Git

Merge & Rebase

Git Merge & Rebase

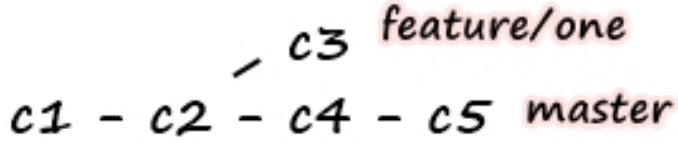
Both git merge and rebase, kind of perform the similar task of combining/joining the development histories together.

Now, what does that mean?
Simple, in git we have concept of branches, we usually have a default (master or main) branch, and then as part of our regular development/fix work we create feature/release branches out of it. Once we are done making our changes in our feature branches we want it to be published to remote branches and also combined with the main or master branch to keep it unified and accessible to others with their changes.

Hence, we need some feature to join these changes/history from different branches.

Both Merge & Rebase let you do that. The end goal is to combine the changes but how it is done is what makes Merge & Rebase different.

Let's understand them with a simple example -



You basically created a feature branch 'feature/one' from master when they were both at commit 'c2' and then you added a new commit 'c3' to the feature and also added 2 new commits 'c4' and 'c5' to master branch.

So, at the moment feature is unaware of the commits c4 & c5 which are on the master and also the master branch has no clue of commit c3 which is on feature.

Ideally that's fine as both are different branches, but if there is a need to combine them into one destination, then it is needed to know about all these commits.

Git Merge & Rebase

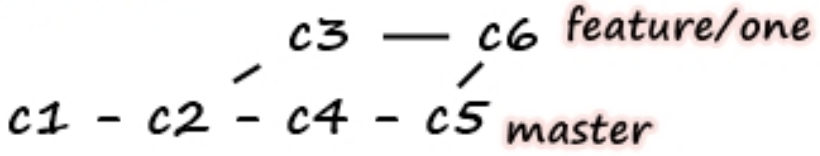
Now, suppose we want to combine the changes from the master branch to feature/one branch.

**Use the image in last slide as starting point for each of the below operation.

1. **Let's perform Merge –**

   git checkout feature/one

   git merge master



You are at your feature branch and performed 'git merge master' to merge the changes of master branch into your feature branch. So, in order to merge the changes a new merge commit 'c6' got created which binds histories from both the branches on feature branch.

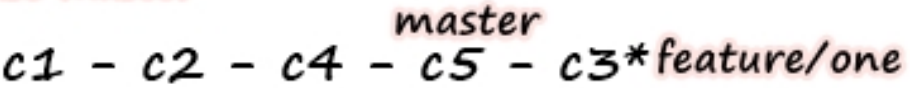
So if you now do a 'git log' you will see a chronological order of commits.

For e.g -> c1 - c2 - c3 - c4 - c5 - c6 (assuming c3 commit was done prior to c4 wrt time)

2. **Let's perform Rebase –**

   git checkout feature/one

   git rebase master



Rebase, as its name suggests, it actually resets the base. So earlier 'c2' was common between feature and master branch, we can call it the base.

Now, when we fire rebase command, it will actually pick all commits in feature after 'c2' which is just 'c3' commit. First it will apply all commits from master into the feature and on top it it applies the 'c3' commit. Why c3* & not c3?

Will explain it.

Git Merge with example - Part1

- We are on the master branch, and it has 2 commits.

```
\gitdemo Let us learn/g/git/merge_and__rebase (master)
$ git log --oneline
b161997 (HEAD -> master, origin/master) Message 2
a7cf16b Message 1
```

- Now, let's create a feature branch 'feature/test_merge' from master branch and switch to it. The commits are same as master branch.

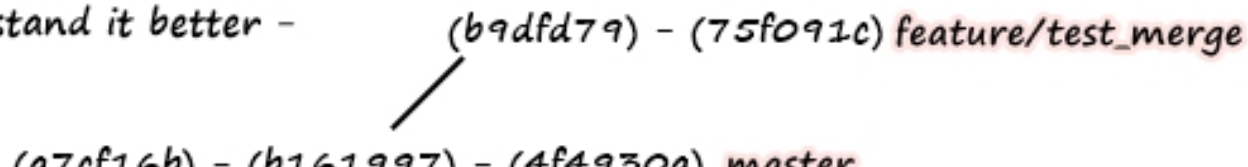
```
\gitdemo Let us learn/g/git/merge_and__rebase (feature/test_merge)
$ git log --oneline
b161997 (HEAD -> feature/test_merge, origin/master, origin/feature/test_merge, master) Message 2
a7cf16b Message 1
```

- Let's add 2 new commits to the feature branch and 1 new commit to the master.

```
\gitdemo Let us learn/g/git/merge_and__rebase (feature/test_merge)
$ git log --oneline
75f091c (HEAD -> feature/test_merge, origin/feature/test_merge) Merge Branch Message 2
b9dfd79 Merge Branch Message 1
b161997 (origin/master, master) Message 2
a7cf16b Message 1
```

```
\gitdemo Let us learn/g/git/merge_and__rebase (master)
$ git log --oneline
4f4930a (origin/master, master) Message 3
b161997 (HEAD -> master, origin/master) Message 2
a7cf16b Message 1
```

Let's draw an image which depicts the commits at the moment to understand it better -



- It's time to do the Merge.

```
\gitdemo Let us learn/g/git/merge_and__rebase (master)
$ git merge feature/test_merge
Updating 4f4930a..59d43fe
Fast-forward
 file2.txt | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 file2.txt
```

We are on the master branch and we merged feature into master.

