

Git(분산형 버전 관리 시스템)

pykwon

1. Git 개요



분산 버전 관리 시스템 중 하나.

완벽한 분산 환경에서 빠르고 단순하게 수백 수천 개의 동시 다발적인 브랜치 작업을 수행하는 것을 목표로 하는 **버전 관리 시스템**

2005년 리누스 토르발스에 의해서 만들어짐

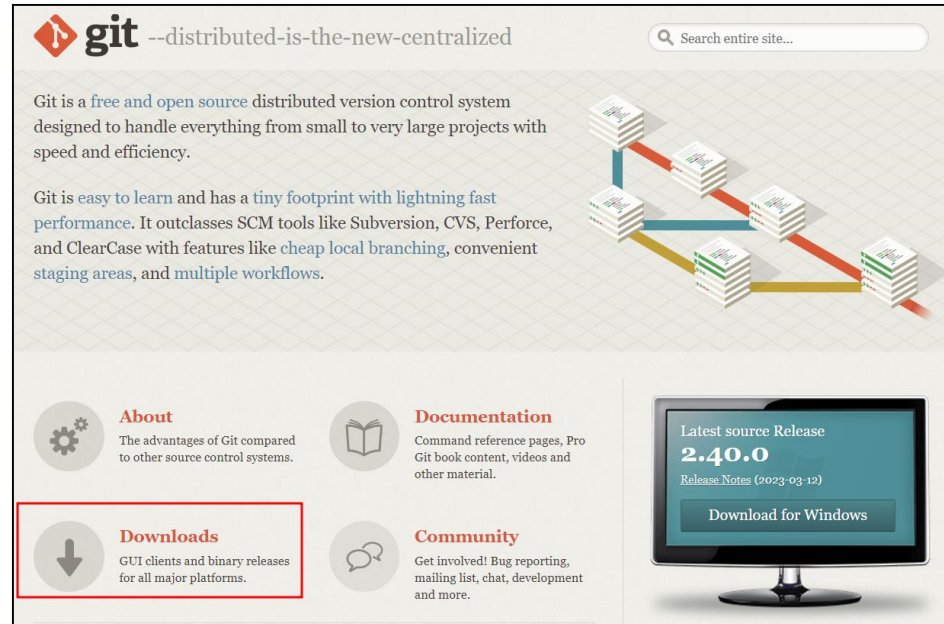
특징

- 로컬 및 원격 저장소 생성
- 로컬 저장소에 파일생성 및 추가
- 수정 내역을 로컬 저장소에 제출
- 파일 수정내역 추적
- 원격 저장소에 제출된 수정 내역을 로컬 저장소에 적용
- master에 영향을 끼치지 않는 브랜치(branch) 생성
- 브랜치 병합(merge)

2. Git 설치

Git 다운로드

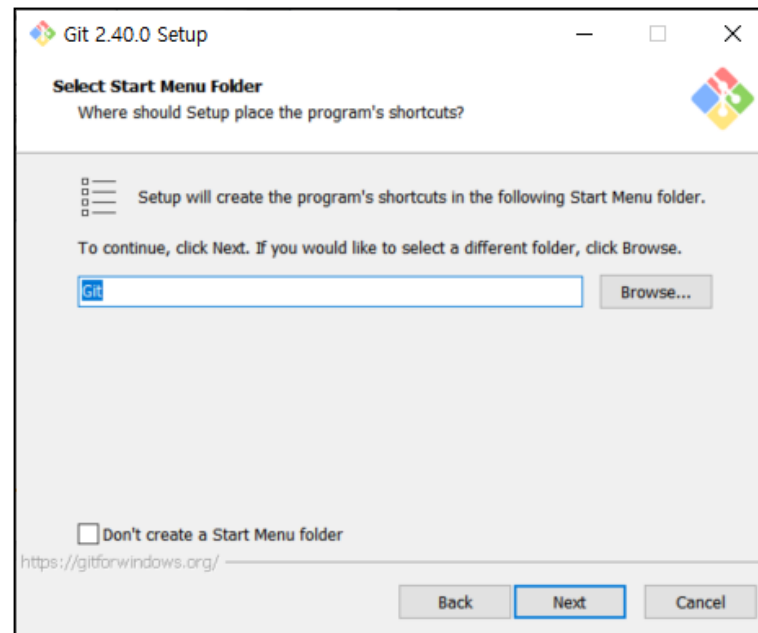
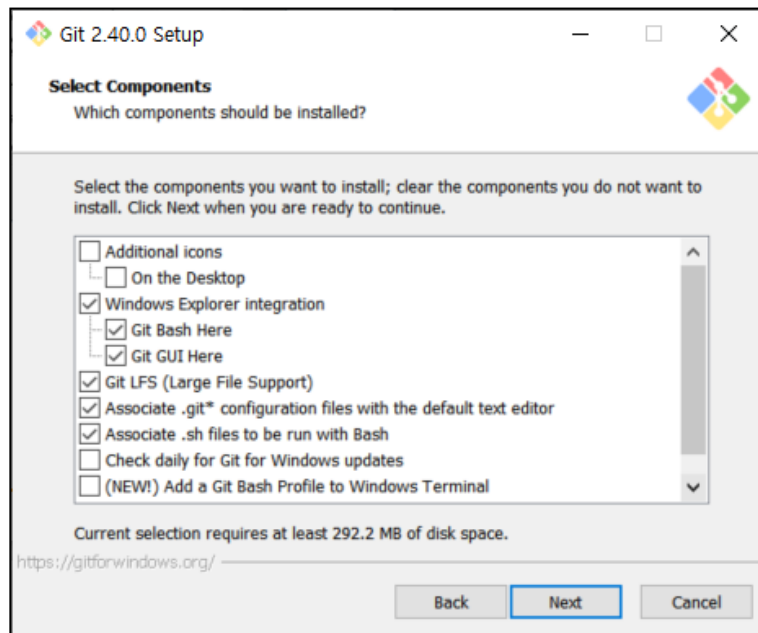
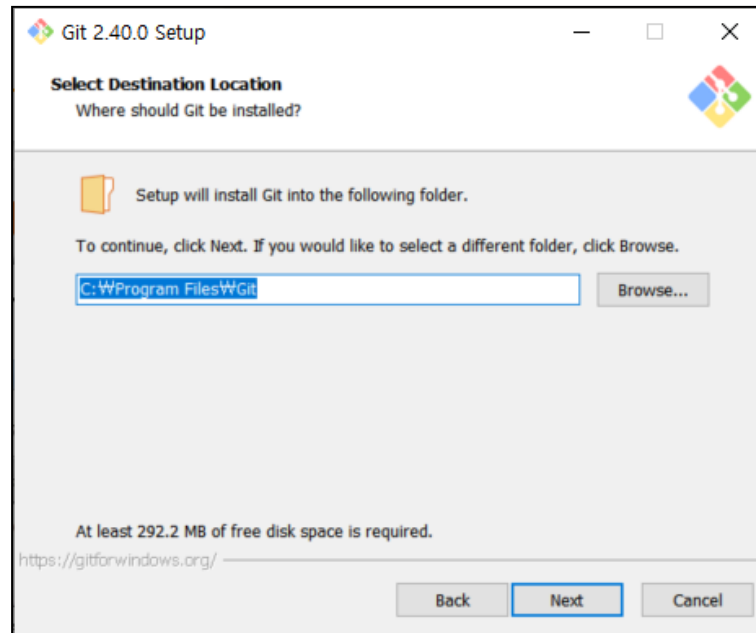
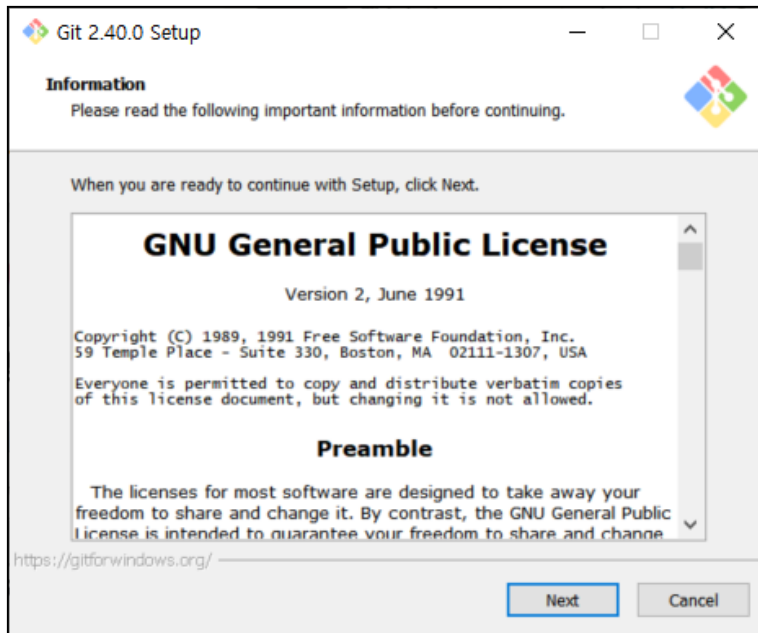
<https://git-scm.com/>



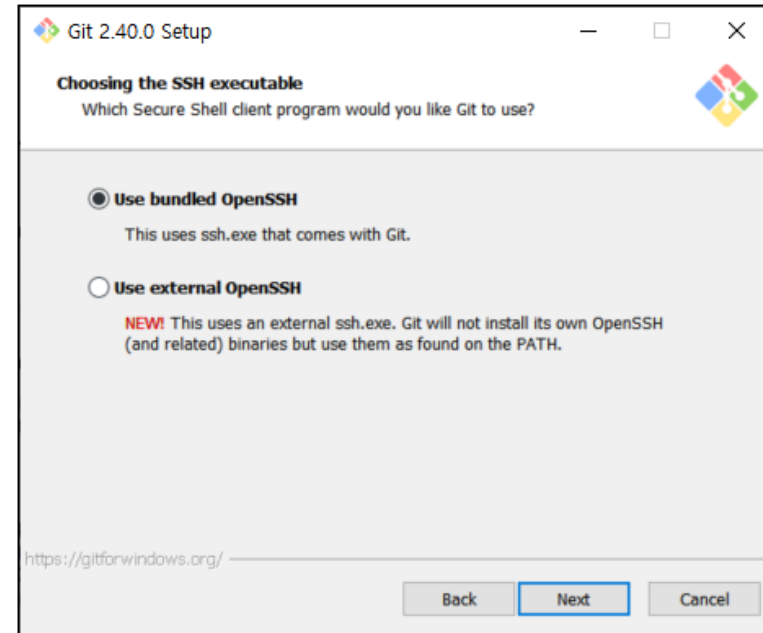
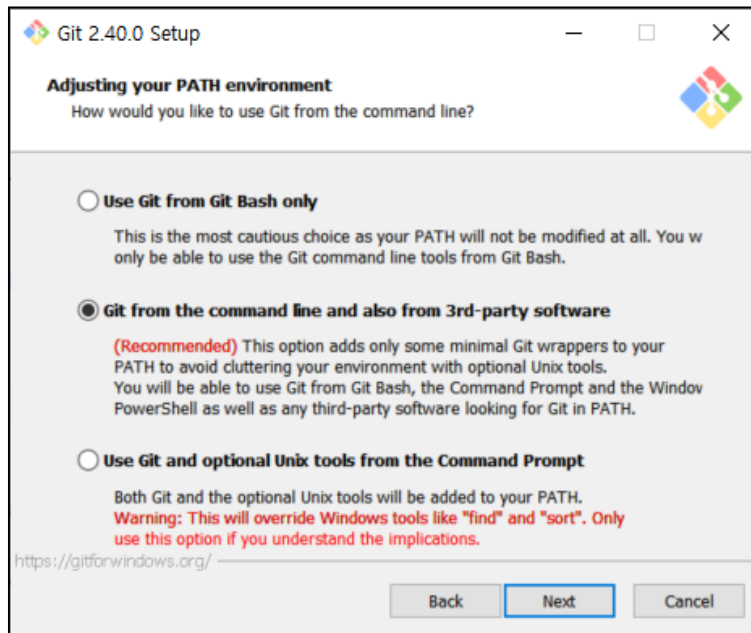
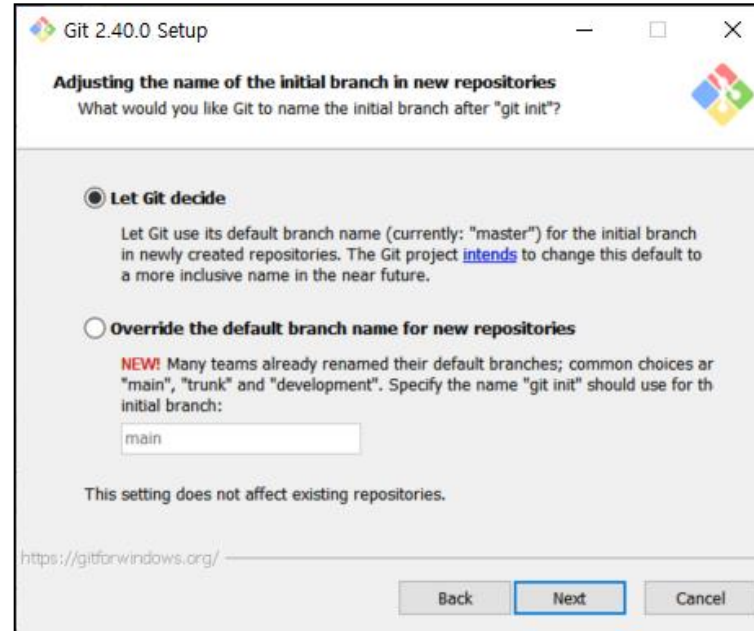
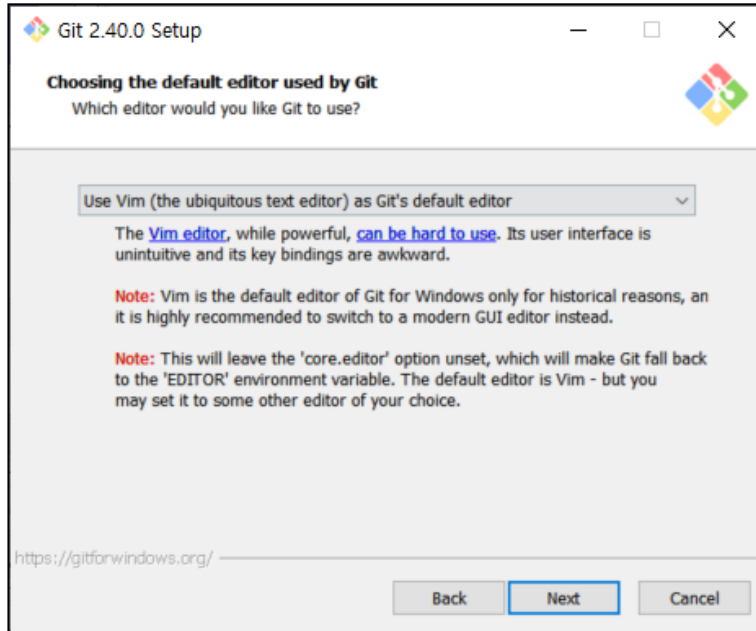
각 운영체제에 맞는 Git 프로그램 다운로드 하기



