**Notes**

**Day 0:**

* Apache web server to the Linux kernel have been “patch-based” development
* “Diff” is simply short for “difference;” (reveals the difference between one or more files)
* “Patch” - A file that contains differences and can be applied to another repository or working directory.
* Git, created by Linus Torvalds, supports this distributed development model, allowing patches to exist separately from the main repository and facilitating pushing and pulling changes into different development trees.
* Instead of sharing patches via email, GitHub users create pull requests to propose changes.
* Git commands:

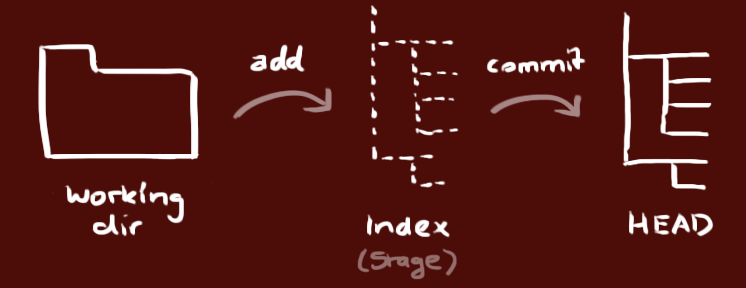
$ git diff

$ git diff > mychanges.patch

$ git apply mychanges.patch

**Getting started with Git:**

* Download git for windows
* “git init” to create a new git repository.
* Local repository consists of three "trees" maintained by git.
  + “**Working Directory**” which holds the actual files.
  + “**Index**” which acts as a staging area.
  + “**HEAD**” which points to the last commit you've made.



* You can propose changes (add it to the **Index**) using
  + git add <filename>
  + git add \*
* Commit the changes
  + git commit –m “commit-message”
  + Note : (Now the file is committed to the HEAD, but not in your remote repository yet.)
* Pushing changes
  + Your changes are now in the HEAD of your local working copy. To send those changes to your remote repository, execute:
    - git push origin master
* If you have not cloned an existing repository and want to connect your repository to a remote server
* git remote add origin <server>
* Branching:
  + Branches are used to develop features isolated from each other. The **master branch** is the "default" branch when you create a repository. Use other branches for development and merge them back to the master branch upon completion.
  + Create a new branch and checkout to it
    - git checkout –b feature\_x
* Switch back to master
  + git checkout master
* Delete a branch
  + git branch –d feature\_x
* To update your local repository to the newest commit, execute:
  + git pull
* to merge another branch into your active branch (e.g. master), use
  + git merge <branch>
* before merging changes, you can also preview them by using
  + git diff <source\_branch> <target\_branch>
* Tagging:
  + it's recommended to create tags for software releases. this is a known concept, which also exists in SVN. You can create a new tag named 1.0.0 by executing -> **git tag 1.0.0 1b2e1d63ff**
  + **1.0.0 1b2e1d63ff** is the commit id of a specific commit that you need to tag
  + You can check the tag by listing all the tags
    - git tag
* Log:
  + you can study repository history using.. -> **git log**
  + To see only the commits of a certain author:
    - git log –author=bob
* To see a very compressed log where each commit is one line:
  + git log –pretty=oneline
* Maybe you want to see an ASCII art tree of all the branches, decorated with the names of tags and branches:
  + git log --graph --oneline --decorate --all
* See only which files have changed:
  + git log --name-status
* For more: -> git log --help
* Replace local changes:
  + In case something goes wrong , and wanted to replace the contents in the last commit in head
    - git checkout -- <filename>
* If you instead want to drop all your local changes and commits, fetch the latest history from the server
  + git fetch origin
  + git reset --hard origin/master



* Git rebase:
  + The second way of combining work between branches is rebasing.
    - git rebase main
* Checking out to commit to point the Head
  + git checkout C3
* Relative Refs:
  + git checkout main^ -> move to the parent commit of the branch
  + When you want to move more levels up the branch tree use,
    - git checkout HEAD~4
* When you wanted to move a branch to a particular commit
  + git branch -f main C0 -> (This command moves the branch main to commit C0)
* Git Reset:
  + *git reset* reverses changes by moving a branch reference backwards in time to an older commit.
    - git reset HEAD~1
  + By default it it git reset --mixed HEAD~1
  + git reset --soft HEAD~1 => commit is removed but the changes are still in staging area.
  + git reset --hard HEAD~1 => commit and also the changes made by the commit is removed locally.
  + In case of mixed the changes are removed from the staging area.
* Git revert:
  + In order to reverse changes and share those reversed changes with others, we need to use git revert.
    - git revert HEAD
* **TIP:** 
  + For git reset we need to give the commit id which we want and all the above commit will be deleted.
  + For git revert we need to give the exact commit id which we need to remove.
* Git cherry-pick:
  + It is used to pick a commit or a series of commit and uses in on a branch
    - git checkout master
    - git cherry-pick <commit 1> <commit 2> ….
* Rebase and Cherry pick nearly do the same task, but when we rebase it will take all the commits from a specified branch to target branch, but cherry pick is used to pick the specified commit and add it to the target branch.
* Git stash:
  + When you are working on a file, and suddenly you get another work to complete, you can save the files that you currently working, in the background using stash.
  + Steps:
    - git add .
    - git stash (Now moved to background)
    - To access the stashed files use -> “git stash pop”
    - To view all the stashed file use -> “git stash list”
    - To access specified stashed file use the stash ID -> “git stash pop 0” Here 0 is the stash id.
* Interactive rebase:
  + When we don’t know the commit hash we use interactive rebase
    - git rebase -i HEAD~4

