Git – 03

목차

Chapter 3. 로컬 저장소 사용을 위한 Git 기본

- 서론
- git init : 저장소 생성
- git add & git commit : 첫 번째 커밋
- git branch & git checkout : 새로운 브랜치 생성과 이동
- git commit –a : 두 번째 커밋
- git merge : master branch와 병합
- 각 branch의 독립성 확인
- 실제 프로젝트에서 발생하는 상황들
- .gitignore : 불필요한 파일 및 폴더 무시
- 충돌 해결
- git log : 기록 보기

서론

- Git을 사용하는 데 꼭 필요한 기본 설정과 로컬 환경에서 혼자 Git을 사용한다는 가정 아래 기본 필요한 기본 명령어 학습
  - 협업 단계는 Part 3에서 학습한다.

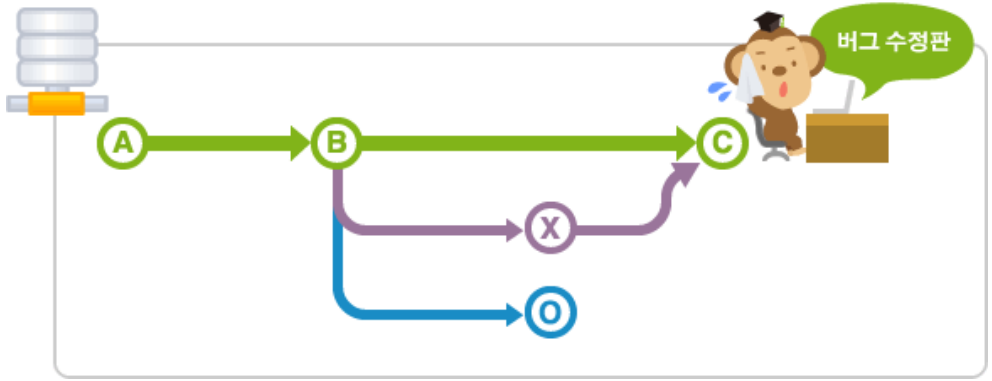
- 이번 장에서는 아래의 시나리오로 기능을 학습한다.
  - 로컬에 저장소 생성
  - 저장소에 파일 생성 및 추가
  - 추가된 파일의 수정
  - master branch에 영향을 끼치지 않는 branch 생성
  - branch merge
  - 충돌 해결
  - 저장소 기록 보기

```
git init : 실행한 위치를 git 저장소로 초기화 한다.
git add 파일이름 : 해당 파일을 Git이 추적할 수 있게 추가한다.
git commit : 변경된 파일을 저장소에 제출한다.
git status : 현재 저장소의 상태를 출력한다.
git branch 이름 : 브랜치(사본작업그룹)을 만든다.
git checkout 브랜치이름 : 현재 작업중인 브랜치이름을 변경한다.
git merge 브랜치이름 : 현재 작업중인 브랜치에서 '브랜치이름'을 가져와서 병합한다. (내꺼에 저거를 병합한다.)
```

서론

- Git의 기본적인 작업 흐름
  - 파일 생성 또는 추가 > 파일 수정 > 수정 내역을 저장소에 제출 > 파일 생성 또는 추가...

- branch 이동을 통해 변경된 작업 흐름



  - 브랜치 생성 > 임시 브랜치로 checkout > 파일 생성 또는 추가 > 파일 수정 > 수정 내역을 저장소에 제출 > 원래의 master 브랜치로 checkout > master 브랜치에 임시 브랜치를 병합 > 브랜치 생성 ...

서론

- 수많은 IDE가 Git을 IDE의 일부분으로, 또는 플러그인이나 외부 Git GUI 프로그램 등을 이용해 포함하고 있다.
  - 하지만 Git의 핵심은 command line 이다.