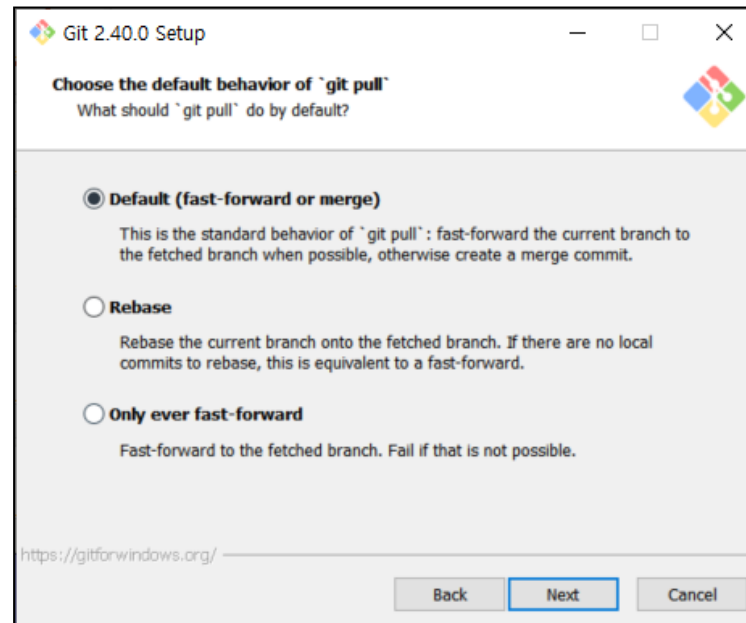
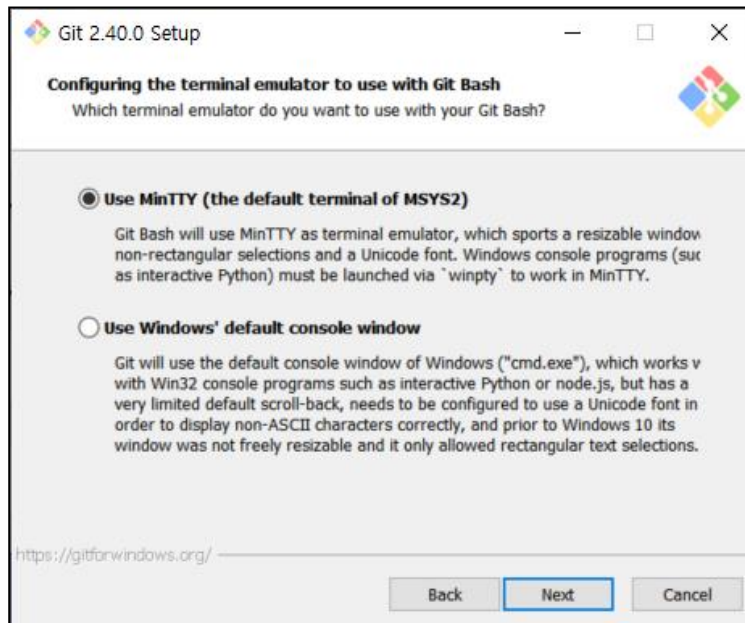
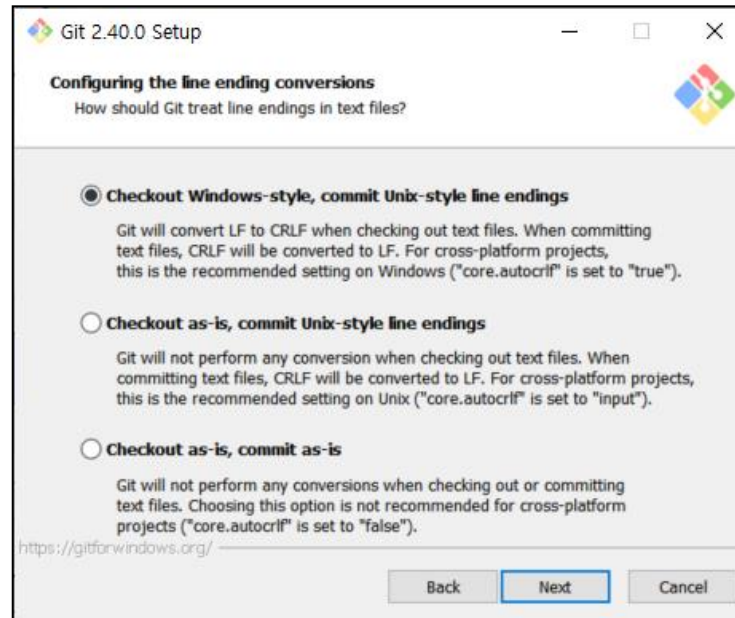
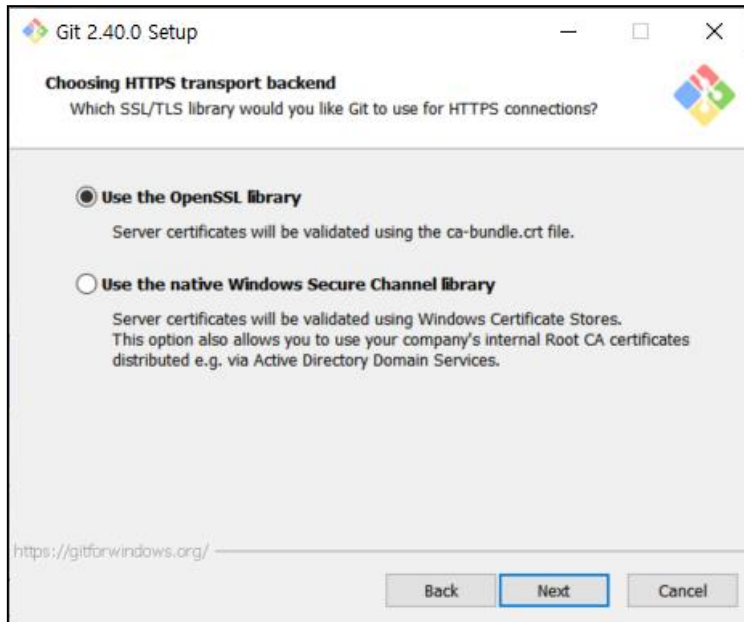
2. Git 설치



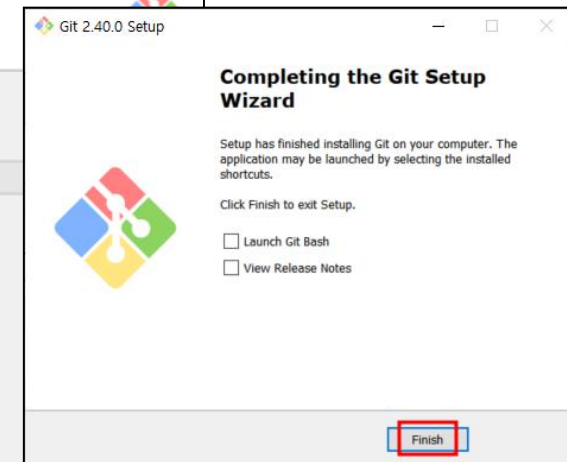
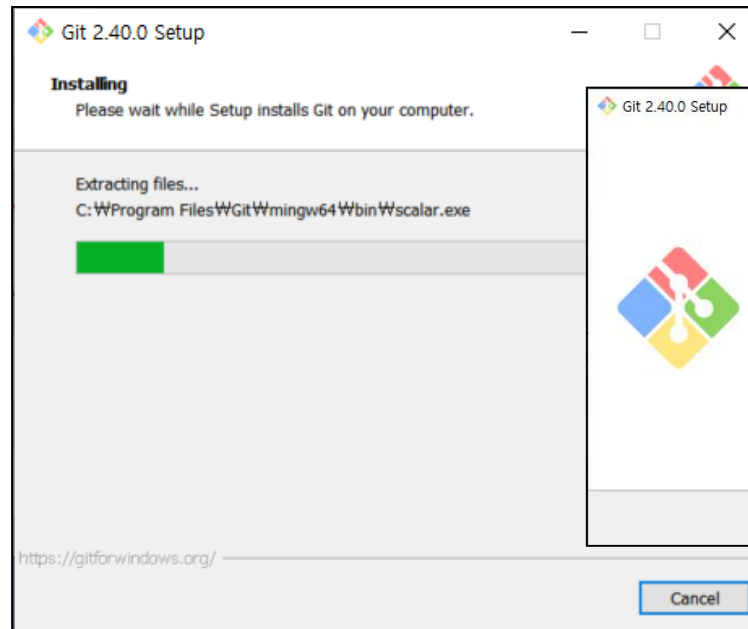
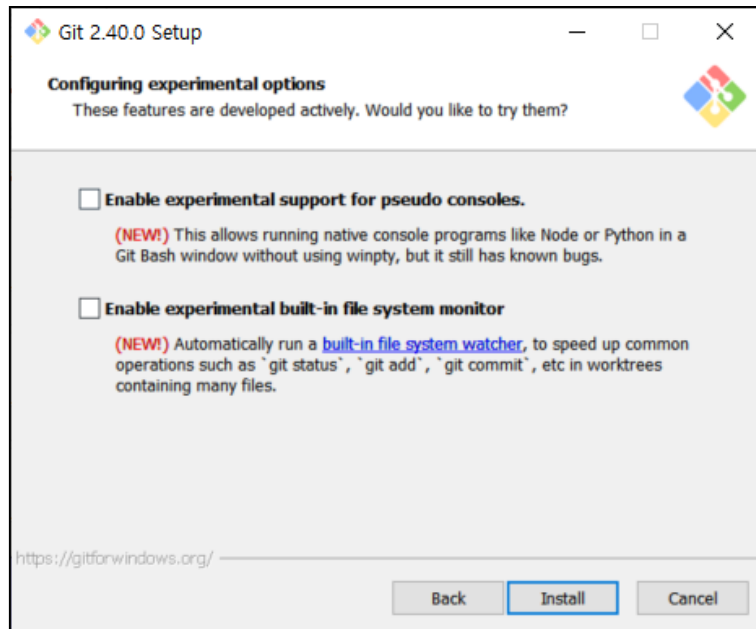
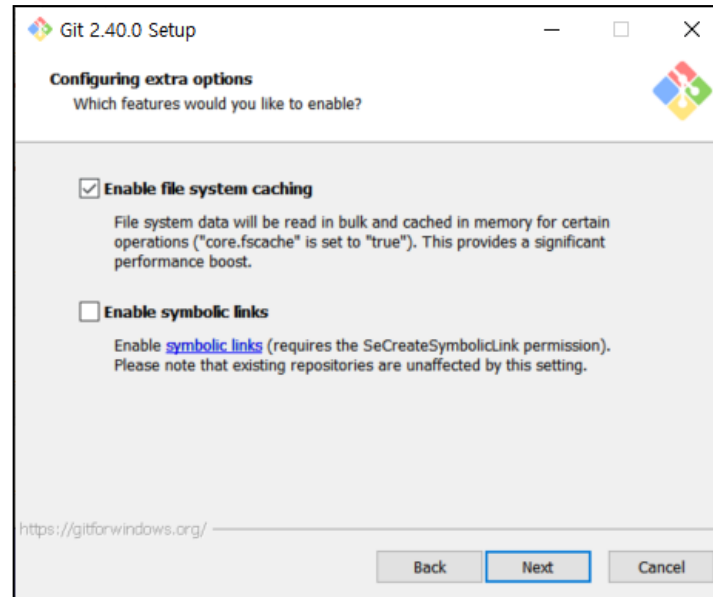
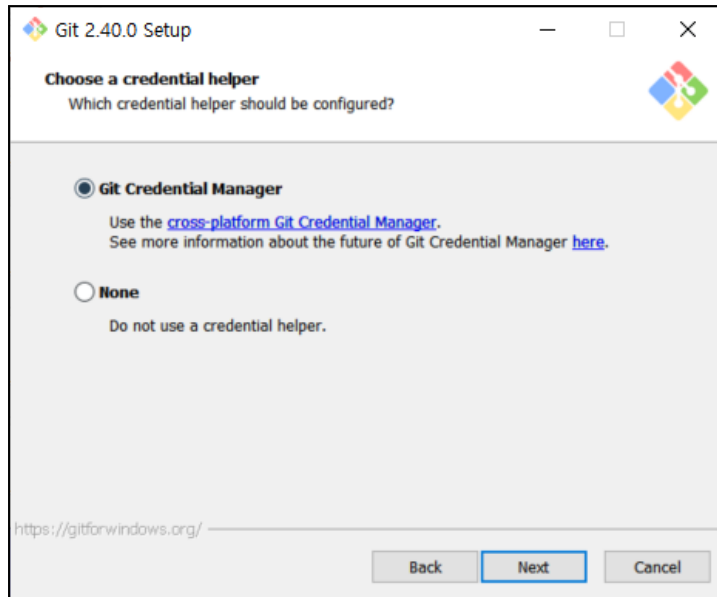
2. Git 설치



2. Git 설치



2. Git 설치



2. Git 설치

3) 사용자 정보 설정

터미널에서 git의 버전을 저장할 때 사용될 사용자 정보(이름,이메일)를 설정한다.