**Tip** : **git pull** is the shorter version of *git fetch* and *git merge*

**git pull -r** every time to pull all the changes from the branch

**git log --oneline** is used to view only the subject of the commit message

**git status** is used to view is there any files that is to be commited

**To merge:**

* Move to the branch which you need to merge and perform **git merge <oldbranch> <newbranch> (**say git merge develop to qa**)**

**Branching strategy:**

* Types of branches:
  + Master
  + Hotfix
  + Feature
  + Releases
  + Develop

**Rules to write a good commit message:**

* Separate subject from body with a blank line
* Limit the subject line to 50 characters
* Capitalize the subject line
* Do not end the subject line with a period
  + Example:
    - Open the pod bay doors
  + Instead of:
    - ~~Open the pod bay doors.~~
* Use the imperative mood in the subject line
  + Imperative mood just means “spoken or written as if giving a command or instruction”. A few examples:
    - Clean your room
    - Close the door
    - Take out the trash
* A properly formed Git commit subject line should always be able to complete the following sentence:
* If applied, this commit will your subject line here
* For example:
  + If applied, this commit will refactor subsystem X for readability
  + If applied, this commit will update getting started documentation
  + If applied, this commit will remove deprecated methods
  + If applied, this commit will release version 1.0.0
  + If applied, this commit will merge pull request #123 from user/branch
* Notice how this doesn’t work for the other non-imperative forms:
  + If applied, this commit will ~~fixed bug with Y~~
  + If applied, this commit will ~~changing behavior of X~~
  + If applied, this commit will ~~more fixes for broken stuff~~
  + If applied, this commit will ~~sweet new API methods~~
* Wrap the body at 72 characters
* Use the body to explain what and why vs. how

**Day 1:**

**Chapter 1 - Clean Code:**

* Code will be there - Code is basically the requirements that we need, and it can never be replaced and destroyed. So code will always be there.
* Bad code - A bad code can also lead a company to be closed as the bad code will lead to unmaintainable code which we can never use.
* Total cost of owning a mess - A messy code can slow down the productivity significantly.
* Grant redesign in the sky - A messy code will be demanded a new system design or redesign which may cost us years to complete.
* Writing messy code to complete the task is like a doctor who skips the hand wash before operation, it would lead to high risks.
* Bjarne Strostrup says that the code should be elegent and efficient.
* Grady Booch says that the code should be simple and direct.
* Big Dave says that a good code must be easy to read and change.

**Chapter 2 - Meaningful Names** by Tim Ottinger:

* Use intention revealing names.
* Avoid disinformation.
  + Like be careful while using lowercase ‘l’ and uppercase ‘O’.
* Make meaningful distinctions
  + Like adding number series and noise words are not enough, try to make the variable that coveys meaning.
* Use pronounceable name.
  + Compare

class DtaRcrd102 {

private Date genymdhms;

private Date modymdhms;

private final String pszqint = "102";

/\* ... \*/

};

to

class Customer {

private Date generationTimestamp;

private Date modificationTimestamp;;

private final String recordId = "102";

/\* ... \*/

};

* Use searchable names.
  + Compare

for (int j=0; j<34; j++) {

s += (t[j]\*4)/5;

}

to

int realDaysPerIdealDay = 4;

const int WORK\_DAYS\_PER\_WEEK = 5;

int sum = 0;

for (int j=0; j < NUMBER\_OF\_TASKS; j++) {

int realTaskDays = taskEstimate[j] \* realDaysPerIdealDay;

int realTaskWeeks = (realdays / WORK\_DAYS\_PER\_WEEK);

sum += realTaskWeeks;

}

* Hungarian notation:
  + In modern programming languages it is not necessary to follow HN and even without using it we could write and maintain a good readable code.
    - int nCount; ❌ (Here the n represents integer i.e number )
    - int count; ✔
* Class names:
  + Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.
* Method names:
  + Methods should have verb or verb phrase names like postPayment, deletePage, or save. Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the javabean standard.
* Don’t be cute:
  + What the function named HolyHandGrenade is supposed to do? Sure, it’s cute, but maybe in this case DeleteItems might be a better name. Choose clarity over entertainment value.

**Chapter 3 - Functions:**

* Small:
  + The function should be as small as possible.
* Do one thing:
  + A function must do one thing, it must do it well and it must do it only.
* Reading code from top to bottom:
  + We want the code to read like a top-down narrative. It’s also called the stepdown narrative.
* Funtion arguments:
  + The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.
* Common monadic forms:
  + Monadic functions are functions that take a single argument. There are **three** common forms of monadic functions, each serving a distinct purpose:
  + Predicate (Questioning a Condition):
    - These functions return a boolean value, indicating whether a certain condition about the argument holds true.
    - Eg:

boolean fileExists(String filename) {

File file = new File(filename);

return file.exists();

}

* Transformation (Operating on the Argument):
  + These functions take an argument, perform an operation on it, and return a transformed value.
  + Eg:

InputStream fileOpen(String filename) {

return new FileInputStream(filename);

}

* Event (Using the Argument to Alter System State):
  + These functions use the argument to trigger an event or alter the system's state. They do not return a value.
  + Eg:

void passwordAttemptFailedNtimes(int attempts) {

if (attempts >= 3) {

lockAccount();

}

}

* Flag arguments:
  + Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!
* Argument Objects:
  + When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.
  + Consider, for example, the difference between the two following declarations:
    - Circle makeCircle(double x, double y, double radius);
    - Circle makeCircle(Point center, double radius);
* Error handling is one thing:
  + Functions should do one thing. Error handing is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword try exists in a function, it should be the very first word in the function and that there should be nothing after the catch/finally blocks.