Git 입문 강의 교재

---

1. Git이란?

버전 관리 시스템(Version Control System)

- 여러 개의 파일이 있을 때, 언제 무엇이 어떻게 바뀌었는지를 기록하고 관리하는 시스템
- 실수했을 때 과거 상태로 되돌아갈 수 있고, 여러 사람이 동시에 작업해도 충돌 없이 관리 가능

Git의 특징

- 분산형 시스템: 인터넷 없이도 로컬에서 기록 관리 가능
- 빠르고 효율적: 대규모 프로젝트에도 빠르게 동작
- 협업에 유리: GitHub, GitLab 등을 통해 여러 사람이 동시에 작업 가능
- Git은 소스코드의 변경 이력을 관리하는 버전 관리 시스템이다.
- 분산형 구조로 로컬에서도 모든 이력을 관리할 수 있다.
- GitHub, GitLab 등과 함께 협업에 널리 사용된다.

---

2. Git 설치 및 초기 설정

1. Git 다운로드: https://git-scm.com
2. 설치 후 사용자 정보 설정:

```
git config --global user.name "사용자이름"
git config --global user.email "이메일주소"
```

---

3. Git 기본 흐름

커밋은 작업 내용을 Git에 저장하는 기록이다. Git은 커밋을 통해 파일의 변경 사항을 시간 순으로 쌓아가며 관리한다. 각 커밋은 고유한 ID와 메시지를 가지며, 언제든지 이전 상태로 되돌아갈 수 있게 해준다.

1. Git 저장소 초기화 또는 클론

```
git init      # 새 저장소 초기화
git clone [URL]  # 기존 저장소 복제
```

1. 작업 후 변경 사항 추가

```
git add 파일명 또는 git add .
```

1. 커밋으로 저장

```
git commit -m "설명 메시지"
```

1. 상태 확인과 로그 조회

```
git status
git log
```

---

4. GitHub 연동

1. GitHub에서 새 저장소 생성
2. 원격 저장소 연결

```
git remote add origin https://github.com/사용자명/저장소명.git
```

1. 브랜치 이름 확인 및 변경 (필요 시)

```
git branch -M main
```

1. GitHub에 업로드

```
git push -u origin main
```

1. 이후 작업 동기화

```
git push  # 업로드
git pull  # 다운로드 및 병합
```

---

5. 브랜치 사용법과 실습

브랜치(branch)란?

- 브랜치는 작업을 나누어 실험하거나 개발할 수 있도록 만들어주는 독립적인 공간

- 예: 'main'은 완성된 코드, 'feature'는 새로운 기능 개발용 공간처럼 활용

왜 사용하는가?

- 실수해도 본줄기(main)에 영향 없음
- 팀원이 동시에 다른 작업 가능

브랜치 관련 주요 개념

- `main`: 기본이 되는 브랜치 (예전엔 master)
- `HEAD`: 현재 내가 위치한 브랜치를 가리키는 포인터
- `merge`: 다른 브랜치의 내용을 현재 브랜치에 합치는 것
- `conflict`: 병합할 때 서로 다른 내용이 겹치면 충돌 발생
- 브랜치는 독립적인 작업 공간이다.

```
git branch feature   # 브랜치 생성
git switch feature   # 브랜치 이동
```

작업 후 커밋하고 병합:

```
git add .
git commit -m "작업 내용"
git switch main
git merge feature
```

브랜치 삭제:

```
git branch -d feature
```

---

6. 실전 예제: GitHub에 처음 프로젝트 올리기

1. 폴더 만들고 이동

```
mkdir my-project
cd my-project
```

1. 초기화 및 파일 생성 → 커밋

```
git init
메모장으로 hello.txt 파일 작성
```

```
git add .
git commit -m "처음 커밋"
```

1. GitHub 저장소 연결 및 업로드

```
git remote add origin https://github.com/사용자명/저장소명.git
git branch -M main
git push -u origin main
```

---

7. 파일 버전 되돌리기

명령어 정리

```
git log                 # 커밋 내역 확인
git checkout 커밋ID -- 파일명     # 특정 파일을 특정 시점으로 복원
git restore 파일명           # 최근 커밋 상태로 복원
git reset --hard 커밋ID        # 전체 프로젝트를 특정 시점으로 복원
```

실습

1. 파일 작성 및 커밋
2. 수정 후 다시 커밋
3. `git log`로 커밋 ID 확인 후 복원

---

8. 고급 기능 실습

git reset이란?

`git reset`은 **지금까지 저장한 커밋 기록을 되돌리는 강력한 명령어**입니다.

- `--soft`: 커밋만 취소하고 변경 내용은 그대로 둔다
- `--mixed`: 커밋 + 스테이징 영역도 취소 (기본값)
- `--hard`: 커밋 + 스테이징 + 실제 파일까지 이전 상태로 돌린다

쉽게 말하면:

- `soft`는 "되돌리지만, 작업 내용은 보존한다"
- `hard`는 "되돌리고, 파일도 그때 시점으로 싹 바꾼다"

주의: `--hard`는 복구가 어려우므로 꼭 필요할 때만 사용!