Git 버전 확인

```
acorn@DESKTOP-OD3GFRF MINGW64 ~  
$ git -v  
git version 2.41.0.windows.1
```

사용자 정보 설정

Git 설치 후 가장 먼저 할 것은 사용자이름과 이메일 주소를 설정하는 것이다.
Git은 커밋할 때마다 이 정보를 사용한다. 한 번 커밋한 후에는 정보를 변경할 수 없다.

```
$ git config --global user.name "pykwon"  
$ git config --global user.email pykwon@hanmail.net  
$ git config user.name  
pykwon  
$ git config user.email  
pykwon@hanmail.net
```


로컬 저장소(Local Repository) 생성 및 관리

1. 기본 명령어

git 기본 명령어

Repository(Git이 버전 관리를 하고 있는 폴더) 관리에 필요한 git 기본 명령어

명령어	기능	설명
git init	특정 디렉토리에 대한 저장소(repository) 생성	실행한 위치를 git 저장소로 지정
git add 파일명	저장소에 파일추가	해당파일을 git이 추적할 수 있도록 저장소에 추가
git commit	저장소에 수정 내용 제출	변경된 파일을 저장소에 제출
git status	저장소 상태 확인	현재 저장소의 상태 출력

Git 저장소 (Local Repository)만들기

주로 다음 두 가지 중 한 가지 방법으로 Git repository를 쓰기 시작한다.

- 1) 아직 버전관리를 하지 않는 로컬 디렉토리 하나를 선택해서 Git repository를 적용하는 방법
- 2) 다른 어딘가에서 Git repository를 Clone 하는 방법

어떤 방법을 사용하든 로컬 디렉토리에 Git repository가 준비되면 이제 뭔가 해볼 수 있다.

2. 로컬 저장소 생성

repository를 만들고 싶은 디렉토리로 이동해서 git을 초기화하면 그때부터 해당 디렉터리에 있는 파일들을 버전관리 할 수 있다.

디렉토리(directory) 생성 및 이동

Git repository를 만들 디렉터를 생성하고 해당 디렉터리로 이동

```
$ cd c:/work
$ mkdir git_test
$ cd git_test
$ ls -al
drwxr-xr-x 1 acorn 197121 0 Jun 13 16:49 ./
drwxr-xr-x 1 acorn 197121 0 Jun 13 16:49 ../
```

현재 디렉토리 위치 확인

```
$ pwd
/c/work/git_test
```

2. 로컬 저장소 생성

git init : Repository 생성 (파일들의 버전을 관리할 준비가 됨)

```
$ git init
```

Initialized empty Git repository in C:/work/git_test/.git/

디렉터리 안의 내용을 확인하면 버전이 저장될 repository인 .git 디렉터리가 생성되어 있다.

```
$ ls -al
```

```
drwxr-xr-x 1 acorn 197121 0 Jun 13 16:54 ./
drwxr-xr-x 1 acorn 197121 0 Jun 13 16:49 ../
drwxr-xr-x 1 acorn 197121 0 Jun 13 16:54 .git/
```

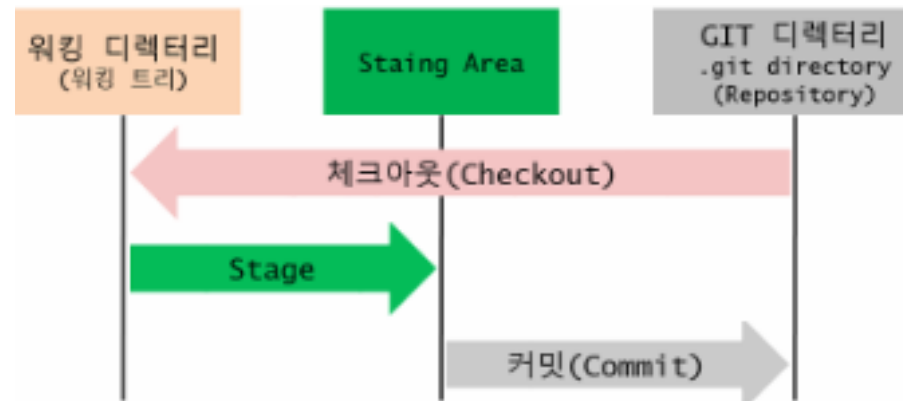
git status : repository 의 상태를 확인

```
$ git status
```

On branch master

No commits yet

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



위 상태 메시지의 의미는 다음과 같다.

- 1) 현재 master 브랜치에 있다.
- 2) 아직 커밋한 파일이 없다.
- 3) 현재 커밋할 파일이 없다.

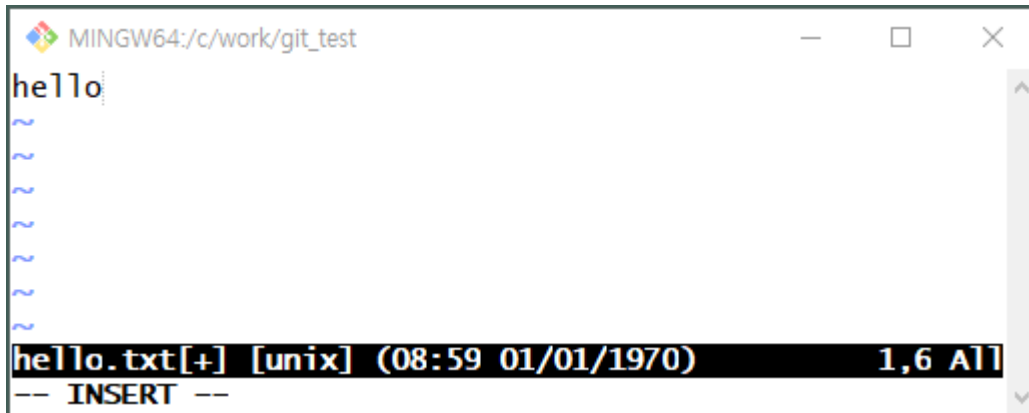
버전 관리 시스템은 브랜치를 지원한다. 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발을 진행할 수 있는데, 이렇게 독립적으로 개발하는 것이 브랜치다. Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다. 기본적으로 Git은 master 브랜치를 만든다. 처음 커밋하면 이 master 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 만들면 master 브랜치는 자동으로 가장 마지막 커밋을 가리킨다.

3. 버전으로 관리할 문서 작성

문서 작성

vi 에디터, 메모장 등의 편집기를 통해서 hello.txt 파일을 생성하고, 파일내용으로 "hello" 라는 문자열을 입력한 후 저장한다.

\$ vi hello.txt



```
MINGW64:/c:/work/git_test
hello
~
~
~
~
~
hello.txt[+] [unix] (08:59 01/01/1970) 1,6 All
-- INSERT --
```

- 1) a 또는 i 키보드 입력 (입력모드)
- 2) 원하는 문자열 입력
- 3) esc 키 (명령모드)
- 4) Shift + : 입력
- 5) :wq 라고 입력하여 파일저장 및 종료

3. 버전으로 관리할 문서 작성

문서 작성 후 저장소 상태 확인

```
$ git status
```

```
On branch master
```

```
No commits yet
```

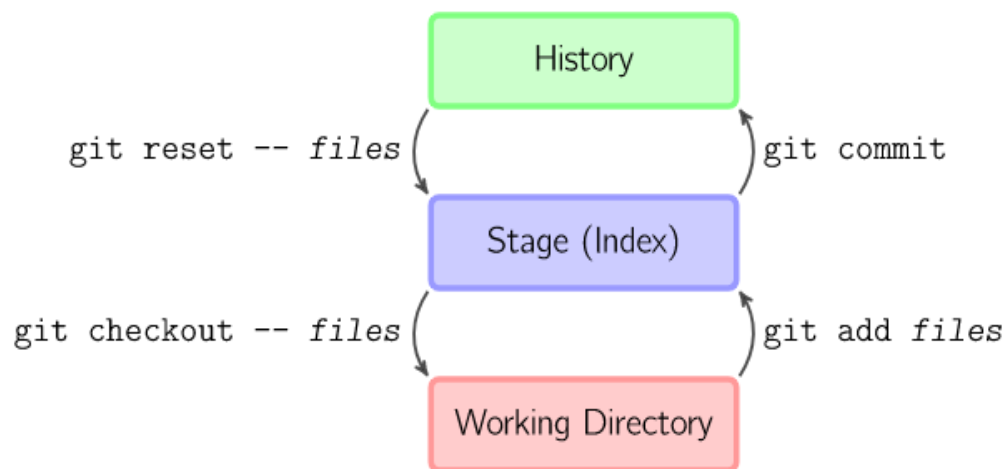
```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
hello.txt
```

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

git에서는 아직 한번도 버전관리 되지 않은 파일을 untracked files라고 한다.
즉 repository에 git이 아직 추적하지 않는 파일이 있음을 알려주는 화면이다.



4. 추적 시킬 파일 등록

추적 시킬 파일 등록: `git add 파일명`

git에게 버전 생성 준비를 지시하는 작업으로서 staging 또는 '스테이지에 올린다' 또는 'index에 등록한다' 라고 부른다.

```
$ git add hello.txt
```

warning: in the working copy of 'hello.txt', LF will be replaced by CRLF the next time Git touches it

참고 : warning은 윈도우의 줄바꿈 문자와 리눅스의 줄바꿈 문자가 서로 다르기 때문에 출력되는 경고로서 무시해도 된다.

```
$ git status
```

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: **hello.txt**

untracked files 문장이 changes to be committed로 바뀌었는데, 이것은 새파일 hello.txt를 앞으로 커밋할 것이라는 의미이다.

5. 버전 만들기

```
git commit -m "메시지 "
```

파일이 stage에 있으면 이제 버전을 만들 수 있다. git에서는 버전을 만드는 작업을 commit 한다고 말한다. 커밋할 때는 그 버전에 어떤 변경사항이 있었는지 메시지를 함께 저장한다.

```
$ git commit -m '처음으로 커밋'
```

```
[master (root-commit) 6d5d10a] 처음으로 커밋
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 hello.txt
```

커밋 후의 결과 메시지를 보면 파일 1개가 변경되었고 파일 1개가 추가되었다.
즉 stage에 있던 hello.txt 파일이 저장소에 추가된 것이다.

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

현재 git 상태를 보면 버전으로 만들 파일이 없고, working tree(작업 디렉토리)도 수정사항 없이 깨끗하다는 메시지가 출력된다.

6. 버전 확인

git log 옵션 : 커밋 내역을 확인할 수 있는 명령어이다.

명령어	설명
<code>git log -p</code>	각 커밋에 적용된 실제 변경 내용을 출력
<code>git log --stat</code>	각 커밋에서 수정된 파일의 통계정보 출력
<code>git log --name-only</code>	커밋 정보 중에서 수정된 파일의 목록만 출력
<code>git log --relative-date</code>	정확한 절대적인 시간이 아닌 상대적인 시간을 출력
<code>git log --graph</code>	브랜치 분기와 병합내역을 아스키 그래프로 출력
<code>git log --oneline</code>	한 줄에 한 커밋 씩 출력. 간략히 볼 때 사용

```
$ git log
```

```
commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772 (HEAD -> master)
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
Date: Tue Jun 13 17:44:27 2023 +0900
```

```
처음으로 커밋
```

가장 최신 버전

방금 커밋한 버전에 대한 설명이 출력된다. 커밋을 만든 사람, 만든 시간, 커밋 메시지가 함께 출력된다. `commit` 옆의 문자는 '커밋해쉬' 라고 부르고 커밋을 구별하는 역할을 한다.

