**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:

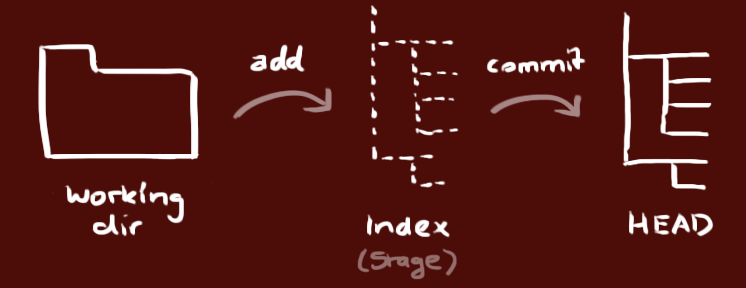
$ git diff

$ 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.
* The Boy Scout Rule
  + It is not enough to write the code clean, we must also clean and change the code you find ugly or not clean.

**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.

**Day 2:**

**Chapter 4 - Comments**

* Comments will not always give us the right information. Because the code evolves and changes. But not the comments.
* So it may contain misinformation and guide us wrong.
* Try writing a clear code with less comments than a complex code with lots of comments
* It takes only a few seconds of thought to explain most of your intent in code. In many cases it’s simply a matter of creating a function that says the same thing as the comment you want to write.
  + Eg: “if (employee.isEligibleForFullBenefits())“
* Legal comments:
  + // Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
  + // Released under the terms of the GNU General Public License version 2 or later.
* Clarification:
  + assertTrue(a.compareTo(a) == 0); // a == a
  + assertTrue(a.compareTo(b) != 0); // a != b
  + assertTrue(ab.compareTo(ab) == 0); // ab == ab
* Warnings of consequences:
  + // Don't run unless you

// have some time to kill.

public void \_testWithReallyBigFile()

{

writeLinesToFile(10000000);

response.setBody(testFile);

response.readyToSend(this);

String responseString = output.toString();

assertSubString("Content-Length: 1000000000", responseString);

assertTrue(bytesSent > 1000000000);

}

* Journal comments must be removed:

\* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);

\* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);

\* 13-Mar-2003 : Implemented Serializable (DG);

\* 29-May-2003 : Fixed bug in addMonths method (DG);

* Closing Brace Comments

} //while

System.out.println("wordCount = " + wordCount);

System.out.println("lineCount = " + lineCount);

System.out.println("charCount = " + charCount);

} // try

catch (IOException e) {

System.err.println("Error:" + e.getMessage());

} //catch

* Try not to use this kind of comments instead try creating smaller functions.
* Commented out code:
  + Remove the code that you commented as nobody will dare to delete the delete that commented code as they don’t know whether it is needed or not.
  + At last it stays like a wine in the bottle forever.
* HTML comments:
  + They are abomination and makes it very tough to read it.
* Too much information:
  + Don’t write paras of comments and irrelevant information. Keep it short and neat.

**Chapter 5 - Formatting**

* Newspaper metaphor:
  + Code just like the newspaper is formatted. i.e., The header, then the first para gives us the idea of the whole article and later we get more details on it.
  + Same like that, Give good name to the function that makes it clear for what it is used for.
  + Then give short information about it then the whole complex code.
* Vertical Openness Between Concepts:
  + Leave lines between the two different concepts or logics in a class to individually identify them easily.
* Vertical distance:
  + Concepts that are closely related should be kept vertically close to each other.
* Variable declarations:
  + Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear a the top of each function.
* Dependent functions:
  + If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow.
* Horizontal openness and density:
  + Leave space between words or operators, brackets when needed.
  + Like leaving a space between “ = “ and space after and before curly braces.
  + Also keep the variables that are related to each other closely.
* Indentation:
  + Maintain the indentation with respect to the scope of the function.
  + Larger scope must be on the left. And the smaller scopes must be within them.

**Chapter 6 - Objects and Data structures:**

* Law of demeter:
  + Each component should have limited knowledge about other components.
  + Each component should talk only to friends not to stranger.
  + Only talk to immediate friends.
* Train wrecks:
  + final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
  + This kind of code is often called a train wreck because it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided [G36]. It is usually best to split them up as follows:
    - Options opts = ctxt.getOptions();

File scratchDir = opts.getScratchDir();

final String outputDir = scratchDir.getAbsolutePath();

* Data Transfer Objects:
  + The most perfect form of a data structure is a class with public variable and no function. They are called as DTO.
  + DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, and so on.