예시

```
git reset --soft HEAD^   # 마지막 커밋 취소 (작업 내용 유지)
git reset --hard HEAD~2  # 두 커밋 전으로 완전 되돌림
```

---

git stash란?

`git stash`는 **작업 중인 변경사항을 임시로 저장하는 기능**입니다.

- 예를 들어: 다른 브랜치로 이동하고 싶은데 변경사항이 남아 있으면 Git이 막음 → 그럴 땐 `stash`로 잠깐 저장!

실습 예시

```
git stash       # 변경 내용 임시 저장
git switch 다른브랜치
작업 마치고 복귀:
git switch 원래브랜치
git stash apply    # 다시 꺼내오기
git stash drop     # 적용 후 삭제
```

정리

- `stash`는 저장
- `apply`는 다시 꺼냄
- `drop`은 삭제
- `list`로 저장된 stash 목록 확인 가능

---

git revert 개념 보충: 왜 충돌이 발생할까?

`git revert`는 특정 커밋의 변경 내용을 "거꾸로 적용"해서 새로운 커밋을 만드는 명령어입니다.

그런데 되돌리려는 커밋이 예전에 만든 것이라면, 현재의 파일 상태와 그 커밋 당시의 상태가 달라져 있을 수 있습니다. 이럴 때 Git은 어떤 내용을 살리고 어떤 것을 없애야 할지 판단하지 못해 **충돌(conflict)** 이 발생합니다.

예시:

- `b.txt` 파일을 예전에 수정한 커밋을 되돌리려고 하는데
- 현재 브랜치에서는 `b.txt`가 이미 삭제된 상태라면? → Git은 "이 파일을 복구해야 하나? 아니면 그대로 삭제 상태를 유지해야 하나?"를 결정 못 함 → 충돌 발생

충돌 해결 방법 요약

1. `git status`로 충돌 파일 확인
2. 해당 파일을 열어 수동으로 내용 수정

3. 수정 후 저장하고 다음 명령 실행:

```
git add 파일명
git revert --continue
```

1. revert 취소하고 싶으면:

```
git revert --abort
```

.gitignore

```
echo "*.log" > .gitignore
git add .gitignore
git commit -m ".gitignore 설정"
```

git revert

```
git log
git revert [커밋ID]   # 되돌리기 커밋 생성
```

git reset

```
git reset --soft HEAD^  # 마지막 커밋만 취소
```

git stash

```
git stash
작업 전환 후 복귀: git stash apply
```

---

9. HEAD와 브랜치 상태 이해하기

detached HEAD란?

`git checkout [커밋ID]` 명령을 사용하면 브랜치가 아닌 특정 커밋 위로 이동하게 되고, 이 상태를 **detached HEAD**라고 부른다.

즉, HEAD가 브랜치를 가리키는 대신 **직접 커밋을 가리키는 상태**가 됨.

이 상태에서는 커밋은 만들 수 있지만, 브랜치에 연결되어 있지 않기 때문에 **나중에 브랜치를 바꾸면 이 커밋은 사라질 수도 있음.**

☞ 해결 방법

1. **브랜치를 만들어서 연결**

```
git checkout -b 새브랜치이름
```

→ 이렇게 하면 detached HEAD 상태에서 만든 커밋이 새 브랜치로 연결됨

1. **원래 브랜치로 돌아가기만 하고 싶다면**

```
git switch main
```

🔍 예시 흐름

```
git log             # 커밋 ID 확인
git checkout abc1234      # 특정 커밋으로 이동 → detached HEAD 상태
수정 후 커밋 (원하면)
git checkout -b temp-recovery # 안전하게 브랜치로 만들기
```

HEAD란?

- 현재 작업 중인 브랜치를 가리키는 Git 내부 포인터
- `HEAD -> main` 이라면 지금 main 브랜치 위에서 작업 중이라는 뜻

origin/main이란?

- 원격 저장소(origin)에 있는 main 브랜치 상태
- `git pull` 로 로컬과 동기화하거나, `git push` 로 로컬 변경을 원격에 반영

도식 예시

```
[원격 저장소 - origin/main]
          ↓
[로컬 브랜치 - main] ← HEAD
```

요약

- HEAD: 현재 브랜치 위치 표시
- main: 로컬에서 작업 중인 브랜치
- origin/main: GitHub(원격)에 있는 브랜치 상태
- `HEAD -> main, origin/main` : 로컬과 원격이 동기화된 상태

```
[원격 저장소 - origin/main]
          ↓
[로컬 브랜치 - main] ← HEAD
```

- HEAD는 현재 브랜치를 가리킴
- origin/main은 GitHub의 브랜치 상태를 의미함

📚 부록

자주 사용하는 명령어 요약

- git init, git clone, git status, git add, git commit
- git push, git pull, git branch, git switch, git merge
- git log, git revert, git reset, git stash

도움말 보기

```
git help [명령어]
```

이 교재는 Git을 처음 배우는 수강생이 개념과 실습을 함께 익힐 수 있도록 구성됨