7. 버전 확인

```
$ git log -p
```

```
commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772 (HEAD -> master)
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
Date: Tue Jun 13 17:44:27 2023 +0900
```

```
처음으로 커밋
```

```
diff --git a/hello.txt b/hello.txt
```

```
new file mode 100644
```

```
index 0000000..ce01362
```

```
--- /dev/null
```

```
+++ b/hello.txt
```

```
@@ -0,0 +1 @@
```

```
+hello
```

git log 명령어의 -p 옵션과 -[숫자] 옵션이 있는데 숫자 옵션은 -2 와 같이 사용하며, 최근 몇개의 내역을 보여줄 것인지를 지정하는 것이고 -p 옵션은 각 커밋의 diff 결과를 보여줌. 예) \$ git log -p -2

```
$ git log --stat
```

```
commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772 (HEAD -> master)
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
Date: Tue Jun 13 17:44:27 2023 +0900
```

```
처음으로 커밋
```

```
hello.txt | 1 +
```

```
1 file changed, 1 insertion(+)
```

각 커밋의 통계를 볼 수 있다. 각 통계는 얼마나 많은 파일이 몇 줄이나 수정되거나 추가 삭제되었는지를 보여주며 마지막에 요약 정보를 보여줌

7. 버전 확인

\$ git log --name-only

commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772 (HEAD -> master)

Author: pykwon <pykwon@hanmail.net>

Date: Tue Jun 13 17:44:27 2023 +0900

처음으로 커밋

hello.txt

\$ git log --relative-date

commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772 (HEAD -> master)

Author: pykwon <pykwon@hanmail.net>

Date: 15 minutes ago : 정확한 시간이 아니라 15분 전 처럼 상대적인 형식으로 보여줌

처음으로 커밋

\$ git log --graph

* commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772 (HEAD -> master)

Author: pykwon <pykwon@hanmail.net>

Date: Tue Jun 13 17:44:27 2023 +0900

처음으로 커밋

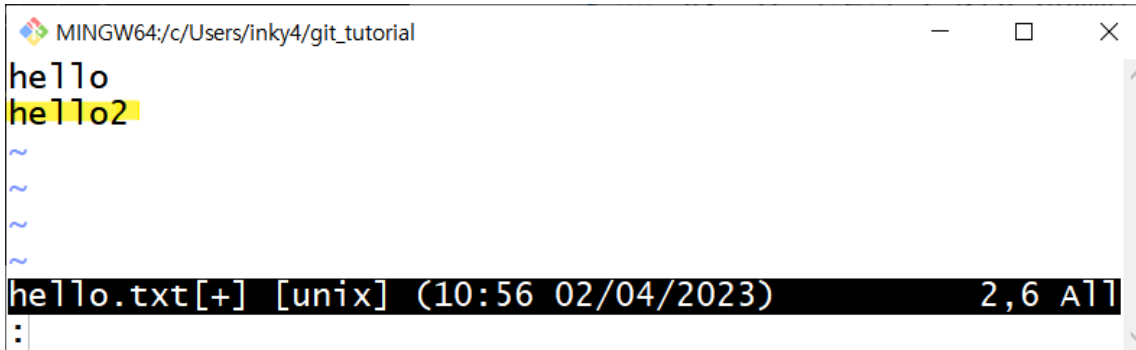
commit/merge 등의 log를 그래프의 형태로 나타내준다. 현재는 작업이 내용이 적어 보이지 않음

8. staging과 commit을 한꺼번에 처리

```
git commit -am '메시지'
```

commit 명령어에 **-am** 옵션을 사용하면 파일을 스테이지에 올리고 커밋하는 과정을 한 번에 처리할 수 있다. 단 반드시 한번이라도 커밋한 적이 있어야 된다.

```
$ vi hello.txt
```



```
MINGW64:/c/Users/inky4/git_tutorial
hello
hello2
~
~
~
~
hello.txt[+] [unix] (10:56 02/04/2023) 2,6 All
:
```

```
$ git commit -am '두 번째 커밋'
```

warning: in the working copy of 'hello.txt', LF will be replaced by CRLF the next time Git touches it

```
[master a89d3b8] 두 번째 커밋
```

```
1 file changed, 1 insertion(+)
```

commit 명령어에 **-am** 옵션을 지정하여 스테이징과 커밋을 한 번에 처리한다.

8. staging과 commit 한꺼번에

\$ git status

On branch master

nothing to commit, working tree clean

\$ git log

commit a89d3b892e0f9731256c35a43822bfce86380c2a (HEAD -> master)

Author: pykwon <pykwon@hanmail.net>

Date: Wed Jun 14 12:46:55 2023 +0900

두 번째 커밋

commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772

Author: pykwon <pykwon@hanmail.net>

Date: Tue Jun 13 17:44:27 2023 +0900

처음으로 커밋

\$ git log --graph

* commit a89d3b892e0f9731256c35a43822bfce86380c2a (HEAD -> master)

| Author: pykwon <pykwon@hanmail.net>

| Date: Wed Jun 14 12:46:55 2023 +0900

| 두 번째 커밋

* commit 6d5d10ab69a298cb7747e64d384cf0b9d9cbf772

Author: pykwon <pykwon@hanmail.net>

Date: Tue Jun 13 17:44:27 2023 +0900

처음으로 커밋

9. 커밋 메시지 수정

git commit --amend

문서의 수정 내용을 기록하는 커밋 메시지를 잘못 입력했다면, 커밋을 만든 즉시 커밋 메시지를 수정할 수 있다.

\$ git commit --amend



```
MINGW64:/c/work/git_test
두 번째 커밋을 수행함 바로 이전에 커밋 메시지를 수정함
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Wed Jun 14 12:46:55 2023 +0900
#
# On branch master
# Changes to be committed:
#   modified:   hello.txt
#
~
~
~
git/COMMIT_EDITMSG[+] [unix] (14:45 14/06/2023) 2,0-1 모두
:wq 메시지 저장 후 탈출
```

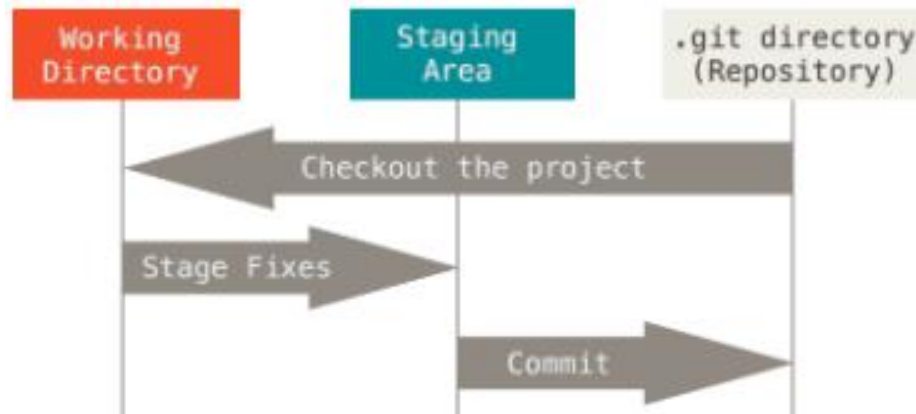
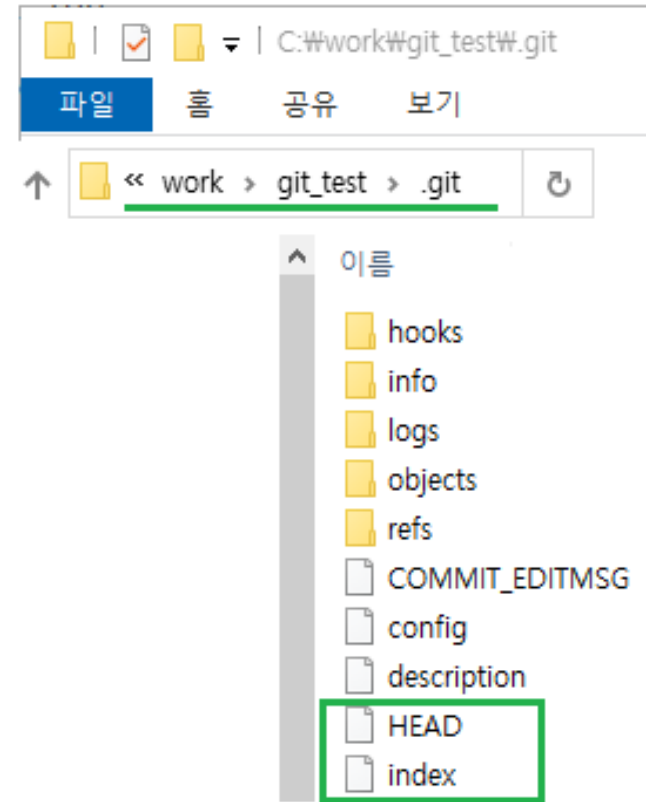
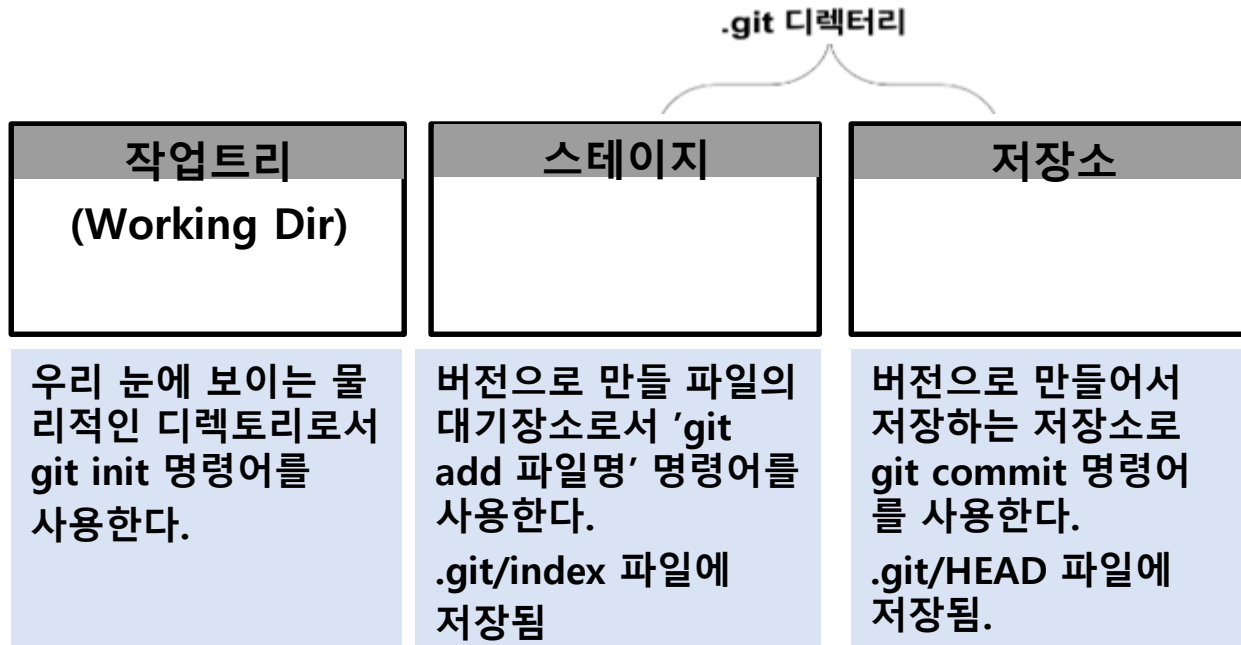
[master 38dfb56] 두 번째 커밋을 수행함
Date: Wed Jun 14 12:46:55 2023 +0900
1 file changed, 1 insertion(+)

\$ git log

하면 원래 커밋 메시지를 수정하고 저장하면 커밋 메시지가 수정되면서 이전 커밋에 더해진다.

10. 저장소 구조

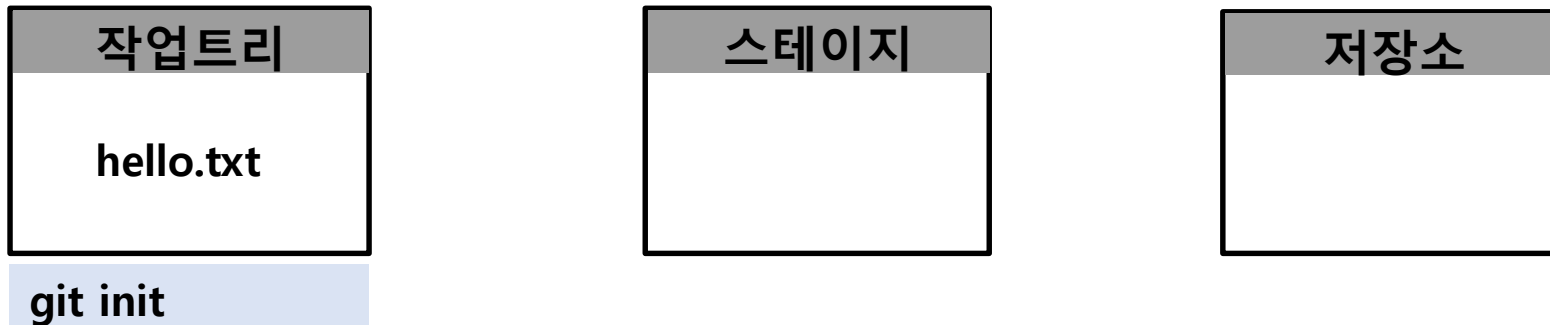
git은 관리할 파일들의 이름을 그대로 유지하면서 수정 내역을 기록한다.
이를 위해서 다음과 같은 구조를 사용한다.



10. 저장소 구조

stage와 repository는 눈에 보이지 않는다. git을 초기화했을 때 만들어지는 .git 디렉터리 안에 숨은 파일 형태로 존재하게 된다.