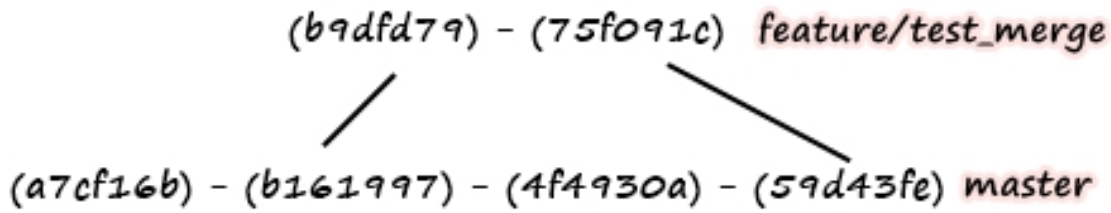
Git Merge with example - Part2

- As we merged the feature branch into the master. Master should now contain all the changes from feature. Btw, when you ran the merge command a text editor was opened for you to give the merge commit message.

```
Merge branch 'master' into feature/test_merge
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
```

- After the merge, history looks like -

```
\gitdemo Let us learn/g/git/merge_and__rebase (master)
$ git log --oneline
59d43fe (HEAD -> master, origin/feature/test_merge, feature/test_merge) Merge branch 'master' into feature/test_merge
4f4930a (origin/master) Message 3
75f091c Merge Branch Message 2
b9dfd79 Merge Branch Message 1
b161997 Message 2
a7cf16b Message 1
```



- So, this is how the merge operation is performed.
You can clearly notice one important thing here, that in the history every commit hash is consistent. In other words no change in the history took place, but the only change is addition of a new merge commit (59d43fe in the above scenario)

- I am sure you can imagine the simplicity in the overall operation while performing the merge. Looks like just a game of pointers.

Git Rebase with example - Part1

- Let's see how Rebase works and how it is different from Merge.

Again, our objective is same. We will create a feature out of master. We will be adding new changes/commits to both feature and master. And then we will do a rebase this time to combine changes from feature to the master branch.

```
\gitdemo Let us learn/g/git/merge__and_rebase (master)
$ git log --oneline
ae8deb9 (HEAD -> master, origin/master) Message 2
728f3f8 Message 1
```

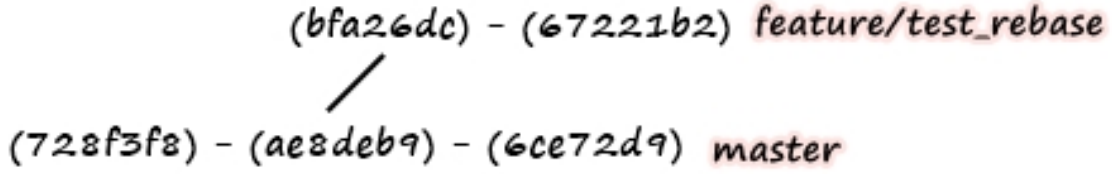
Both 'master' and 'feature/test_rebase' are in-sync at the moment.

```
\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase)
$ git log --oneline
ae8deb9 (HEAD -> feature/test_rebase, origin/master, origin/feature/test_rebase, master) Message 2
728f3f8 Message 1
```

- Now, let's add 2 new commits in the feature branch and 1 new commit in the master branch.

```
\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase)
$ git log --oneline
67221b2 (HEAD -> feature/test_rebase, origin/feature/test_rebase) Rebase branch Message 2
bfa26dc Rebase branch Message 1
ae8deb9 (origin/master, master) Message 2
728f3f8 Message 1
```

```
\gitdemo Let us learn/g/git/merge__and_rebase (master)
$ git log --oneline
6ce72d9 (HEAD -> master) Message 3
ae8deb9 (origin/master) Message 2
728f3f8 Message 1
```



Git Rebase with example - Part2

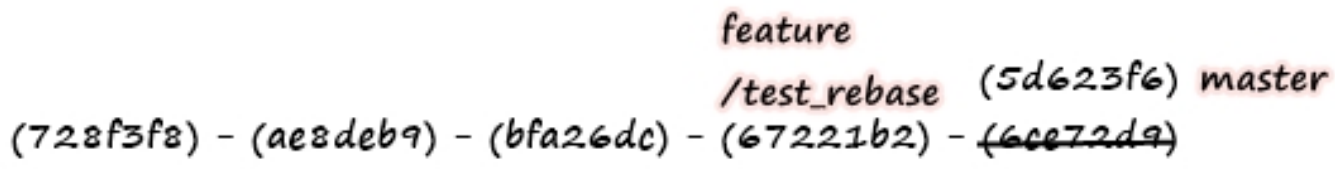
- It's time do perform Rebase.

```
\gitdemo Let us learn/g/git/merge__and_rebase (master)
$ git rebase feature/test_rebase
First, rewinding head to replay your work on top of it...
Applying: Message 3
```

- What exactly happened here?

