오픈소스소프트웨어-중간고사

1차시

- 오픈소스소프트웨어: 소스 코드의 공개를 뜻하는 용어

소스 코드가 공개적으로 접근할 수 있게 설계되어 누구나 자유롭게 확인,수정,제작.재배포 가능

미래의 공동 연구를 위해 저작권자는 보호하며 공유하면서 연구하는 시스템

- 깃

소스 코드 관리를 위한 **분산 버전관리 시스템**

- 깃허브

5천만 명이 넘는 개발자와 함께 소프트웨어를 만드는 개발 플랫폼

깃 기반의 저장소 및 소프트웨어 협업 개발을 위한 웹 호스팅 서비스

장점: 1. 기업에 상관 없이 중요 개발운영 기능을 제공

2. 협업 시 전 세계 소프트웨어 개발자와 함께 작업 가능

2차시

버전 관리 필요성: 과거 지점의 버전으로 돌아가 누가, 무엇을 수정했는지 파악 가능

버전 관리: 시간 흐름에 따라 파일 집합에 대한 변경 사항을 추적,관리

저장과 백업 기능이 있음, 여러 사람이 추적 관리가 가능 (마치 시간 여행)

- 버전 관리 도구 종류

1. 명령어 줄 인터페이스 방식 (CLI)
2. 그래픽 사용자 인터페이스 방식 (GUI)

- 커밋: '~적어두다'
- 버전관리의 커밋: 저장소의 현재 상태를 저장 (스냅샷 사진을 찍는 것)
- + 파일 집합의 변경 내용을 깃 저장소에 기록하는 작업

→ 이전 커밋 상태부터 현재 상태의 변경 이력이 기록된 커밋이 생성됨

→ 시간순으로 저장됨

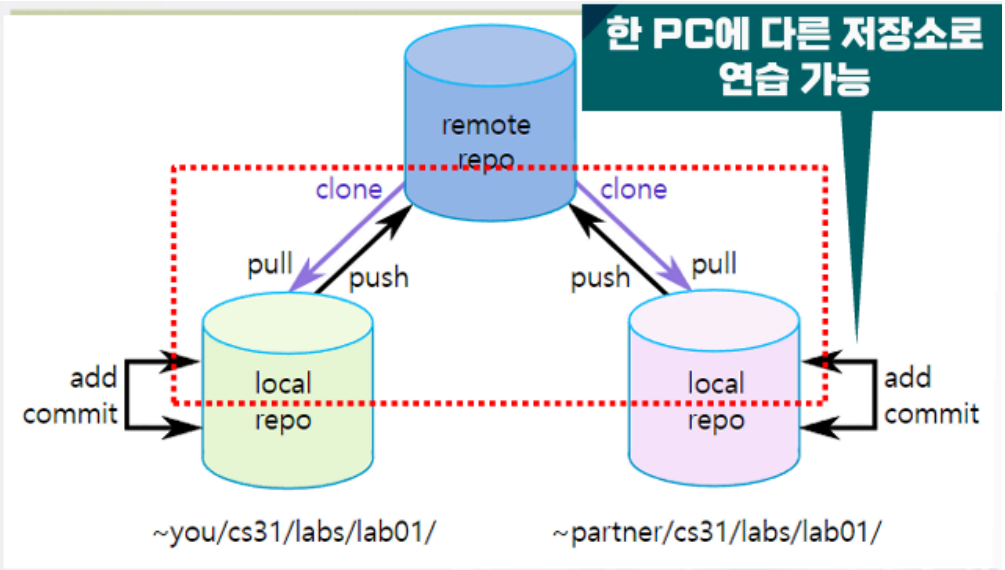
- 헤드(HEAD): 가장 최근의 커밋을 가르키는 포인터

- 저장소: 파일이나 폴더를 저장해 두는 곳

1. 원격저장소: 클라우드 같은 공유하고 전용 서버에서 관리됨
2. 지역저장소: 내 컴퓨터에서 저장되는 것

- 원격 저장소와 지역 저장소의 명령

1. 클론: 서버의 원격 저장소를 지역 저장소에 복제
2. 푸쉬: 지역저장소에서 원격저장소로 올리기
3. 풀: 원격 저장소에서 지역 저장소로 내리기
4. 애드: 파일을 작영 영역에서 스테이징 영역에 추가
5. 커밋: 스테이징 영역에서 커밋을 진행해 변경점을 기록



3차시

- 깃: 리누스 토발즈가 개발

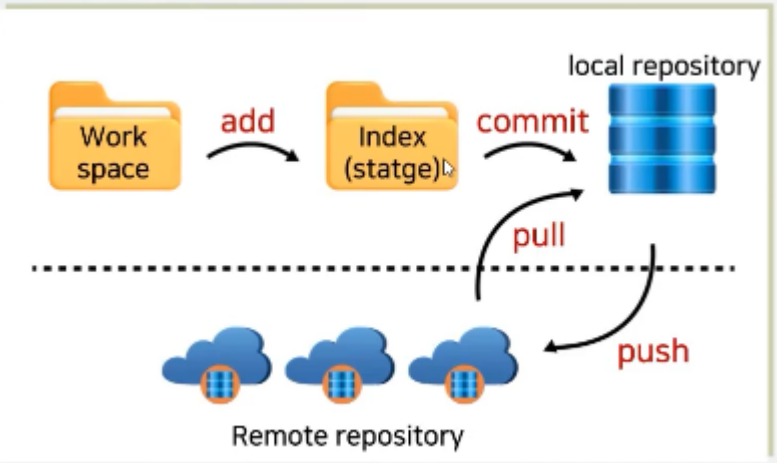
- 깃 사용 장점: 모든 개발자는 지역 시스템에 코드의 전체 사본을 소유하며 소스 코드의 모든 변경점은 다른 사용자가 추적이 가능함
- 깃 기능: **컴퓨터 파일의 변경을 추적하는 데 사용되는 버전 관리 시스템**