1) git init 으로 초기화하고 hello.txt 문서를 작성한 후



```
$ git status
On branch master

No commits yet

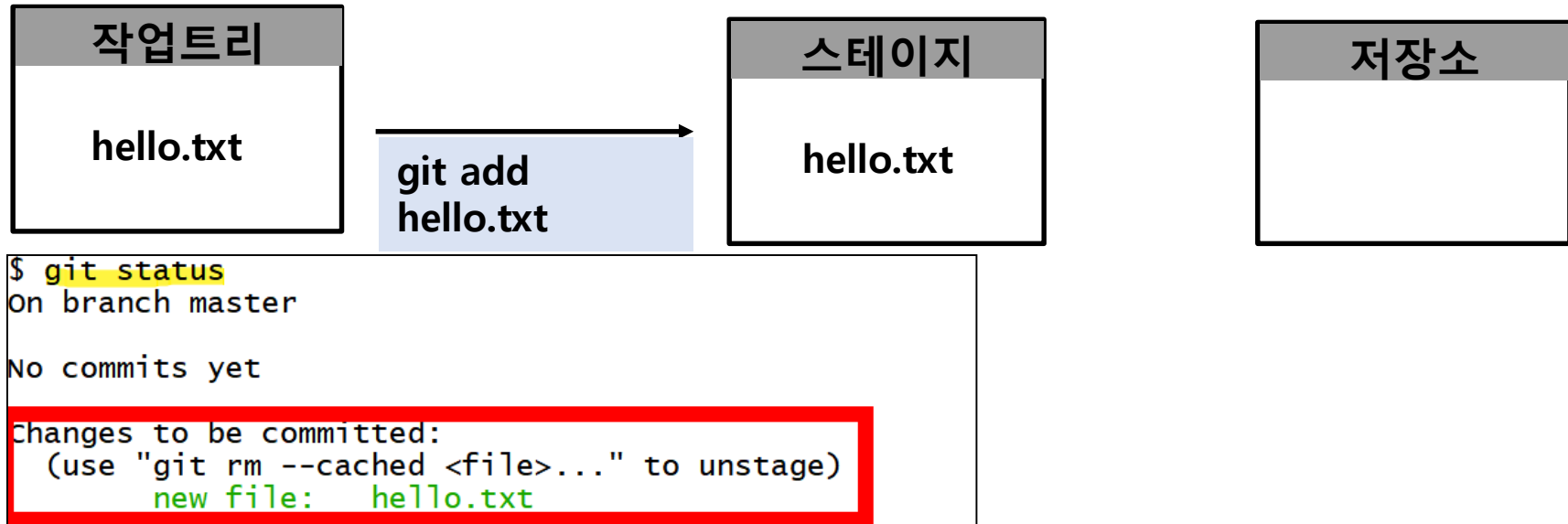
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello.txt

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

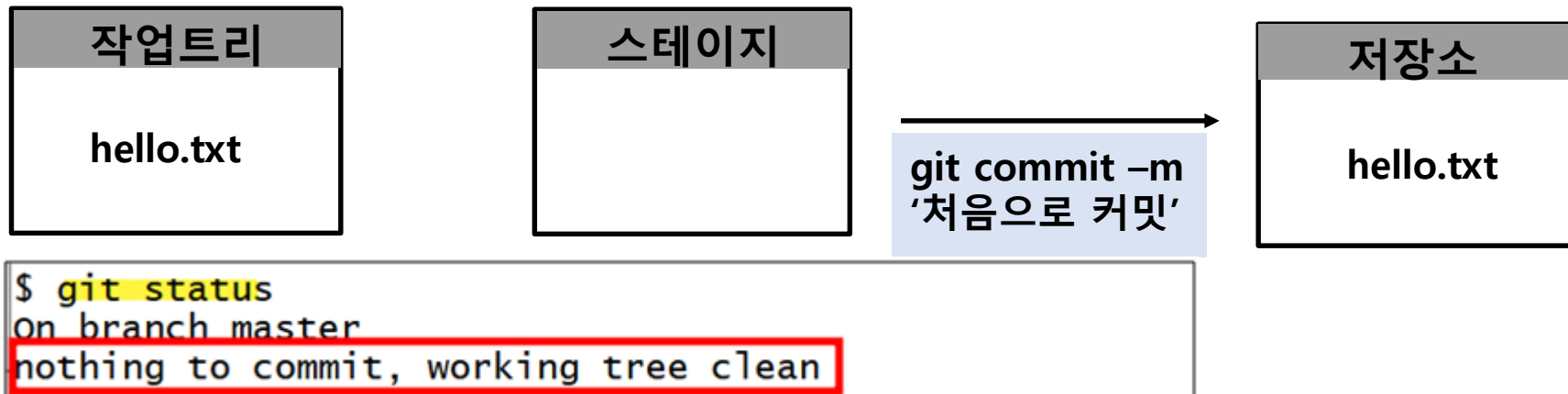
MINGW64 ~/git_tutorial (master)
```

10. 저장소 구조

2) `git add hello.txt` 명령어를 이용해 추적파일로 지정한 후



3) `git commit -m "first commit"` 명령어로 저장소에 제출한 후



변경사항 비교

1. 변경사항 비교

git diff

git diff 명령어를 사용하면 수정내용이 포함된 작업 트리에 있는 파일과 최신 커밋 내용을 비교해서 수정파일을 커밋하기 전에 최종적으로 검토할 수 있다.



1) 현재상태 확인

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

```
$ cat hello.txt
```

```
hello
```

```
hello2
```

1. 변경사항 비교

2) hello.txt 파일 내용 수정

\$ vi hello.txt

```
MINGW64:/c/work/git_test
hello
hello2
hello3
~
~
~
hello.txt[+] [unix] (12:46 14/06/2023) 3,6 모 두
:wq
```

작업트리
hello.txt
hello hello2 hello3

저장소
hello.txt
hello hello2

\$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: hello.txt

no changes added to commit (use "git add" and/or "git commit -a")

1. 변경사항 비교

3) 변경사항 비교



```
$ git diff
```

```
diff --git a/hello.txt b/hello.txt
```

```
index 97531f3..3cea797 100644
```

```
--- a/hello.txt
```

```
+++ b/hello.txt
```

```
@@ -1,2 +1,3 @@
```

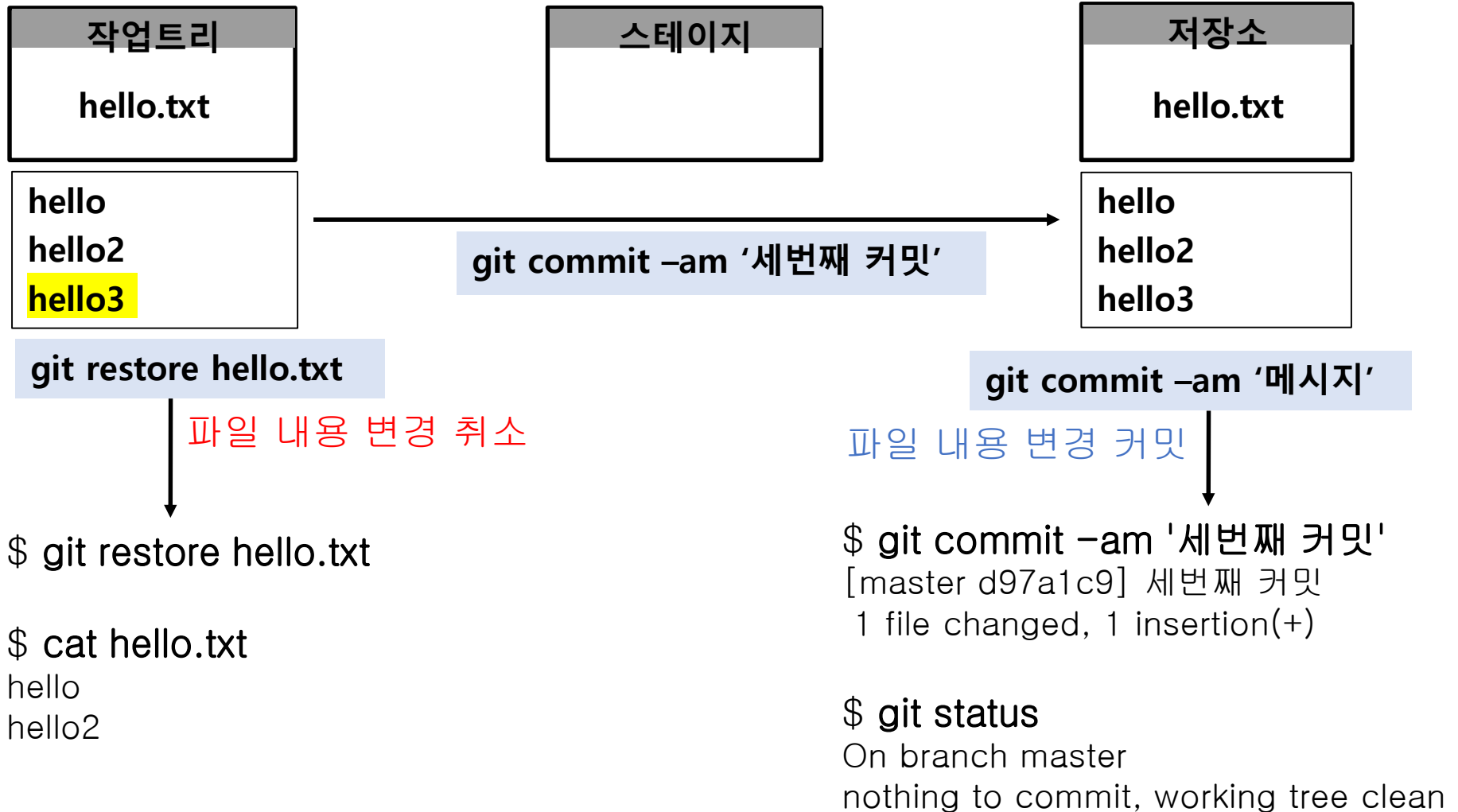
```
hello
```

```
hello2
```

```
+hello3
```


1. 변경사항 비교

4) 최종처리 작업 (커밋 또는 무시)

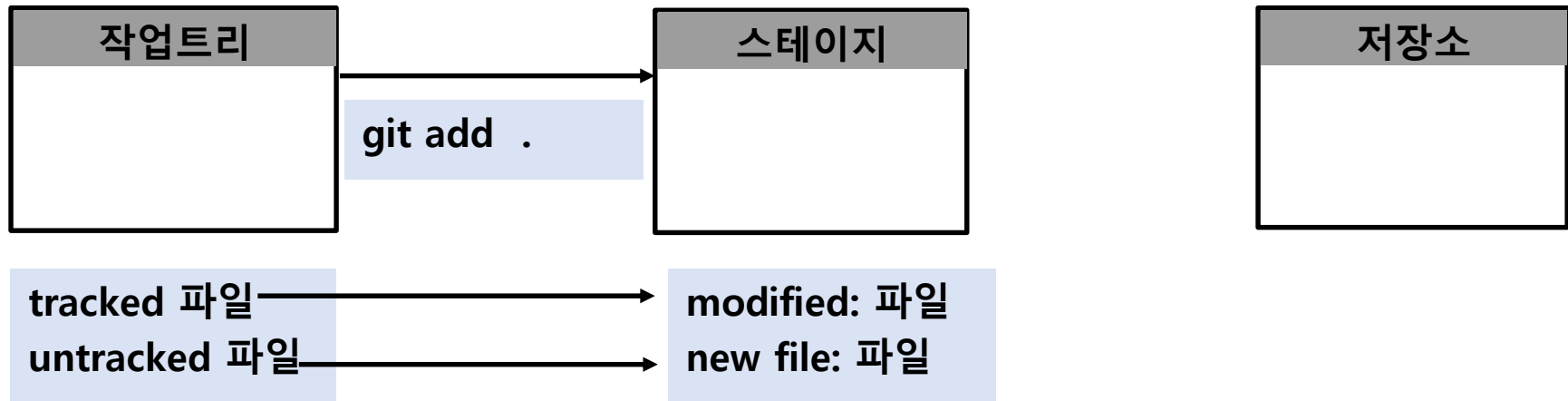


각 버전 단계에서 파일 상태 확인

1. 버전 단계마다의 파일 상태

git에서는 버전을 만드는 각 단계마다 파일 상태를 다르게 표시한다.

따라서 파일의 상태를 이해하면 이 파일이 버전 관리의 여러 단계 중에서 어디에 있는지 알 수 있다.



2. tracked vs untracked 파일

tracked 파일은 git이 한 번이라도 커밋을 한 파일의 수정여부를 계속 추적하는 파일을 의미한다.

untracked 파일은 한 번도 버전관리를 하지 않은 파일을 의미한다.

1) 기존 hello.txt 파일을 수정하고, 새로운 world.txt 파일 추가

```
$ vi hello.txt
```

```
$ vi world.txt
```

```
$ ls -al
```

```
drwxr-xr-x 1 acorn 197121 0 6월 14 16:02 ./
drwxr-xr-x 1 acorn 197121 0 6월 13 16:49 ../
drwxr-xr-x 1 acorn 197121 0 6월 14 15:43 .git/
-rw-r--r-- 1 acorn 197121 23 6월 14 15:42 hello.txt
-rw-r--r-- 1 acorn 197121 6 6월 14 16:02 world.txt
```

작업트리

hello.txt

world.txt

2. Tracked vs untracked 파일

2) 파일 상태 확인

```
$ git status
```

```
On branch master
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working directory)
```

```
modified:   hello.txt
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
world.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

hello.txt 파일은 **changes not staged for commit**로서 변경된 파일이 스테이지에 올라가지 않았음을 알 수 있고 modified: 라고 표시되어 있어서 파일이 수정되었음을 알 수 있다. 이렇게 git은 한 번이라도 커밋한 파일의 수정여부를 계속 추적하며 git이 추적하고 있다는 뜻에서 **tracked** 파일이라고 부른다.

한편 world.txt 파일은 한 번도 git이 버전관리를 하지 않았고 수정내역을 추적하지 않았기 때문에 **untracked files**라고 출력된다.

2. Tracked vs untracked 파일

3) new file 및 modified 파일을 스테이지에 올리기

```
$ git add .
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: hello.txt

new file: world.txt

마지막 커밋 이후 수정된 **hello.txt** 파일 (tracked)은 **modified:** 로 표시되고 버전관리가 처음인 **world.txt** 파일 (untracked)은 **new file:**로 표시된다.

```
$ git commit -m 'new file world.txt'
```

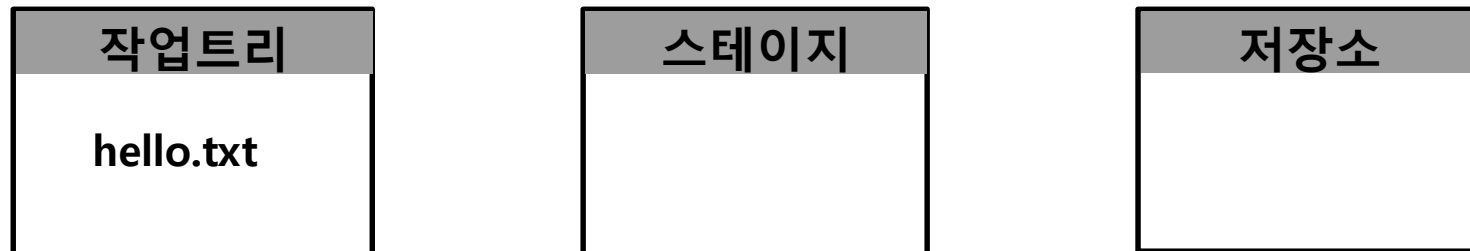
```
[master 655a159] new file world.txt
```

```
2 files changed, 2 insertions(+), 1 deletion(-)
```

```
create mode 100644 world.txt
```

3. Tracked 파일의 상태 (unmodified, modified, staged)

tracked 파일은 작업 트리에 있는지 스테이지에 있는지 등 더 구체적인 상태를 알 수 있다.