1. The base has been reset, from the last base to a new base.
Earlier it was (ae8deb9) and now has been reset to (67221b2)
2. The new commits from master (which were not in feature) are applied on top.
Message 3 in this case.

```
\gitdemo Let us learn/g/git/merge__and_rebase (master)
$ git log --oneline
5d623f6 (HEAD -> master) Message 3
67221b2 (origin/feature/test_rebase, feature/test_rebase) Rebase branch Message 2
bfa26dc Rebase branch Message 1
ae8deb9 (origin/master) Message 2
728f3f8 Message 1
```



- Rebase operation is performed on the master branch. You can clearly see what happened. Firstly all the commits of the 'feature/test_rebase' are applied on the last common commit between master and feature (ae8deb9). And then on top of it the new commit from master is applied (Message 3). But wait, the commit hash looks different. Earlier it was 6ce72d9 and now it is 5d623f6. Why?

To Summarize -

It looks like both Rebase and Merge did there job well. And yes that's true.
In both the cases we got the feature branch changes combined into master branch.

But, one thing which is still not clear, why the commit hash changed in case of Rebase operation.
Yes, that's how Rebase is done. When you did the merge, no change was done wrt history (no commit hash changed), just a new merge commit was added.
But in case of Rebase, it actually rewrites the commit history. So, it is not basically a lift and shift of pointers but also rewiting a new commit of same changes.

We first get the new base and then on top of it we write again the changes as new commits.

And that's what makes Rebase a little risky and complicated operation if not performed carefully as it is rewriting the history.

It is always suggested to avoid doing Rebase in a branch/repo being used publicly or shared with other developers as it could create a mess. Use Merge in such scenarios.

You can use Rebase when your changes are on your feature and don't affect others. Like performing squash of commits etc.

Git pull vs Git pull --rebase

git pull or git pull --merge

Performs : git fetch + git merge

This is the default case.

In this case, your local changes are merged with the remote changes. A new merge commit is created pointing to the latest local commit and the latest remote commit.

git pull --rebase

Performs : git fetch + git rebase

In this case, your local changes are reapplied on top of the remote changes. So, the new changes on your local are picked, first the remote changes are applied which gives a new base and then your changes that were picked are applied on top as new commits.

Bonus command :P

git checkout -

To switch to the last branch you were at.

This command can be used to switch to and fro between the branches.

Resolving Conflicts

To generate a merge conflict - We created a text file 'conflict.txt' on both master and feature branch. Added 2 different lines in file on each branch. Now we are trying to merge master changes into the feature branch.

```
\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase)
$ git merge master
CONFLICT (add/add): Merge conflict in conflict.txt
Auto-merging conflict.txt
Automatic merge failed; fix conflicts and then commit the result.

\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase|MERGING)
$ cat conflict.txt
<<<<<<< HEAD
conflict message xyzzzz
conflict message abcdd
=======
conflict message 123
conflict message 1234444
>>>>>>> master
```

The moment we did 'git merge master' on feature, it gave us merge failed error and is expecting us to resolve the conflicts manually.
We checked the content of conflict.txt file.
The content between "<<<<<<<<< HEAD" & "=======" is from our current branch which is feature here. And the stuff between "======" & ">>>>>>> master" is from the branch which we are merging which is master here.

Suppose we want to keep the change from feature branch, hence we will remove what's coming from the master and add this file again.

```
\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase|MERGING)
$ cat conflict.txt
conflict message xyzzzz
conflict message abcdd

\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase|MERGING)
$ git add conflict.txt

\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase|MERGING)
$ git merge --continue
[feature/test_rebase 06d9ae0] Merge branch 'master' into feature/test_rebase

\gitdemo Let us learn/g/git/merge__and_rebase (feature/test_rebase)
$ git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 12 threads
```

git merge --continue

To continue the merge operation.

git merge --abort

To abort the merge process.

It will try to reconstruct the pre-merge state.

Thanks!!!

Git Stash

stash ~
to store safely in a hidden or secret place.

Let's understand with an actual scenario -

You are currently inside the local repository 'git-demo1' which is in-sync with the remote repository and you don't have any local changes.

```
RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git log
commit ade4aeecf159dce768454bf44eb415be327aa68b (HEAD -> master, origin/master)
Author:
Date:

    Adding my third commit

commit afafb9ee70247e0c1669881394b7632fbd9b8916
Author:
Date:

    Adding my second commit

commit dd506fcfa6f4e2a78b0c31c7d5decdc8c7f862c2
Author:
Date:

    Adding my first file
```

There is a text file 'test.txt' which we will be using for the demo. Printed the content of the file.

```
RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ ls
test.txt

RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ cat test.txt
This is my first line.
Adding a second line
Adding third line
```

