Session 2

Revision Selection

하나의 커밋을 선택하기 위해 40자로 이루어진 commit hash를 활용할 수도 있지만 더 인간 친화적인 방법들이 존재한다.

Short SHA-1

40자의 hash를 치는 대신 아래와 같이 부분적인 hash code로도 해당 커밋을 볼 수 있다.

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

> 2019년에 리눅스 커널은 875,000개의 커밋이 있었는데 앞 12자리로 각 커밋을 구분할 수 있었다.

Branch References

브랜치의 마지막에 있는 커밋이라면 브랜치 이름으로 커밋을 선택할 수도 있다.

```
$ git show topic1
```

Reflog Shortnames

git은 최근 몇 달간 HEAD 포인터의 변경 기록을 추적하는 reflog를 관리한다.

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive'
strategy.
1c002dd HEAD@{2}: commit: Add some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

아래의 명령어를 이용해 더 읽기 쉬운 방식으로 볼 수도 있다.

```
$ git log -g
```

이렇게 알게 된 reflog를 바탕으로 다음과 같이 커밋에 접근이 가능하다.

```
$ git show HEAD@{5}
```

Ancestry Reference

`^`를 이용해 부모 커밋을 선택할 수 있고, merge commit인 경우 `HEAD^`, `HEAD^2`와 같은 방법으로 두 부모 커밋을 구분해 선택할 수 있다.

> `HEAD^`가 merge target 브랜치에서의 부모 커밋을, `HEAD^2`가 merge된 브랜치에서의 부모 커밋을 나타낸다.

`~`을 이용해서도 부모 커밋을 선택할 수 있는데 `HEAD~2`, `HEAD~4`와 같이 뒤에 숫자가 붙는 경우 몇 번째 조상 커밋을 선택할지 지정하는 기능을 한다.

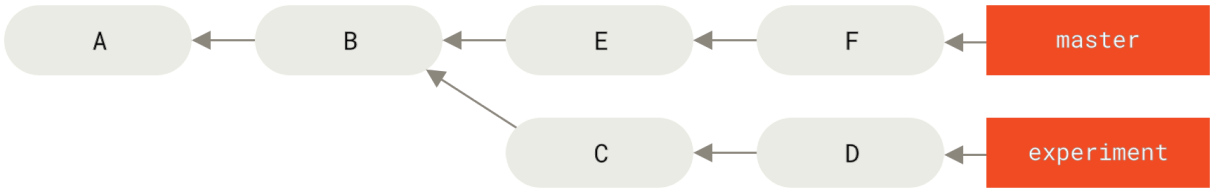
> `HEAD~~~`와 `HEAD~3`은 동일한 뜻이다.

> `^`와 `~`를 결합해 `HEAD~3^2`와 같이 커밋을 선택할 수도 있다.

Commit Ranges

여러 개의 커밋을 선택할 수 있는 기능으로 브랜치끼리 비교 시 유용하게 사용될 수 있다.

> "A 브랜치에는 있는데 B 브랜치에는 없는 커밋은 뭐지?"와 같은 상황에서 쓸 수 있다.



예를 들어 experiment 브랜치에는 존재하지만 master 브랜치에는 없는 커밋을 보고 싶으면 다음과 같이 입력하면 된다.

```
$ git log master..experiment
D
C
```

여러 개의 브랜치 비교를 위해서는 `--not`이나 `^`를 이용하면 된다. 예를 들어 브랜치 A와 B에는 존재하지만 C에는 없는 커밋을 보고 싶다면 다음과 같은 명령어를 이용하면 된다.