1) 현재 수정사항이 없는 경우

```
$ git status
On branch master
nothing to commit, working tree clean
```

unmodified 상태

3. Tracked 파일의 상태 (unmodified, modified, staged)

2) 현재 수정 사항이 있는 경우 (world.txt 내용 수정)

```
$ vi world.txt
```

```
$ git status
On branch master
Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
        modified:   world.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

3) 스테이징 및 커밋

```
$ git add world.txt
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
    (use "git restore --staged <file>..." to unstage)
```

```
        modified:   world.txt
```

```
$ git commit -m 'modified world.txt'
```

```
[master 58dbe7d] modified world.txt
```

```
1 file changed, 1 insertion(+)
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

불필요한 파일 및 폴더 무시

1. 버전 관리에서 제외

.gitignore 파일

버전 관리 중인 디렉터리 안에 버전 관리를 하지 않을 특정 파일이나 폴더가 있다면 .gitignore 파일을 만들고 그 안에 파일 또는 폴더명을 지정하면 된다.

1) 현재 디렉터리에 tmp 폴더와 Test.java 파일 생성 (tmp 폴더에 exam.txt 파일 추가)

```
$ pwd  
/c/work/git_test
```

```
$ mkdir tmp
```

```
$ touch Test.java
```

```
$ ls -al  
drwxr-xr-x 1 acorn 197121 0  6월 14 17:51 ./  
drwxr-xr-x 1 acorn 197121 0  6월 13 16:49 ../  
drwxr-xr-x 1 acorn 197121 0  6월 14 16:29 .git/  
-rw-r--r-- 1 acorn 197121 27  6월 14 16:07 hello.txt  
-rw-r--r-- 1 acorn 197121 0  6월 14 17:51 Test.java  
drwxr-xr-x 1 acorn 197121 0  6월 14 17:51 tmp/  
-rw-r--r-- 1 acorn 197121 13  6월 14 16:25 world.txt
```

```
$ cd tmp  
$ vi exam.txt
```

```
$ ls  
exam.txt  
$ pwd  
/c/work/git_test/tmp  
$ cd ..  
$ pwd  
/c/work/git_test
```

1. 버전 관리에서 제외

2) 상태정보 확인

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Test.java
    tmp/

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

참고 : 버전 관리에서 제외하기 위한 .gitignore 파일을 작성

.gitignore 파일의 정의

해당 프로젝트 내에서 불 필요하다고 느끼는 특정 '파일' 및 '디렉토리 경로'에 대해서 Repository에 올리지 않기 위해서 이 파일들을 무시하기 위한 정보를 가지고 있는 파일(.gitignore)을 의미함.

.gitignore에 포함되는 정보

- 용량이 커서 제외 되어야 할 파일 혹은 디렉토리 경로
- 보안적인 문제에서 걸려 제외 되어야 할 파일 혹은 디렉토리 경로
- 불 필요하다고 판단 되어 제외 되어야 할 파일 혹은 디렉토리 경로

.gitignore 파일 작성 예

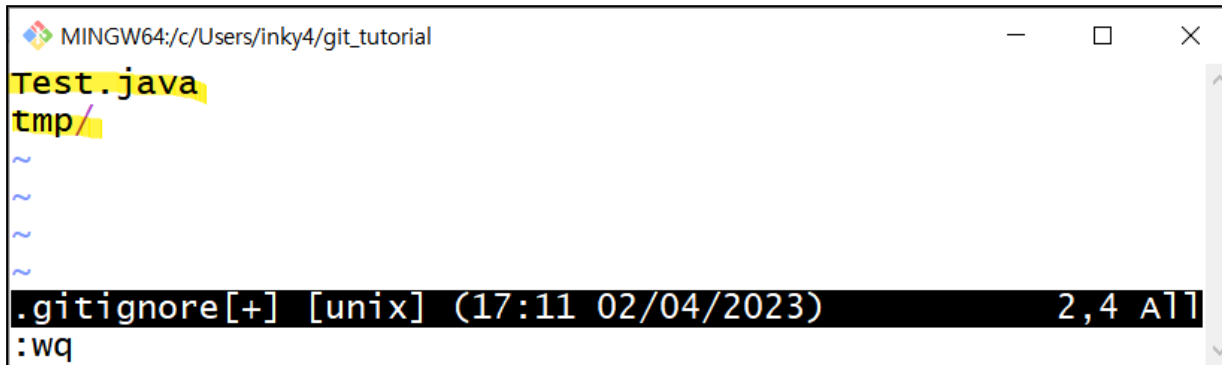
- 1) '파일명'으로 제외하는 방법 (* 해당 방법은 경로 상관없이 지정한 파일명으로 모두 제외할 수 있다)
ignoreFileName.js
- 2) 특정 '파일'만 제외하는 방법 (* 현재 기준을 .ignore파일이 있는 경로라고 생각하면 된다)
src/ignoreFileName.js
- 3) 특정 '디렉토리' 기준 이하 파일들 제외 방법
node_module/

1. 버전 관리에서 제외

3) .gitignore 파일 작성

```
$ touch .gitignore
```

```
$ vi .gitignore
```



The screenshot shows a Windows command prompt window titled "MINGW64:/c/Users/inky4/git_tutorial". The window displays the contents of the .gitignore file, which includes "Test.java" and "tmp/". The status bar at the bottom indicates that the file is tracked and has 2,4 lines of code.

```
Test.java
tmp/
~
~
~
~
.gitignore[+] [unix] (17:11 02/04/2023) 2,4 All
:wq
```

4) 상태정보 확인

```
$ git status
```

```
On branch master
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
.gitignore
```

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

.gitignore 파일 안에 명시된 Test.java 와 tmp 폴더가 버전 관리 목록에서 제외된 것을 확인할 수 있다.

1. 버전 관리에서 제외

5) 스테이징 및 커밋

```
$ git add .gitignore
```

```
$ git commit -m 'git ignore add'
```

```
[master 75839c1] git ignore add
```

```
1 file changed, 2 insertions(+)
```

```
create mode 100644 .gitignore
```

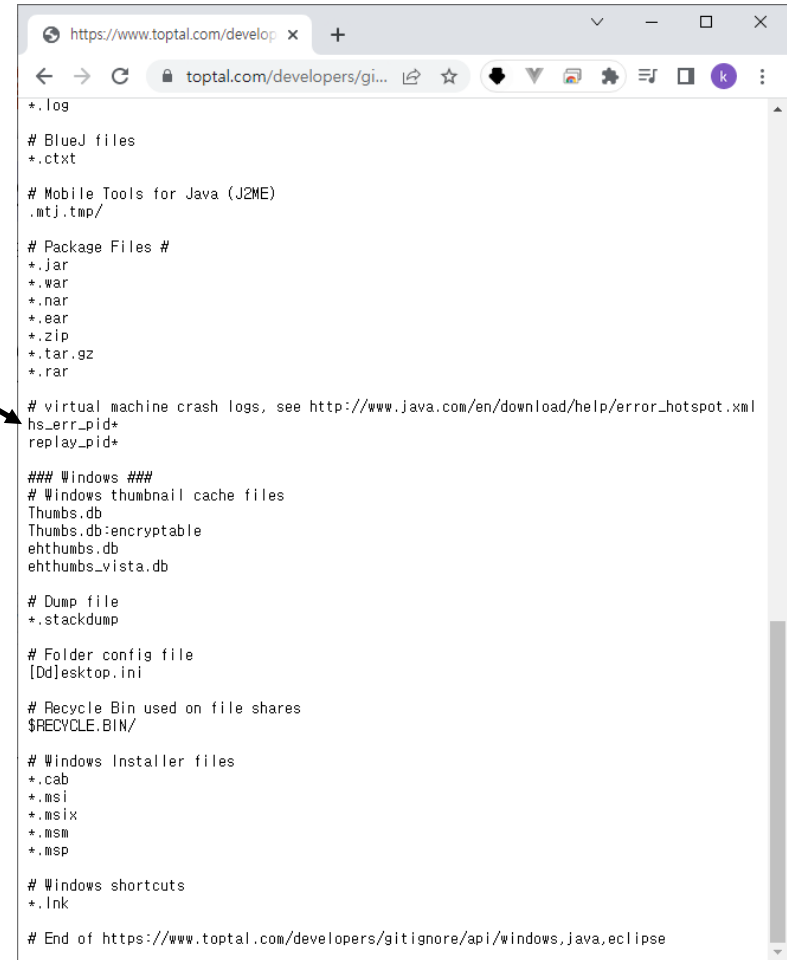
2. 웹 사이트를 이용하여 .gitignore 파일 작성하기

<https://www.toptal.com/developers/gitignore>

.gitignore.io는 Git 리포지토리를 위한 .gitignore 파일을 만드는 데 도움이 되도록 설계된 웹 서비스



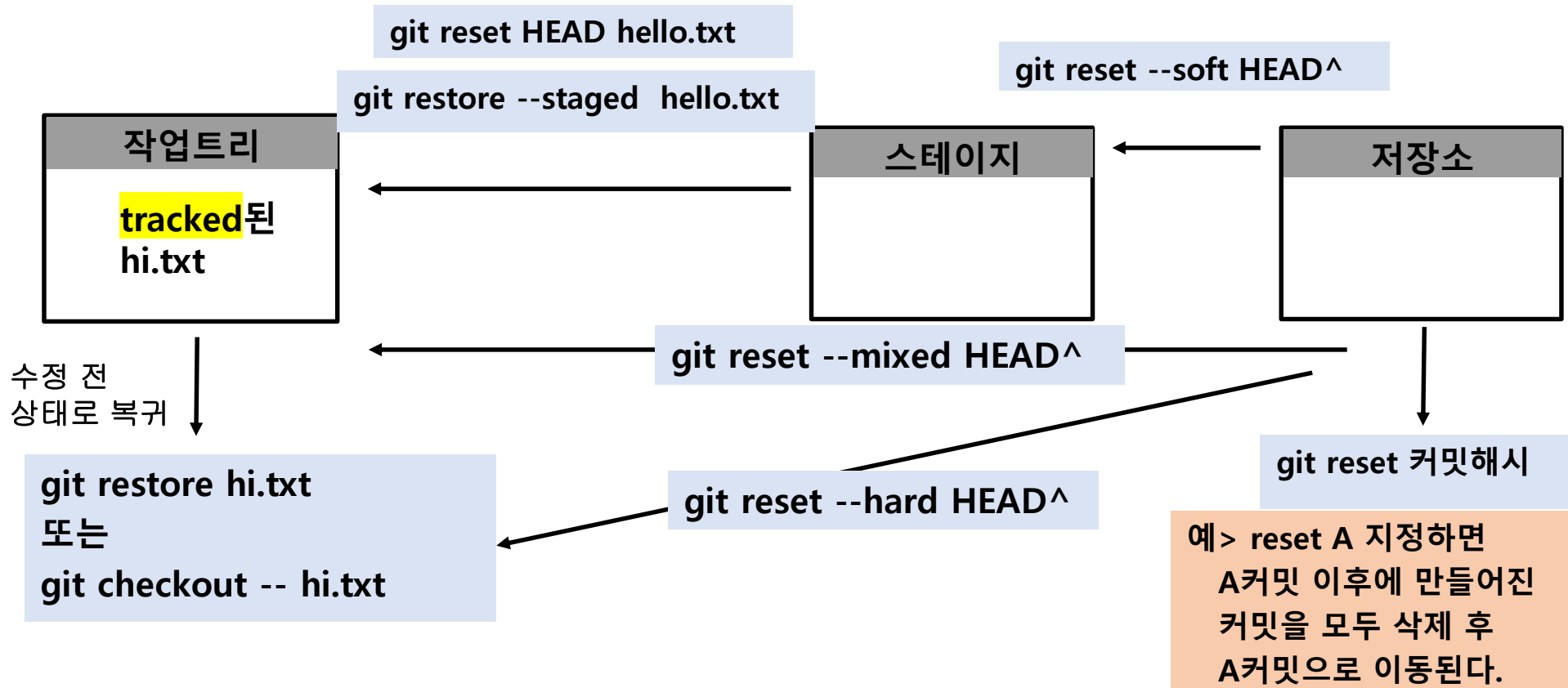
생성 입력 영역에 Windows하고 엔터,
이어서 Java 이런 식으로 입력해 주면 된다.



작업 되돌리기

1. 작업 되돌리기

작업트리에서 수정했던 변경사항을 취소하거나 스테이지에 올렸던 파일을 내리거나 커밋을 취소하는 등 각 단계에서 수행했던 작업을 되돌리는 작업을 수행할 수 있다.
주의할 점은 반드시 tracked 된 파일이어야 된다.



2. 작업트리에서 수정한 파일내용 되돌리기

git checkout -- hi.txt 또는 git restore hi.txt

1) 새로운 repository 생성 후 파일 생성 및 커밋

```
$ cd ..  
$ ls  
  git_test/  resou/ ...  
  
$ mkdir git_test2  
$ cd git_test2  
$ git init  
Initialized empty Git repository in  
C:/work/git_test2/.git/  
  
$ vi hi.txt  
$ cat hi.txt  
hi
```



```
$ git add hi.txt  
$ git status  
On branch master  
No commits yet  
Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
    new file:   hi.txt
```

```
$ git commit -m 'first commit'  
[master (root-commit) 27f90f0] first commit  
 1 file changed, 1 insertion(+)  
 create mode 100644 hi.txt
```

```
$ git status  
On branch master  
nothing to commit, working tree clean
```

2. 작업트리에서 수정한 파일내용 되돌리기

2) 파일 수정

```
$ vi hi.txt
$ cat hi.txt
hi
hi2
```

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what ...)

