

# KOSS SELECTION TASK

Presentation on new git concepts

AMRITA GHOSH

ROLL NO: 21AE30004

Email id:

kgpamrita@gmail.com

## ACKNOWLEDGEMENT

I have taken help from these sites :

- <https://www.atlassian.com/git/tutorials>
- <https://www.javatpoint.com/>
- [https://www.git-tower.com/windows?utm\\_source=learn-website&utm\\_medium=navigation](https://www.git-tower.com/windows?utm_source=learn-website&utm_medium=navigation)
- <https://www.youtube.com/watch?v=8JJ101D3knE>

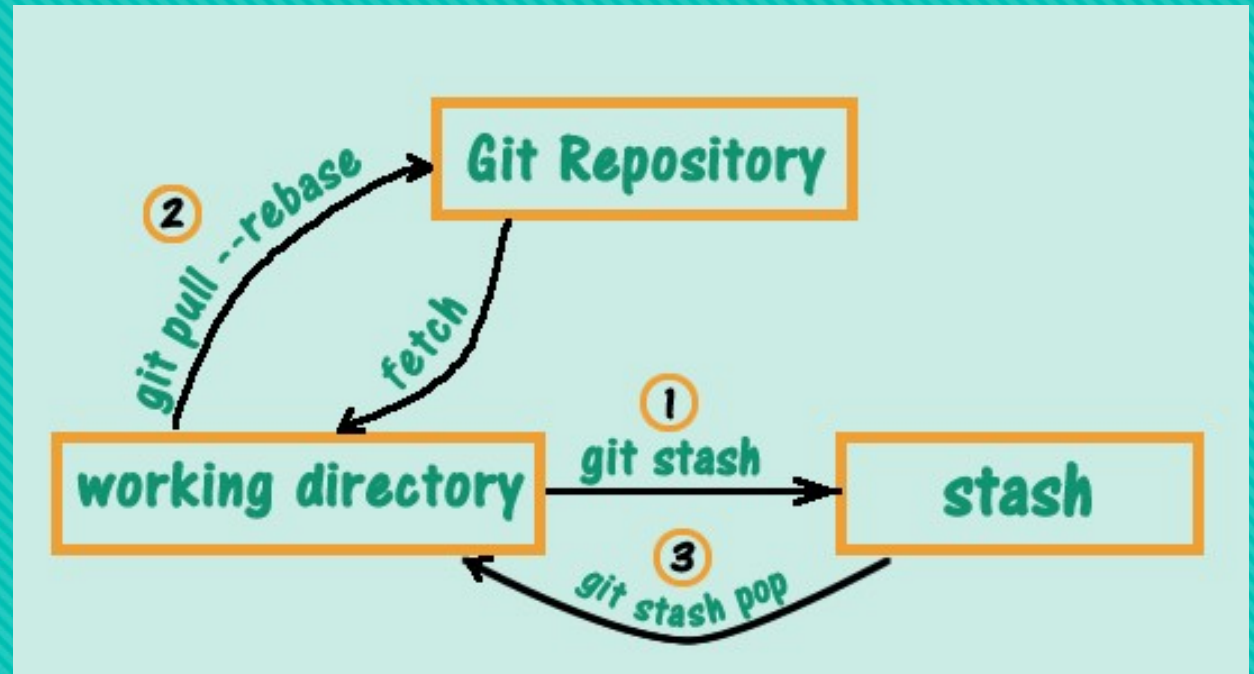
I have used some images available on Google as well.

This was the first time I tried to understand git. It was an interesting experience. Looking forward to learning more.

Amrita Ghosh.

# Git stash command.

Git stash is extremely useful when you have some changes that you want to save but aren't ready to make a commit. Git stash stores the changes you made to the working directory locally and allows you to retrieve the changes when you need them.



The simplest command to stash your changes is `git stash`:

```
$ git stash
```

```
Saved working directory and index state WIP on master; d7435644 Feat:  
configure graphql endpoint
```

```

amrit@Amrita-LAPPY MINGW64 /C/Ghosh (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.js
        new file:   file2.js

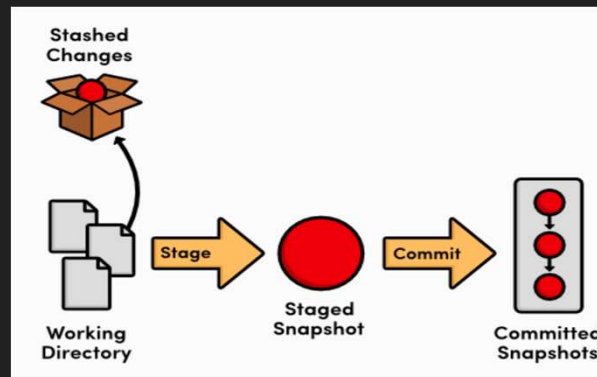
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.js

amrit@Amrita-LAPPY MINGW64 /C/Ghosh (master)
$ git stash
Saved working directory and index state WIP on master: 96983f5 remove bin direc
tory

