```
                    **********************************
                    **********************************
                    **                              **
                    **         Git Toolkit          **
                    **                              **
                    **********************************
                    **********************************
```

++
++++ Git Overview ++
++

Other version control systems like SVN (Subversion) use a CENTRALIZED system. This means that there is a central master repository stored on one computer/server. To begin working under a CENTRALIZED system you follow these general steps:

1. Checkout a copy of the code from the master repository.
2. Make changes to the code.
3. Commit your changes back to the master repository.

Git uses a DISTRIBUTED system. Each team member maintains their own repository!

Advantages:
- Faster because there is no need to communicate with a central server constantly.
- No network access required. Commit changes to local repository and push them to the server later.
- No single failure point. All users have all or nearly all the files and changes locally.

http://www.git-tower.com/learn/content/01-ebook/01-command-line/07-appendix/03-from-subversion-to-git/centralized-vs-distributed.png

The general steps of working with a git repository is as follows:

1. Clone the git repository. This copies and exact copy of the repository to your local machine.
2. Make changes and commit them, as you work, to your local repository.
3. When you want to share the changes you have made you can do so by pushing your changes to the "master" repository where other team members are expecting them to be found.

++
++++ Git Help ++
++

If you every encounter difficulty you can issue the following commands to get help.

```
$ git help

$ git help <command>
```

You can also visit http://git-scm.com/doc for additional help.

++
++++ Setup Git +++
++

Before you get started it is important to configure git to your liking. Git uses a series of configuration files to determine some important configuration options.

At the SYSTEM(system) level git looks at the /etc/gitconfig file. This is rarely used.
At the GLOBAL(user) level git looks at the ~/.gitconfig file.
At the LOCAL(project) level git looks at the .git/config file of the repository you are currently in.

The settings at the LOCAL level supersede the settings at the GLOBAL and SYSTEM level.
The settings at the GLOBAL level supersede the settings at the SYSTEM level.

The nice thing about these configuration files is that they are simple plain text files. This means that they are easily read and can be copied over to another system.

There are many configuration options you can set. Listed here are some of the most important.

```
#Set your name.
$ git config --global user.name "John Doe"

#Set your email.
$ git config --global user.email johndoe@example.com

#Set your preferred default editor.
$ git config --global core.editor emacs

#Color your output so you can quickly tell what is going on.
$ git config --global color.ui true

#List your configuration options.
$ git config --list
```

Initializing a Repository

To start using git you must first initialize a repository. To do this you must first navigate to the directory you want to put under version control. This directory can already have files in it, or it can be empty.

```
$ cd path/to/your/proj
```

Once inside the proper directory you now can initialize the repository.

```
$ git init
```

Now anything you do within this directory can be tracked by git. The magic that makes git work is stored in the .git directory that was created when you ran the previous command. The files contained within this directory are not intended to be edited by the user. You can see that it has been created.

```
$ ls -a
```

Edit, Add, Commit

Once you have initialized the repository you can add, remove, and edit files, as you normally would. Once you are ready to commit your changes, essentially taking a snapshot of your current state of your files, you must add/stage the files so git knows you want the changes you made to be committed to the repository.

```
    #Running this command from the top directory of your project will add all your changes.
    $ git add .

    #You can also selectively add files if you want to commit some, but not others.
    $ git add <list of files>
```

Once you have added the files you want to commit you can see the status of all your files.

```
    $ git status

    #Useful options
        -s = condensed output

    #Common letter codes found in output
        A = added
        D = deleted
        R = renamed
        C = copied
        M = modified
```

If you see any files listed which you don't want committed, follow the instructions git gives you at the top of the 'git status' output to unstage them.

Now that all the changes you have made have been staged you can commit your changes to the local repository. This saves the state of all the files you have added in the previous step so that you can go back later if you make a mistake.

```
git commit -m "be sure to include a good message outlining the changes you made."
```

++
++++ Commit Message Tips ++
++

A good commit message has the following attributes:

1. Describes the changes you made.
2. Short
3. Uses present tense
    - Good: fixes a bug
    - Bad: I fixed a bug

++
++++ Git Log Command ++
++

The 'git log' command allows you to see a log of all the commits that have been made to the repository.

```
    #Useful options
        -n <number> = only show the <number> most recent commits.
        -since=<yyyy-mm-dd> = shows only commits since <yyyy-mm-dd>
        -author=<author>
        -grep=<word|pattern> = show logs with certain <word|pattern>
        -i = ignore case
```

++
++++ Git Architecture ++
++

Git uses a three tier system described as follows.

TIER 1:
    The first tier is the WORKING COPY. This is the directory where you work on your project by adding files, editing files, and removing files.

TIER 2:
    The second tier is the STAGING INDEX. This is where changes which you have added using 'git add' are stored.

TIER 3:
    The third tier is the REPOSITORY. This is where the files in the STAGING INDEX go when you run the 'git commit' command. When a commit occurs a SHA-1/checksum value is generated which is unique to that commit. This SHA-1/checksum value is useful when needing to referring to a specific commit. It is listed at the top of each log entry. Additionally each commit stores the value of the commit that was immediately previous to it (its parent). In this way a linked-list of commits is generated.

++
++++ The HEAD Pointer ++
++

The HEAD pointer is an integral part of the git architecture. It is not necessarily essential for a git user to be aware of, but understanding the HEAD pointer is important when trying to understand how many of git's features work.

Essentially, HEAD is a pointer to the most recent commit in the currently checked-out repository. It allows git to easily determine what the parent should be of a new commit.

