

Subject Name

Source Code Management

Subject Code

CS81

Cluster:

Alpha

Department

: DSE

CHITKARA
UNIVERSITY



Submitted By:

Snaurya Unawan

2110991305

G-17

Submitted To:

Dr. Vikas Lamba

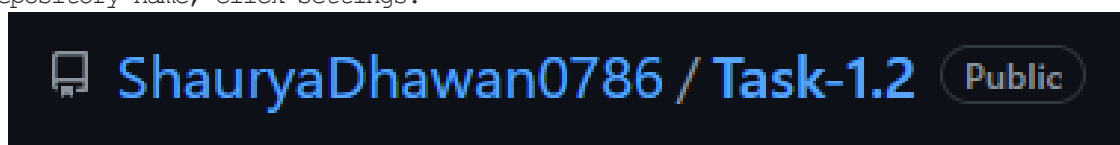
Task 1.2

Goals

S.No.	LIST OF FEATURES	Page
1.2.1	Add collaborators on Github repo	1-3
1.2.2	Fork and commit	4-8
1.2.3	Merge and Resolve conflicts created due to own activity and collaborators activity	9-11
1.2.4	Reset and Revert	12-22

1.2.1 Add Collaborators on Github

1. Ask for the username of the person you're inviting as a collaborator.
2. On GitHub.com, navigate to the main page of the repository. Under your repository name, click Settings.



3. In the "Access" section of the sidebar, click Collaborators & teams.

Manage access



You haven't invited any collaborators yet

Add people

5. In the search field, start typing the name of person you want to invite, then click a name in the list of matches.
6. Click **Add NAME to REPOSITORY**.



Add a collaborator to Task-1.2



Aryan Shubbu
LoneExpert





Add LoneExpert to this repository

Manage access

☐ Select all

Find a collaborator...

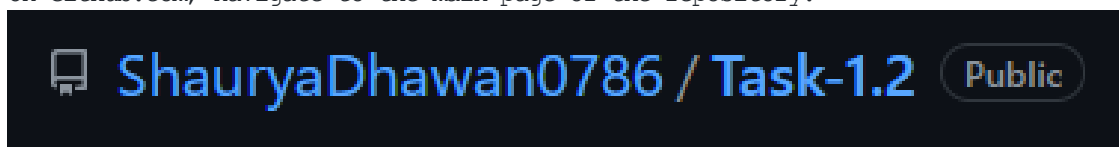
☐  **Aryan Shubbu**
Awaiting LoneExpert's response

Pending Invite 

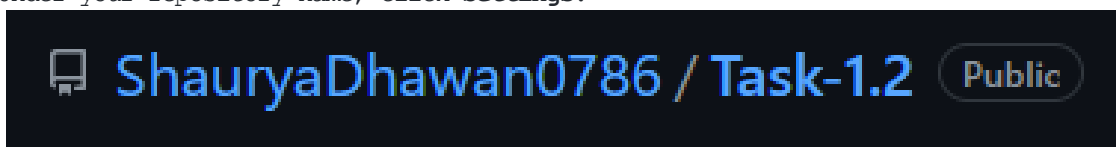
- The user will receive an email inviting them to the repository. Once they accept your invitation, they will have collaborator access to your repository.