- 다음과 같은 순서로 세분화 하여 실습한다.
  - 저장소 생성
  - 저장소에 Hello World를 출력하는 프로그램 작성 및 추가
  - 커밋
  - hotfix 브랜치 생성 및 이동
  - 프로그램 수정
  - 커밋
  - master 브랜치에 병합
  - master 브랜치에 변경점 하나 추가
  - 커밋
  - hotfix 브랜치에 변경점 하나 추가
  - 커밋
  - master와 hotfix 브랜치 사이에 영향이 없음을 확인
  - 불필요한 프로젝트 파일을 관리 대상에서 제외하기
  - 충돌 해결
  - 기록 보기

git init

- 저장소 생성
  - mkdir git_tutorial
  - cd git tutorial
  - git init

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial
$ git init
Initialized empty Git repository in C:/Users/kyh/Desktop/git_tutorial/.git/

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$
```

```
root@netsys:~/LabSeminar/git_tutorial# git init
초기화: 빈 깃 저장소, 위치 /root/LabSeminar/git_tutorial/.git/
```

git add & git commit

- 첫 번째 커밋
  - vi hello.py
  - git status
    - commit 전 먼저 저장소 상태를 확인
    - 아직 git에서 추적하지 않은 hello.py 파일이 저장소에 있다고 알려준다.
    - 또한, 이 파일을 추적하기 위해 git add 명령을 사용해야 하는 것도 알려준다.

```
$ git status
On branch master

No commits yet

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

        hello.py

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

```
root@netsys:~/LabSeminar/git_tutorial# git status
현재 브랜치 master

최초 커밋

추적하지 않는 파일:
  (커밋할 사항에 포함하려면 "git add <파일>..."을 사용하십시오)

        hello.py

커밋할 사항을 추가하지 않았지만 추적하지 않는 파일이 있습니다 (추적하려면 "git
add"를 사용하십시오)
```

git add & git commit

- 첫 번째 커밋
  - git add hello.py
    - 아무런 메시지가 없으면 성공적으로 Git이 추적을 시작할 수 있게 저장소에 추가된 것이다.
    - 제대로 추가 되었는지를 확인하려면 git status 명령을 다시 입력한다.

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git add hello.py

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git status
On branch master

No commits yet

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

        new file:   hello.py
```

```
root@netsys:~/LabSeminar/git_tutorial# git add hello.py
root@netsys:~/LabSeminar/git_tutorial# git status
현재 브랜치 master

최초 커밋

커밋할 변경 사항:
  (스테이지 해제하려면 "git rm --cached <파일>..."을 사용하십시오)

        새 파일:       hello.py
```

git add & git commit

- 첫 번째 커밋
  - git commit
    - 명령을 수행하면 commit massage를 작성하는 화면이 나온다.

```
 GNU nano 2.5.3 파일: .../LabSeminar/git_tutorial/.git/COMMIT_EDITMSG 변경됨.

create "Hello World" program
# 변경 사항에 대한 커밋 메시지를 입력하십시오. '#' 문자로 시작하는
# 줄은 무시되고, 메시지를 입력하지 않으면 커밋이 중지됩니다.
# 현재 브랜치 master
#
# 최초 커밋
#
# 커밋할 변경 사항:
#       새 파일:       hello.py
#
```

    - commit message를 작성한다.

```
$ git commit
[master (root-commit) 1462274] create "Hello World" program
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

```
root@netsys:~/LabSeminar/git_tutorial# git commit
[master (최상위-커밋) e3cd1e0] create "Hello World" program
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

NSL

git branch & git checkout

- 새로운 브랜치 생성과 이동
  - 기본적으로 Git을 사용할 때는 전과 같이 파일을 추가하고, 수정하고, 커밋하면서 프로젝트를 진행하면 된다.
  - 이번에는 기존 프로젝트에 영향이 가지 않는 새로운 실험적 기능을 추가해야 하거나, 기존 기능을 변경해야 하는 경우를 고려해보자.

  - git branch
    - 어떤 branch가 있는지 볼 수 있다.

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git branch
* master
```

```
root@netsys:~/LabSeminar/git_tutorial# git branch
* master
```

NSL

git branch & git checkout

- 새로운 브랜치 생성과 이동
  - git branch hotfix
    - 브랜치를 생성한다.

