◈ 오픈소스 소프트웨어
[9주차]

● OSS
:누구나 제한 없이 그 코드를 보고 사용 할 수 있는 오픈소스 라이선스를 만족하는 SW

(= 카피레프트[Copyleft]: 저작권은 인정하되 소프트웨어 복제를 통해 사용권리를 줌)
- 소프트웨어의 소스코드를 자유롭게 읽고 수정 및 재배포가 가능
- 리처드 스톨먼이 자유 소프트웨어 재단을 설립하며 출발함.
*일반적으로 소스코드를 변경하고, 변경한 소프트웨어를 공유하려면,
원래 프로젝트에 해당 변경 사항을 다시 공유해야한다.

● 오픈소스 지원 관리 서버
- 소스코드를 통해 여러 개발자가 협업하고 공유하며 이를 지원하는 방안을 마련
EX) github, gitlab, bitbucket

● OSS 장단점

장점
- 소스코드를 공개
- 커스터마이징과 혁신 지원

단점
- 공개의 의무
- 품질보증 및 유지보수, 보안 등의 어려움

● 오픈소스 개발 모델

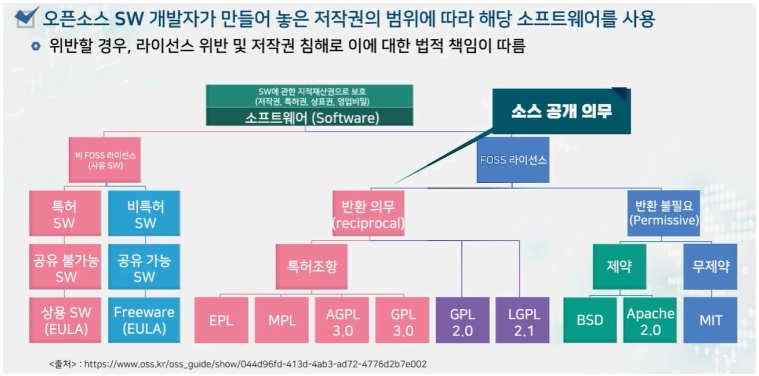
“LAMP”

-Linux : 세계 최대 규모의 오픈소스 프로젝트
-Apache : 초기 웹에서 핵심 역할을 한 크로스 플랫폼 웹 서버
-MySQL: 데이터베이스 기반 웹 앱에서 사용하는 오픈소스 관계형 데이터베이스 관리 시스템
-PHP : 소프트웨어 개발에 사용되는 범용 스크립트 언어

●오픈소스 소프트웨어 라이선스 종류

- GNU
- MIT License
- Apache
- BSD
- MySQL
- SUSE
- Ubuntu

● 오픈소스 라이선스 저작권



*반환이란?
: 오픈소스를 가지고 새로운 것을 개발했을 때 개인만이 아닌 공적으로 사용 가능하게 여는것.

- MIT License 는 반환이 불필요한 라이선스 중에서도 가장 제약이 없다.

● GPL, AGPL, LGPL, MPL, EPL 등
- 링크되거나 코드가 포함된 SW의 소스코드를 공개
- 특허, 영업비밀, 핵심 기술 등이 외부로 유출될 가능성이 있으니 주의
- 저작권 고지 의무

1. GPL(GNU - General Public License)
   - 자유소프트웨어재단에서 만든 라이선스,
      - 프로그램은 목적이나 형태의 제한 없이 사용 가능
      - 프로그램을 이후 수정하고 배포하는 모든 경우에 무조건 GPL로 공개해야함.

   - GPL 라이선스 적용 예
      - 리눅스 커널, Git, 마리아 DB 등..

2. AGPL(GNU - Affero GPL)
   - GPL을 기반으로 만든 라이선스
      - 서버에서 프로그램을 실행해서 다른 사용자들과 통신 중 프로그램의 소스를
        사용자들이 다운로드 할 수 있도록 해야하는 조항을 포함.