Now, let's add new lines in the text file. Added 2 new lines.

```
             -RGO03A4 MINGW64 /g/git/git-demo1 (master)
$ echo "This is my fourth line" >> test.txt

             -RGO03A4 MINGW64 /g/git/git-demo1 (master)
$ echo "This is my fifth line" >> test.txt

             -RGO03A4 MINGW64 /g/git/git-demo1 (master)
$ cat test.txt
This is my first line.
Adding a second line
Adding third line
This is my fourth line
This is my fifth line
```

You can see the difference wrt HEAD using command 'git diff HEAD'. Green lines were added.

```
             -RGO03A4 MINGW64 /g/git/git-demo1 (master)
$ git diff HEAD
diff --git a/test.txt b/test.txt
index bfa3f7c..413af00 100644
--- a/test.txt
+++ b/test.txt
@@ -1,3 +1,5 @@
 This is my first line.
 Adding a second line
 Adding third line
+This is my fourth line
+This is my fifth line
```

Now due to some reason you don't want these new changes in the text file now, but you want to keep them stored somewhere locally as you might need them later.

You basically need a clean working directory at the moment.

'git stash' can help you in this scenario. The command saves your local modifications away and reverts the working directory to match the HEAD commit.

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

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:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

               RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash save "Stashed 2 lines of txt file"
Saved working directory and index state On master: Stashed 2 lines of txt file

              -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: On master: Stashed 2 lines of txt file
```

'git status' shows that the test.txt file has been modified.
But as we wanted to stash the changes we ran 'git stash save "message"' command and did 'git stash list' to check the stash and we can see that our change has been saved.

@mayank_ahuja

Now if you do 'git status' you will se clean working directory. The 2 lines that we added in the txt file are gone.

```
             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

```
$ cat test.txt
This is my first line.
Adding a second line
Adding third line
```

But what if you need the stashed changes back? You need those 2 lines back in the text file.
'git stash pop' command gives you the last stashed change.

```
$ git stash pop
On branch master
Your branch is up to date with 'origin/master'.

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:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (7b37ae0d60a7fe1ccdd2eeb27667f0b306899ed7)
```

```
$ cat test.txt
This is my first line.
Adding a second line
Adding third line
This is my fourth line
This is my fifth line
```

That's mainly how stashing works. Now let's learn some more commands wrt git stash.

'git stash list' - to know all stashed changes that are available/stored

```
$ git stash list
stash@{0}: On master: my 2nd stash
stash@{1}: On master: my first stash
```

'git stash save' is now deprecated and hence you can use 'git stash push' instead to stash a change.

git stash push -m "message"

```
             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ echo "This is my sixth line" >> test.txt

              RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash push -m "my 3rd stash"
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory
Saved working directory and index state On master: my 3rd stash

              RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: On master: my 3rd stash
stash@{1}: On master: my 2nd stash
stash@{2}: On master: my first stash
```

'git stash pop' vs 'git stash apply'

You can use either of them to re-apply the stashed change to the working directory, but the difference is, pop will remove the change from stash (stack) but when you do apply it's kept intact in the stack.

git stash pop

```
              -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: On master: my 3rd stash
stash@{1}: On master: my 2nd stash
stash@{2}: On master: my first stash

              -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash pop
On branch master
Your branch is up to date with 'origin/master'.

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:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (8859dd231aa7f18e7ee1a71c5136f12430eba744)

              -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: On master: my 2nd stash
stash@{1}: On master: my first stash
```

@mayank_ahuja

git stash apply

```
             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: On master: my 3rd stash
stash@{1}: On master: my 2nd stash
stash@{2}: On master: my first stash

             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash apply
On branch master
Your branch is up to date with 'origin/master'.

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:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: On master: my 3rd stash
stash@{1}: On master: my 2nd stash
stash@{2}: On master: my first stash
```

You can clearly see that even though the change was re-applied the entry is still present in the stash.

git stash apply <index>

```
             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: WIP on master: ade4aee Adding my third commit
stash@{1}: On master: my 3rd stash
stash@{2}: On master: my 2nd stash
stash@{3}: On master: my first stash

              RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash apply 2
On branch master
Your branch is up to date with 'origin/master'.

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:   test.txt
```

You can specify a specific index from the list to be applied.

git stash clear

```
             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list
stash@{0}: WIP on master: ade4aee Adding my third commit
stash@{1}: On master: my 3rd stash
stash@{2}: On master: my 2nd stash
stash@{3}: On master: my first stash

             -RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash clear

              RGOO3A4 MINGW64 /g/git/git-demo1 (master)
$ git stash list

              RGOO3A4 MINGW64 /g/git/git-demo1 (master)
```

Clears the stashed items.
List shows no entries.

@mayank_ahuja

git
reset / revert

Let's UNDO

Current state - The local repository is in-sync with the remote, we have 2 text files (demoOne & demoTwo) that already exsists on remote and our working directory is clean(no local changes at the moment).

