**SVS DevNet Hands-On Week 1**

1. OOP Concepts & Coding Best Practices

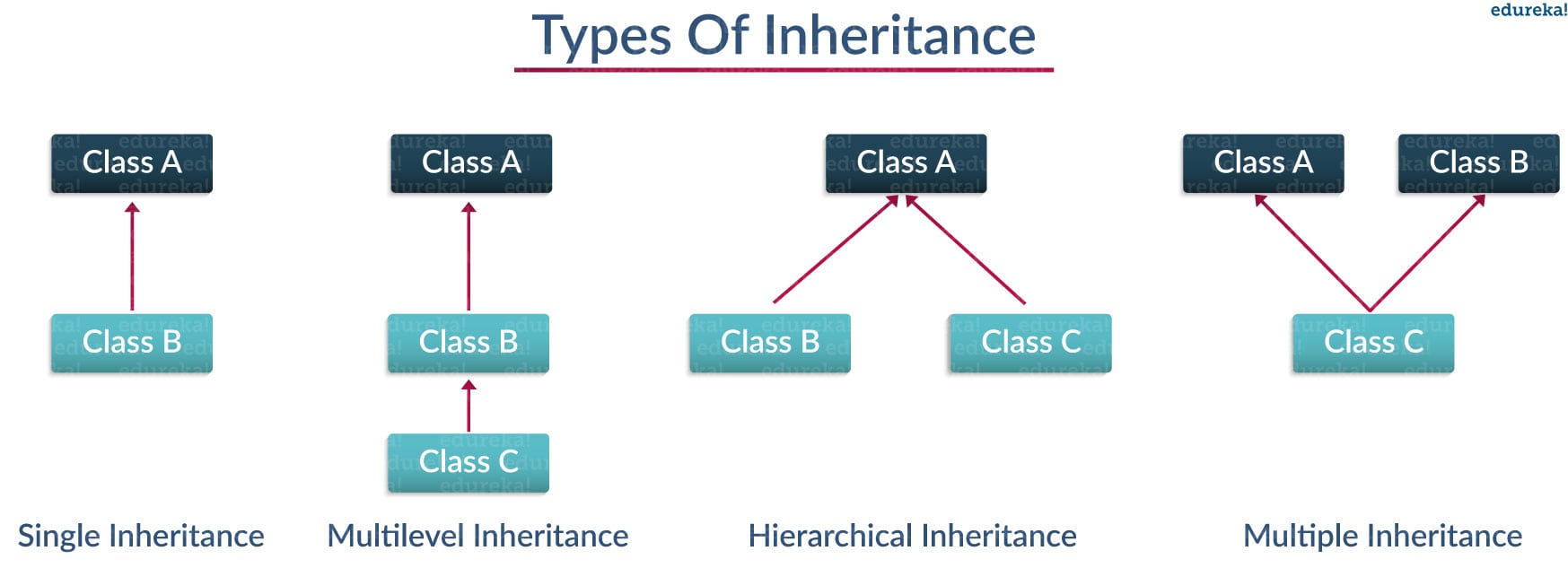
OOP -> Object Oriented Programming

Real world entities are modelled as objects while programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function. Another objective is to create reusable code.

|  |  |
| --- | --- |
| **Object-Oriented Programming (OOP)** | **Procedural-Oriented Programming (Pop)** |
| It is a bottom-up approach | It is a top-down approach |
| Program is divided into objects | Program is divided into functions |
| Makes use of *Access modifiers*  ‘public’, private’, protected’ | Doesn’t use *Access modifiers* |
| It is more secure | It is less secure |
| Object can move freely within member functions | Data can move freely from function to function within programs |
| It supports inheritance | It does not support inheritance |

The common OOP Concepts are :

1. Abstraction
2. Encapsulation
3. Modularity
4. Inheritance
5. Polymorphism
6. Abstraction :
7. Encapsulation :
8. Modularity :
9. Inheritance :



1. Polymorphism :

**Coding Best Practices :**

[DRY](https://thevaluable.dev/dry-principle-cost-benefit-example/) – Do not Repeat Yourself

[KISS](https://people.apache.org/~fhanik/kiss.html) – Keep it Simple, Stupid !