   - AGPL 라이선스 적용 예
      - 몽고DB 등

3. LGPL(GNU - Lesser GPL)
   - 보다 완화된 GPL 라이선스

- 전체 소스코드를 공개하지 않고 사용된 오픈소스 라이브러리에 대한 소스코드만 공개

- LGPL 라이선스 적용 예
  - 파이어폭스(v2.1) 등

4. Apache License
   - 소스코드 공개에 대한 의무사항은 라이선스에 포함되어 있지 않음
     - 아파치 라이선스 소스코드를 수정하고 배포 시에 아파치 라이선스 버전 2.0을 꼭 포함시켜야 하고 아파치 재단의 소프트웨어라는 사실을 명시

   - Apache License 적용 예
     - 안드로이드, 하둡 등

5. MIT License
   - MIT에서 학생들을 지원하기 위해서 만든 라이선스
     - 라이선스 저작권 관련 명시 의무
     - 가장 느슨한 조건을 가지고 있어서 많은 사람들이 사용하기용이
   - MIT 라이선스 적용 예
     - 부트스트랩, Angular.js 등…

● 의무 강도에 따른 분류

카테고리	소스코드 공개 의무	사례
Strong copy	2차 저작물 소스코드 공개 의무 있음	GPL
Weak copyleft	특정 조건에서 2차 저작물 소스코드 공개 의무 없음	LGPL
Non copyleft	2차 저작물 소스코드 공개 의무 없음	Apache, BSD, MIT

GPL > LGPL > Apache, BSD, MIT

◈ 임시 저장 stash

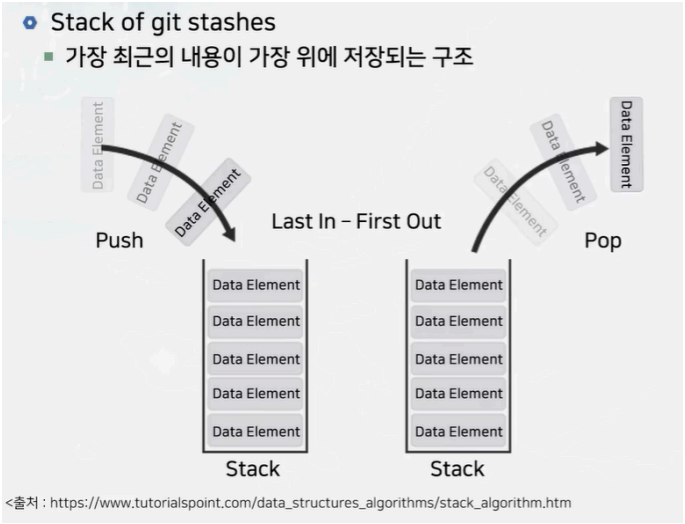
● 깃 4영역
- 깃 에는 작업 디렉토리, 스테이징 영역, 깃 저장소로 3 영역이 존재한다고 했다.
거기에 임시 저장소(스택구조)인 stash를 합해서 4영역이라고 부른다.

● Git Stash
- 사전적 의미 stash : 놓다, 남겨두다

-커밋할 필요 없이 파일의 변경 사항을 숨기거나 비밀리에 저장할 수 있는 도구
- 작업 디렉토리와 스테이징 영역의 값을 깃 저장소의 마지막 커밋 내용으로 복사

- 따로 안전한 곳에 저장했다가 다시 가져올 수 있는 기능

● 스택 구조



- Push : 자료를 넣음
- Pop : 자료를 꺼냄

● Git stash 필요성
- 브랜치 이동 또는 이전 커밋으로 이동할때, 커밋할게 없고, 작업 트리가 깨끗해야한다.
- 수정내용이 있거나 커밋할 게 있는 상황에서 브랜치 이동 또는 이전 커밋으로 이동하려면 현재 작업 공간의 수정 내용과 인덱스의 내용을 보관할 필요가 있음

● Stash로 저장되는 내용
- 작업 디렉토리 내용과 스테이징 영역 내용이 stash에 저장되고

작업 디렉토리 내용과 스테이징 영역 내용이 최신 커밋 자료로 남음

● 임시 저장 명령 stash

- 작업 폴더와 스테이징 영역을 stash에 저장하고 작업 폴더를 정리
$ git stash
$ git stash -m '메시지'
$ git stash save
$ git stash save '메시지'

- 옵션
--keep-index, -k: 스테이징 영역은 제외하고 작업 폴더만 저장
--include-untracked, -u: Untracked 파일도 포함해 저장

● 최근 임시저장 내용을 가져와 반영, stash 목록은 그대로
- 기본으로 작업 디렉토리 내용만 다시 복사해 활용
$ git stash apply
- 스테이지 영역도 함께 복사하기 위해서는 옵션 사용
$ git stash apply --index

*저장할 때는 작업/스테이징 모두를 저장, 불러올때는 작업에서만 불러옴

● 주요 옵션
- 옵션 -k | --keep-index
: 스테이징 영역은 저장하지 않고 그대로 놔둠 -> checkout 불가능
- 옵션 -u | --include-untracked
: untracked 파일도 함께 저장
- 옵션 -p | --patch
: 대화형 프롬프트를 통해 자신이 stash에 저장할 파일과 저장하지 않을 파일 지정가능.