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file

\gitdemo Let us learn/g/git/git-demo (master)
$ ls
demoOne.txt  demoTwo.txt

\gitdemo Let us learn/g/git/git-demo (master)
$ cat demoOne.txt
First line
Second line
Third line
Fourth line

\gitdemo Let us learn/g/git/git-demo (master)
$ cat demoTwo.txt
Valid line

\gitdemo Let us learn/g/git/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Let's add a new line in each file and create a new commit for each change.

```
\gitdemo Let us learn/g/git/git-demo (master)
$ echo "We need to revert this line" >> demoOne.txt

\gitdemo Let us learn/g/git/git-demo (master)
$ git add .

\gitdemo Let us learn/g/git/git-demo (master)
$ git commit -m "adding an invalid line to demoOne file"
[master 3ef3679] adding an invalid line to demoOne file
 1 file changed, 1 insertion(+)
```

```
\gitdemo Let us learn/g/git/git-demo (master)
$ echo "We need to revert this line" >> demoTwo.txt

\gitdemo Let us learn/g/git/git-demo (master)
$ git add .

\gitdemo Let us learn/g/git/git-demo (master)
$ git commit -m "adding an invalid line to demoTwo file"
```

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit f40d19d45ebfc0d75e0d186d4223465239d61fe9 (HEAD -> master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:51:49 2022 +0530

    adding an invalid line to demoTwo file

commit 3ef367977117172a4bcf3aa42d71e83c032d6d25
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:51:25 2022 +0530

    adding an invalid line to demoOne file

commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file
```

This is how the commits look like now

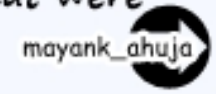
2 new commits added in our local repo.

We are actually now 2 commits ahead of the remote.

git reset --soft <commit id>

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git reset --soft ca54bc9c0486922e6683eaf8442d1e9dea85afb9

\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file

\gitdemo Let us learn/g/git/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   demoOne.txt
        modified:   demoTwo.txt
```

What actually happened?
As the name suggests, reset command is used to reset/undo the changes.
It actually moves the HEAD pointer. Earlier the HEAD was pointing to some 'f40d1..' commit, but now we actually told git to reset the pointer to commit 'ca54bc..'. It actually removed all the commits after 'ca54bc..' ,
in our case 2 commits. 'Soft' flag tells git to not delete the changes that were part of the removed commits and keep them staged(green).



As we saw that after the 'soft' reset, the changes were Staged.
Now let's create a new commit, it will commit the 2 files that were reverted in the soft reset.
Added those 2 files as part of new commit '85221..'. HEAD pointer updated.

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git commit -m "adding back the 2 reverted files"
[master 852215e] adding back the 2 reverted files
 2 files changed, 2 insertions(+)

\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit 852215eedecd486eb6ec897fb2f6a1b1c4c9975c (HEAD -> master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 22:13:38 2022 +0530

    adding back the 2 reverted files

commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file
```

- You can either pass a commit id to perform reset operation, or you can also give the reference wrt HEAD (e.g HEAD~1 -> reset to 1 commit after current HEAD, HEAD~2 -> 2 commits after HEAD etc.)
- When you perform reset, it always undo all commits after specified commit, and not just that commit or a specific range.

git reset --mixed <commit from HEAD>

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git reset --mixed HEAD~1
Unstaged changes after reset:
M       demoOne.txt
M       demoTwo.txt

\gitdemo Let us learn/g/git/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

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:   demoOne.txt
        modified:   demoTwo.txt
```

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file
```

What actually happened?

When you pass the flag 'mixed' (which is default), it behaves similar to 'soft' in the sense that the changes are not lost from the local but are unstaged. So, you need to add them again if you need them back before doing commit or may be stash them. Here we moved 1 commit ahead from prev HEAD. Or in other words removed all commits after HEAD-1

git reset --hard <commit from HEAD>

As in the last change we did soft reset, let's add back the unstaged files and also commit them. Added a new commit '8b2ea..'. HEAD pointer updated.

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git add .
```

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git commit -m "adding back again the 2 reverted files"
```

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit 8b2ea3baf12bc627b46a927e732d824d55acc298 (HEAD -> master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 22:14:30 2022 +0530

    adding back again the 2 reverted files

commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file
```

Now, when we use the flag 'hard' in the reset command. It resets the commit but also removes all changes from local which where part of those commit(s).

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git reset --hard HEAD~1
HEAD is now at ca54bc9 adding demoTwo file

\gitdemo Let us learn/g/git/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Working directory is clean.

git revert <commit id>

This is the current state, we are in sync with the remote repo.

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file
```

Now let's see what revert does.

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git revert ca54bc9c0486922e6683eaf8442d1e9dea85afb9
[master 5787afb] This is a Revert command demo ~ Reverting "adding demoTwo file"
 1 file changed, 1 deletion(-)
 delete mode 100644 demoTwo.txt
```

git revert command sounds similar to reset, and yes both are like performing the undo operation, but reset was much cleaner that revert.

Why so? When you perfrom the revert operation, it will undo the commit (or range of commits) but it also adds extra commits to perform this revert.

So, basically git history would have the commit that had the change plus a commit which was used to revert the change.

Let's see that in the next slide, what happened when we fired the revert.