C:\Users\sucvenka>python

Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> import this

***The Zen of Python, by Tim Peters***

***Beautiful is better than ugly.***

***Explicit is better than implicit.***

***Simple is better than complex.***

***Complex is better than complicated.***

***Flat is better than nested.***

***Sparse is better than dense.***

***Readability counts.***

***Special cases aren't special enough to break the rules.***

***Although practicality beats purity.***

***Errors should never pass silently.***

***Unless explicitly silenced.***

***In the face of ambiguity, refuse the temptation to guess.***

***There should be one-- and preferably only one --obvious way to do it.***

***Although that way may not be obvious at first unless you're Dutch.***

***Now is better than never.***

***Although never is often better than \*right\* now.***

***If the implementation is hard to explain, it's a bad idea.***

***If the implementation is easy to explain, it may be a good idea.***

***Namespaces are one honking great idea -- let's do more of those!***

>>>

https://gist.github.com/sloria/7001839

**API Data Formats**

Data format represents syntax coding data that could be read by another machine but in such a way that is easy to understand for humans, too.

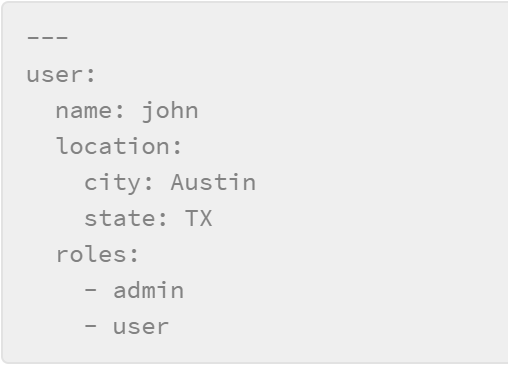
**Common data formats and their uses**

|  |  |
| --- | --- |
| Data format | Use |
| YAML Ain't Markup Language (YAML) | Configuration files |
| JavaScript Object Notation (JSON) | Communication server—web page, configuration files |
| eXtensible Markup Language (XML) | Transformation with XSL, Applying XML schemas |

In the above three formats, data is generally represented as key-value pairs.

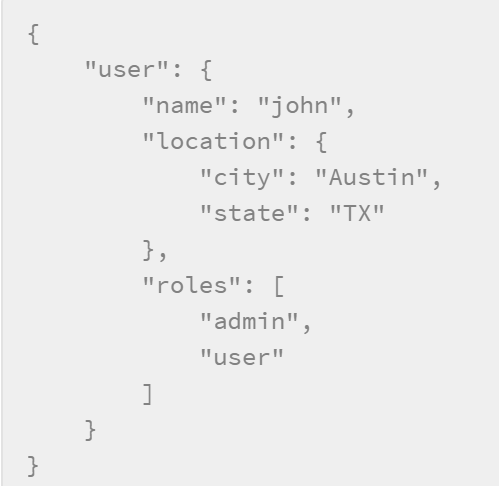
**YAML https://yaml.org/refcard.html**

* Not a markup language like JSON and XML. YAML is the most humanly readable of all the formats and at the same time is just as easy for programs to use.
* Whitespaces are significant to YAML because whitespace indentation defines a structure of a YAML file. Tab indentations are not allowed in YAML because they are treated differently by different tools



**JSON**

* JSON syntax uses curly braces, square brackets, and quotes for its data representation.
* All whitespaces that you see in JSON is just for humans consuming and reading the data; they have nothing to do with how the JSON file will be consumed by an application or script (except whitespaces within a value of a key).