Removing collaborator permissions from a person contributing to a repository

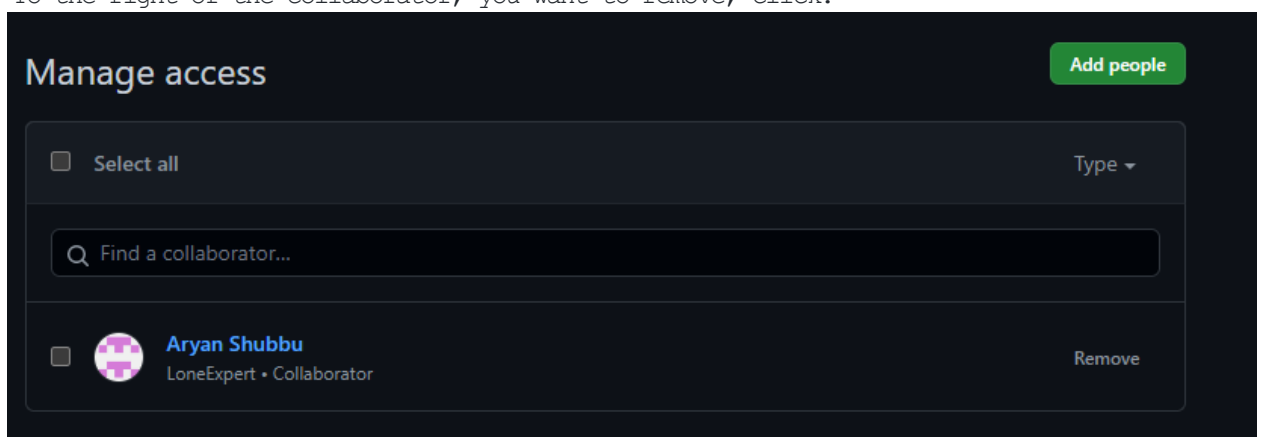
- On GitHub.com, navigate to the main page of the repository.



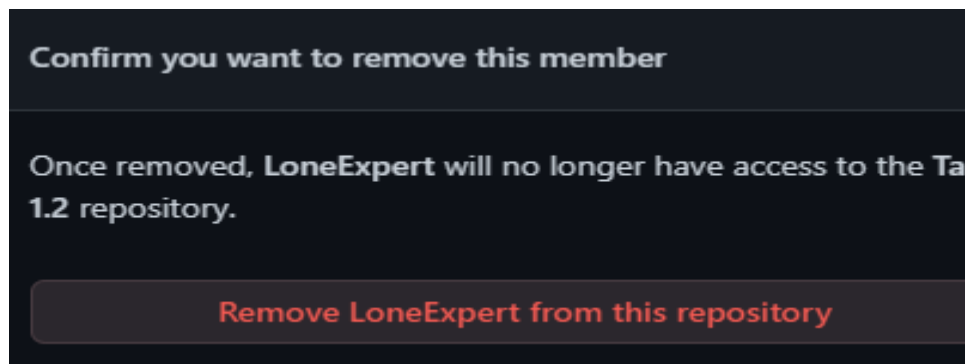
- Under your repository name, click **Settings**.



- In the "Access" section of the sidebar, click **Collaborators & teams**.
- To the right of the collaborator, you want to remove, click.



- Now confirm if you want to remove the collaborator from this repository

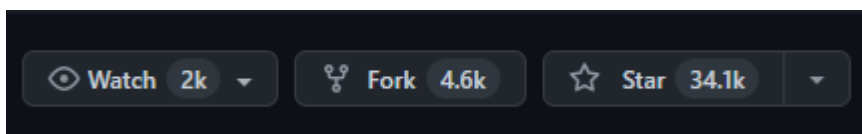


After using GitHub by yourself for a while, you may find yourself wanting to contribute to someone else's project. Or maybe you'd like to use someone's project as the starting point for your own. This process is known as forking.

Creating a "fork" is producing a personal copy of someone else's project. Forks act as a sort of bridge between the original repository and your personal copy. You can submit pull requests to help make other people's projects better by offering your changes up to the original project. Forking is at the core of social coding at GitHub.

Forking a repository

1. Navigate to the project which you want to fork.
2. Example we want to fork https://github.com/Sajalsharma1001/First_Repository.git.



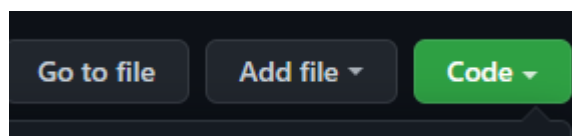
3. Click **Fork**.
4. GitHub will take you to your copy (your fork) of the ``First_Repository``.

