

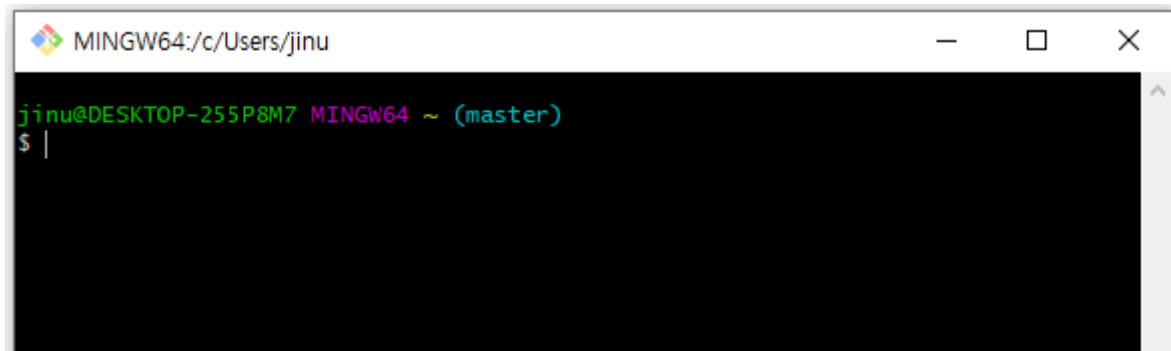


git 사용법

1 Git Bash

❖ Git Bash 실행 및 CLI 기본 명령어 파악하기

- 윈도우의 시작 버튼을 클릭해서 Git Bash를 찾아서 실행하면 아래와 같은 CLI 명령을 입력할 수 있는 창이 나옵니다.



```
MINGW64:/c/Users/jinu  
jinu@DESKTOP-255P8M7 MINGW64 ~ (master)  
$ |
```

- \$ 기호와 윗줄에 표시된 경로 등을 합쳐서 프롬프트라고 합니다.
- 프롬프트는 CLI에서 가장 기본적인 정보를 보여줍니다.
- 'jinu'는 내 컴퓨터의 사용자 아이디, 'DESKTOP-255P8M7'은 현재 PC 이름. '~'는 현재 폴더 위치입니다.
- 기본적으로 Git Bash를 시작하면 현재 폴더는 사용자의 홈 폴더에서 시작합니다.
- 홈 폴더의 전체 경로는 윈도우 10 기준으로 'c:\Users\사용자ID'가 되는데 이를 줄여서 '~'로 나타내는 것입니다.

1 Git Bash

❖ Git Bash 실행 및 CLI 기본 명령어 파악하기

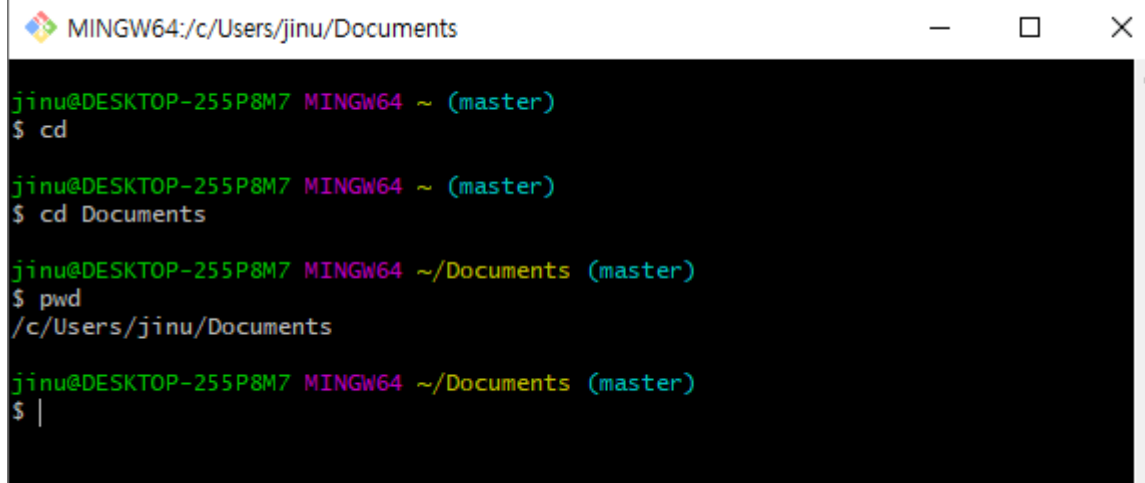
- 본격적인 Git 명령어를 시작하기 전에 우선 Git Bash에서 사용할 기본 명령, 즉 폴더를 만들거나 위치를 이동하는 방법 등을 소개합니다.
- 지금 당장 필요한 명령 몇 가지만 정리한 것이므로 관련하여 더 자세히 공부하고 싶다면 ‘리눅스 명령어 공부하기’와 같은 키워드로 검색하거나 관련 도서를 찾아보길 바랍니다.