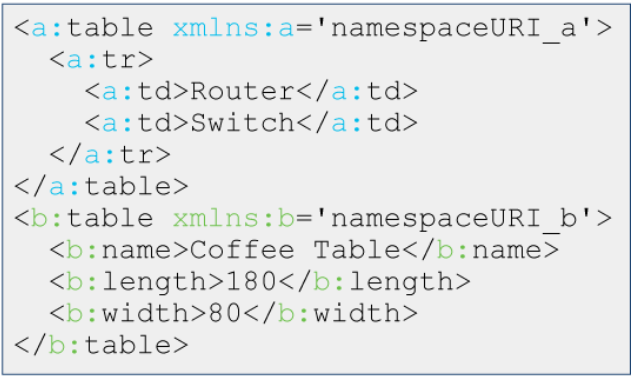


**XML https://www.w3schools.com/xml/default.asp**

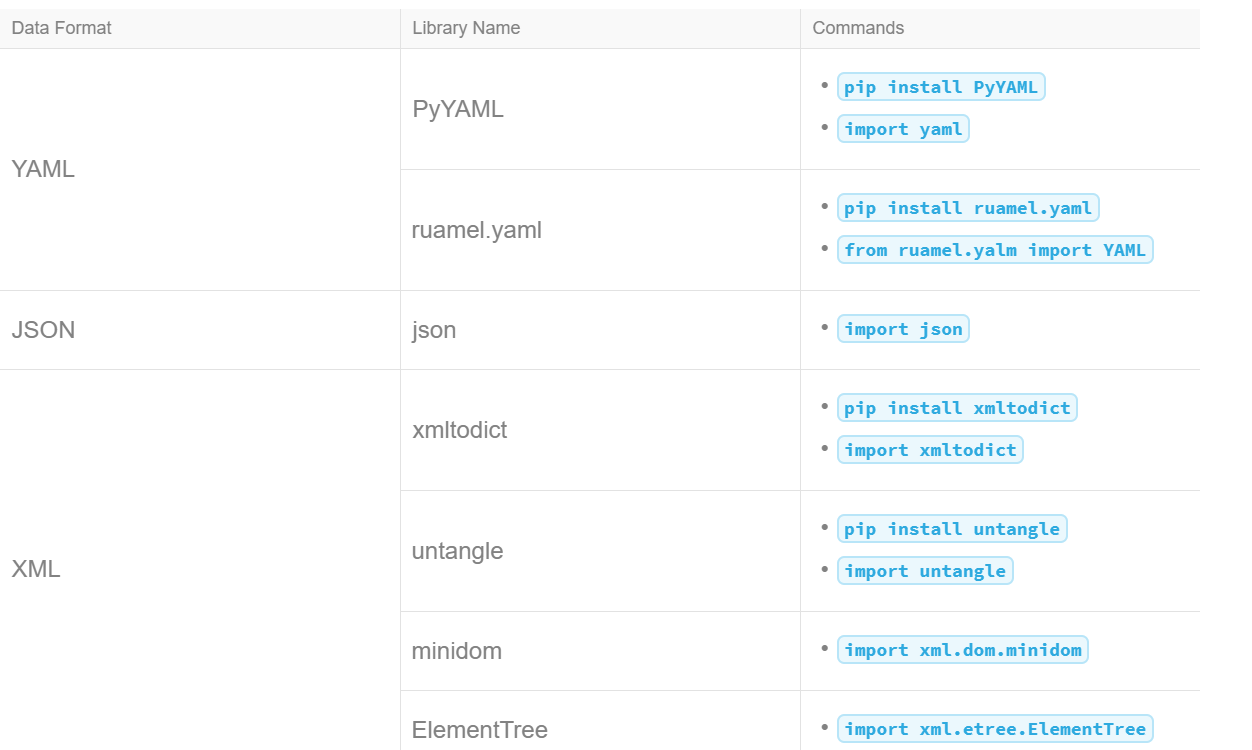
For that purpose, XML code heavily uses **<tags></tags>** to surround elements in form as**<key>value</key>**. All the information about an object is defined inside the opening **<tag>** and **</tag>** with a slash to indicate a closing tag.

XML Namespaces are used to resolve conflicts in tags

Eg:



**Python Libraries to parse the three data formats:**



**Test Driven Development**

Test-driven development (TDD) is a software-development methodology where you write the test code before the actual production code.

**Steps:**

Development is done in iterations, where you do the following:

1. Write tests.
2. Run these tests; they must fail (code may not even compile).
3. Write source code.
4. Run all tests; they must pass.
5. Refactor the code where necessary.

**Benefits:**

* Gives you a clear goal—make the tests pass
* Shows specification omissions and ambiguities before writing code, avoiding potentially costly rewrites
* Uncovers edge cases for you to address from the start
* Makes debugging easier and faster, because you can simply run the tests
* Passing each test is a small victory that drives you forward.