1. 여러 개발자가 함께 작업
2. 소스 코드의 변경 사항을 추적하는 데 사용
3. 소스 코드 관리에 분산 제어 도구가 사용
4. 여러 개의 평행 분기를 통해 비선형 개발을 지원 (브랜치라는 기능)

- 하나의 노선이 아니라 다른 노선도 생기면서 합쳐지는 것 / 새로운 수정을 할 수 있는 또 다른 버전의 작업 흐름 (HEAD 파일을 통째로 복사해 독립적으로 다시 개발을 진행하는 개념)
- 깃을 설치하면 CLI방식과 GUI 방식 2가지가 있는데 CLI방식을 할 수 있으면 GUI를 쓸 수 있지만 GUI에서 CLI를 다루기는 어렵다

- 깃 내부 저장소 영역

1. 작업 영역: 눈에 보이는 작업 공간,폴더 등
2. 스테이징 영역
3. 깃 저장소
4. 임시 저장소



- 깃 허브: 버전 관리를 위한 서버 저장소 및 프로젝트 개발을 위한 협업 관리 서비스

1. 세계 최대의 오픈 소스 공유 플렛폼
2. 개발자들의 소셜 네트워크
3. **깃과 깃허브는 다름**

- 깃은 버전 관리 제어 소프트웨어 / 깃 허브는 개발자들의 협업 커뮤니티로 웹 호스팅 서비스

2018년 6월 마이크로소프트가 사며 엄청난 수의 개발자들이 사용 중

5차시

- git config —설정 범위 설정 변수 설정값
- 설정 범위 종류: 1. system →모든 사용자 2. global → 현재 사용자의 모든 저장소 3. local→ 현재 사용자의 현재 저장소 / 보통은 global 사용
- git config —global user.name mingduddu :사용자 이름 설정
- git config —global user.email alice46452@naver.com : 사용자 이메일 설정
- git config —global core.autocrlf true :자동 줄바꿈
- git config —global core.safecrlf false : 줄바꿈 안전 설정
- git config —global core.editor 'code —wait' :기본 편집기 설정
- git config —global init.defaultBranch main : 기본 브랜치 이름 설정

- git init basic :basic이라는 깃 저장소를 새로 만듬
- git init . : 현재 폴더를 git 저장소로 만듬
- cd basic : basic이라는 폴더로 이동
- ls -al : 폴더 안에 어떤 파일이 있는지 확인 가능 (숨겨져 있는 애도 알 수 있음)

6차시

- Visual studio code: 마이크로소프트에서 개발한 오픈 소스 에디터 소프트웨어

1. 가볍지만 강력한 소스 코드 편집기
2. 여러 언어를 기본적으로 지원하고 확장 기능이 많음
3. 파일 편집과 버전 관리도 지원함

- pwd - 현재 폴더 표시
- cd - 폴더 바꿈
- mkdir 폴더 이름 - 폴더 만들기
- ls - 폴더 리스트 (-l,-a,-al 있음)
- touch 파일 이름 - 빈 파일을 만듬
- echo - 거울 같이 입력한 내용을 그대로 화면에 출력해줌
- cat - 파일의 내용을 출력
- cp a b 파일 a를 b로 복사
- mv f1 f2 - 파일 f1을 f2로 이름 수정
- > - 기존에 있는 파일 내용을 지우고 저장
- >>- 기존 파일 내용 뒤에 덧붙여서 저장

상태 코드 의미

코드	의미	설명
	수정되지 않음 (Unmodified)	파일이 깨끗한 상태입니다.
M	수정됨 (Modified)	파일이 수정되었습니다.
A	추가됨 (Added)	새로운 파일이 스테이징되었습니다.
D	삭제됨 (Deleted)	파일이 삭제되었습니다.
R	이름이 변경됨 (Renamed)	파일의 이름이 변경되었습니다.
C	복사됨 (Copied)	파일이 복사되었습니다.
U	병합되지 않음 (Unmerged)	병합 충돌이 발생했습니다.
?	추적되지 않음 (Untracked)	Git이 아직 추적하지 않는 파일입니다.
!	무시됨 (Ignored)	`.gitignore` 파일에 의해 무시되는 파일입니다.

7차시

- add: working tree → staging area
- commit: staging area → git reposiory

- git status : 깃 저장소의 현재 상태를 확인하는 명령어 (작업 트리와 스테이징 영역)

1. git status : 현재의 상태를 표시
2. git status -s : 현재의 상태를 간단히 표시

- git commit : 스테이징 영역에 있는 내용을 스냅샷을 찍어 버전으로 저장
- git commit -am 메세지는 추적 안 된 파일을 바로 커밋할 수 없음 한 번 커밋이 된 상태여야 함 (즉 한 번 추적이 된 상태여야 하는 것)

주요 명령	
$ git commit	커밋 메시지를 입력할 기본 편집기 실행됨
$ git commit -m 'message'	커밋 메시지를 직접 입력 [-m | --message]
$ git commit -a -m 'message' $ git commit -am 'message'	추가와 커밋을 함께 실행 [-a | --all]

- git log: 로그 이력 보기
- git log는 커밋의 흐름을 보는데 제일 자주 사용함
- 커밋 id → 커밋 헤드 → 브랜치 이름 → 커밋 메세지 → 제목 순서로 결과가 출력됨
- git log -p: 커밋 정보 뿐 아니라 커밋 파일의 차이가 표현이 됨

주요 명령	
$ git log	로그 이력 정보를 표시
$ git log --oneline	로그 이력을 한 줄로 표시
$ git log [--patch | -p]	로그 이력과 함께 파일의 변화(이전 커밋과의 차이)를 표시

파일 차이 표시 해석

구분	내용	해석	
비교 파일	diff -git a/hello.txt b/hello.txt	비교 대상 이전 파일 a/hello.txt	비교 대상 이후 파일 b/hello.txt
파일 모드	new file mode 100644	파일모드 100644: 100(실행파일) 644[110 100 100](파일참조 권한)	
파일 ID	index 0000000..359af40	SHA1에 의한 파일 ID: 359af40	
이전 이후	--- /dev/null +++ b/hello.txt	이전 파일(---) 없던 파일	이후 파일(+++) 생성된 파일 이름 hello.txt
내용 비교 요약	@@ -0,0 +1 @@ @@ -0,0 +1 @@ 앞뒤로 @@는 의미 없이 표시	이전 파일은 없던 것 @@ -0,0	이후 파일은 1줄부터 한 줄 표시 +1은 +1, 1을 의미 +1 @@
내용 차이	+create	없던 내용	첫 줄이 create