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git branch hotfix

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git branch
  hotfix
* master
```

```
root@netsys:~/LabSeminar/git_tutorial# git branch hotfix
root@netsys:~/LabSeminar/git_tutorial# git branch
  hotfix
* master
```

  - branch의 이름 앞에 *표시는 현재 작업 중인 branch를 나타낸다.

git branch & git checkout

- 새로운 브랜치 생성과 이동
  - git checkout hotfix
    - hotfix branch로 이동

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git checkout hotfix
Switched to branch 'hotfix'
```

```
root@netsys:~/LabSeminar/git_tutorial# git checkout hotfix
'hotfix' 브랜치로 전환합니다
```

  - git checkout -b hotfix
    - 브랜치를 만들면서 바로 checkout 할 수 있다.
  - 지금부터 하는 작업은 오직 hotfix 브랜치에만 영향을 끼치게 된다.
    - 파일 수정, 삭제, 추가 등등
  - 작업을 완료한 후 다시 master branch로 checkout하면, hotfix에서 작업한 모든 것을 해당 브랜치의 최종 커밋 상태로 보존한 후, master branch의 최종 커밋 상태로 파일들이 변경된다.

git branch & git checkout

- 새로운 브랜치 생성과 이동
  - hello.py 코드를 수정한다.

```
kyh@DESKTOP-0UBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ cat hello.py
print("Hello World")
print("Branch Test")
```

```
root@netsys:~/LabSeminar/git_tutorial# python3 hello.py
Hello World
Branch Test
```

git commit -a

- 두 번째 커밋
  - git status

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

        modified:   hello.py

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

```
root@netsys:~/LabSeminar/git_tutorial# git status
현재 브랜치 hotfix
커밋하도록 정하지 않은 변경 사항:
  (무엇을 커밋할지 바꾸려면 "git add <파일>..."을 사용하십시오)
  (작업 폴더의 변경 사항을 버리려면 "git checkout -- <파일>..."을 사용하십시오)

        수정함:        hello.py

커밋할 변경 사항을 추가하지 않았습니다 ("git add" 및/또는 "git commit -a"를
사용하십시오)
```

  - 변경된 내역이 커밋될 준비가 되지 않았다.
    - git add 이후 git commit 하거나 또는
    - git commit -a 하여 변경된 저장소 파일 모두를 커밋하면 된다.

NSL

git commit -a

- 두 번째 커밋
  - git commit –a
  - commit message로는 added output “Branch Test”를 넣어준다.

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ git commit -a
[hotfix 0c3d2d2] added output "Branch Test"
 1 file changed, 1 insertion(+)
```

  - git status
    - 커밋할 것이 없다는 메시지 확인 가능

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ git status
On branch hotfix
nothing to commit, working tree clean
```

git commit -a

- 두 번째 커밋
  - commit message는 반드시 남겨야 한다.
    - vim을 사용하기 번거롭다면 아래의 명령을 이용한다.

  - git commit –m “message” 또는
  - git commit –am “added output ‘Branch Test’”

```
root@netsys:~/LabSeminar/git_tutorial# git commit -am "added output 'Branch Test'"
[hotfix 20ed8e7] added output 'Branch Test'
 1 file changed, 1 insertion(+)
root@netsys:~/LabSeminar/git_tutorial# git status
현재 브랜치 hotfix
커밋할 사항 없음, 작업 폴더 깨끗함
```

NSL

git merge

- master branch와 병합하기
  - git checkout master
  - git status
  - cat hello.py

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ git checkout master
Switched to branch 'master'

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git status
On branch master
nothing to commit, working tree clean

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ cat hello.py
print("Hello World")
```

  - master branch로 checkout하고, hello.py의 내용을 관찰한다.
    - hotfix branch가 아니라 master branch로 돌아온 것을 확인할 수 있다.

git merge

- master branch와 병합하기
  - git merge hotfix
    - 현재 작업중인 branch에 hotfix branch를 가져와서 병합한다.

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git merge hotfix
Updating 1462274..0c3d2d2
Fast-forward
 hello.py | 1 +
 1 file changed, 1 insertion(+)

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ cat hello.py
print("Hello World")
print("Branch Test")
```

  - hello.py 파일에 1행이 추가되었으며, 병합이 제대로 된 것을 확인할 수 있다.

