

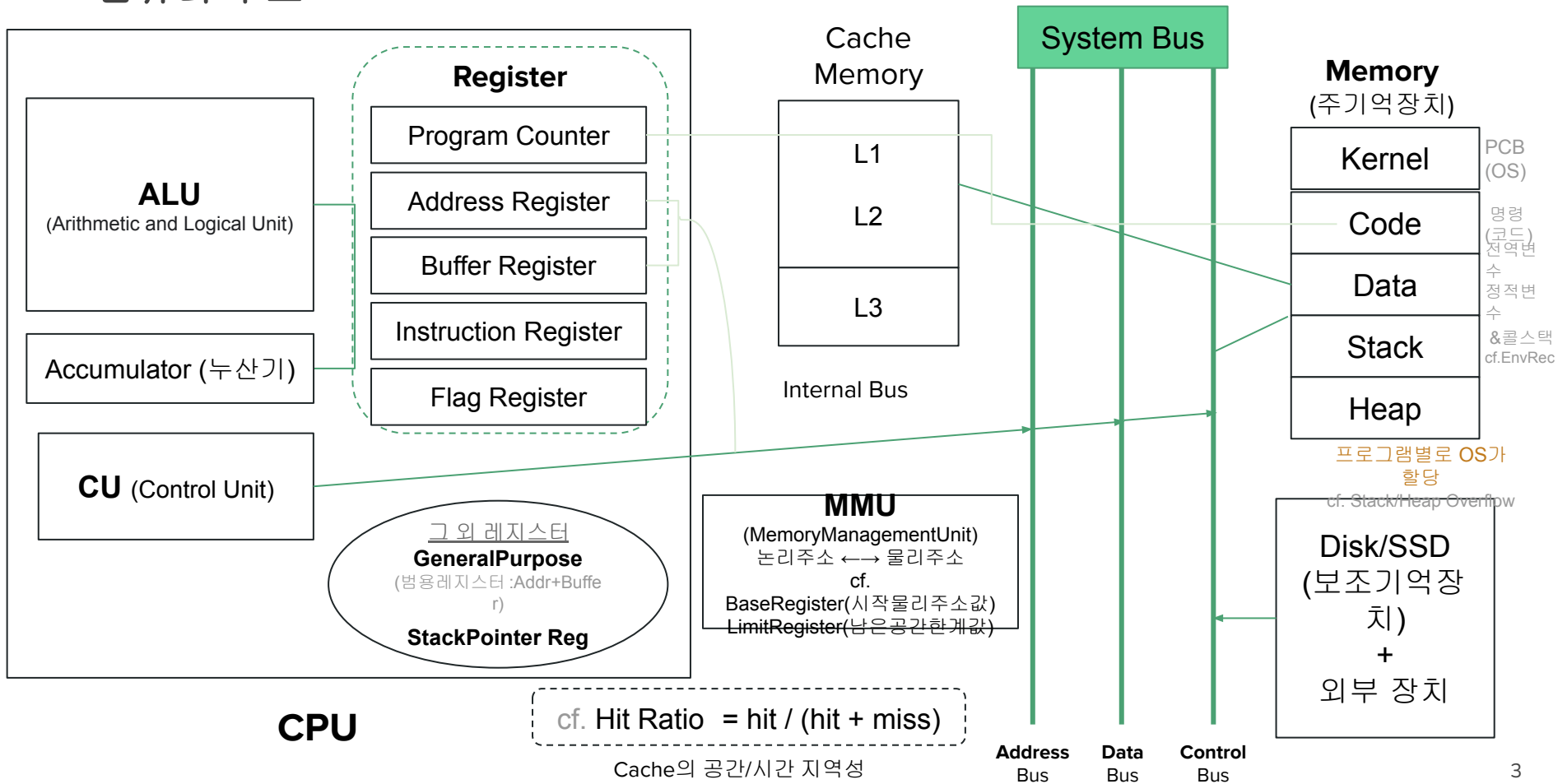
# Computer Architecture

컴퓨터 개발 과정

Git with Github

# 01. Computer Architecture / Compiler

# 컴퓨터 구조



# 명령어 구조

16bit Mem Slot(8bit)  
⇒ 32bit (2개 명령)  
⇒ 64bit (4개 명령)