- git show : 특정한 커밋 정보를 확인하는 명령어
- git show는 흐름이 아닌 특정 커밋의 상세 정보와 현재 커밋과의 차이를 보기 위해서 사용함

주요 명령	
$ git show	마지막 커밋(HEAD)의 커밋 정보 표시
$ git show --oneline	커밋 로그 한 줄과 파일 차이 표시
$ git show -s	파일 차이는 표시되지 않음
$ git show [HEAD]	지정한 HEAD의 커밋 정보 표시
$ git show [commitID]	지정한 commitID의 커밋 정보 표시

<💯> — git log는 흐름을 파악하고 git show는 비교와 상세정보를 아는데 최적화 된 것

8차시

수행 순서와 명령어	작업 디렉토리(폴더) [hello.txt]	스테이지(Index) 영역 [hello.txt]	깃 저장소와 커밋 이력	

❸ `$ git commit -m A`
`$ git status`
`$ git log`
`$ git log --oneline`
`$ git log [--patch | -p]`
	aaa	aaa	A
			aaa
				HEAD

❷ `$ git add hello.txt`
`$ git status`
`$ git status [--short | -s]`
`A hello.txt`				

❶ `$ echo aaa > hello.txt`
`$ cat hello.txt`
`$ git status [--long]`
`$ git status -s`
`?? hello.txt`				

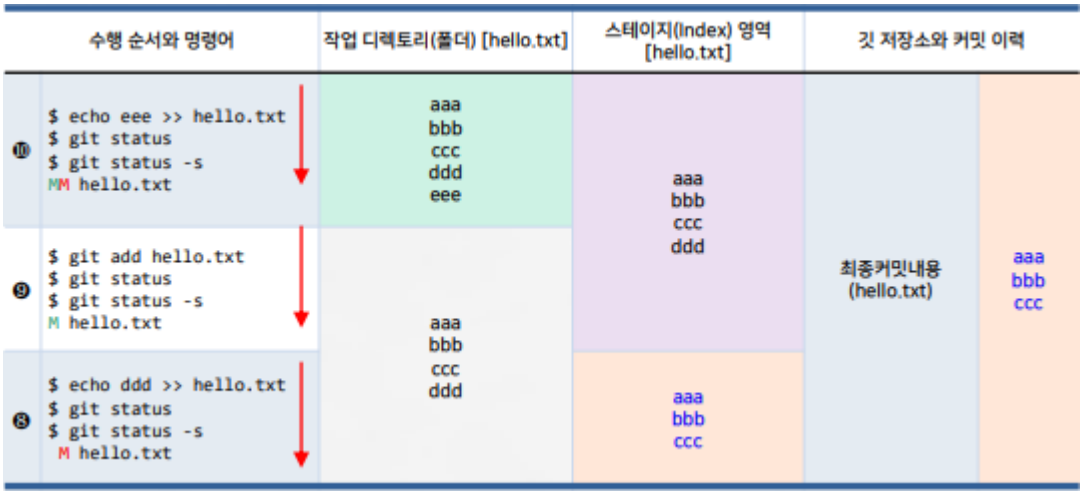
수행 순서와 명령어	작업 디렉토리(폴더) [hello.txt]	스테이지(Index) 영역 [hello.txt]	깃 저장소와 커밋 이력	

❻ `$ echo ccc >> hello.txt`
`$ cat hello.txt`
`$ git status`
`$ git status -s`
`M hello.txt`
	aaa
	bbb
	ccc
		aaa
		bbb		

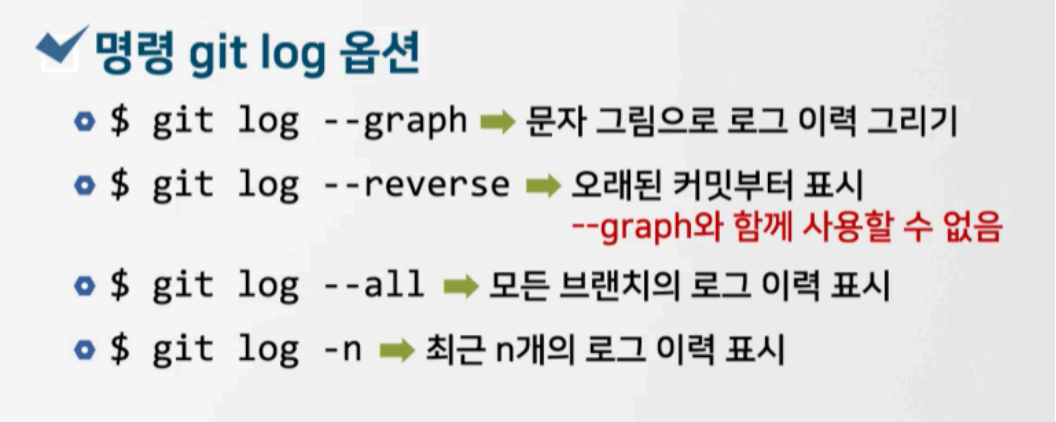
❺ `$ git commit -am B`
`$ git status`
`$ git log`
`$ git log --oneline`
`$ git log [--patch | -p]`
	aaa
	bbb
			B
			aaa
			bbb
				HEAD

❹ `$ echo bbb >> hello.txt`
`$ cat hello.txt`
`$ git status`
`$ git status -s`
`M hello.txt`
		aaa	↓	

첫번째 행이 스테이징 영역 두번째 행이 작업 디렉토리 영역



- git log —graph: 문자 그림으로 로그 이력 그리기 (브랜치 흐름도 표시가 됨)