  - git merge master 명령을 썼다면?
    - hotfix branch에서 위의 명령을 사용했다면, master branch의 내용은 변경되지 않고, master branch의 내용을 기준으로 hotfix branch에 병합된다.

각 branch의 독립성 확인

- 세 번째 커밋
  - hello.py를 수정한다.
    - print("Check point 1") 추가
    - print("Check point 2") 추가
  - git commit -a
  - commit message는 added output "check point"

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git commit -a
[master 8026d2b] added output "check point"
 1 file changed, 2 insertions(+)
```

```
root@netsys:~/LabSeminar/git_tutorial# git commit -a
[master 0c08c5d] added output "check point"
 1 file changed, 2 insertions(+)
```

  - master branch의 hello.py가 수정되었다.

각 branch의 독립성 확인

- 네 번째 커밋
  - git checkout hotfix
  - hello.py 수정
    - print("hotfix check point") 추가
  - git commit –a
    - commit message는 added output "hotfix check point"

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

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ vi hello.py

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ git commit -a
[hotfix e5c6494] added output "hotfix check point"
 1 file changed, 1 insertion(+)

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (hotfix)
$ cat hello.py
print("Hello World")
print("Branch Test")
print("hotfix check point")
```

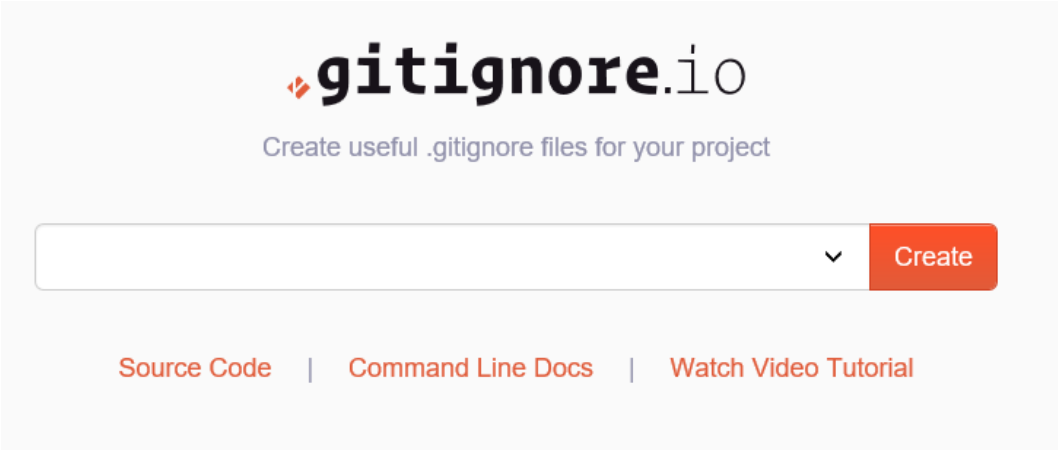
  - 결과는 hotfix branch의 hello.py가 독립적으로 수정되었다는 것을 보여준다.