명령	설명
pwd	현재 폴더의 위치를 확인합니다.
ls -a	현재 폴더의 파일 목록을 확인합니다. -a 옵션을 이용해 숨김 파일도 볼 수 있습니다.
cd	홈 폴더로 이동합니다. 홈 폴더는 사용자 이름과 폴더명이 같고 내 문서 폴더의 상위 폴더입니다.
cd	<폴더이름> 특정 위치의 디렉토리로 이동합니다.
cd../	현재 폴더의 상위 폴더로 이동합니다.
mkdir	<새폴더이름> 현재 폴더의 아래에 새로운 폴더를 만듭니다.
echo "Hello Git"	메아리라는 뜻 화면에 “ ” 안의 문장인 “ Hello Git ” 을 표시합니다.

1 Git Bash

❖ Git 로컬저장소 생성하기

- Git Bash를 이용해 로컬저장소를 만들고 Git 프로젝트를 시작해 봅시다.
- Git Bash를 실행한 후 다음과 같이 명령어를 입력하여 [내 문서] 폴더로 이동합니다.
- 참고로 Git Bash에서는 폴더명의 일부를 입력하고 [tab]을 누르면 자동완성 기능을 사용할 수 있습니다.



```
MINGW64:/c/Users/jinu/Documents
jinu@DESKTOP-255P8M7 MINGW64 ~ (master)
$ cd

jinu@DESKTOP-255P8M7 MINGW64 ~ (master)
$ cd Documents

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents (master)
$ pwd
/c/Users/jinu/Documents

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents (master)
$ |
```

1 Git Bash

❖ Git 로컬저장소 생성하기

- 사실 처음 입력한 `cd` 명령은 입력하지 않아도 됩니다.
- `cd` 명령은 홈 폴더로 이동하는 명령인데 처음 `Git Bash`의 기본 시작 폴더가 홈 폴더이기 때문이죠.
- 기타 실수를 방지하기 위해서 추가한 것입니다.
- `cd Documents` 명령이 끝난 이후에 확인을 위해 `pwd` 명령도 수행했습니다.
- 현재 폴더 확인은 프롬프트를 통해서 확인할 수 있지만 좀 더 확실하기 위해 `pwd` 명령을 사용했습니다.
- 그만큼 `CLI`에서는 꼼꼼한 확인이 매우 중요합니다.

1 Git Bash

❖ Git 로컬저장소 생성하기

- 이어서 Git 로컬저장소를 위한 폴더를 만들고 이동한 후 `git status` 명령을 실행해 봅니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents (master)
$ mkdir hello-git-cli

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents (master)
$ cd hello-git-cli

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ pwd
/c/Users/jinu/Documents/hello-git-cli

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$
```

- 위 명령으로 아래 경로에 `[hello-git-cli]` 폴더가 생성됩니다.
내 컴퓨터>Documents>hello-git-cli

1 Git Bash

❖ Git 로컬저장소 생성하기

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git status
warning: could not open directory 'Application Data/': Permission denied
warning: could not open directory 'Cookies/': Permission denied
warning: could not open directory 'Local Settings/': Permission denied
warning: could not open directory 'My Documents/': Permission denied
warning: could not open directory 'NetHood/': Permission denied
warning: could not open directory 'PrintHood/': Permission denied
warning: could not open directory 'Recent/': Permission denied
warning: could not open directory 'SendTo/': Permission denied
warning: could not open directory 'Templates/': Permission denied
warning: could not open directory '시작 메뉴 /': Permission denied
On branch master

No commits yet
```

- 반드시 익혀야 하는 명령이 나왔습니다.
- **git status**은 **Git** 저장소의 상태를 알려주는 명령으로 자주 사용합니다.
- 그런데 막상 **git status** 명령을 실행하면 에러가 발생합니다.
- 에러 메시지를 보면 **‘.git** 폴더가 없다 (= 현재 디렉토리는 **Git** 저장소가 아니다)’라고 알려줍니다.
- 즉, **git status** 명령은 **Git** 저장소 (정확하게는 워킹트리) 에서만 정상적으로 수행되는 명령입니다.

1 Git Bash

❖ Git 로컬저장소 생성하기

- Git 워킹트리에 대해서는 바로 뒤에 이어서 설명하겠습니다.
- 우선 다음과 같이 **git status** 명령에 대해 정리하고 넘어갑니다.

git status	Git 워킹트리의 상태를 보는 명령으로. 매우 자주 사용합니다. 워킹트리가 아닌 폴더에서 실행하면 오류가 발생합니다.
git status -s	git status 명령 보다 짧게 요약해서 상태를 보여주는 명령으로, 변경된 파일이 많을 때 유용합니다.

