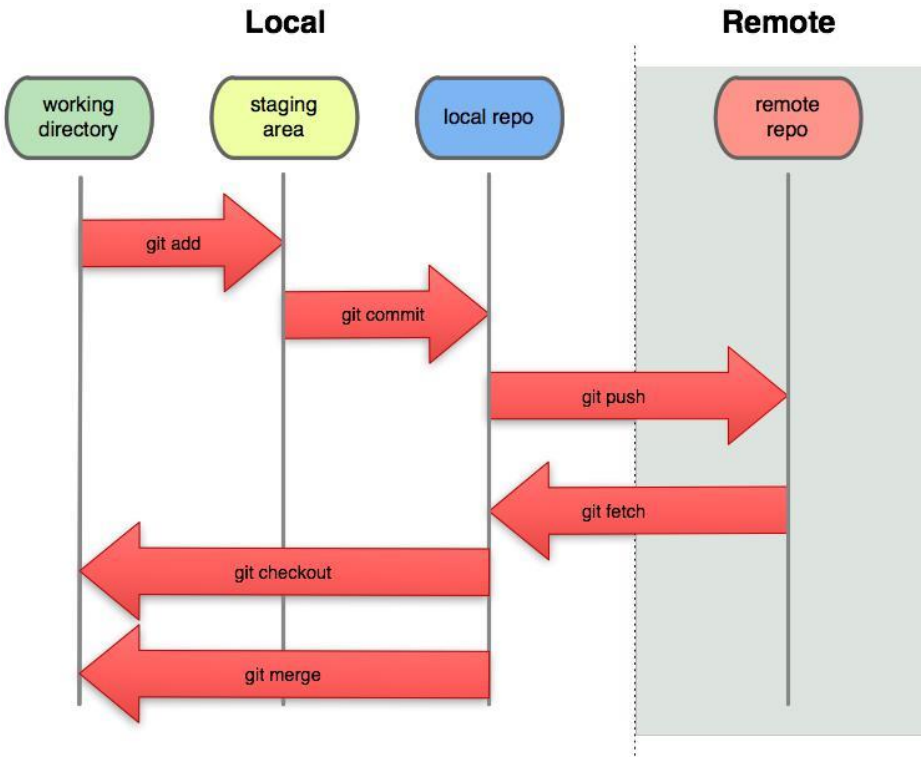
To inspect the contents of a remote branch:

$ git checkout origin/feature123
$ ls

Note, you are in a detached HEAD state when browsing remote branches. Don't make any changes. To return to the working area:

$ git checkout master

Here is a nice visual to show the three/four different staging areas:



git clone copies a remote repository to a local repository and creates a working directory for new changes. Local commits update the local repository. git push sends a copy of the local repo to the remote repo. git fetch (to be discussed shortly) copies changes on the remote repo to the local one.

Let's do an example. In this example, you will create a new branch and push it back to the central repository. Another developer will download (pull) the branch, make an update and send it back to the server. You will download the updated branch and incorporate it with your work,

First, create a new repository on github. It will have one branch called master. Clone it:

$ git clone https://github.com/burrise/teamrepo.git

$ cd teamrepo

A new repository has only one branch:

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
```

You will be simulating a second developer later in this exercise. Create another clone to simulate a second developer:

$ cd ..

$ git clone https://github.com/burrise/teamrepo.git teamrepocopy

Switch back to the first clone:

$ cd teamrepo

Create a new branch and check it out with one command:

$ git checkout –b fix123

Add a new file on the branch:

$notepad file.c

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren't automatically synchronized to the remotes you write to – you have to explicitly push the branches you want to share. In other words, the following doesn't upload the new branch just created:

$ git checkout master
$ git push

To push the branch just created:

$ git push origin fix123

Switch to the second clone (teamrepocopy) which is being used to simulate a second developer:

$ cd ../teamrepocopy

Verify there is only one remote branch:

$ git branch –r

Fetch any new commits and/or branches on the server:

$ git fetch origin

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/burrise/teamrepo
 * [new branch]      fix123     -> origin/fix123
```

You should see a message that a new branch was found on the server and copied down. To verify:

$ git branch –r

When you do a fetch that brings down new remote branches, you don't automatically have local, editable copies of them. In other words, in this case, you don't have a new fix123 branch – you only have an origin/fix123 pointer that you can't modify.

To merge this work into your current working branch, you can run git merge origin/fix123. If you want your own fix123 branch that you can work on, you can base it off your remote branch:

$ git checkout -b fix123 origin/fix123

Checking out a local branch from a remote branch automatically creates what is called a "tracking branch" (or sometimes an "upstream branch"). Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type git pull, Git automatically knows which server to fetch from and branch to merge into.

Make a change:

$ notepad file.c

$ git add .

(git add . will add all changed files to the staging area)

$ git commit -m 'second developer added a fourth line'

Push the changes to the server:

$ git push

Switch back to the first clone:

$ cd ../teamrepo

Fetch the changes:

$ git fetch

You can see that fetch just downloads the changes to local repo. It doesn't merge the changes into the working directory:

$ git log --oneline origin/fix123

```
2cec3fd second developer added a fourth
line 750e024 initial checkin of file.c
f39da94 Initial commit
```

$ git log –oneline

```
750e024 initial checkin of file.c
f39da94 Initial commit
```

If you want to see the details of the differences:

$ git diff fix123 origin/fix123

To merge:

$ git merge origin/fix123

Here is a useful command when working with remote repositories. The following command shows the details of the remote repo called origin:

$ git remote show origin

```
* remote origin
  Fetch URL: https://github.com/burrise/teamrepo.git
  Push  URL: https://github.com/burrise/teamrepo.git
  HEAD branch: master
  Remote branches:
    fix123 tracked
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local refs configured for 'git push':
    fix123 pushes to fix123 (up to date)
    master pushes to master (up to date)
```

Notice the command above will tell you if a branch on the local repo is out of date with its corresponding branch on the remote server.