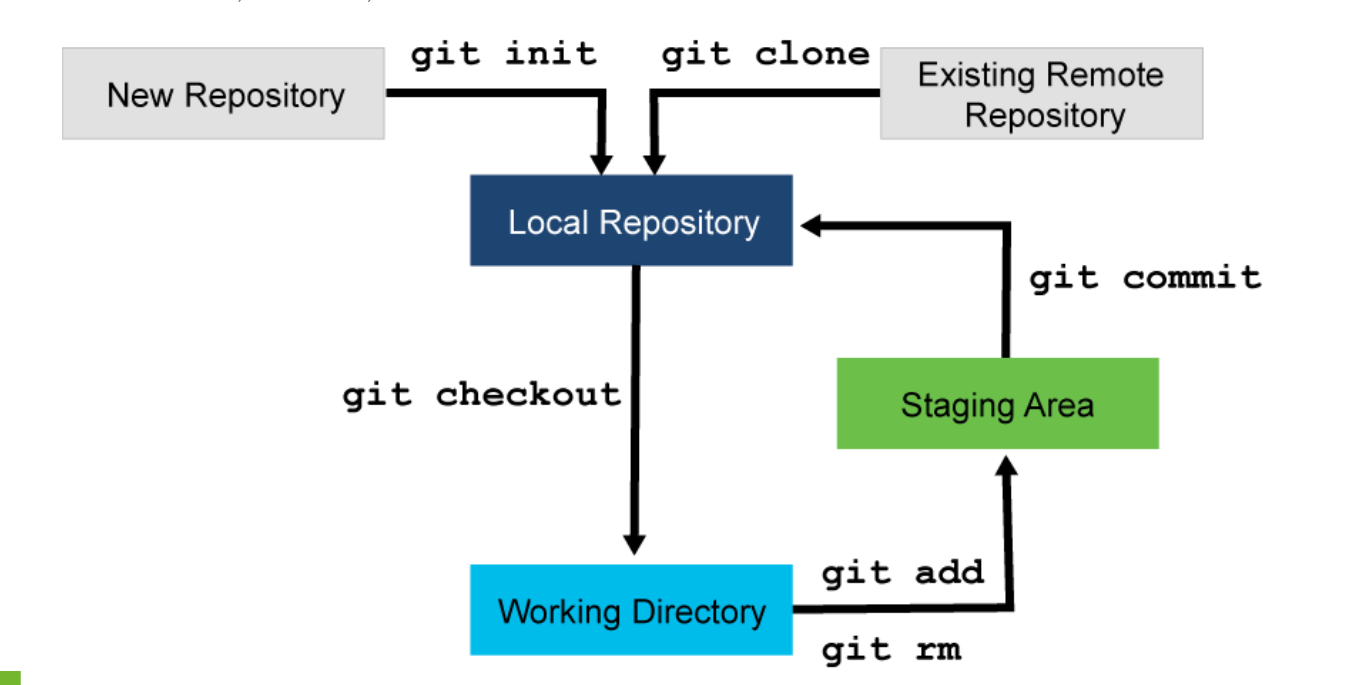
**Benefits of Modular design**:

1. **Maintain modularity :** Modules ensure that the code is easier to understand, and they lower the overwhelming feeling when you delve into the code for improvements or feature additions. They should encapsulate parts of functionality of a system and constrain how these parts interact among each other.
2. **Loose Coupling :** Loose coupling means reducing dependencies of a module, class, or function that uses different modules, classes, or functions directly. Loosely coupled systems tend to be easier to maintain and more reusable.
3. **Cohesion :** The purpose of a class or module should be focused on one thing and not too broad in its actions. Modules that contain strongly related classes and functions can be considered to have strong or high cohesion.

**Version Control with Git**

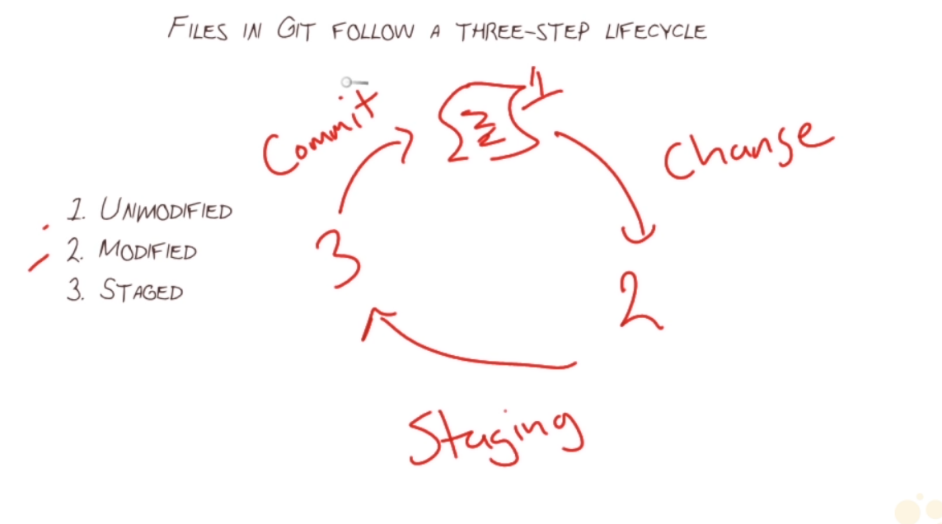
Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. One of the most important elements of Git is a repository. A repository is a directory that is initialized with Git.