● 목록 보기
$ git stash list
ex) stash@{0} : WIP on main ~
stash@{1}
stash@{2}
* 가장 최신 것이 0번

● 특정 stash 확인
- 해당 stash 항목이 생성되었을 때의 커밋 자료와 최신 stash 항목 간의 차이로 표시
$ git stash show
ex) h.txt | 2 ++
1 file changed, 2 insertions(+)

$ git stash show -p

- 해당 stash 항목이 생성되었을 때의 커밋 자료와 해당 stash 항목 간의 차이로 표시
$ git stash show stash@{n}
$ git stash show stash@{n} -p

*-p: 파일 내용의 차이까지 보이기
● 임시 저장된 stash 반영
- 최근 또는 지정된 임시저장소 내용을 가져와 반영하고 삭제
$ git stash pop
$ git stash pop stash@{n}

pop은 반영후에 삭제됨.
apply는 삭제되지 않음.

- 최근 또는 지정된 임시저장소 내용을 가져와 반영, 작업 디렉토리만 반영,
stash 목록은 그대로
$ git stash apply
$ git stash apply stash@{n}

- 최근 또는 지정된 임시저장소 내용을 가져와 반영, 작업 디렉토리와 스테이징 영역도 반영,
  stash 목록은 그대로
$ git stash apply --index
$ git stash apply --index stash@{n}

● 특정 stash 삭제와 모든 stash 삭제
- 최근 임시저장 내용을 삭제
$ git stash drop
- 지정된 임시저장 내용을 삭제
$ git stash drop stash@{n}
- 모든 stash 목록을 모두 제거
$ git stash clear

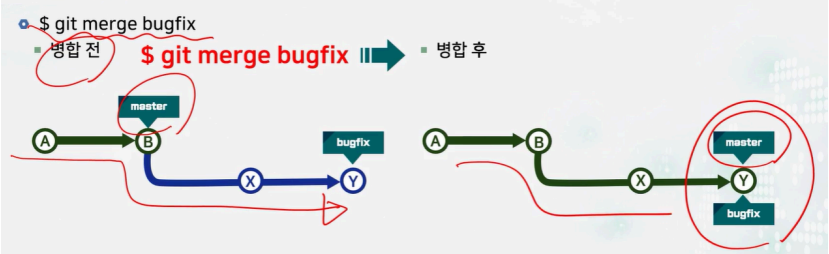
● Untracked 파일 삭제
- $ git clean
:바로 삭제되지 않음
- $ git clean -i
: Commands 가 나타나면 숫자 또는 맨 앞글자를 입력해 삭제 가능.
- $ git clean -f
: 무조건 삭제

● 임시 저장 복원 시 충돌 발생
$ git stash pop
~
Unmerged paths:
        (use “git restore --staged <file>...” to unstage)
        (use …

◈다양한 브랜치 병합
[10주차]
● 병합(merge)
- 두개의 브랜치를 하나로 모으는 과정
    - fast-foward(빨리 감기) 병합 : 커밋의 수는 동일(단순 브랜치 이동)
    - 3-awy병합 : 새로운 커밋 발생

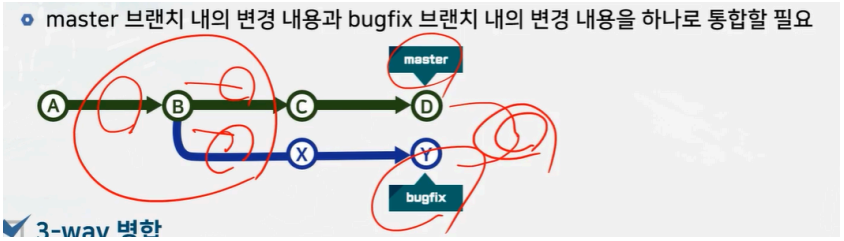
● fast-foward 병합 조건
: 현재 브랜치 master가 병합할 대상 커밋의 직접적인 뿌리가 되는 경우
- 일렬 상태

- 브랜치 마스터에서 병합 명령
$ git merge bugfix



● 3-way 상태: 두 분기가 갈라진 상태
: 두 브랜치의 조상이 같은 경우

- 새로운 커밋 E 생성
$ git merge bugfix [-m ‘message’]
* -m 입력이 없을경우 편집기실행, ‘Merge branch bugfix’이 기본 메시지 내용



● 일렬 상태에서 기본이 fast forward 병합
- 병합 기본은 fast forward 병합

- 마스터 브랜치에서 브랜치 dev1을 병합
$ git merge dev1

- 마스터에서 dev1을 non fast forward로 병합(3-way 병합)
:옵션 --no-ff (Non fast-forward)
$git merge dev1 --no-ff | $ git merge --no-ff dev1