Cloning a fork

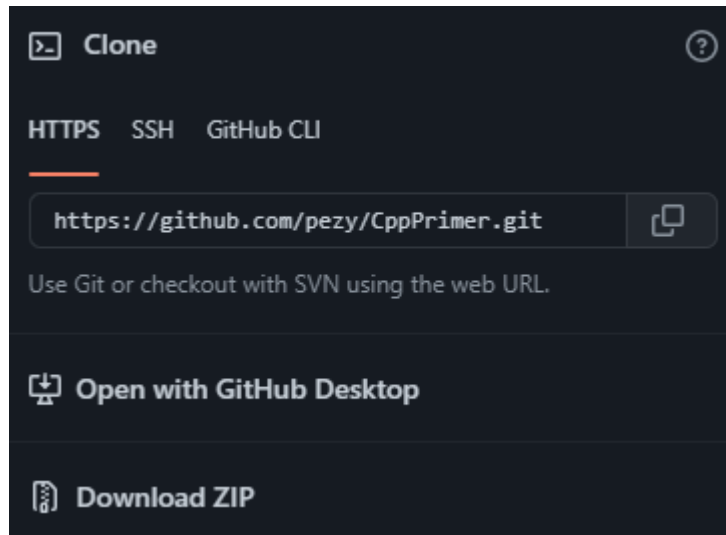
You've successfully forked the ``First_Repository`` repository, but so far, it only exists on GitHub. To be able to work on the project, you will need to clone it to your computer.

You can clone your fork with the command line, GitHub CLI, or GitHub Desktop.

1. On GitHub, navigate to **your fork** of the ``First_Repository`` repository.
2. Above the list of files, click **Code**.



3. To clone the repository using HTTPS, under "Clone with HTTPS", click. To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click. To clone a repository using GitHub CLI, click **Use GitHub CLI**, then click.



4. Open Git Bash.
5. Change the current working directory to the location where you want the cloned directory.
6. Type `git clone`, and then paste the URL you copied earlier.
7. Press **Enter**. Your local clone will be created.

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git clone https://github.com/ShauryaDhawan0786/CppPrimer.git
Cloning into 'CppPrimer'...
remote: Enumerating objects: 4935, done.
remote: Total 4935 (delta 0), reused 0 (delta 0), pack-reused 4935
Receiving objects: 100% (4935/4935), 1.29 MiB | 957.00 KiB/s, done.
Resolving deltas: 100% (2730/2730), done.
Updating files: 100% (546/546), done.
```

Making and pushing changes

Go ahead and make a few changes to the project.

When you're ready to submit your changes, stage and commit your changes. tells Git that you want to include all of your changes in the next commit. `git commit` takes a snapshot of those changes.

- o `git add -A`
- o `git commit -m "a short description of the change"`

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT/CppPrimer (master)
$ touch new.txt

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT/CppPrimer (master)
$ git add --a

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT/CppPrimer (master)
$ git commit -m "file added"
[master 267c7dd] file added
1 file changed, 1 insertion(+)
create mode 100644 new.txt
```

When you stage and commit files, you essentially tell Git, "Okay, take a snapshot of my changes!" You can continue to make more changes, and take more commit snapshots.

Right now, your changes only exist locally. When you're ready to push your changes up to GitHub, push your changes to the remote.