**Components of Git Architecture:**

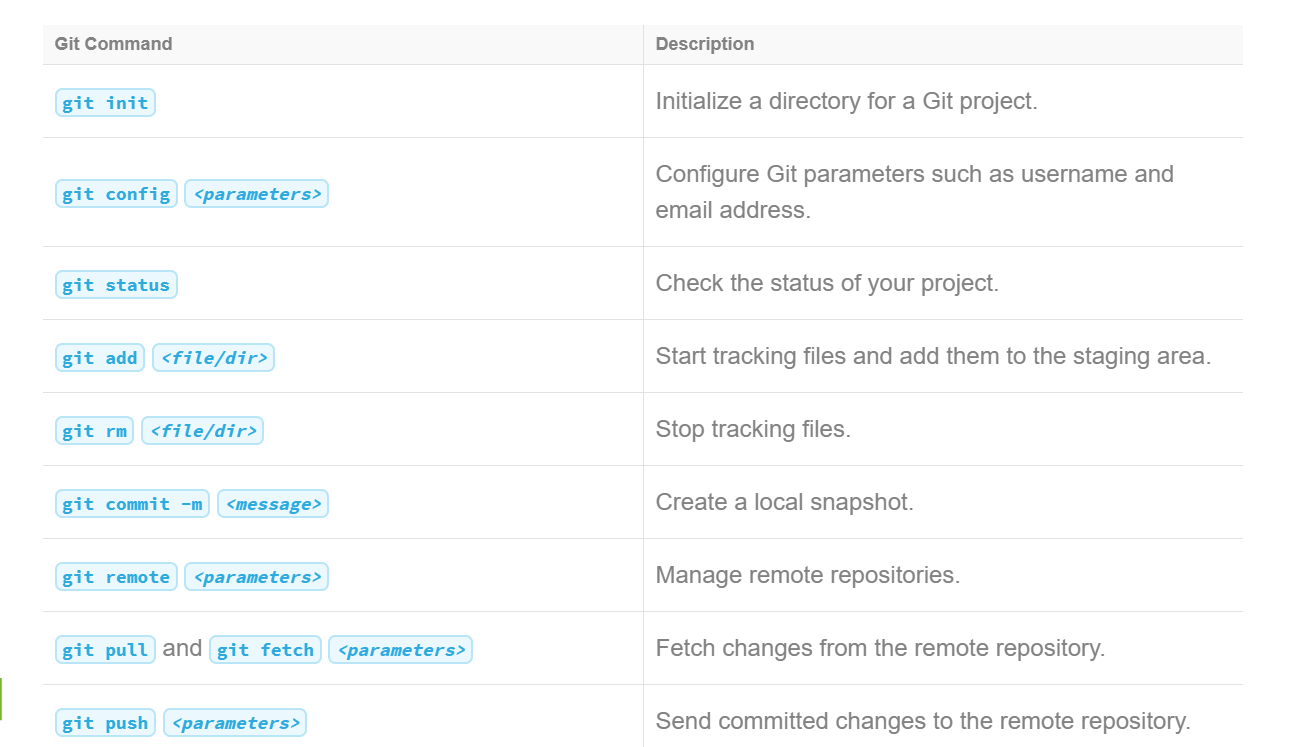


1. **Remote repository:** A remote repository is where the files of the project reside, and it is also from where all other local copies are pulled. It can be stored on an internal private server or hosted on a public repository such as GitHub, GitLab, or BitBucket.
2. **Local repository:** A local repository is where snapshots, or commits, are stored on the local machine of each individual.
3. **Staging area:** The staging area is where all the changes you actually want to perform are placed. You, as a project member, decide which files Git should track.
4. **Working directory:** A working directory is a directory that is controlled by Git. Git will track differences between your working directory and local repository, and between your local repository and the remote repository.

**Git file lifecycle**



**Git commands**



These commands will be explained using the git workflow.

A Git workflow usually consists of the following actions:

1. **Directory initialization for Git**

**Command:** git init

The present working directory will be started to be tracked by git. You can also downloading an existing repository using the git clone command.

**Command:** git clone <url>

In order to set the username ‘John Smith’ and his email of the git user, the following commands are used

git config -global user.name "John Smith"

git config -global user.email [john.smith@cisco.com](mailto:john.smith@cisco.com)

1. **Adding a file**

Files are added to the staging area using the command. The file switch.cfg is added in the following example:

git add switch.cfg

1. **Committing a file to a local repository**

To commit the changes made to the files, the following command is used and a message “initial commit” is added to the commit.

git commit -m "Initial commit"

git commit –amend --------🡪 replaces previous commit

1. **Pushing a file to a remote repository**

Git commit creates a checkpoint in your local repo. In order to share these changes to a remote repository, the following commands are used

git remote add origin [https://github.com/<user>/<repo](https://github.com/%3cuser%3e/%3crepo)> ------🡪remote repo URL is added

git remote -v ----🡪 lists remote repos

git push local master ---🡪 Pushes the mater branch to the remote repository set using the previous commands

1. **Updating a local repository by pulling changes from a remote repository**

The command ‘**git pull**’ pulls all changes from the remote repository set.