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Interactive Staging

`add -i`를 이용해 기존에 `add` 명령어를 이용하는 것보다 자세하게 staging할 파일을 관리할 수 있다.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now>
```

`p` 옵션을 이용해 한 파일의 변경사항들을 부분별로 나눠서 staging area에 올릴지 말지 고를 수 있다.

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

기본적으로 `y/n`을 이용해 해당 hunk를 넣을지 말지 결정할 수 있다. `git add -p <file>` 명령어를 통해서도 같은 일을 할 수 있다.

> 이런 기능은 gui를 이용하는 게 더 빠르지만 이런 방식으로 작동한다는 걸 알아두자.

Stash

A라는 브랜치에서 개발을 하다가 B라는 브랜치로 checkout을 하고 싶은데 현재 작업중인 내용이 있고 B라는 브랜치와 충돌이 발생할 경우 checkout을 할 수 없다. 이럴 때 stash 기능을 이용하면 문제 없이 다른 브랜치로 이동할 수 있게 된다. stash는 변경사항을 담을 수 있는 서랍과 같다.

push

변경사항들을 stash 하고 싶다면 `git stash` 또는 `git stash push` 명령어를 이용하면 된다.

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

내가 이전에 stash 했던 변경사항들을 보려면 `git stash list`를 이용하면 된다.

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

apply & pop

특정 stash의 변경사항을 적용시킬 때는 `$ git stash apply stash@{번호}`를, 가장 최근 stash를 적용시킬 때는 `$ git stash apply`를 입력하면 된다.

> `$ git stash -u` 명령어를 입력하면 untracked 상태인 파일들도 stash 할 수 있다.

> `$ git stash pop` 명령어를 통해 스택 최상단의 stash를 적용하며 stack에서 삭제할 수도 있는데 충돌나면 소중한 변경사항이 사라질 수 있으니 주의해서 사용하자.

> `$ git stash branch <new branchname>`를 이용하면 충돌이 나는 상황을 모면할 수 있다.

drop

적용이 끝난 stash는 `git stash drop` 명령어를 이용해 삭제할 수 있다.

```
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

위에서 알아본 interactive staging처럼 `git stash --patch | -p`를 이용하면 파일을 부분별로 stash할 수 있다. 마지막으로 `git clean` 명령어를 이용해 untracked 상태인 파일들을 삭제할 할 수도 있다.

Searching

grep

git에서 제공하는 검색 기능으로 현재 working tree뿐만 아니라 과거의 snapshot에 대해서도 검색할 수 있다.
`$ git grep <string> <revision>`

log searching

`$ git log -S <string>` 옵션을 이용하면 과거 기록들 중 `<string>`에 해당하는 문자열을 변경했던 커밋들만을 찾을 수 있다. 이를 통해 특정 함수나 변수가 언제 선언됐는지 쉽게 찾을 수 있다.

`$ git log -P :<function>:<file>` 명령어를 이용하면 `<file>`안에 있는 `<function>`이 생성된 커밋부터, 내용을 변경시킨 커밋들을 찾아 변경사항들을 볼 수 있다.

Bisect

버그가 발생하긴 했는데 언제부터 났는지 모르겠다면 쓸 수 있는 명령어이다. 이 명령어를 의미 있게 사용하기 위해서는 전제조건으로 커밋을 잘 분리해 놨어야 한다.

> 뭉탱이로 커밋을 막 작성했다면 커밋을 찾아도 어디서 버그가 발생했는지 찾는데 한세월이다.

기본적인 작동 원리는 커밋 기록들을 이분탐색하며 어떤 커밋에서 처음으로 버그가 발생했는지 찾을 수 있게 해주는 명령어이다.

$ git bisect start
$ git bisect bad # Current version is bad
$ git bisect good v2.6.13-rc2 # v2.6.13-rc2 is known to be good
정상적으로 작동했던 커밋과, 버그가 발생한 커밋을 위와 같이 넣고 시작하면 이분탐색이 시작된다.

이분탐색을 하며 버그가 발생하면 `git bisect bad`, 발생하지 않으면 `git bisect good`를 입력하며 진행할 수 있다. 이렇게 진행하다보면 가장 처음 버그가 발생한 커밋을 찾을 수 있다. 해당 커

밋은 `refs/bisect/bad`에 등록되고 추후에 `$ git bisect reset bisect/bad`를 통해 해당 커밋으로 이동할 수도 있다.

이분탐색이 끝났다면 `$ git bisect reset`를 입력해 bisect을 시작한 커밋으로 HEAD를 되돌릴 수도 있고, `$ git bisect reset <commit>`을 입력해 원하는 커밋으로 이동할 수도 있다.

Rewriting History

Amend commit

가장 마지막 커밋을 수정하기 위해서 `$ git commit --amend` 명령어를 이용할 수 있다. 만약 커밋 메시지만 수정하고 싶다면 `$ git commit --amend --no-edit`을 사용해도 된다.