Understanding of HEAD is particularly important when it comes to branching. The branch generated by default when you initialize a project under git is the "master" branch. If you generate another branch in the repository then the HEAD will point to the the most recent commit of that new branch.

```
$ cat .git/HEAD
ref: refs/heads/master

$cat ./refs/heads/master
8ddf5a4e167582d099f166d9cb4f9a4c6b976f11

$git log -n 1
commit 8ddf5a4e167582d099f166d9cb4f9a4c6b976f11
Author: Marty McFly <mmcfly@timetravel.future.com>
Date:   Sat Apr 25 17:14:12 2072 -0400
This is a generic message.
```

++
++++ Using Diff to View Changes ++
++

You will often want to check to see how the modifications you have made to a file in the working directory compare with the most recent copy of the file in the repository/staging area. You can do this using the following commands.

```
#Run diff on all modified files in you working directory.
$ git diff

#Run diff on a specific file
```

```
    $ git diff <filename>

    #Useful options
        --staged = shows differences between files in the staging area and the repository only
```

++
++++ Removing Files ++
++

Sometimes a file in your project is no longer needed and you would like to remove it so that it is no longer a part of your most up-to-date version of your project.

```
    #Deletes the local copy and adds a change to the staging area indicating that the file will
    be removed.
    $ git rm <filename>
```

It is sometimes the case that you will have made changes to the file (not committed to the repository), but you still want to delete the file. If you use the traditional 'git rm' command you will be prompted with a message stating "error: '<filename>' has local modifications." In this case you have two options to remove the file.

```
    #Removes the file from your working copy and adds a change to the staging index indicating
    the file is to be removed. You will NOT be able to recover any modifications to the file
    which you have made since the file was last committed to the repository. To see what you
    will lose and not be able to recover use the 'git diff' command.
    #This will also add the remove to the staging area automatically.
    $ git rm -f <filename>

    #Adds a change to the staging index indicating the file is to be removed, but keeps the
    file in your working directory.
    $ git rm --cached <filename>
```

The nice thing about git is all changes COMMITTED to the repository are recoverable even if you remove a file as described above. See the section on Undoing Changes for detailed instructions on how to do this.

```
    #**WARNING** this will overwrite the <filename> in your working directory.
    $ git checkout <SHA-1/checksum which has your file> -- <filename>
```

++
++++ Moving Files ++
++

Moving files using git is very similar to removing files using git.

```
    #Git will not allow the move if file2 already exists.
    #This will also add it to the staging area automatically.
    $ git mv <file1> <file2>
```

++
++++ Undoing Changes ++
++

If you make a mistake in editing a file you can retrieve the files back from the repository.

```
    #This command is used for more two purpose. It is used to checkout a branch, but it can
    also be used to recover files.
```

```
    #It works by going into the repository and retrieving any file or branch you want.
    #The '--' indicates to git that you are wanting to checkout a files. This way if you have a
    branch called <filename> and a file called <filename> git will know you want the files
    instead of the branch.
    #**WARNING** this will overwrite the <filename> in your working directory.
    $ git checkout -- <filename> or <directoryname>

    #Retrieve a file from a specific commit.
    #**WARNING** this will overwrite the <filename> in your working directory.
    $ git checkout <SHA-1/checksum which has the correct version of your file> -- <filename>
```

++
++++ Ammending Commits ++
++

Sometimes you will make a commit which includes changes you did not want to include. You can undo this change by amending the most recent commit. To do this, simply change the files in your working directory until they are to your liking. Add the changes to the staging index as you normally would.

```
    $ git add <filename>

    $ git rm <filename>
```

You can then amend/overwrite your previous commit.

```
    $ git commit --amend -m "your message here."
```

You can also just change your most recent commit message.

```
    $ git commit -amend -m "just updating my most recent commit message."
```

It is important to note that due to the linked-list method of storing commits and git's data integrity measures, you can only amend your most recent commit.

++
++++ Creating a Remote Repository ++
++

Make sure everyone you want to be able to work on the repository has an account on the system.

```
    $ sudo adduser myUser
```

Make sure everyone you want to be able to work on the repository is in the same group.

```
    #The specific commands to accomplish this task may differ on your system.
    $ sudo addgroup myGroup
    $ useradd -G myGroup myUser
```

Create a directory to store your project and change the permissions to allow members of anyGroup to access the files.

```
    $ mkdir myProj.git
    $ chgrp myGroup myProj.git
    #Sets the appropriate permissions to allow group members to work with the repository.
    $ chmod g+rws myProj.git
```

Initialize the git repository using the "--bare" option. Bare repositories contain only the

.git folder and no working copies of your source files (http://www.saintsjd.com/2011/01/what-is-a-bare-git-repository/). Also use the "--shared" option to ensure members of myGroup are allowed to use the repository.

```
$ cd myProj.git
$ git init --bare --shared
```

From the local host which houses the git repository that contains the actual project run the following.

```
$ git remote add origin username@some.where.edu:/path/to/git/repo
$ git push origin master
```

Now you can make changes to the working directory, commit those changes to the local repository, and then push those changes to the remote repository using the following.

```
$ git push
```

You will also be able to get the latest changes that have been added to the remote repository.

```
$ git pull
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++ Useful Commands ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#Shortcut which will add 'git all' files and then perform a 'git commit'.
#Files that are not tracked and files that are to be deleted will not be included.
#Be thoughtful when using this command.
$ git commit -a

#Get see detailed information for a specific commit.
$ git show <SHA-1/checksum>

#BitBucket push
$ git push -u origin master
```