1 Git Bash

❖ Git 로컬저장소 생성하기

- 이제 우리가 만든 폴더를 Git 저장소로 만들어 보겠습니다.
- 앞서 생성한 [hello-git-cli] 폴더에서 다음처럼 실행합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git init
Reinitialized existing Git repository in C:/Users/jinu/Dropbox/내 PC (DESKTOP-255P8M7)/Documents/hello-git-cli/.git/

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ ls -a
./ ../ .git/

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$
```

- 위 실습에서 새로운 명령을 사용했습니다.
- **git init** 명령은 현재 폴더에 Git 저장소를 생성합니다.

1 Git Bash

❖ Git 로컬저장소 생성하기

- 명령의 결과는 '비어 있는 Git 저장소를 .git 폴더에 만들었다'라는 내용입니다.
- 그리고 `ls -a` 명령으로 현재 폴더 내 파일 목록을 확인해 보면 [`.git`]이라는 폴더가 생긴 걸 알 수 있습니다.
- 이 폴더가 Git의 로컬저장소입니다.
- 또한 앞서 실습에서 에러가 발생한 `git status` 명령도 정상 동작되었습니다.
- 이로써 이 폴더가 이제 Git 저장소가 되었다는 걸 알 수 있습니다.
- `git init` 명령어 전후에 프롬프트의 변화도 확인했나요?
- 다시 한번 위 코드를 살펴보세요.
- '(master)' 표시를 확인할 수 있습니다.
- 지금까지 실습으로 기억해야 하는 중요한 개념이 하나 더 있습니다.
- 로컬저장소가 있는 현재 폴더, 다시 말해서 일반적인 작업 폴더를 Git 용어로 뭐라고 할까요?
- Git에서는 작업 폴더를 '워킹트리'라고 합니다.
- 이 단어도 꼭 기억해 주세요.

1 Git Bash

❖ Git 로컬저장소 생성하기

- Git을 처음 배우면 몇 가지 혼동되는 용어가 있습니다.
- 별 것 아닌 것 같지만 용어의 혼동이 Git의 개념을 이해하는데 큰 오해를 불러일으키기 때문에 정확히 정리하고 넘어갑시다.

워킹트리	일반적인 작업이 일어나는 곳
로컬저장소	.git 폴더. 커밋은 여기에 들어 있다.
작업 폴더	워킹트리 + 로컬저장소
Git 저장소	엄밀하게는 로컬저장소를 의미하지만 넓은 의미로 작업 폴더를 의미하기도 한다.

1 Git Bash

❖ 옵션 설정하기

- Git을 사용하기 위해서 해야 할 일이 더 있습니다.
- **git config** 명령을 사용해서 **Git** 옵션 설정을 해야 합니다.

<code>git config --global <옵션명></code>	지정한 전역 옵션의 내용을 살펴봅니다.
<code>git config --global <옵션명> <새로운 값></code>	지정한 전역 옵션의 값을 새로 설정합니다.
<code>git config --global --unset <옵션명></code>	지정한 전역 옵션을 삭제합니다.
<code>git config --local <옵션명></code>	지정한 지역 옵션의 내용을 살펴봅니다.
<code>git config --local <옵션명> <새로운 값></code>	지정한 지역 옵션의 값을 새로 설정합니다.
<code>git config --local --unset <옵션명></code>	지정한 지역 옵션의 값을 삭제합니다.
<code>git config --system <옵션명></code>	지정한 시스템 옵션의 내용을 살펴봅니다.
<code>git config --system <옵션명> <값></code>	지정한 시스템 옵션의 값을 새로 설정합니다.
<code>git config --system --unset <옵션명> <값></code>	지정한 시스템 옵션의 값을 삭제합니다.
<code>git config --list</code>	현재 프로젝트의 모든 옵션을 살펴봅니다.

1 Git Bash

❖ 옵션 설정하기

- `git config` 명령으로는 옵션을 보거나, 값을 바꿀 수 있습니다.
- Git의 옵션에는 지역 옵션과 전역 옵션, 시스템 환경 옵션의 세 종류가 있습니다.
- 시스템 환경 옵션은 PC 전체의 사용자를 위한 옵션, 전역 옵션은 현재 사용자를 위한 옵션이고, 지역 옵션은 현재 Git 저장소에서만 유효한 옵션입니다.
- 우선순위는 지역 옵션>전역 옵션>시스템 옵션 순으로 지역 옵션이 가장 높습니다.
- 일반적으로 개인 PC에서는 전역 옵션을 많이 사용하는데, 공용 PC처럼 여러 사람이 사용하거나 Git을 잠깐만 써야 할 일이 있다면 지역 옵션을 사용해야 합니다.
- 시스템 옵션은 Git이나 소스트리 설치 시에 몇 가지 값들이 지정되는데 직접 수정하는 일은 그리 많지 않습니다.

1 Git Bash

❖ 옵션 설정하기

- 옵션 값을 이용해서 여러 가지 설정이 가능한데 지금은 필수적인 값인 `user.name`(사용자 이름), `user.email`(이메일), `core.editor`(기본 에디터) 세 옵션의 값을 입력해 보겠습니다.
- 먼저 이름만 바꿔 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git config --global user.name

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git config --global user.name "jinu"

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git config --global user.name
jinu

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ |
```

- 위 예시는 전역 변수인 `user.name`의 값을 `jinu`로 설정한 것입니다.
- 같은 방식으로 `user.email` 값도 변경합니다.
- 앞서 말했듯이 작업 PC가 공용이거나 프로젝트마다 값을 따로 설정하고 싶을 경우 `--global` 옵션을 빼고 지역 옵션을 설정하면 됩니다.