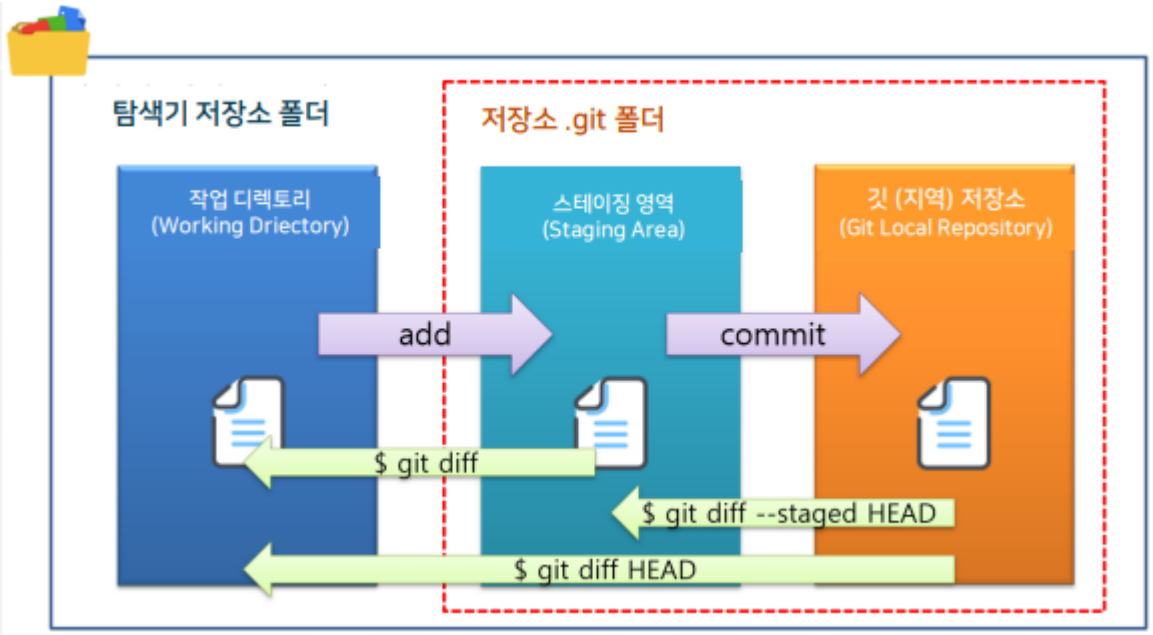
- git checkout HEAD~ : 이전 커밋으로 이동할 수 있음 - 현재 상태가 깨끗해야 과거로 돌아갈 수 있음 (add나 현재 수정 사항이 있으면 안 됨)
- git checkout - : 이전 checkout으로 이동
- git checkout main : 최신 커밋으로 이동
- git checkout을 통해 HEAD 현재 작업 중인 공간이 분리되면 **detached 상태라고 함**

rm -rf (폴더 이름) 하면 삭제 실킬 수 있음

git stash를 사용하면 깃 저장소에 있는 걸 작업 디렉토리와 스테이지 영역과 동일하게 만들 수 있음

10차시

- 작업 디렉토리가 폴더에 보이는 영역이고
- .git 안에 정보로만 존재하는 영역이 스테이징 영역과 깃 저장소 영역임



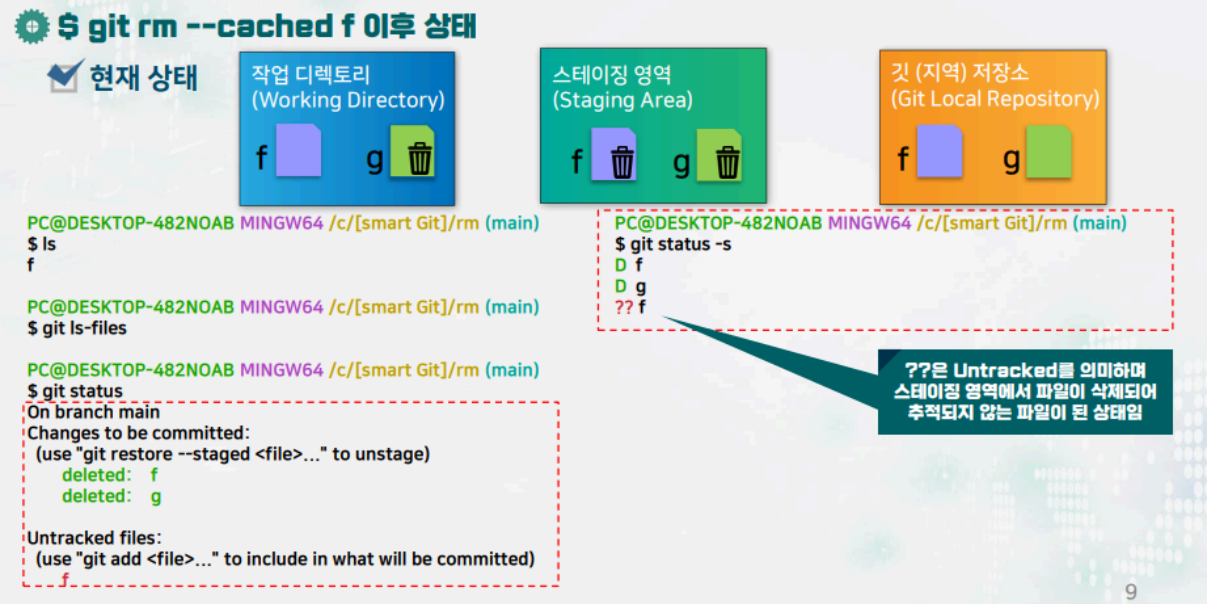
- git diff : 스테이징 영역 기준 작업 디렉토리와 비교
- git diff —staged : HEAD(제일 최근 버전) 기준으로 스테이징 영역과 비교
- git diff HEAD : HEAD 기준으로 작업 디렉토리와 비교
- git diff HEAD~2 HEAD : 전전 커밋과 최근 커밋을 비교
- git diff (기준점) (비교할 대상) 인 듯 아무것도 표시하지 않는다면 기본적으로 작업 디렉토리가 비교할 대상이 됨
- git diff를 사용했을 때 - - -가 뜬다면 삭제가 된 내용 (파일 이전) +++가 뜬다면 추가된 내용 (파일 이후)