(use "git restore <file>..." to discard ...)

modified: hi.txt

no changes added to commit (use "git add" and/or
"git commit -a")

```
$ git diff hi.txt
```

```
diff --git a/hi.txt b/hi.txt
```

```
index 45b983b..f4acb98 100644
```

```
--- a/hi.txt
```

```
+++ b/hi.txt
```

```
@@ -1 +1,2 @@
```

```
hi
```

```
+hi2
```

3) 파일 변경 내용 되돌리기

```
$ git checkout -- hi.txt
```

또는 \$ git restore hi.txt

```
$ cat hi.txt
```

```
hi
```

```
$ git status
```

On branch master

nothing to commit, working tree
clean

3. 스테이징 파일 되돌리기

git reset HEAD hi.txt

또는

git restore --staged hi.txt

1) 현재 파일 내용

```
$ cat hi.txt  
hi
```

2) 파일 수정 후 스테이징

```
$ vi hi.txt  
$ cat hi.txt  
hi  
fine
```

```
$ git add hi.txt  
$ git status  
On branch master  
Changes to be committed:  
  (use "git restore --staged<file>..." to  
  unstage)  
    modified:   hi.txt
```

```
$ git reset HEAD hi.txt
```

Unstaged changes after reset:

M hi.txt

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what ...)

(use "git restore <file>..." to discard changes ...)

modified: hi.txt

no changes added to commit (use "git add" ...)

```
$ git add hi.txt
```

<== 다시 스테이징

```
$ git restore --staged hi.txt
```

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what ...)

(use "git restore <file>..." to discard changes ...)

modified: hi.txt

no changes added to commit (use "git add" ...)

4. 최신 커밋 되돌리기

git reset HEAD^

명령어 종류	설 명
--soft HEAD^	최근 커밋을 하기 전 스테이지 상태로 되돌리기
--mixed HEAD^	최근 커밋과 스테이징 하기 전 상태로 되돌리기
--hard HEAD^	최근 커밋, 스테이징, 파일 수정 하기 전 상태로 되돌리기

1) 수정된 파일 커밋

```
$ cat hi.txt
```

```
hi  
fine
```

```
$ git status
```

```
On branch master
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working directory)
```

```
modified: hi.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git add hi.txt
```

```
$ git commit -m 'second commit'
```

```
[master 8d933a0] second commit
```

```
1 file changed, 1 insertion(+)
```

4. 최신 커밋 되돌리기

2) 커밋 내역 목록

\$ git log

commit 8d933a032c41224c4198875100ecd99ce6d8d36a (HEAD -> master)

Author: pykwon <pykwon@hanmail.net>

Date: Thu Jun 15 16:06:29 2023 +0900

second commit

commit 27f90f0b1063a73e31e9295c2cea502e6de3796e

Author: pykwon <pykwon@hanmail.net>

Date: Thu Jun 15 14:45:18 2023 +0900

first commit

3) 최신 커밋 되돌리기 (스테이징도 함께 취소)

\$ git reset HEAD^

Unstaged changes after reset:

M hi.txt

\$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: hi.txt

no changes added to commit (use "git add" and/or "git commit -a")

4) 커밋 내역 목록

\$ git log

commit 27f90f0b1... (HEAD -> master)

Author: pykwon <pykwon@hanmail.net>

Date: Thu Jun 15 14:45:18 2023 +0900

first commit

5. 특정 커밋으로 되돌리기

git reset --hard 커밋해시

지정된 커밋해시 이후의 모든 커밋을 삭제하고 지정된 커밋으로 이동된다.
따라서 지정된 커밋이 최신 커밋이 된다.

1) 파일 수정 후 여러 번 커밋하기

```
$ vi hi.txt
```

```
$ git commit -am 'third commit'
```

```
[master 7d5594e] third commit
```

```
1 file changed, 2 insertions(+)
```

```
...
```

```
$ git log
```

```
commit 189ad0ea71135188933f6e222c0ba8030bc2c125 (HEAD -> master)
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
    fives commit
```

```
commit cb1c67e68dcc7c3f1e8a7d18b4132c69f9d6188
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
    forth commit
```

```
commit 7d5594e001cb380e5c4ea179dadf1a6ce0456157
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
    third commit
```

```
commit 27f90f0b1063a73e31e9295c2cea502e6de3796e
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
    first commit
```

```
$ cat hi.txt
```

```
hi
```

```
fine
```

```
good1
```

```
good2
```

```
good3
```

5. 특정 커밋으로 되돌리기

2) 특정 커밋으로 되돌리기 (second commit 으로 되돌리기)

```
commit 7d5594e001cb380e5c4ea179dadf1a6ce0456157
Author: pykwon <pykwon@hanmail.net>
third commit
```



```
$ git reset --hard
7d5594e001cb380e5c4ea179dadf1a6ce0456157
HEAD is now at 7d5594e third commit
```

3) 커밋 내역 목록

```
$ git log
commit 7d5594e001cb380e5c4ea179dadf1a6ce0456157 (HEAD -> master)
Author: pykwon <pykwon@hanmail.net>
Date: Thu Jun 15 16:28:44 2023 +0900
third commit
commit 27f90f0b1063a73e31e9295c2cea502e6de3796e
Author: pykwon <pykwon@hanmail.net>
Date: Thu Jun 15 14:45:18 2023 +0900
first commit
```

4) 현재 파일 내용

```
$ cat hi.txt
hi
fine
good1
```


브랜치 (Branch)

1. 브랜치 개요

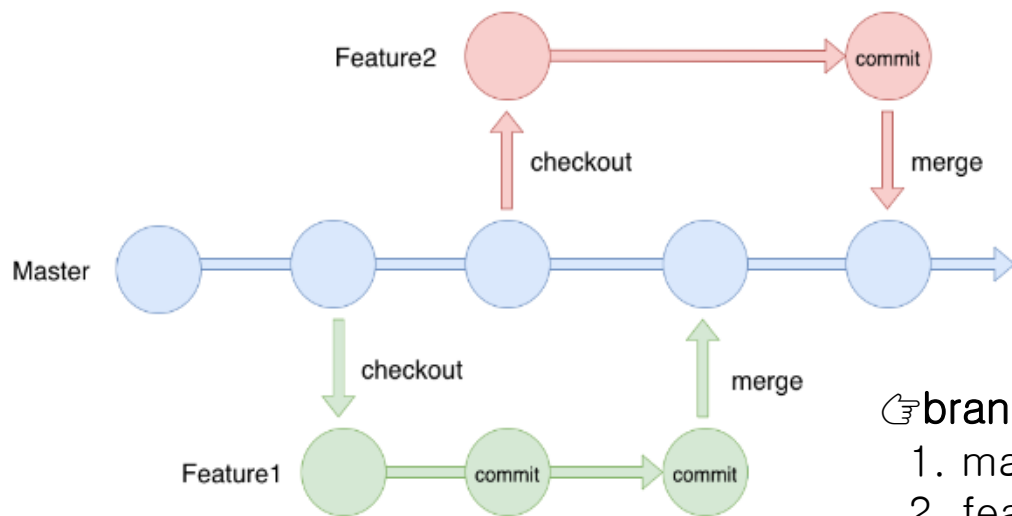
브랜치 용도

기존 파일에 새로운 기능을 추가하고자 할 때, 제대로 동작하는 소스는 그대로 둔 채 새로운 소스를 추가한 버전을 따로 만들어 관리하는 경우에 사용된다.

브랜치 기능

git으로 버전 관리를 시작하면 기본적으로 master 브랜치가 만들어진다. 사용자가 커밋할 때 마다 master 브랜치는 최신 커밋을 가리킨다.

이 때 새로운 브랜치를 만들면 기존에 저장한 파일은 master 브랜치에 그대로 유지하면서 기존 파일 내용을 수정하거나 새로운 기능을 구현할 파일을 만들 수 있다.



branch에서 merging하는 과정

1. master branch를 checkout 한다.
2. feature branch를 각자 만든다.
3. feature branch에서 각자 개발한다.
4. 개발이 끝나면 commit 한다.
5. feature branch를 master branch와 합친다(merge).

1. 브랜치 개요

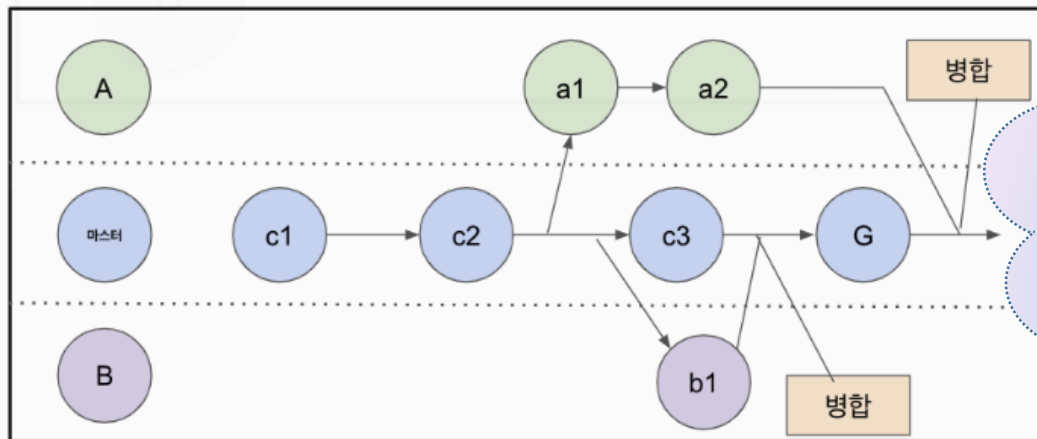
버전 관리 시스템은 브랜치를 지원한다. 개발을 하다 보면 코드를 여러 개로 복사해야 하는 일이 자주 생긴다. 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발을 진행할 수 있는데, 이렇게 독립적으로 개발하는 것이 브랜치다.

브랜치 모델이 Git의 가장 큰 장점이며, Git이 다른 것들과 구분되는 특징이라고 할 수 있다. 도대체 어떤 점이 그렇게 특별한 것일까?

Git의 브랜치는 매우 가볍다. 순식간에 브랜치를 새로 만들고 브랜치 사이를 이동할 수 있다.

Git은 브랜치를 만들어 작업하고 나중에 Merge 하는 방법을 권장한다.

결국 Git 브랜치에 능숙해지면 융통성 있는 프로젝트 개발이 가능해질 것이다.



Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다.

기본적으로 Git은 master 브랜치를 만든다. 처음 커밋하면 이 master 브랜치가 생성된 커밋을 가리킨다.

이후 커밋을 만들면 master 브랜치는 자동으로 가장 마지막 커밋을 가리킨다.

분기했던 새로운 브랜치에서 원하는 작업을 모두 끝낸 후 새 브랜치에 있던 파일을 원래 master 브랜치에 합칠 수 있다. (병합, merge)

2. 브랜치 관리에 필요한 기본 명령어

명령어	기 능
git branch	브랜치 목록 보기
git branch 이름	지정된 이름의 새로운 브랜치 생성하기
git branch -d 이름	병합 후에 기존 브랜치 삭제하기 병합 전에 브랜치 삭제하기 위해서는 -D 옵션 사용한다.
git checkout 이름	브랜치 변경(이동)하기
git merge 이름	현재 작업 중인 브랜치에서 지정된 이름의 브랜치를 가져와서 병합하기
git log --oneline --branches	모든 branch 커밋 이력을 한 줄로 출력하기
git log master..브랜치명	master 브랜치 기준으로 브랜치명 브랜치와의 차이점 출력

3. 브랜치 관리

1) 브랜치 목록 보기

```
$ git branch
```

- * master

2) 브랜치 생성 git branch 브랜치명

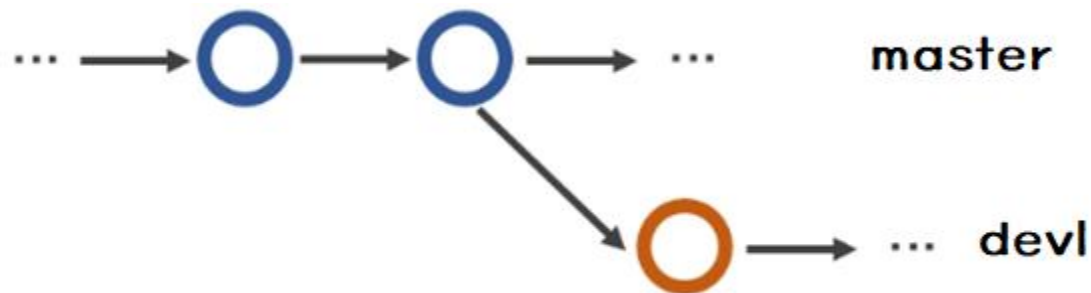
```
$ git branch dev1
```

```
$ git branch
```

dev 1

- * master

← * 는 현재 사용 중인 브랜치를 의미한다.



3) 브랜치 변경

```
$ git checkout dev1
```

Switched to branch 'dev1'

← master 브랜치에서 dev1 브랜치로 변경한다.

```
$ git branch
```

- * dev1

master

지금부터 하는 모든 작업은 dev1 브랜치만 영향을 미친다. (파일 수정, 삭제 등) 마음껏 작업하고 commi한 후에 master 브랜치로 checkout하면, dev1에서 했던 모든 작업을 해당 브랜치의 최종 commit 상태로 보존한 후에 필요한 경우, master 브랜치의 최종 커밋 상태로 파일이 변경된다.