## Instruction

Operation Code  
(4bit)

Operand  
(6\*2 bit)

1 + 2

MOVE  
STORE(FETCH)  
LOAD  
PUSH  
POP  
JUMP  
CALL  
HALT

READ/WRITE/STARTIO/TESTIO

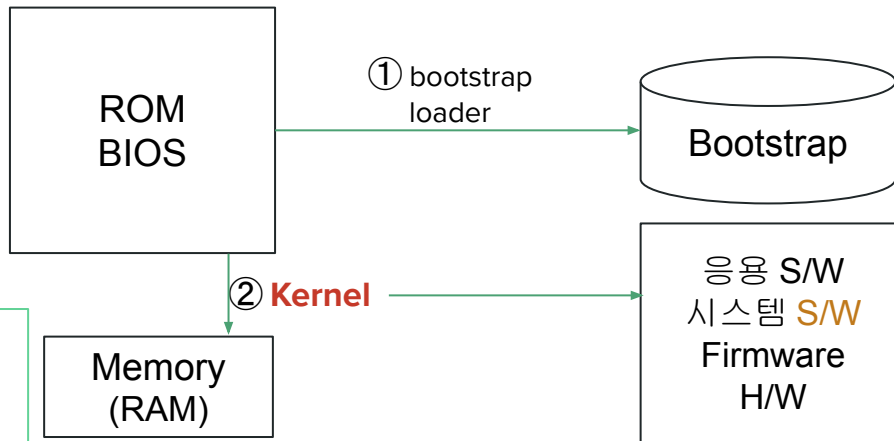
ADD/SUBTRACT/MULTIPLY/DIVIDE  
INCREMENT/DECREMENT  
AND/OR/NOT/XOR  
COMPARE

A = 1  
B = 2  
C = A + B

STORE 1 &A  
STORE 2 &B  
LOAD &A  
ADD &B  
STORE &C

**CISC** (Complex Instruction Set Computer/CPU)  
**RISC** (Reduced Instruction Set Computer/CPU)

# OS (Operating System)



## <OS>

**Unix** (1970)

- Bell KennethThompson/DennisRitch

**Linux** (1991) LinusTorvalds, [배포판\(계열\)](#)

**Windows** (1985) MS PaulAllen/BillGates

Virtual Memory/Swap  
Fragmentation/Paging  
cf. GarbageCollector

Partitioning (논리)  
Formating(FS결정)  
Index Block  
(FAT:Cluster,Unix:i-node)  
Mirroring(Raid1/Raid5)  
NFS, HDD/SSD  
SATA/SCSI/SAS

## <Memory>

**DRAM**(Dynamic RAM): **RAM** 고집적/저속

**SRAM**(Static RAM): **Cache** 저집적/고속

**SDRAM**(SyncDynamic RAM) clock신호와 동기

**DDR SDRAM** 최근 대중적은 DDR4

SDRAM(16배속)

(Double Data Rate SDRAM) 2배 대역폭/속도

## Process vs Thread

**H/W Multi-Thread & S/W Multi-Thread** cf. Hyper-Thread  
코드/데이터/힙은 공유, 레지스터/스택/프로그램카운터는 각 스레드별

cf. **IPC**(Inter-Process Communication) File | SharedMem.  
⇒ MutexLock, Semaphore(영속속 각 자원별), Monitor(큐)  
방식

⇒ DeadLock발생조건: 상호배제, 점유와대기, 비선점, 원형대기

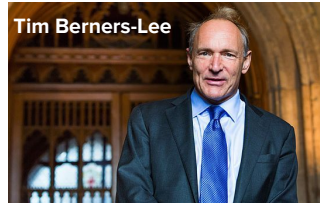
**PCB**(Process Control Block) for **Context Switching**

# 연대별 언어 **chronological programming language**

- 1943: Plankalkül (**MachineLang**)  
ENIAC CS(10진수) → UNIVAC (1949)
- 1951: **Assembly** (**2GL**)
- 1954: **Fortran**
- 1958: Algol(A) 58
- 1958: **Lisp**
- 1959: **COBOL** ← FACT (1959)
- 1962: APL, Simular, SNOBOL
- 1964: **BASIC**, PL/1(IBM)
- 1969: **Pascal** (교육용), **B** KennethThompson
- 1970: Forth
- 1970~1984: **CommonLisp**
- 1972: **C** ← **B** ← CPL(1963,Cambridge)
- 1975: **Scheme** ← Lisp
- 1978: **SQL**
- 1978~1983: Ada (미국방부, F22)
- 1980: **Self** ← CommonLisp
- 1983: **C++** ← C with Classes(1980)
- 1984: MATLAB, Eiffel(1985)