1 Git Bash

❖ 옵션 설정하기

- 중요한 설정이 하나 더 남아 있습니다.
- CLI를 사용하면 텍스트 에디터를 쓸 일이 생기는데, 현재 Git Bash의 기본 에디터는 보통 리눅스 운영체제에서 주로 쓰는 vim이나 nano로 설정되어 있습니다.
- vim을 잘 사용하면 그대로 뒤도 되지만 그렇지 않다면(또는 vim이 뭔지 모를 경우) 기본 에디터를 비주얼 스튜디오 코드(Visual studio Code)로 변경하는 것이 좋습니다.
- 만약 이미 비주얼 스튜디오 코드가 기본 에디터로 되어 있다면 그대로 두면 됩니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git config core.editor

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git config --global core.editor

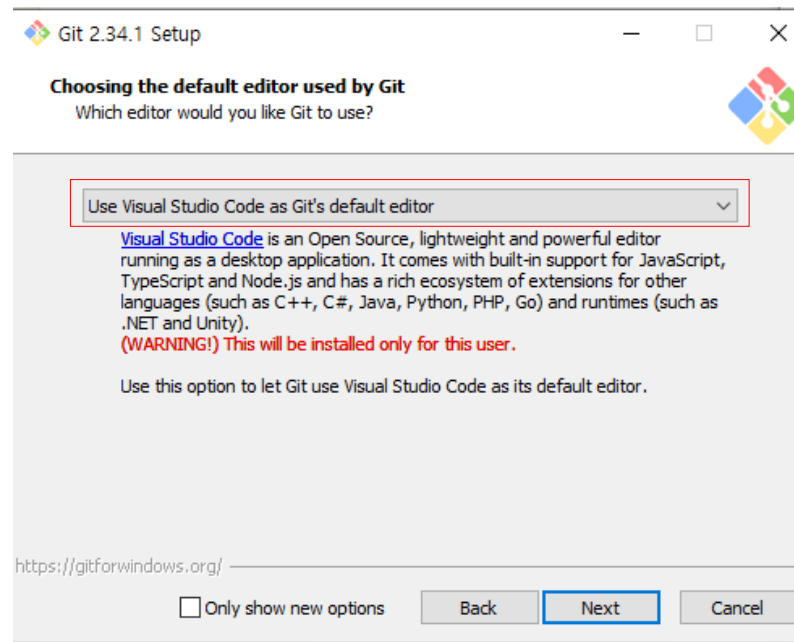
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git config --system core.editor

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ |
```

1 Git Bash

❖ 옵션 설정하기

- `git config` 명령을 이용해서 변경해도 되는데 여러분이 실수할 가능성이 많아서 `git-scm.com`에서 다시 `git`을 다운받아서 재설치하는 것을 권장합니다.
- 재설치를 위해서는 일단 작업 중이던 `Git bash` 창을 모두 닫고 재설치 과정을 진행합니다.
- 재설치 과정에서 기본 에디터 선택이 나오는데 아래 그림처럼 기본 에디터를 비주얼 스튜디오 코드를 선택해야 합니다.



1 Git Bash

❖ 옵션 설정하기

- 재설치가 완료된 후 다시 한 번 Git 환경 변수를 살펴보겠습니다.
- 정상적으로 비주얼 스튜디오 코드로 기본 에디터가 변경된 것을 알 수 있습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~ (master)
$ git config --system core.editor

jinu@DESKTOP-255P8M7 MINGW64 ~ (master)
$ git config --global core.editor
"C:\Users\jinu\AppData\Local\Programs\Microsoft VS Code\bin\code" --wait

jinu@DESKTOP-255P8M7 MINGW64 ~ (master)
$
```

2 기본 CLI 명령어 살펴보기

❖ 스테이징과 커밋을 수행하는 `add`, `commit`

- 먼저 명령에 대해 간단히 복습하고, 실습해 보겠습니다.

<code>git add 파일1 파일2 ...</code>	파일들을 스테이지에 추가합니다. 새로 생성한 파일을 스테이지에 추가하고 싶다면 반드시 <code>add</code> 명령을 사용합니다.
<code>git commit</code>	스테이지에 있는 파일들을 커밋합니다.
<code>git commit -a</code>	<code>add</code> 명령을 생략하고 바로 커밋하고 싶을 때 사용합니다. 변경된 파일과 삭제된 파일은 자동으로 스테이징되고 커밋됩니다. 주의할 점은 <code>untracked</code> 파일은 커밋되지 않는다는 것입니다.
<code>git push [-u] [원격저장소별명] [브랜치이름]</code>	현재 브랜치에서 새로 생성한 커밋들을 원격저장소에 업로드합니다. <code>-u</code> 옵션으로 브랜치의 업스트림을 등록할 수 있습니다. 한 번 등록한 후에는 <code>git push</code> 만 입력해도 됩니다.
<code>git pull</code>	원격 저장소의 변경사항을 워킹트리에 반영합니다. 사실은 <code>git fetch + git merge</code> 명령입니다.
<code>git fetch [원격저장소별명] [브랜치이름]</code>	원격 저장소의 브랜치와 커밋들을 로컬저장소와 동기화합니다. 옵션을 생략하면 모든 원격저장소에서 모든 브랜치를 가져옵니다.
<code>git merge 브랜치이름</code>	지정한 브랜치의 커밋들을 현재 브랜치 및 워킹트리에 반영합니다.