When we fired the revert command in prev slide, it actually asked to add a commit message for the revert we are going to perform.

```
This is a Revert command demo ~ Reverting "adding demoTwo file"

This reverts commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is up to date with 'origin/master'.
#
# Changes to be committed:
#       deleted:    demoTwo.txt
#
~
~
~
~
~
~
```

With revert we can actually undo any specific commit or range of commits. With reset it was resetting everything after the specified commit. We can't specify a range there.

```
\gitdemo Let us learn/g/git/git-demo (master)
$ git log
commit 5787afbe43584d7e583b5689649c1cf1acfe526d (HEAD -> master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 22:15:33 2022 +0530

    This is a Revert command demo ~ Reverting "adding demoTwo file"

    This reverts commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9.

commit ca54bc9c0486922e6683eaf8442d1e9dea85afb9 (origin/master)
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:58 2022 +0530

    adding demoTwo file

commit 38630815df479ac0443ee344706df54e354b4885
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:39:09 2022 +0530

    adding few lines to demoOne file

commit a28a1efde08ac886d7160bea3b085884025b7a1d
Author: mydemo222 <email@example.com>
Date:   Tue Apr 26 21:38:20 2022 +0530

    adding demoOne file
```

A new revert commit has been added.

Now, do a 'git push' if you want this change to appear on the remote as well

So, to summarize -

'git revert' - does the undo operation but it creates a new commit that undoes the changes from a previous commit or range of commits.
So, it doesn't modify existing history. No existing commit is removed.
It just adds new commits. Sounds 'safe' as history is not rewritten.

'git reset' - it also does the undo without adding any new commit.
But when we do reset we are actually moving the pointer based on commit we want to reset to, so yes the existing history will get changed.
It is little complicated and can be risky if not used carefully.

Extras -

to revert range of commits
`git revert --no-commit HEAD~3..HEAD`
(This will actually revert the 3 commits from HEAD)
(You can also use --no-commit option with revert. The --no-commit option tells git to do the revert, but do not commit it automatically. You can perform commit later using 'git commit').

That's it. Thank you.

Git Commands Explained

Let's get started

mayank_ahuja

git pull is git fetch + git merge -> Part 1

- Assume the local branch is in-sync with the remote branch at the moment. Both 'master' (local copy) and 'origin/master' (pointer to remote master) are pointing to the same commit '54ad..'.

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit 54ad99a28a346989955ea2a192baefe116bd126f (HEAD -> master, origin/master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:38:50 2022 +0530

    Create test.txt
```

- Now, suppose one of your teammate has added/pushed a new file on master remote branch. You local master is not aware of that change.
Let's do 'git fetch' and see what happens.

git fetch

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:mydemo222/git-commands
   54ad99a..b5dcb06  master     -> origin/master
```

What happened? Let's do 'git log' and see :

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit 54ad99a28a346989955ea2a192baefe116bd126f (HEAD -> master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:38:50 2022 +0530

    Create test.txt
```

Oh! I can only see 'master' pointing to '54ad..', but where is 'origin/master' ?
When you did 'git fetch' the origin/master got updated to what's in the remote master. As your teammate added a new change, it must be pointing to it.

git pull is git fetch + git merge -> Part 2

- Let's find out where is 'origin/master' git log origin/master

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log origin/master
commit b5dcb0625534c04c4a8e4c807b1beedf8ea1d030 (origin/master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:41:45 2022 +0530

    Create hello.txt

commit 54ad99a28a346989955ea2a192baefe116bd126f (HEAD -> master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:38:50 2022 +0530

    Create test.txt
```

- See, the origin/master points to the latest commit which is on remote master, but your local master still points to '54ad..' old commit.

How can we sync them again?

we need to do a merge now. git merge

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git merge
Updating 54ad99a..b5dcb06
Fast-forward
 hello.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hello.txt
```

Awesome, both origin/master and master are now pointing to the same commit.

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit b5dcb0625534c04c4a8e4c807b1beedf8ea1d030 (HEAD -> master, origin/master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:41:45 2022 +0530

    Create hello.txt

commit 54ad99a28a346989955ea2a192baefe116bd126f
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:38:50 2022 +0530

    Create test.txt
```

git pull is git fetch + git merge -> Part 3

- We can perform the complete 'git fetch' + 'get merge' thing by just doing 'git pull'.

Assume, that your teammate again added a new text file on remote master and you local master is unaware of the change.

Current state - 'git log'

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit b5dcb0625534c04c4a8e4c807b1beedf8ea1d030 (HEAD -> master, origin/master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:41:45 2022 +0530

    Create hello.txt

commit 54ad99a28a346989955ea2a192baefe116bd126f
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:38:50 2022 +0530

    Create test.txt
```

- Let's do 'git pull' and see what happens : git pull

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:mydemo222/git-commands
   b5dcb06..e76592e  master     -> origin/master
Updating b5dcb06..e76592e
Fast-forward
 pull.txt | 1 +
```

When we did 'git pull', it first performed 'git fetch' followed by 'git merge'

'origin/master' & 'master' pointing to same new commit.