 git push

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT/CppPrimer (master)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 277 bytes | 92.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/ShauryaDhawan0786/CppPrimer.git
9aaadff..267c7dd master -> master
```

Making a pull request

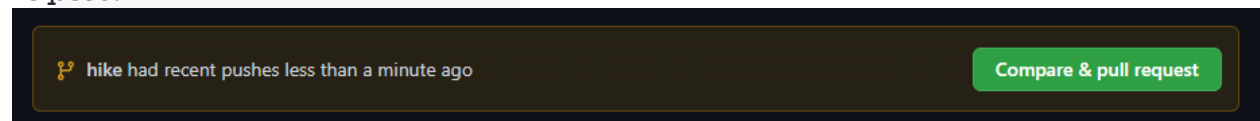
At last, you're ready to propose changes into the main project! This is the final step in producing a fork of someone else's project, and arguably the most important. If you've made a change that you feel would benefit the community as a whole, you should definitely consider contributing back.

To do so, head on over to the repository on GitHub where your project lives.

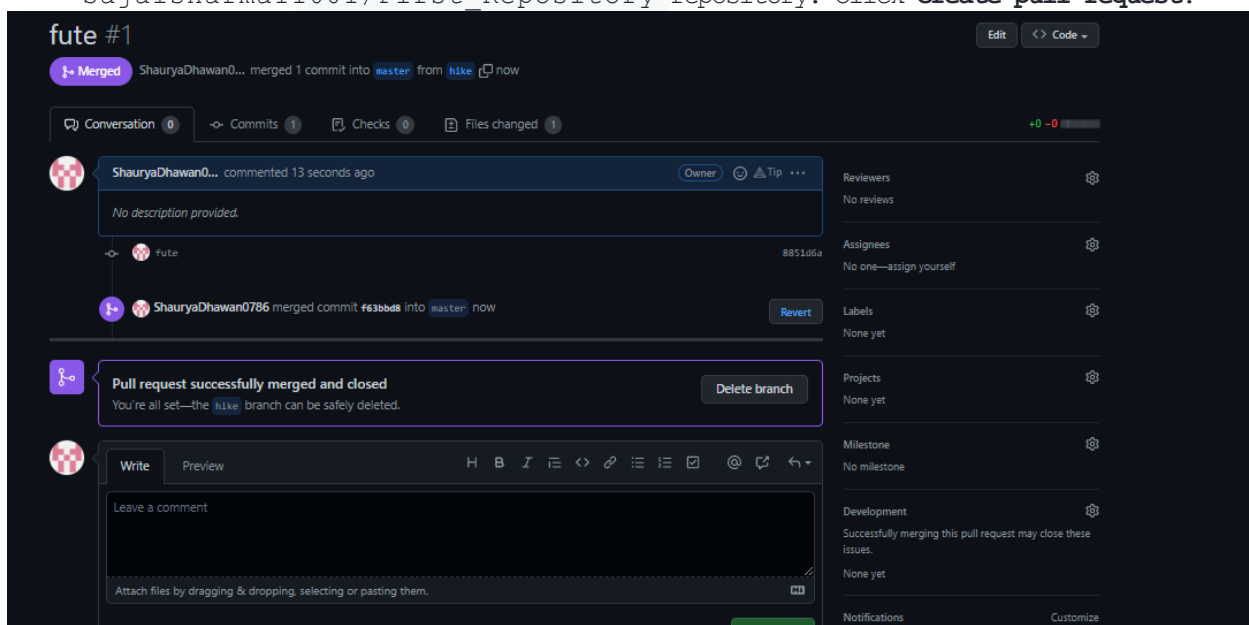
For example,

It would be at https://github.com/satyum/First_Repository.

You'll see a banner indicating that your branch is one commit ahead of Samarth1248/First_Repository:master. Click **Contribute** and then **Open a pull request**.

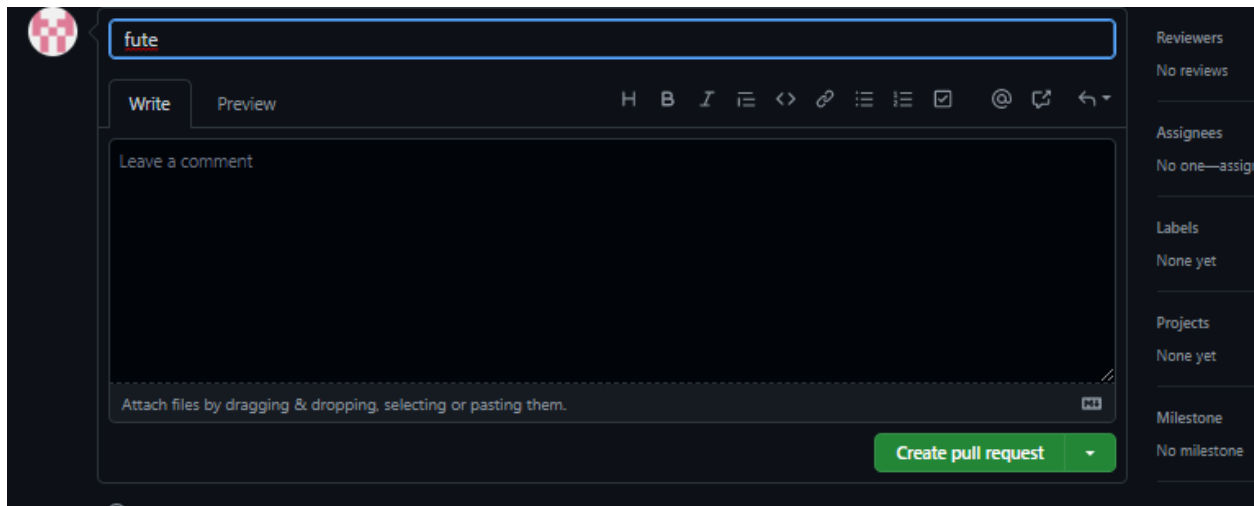


GitHub will bring you to a page that shows the differences between your fork and the Sajalsharma11001/First_Repository repository. Click **Create pull request**.



GitHub will bring you to a page where you can enter a title and a description of your changes. It's important to provide as much useful information and a rationale for why you're making this pull request in the first place. The project owner needs to be able

to determine whether your change is as useful to everyone as you think it is. Finally, click **Create pull request**.



Managing feedback

Pull Requests are an area for discussion. For other projects, don't be offended if the project owner rejects your pull request, or asks for more information on why it's been made. It may even be that the project owner chooses not to merge your pull request, and that's totally okay. Your copy will exist in infamy on the Internet. And who knows--maybe someone you've never met will find your changes much more valuable than the original project.

Finding projects

You've successfully forked and contributed back to a repository.

1.2.3 Merge and Resolve conflicts created due to own activity and collaborators activity

Collaborating with a trusted colleague without conflicts

We start by enabling collaboration with a trusted colleague. We will designate the Owner as the person who owns the shared repository, and the Collaborator as the person that they wish to grant the ability to make changes to their repository. We start by giving that person access to our GitHub repository.

Step 1: Collaborator clone

Step 2: Collaborator Edits

Step 3: Collaborator commit and push

Step 4: Owner pull

Step 5: Owner edits, commit, and push

Step 6: Collaborator pull



Merge conflicts

A merge conflict occurs when both the owner and collaborator change the same lines in the same file without first pulling the changes that the other has made. This is most easily avoided by good communication about who is working on various sections of each file, and trying to avoid overlaps. But sometimes it happens, and *git* is there to warn you about potential problems. And *git* will not allow you to overwrite one person's changes to a file with another's changes to the same file if they were based on the same version.

The main problem with merge conflicts is that, when the Owner and Collaborator both make changes to the same line of a file, *git* doesn't know whose changes take precedence. You have to tell *git* whose changes to use for that line.

5.5 How to resolve a conflict

o Abort

Sometimes you just made a mistake. When you get a merge conflict, the repository is placed in a 'Merging' state until you resolve it. There's a command line command to abort doing the merge altogether:

SYNTAX:

```
git merge --abort
```

Of course, after doing that you still haven't synced with your collaborator's changes, so things are still unresolved. But at least your repository is now usable on your local machine.

o Checkout

The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the command line *git* program to tell *git* to use either *your* changes (the person doing the merge), or *their* changes (the other collaborator).

- o keep your collaborators file: `git checkout --theirs conflicted_file.Rmd`
- o keep your own file: `git checkout --ours conflicted_file.Rmd`

Once you have run that command, then run `add`, `commit`, and `push` the changes as normal.

o Pull and edit the file

Steps:

1. Owner and collaborator ensure all changes are updated
2. Owner makes a change and commits
3. Collaborator makes a change and commits on the same line
4. Collaborator pushes the file to GitHub

5. Owner pushes their changes and gets an error

6. Owner pulls from GitHub to get Collaborator changes

7. Owner edits the file to resolve the conflict

8. Owner commits the resolved changes

9. Owner pushes the resolved changes to GitHub

10. Collaborator pulls the resolved changes from GitHub

11. Both can view commit history

- o Workflows to avoid merge conflicts

Some basic rules of thumb can avoid the vast majority of merge conflicts, saving a lot of time and frustration. These are wording our teams live by:

- o Communicate often
- o Tell each other what you are working on
- o Pull immediately before you commit or push
- o Commit often in small chunks.

A good workflow is encapsulated as follows:

```
Pull -> Edit -> Add -> Pull -> Commit -> Push
```

Always start your working sessions with a pull to get any outstanding changes, then start doing your editing and work. Stage your changes, but before you commit, pull again to see if any new changes have arrived. If so, they should merge in easily if you are working in different parts of the program. You can then Commit and immediately Push your changes safely.

12.4 ~~RESET AND REVERT~~

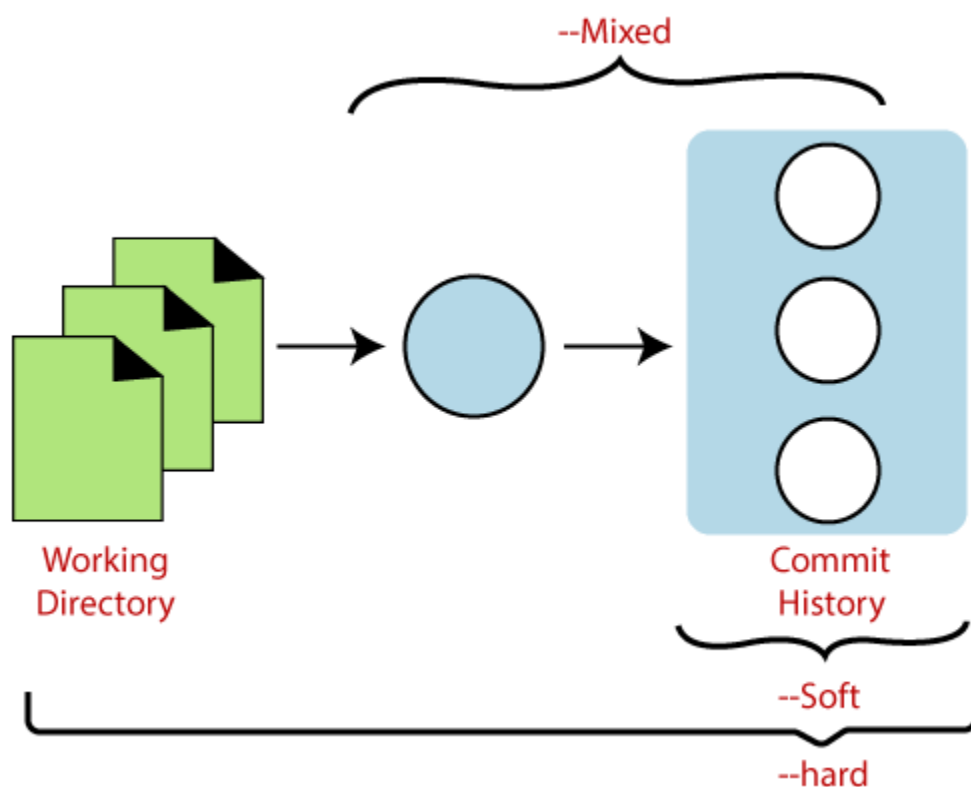


The term reset stands for undoing changes. The git reset command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.

- o **Soft**
- o **Mixed**
- o **Hard**

If we say in terms of Git, then Git is a tool that resets the current state of HEAD to a specified state. It is a sophisticated and versatile tool for undoing changes. It acts as a **time machine for Git**. You can jump up and forth between the various commits. Each of these reset variations affects specific trees that git uses to handle your file in its content.

Additionally, git reset can operate on whole commits objects or at an individual file level. Each of these reset variations affects specific trees that git uses to handle your file and its contents.



Git uses an index (staging area), HEAD, and working directory for creating and reverting commits.

The working directory lets you change the file, and you can stage into the index. The staging area enables you to select what you want to put into your next commit. A commit object is a cryptographically hashed version of the content. It has some Metadata and points which are used to switch on the previous commits.

Git Reset to Commit

Sometimes we need to reset a particular commit; Git allows us to do so. We can reset to a particular commit. To reset it, git reset command can be used with any option

supported by reset command. It will take the default behavior of a particular command and reset the given commit. The syntax for resetting commit is given below:

```
1. $ git reset <option> <commit-sha>
```

These options can be

- o --soft
- o --mixed o
- Hard

Let's understand the different uses of the git reset command.

Git ~~Reset~~ Hard

It will first move the Head and update the index with the contents of the commits. It is the most direct, unsafe, and frequently used option. The -hard option changes the Commit History, and ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory need to reset to match that of the specified commit. Any previously pending commits to the Staging Index and the Working Directory get reset to match Commit Tree. It means any awaiting work will be lost.

Let's understand the --hard option with an example. Suppose I have added a new file to my existing repository. To add a new file to the repository, run the below command:

```
1. $ git add <file name>
```

To check the status of the repository, run the below command:

```
2. $ git status
```

To check the status of the Head and previous commits, run the below command:

```
3. $ git log
```

Consider the below image:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ touch future.txt

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git add --a

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git commit -m "gang"
[master a5e1649] gang
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 future.txt

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git log --oneline
a5e1649 (HEAD -> master) gang
fb3d899 figh
205361a Revert "fish added"
09dda3d dog added
e5869c4 fish added
b6710b4 ka1 added
c606358 naya added
a791c74 merge solved
1b1e773 done
d4604ce (super) updated
5632314 file added
5f4dc60 file added
```

In the above output, I have added a file named **abc.txt**. I have checked the status of the repository. We can see that the current head position yet not changed because I have not committed the changes. Now, I am going to perform the **reset --hard** option. The git reset hard command will be performed as:

4. \$ git reset --hard

Consider the below output:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git reset b6710b4 --hard
HEAD is now at b6710b4 ka1 added
```

As you can see in the above output, the **-hard** option is operated on the available repository. This option will reset the changes and match the position of the Head before the last changes. It will remove the available changes from the staging area. Consider the below output:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git status
On branch master
nothing to commit, working tree clean
```

The above output is displaying the status of the repository after the hard reset. We can see there is nothing to commit in my repository because all the changes removed by the

reset hard option to match the status of the current Head with the previous one. So, the file **abc.txt** has been removed from the repository.

Generally, the reset hard mode performs below operations:

- o It will move the HEAD pointer.
- o It will update the staging Area with the content that the HEAD is pointing.
- o It will update the working directory to match the Staging Area.

Git Reset Mixed

A mixed option is a default option of the git reset command. If we would not pass any argument, then the git reset command considered as **-mixed** as default option. A mixed option updates the ref pointers. The staging area also reset to the state of a specified commit. The undone changes transferred to the working directory. Let's understand it with an example.

Let's create a new file say **nw.txt**. Check the status of the repository. To check the status of the repository, run the below command:

1. `$ git status`

It will display the untracked file from the staging area. Add it to the index. To add a file into stage index, run the git add command as:

2. `$ git add <filename>`

The above command will add the file to the staging index. Consider the below output:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ touch like.txt

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git add --a

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   like.txt
```

In the above output, I have added a **newfile2.txt** to my local repository. Now, we will perform the reset mixed command on this repository. It will operate as:

3. `$ git reset --mixed`

The above command will reset the status of the Head, and it will not delete any data from the staging area to match the position of the Head. Consider the below output:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git reset --mixed

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    like.txt

nothing added to commit but untracked files present (use "git add" to track)
```

From the above output, we can see that we have reset the position of the Head by performing the `git reset -mixed` command. Also, we have checked the status of the repository. As we can see that the status of the repository has not been changed by this command. So, it is clear that the mixedmode does not clear any data from the staging area.

Generally, the reset mixed mode performs the below operations:

- o It will move the HEAD pointer
- o It will update the Staging Area with the content that the HEAD is pointing to.

It will not update the working directory as git hard mode does. It will only reset the index but not the working tree, then it generates the report of the files which have not been updated.

Git Reset Head (Git Reset Soft)

The soft option does not touch the index file or working tree at all, but it resets the Head as all options do. When the soft mode runs, the refs pointers updated, and the resets stop there. It will act as git amend command. It is not an authoritative command. Sometimes developers considered it as a waste of time.

Generally, it is used to change the position of the Head. Let's understand how it will change the position of the Head. It will use as:

1. `$ git reset--soft <commit-sha>`

The above command will move the HEAD to the particular commit. Let's understand it with an example.

I have made changes in my file newfile2.txt and commit it. So, the current position of Head is shifted on the latest commit. To check the status of Head, run the below command:

2. `$ git log`

Consider the below output:

From the above output, you can see that the current position of the HEAD is on `c55453af995d2f12bce805311a8aeb7103765517` commit. But I want to switch it on my older commit `9479df87a319047e1a540f5c422`. Since the commit-sha number is a unique number that is provided by sha algorithm.