2 기본 CLI 명령어 살펴보기

❖ 스테이징과 커밋을 수행하는 `add`, `commit`

- 커밋을 실행하기 위해 먼저 간단한 파일을 만들겠습니다.
- 비주얼 스튜디오 코드를 사용해도 되지만 번거로우니 **CLI**에서 바로 작업하겠습니다.
- 먼저 `echo` 명령을 이용해서 `file1.txt` 파일을 하나 만듭니다.
- **CLI**가 익숙하지 않다면 탐색기로 작업 폴더를 열어서 확인해 보는 것도 좋은 방법입니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ echo "hello git"
hello git

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ echo "hello git" > file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ ls
file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file1.txt
```

2 기본 CLI 명령어 살펴보기

❖ 스테이징과 커밋을 수행하는 `add`, `commit`

- `git status` 명령으로 상태를 살펴보면 `file1.txt` 라는 파일이 생성되었고 `untracked` 상태임을 확인할 수 있습니다.
- 변경 내용을 스테이지에 추가해 보겠습니다.
- 스테이지에 추가하는 명령은 `add`입니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git stauts
git: 'stauts' is not a git command. See 'git --help'.

The most similar command is
    status

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   file1.txt
```

2 기본 CLI 명령어 살펴보기

❖ 스테이징과 커밋을 수행하는 `add`, `commit`

- 우리가 만든 `file1.txt` 파일이 스테이지 영역에 추가된 것을 확인할 수 있습니다.
- `git add <file>...` 명령을 사용하면 커밋에 포함할 수 있다는 내용을 볼 수 있습니다.
- 참고로 ...의 의미는 한 번에 여러 파일 이름을 지정할 수도 있다는 뜻입니다.

2 기본 CLI 명령어 살펴보기

❖ **reset** 명령으로 스테이징 취소하기

- 위 실행결과와 아래 보면 `git rm --cached <file>...` 명령으로 스테이지에서 내릴 수 있다(unstage)는 메시지가 있습니다.
- 그런데 스테이지에서 내리기 위해 저 명령보다 자주 사용하는 명령이 있습니다.
- **git reset** 명령인데요, 이것을 사용하면 더 쉽게 파일을 스테이지에서 내릴 수 있습니다.
- 이 명령은 워킹트리의 내용은 그대로 두고 해당 파일을 스테이지에서 만 내립니다.
- 리셋에는 세 가지 옵션(soft, mixed, hard)을 사용할 수 있습니다.
- 지금처럼 옵션 없이 사용하면 **mixed reset**으로 동작합니다.
- 다른 옵션에 대해서는 나중에 다시 살펴 보겠습니다.
- 그리고, 스테이지에서 내리는 작업을 ‘언스테이징’이라고 합니다.
- 언스테이징이라는 단어도 기억해 주세요.

2 기본 CLI 명령어 살펴보기

❖ reset 명령으로 스테이징 취소하기

- 스테이지에서 파일 언스테이징하기

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git reset file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ ls
file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ cat file1.txt
hello git
```

- 위 실습은 file1.txt를 git reset 명령으로 언스테이징하고, cat 명령으로 파일 내용이 변경되었는지 확인하는 것입니다.
- 예제에서 보는 것처럼 파일의 내용은 그대로 두고 단지 언스테이징만을 할 것을 알 수 있습니다.

2 기본 CLI 명령어 살펴보기

❖ CLI로 첫 번째 커밋 생성

- 이제 커밋을 실행해 보겠습니다.
- 좀 전에 언스테이징을 했으므로 다시 `git add` 명령을 실행한 후 커밋합니다.
- 커밋은 `git commit` 명령으로 수행합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git commit
[master (root-commit) df4ff39] 첫 번째 커밋
1 file changed, 1 insertion(+)
create mode 100644 file1.txt
```


2 기본 CLI 명령어 살펴보기

❖ CLI로 첫 번째 커밋 생성

- 위 실습에서 `git commit` 명령을 실행하면 다음과 같이 비주얼 스튜디오 코드가 열립니다.
- 여기서 커밋 메시지를 잘 적고, `[ctrl + s]`를 눌러 저장한 후 비주얼 스튜디오 코드를 닫습니다.

```
COMMIT_EDITMSG X
C: > Users > jinu > Dropbox > 내 PC (DESKTOP-255P8M7) > Documents > hello-git-cli > .git > COMMIT_EDITMSG
1  첫 번째 커밋
2
3  간단하게 hello git이라고 쓴 내용을 커밋했다.
4  # Please enter the commit message for your changes. Lines starting
5  # with '#' will be ignored, and an empty message aborts the commit.
6  #
7  # On branch master
8  #
9  # Initial commit
10 #
11 # Changes to be committed:
12 #   new file:   file1.txt
13 #
14
```

2 기본 CLI 명령어 살펴보기

❖ CLI로 첫 번째 커밋 생성

- 그럼 소스트리처럼 커밋이 생성됩니다.
- 어렵지 않죠?
- 이때 첫 줄과 둘째 줄 사이는 반드시 한 줄 비워야 합니다.
- 그리고 첫 줄에는 작업 내용의 요약, 다음 줄에는 자세하게 작업 내용을 기록합니다.
- 첫 줄은 제목이고 그 다음 줄은 본문이라고 생각하면 됩니다.
- 로그를 볼 때나 **GitHub**의 **Pull Request** 메뉴 등에서 이 규칙을 활용해서 내용을 자동으로 구성하기 때문에 꼭 지키는 것이 좋습니다.
- 만약 **git commit** 명령을 실행한 후 갑작스런 변심 등의 이유로 커밋을 하고 싶지 않다면 비주얼 스튜디오 코드에서 아무 것도 추가하지 않고 **[X]** 버튼을 눌러 종료합니다.
- 그럼 커밋도 자동으로 취소됩니다.