**Other commands:**

**git status**

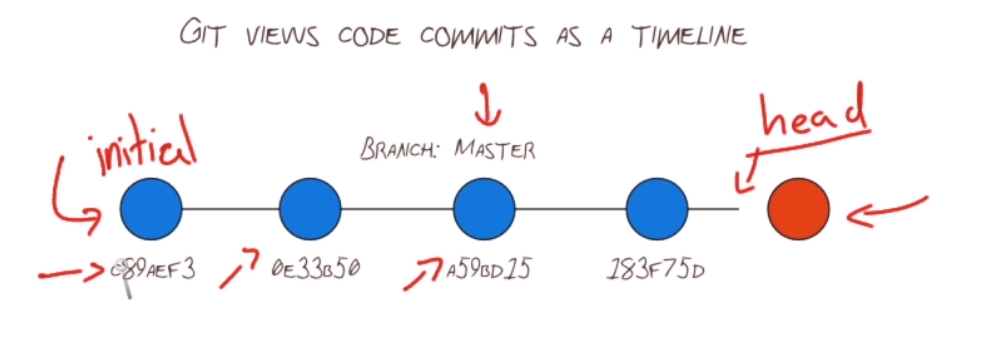
It shows what branch you are on and changes not staged for commit.

**Git stash and git stash pop**

You can use the **git stash** command to temporarily save your current changes. Once you finish with other work, you can then use the **git stash pop** command to retrieve your previously saved changes.

**Git log**

Git views code commit as a timeline



The git log command shows the history of commits and the hashes. You can add the parameter –oneline to show a brief view of all commits

Eg:

**$ git log** --oneline

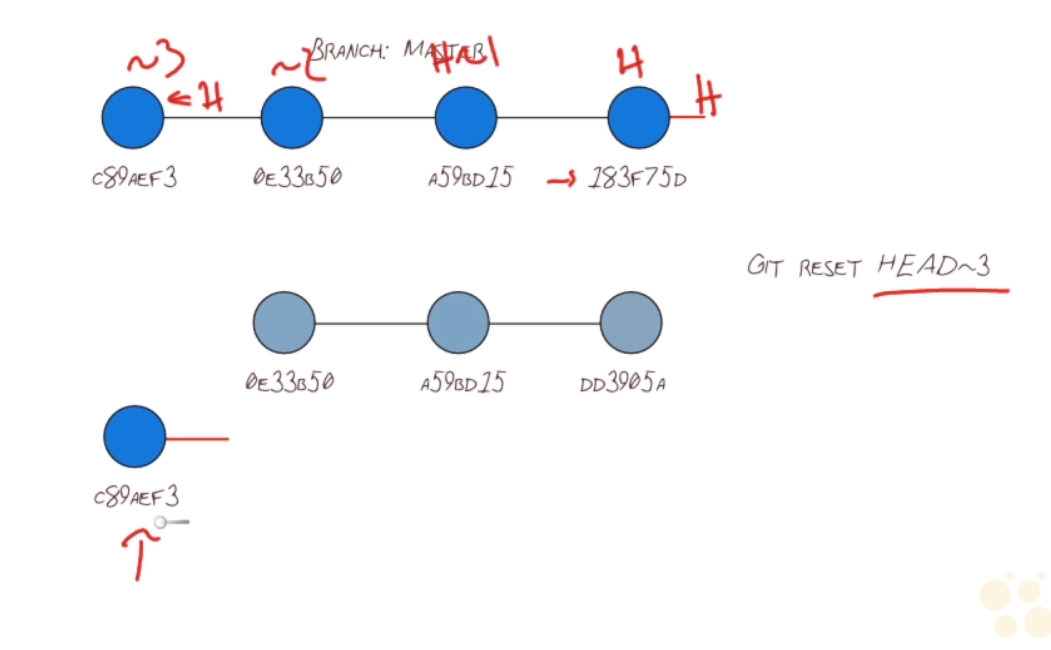
11123ce (HEAD -> master) Configure interface Loopback0 on switch2