```

## How Stash command works

1. Save changes to branch A.
2. Run git stash.
3. Check out branch B.
4. Fix the bug in branch B.
5. Commit and (optionally) push to remote.
6. Check out branch A
7. Run git stash pop to get your stashed changes back.



Before you can run git stash, you need to have some uncommitted changes in your Git repository. By default, git stash stores (or "stashes") the uncommitted changes (staged and unstaged files) and overlooks untracked and ignored files. Usually, you don't need to stash untracked and ignored files, but sometimes they might interfere with other things you want to do in your codebase.

You can use additional options to let git stash take care of untracked and ignored files:

`git stash -u` or `git stash --include-untracked` stash untracked files.

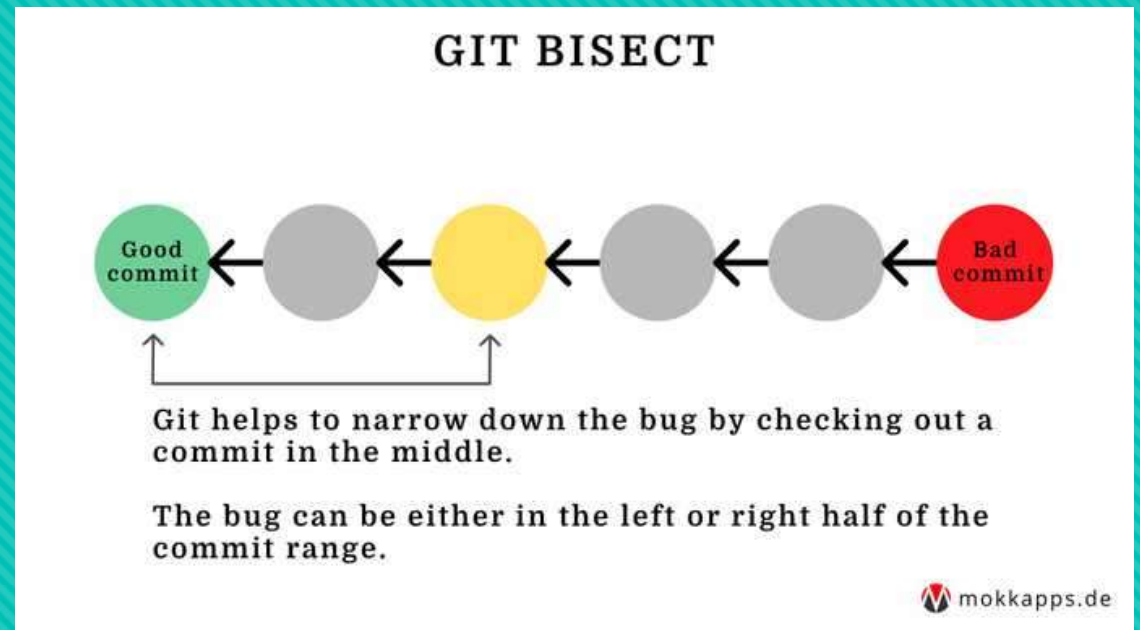
`git stash -a` or `git stash --all` stash untracked files and ignored files.

To stash specific files, you can use the command `git stash -p` or `git stash -patch`:



# Git Bisect Command

`git bisect` performs a binary search to find the faulty commit



The `git bisect` command is used to discover the commit that has introduced a bug in the code. It helps track down the commit where the code works and the commit where it does not, hence, tracking down the commit that introduced the bug into the code.

The following code demonstrates the use of git bisect to find the faulty commit.

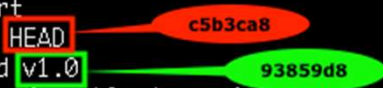
1. Initializing the repository
2. Creating commits to demonstrate git bisect
3. Finding the log history and narrowing down the bad commit.