> amend commit을 만들 때 이미 push한 commit이라면 다른 방법을 생각해보자, 커밋의 내용이 바뀌지 않더라도 커밋 해쉬값이 바껴 브랜치가 꼬일 수 있다.

Interactive Rebasing

만약 여러 커밋의 커밋 메시지를 수정하고 싶다면 `$ git rebase -i` 명령어를 이용하면 된다. `rebase` 명령어는 전에 브랜치를 합칠 때 사용했던 그 명령어와 동일한 명령어지만 `-i` 옵션을 통해 더 많은 일들을 할 수 있게 도와준다.

예를 들어 최근 3개의 커밋 메시지를 변경하고 싶다면

```
$ git rebase -i HEAD~3
```

를 입력한 후에

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log
message
# x, exec <command> = run command (the rest of the line) using shell
```

```
# b, break = stop here (continue rebase later with 'git rebase --
continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

위와 같이 세 커밋을 전부다 `edit` 또는 `reword` 상태로 변경하면 된다. `edit`의 경우에는 실제로 커밋 내용까지도 수정이 가능하고, `reword`는 커밋 메시지만을 수정한다.

`squash`, `fixup`은 여러 커밋을 합칠 때 사용할 수 있고, 커밋끼리의 순서를 변경해서 저장하면 그 순서가 반영된다.

커밋 분리하기

위에서 살펴본 `edit` 상태에서는 staging area에 등록돼 있는 파일들을 일부 unstage 시키는 방법으로 한 커밋을 여러개의 커밋으로 분리할 수 있다. 이후에 `git rebase --continue`를 입력하면 한 커밋이 두 개로 분리된다.

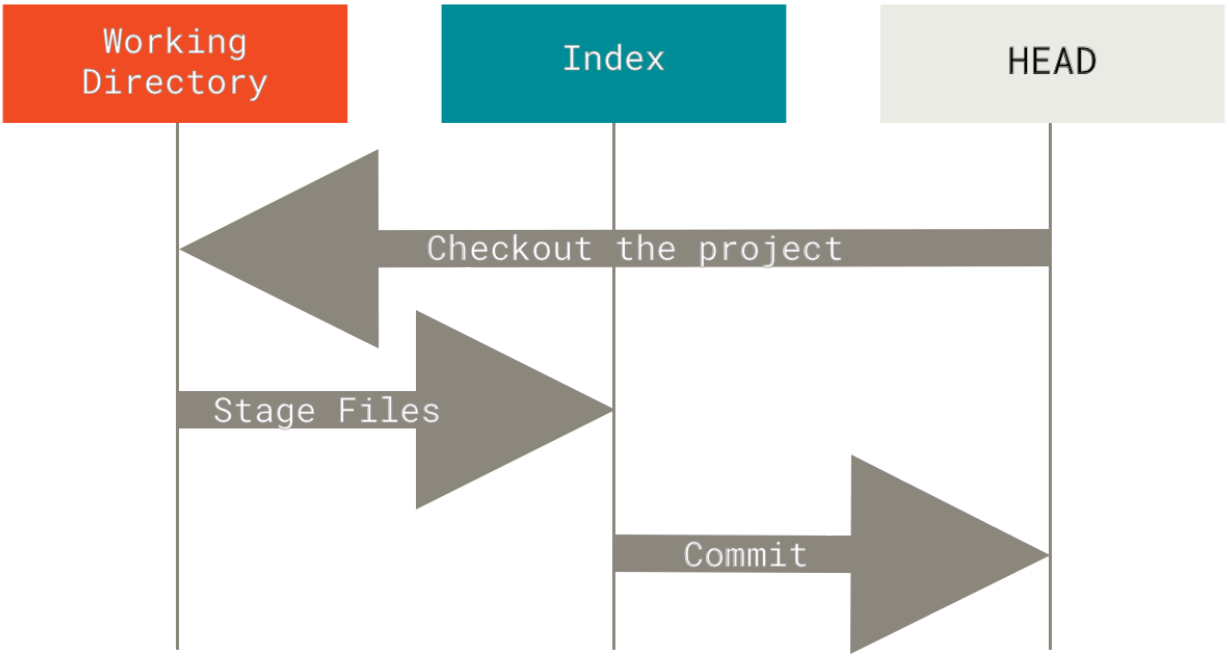
커밋 삭제하기

`drop` 옵션을 이용하면 커밋을 통째로 삭제시켜 버릴 수 있다.