**Trees:**

* + Root - A node that has no parent
  + Leaf - A node that has no child
  + Depth - No of edges from the root to node
  + Height - No of max edges from node to leaf
  + Predecessor - All the nodes from from root to node
  + Successor - All the nodes from node to leaf
  + Siblings - The nodes of same parent
  + Nodes = n, Edges = n-1 (as root node does not have any edge)
  + Degree - no of children of a node
  + Max no of nodes at level i for binary tree = 2^i
  + Max no of nodes in a binary tree = 2^(h+1) -1
  + Min no of nodes in a binary tree of height h = h+1
  + Max height of a binary tree with n nodes = n-1 nodes
* Full/Proper/Strict Binary tree:
  + Tree with 0 or 2 children / All nodes except leaf has two children
* Complete binary tree:
  + All levels except the last level must be completely filled.
  + And in the last level it must be filled from the left to right.
* Perfect binary tree:
  + All levels are completely filled.
  + Or all the leaf nodes must be in the same level.
* Degenerate binary tree:
  + All the internal nodes must have only one child.
* Balanced binary tree:
  + Absolute difference between height of left and right sub tree must be maximum of 1.
* Tree Traversal:
  + Preorder, Inorder, Postorder => It is said based on root and its placement.
  + Preorder: Root , Left, Right
  + Inorder: Left, Root, Right
  + Postorder: Left, Right, Root
* Binary Search tree:
  + Binary search tree is a data structure that has smaller element in the left and greater element in the right side.

**Day 3:**

**Chapter 7 - Error Handling:**

* Use Exceptions Rather Than Return Code:
  + Use try catch instead of returning error in if else.
* Provide context with exceptions:
  + Must provide enough information for the error.
  + You may use stack trace, but it’s not enough. If it is a login error, throw the error with appropriate message with it.
* Don’t return null
  + High risk of getting null pointer exception
* Don’t pass null

**Chapter 8 - Boundaries:**

* Using third party code:
  + The service providers focus on all the users as they can get a large audience. But it may be a problem for the service users.
  + For example say, I use a map container n my application and I don’t need the user to add or clear points in the map.
  + But the map service provider may default may have the option to clear and add points in the map.
* Exploring and learning boundaries:
  + Testing the open source API is more time consuming and also integration of it.
  + So we can make a learning log.
  + Just call the API, and make some log statements to understand it.
* No surety that the open source will work forever.
  + Every time new updates from the third party API is given, it’s a new risk then.
  + As the newer version may not be compatible with out project.
  + Which makes us to stay in the older version for a long time than usual.
* Using code that does not exist yet:
  + Sometimes there may be time like we don’t have the API payloads to develop a module.
  + Instead of stopping the work we can just create a dummy function or variables that may act as an API until the real API is created.
* Clean boundaries:
  + Our code must be less dependent on the third party code.
  + We must rather depend on code that we have more control on.
  + As we don’t have much control over third party particulars we must have less contact with it.
  + A good code takes less time and cost to design, and we must code that way.

**Chapter 9 - Unit tests:**

* Three laws of TDD(Test Driven Development):
  + You cannot make any production code without writing unit test code for it.
  + You cannot write more unit test code that is sufficient to fail.
  + You cannot write any more production code before making the current failing test pass.
* Keeping test clean:
  + Test code is as important as production code.
  + Back in days, there was a complain among the developers on the test suite.
  + As the production code changes the test fails. This made them completely avoid the test suites.
  + Later it ended up with dozens of bugs and frustrated customers.
  + If they maintained the test code clean this might not be happened.
* Tests enable flexible coding:
  + Without tests, every change is a possible bug.
  + But with test we no need to worry about the bug as it would let us know.
* Test codes must be very readable and clean.
* Every test function must have only one test statement.
* Clean tests follows five other rules called : **F.I.R.S.T**
  + **F**ast : The test codes must run fast. If not we’ll not run that often. If not running often we may get bugs.
  + **I**ndependent : It must not depent on each other. If it does, then the failure of one test will lead the others to fail.
  + **R**epeatable : The test must be repeatable in any environment. Like in production env, QA env and even without network connection.
  + **S**elf validating: The test file must return a Boolean value. whether true or false. We must not manually read the logs to validate the tests whether it is passed or not as it may take more time.
  + **T**imely : We must write the tests in timely fashion. It must be written before production code. If we write test after production code we may find it very hard to test the code, as we may not designed the production code to be testable.

**Day 4:**

**Chapter 10 - Classes**

* **Class organization:**
  + Class should begin with list of variables first.
  + Public and static constraints first, then private static constraints.
  + After the variables comes the public func then the private func.
* Classes must be encapsulated.
* Classes should be small. And each class must have single responsibility only.
* Cohesion:
  + Maintaining the variables and the methods are strongly related to each other.
  + Maintaining cohesion results in creating many small classes.
* Isolating from change:
  + Instead of writing concrete classes, using abstract classes and interface gives the advantage of isolation of the impact of changes those details.

**Chapter 11 - Smells and Heuristics:**

**Comments:**

* Inappropriate Information:
  + Comments must contain only the technical notes about the code and design.
  + Details like author name and last updated time must not appear in code.
* Obsolete comment:
  + A comment that has gotten old, irrelevant, and incorrect is obsolete.
  + These kind of comments must be either removed or updated frequently.
* Redundant comment:
  + A comment that describes itself. For eg : i++; // increment i.
* Poorly written comment:
  + A comment that is worth write is to be worth written well. Carefully use the words and punctuation to write comments.
* Commented out code:
  + Must not be kept, they must be deleted or removed if the code is not necessary.