2 기본 CLI 명령어 살펴보기

❖ CLI로 첫 번째 커밋 생성

- 성공적으로 커밋을 완료했다면 `git status` 명령을 실행해 보세요.
- 워킹트리와 스테이지 영역이 깨끗해진 걸 확인할 수 있을 것입니다.
- 성공적으로 커밋을 완료하면 그 커밋 시점의 파일 상태로 언제라도 복구할 수 있습니다.
- 그리고 커밋은 절대 사라지지 않습니다.

2 기본 CLI 명령어 살펴보기

❖ CLI로 log 살펴보기

- `git log` 명령으로 `git`의 커밋 히스토리를 확인해 봅니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git status
On branch master
nothing to commit, working tree clean

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git log --branches --decorate --graph -online
fatal: unrecognized argument: -online

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git log --branches --decorate --graph --online
* df4ff39 (HEAD -> master) 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ |
```

- 커밋 히스토리에 보이는 앞의 16진수 7자리 숫자는 커밋 체크섬 혹은 커밋 아이디입니다.
- **SHA1** 해시 체크섬 값을 사용하는데, 전 세계에서 유일한 값을 가지므로 여러분은 저와 다른 값이 나올 것입니다.
- 실제로 커밋 체크섬은 40자리인데 앞의 7자리만 화면에 보여줍니다.

2 기본 CLI 명령어 살펴보기

❖ CLI로 log 살펴보기

- 보통 `git log` 명령은 옵션 없이 써도 되지만 위 실습에서는 긴 옵션으로 사용했습니다.
- 이는 개인적인 성향으로, 위와 같이 길게 옵션을 입력하면 예쁘고, 간결한 결과가 출력됩니다.
- 각각의 옵션의 조합에 따라 결과가 달라집니다.
- 개인적으로 자주 사용하는 옵션들은 아래와 같은 데 여러분도 다양한 조합으로 실험하고 익혀 보세요.

<code>git log</code>	HEAD와 관련된 커밋들이 자세하게 나옵니다.
<code>git log --oneline</code>	간단히 커밋 해시와 제목만 보고 싶을 때
<code>git log --graph --decorate</code>	HEAD와 관련된 커밋들을 조금 더 자세히 보고 싶을 때
<code>git log --oneline --graph --all --decorate</code>	모든 브랜치들을 보고 싶을 때 사용하는 명령입니다.
<code>git log --oneline -n5</code>	내 브랜치의 최신 커밋을 5개만 보고 싶을 때 사용합니다.

2 기본 CLI 명령어 살펴보기

❖ 좋은 커밋 메시지의 7가지 규칙

- 제목과 본문을 빈 줄으로 분리한다.
- 제목은 50자 이내로 쓴다.
- 제목을 영어로 쓸 경우 첫 글자는 대문자로 쓴다.
- 제목에는 마침표를 넣지 않는다.
- 제목을 영어로 쓸 경우 동사원형(현재형)으로 시작한다.
- 본문을 72자 단위로 줄바꿈한다.
- 어떻게 보다 무엇과 왜를 설명한다.

2 기본 CLI 명령어 살펴보기

❖ 도움말 기능 사용하기

- Git에는 각 명령의 도움말을 볼 수 있는 명령이 있습니다.
- 여러분이 모르는 명령이 있거나 그 명령의 자세한 옵션들이 보고 싶을 때에는 **git help** 명령을 사용하면 됩니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git help status

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git help commit

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git help add

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ |
```

git-add(1) Manual Page

NAME

git-add - Add file contents to the index

SYNOPSIS

```
git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
        [--edit | -e] [--[no-]all | --[no-]ignore-removal | [--update | -u]] [--sparse]
        [--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing] [--renormalize]
        [--chmod=(+|-)X] [--pathspec-from-file=<file>] [--pathspec-file-nul]
        [--] [<pathspec>...]
```

2 기본 CLI 명령어 살펴보기

❖ 도움말 기능 사용하기

- 도움말 명령을 수행하면 웹 브라우저가 열리면서 다음과 같이 해당 명령어에 대한 내용이 표시됩니다.
- 해당 명령이 정확히 의미하는 바가 무엇인지 알 수 있고, 사용할 수 있는 옵션이 총망라되어 있습니다.
- 안타깝게도 영어로 표기된다는 흠이 있지만, 그래도 매우 유용한 기능입니다.