● non fast-forward 병합
: 병합시에 'fast-forward 병합'이 가능한 경우라도 3-way 병합을 수행
- 병합된 브랜치가 그대로 남기 때문에 브랜치 관리명에서 유용(이력이 남음)

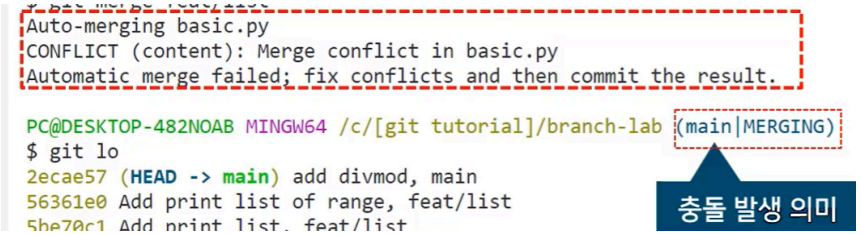
● 병합의 다양한 옵션 종류
- $ git merge {병합할 브랜치 명}: 보통 병합
- $ git merge --no-ff {병합할 브랜치 명} : 무조건 3-way 병합
- $ git merge –ff-only {병합할 브랜치 명} : 상태가 fast-forward인 일렬 상태에서만 병합 진행
- $ git merge –squash {병합할 브랜치 명}
: 현재 브랜치에 병합 대상과의 합치는 커밋을 하나 생성해 병합

*옵션 --squash
: 강압적인 병합

$ git merge --squash hotfix
- 커밋 이력과 병합되는 브랜치 이력도 남기지 않음
새로운 커밋에 상대 브랜치의 내용을 합해 새로운 커밋으로 병합

◈ 병합 충돌과 해결

● 병합에서 충돌 발생 표기



- 병합 충돌시 브랜치가 (main)->(main|MERGING) 으로 바뀜

● 병합 충돌 이해
- 3-way 상태에서 두 브랜치의 동일 조상을 기준으로
병합할 두 브랜치의 마지막 커밋을 비교했을때

- 파일 충돌 없음
: 수정되지 않거나 한쪽 브랜치에서만 수정되면
- 파일 충돌 발생
: 두 브랜치 모두에서 변경 사항이 달리 발생한 파일

- 충돌 해결
: 충돌이 발생한 파일 을 직접 편집(add, commit 진행)
- 충돌한 파일 내부 표시
: 3개의 표시로 구분 (Vscode)
<<<<<<HEAD
현재 브랜치 HEAD의 수정 내용(Current/Both)
=======
병합되는 브랜치 feat/list의 수정 내용(Incoming/Both)
>>>>>> feat/list

*편집후에는 3개의 표시를 소거해야한다.

● 병합 취소
$ git merge --abort

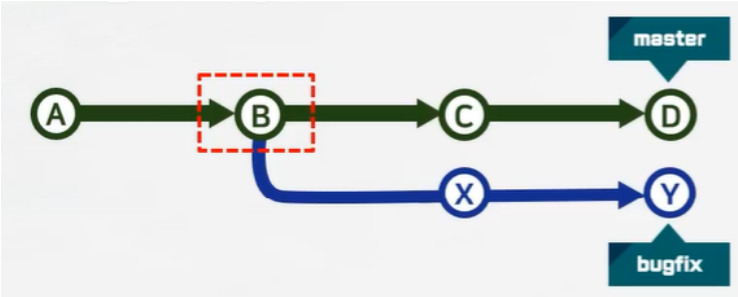
● 다시 병합
$ git merge feat/list

● 추가, 커밋 다시
$ git commit -am ‘message’

■SUMMARY
: 파일을 만들고 add, commit을 실행후에 다른 브랜치에서도 커밋을 하고나서 브랜치 합병에 대해 배웠고, 합병충돌시에 vscode를 통해 나타나는 각 브랜치 별 특징과 수정방법에 대해 배웠다.

◈ 브랜치 리베이스 rebase

● base의 이해



- 위의 그림에서 mater브랜치와 bugfix 브랜치의 공통조상인 커밋 B를 base라고 칭함.

● rebase의 이해
- base를 B에서 D로 수정후 bugfix를 D이후에 배치한다.
- 이후 다시 'fast -forward 병합'을 수행하여 master 브랜치의 HEAD가 bugfix 브랜치의 커밋으로 이동해 로그이력이 분기없이 하나의 줄기로 이어진다.