We can now categorize the first commit as good and the last commit as bad. We check the contents of the test file.


git bisect now takes us to another commit. We repeat the process till the new commit to being bad.

git bisect finally returns the commit to us where the first error took place.

```
[Divya1@Divya:my_nav_app [dev] $git bisect start
[Divya1@Divya:my_nav_app [dev] $git bisect bad HEAD
[Divya1@Divya:my_nav_app [dev] $git bisect good v1.0
Bisecting: 2 revisions left to test after this (roughly 1 step)
[f61a7e8dbaa2492a03c877ffcba43eae704ded31] Adding clean function
Divya1@Divya:my_nav_app [(no branch, bisect started on dev)] $
```



* c5b3ca8	2019-08-07	Revised version to 1.1 (HEAD -> dev, master) [developer]	bad commit
* 84f9504	2019-08-07	Deleting files using library clean function [developer]	
* a6ac769	2019-08-02	Changed color for menu buttons (tag: bug) [developer]	first suspect
* f61a7e8	2019-08-07	Adding clean function [developer]	good commit
* 69f4813	2019-08-07	Add application paths in Readme file [developer]	
* 93859d8	2019-08-02	Running first CI/CD pipeline-code,build,run and test the code (tag: v1.0) [developer]	





# Git relog Command

The git relog command is used for Git to record updates made to the tip of branches. It allows returning to commits even to the ones that are not referenced by any branch or any tag. After rewriting history, the relog includes information about the previous state of branches and makes it possible to go back to that state if needed.

Reference logs, or "reflogs", record when the tips of branches and other references were updated in the local repository. Reflogs are useful in various Git commands, to specify the old value of a reference.

Git Reflog



  
**KEEP  
CALM  
AND  
USE  
GIT REFLOG**

## Recover a deleted branch using Git Reflog

Step 1: History logs of all the references

Step 2: Identify the history stamp

Step 3: Recover



Output:

```
[Divya1@Divya:learn_branching [master] $git checkout -b preprod HEAD@{4}
Switched to a new branch 'preprod'
[Divya1@Divya:learn_branching [preprod] $
```

```
jekyll master $ git reflog
6703083... HEAD@{0}: pull origin master: Fast forward
fe71d2b... HEAD@{1}: commit: Updating README with added
eac6b03... HEAD@{2}: commit: Making sure that posts fla
f682c8f... HEAD@{3}: commit: Added publish flag to post
9478f1b... HEAD@{4}: rebase: Modifying the README a bit
7e178ff... HEAD@{5}: checkout: moving from master to 7e
cda3b9e... HEAD@{6}: HEAD~1: updating HEAD
3b75036... HEAD@{7}: merge newfeature: Merge made by re
cda3b9e... HEAD@{8}: commit: Modifying the README a bit
6981921... HEAD@{9}: checkout: moving from newfeature t
7e178ff... HEAD@{10}: commit: Rewriting history is fun
4a97573... HEAD@{11}: commit: all done with TODOs
eb60603... HEAD@{12}: commit: README
```



# Git diff command

Diffing is a function that takes two input data sets and outputs the changes between them. `git diff` is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more.



Diff command is used in git to track the difference between the changes made on a file. Since Git is a version control system, tracking changes are something very vital to it. Diff command takes two inputs and reflects the differences between them.

The `git diff` command is a widely used tool to track the changes.

The `git diff` command allows us to compare different versions of branches and repositories. To get the difference between branches, run the `git diff` command as follows:

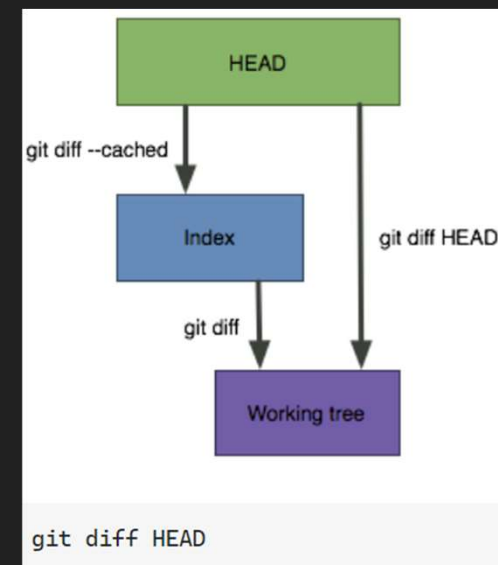
```
$ git diff <branch 1> <branch 2>
```

The above command will display the differences between branch 1 and branch 2. So that you can decide whether you want to merge the branch or not.