3 원격저장소 관련 CLI 명령어

❖ `remote`, `push`, `pull`

- 커밋을 했으니 이제 남은 작업은 원격저장소에 `push`를 보내는 것입니다.
- 그럼 원격저장소 등록을 할 차례겠죠?
- 먼저 `GitHub`에 접속한 후 새로운 프로젝트를 하나 만듭니다.
- 저는 `[hello-git-cli]` 이름으로 만들었습니다.
- 이때 주의할 점이 하나 있는데 아래 그림처럼 옵션을 선택해 비어 있는 프로젝트로 만들어야 한다는 점입니다.
- 이렇게 하면 저장소를 클론해 와도 비어 있는 폴더만 생기고, 최초 커밋을 직접 생성해야 합니다.
- 그렇지 않을 경우에는 이미 생성된 커밋과 우리가 생성할 커밋이 충돌을 발생시 키므로 `push --force` 옵션을 이용해 강제 `push`를 해야 합니다.
- `Push --force`는 사실 좋지 않은 명령입니다.
- `Git` 사용에 익숙해질 때까지는 가급적 자제하는 편이 좋습니다.

3 원격저장소 관련 CLI 명령어



❖ remote, push, pull

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?



[Import a repository.](#)

Owner * Repository name *

 jjin300 / 

Great repository names are short and memorable. Need inspiration? How about [fictional-palm-tree?](#)

Description (optional)

- ☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.
- ☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

- ☐ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)
- ☐ **Add .gitignore**
Choose which files not to track from a list of templates. [Learn more.](#)
- ☐ **Choose a license**
A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

3 원격저장소 관련 CLI 명령어

❖ remote, push, pull

- 원격 저장소를 등록하는 CLI 명령어를 살펴보겠습니다.

<code>git remote add <원격저장소 이름> <원격저장소 주소></code>	원격저장소를 등록합니다. 원격저장소는 여러 개 등록할 수 있지만 같은 별명의 원격저장소는 하나만 가질 수 있습니다. 통상 첫 번째 원격저장소를 origin으로 지정합니다.
<code>git remote -v</code>	원격저장소 목록을 살펴봅니다.

- 프로젝트를 만들면 원격저장소 URL이 표시됩니다.
- 이 URL을 origin이라는 이름으로 등록하고 push를 시도해 봅니다.
- 여러분과 저는 아이디가 다르기 때문에 원격저장소 URL도 다릅니다.
- Git Bash에서 붙여넣기는 일반적으로 사용하는 [ctrl + v]가 아니라 [shift + insert]입니다.

3 원격저장소 관련 CLI 명령어

❖ remote, push, pull

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git remote add origin https://github.com/jjin300/hello-git-cli.git

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git remote -v
origin  https://github.com/jjin300/hello-git-cli.git (fetch)
origin  https://github.com/jjin300/hello-git-cli.git (push)

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ |
```

- 아쉽게도 `git push` 명령이 실패했습니다.
- 에러 메시지를 꼼꼼하게 읽어야 할 타이밍입니다.

3 원격저장소 관련 CLI 명령어

❖ remote, push, pull

- 에러 메시지를 읽어 보면 로컬저장소의 [master] 브랜치와 연결된 원격저장소의 브랜치가 없어서 발생한 오류라는 걸 알 수 있습니다.
- 그리고 업스트림 w.이라는 텍스트가 보이는데, 업스트림 브랜치는 로컬저장소와 연결된 원격저장소를 일컫는 단어입니다.
- 영어로는 상류라는 뜻이니 꽤 적절한 단어라고 생각됩니다.
- 업스트림 브랜치 설정을 위해서 에러 메시지가 알려준 대로 **-set-upstream**을 쓰거나 이 명령의 단축 명령인 **-U** 옵션을 사용합니다.
- 그러면 이후에는 origin 저장소의 [master] 브랜치가 로컬저장소의 [master] 브랜치의 업스트림으로 지정되어 **git push** 명령어만으로도 에러 없이 push가 가능해집니다.
- 참고로 **Git Bash**에서 긴 명령은 대시 두 개 짧은 명령은 대시 한 개로 시작하는 경우가 많으니 이것도 기억해 두세요.

3 원격저장소 관련 CLI 명령어

❖ remote, push, pull

- 이제 업스트림을 지정하면서 push를 다시 시도해 봅니다.
- 만약 인증 관련 정보가 저장되어 있지 않다면 업스트림 지정 및 최초 push를 할 때 팝업 창이 나타나 GitHub 로그인을 요청합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jjin300/hello-git-cli.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git log --oneline -n1
df4ff39 (HEAD -> master, origin/master) 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ git push
Everything up-to-date

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$
```

3 원격저장소 관련 CLI 명령어

❖ remote, push, pull

- `-u` 옵션을 지정해서 `push`가 정상적으로 성공했습니다.
- 그리고 `git log` 명령을 통해 보면 `HEAD`는 `[master]`를 가리키고 있고, `[origin/master]` 브랜치도 생겨난 것을 볼 수 있습니다.
- `HEAD`는 항상 현재 작업 중인 브랜치 혹은 커밋을 가리킵니다.
- 지금 `HEAD`가 가리키는 `[master]`는 로컬의 `[master]` 브랜치이고, `[origin/master]`는 원격저장소인 GitHub의 마스터 브랜치입니다.
- 지금 현재 상태는 `HEAD`, `master`, `origin/master` 모두 똑같은 커밋인 커밋 `df4ff39`를 가리킵니다.
- 마지막으로 `git push` 명령을 한 번 더 수행했는데 이번에는 에러 없이 잘 수행되었습니다.
- 이전에 이미 `-u` 옵션으로 업스트림을 지정했기 때문이겠죠?
- 더 이상 `push`할 게 없었기 때문에 `Everything up-to-date`라는 결과 메시지가 화면에 표시됩니다.

