**Terminology**

**Version Control System / Source Code Manager**

A **version control system** (abbreviated as **VCS**) is a tool that manages different versions of source code. A **source code manager** (abbreviated as **SCM**) is another name for a version control system.

Git is an SCM (and therefore a VCS!).

**Commit**

Git thinks of its data like a set of snapshots of a mini filesystem. Every time you **commit** (save the state of your project in Git), it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

**Repository / repo**

A **repository** is a directory which contains your project work, as well as a few files (hidden by default on Mac OS X) which are used to communicate with Git. Repositories can exist either locally on your computer or as a remote copy on another computer. A repository is made up of commits.

**Working Directory**

The **Working Directory** is the files that you see in your computer's file system. When you open your project files up on a code editor, you're working with files in the Working Directory.

This is in contrast to the files that have been saved (in commits!) in the repository.

When working with Git, the Working Directory is also different from the command line's concept of the *current working directory* which is the directory that your shell is "looking at" right now.

**Checkout**

A **checkout** is when content in the repository has been copied to the Working Directory.

**Staging Area / Staging Index / Index**

A file in the Git directory that stores information about what will go into your next commit. You can think of the **staging area** as a prep table where Git will take the next commit. Files on the Staging Index are poised to be added to the repository.

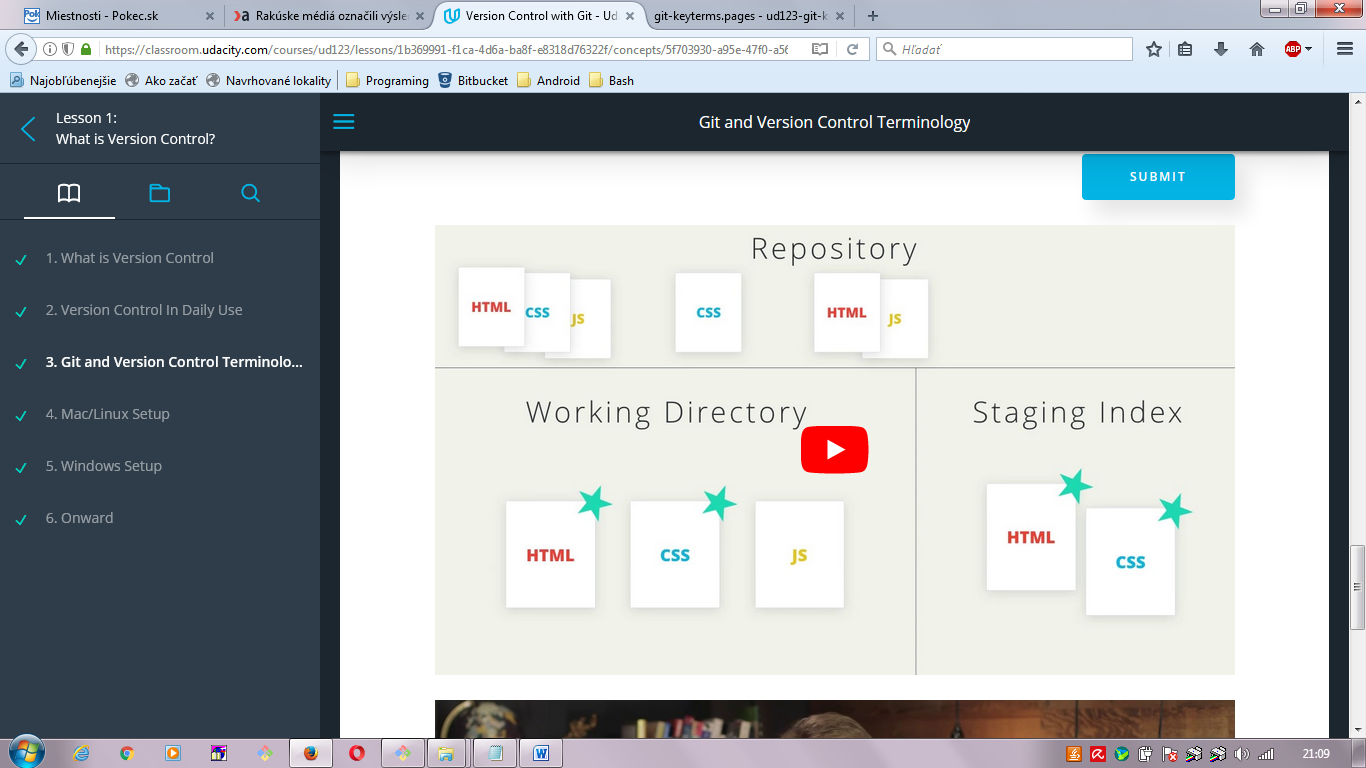
**SHA**

A **SHA** is basically an ID number for each commit. It is a 40-character string composed of characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. "SHA" is shorthand for "Secure Hash Algorithm".

## Branch

A **branch** is when a new line of development is created that diverges from the main line of development. This alternative line of development can continue without altering the main line.