> 부모 커밋이 변경되면 뒤에 있는 모든 커밋의 내용들도 변경되기 때문에 쫄리면 언제든지 `$ git rebase --abort`를 입력하자.

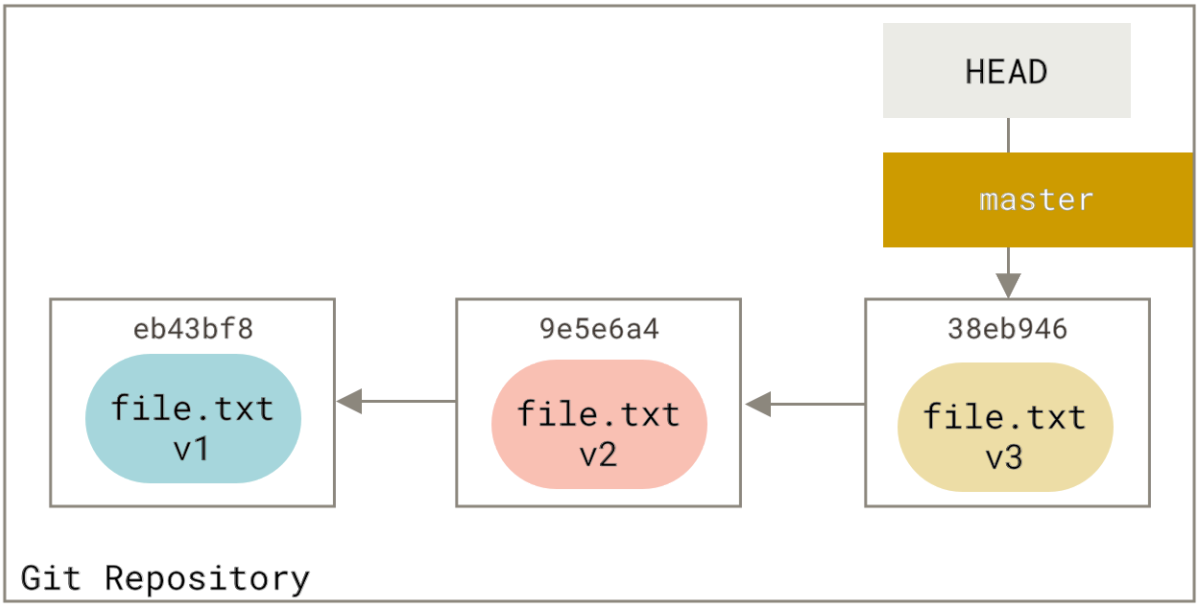
Reset Demystified

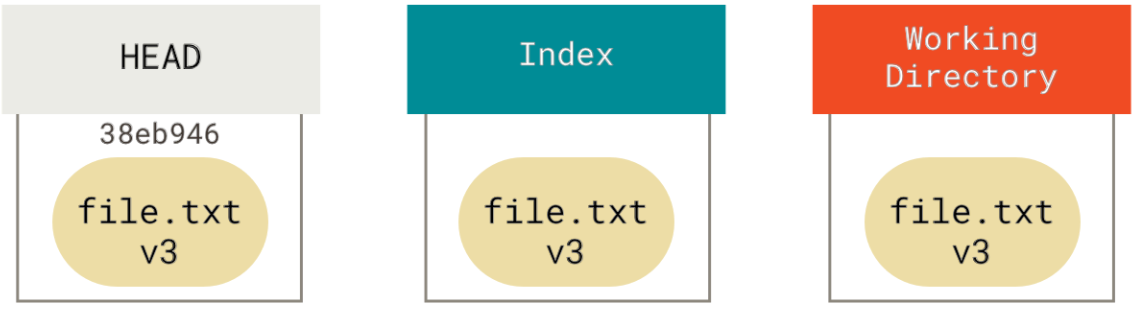
git에서 관리하는 묶음을 하나의 tree로 생각하면 다음과 같이 3개의 tree가 존재한다. tree를 파일들의 묶음 또는 파일에 대한 변경사항들의 묶음이다.