e871a41 Configure interface Loopback0 on switch1

**git reset**

Removes commits resetting the head

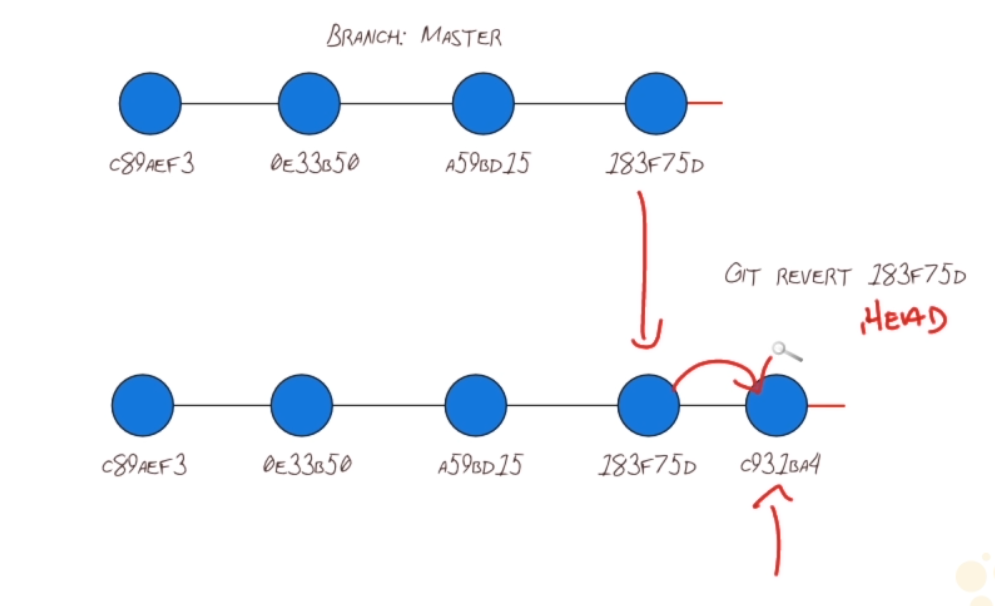
Eg: git reset HEAD~3 ------------🡪 moves the had three commits back. The three commits it went through are detached.



**Git revert**

Git reverts produces revserse commits and thereby preserves the history

Eg: git revert 183F75D



**Git restore –staged <filename>** -------🡪 moves file out of staging area

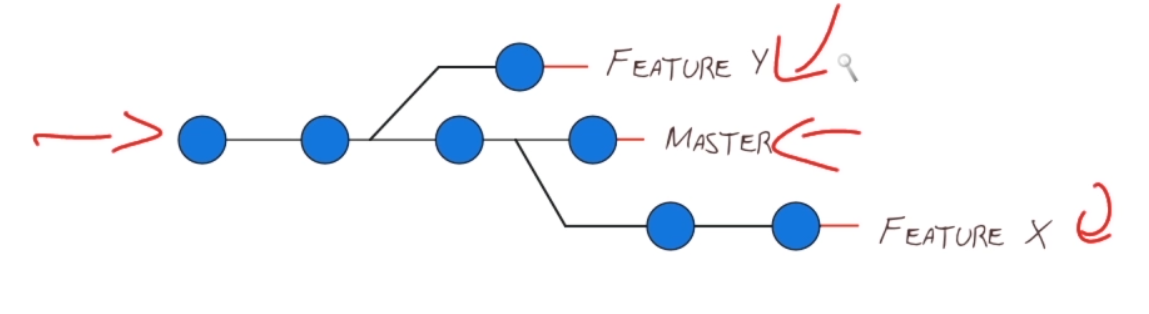
**Git rm <filename>** ------🡪 removes filename from repository and system

**Git reset –hard HEAD** --🡪undoes previous command

**Branching with git**

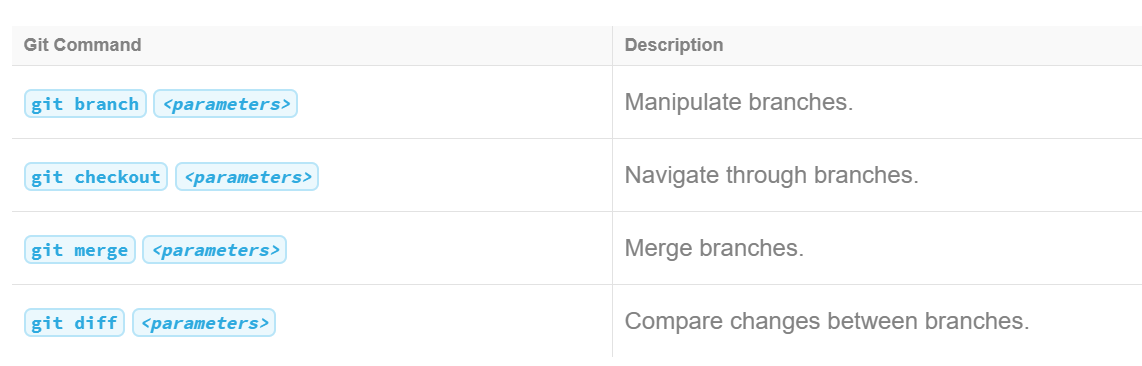
In git, you can create branches which are nothing but alternate timelines that do not interfere with each other. By default, all repositories start off with the ‘master’ branch.