To switch the HEAD, run the below command:

1. `$ git reset -soft ae93fdcaa664a9479df87a319047e1a540f5c422`

The above command will shift my HEAD to a particular commit. Consider the below output:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ touch hello.txt

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git add --a

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git commit -m "hello added"
[master d6d5924] hello added
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 hello.txt
create mode 100644 like.txt

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git log --oneline
d6d5924 (HEAD -> master) hello added
b6710b4 kal added
c606358 naya added
a791c74 merge solved
1b1e773 done
d4604ce (super) updated
5632314 file added
5f4dc60 file added

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git reset b6710b4 --soft

DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git log --oneline
b6710b4 (HEAD -> master) kal added
c606358 naya added
a791c74 merge solved
1b1e773 done
d4604ce (super) updated
5632314 file added
5f4dc60 file added
```

As you can see from the above output, the HEAD has been shifted to a particular commit by git reset --soft mode.



In Git, the term revert is used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. However, it is not a traditional undo alternative. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit. Generally, git revert is a commit.

It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.

Moreover, we can say that git revert records some new changes that are just opposite to previously made commits. To undo the changes, run the below command: **Syntax:**

1. \$ git revert

Git Revert to Previous Commit

Suppose you have made a change to a file say **newfile2.txt** of your project. And later, you remind that you have made a wrong commit in the wrong file or wrong branch. Now, you want to undo the changes you can do so. Git allows you to correct your mistakes.

As you can see from the above output that I have made changes in newfile2.txt. We can undo it by git revert command. To undo the changes, we will need the commit-ish. To check the commit-ish, run the below command:

1. \$ git log

Consider the below output:

```
DELL@DESKTOP-JNGLI6M MINGW64 ~/Desktop/SCM PROJECT (master)
$ git log
commit fb3d8990e5f5fac1dd48449990f86f707fd2800a (HEAD -> master)
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 02:14:45 2022 +0530

    fish

commit 205361a6a62a33ed12c740555b235de08f2fd58b
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 02:09:59 2022 +0530

    Revert "fish added"

    This reverts commit e5869c4bd61d65506c594d7f0f16da6687bf9a97.

commit 09dda3d2a00006a9593db20b14ec1ff7a53ccd4d
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 02:09:24 2022 +0530

    dog added

commit e5869c4bd61d65506c594d7f0f16da6687bf9a97
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 02:08:57 2022 +0530

    fish added

commit b6710b44d6551ab9f7517cec8d8bbb10f6f77db9
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 01:57:51 2022 +0530

    kal added

commit c60635875a841073a9441f2c4fd715f11c11f797
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 01:57:27 2022 +0530

    naya added

commit a791c740fa8d85e413c91fcf3785b24967775da7
Merge: 1b1e773 d4604ce
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 01:56:30 2022 +0530
```

In the above output, I have copied the most recent commit-ish to revert. Now, I will perform the revert operation on this commit. It will operate as:

```
2. $ git revert ae93fdcaa664a9479df87a319047e1a540f5c422
```

The above command will revert my last commit. Consider the below output:

```
commit 205361a6a62a33ed12c740555b235de08f2fd58b
Author: shauryadhawan <shaurya1305.be21@chitkara.edu.in>
Date:   Fri May 27 02:09:59 2022 +0530

    Revert "fish added"

    This reverts commit e5869c4bd61d65506c594d7f0f16da6687bf9a97.
```

As you can see from the above output, the changes made on the repository have been reverted.

Difference Table

git checkout	git reset	git revert
Discards the changes in the working repository.	Unstages a file and bring our changes back to the working directory	Removes the commits from the remote repository.
Used in the local repository.	Used in local repository	Used in the remote repository
Does not make any changes to the commit history.	Alters the existing commit history	Adds a new commit to the existing commit history.
Moves HEAD pointer to a specific commit.	Discards the uncommitted changes.	Rollbacks the changes which we have committed.
Can be used to manipulate commits or files.	Can be used to manipulate commits or files.	Does not manipulate your commits or files.