3. 브랜치 관리

4) dev1 브랜치에서 파일 수정 및 생성

```
$ cat hi.txt
```

```
hi  
fine  
good1
```

```
$ vi hi.txt
```

```
$ cat hi.txt
```

```
hi  
fine  
good1  
good2 by dev1
```

```
$ git commit -am 'good2 by dev1 commit'
```

```
[dev1 3aee448] good2 by dev1 commit  
1 file changed, 1 insertion(+)
```

```
$ touch main.html
```

```
$ touch list.html
```

```
$ ls
```

```
hi.txt list.html main.html
```

```
$ git add list.html
```

```
$ git commit -m 'list by dev1 commit'
```

```
[dev1 92f4c79] list by dev1 commit  
1 file changed, 0 insertions(+), 0  
deletions(-)
```

```
create mode 100644 list.html
```

```
$ git status
```

```
On branch dev1
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what  
will be committed)
```

```
main.html
```

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

3. 브랜치 관리

5) dev1 브랜치에서 커밋 이력

```
$ git log --graph
* commit 92f4c79a3801705496c0e4f5318b56d7611e4811 (HEAD -> dev1)
| Author: pykwon <pykwon@hanmail.net>
|| list by dev1 commit
|
* commit 3aee44890a1999dbf3e63b5381a44268ac6bf85d
| Author: pykwon <pykwon@hanmail.net>
| good2 by dev2 commit
|
* commit 7d5594e001cb380e5c4ea179dadf1a6ce0456157 (master)
| Author: pykwon <pykwon@hanmail.net>
| third commit
|
* commit 27f90f0b1063a73e31e9295c2cea502e6de3796e
  Author: pykwon <pykwon@hanmail.net>
    first commit
```

6) 모든 브랜치에서 커밋 이력

```
$ git log --oneline --branches
92f4c79 (HEAD -> dev1) list by dev1 commit
3aee448 good2 by dev1 commit
7d5594e (master) third commit
27f90f0 first commit
```

3. 브랜치 관리

7) master 브랜치 기준으로 dev1 브랜치 차이점 출력

```
$ git log
```

```
commit 7d5594e001cb380e5c4ea179dadf1a6ce0456157 (HEAD -> master)
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
third commit
```

```
commit 27f90f0b1063a73e31e9295c2cea502e6de3796e
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
first commit
```

```
$ git log master ..dev1
```

```
commit 92f4c79a3801705496c0e4f5318b56d7611e4811 (dev1)
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
list by dev1 commit
```

```
commit 3aee44890a1999dbf3e63b5381a44268ac6bf85d
```

```
Author: pykwon <pykwon@hanmail.net>
```

```
good2 by dev2 commit
```

master 브랜치에는 없고 dev1 브랜치에만 있는 커밋 정보 출력된다.

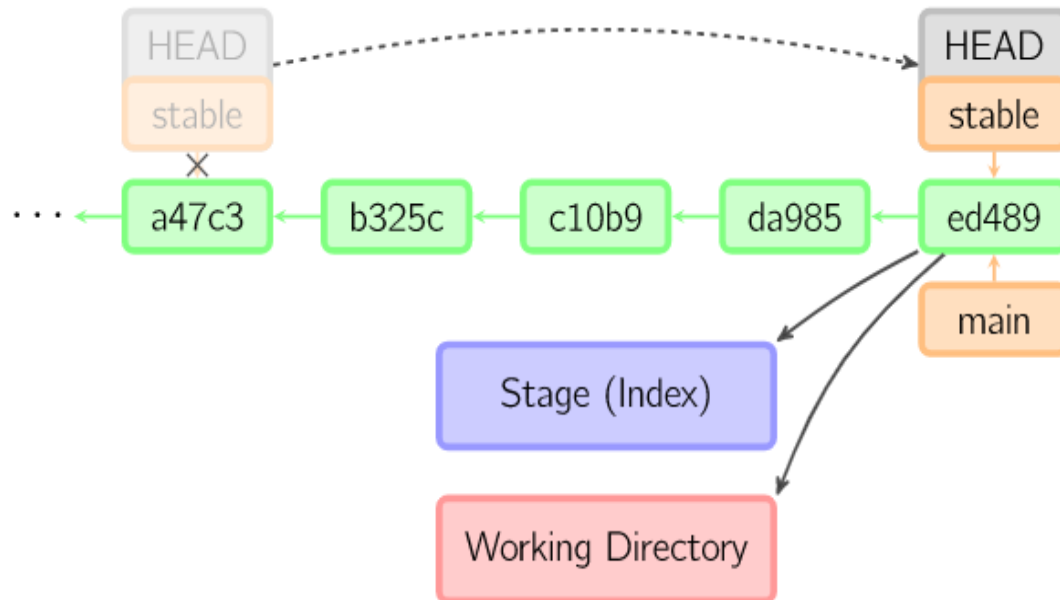
4. 브랜치 병합

Merge :

Merge 명령은 다른 커밋들을 하나로 합쳐서 새로운 커밋을 만든다. Merge 명령을 실행하기 전에 Stage 영역에 작업 중인 파일이 없는지 꼭 확인해 둔다.

Merge하는 가장 간단한 경우는 Merge할 대상이 현재 커밋의 직접적인 뿌리가 되는 경우인데, 이 때는 합칠 내용이 없다. 다음은 현재 커밋이 Merge할 대상의 직접적인 뿌리가 되는 경우인데, 이 때는 fast-forward Merge가 실행되는데 간단히 가리키는 지점이 대상 커밋이 되고 대상 커밋의 내용을 Checkout 한다.

```
git merge main
```



4. 브랜치 병합(merge)

작성된 각 브랜치에서 작업을 하다가 어느 시점에는 브랜치 작업을 마무리하고 기존 master 브랜치와 합해야 된다. 이것을 브랜치 병합(merge)이라고 부른다.

1) master 브랜치 확인

```
$ git branch
dev1
* master
```

2) master 브랜치와 dev1 브랜치 병합

```
$ git merge dev1
Updating 7d5594e..92f4c79
Fast-forward
 hi.txt   | 1 +
 list.html | 0
2 files changed, 1 insertion(+)
create mode 100644 list.html
```

master 브랜치에서 분기한 후에 master 브랜치에서 변경 사항이 없는 경우에 분기한 브랜치를 병합하는 것은 매우 간단하다. 분기한 브랜치에서 만든 최신 커밋을 master 브랜치가 참조하도록 하면 된다. 이것을 Fast-forward (빨리 감기) 병합이라고 한다.

4. 브랜치 병합

3) master 커밋 이력 보기

```
$ git log
```

```
commit 92f4c79a3801705496c0e4f5318b56d7611e4811 (HEAD -> master, dev1)
```

```
list by dev1 commit
```

```
commit 3aee44890a1999dbf3e63b5381a44268ac6bf85d
```

```
good2 by dev1 commit
```

```
commit 7d5594e001cb380e5c4ea179dadf1a6ce0456157
```

```
third commit
```

```
commit 27f90f0b1063a73e31e9295c2cea502e6de3796e
```

```
first commit
```

5. 브랜치 충돌

git에서는 줄 단위(line)로 변경 여부를 확인한다. 따라서 브랜치에 같은 파일의 이름을 가지고 있으면서 같은 줄을 수정하고 병합하면 브랜치 충돌(conflict)이 발생된다.

해결방법은 명시적으로 사용자가 충돌위치를 수정하고 커밋해서 처리한다.

1) 현재 상태 보기

```
acorn@DESKTOP-.../c/work/git_test2 (master)
$ git log master..dev1
acorn@DESKTOP-.../c/work/git_test2 (master)
$ git log dev1..master
```

2) master 브랜치에서 hi.txt 수정

```
acorn@DESKTOP-... /c/work/git_test2 (master)
$ vi hi.txt
$ cat hi.txt
hi
fine
good1
good2 by dev1
good3 by master

$ git add hi.txt
$ git commit -m 'good3 by master commit'
[master badd9a6] good3 by master commit
1 file changed, 1 insertion(+)
```

3) dev1 브랜치에서 hi.txt 수정

```
$ git checkout dev1
Switched to branch 'dev1'

acorn@DESKTOP-.../c/work/git_test2 (dev1)
$ vi hi.txt
$ cat hi.txt
hi
fine
good1
good2 by dev1
good3 by dev1

$ git status
...
    modified:   hi.txt

$ git add hi.txt
$ git commit -m 'good3 by dev1'
[dev1 3c7ccdb] good4 by dev1
1 file changed, 1 insertion(+)
```

5. 브랜치 충돌

4) master 브랜치와 dev1 브랜치 병합

```
$ git checkout master  
Switched to branch 'master'
```

```
acorn@DESKTOP-OD3GFRF MINGW64 /c/work/git_test2 (master)
```

```
$ git merge dev1
```

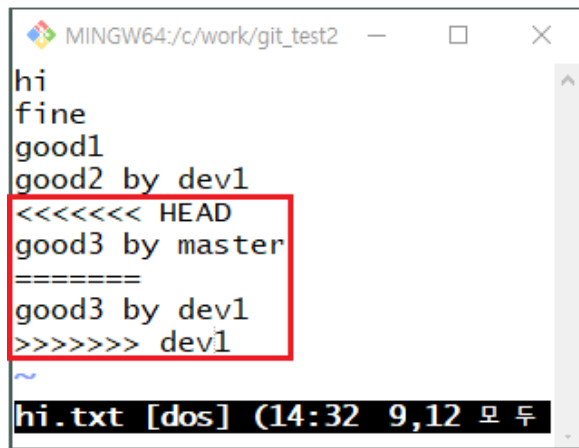
```
Auto-merging hi.txt
```

```
CONFLICT (content): Merge conflict in hi.txt
```

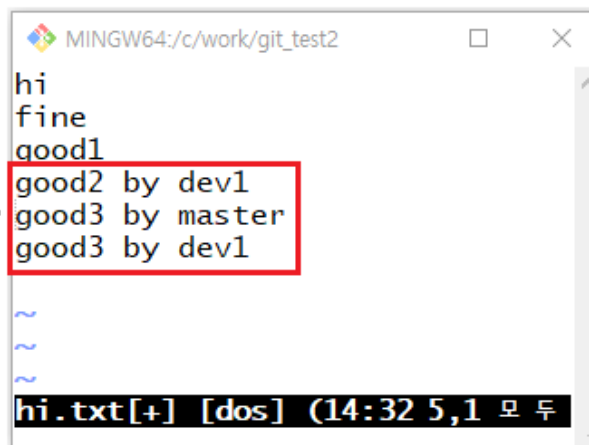
```
Automatic merge failed; fix conflicts and then commit the result.
```

```
acorn@DESKTOP-OD3GFRF MINGW64 /c/work/git_test2 (master|MERGING)
```

```
$ vi hi.txt
```



```
hi  
fine  
good1  
good2 by dev1  
<<<<<<< HEAD  
good3 by master  
=====  
good3 by dev1  
>>>>> dev1  
~  
hi.txt [dos] (14:32 9,12 모두
```



```
hi  
fine  
good1  
good2 by dev1  
good3 by master  
good3 by dev1  
~  
~  
hi.txt[+] [dos] (14:32 5,1 모두
```

충돌시
명시적으로
해결

```
acorn@DESKTOP-OD3GFRF MINGW64 /c/work/git_test2 (master|MERGING)
```

```
$ git commit -am 'master dev1 merge'
```

```
[master 8229dd8] master dev1 merge
```

← 변경사항 커밋

5. 브랜치 충돌

5) 커밋 이력 보기

```
$ git log --oneline
8229dd8 (HEAD -> master) master dev1 merge
7187e09 (dev1) good3 by dev1
badd9a6 good3 by master commit
92f4c79 list by dev1 commit
3aee448 good2 by dev1 commit
7d5594e third commit
27f90f0 first commit
```

6. 병합시킨 브랜치 삭제

```
acorn@DESKTOP-OD3GFRF MINGW64 /c/work/git_test2 (master)
```

```
$ git branch
```

```
dev1
```

```
* master
```

```
$ git branch -d dev1
```

```
Deleted branch dev1 (was 7187e09).
```

```
$ git branch
```

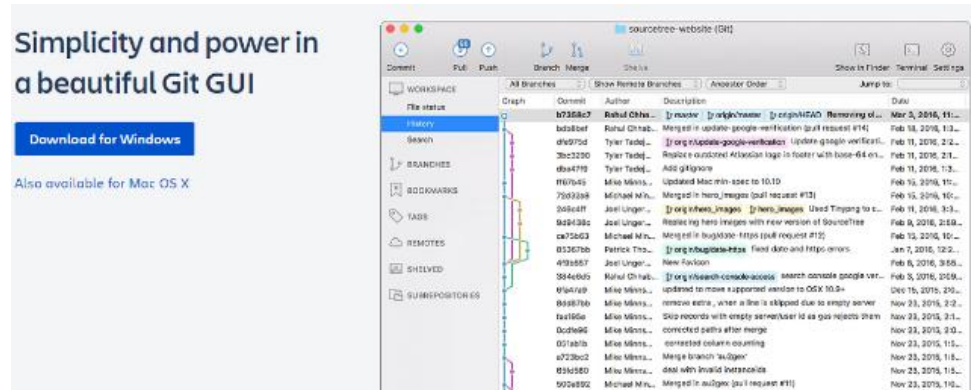
```
* master
```

11-1

Local Repository에 대한 다양한 경우에 대해 좀 더 많은 학습이 필요합니다.
검색과 실습을 통해 많은 경험을 해 보세요.

참고로 Git을 좀더 편하게 사용하기 위해 **Git GUI**를 사용할 수도 있습니다.
Git GUI 중 대표적인 툴이 바로 소스트리(SourceTree) 입니다.

수고하세요~~~



★알림★

깃허브(GitHub)는 분산 버전 관리 툴인 Git 저장소 호스팅을 지원하는 웹 서비스입니다.

깃허브는 영리적인 서비스와 오픈 소스를 위한 무상 서비스를 모두 제공합니다.
원격 저장소인 깃허브는 가장 인기 있는 Git 저장소 호스팅 서비스입니다.
우리의 두 번째 학습 목표입니다.