*git branch <branch\_name>*  command creates a new branch with the specified name. The diagram below shows two branches – feature X and feature Y created from the master branch.



**Scenarios when branching is used:**

* New features are being developed
* Bug fixes are being done



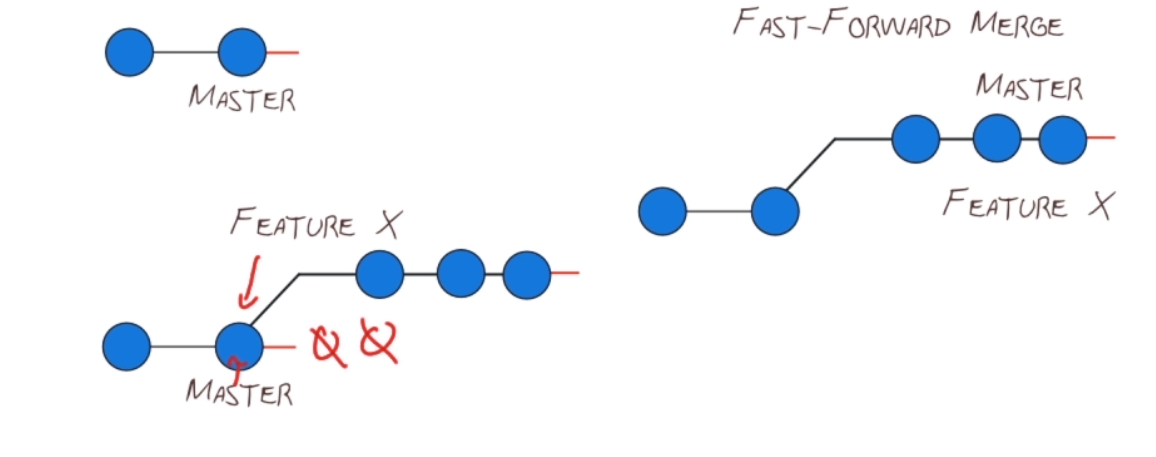
**Git checkout**

* Used to switch to another branch
  + git checkout <branchname>
* Used to checkout a remote branch
  + First the remote branch contents has to be fetched using the ‘git fetch –all’ command. After this: git checkout <remotebranch>
* Used to create a new branch and switch to it
  + git checkout -b <new-branch>

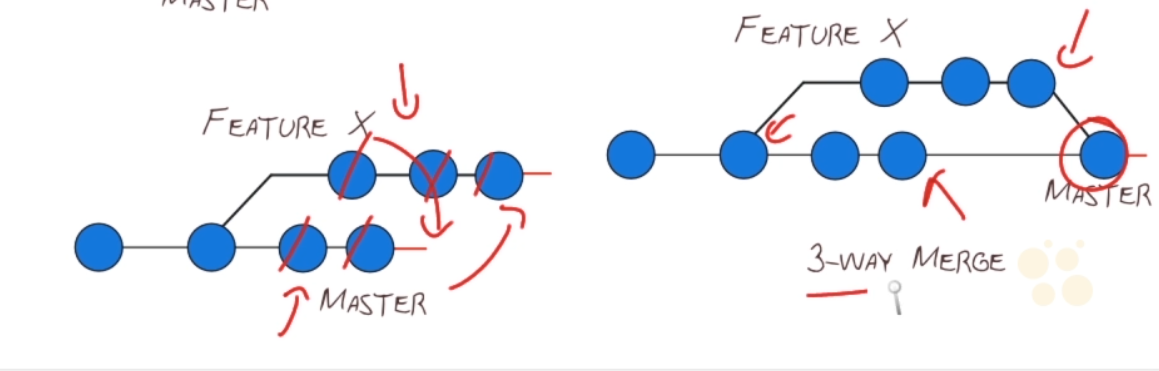
**Merging branches**

Use the git merge command to merge changes from another branch to the master branch and the command git push origin master to push the code to the remote repository.

1. Fast-forward merge is shown below



1. Three-way merge



Merge conflicts are common when using Git. They can arise in these scenarios:

* Two developers changed the same line or lines in the same file.
* One developer deleted the file, while the other developer changed it.
* One developer moved the file to other directory, while the other developer changed it.
* Both developers moved the file to different directories.

Git has a merge conflict resolution algorithm to handle conflicts, but in many cases it may not prove to be sufficient. In such cases one must manually resolve conflicts.