3 원격저장소 관련 CLI 명령어

❖ remote, push, pull

- push와 쌍으로 존재하는 명령인 pull은 더 간단합니다.
- CLI에서는 간단히 `git pull` 명령을 입력하면 됩니다.
- pull에 대해서는 나중에 다시 살펴볼 것입니다.

3 원격저장소 관련 CLI 명령어

❖ clone

- CLI에서는 `git clone` 명령을 이용합니다.
- `git clone` 명령을 사용할 때 한 가지 팁이 있습니다.
- 바로 저장소 주소가 꼭 원격일 필요는 없다는 것입니다.
- 때에 따라 로컬저장소를 클론으로 복제하면 편리하게 사용할 수 있습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ pwd
/c/Users/jinu/documents/hello-git-cli

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli (master)
$ cd ../

jinu@DESKTOP-255P8M7 MINGW64 ~/documents (master)
$ git clone https://github.com/jjin300/hello-git-cli.git
fatal: destination path 'hello-git-cli' already exists and is not an empty
tory.

jinu@DESKTOP-255P8M7 MINGW64 ~/documents (master)
$ |
```

- 위의 실습을 보면 `[hello-git-cli]` 원격저장소를 클론으로 복제하려다 실패했습니다.
- 왜 그럴까요?
- [새로운 폴더명] 옵션을 지정하지 않으면 복제한 프로젝트 이름과 같은 폴더를 만들게 되는데 이미 `[hello-git-cli]`라는 폴더가 있기 때문에 실패한 것입니다.

3 원격저장소 관련 CLI 명령어

❖ clone

- 이번에는 [새로운 폴더명] 옵션을 지정해서 다시 시도해 봅니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents (master)
$ git clone https://github.com/jjin300/hello-git-cli.git hello-git-cli2
Cloning into 'hello-git-cli2'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents (master)
$ cd hello-git-cli2

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ git log --oneline
df4ff39 (HEAD -> master, origin/master, origin/HEAD) 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ git remote -v
origin https://github.com/jjin300/hello-git-cli.git (fetch)
origin https://github.com/jjin300/hello-git-cli.git (push)

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$
```

- 이번에는 `git clone` 명령이 정상적으로 성공했습니다.
- 명령의 결과로 `[hello-git-cli2]` 폴더가 생기고, 그 안에는 `[master]` 브랜치의 최신 커밋으로 체크아웃되었습니다.

3 원격저장소 관련 CLI 명령어

❖ clone

- 이 저장소에서 다시 한번 커밋과 push를 실행해 봅시다.
- 이후 저장소의 상태는 아래와 같습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ echo "second" >> file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ cat file1.txt
hello git
second

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ git commit -a
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory
[master 1e40ead] 두 번째 커밋
1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 273 bytes | 273.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jjin300/hello-git-cli.git
df4ff39..1e40ead master -> master

jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ git log --oneline
1e40ead (HEAD -> master, origin/master, origin/HEAD) 두 번째 커밋
df4ff39 첫 번째 커밋
```

3 원격저장소 관련 CLI 명령어

❖ clone

- `git commit -a` 옵션을 사용하면 기존에 커밋 이력이 있는 파일, 즉 `modified` 상태의 파일의 스테이징 과정을 생략할 수 있습니다.
- 다시 첫 번째 저장소로 돌아가서 `git pull` 명령을 실행해 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/documents/hello-git-cli2 (master)
$ cd ~/Documents/hello-git-cli

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline
df4ff39 (HEAD -> master, origin/master) 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 253 bytes | 8.00 KiB/s, done.
From https://github.com/jjin300/hello-git-cli
   df4ff39..1e40ead master    -> origin/master
Updating df4ff39..1e40ead
Fast-forward
 file1.txt | 1 +
 1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline
1e40ead (HEAD -> master, origin/master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ cat file1.txt
hello git
second
```

3 원격저장소 관련 CLI 명령어

❖ clone

- 위 과정을 보면 일단 처음 생성했던 Git 저장소로 이동한 후 `git pull` 명령을 수행했습니다.
- 나중에 다시 살펴보겠지만 `pull = fetch + merge`라는 사실을 떠올리고 이 장을 마치면 됩니다.
- 다음 장에서는 `branch`와 `merge`, `checkout`에 대해 살펴보겠습니다.
- CLI가 익숙치 않아서 어려움이 느껴진다면 이 장에서 배운 개념을 다시 떠올리면서 실습을 다시 한번 진행해 보세요.



Thank You !