Sometimes, you might want to compare how a certain file differs in two branches. You can do this simply by adding the file's path:

```
$ git diff main feature/login index.html
```

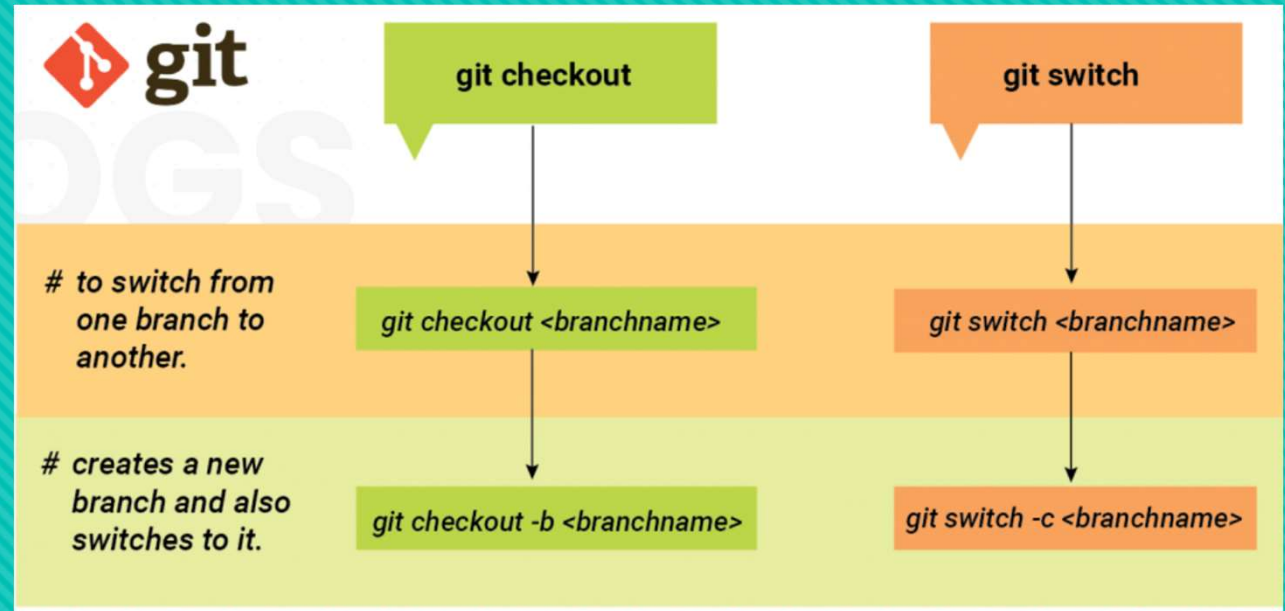
```
$ git diff index.html
diff --git a/index.html b/index.html
index f42e433..59c866e 100644
--- a/index.html
+++ b/index.html
@@ -1,8 +1,7 @@
<ul>
  <li>blue item</li>
  <li>red item</li>
- <li>green item</li>
- <li>purple item</li>
+ <li>orange item</li>
</ul>
```





# Git switch Command

The `git checkout` command allows you to switch branches by updating the files in your working tree to match the version stored in the branch that you wish to switch to



Switch to a specified branch. The working tree and the index are updated to match the branch. All new commits will be added to the tip of this branch.



The most common scenario is to simply specify the local branch you want to switch to:

```
$ git switch other-branch
```

This will make the given branch the new HEAD branch. If in one go, you also want to create a new local branch, you can use the "-c" parameter:

```
$ git switch -c new-branch
```

If you want to check out a remote branch (that doesn't yet exist as a local branch in your local repository), you can simply provide the remote branch's name. When Git cannot find the specified name as a local branch, it will assume you want to check out the respective remote branch of that name:

```
$ git switch remote-branch
```

This will not only create a local branch but also set up a "tracking relationship" between the two branches, making sure that pulling and pushing will be as easy as "git pull" and "git push".