11차시

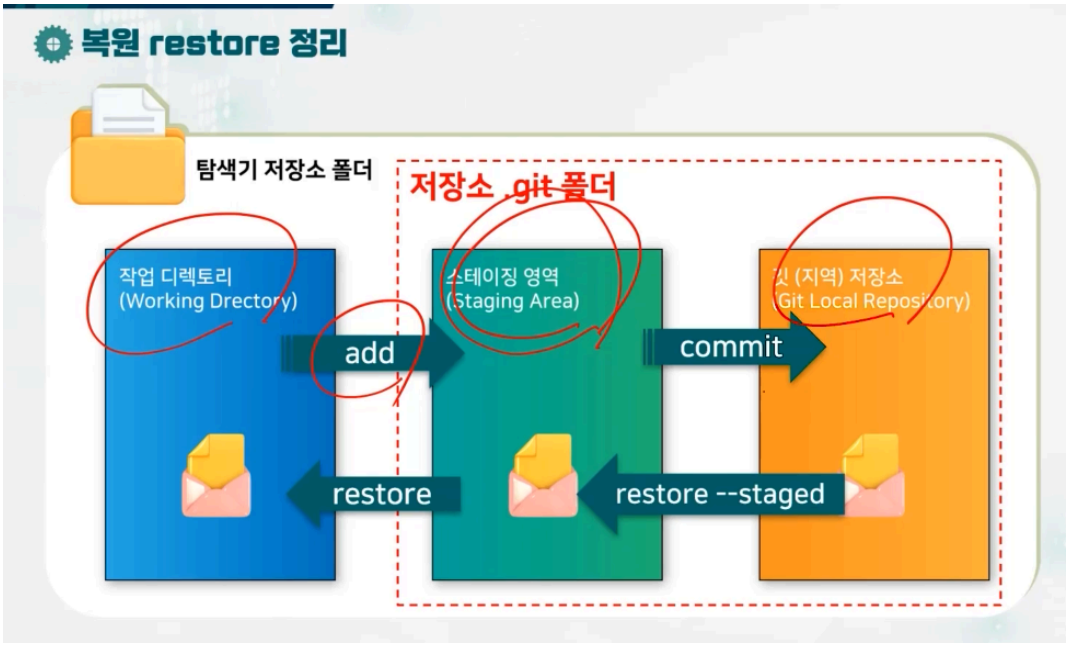
- rm [file] : 작업 디렉토리에서만 삭제하는 명령어
- git rm [file] :작업 디렉토리와 스테이징 영역에서 모두 삭제됨
- git rm —cached[file] : 스테이징 영역에서 file이 삭제됨 작업 디렉토리는 괜찮음

> 항상 작업 디렉토리는 스테이징 영역과 비교하고 스테이징 영역은 깃 저장소와 비교를 한다

- status는 작업 디렉토리와 스테이징 영역을 비교하는데 스테이징 영역에서 삭제되고 디렉토리에 있다면 초록색 D와 ??가 떠서 untracked file이고 둘다 없다면 초록색 D만 뜬다



- git restore : 삭제나 수정이 된 파일을 복구 가능 / 기본적으로 작업 디렉토리 영역과 스테이징 영역이 같은 상태가 됨
- git restore —staged f : 스테이징 영역이 깃 저장소와 같은 상태가 됨 항상 staged를 붙이면 스테이징과 깃 저장소를 비교하거나 그 기능을 한다는 것을 기억해야함
- git restore —source=HEAD—staged—worktree [file]을 사용하면 모두 영역이 같은 상태가 됨
- —source 는 from과 같은 의미로 사용이 됨



—source 는 from과 같은 의미로 사용이 됨

12차시

git config —global alias.별칭 이름 '설정할 명령어' : (명령어를 내 맘대로 설정 가능)

```
승 세 훈 @DESKTOP-3J3F60S MINGW64 ~/gfile (main)
$ git config --global alias.ss 'status -s'
```

깃 저장소는 rm을 사용할 수 없음 대신 스테이징 영역에서 삭제된 걸 커밋하면 최근 커밋에서 g라는 파일이 삭제되는 원리

13차시

- 버전: 프로그램을 수정하거나 개선할 때마다 코드를 구분하려고 부여된 식별자를 의미 - 보통 두 자리나 세자리의 형태로 숫자를 사용하거나 년,월을 사용함
- SemVer 방식:세 자리 숫자 형태로 표기하는 버전 (major.minor.patch)

1. major(메이저) 번호: 첫 자리가 0으로 시작하면 아직 초기 개발 중인 제품이라는 의미 정식 버전은 1부터 시작하는데 이를 메이저 버전
2. minor(마이너) 번호: **메이저 버전에서 기능을 추가**하거나 **변경 사항이 있을 때** 바꿈
3. patch(패치) 번호: 버그 수정 등 미미한 변화가 있을 때 바꿈

태그(tag) 기능: 특정 커밋에 버전 번호나 다른 이름을 부여하는 기능

1. **주석 태그: 태그 이름 + 정보 (git tag -a)**
2. 일반 태그: 태그 이름만 포함 (git tag)

주석 태그 생성

- 주석 태그:누가,언제,태그 메세지 등의 정보가 있는 태그로 중복이 불가능 함
- **git tag -a** v1.0.0 -m 'first version' 이런 형식으로 사용
- git tag는 특정 commit에 태그를 붙이는 거고 -a는 주석이라는 의미 -m은 메세지 추가하는 거 태그는 항상 커밋에 붙이는 거
- -a를 안 붙인다면 일반 태그라는 거임 대신 일반 태그는 -m을 사용하지 못함
- 또한 주석 태그는 사용하면 누가,언제,이메일 이런 정보가 보이는데 일반 태그는 보이지 않음 + 메세지도 사용 못함

태그 목록을 보는 방법

- git tag는 예전 태그부터 표시
- git log는 최신 커밋부터 표시
- git show v1.0.0을 이용하면 (태그를 달아놨다면) 주석 태그의 여러 정보가 표시됨
- git tag에서 -d라는 속성을 사용한다면 태그만 삭제할 수 있다.