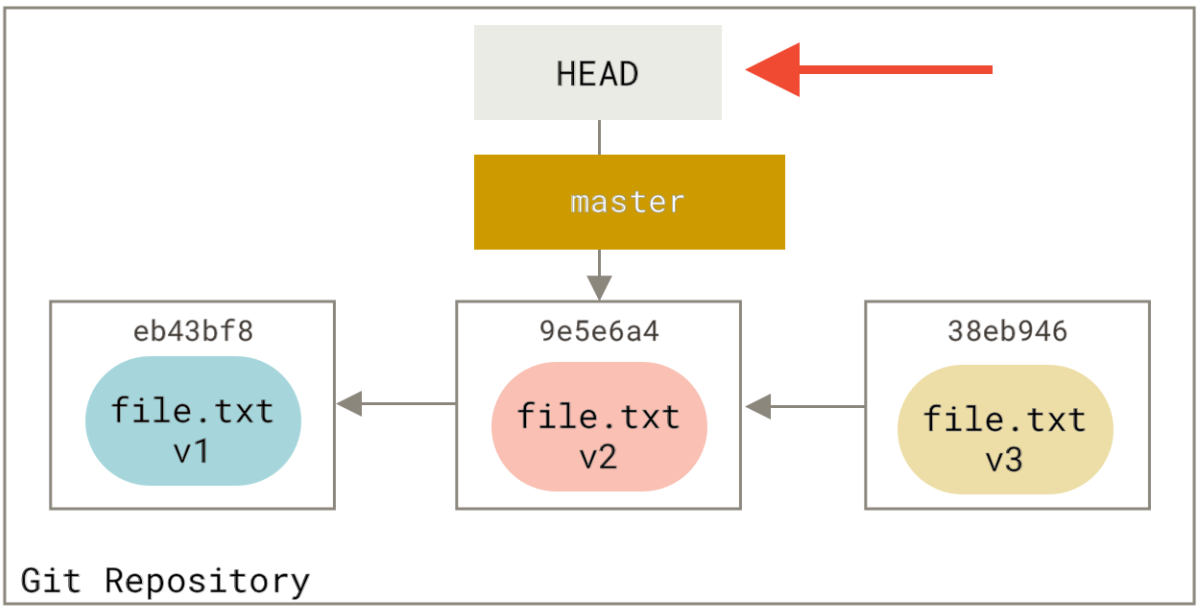
- Working Directory: 실제 내 변경사항들이 저장되는 tree
- Index: `add` 명령어를 통해 staged된 변경사항들로 다음 커밋의 후보 tree
- HEAD: 현재 가르키고 있는 브랜치의 마지막 커밋에 대한 tree