실제 프로젝트에서 발생하는 상황들

- 파일 무시, 충돌 해결, commit 내역 확인
  - 프로젝트를 진행하다 보면 부수적으로 다양한 파일이 만들어진다.
    - Git이 굳이 추적해야 할 필요가 없는 파일들
    - 보통 입, 출력용 데이터나 각종 로그 파일들이 이에 해당된다.
  - 저장할 필요가 없는 파일들을 무시하기 위해 .gitignore 라는 파일을 이용한다.
    - 다른 방법들도 있지만, 이것으로 충분하다.
  - 이전의 실습에서는 master, hotfix 두 개의 branch만 다루었다.
    - 어느 branch에서 무엇을 수정했는지 딱히 비교해보지 않아도 되었다.
    - 하지만 한 번에 서너 개 혹은 그 이상의 브랜치를 사용하는 개인 프로젝트라면, 버그가 발생한 코드 부분마다 브랜치를 하나씩 따로 생성해서 조금씩 고쳐나가게 되고, 언젠가 각 브랜치가 하나의 파일을 동시에 수정하는 경우가 생긴다.
    - 그리고 master 브랜치로 병합하면 두 브랜치에서 동시에 같은 파일을 수정했다고 충돌이 일어난다.
  - 마지막으로 모든 작업을 마치고 commit 로그를 살펴본다.
    - 지금까지 무엇이 어떻게 진행되었고, 어떤 브랜치에 어떤 브랜치가 병합되었는지 등을 알 수 있다.

.gitignore

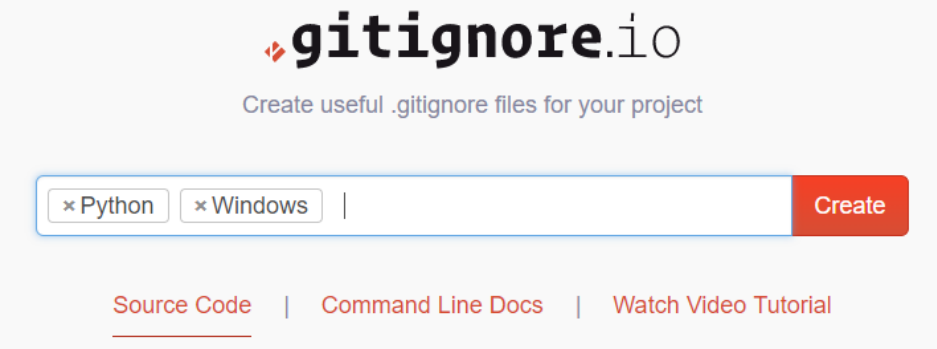
- 불필요한 파일 및 폴더 무시
  - touch .gitignore
    - 파일 생성
  - 해당 파일은 일련의 파일 목록과 파일을 구분할 수 있는 패턴의 모음으로 라인 하나가 패턴 하나를 가리킨다.
    - 자세한 내용은 http://git-scm/docs/gitignore 참조
  - 운영체제나 개발환경에 맞춰서 파일을 자동으로 생성해주는 웹 어플리케이션 : https://www.gitignore.io/



NSL

.gitignore

- 불필요한 파일 및 폴더 무시
  - 웹 어플리케이션에서 사용중인 운영체제, IDE, 프로그래밍 언어 등을 입력하고 생성한다.



  - 생성 후 안의 내용을 .gitignore 파일에 복사

  - git add .gitignore
  - git commit –m "added '.gitignore' file

  - 이제 커맨드 라인으로 파이썬 프로젝트를 작업할 때 불필요한 파일이 Git 저장소에 추가되는 것을 방지할 수 있다.

충돌 해결

- 충돌 해결
  - git checkout master
  - git merge hotfix
  - cat hello.py

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

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
$ git merge hotfix
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master|MERGING)
$ cat hello.py
print("Hello World")
print("Branch Test")
<<<<<<< HEAD
print("Check point 1")
print("Check point 2")
=======
print("hotfix check point")
>>>>>>> hotfix
```

  - <<<<<<<HEAD는 충돌이 발생한 시작 지점을 나타내고, >>>>>>> hotfix는 끝 부분을 나타낸다.
  - 중간에 ======= 표시로 어디까지가 어디에 속해 있는지 경계를 표시해준다.

충돌 해결

- 충돌 해결
  - Git은 해당 행이 어떤 의미를 지니고 있는지 알 수 없으므로 어떤 수정 사항을 선택할 것인지 사용자에게 일임한다.
  - 이러한 경우 충돌이 발생한 두 브랜치 중 하나의 내용을 선택하거나, 두 수정 내역을 합치는 등 수동으로 충돌을 해결해야 한다.
    - 즉, 충돌을 해결하기 위해서는 해당 부분의 의미를 이해하고 있어야 한다.
  - 편집기를 열고 아래와 같이 수정한다.