Going back to the example of save point in a game, you can think of a branch as where you make a save point in your game and then decide to try out a risky move in the game. If the risky move doesn't pan out, then you can just go back to the save point. The key thing that makes branches incredibly powerful is that you can make save points on one branch, and then switch to a different branch and make save points there, too.



## First Time Git Configuration

Before you can start using Git, you need to configure it. Run each of the following lines on the command line to make sure everything is set up.

# sets up Git with your name

git config --global user.name "<Your-Full-Name>"

# sets up Git with your email

git config --global user.email "<your-email-address>"

# makes sure that Git output is colored

git config --global color.ui auto

# displays the original state in a conflict

git config --global merge.conflictstyle diff3

git config --list

**Git & Code Editor**

The last step of configuration is to get Git working with your code editor.

git config --global core.editor "'C:/Program Files/Sublime Text 2/sublime\_text.exe' -n -w"

## git init

command to create a new, empty repository in the current directory.

Running this command creates a hidden .git directory. This .git directory is the brain/storage center for the repository. It holds all of the configuration files and directories and is where all of the commits are stored.

**git clone**

command is used to create an identical copy of an existing repository.

$ git clone <path-to-repository-to-clone> <new name>

This command:

* takes the path to an existing repository
* by default will create a directory with the same name as the repository that's being cloned
* can be given a second argument that will be used as the name of the directory
* will create the new repository inside of the current working directory

**git status**

command will display the current status of the repository.

This command will:

* tell us about new files that have been created in the Working Directory that Git hasn't started tracking, yet
* files that Git *is* tracking that have been modified
* a whole bunch of other things that we'll be learning about throughout the rest of the course ;-)

**git log**

command is used to display all of the commits of a repository.

By *default*, this command displays:

* the SHA
* the author
* the date
* and the message

...of every commit in the repository. The important keys for Less are:

* to scroll down by a line, use j or ↓
* to scroll up by a line, use k or ↑
* to scroll down by a page, use the spacebar or the Page Down button
* to scroll up by a page, use b or the Page Up button
* to quit, use q

**git log --oneline**

displays information:

* lists one commit per line
* shows the first 7 characters of the commit's SHA
* shows the commit's message

**git log --stat**

displays information:

* displays the file(s) that have been modified
* displays the number of lines that have been added/removed
* displays a summary line with the total number of modified files and lines that have been added/removed

**git log –p**

displays information this command adds the following to the default output:

* displays the files that have been modified
* displays the location of the lines that have been added/removed
* displays the actual changes that have been made

## git log --decorate

will show us some details that are hidden from the default view.

* ***tag: v1.0***? That's the tag! Remember that tags are associated with a specific commit. This is why the tag is on the same line as the commit's SHA.
* ***HEAD -> master***? That's information about a *branch*!

**NOTE:**

You already know how to "log" information with:

* git log
* git log --oneline
* git log --stat
* git log -p

But did you know, you can supply the SHA of a commit as the final argument for all of these commands? For example:

**$ git log -p fdf5493**

## git log --oneline --decorate --graph --all

we can see all branches at once in the git log output.

We'll use the new --graph and --all flags.

**git show**

displays only the most recent commit. Typically, a SHA is provided as a final argument:

$ git show fdf5493

The git show command will show only one commit. The output of the git show command is exactly the same as the git log -p command.

However, git show can be combined with most of the other flags we've looked at:

* --stat - to show the how many files were changed and the number of lines that were added/removed
* -p or --patch - this the default, but if --stat is used, the patch won't display, so pass -p to add it again
* -w - to ignore changes to whitespace

**git add**

is used to move files from the Working Directory to the Staging Index.

$ git add <file1> <file2> … <fileN>

This command:

* takes a space-separated list of file names
* alternatively, the period . can be used in place of a list of files to tell Git to add the current directory (and all nested files)

**git commit**

takes files from the Staging Index and saves them in the repository.

This command:

* will open the code editor that is specified in your configuration

git config --global core.editor <your-editor's-config-went-here>

Inside the code editor:

* a commit message must be supplied
* lines that start with a # are comments and will not be recorded
* save the file after adding a commit message
* close the editor to make the commit

## Bypass The Editor With The -m Flag

$ git commit -m "Initial commit"

In the example above, the text "Initial commit" is used as the commit message. Be aware that you can't provide a description for the commit, only the message part.

**git commit --amend**

you can alter the *most-recent* commit.

If you Working Directory is clean (meaning there aren't any uncommitted changes in the repository), then running git commit --amend will let you:

* 1. provide a new commit message
  2. include files (or changes to files) you might've forgotten to include

Instead, you can amend the last commit (the one that updated the color of all of the other links) to include this forgotten one. To do get the forgotten link included, just:

* edit the file(s)
* save the file(s)
* stage the file(s)
* and run git commit --amend

## Good Commit Messages

**Do**

* do keep the message short (less than 60-ish characters)
* do explain what the commit does (not how or why!)