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit e76592e5404ccdcaa28e9961a8a9d12ed5b3ef95 (HEAD -> master, origin/master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:42:50 2022 +0530

    Create pull.txt

commit b5dcb0625534c04c4a8e4c807b1beedf8ea1d030
Author: Demo User <mak.alex09@gmail.com>
```

Creating and Publishing a Local branch to Remote

- create a local branch 'feature/demo1' and switch to newly created branch.

`git checkout -b feature/demo1` OR `git branch feature/demo1` `git checkout feature/demo1`

```
\gitdemo Let us learn/g/git/git-commands (feature/demo)
$ git checkout -b feature/demo1
Switched to a new branch 'feature/demo1'
```

- remote is unaware of this new branch, let's publish it to the remote

```
\gitdemo Let us learn/g/git/git-commands (feature/demo1)
$ git pull
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
```

`git push -u origin feature/demo1`

```
\gitdemo Let us learn/g/git/git-commands (feature/demo1)
$ git push -u origin feature/demo1
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'feature/demo1' on GitHub by visiting:
remote:      https://github.com/mydemo222/git-commands/pull/new/feature/demo1
remote:
To github.com:mydemo222/git-commands.git
 * [new branch]      feature/demo1 -> feature/demo1
Branch 'feature/demo1' set up to track remote branch 'feature/demo1' from 'origin'.
```

- now if I do 'git pull', your remote is aware of this branch

```
\gitdemo Let us learn/g/git/git-commands (feature/demo1)
$ git pull
Already up to date.
```

Exploring git log command to view commit logs

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit 88ea3093a6e932d1d57a3fdfe7e781ed6e13b6d1 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:43:06 2022 +0530

    Third commit

commit 9b4ecd547212dba91968120b3239b9b05d6d959c
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:42 2022 +0530

    Second commit

commit 0baba06ee243303430e374a87fe4364a0445f971
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:21 2022 +0530

    First commit
```

- viewing last n commits (in this case 2 commits)

git log -n2

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log -n2
commit 88ea3093a6e932d1d57a3fdfe7e781ed6e13b6d1 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:43:06 2022 +0530

    Third commit

commit 9b4ecd547212dba91968120b3239b9b05d6d959c
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:42 2022 +0530

    Second commit
```

- one liner info of every commit

git log --oneline

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log --oneline
88ea309 (HEAD -> master, origin/master) Third commit
9b4ecd5 Second commit
0baba06 First commit
```

Exploring git log command to view commit logs

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit 88ea3093a6e932d1d57a3fdfe7e781ed6e13b6d1 (HEAD -> master, origin/master)
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:43:06 2022 +0530

    Third commit

commit 9b4ecd547212dba91968120b3239b9b05d6d959c
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:42 2022 +0530

    Second commit

commit 0baba06ee243303430e374a87fe4364a0445f971
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:21 2022 +0530

    First commit
```

- skipping last n commits (in this case 2 commits)

  git log --skip=2

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log --skip=2
commit 0baba06ee243303430e374a87fe4364a0445f971
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:21 2022 +0530

    First commit
```

- grep commits by string/pattern

  git log --grep="Second"

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git log --grep="Second"
commit 9b4ecd547212dba91968120b3239b9b05d6d959c
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 20:42:42 2022 +0530

    Second commit
```

git cherry-pick is cool

- As the name sounds, it is used to pick changes from existing commit and apply them by creating a new commit.

Let's see a simple example.

We are on 'feature/demo' branch currently and we have added a new change with commit id '32904...'- Created hello file

```
\gitdemo Let us learn/g/git/git-commands (feature/demo)
$ git log
commit 32904343f494309dcc6b3dd8b2fb41f9b3d6c6c7 (HEAD -> feature/demo, origin/feature/demo)
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 22:11:24 2022 +0530

    Created hello file

commit e76592e5404ccdcaa28e9961a8a9d12ed5b3ef95 (origin/master, master)
Author: Demo User <mak.alex09@gmail.com>
Date:   Thu Apr 28 21:42:50 2022 +0530

    Create pull.txt
```

- We can see the 'master' is behind by 1 commit from the feature.

We know one way to updated 'master' is by doing merge with the feature branch.

But here, we will see how cherry-pick works. We can pick a commit and apply.

So let's pick the commit '32904..' and apply it to master.

Let's switch to master and do cherry pick.

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git cherry-pick 32904343f494309dcc6b3dd8b2fb41f9b3d6c6c7
[master b2c0728] Created hello file
 Date: Thu Apr 28 22:11:24 2022 +0530
 1 file changed, 1 insertion(+)

\gitdemo Let us learn/g/git/git-commands (master)
$ git log
commit b2c072893f36151e1bf422c308f46d36dfb934bc (HEAD -> master)
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 22:11:24 2022 +0530

    Created hello file
```

After cherry-pick the same changes from that commit are applied to master branch. But it created a new commit with a different sha1.