```
print("Hello World")
print("Branch Test")
print("Check point 1")
print("Check point 2")
print("hotfix check point")
```

  - 충돌을 직접 해결했으니, 다시 commit 한다.
  - git commit -a -m “conflict resolved”

```
kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master|MERGING)
$ git commit -a -m "conflict resolved"
[master 391e0c6] conflict resolved

kyh@DESKTOP-OUBU3AH MINGW64 ~/Desktop/git_tutorial (master)
```

git log

- 기록 보기
  - git log
    - 아래는 옵션

옵션	설명
`-p`	각 커밋에 적용된 패치를 보여준다.
`--word-diff`	diff 결과를 단어 단위로 보여준다.
`--stat`	각 커밋에서 수정된 파일의 통계정보를 보여준다.
`--shortstat`	`--stat` 명령의 결과 중에서 수정한 파일, 추가된 줄, 삭제된 줄만 보여준다.
`--name-only`	커밋 정보중에서 수정된 파일의 목록만 보여준다.
`--name-status`	수정된 파일의 목록을 보여줄 뿐만 아니라 파일을 추가한 것인지, 수정한 것인지, 삭제한 것인지도 보여준다.
`--abbrev-commit`	40자 짜리 SHA-1 체크섬을 전부 보여주는 것이 아니라 처음 몇 자만 보여준다.
`--relative-date`	정확한 시간을 보여주는 것이 아니라 `2 주전`처럼 상대적인 형식으로 보여준다.
`--graph`	브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
`--pretty`	지정한 형식으로 보여준다. 이 옵션에는 oneline, short, full, fuller, format이 있다. format은 원하는 형식으로 출력하고자 할 때 사용한다.
`--oneline`	`--pretty=oneline --abbrev-commit` 옵션을 함께 사용한 것과 동일하다.

git log

- 기록 보기
  - git log --graph
    - 기본적으로 40글자의 SHA-1 checksum 값, 커밋한 사용자, 커밋 시각, 커밋 메시지 등의 내역을 확인할 수 있다.
    - 왼쪽의 녹색과 빨간색 세로 점선은 분기 내역을 보여준다.

```
root@netsys:~/LabSeminar/git_tutorial# git log --graph
*   commit 647e17ad35ef28562724fad51ecbad26d941b02d
|\  Merge: 0c08c5d d1515ea
| | Author: kyh <yh0921k@gmail.com>
| | Date:   Mon Feb 5 16:07:43 2018 +0900
| |
| |     conflice resolved
| |
| * commit d1515eaa0602fcdcc9be0a8b4ac0eacf52b31b48
| | Author: kyh <yh0921k@gmail.com>
| | Date:   Mon Feb 5 14:53:18 2018 +0900
| |
| |     added '.gitignore' file
| |
| * commit 8870789a65edf5964669accaf2861bdda3e85909
| | Author: kyh <yh0921k@gmail.com>
| | Date:   Mon Feb 5 14:15:30 2018 +0900
| |
| |     added output "hotfix check point"
| |
* | commit 0c08c5d8cc4ada2935fe70ce6cbf3d94cc816f18
|/  Author: kyh <yh0921k@gmail.com>
|   Date:   Mon Feb 5 13:58:34 2018 +0900
|
|       added output "check point"
|
* commit 20ed8e75e4c7e7051f1df4b68680a91bc823a8a2
| Author: kyh <yh0921k@gmail.com>
| Date:   Mon Feb 5 13:35:39 2018 +0900
|
|     added output 'Branch Test'
|
* commit e3cd1e0c14d08d3ae21a9af9691dc3b2730fe2c4
  Author: kyh <yh0921k@gmail.com>
  Date:   Mon Feb 5 10:09:37 2018 +0900

      create "Hello World" program
```

NSL