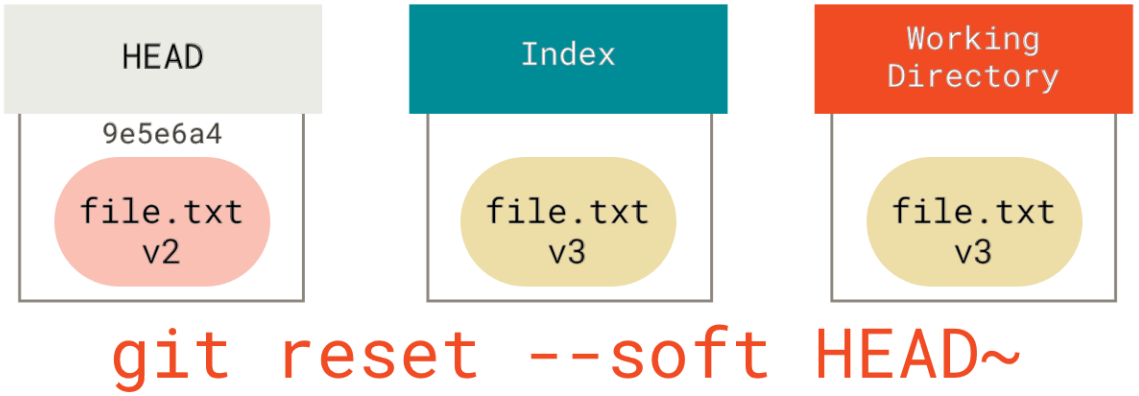
현재 아래와 같은 상황이라고 가정하면





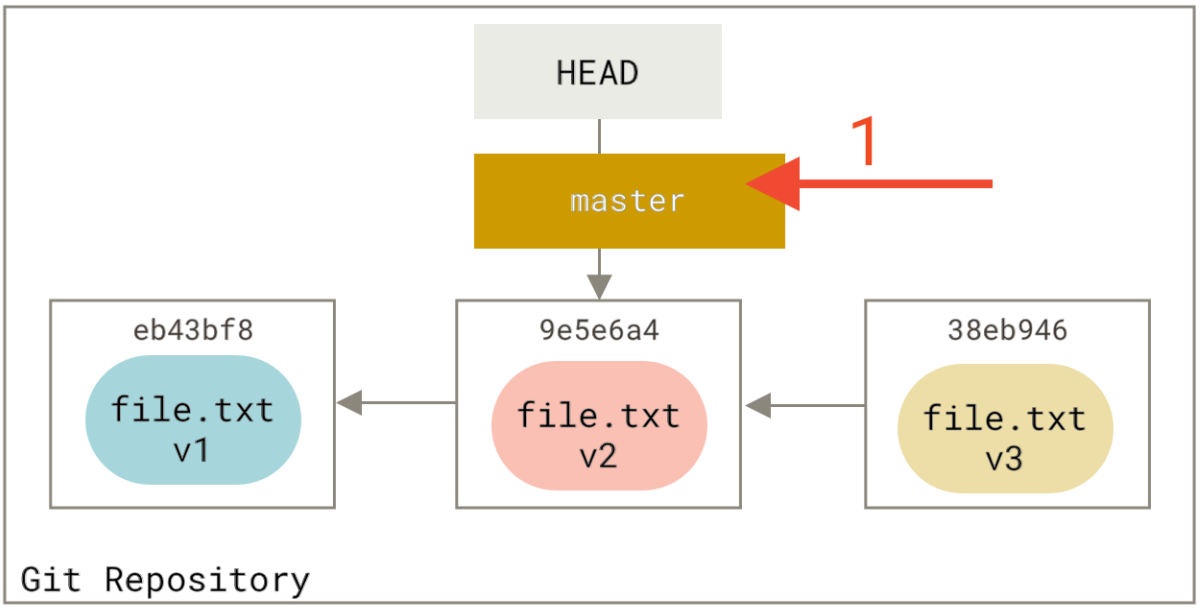
`$ git reset --soft HEAD~`는 HEAD tree만 v2로 만든다.