Let's see the difference

- difference between 2 commits git diff commi1 commit2

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git diff 54ad99a28a346989955ea2a192baefe116bd126f a7e102b00a994cce983109bd2f6e5ca81dba4d38
diff --git a/hello.txt b/hello.txt
new file mode 100644
index 0000000..c6ba641
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1,5 @@
+Helloooo
+Line 2 ..............
+Line 3 ..............
+Line 4 ..............
+Line 5 ..............
diff --git a/pull.txt b/pull.txt
new file mode 100644
index 0000000..7bee91d
--- /dev/null
+++ b/pull.txt
@@ -0,0 +1 @@
+Pull file
```

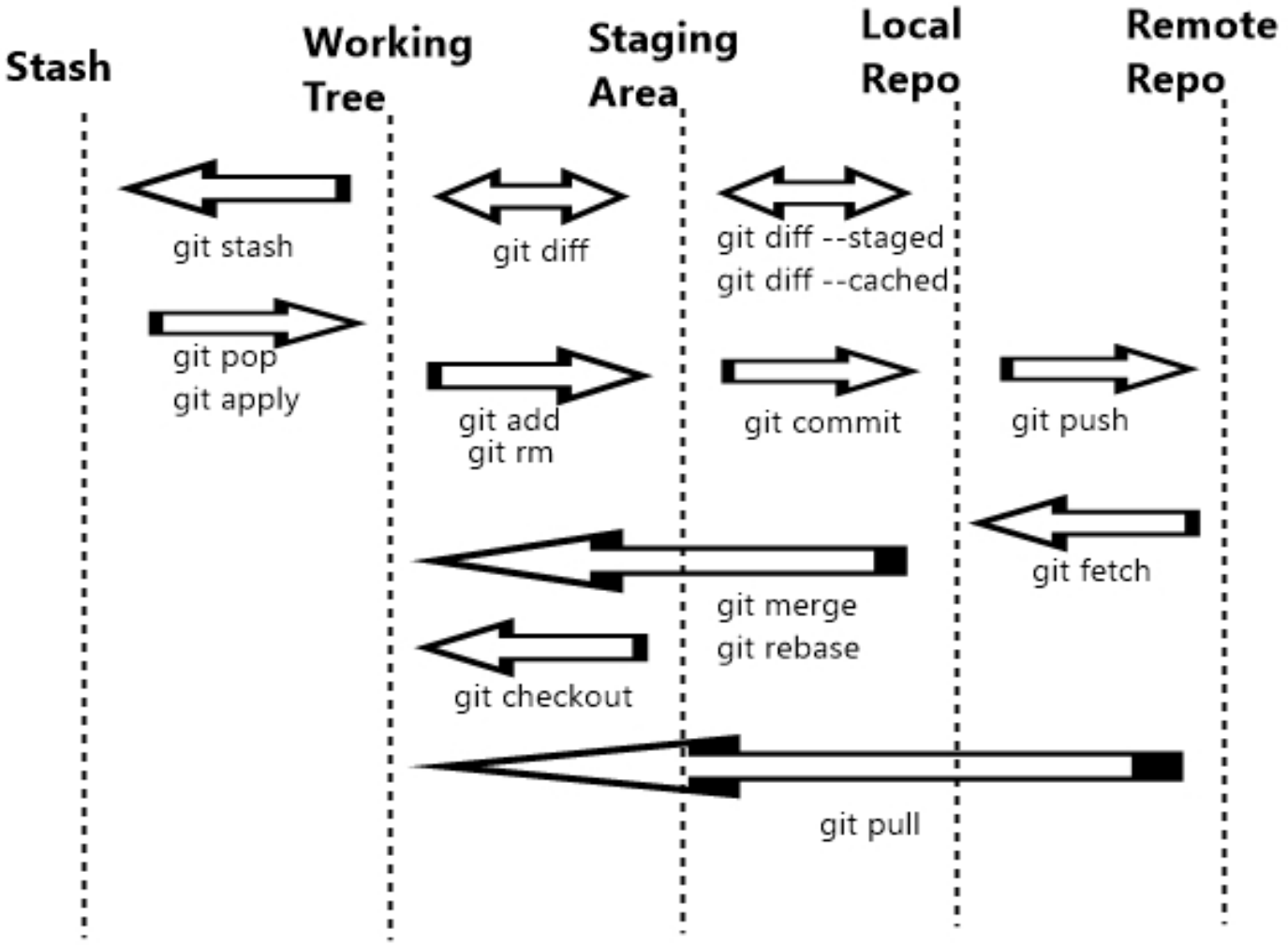
a/... from commit1 '54ad...' & b/... is from commit2 'a7e10..'. You can see 2 files (hello.txt and pull.txt) were added in commit2.

- similarly you can see diff of tip between two branches
  git diff branchA branchB
- to see difference between your local (unstaged) vs staged(indexed) changes
  git diff
- to see difference between staged(indexed) changes vs what's commited your local repo git diff --staged
  OR
  git diff --cached

- to see changes in a commit
  git show <commit>

```
\gitdemo Let us learn/g/git/git-commands (master)
$ git show a7e102b00a994cce983109bd2f6e5ca81dba4d38
commit a7e102b00a994cce983109bd2f6e5ca81dba4d38 (HEAD -> master)
Author: mydemo222 <email@example.com>
Date:   Thu Apr 28 22:50:13 2022 +0530

    Adding new lines

diff --git a/hello.txt b/hello.txt
index 44f954a..c6ba641 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1,2 +1,5 @@
-Hello File
 Helloooo
+Line 2 ..............
+Line 3 ..............
+Line 4 ..............
+Line 5 ..............
```

In a nutshell



mayank_ahuja