14차시

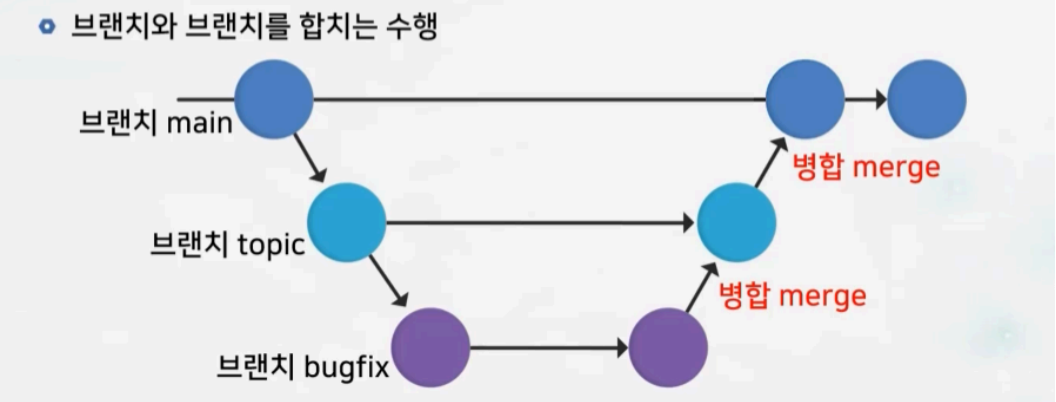
브랜치 : 커밋 사이를 가볍게 이동할 수 있는 포인터

깃에서의 브랜치 : 버전 관리를 수행하던 파일을 통째로 복사해 독립적으로 다시 개발을 진행하는 개념임

- 여러 개발자가 남을 신경 쓰지 않고 하나의 프로젝트에서 여러 갈래로 나누어 관리 되기 대문에 다양하게 작업을 진행할 수 있음

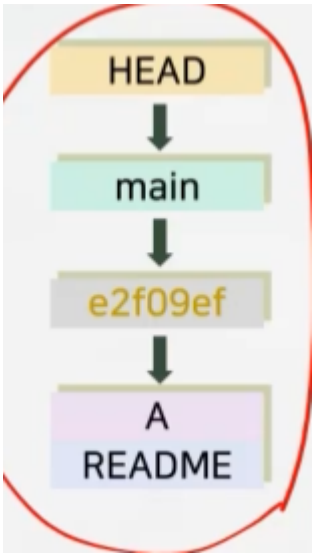
브랜치 사용 장점 : 내 개인 저장소에서는 다른 브랜치에 영향 없이 작업 가능

브랜치 병합(merge) : 독립된 브랜치에서 마음대로 소스 코드를 변경하여 작업한 후 원래 버전과 합칠 수 있음



HEAD:작업 중인 브랜치의 최신 커밋을 가리키는 포인터

결과 표시에서 (HEAD → main) 뜨는 건 main은 최근 커밋을 의미하고 HEAD는 현재 작업 중인 브랜치인 main을 가리킨다는 뜻

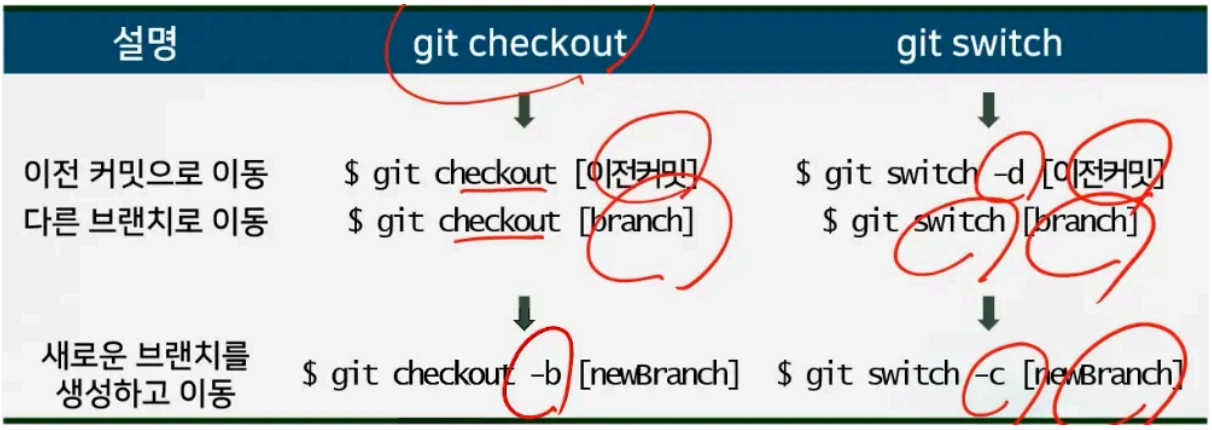


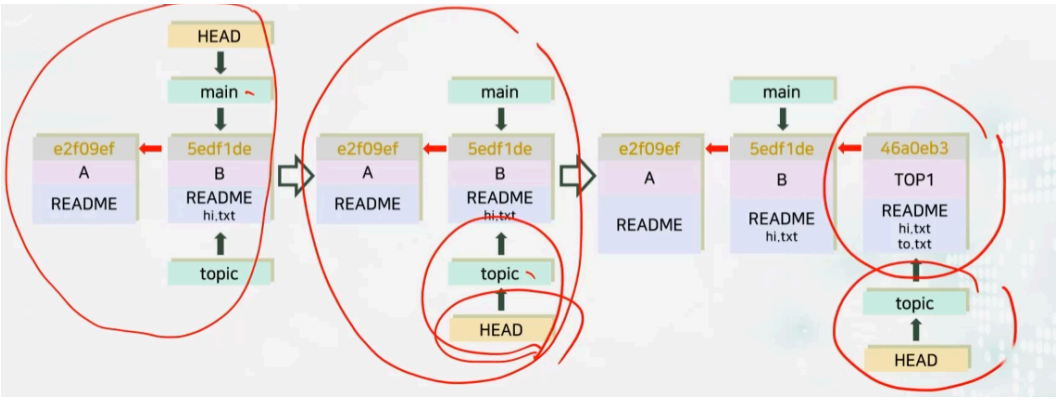
- git switch -c bname : bname의 브랜치를 생성하고 헤드를 이동
- git checkout -b bname : 위의 명령어와 같은 의미
- git branch: 커밋이 발생한 브랜치 목록을 보여줌
- git branch -v: 브랜치마다 마지막 커밋 id와 메세지도 표시가 됨