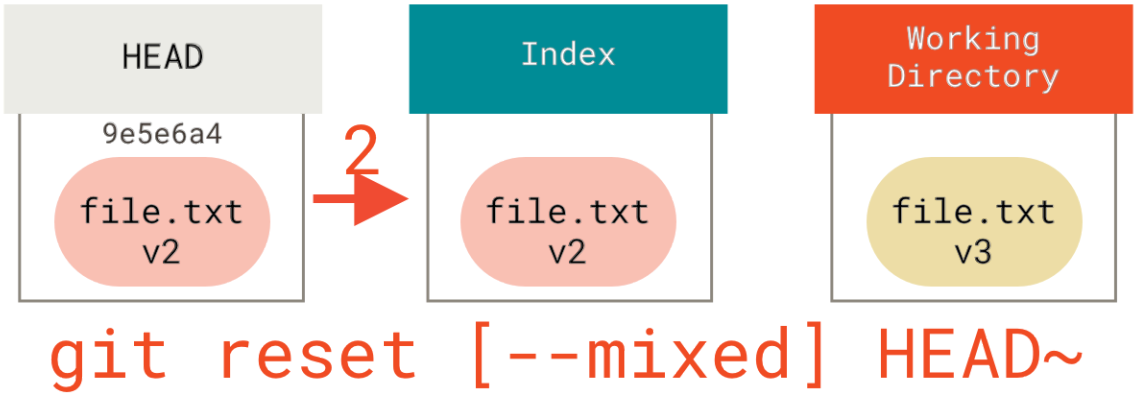




git reset --soft HEAD~

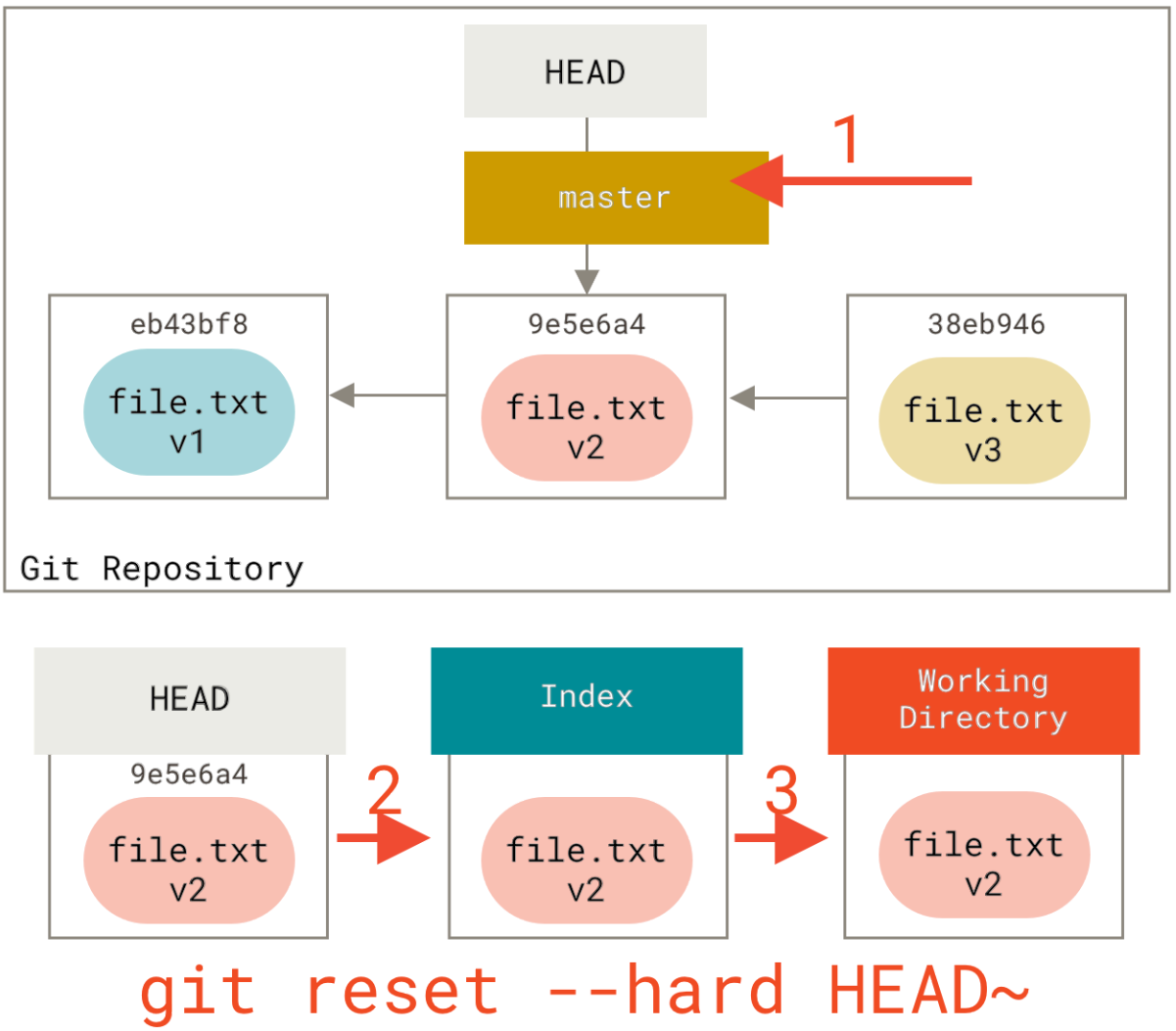
`$ git reset HEAD~`는 HEAD, Index를 v2로 만든다.





마지막으로 `$ git reset --hard HEAD~`는 HEAD, Index, Working Directory에서 v3에 대

한 변경사항을 모두 삭제하고 v2로 되돌린다.



`checkout`과 `reset` 모두 인자로 `<path>`를 넣을 수 있는데 HEAD는 이동시키지 않고 `<path>`에 해당하는 파일의 내용만 특정 커밋에서 불러올 수 있다.

> `$ git checkout <commit> <paths>`는 Working Directory의 모든 변경사항을 삭제시키기 때문에 주의해서 사용해야 한다.