**Do not**

* do not explain why the changes are made (more on this below)
* do not explain how the changes are made (that's what git log -p is for!)
* do not use the word "and"
  + if you have to use "and", your commit message is probably doing too many changes - break the changes into separate commits
  + e.g. "make the background color pink and increase the size of the sidebar"

**git diff**

is used to see changes that have been made but haven't been committed, yet. This command displays:

* the files that have been modified
* the location of the lines that have been added/removed
* the actual changes that have been made

## Git Ignore

To recap, the .gitignore file –list of files is used to tell Git about the files that Git should not track. This file should be placed in the same directory that the .git directory is in.

## git tag

will display all tags that are in the repository.

## git tag tag\_Name SHA (whitout SHA is last commit)

Make an unsigned, annotated tag object

## git tag -a (or --annotate) tag\_Name SHA (whitout SHA is last commit)

Make an unsigned, annotated tag object

## git tag -d (or --delete) tag\_Name

Delete existing tags with the given names.

**git branch**

command is used to manage branches in Git:

# to list all branches

$ git branch

# to create a new "footer-fix" branch at commit SHA

$ git branch footer-fix SHA

# to delete the "footer-fix" branch

$ git branch –d(or –-delete) footer-fix

This command is used to:

* list out local branches
* create new branches
* remove branches : -
* 1. One thing to note is that you can't delete a branch that you're currently on.
* 2. Git won't let you delete a branch if it has commits on it that aren't on any other branch - To force deletion, you need to use a capital D flag - ***git branch –DbranchName***

**git checkout branchName**

switch between branches. Each time we make a commit, it will be added to the current branch. So even though we created the new branche, no new commits will be added to it since we haven't *switched to it*, yet.

It's important to understand how this command works. Running this command will:

* remove all files and directories from the Working Directory that Git is tracking
  + (files that Git tracks are stored in the repository, so nothing is lost)
* go into the repository and pull out all of the files and directories of the commit that the branch points to.

**git checkout** **–b newBranch fromBranch**

can actually create a new branch, too? If you provide the -b flag, you can create a branch *and* switch to it all in one command.

$ git checkout -b footer master

**git merge**

is used to combine branches in Git. When a merge happens, Git will:

* look at the branches that it's going to merge
* look back along the branch's history to find a single commit that both branches have in their commit history
* combine the lines of code that were changed on the separate branches together
* **makes a commit** to record the merge

There are two types of merges:

* Fast-forward merge – the branch being merged in must be *ahead* of the checked out branch. The checked out branch's pointer will just be moved forward to point to the same commit as the other branch.
* the regular type of merge
  + two divergent branches are combined
  + a merge commit is created

As of right now, we do not know how to undo changes, but if you make a merge on the wrong branch, use this command to undo the merge:

**git reset --hard HEAD^**

**Merge Conflict Recap**

A merge conflict happens when the same line or lines have been changed on different branches that are being merged. Git will pause mid-merge telling you that there is a conflict and will tell you in what file or files the conflict occurred. To resolve the conflict in a file:

* locate and remove all lines with merge conflict indicators
* determine what to keep
* save the file(s)
* stage the file(s)
* make a commit

Be careful that a file might have merge conflicts in multiple parts of the file, so make sure you check the entire file for merge conflict indicators - a quick search for <<< should help you locate all of them

**git revert** **<SHA of commit >**

command is used to reverse a previously made commit-**revert** a specific commit, Git takes the changes that were made in commit and does the exact opposite of them. If a character is added in commit A, if Git reverts commit A, then Git will make a new commit where that character is deleted. It also works the other way where if a character/line is removed, then reverting that commit will add that content back!

This command:

* will undo the changes that were made by the provided commit
* creates a new commit to record the change

**Relative Commit References**

when you'll want to reference a commit relative to another commit.

^ indicates the parent commit

~ indicates the *first* parent commit

* HEAD^^^
* HEAD~3

the great-grandparent commit – the following indicate the great-grandparent commit of the current commit

The main difference between the ^ and the ~ is when a commit is created from a merge. A merge commit has two parents. With a merge commit, the ^ reference is used to indicate the first parent of the commit while ^2 indicates the second parent. The first parent is the branch you were on when you ran git merge while the second parent is the branch that was merged in.

**git reset - -mixed, --soft, --hard**

The git reset command is used to reset (erase) commits:

$ git reset <reference-to-commit>

It can be used to:

* move the HEAD and current branch pointer to the referenced commit
* erase commits with the --hard flag
* moves committed changes to the staging index with the --soft flag
* unstages committed changes --mixed flag

You've got to be careful with Git's resetting capabilities. This is one of the few commands that lets you erase commits from the repository. If a commit is no longer in the repository, then its content is gone.

To alleviate the stress a bit, Git does keep track of everything for about 30 days before it completely erases anything. To access this content, you'll need to use the git reflog command. Check out these links for more info:

**Backup Branch**

Remember that using the git reset command will erase commits from the current branch. So if you want to follow along with all the resetting stuff that's coming up, you'll need to create a branch on the current commit that you can use as a backup.

Before I do any resetting, I usually create a backup branch on the most-recent commit so that I can get back to the commits if I make a mistake: **$ git branch backup**