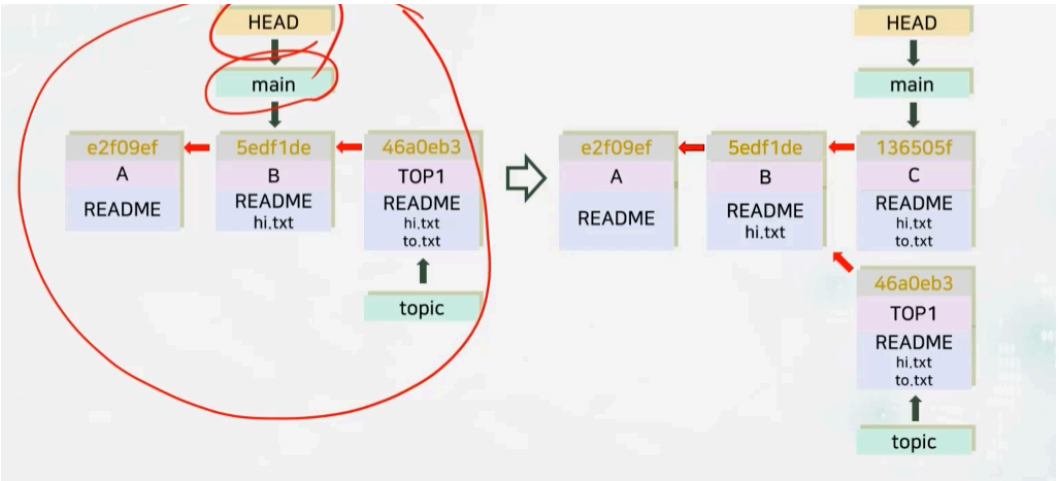
```
$ git branch -v
* main   34a7782 B
  topic 34a7782 B
```

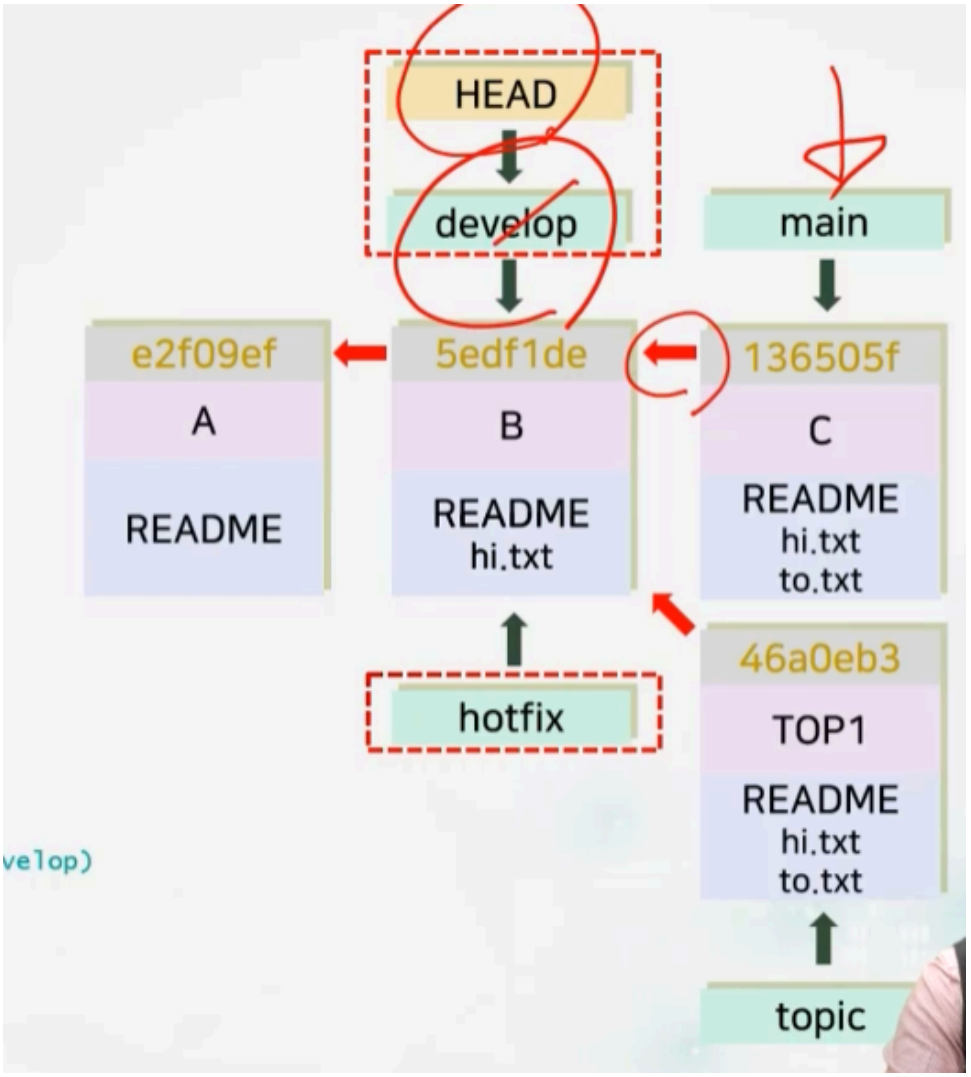
- 브랜치에서 벗어난다면 detached HEAD 상태가 되는 것

- git branch -d bname: 브랜치 삭제 (대문자로 사용하면 강제 삭제)