If you have local modifications that would conflict with the branch you want to switch to, you can instruct Git to clear your working copy of any local changes

```
$ git switch other-branch --discard-changes
```

Finally, if you want to switch back to the previously checked out branch, you can simply do this by specifying only the "-" character:

```
$ git switch -
```

```
dev@schkn$ git switch feature
Switched to branch 'feature'
Your branch and 'origin/feature' have diverged,
and have 1 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)
```

To create a new branch use *git switch -c <branchName>* command.

```
aadit@DESKTOP-2RAM82V MINGW64 /c/RetargetCommon/GitLearning/gitSwitchDemo (main)
$ git branch -a
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main

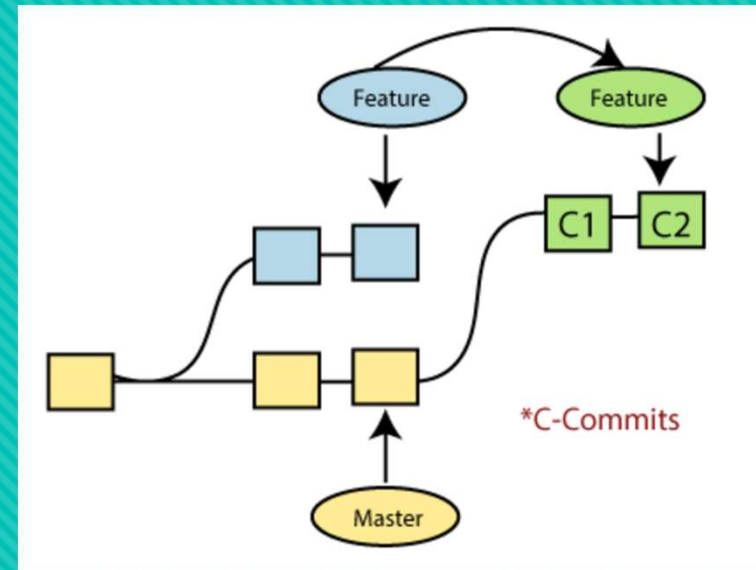
aadit@DESKTOP-2RAM82V MINGW64 /c/RetargetCommon/GitLearning/gitSwitchDemo (main)
$ git switch -c branchA
Switched to a new branch 'branchA'

aadit@DESKTOP-2RAM82V MINGW64 /c/RetargetCommon/GitLearning/gitSwitchDemo (branchA)
$ git branch -a
* branchA
main
remotes/origin/HEAD -> origin/main
remotes/origin/main
```

Please note here that you will be switched to the new branch using the *git switch* command like *git checkout*.

# Git rebase command

Rebasing is a process to reapply commits on top of another base trip. It is used to apply a sequence of commits from distinct branches into a final commit. It is an alternative of git merge command. It is a linear process of merging.



In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and it visualized the process in the environment of a feature branching workflow.

When you made some commits on a feature branch (test branch) and some in the master branch. You can rebase any of these branches.

Syntax:

```
$git rebase <branch name>
```

If there are some conflicts in the branch, resolve them, and perform below commands to continue changes:

```
$ git status
```

It is used to check the status,

```
$git rebase --continue
```

The above command is used to continue with the changes you made. If you want to skip the change, you can skip as follows:

```
$ git rebase --skip
```

When the rebasing is completed. Push the repository to the origin. Consider the below example to understand the git merge command.

Suppose that we have a branch say test2 on which you are working. You are now on the test2 branch and made some changes in the project's file newfile1.txt.

Add this file to the repository:

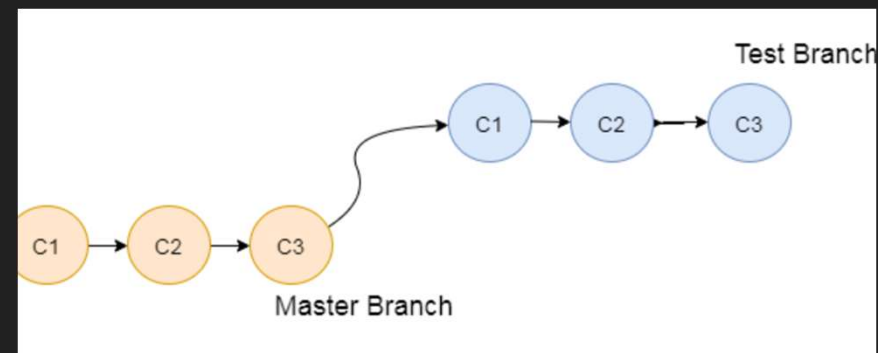
```
$ git add newfile1.txt
```

Now, commit the changes. Use the below command:

```
$ git commit -m "new commit for test2 branch."
```