**Environment:**

* Build
  + Building a project must be a single step and easy. Like cd into frontend folder and writing “npm i” to install. It is very simple and straight forward.
* Test:
  + All the unit test must be done using a single command or a single button in an IDE.
* Functions:
  + A function without argument is best. And it is ok with one or two arguments.
  + Even three is questionable and should be avoided.
  + Avoid flag arguments which means it does more than one task.
  + Remove the dead function that is never called.

**General:**

* Multiple languages:
  + Even it is not completely possible to avoid using multiple programming languages in a single file, we must take that pain and try minimizing it.
* Check the boundary conditions:
  + The logic may work in few cases but may fail in edge cases. Make sure to make them work.
* Override safeties:
  + Ignoring or turning off the warning may let us to build the app. But it may cause potential threats later.
* Duplications:
  + Must avoid duplication of code and it must be reusable.
  + And it causes less error.
* Too much information:
  + Always try to avoid exposing the details to other classes and functions.
* Dead code:
  + Remove the dead code when you find it. It complies everytime but never executes.
* Vertical separation:
  + The variables related to the functions should be closely placed.
* Inconsistency:
  + If you are doing something in some order, try to stick with it.
  + For example if you name something like “processVerificationRequest” try to keep the name of the related variable as “processDeletionRequest” with common prefix process.
* Feature Envy:
  + A method in a class should depend on the method and variable of that same class. Not the other class.
* Misplaced responsibility:
  + As a programmer we must know where to place the code and where not to.
* Inappropriate static:
  + Making inappropriate static variables may cause of creating a common variable for all the objects rather than creating it for each object.
* Give variables a name that indicates what value do they hold.
* Same for the function name, it should say what it does through the name.
* Understand the algorithm:
  + Even though the function gives right answer, we must know how it works, and what is right.
* Prefer polymorphism than using if-else and switch statements.
* Every team should follow certain coding standards.
* Be precise:
  + Very precise and cautious is must. For example if we are calling a function, make sure it is not returning null.
* Structure over convention:
  + Naming conventions are great, but it is not greater than an abstract class with abstract method. Because it enforces the sub class to implement the methods.
* Encapsulate conditions:
  + For example:

if (shouldBeDeleted(timer))

is preferable to:

if (timer.hasExpired() && !timer.isRecurrent())

* Avoid negative conditions:

For example:

if (buffer.shouldCompact())

is preferable to:

if (!buffer.shouldNotCompact())

* Keep configurable data in high level or in top.
  + For example like importing the data base connection at the starting and configuring it.
* Avoid transitive navigation.
  + Writing shy code. (Law of demeter)
  + It is like not all the modules must know about other modules.

**Java:**

* Avoid long imports.
  + Instead you can just “import java.util.\*;”
* Don’t inherit constants.
* We must use Enums over constants.

**Names:**

* Use descriptive names.
* Choose Names at the Appropriate Level of Abstraction.

public interface Modem {

boolean dial(String phoneNumber);

boolean disconnect();

boolean send(char c);

char recv();

String getConnectedPhoneNumber();

}

Instead of the above code we can write as,

public interface Modem {

boolean connect(String connectionLocator);

boolean disconnect();

boolean send(char c);

char recv();

String getConnectedLocator();

}

* Use Nomenclature where possible:
  + Use the same word or term or some structure that everybody uses.
  + Like using camelCase and toString method is java to print the objects.
* Unambiguous Names:
  + Give more specific names.
  + For eg: The name temp is not that specific. But using tempFilePath is more specific.
* Use tiny names for short scope and large names for long scope.
* Avoid encoding:
  + int iAge; // 'i' prefix indicates it's an integer
  + float fSalary; // 'f' prefix indicates it's a float
  + These are all the Hungarian pollutions in variable name and it should be avoided in todays env.

**Tests:**

* Insufficient tests:
  + Write test codes that cover and check all the calculations.
* Use coverage tool in testing:
  + Many IDE’s provide markings of the coverage in green and red.
* Don’t skip trivial tests:
  + They are easy to write and their documentary value is higher than the cost to produce them.
  + **Definition**: Trivial tests refer to test cases that cover straightforward and simple scenarios within a software application.
* An Ignored Test Is a Question about an Ambiguity;
  + Ambiguity means it is uncertain about it behavior or meaning.
* Test boundary conditions.
* Exhaustively test near bugs:
  + When you find a bug, search for more bugs the bugs may not be alone.
* Patterns of failure:
  + Sometimes we may find situations like where the test cases fails in some order.
  + Sometimes we may get error for numbers having length greater that 5.
  + Sometimes we may get error for all negative number.
* Test coverage pattern should be revealing.
  + We must identify the problem just by looking the passing test cases in test.
  + It must give us some idea where the error occurs.
* Test should be very fast.