git branch —merged : 현재 작업 브랜치를 기준으로 병합된 브랜치 목록 표시

git branch —no-merged : 현재 작업 브랜치를 기준으로 병합되지 않은 브랜치 목록 표시

- 위의 2개의 명령어 뒤에다가 브랜치 이름을 붙이면 그 브랜치 이름의 기준으로 각 기능에 맞춰 목록이 표시가 됨

16차시

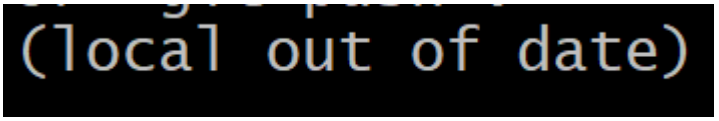
- git clone [복사된-주소]:원격 저장소(깃 허브)와 동일한 이름으로 복제
- git clone [복사된-주소][새로운-폴더명]:새로운 폴더명으로 복제
- git clone [복사된-주소]. : 현재 폴더로 바로 복제
- git remote: 원격 저장소 목록이 나옴 (origin이라는 이름이 나옴
- git remote -v: 원격 저장소의 주소가 같이 나옴

```
PC@DESKTOP-482NOAB MINGW64 /c/[git tutorial]/git-clone (main)
$ git remote -v
origin  https://github.com/ai7dnn/git-clone.git (fetch)
origin  https://github.com/ai7dnn/git-clone.git (push)
```

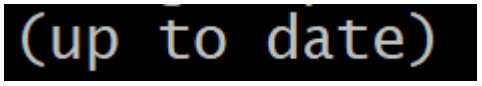
- git remote add origitn URL: 원격 저장소 별칭 저장
- git remote show origin: 저장소의 자세한 정보 (모든 브랜치 보기 가능)
- git remote rename origin org : 원격 저장소 이름 다시 짓기
- git remote rm org: 삭제

> **git clone을 이용해 원격 저장소를 연동시켰을 경우 원격 저장소와 관련된 명령어는 git remote를 이용한다 (설정은 git config를 이용하는 것처럼)**

- local out of date가 show 했을 때 보이는 경우 지역 저장소와 원격 저장소의 내용이 다르다는 걸 알 수 있음



- up to date는 원격 저장소와 지역 저장소의 내용이 같다는 의미



- 저 핑크색이 원격 저장소 파일의 현재 상태 저렇게 브랜치가 앞에 있다면 fast-forwardable이라고 지역 저장소가 현재 앞서 나가있다라는 게 출력됨

```
승 세 훈 @DESKTOP-3J3F60S MINGW64 ~/[git-master]/repo-sync (main)
$ git log --oneline
2e36cc5 (HEAD -> main) add client.md
f8d4fef (origin/main, origin/HEAD) Update README.md
b1b723d A
403f2dc Initial commit
```

```
Local ref configured for 'git push':
  main pushes to main (fast-forwardable)
```

17차시

- 프로젝트 중 에러 나오면 계정마다 토큰 받아서 접근 권한을 받아야 함

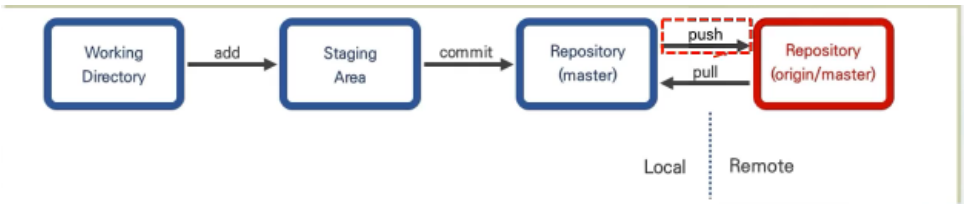
- push:밀기 (지역저장소에서 원격저장소로 미는 것)
- pull:끌기 (원격저장소에서 지역저장소로 당기는 것)

> 풀은 오류가 없는데 푸쉬는 오류가 많이 생김 - 푸쉬는 원격 저장소에 소스를 올리는 거기 때문에 권한이 없거나 로그인이 안 되어있으면 오류남

- git push -u https://{token}@github.com/{username}/{repo_name}.git

```
파이썬@DESKTOP-8TN3J1L MINGW64 ~/git-clone (main)
$ git push -u https://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@github.com/ai7dnn/git-clone.git
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 240 bytes | 240.00 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/ai7dnn/git-clone.git
   1470320..358adce  main -> main
branch 'main' set up to track
'https://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@github.com/ai7dnn/git-clone.git/main'.
```

- git push <저장소별칭명> <브랜치명>: 지역 저장소에 있는 코드 변경 이력이 원격 저장소로 전송됨
- 내 저장소나 내가 협업자로 등록되어 있어야지만 가능하다



- git config --global push.default current: 이걸 한 번 설정하면 이후 같은 브랜치와 저장소에 push가 진행됨 (바꿀거면 이름을 다시 지정해주면 ok)

- git pull: fetch 명령과 merge 명령을 합해 순차적으로 진행하는 것
- fetch는 원격 저장소의 정보를 로컬 저장소로 가져오는 명령 (origin/master라는 브랜치가 새로 생김)
- merge는 변경된 정보를 로컬 저장소의 내용과 병합
- 패치만 하면 내 지역저장소에는 변경은 안 되는데 merge를 해야 내 지역저장소 브랜치에 병합이 됨

fetch: 원격 저장소에서 지역 저장소로 소스를 가져올 뿐 병합을 하지는 않음

- git fetch origin - 원격 저장소를 pull 하는 것은 원격저장소 브랜치가 새로 생기는 건데 fetch는 이 원격 저장소 브랜치에 새롭게 커밋은 되지만 log로 안 보이게 숨겨져 있는 거 (log로 보면 최근 커밋의 위에 있음)

이후 git merge origin/main을 사용하면 병합이 됨 (뒤에 브랜치 이름을 붙여주는 이유는 원격 저장소 브랜치가 여러 개가 되면 혼선이 생길 수 있기 때문)