* 브랜치 충돌이 발생할 수 있음

$ git rebase master
: master의 위치는 그대로 유지되고, bugfix의 이력이 통째로 이동

$ git merge bugfix
: fast-forward 병합이 실행됨.

● rebase에서의 충돌
- 이동되는 X와 Y의 내용이 master의 C, D 커밋들과 충돌하는 부분이 생길 수 있다.

- 해결 절차
  1. 파일 수정
  2. 파일 추가
  $ git add <수정파일>
  3. rebase 계속 수행, 마지막 메시지 메시지 수정
  $ git rebase --continue

■ merge VS rebase
● merge
: 여러 분기가 생긴 변경 내용의 이력이 모두 그대로 남아있기 때문에 이력이 복잡해짐
● rebase

: 히스토리가 선형으로 단순해지고 좀 더 깨끗한 이력을 남김
: 원래의 커밋 이력이 변경됨 (정확한 이력을 남겨야 할 필요가 있을 경우 사용하면 안됨)

■ fast-forward merge VS rebase

● fast-forward merge

- 조상에 위치한 브랜치에서 선형 브랜치의 마지막으로 이동

● rebase

- 두 갈래의 브랜치에서 기존의 베이스를 수정.

● $ git rebase <newparent> <branch>

- 일반적 rebase

$ git checkout topic : 브랜치 topic이동
$ git rebase main : main을 rebase
$ git checkout main : 다시 main으로 이동
$ git merge topic : fast-forward 병합 수행

- 다른 rebase 방법

$ git rebase main topic : 어느 브랜치든 main topic 순서로 재배치 방법(rebase)
$ git checkout main : main으로 이동 (fast-forward)
$ git merge topic : 브랜치 병합

◈ 최신 커밋 수정

c 기본 편집기 설정
$ git config --global core.editor ‘code --wait’

● 설정된 편집기로 최근 커밋 메시지 수정
$ git commit --amend

● 최근 커밋 메시지를 직접 입력해 수정
$ git commit --amend -m “an updated commit message”

● HEAD의 내용 수정
- 새로운 파일을 추가하거나 파일을 수정한 후 새 커밋을 생성하지않고 최신 커밋에 반영

● 커밋 옵션 --amend 사용
$ git commit --amend :기본 편집기로 메시지 편집
$ git commit --amend -m ‘new message’ : 명령어 상에 직접 메시지 입력
* 메시지를 수정할 시 커밋 아이디도 변경됨.
$ git commit --amend --no-edit : 메시지 수정 없이 다시 커밋 수정

● rebase의 --interactive를 사용

- 작업공간이 깨끗한 이후, 이전 여러 개의 커밋을 수정
- 커밋 시퀀스를 새로운 기본 커밋에 결합
$ git rebase [-i | -interactive] HEAD~3
HEAD~3 : 수정할 커밋의 직전 커밋
(HEAD~2 부터 수정 가능)
-> HEAD~2, HEAD~1, HEAD 3개의 행 편집 가능(로그 결과와 다르게 오래된 커밋부터 표시)
● rebase -i 대화형 명령어
p(ick) : 해당 커밋을 수정하지 않고 그대로 사용
r(eword): 개별 커밋 메시지를 다시 작성
s(quash) : 계속된 이후 커밋을 이전 커밋에 결합
d(rop) : 커밋 자체를 삭제

● 명령 squash 방법
- 상위에 위치한 커밋에 pick이 있으면서 아래로 연속적으로 squash 명령
: squash는 제거됨.

● 명령 drop
[drop | d]마지막 커밋을 제거함

◈Vscode에서의 깃 활용

+ = add
Commit = commit
discard = git restore
unstage = git restore --staged

◈버전 되돌리기 reset

●reset 이해
커밋 이력에서 이전 특정 커밋으로 완전히 되돌아간다.
- 이동 후 해당 커밋 이후의 이력은 모두 사라진다.(옵션으로 처리가능)

●–hard: 작스깃 모두 복사
: 다시 마지막 이전 head로 돌아가려면 commit만 필요
●–mixed: 스깃 복사(기본)
: 다시 마지막 이전 head로 돌아가려면 add, commit이 필요
●–soft: 깃 복사
: 다시 마지막 이전 head로 돌아가려면 파일수정, add, commit 필요

->>>> $ git reset –hard ORIG_HEAD를 이용하면 모든 과정이 생략가능하다.

◈커밋 취소 revert

-지정한 특정 커밋을 취소해 바로 이전 상태로 되돌리는 방법
●옵션 –no-edit
:커밋 메시지가 자동으로 ‘Revert “이전 커밋 메시지”으로 지