The output will look like

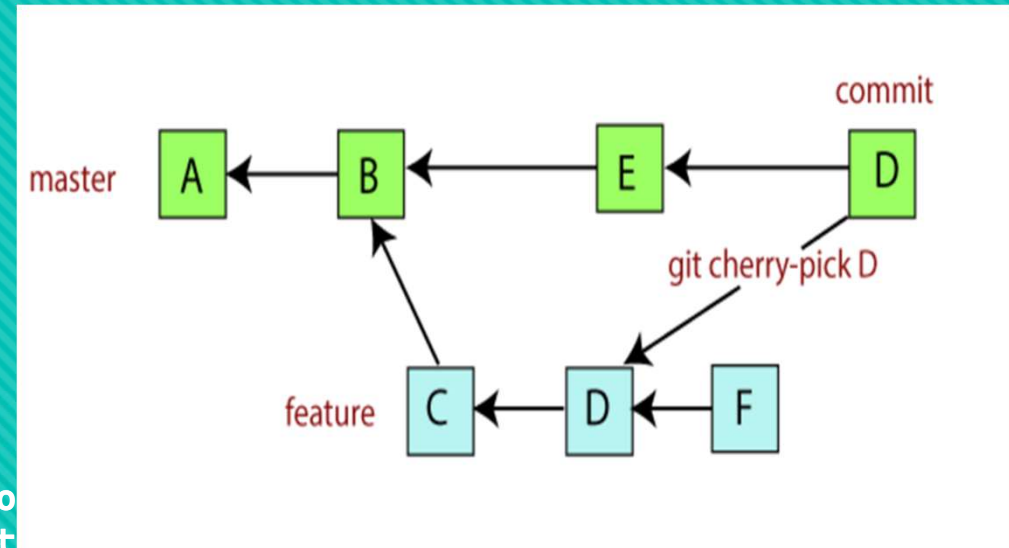
```
[test2 a835504] new commitfor test2 branch
1 file changed, 1 insertion(+)
```





# Git cherry-pick Command

`git cherry-pick` is a powerful command that enables arbitrary Git commits to be picked by reference and appended to the current working HEAD. Cherry-picking is the act of picking a commit from a branch and applying it to another. `Git cherry-pick` can be useful for undoing changes. For example, say a commit is accidentally made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong. The `git cherry-pick` is used to access the changes introduced to a sub-branch, without changing the branch.



The main motive of a cherry-pick is to apply the changes introduced by some existing commit. A cherry-pick looks at a previous commit in the repository history and updates the changes that were part of that last commit to the current working tree.

To demonstrate how to use `git cherry-pick` let us assume we have a repository with the following branch state:

```
a - b - c - d    Main
      \
        e - f - g Feature
```

`git cherry-pick` usage is straightforward and can be executed like:

```
git cherry-pick commitSha
```

In this example, `commitSha` is a commit reference. You can find a commit reference by using the `git log`. In this example, we have constructed let's say we wanted to use commit 'f' in main. First we ensure that we are working on the main branch.

```
git checkout main
```

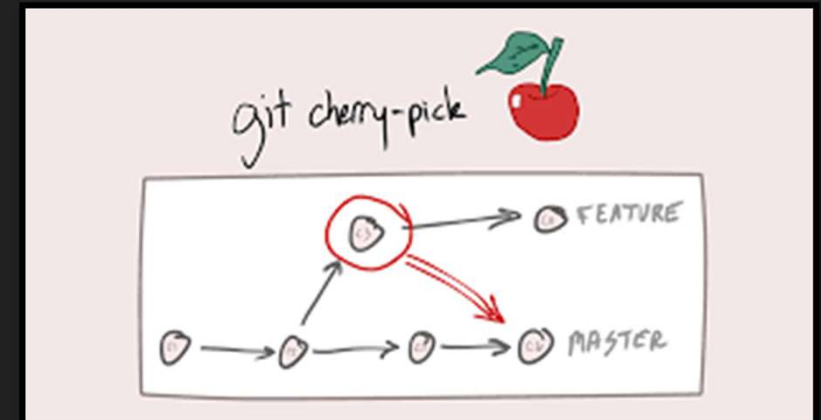
Then we execute the cherry-pick with the following command:

```
git cherry-pick f
```

Once executed our Git history will look like:

```
a - b - c - d - f    Main
      \
        e - f - g Feature
```

The `f` commit has been successfully picked into the main branch



# Conclusion:

This was a short presentation from my side on

`git stash`

`git bisect`

`git reflog`

`git diff`

`git switch`

`git rebase`

`git cherry-pick`



Thank

you