- 1986: **Object-C**, **ErLang** → Elixir
- 1987: **Perl**, Mathematica(1988)
- 1989~1998: **XML** ⇒ + HTML = **XHTML** → JSX
- 1989~1991: **HTML** ← XML ← Enquire(1980)
- 1990: **Haskell**
- 1991: **Python** ← Perl, VisualBASIC
- 1993: **Ruby**, Lua, **R** → Factor(2003)
- 1991~1995: **Java**(SUN) ← Oak ← C++
- 1994: **CLOS** ← Part of CommonLisp(1970)
- 1995: **JS**, **PHP**, **Delphi** ← Pascal IDE/Lang
- 1996: OCaml → **Rust**(2015), **CSS1**
- 1999: D ← C++, XSLT → **CSS3**(2005)
- 2000: **ActionScript** ← ECMAScript
- 2001: **C#**(Delphi+Java), VS.Net → **F#** (2002)
- 2009: **Clojure** ← Lisp
- 2003: **Groovy**, **Go**lang, **Scala**, Factor
- 2011: **Xamarin**, **Dart**(Google), **Elixir**
- 2011~2016: **Kotlin**(JetBrains), **TS**(2012)
- 2014: **Swift**, **Rust**(2015)

# JS의 탄생 배경 (feat. Browser)



- 1989, **URL, HTTP, HTML** → HyperText Browser (by Sir Tim Berners-Lee)
- 1993, **NCSA Mosaic** Graphic Browser (모자이크)
  - 일리노이 대학교(UIUC), NCSA(National Center for Supercomputing Applications)
- 1994, Mosaic Communications's **Mosaic Netscape 0.9**
  - Jim Clark이 NCSA의 Marc Andreessen 영입하여 전면 재개발. (**Mozilla** = Mosaic + Gecko)
- 1995, **Mosaic Netscape** → **Netscape Navigator**
  - 브라우저 점유율 75%, NCSA와 상표권 문제로 이름 변경.  
(Mosaic Communications → Netscape Communications)
- "브라우저는 준비되었는데.. 사이트가 별로 없네?"  
"Java Applet은 너무 무겁고 **Sandbox** 문제도 있고..."



```
37.31" 0.000 - - - "Mozilla/5.0 (X11; Linux  
237.31" 0.000 - - - "Mozilla/5.0 (X11; Lin  
h" "Gecko/20021109)
```



# JS의 탄생과 발전

- '모든 사용자가 편하게 웹 사이트를 만들게 하자!'
  - ⇒ Scheme Runtime을 **Netscape**에 Embed하자!
  - ⇒ Self, Scheme의 전문가 **Brendan Eich**을 영입!
  - ⇒ LiveScript (코드명 **Mocha**) 완성! (1995년 9월)
- (Applet이 표준, 2011년까지 Java가 JS 보다 빨랐다!)
  - ⇒ 'Live를 Java로 바꿔야 사람들이 인정하고 사용하겠군'
  - ⇒ **LiveScript** ⇒ **JavaScript**
- 이름을 바꿔도 사람들은 인정하지 않아! (cf. MS's JScript, 1996 IE3)
  - ⇒ **ECMA**에 찾아가 보자!

cf. jQuery(2006), CSS3(2005), JSON(2001)

TypeScript(2012), React(2011), ReactTS(2020), Dart/Kotlin/...





# Compiler

Interpreter Language

Compiler Language

## 1. Lexical Analyzer

Tokenizer(Scanner) & Lexer  $\Rightarrow$  Symbol

## 2. Syntax Analyzer (Parser)

$\Rightarrow$  AST(Abstract Syntax Tree) cf. Annotated AST

$\Leftrightarrow$  Token(Parsing) Tree cf. TS's Symbol Table

## 3. Intermediate Code Generator (Semantic Analyzer)

Type Checking  $\Rightarrow$  Byte Code (Object file)

## 4. Code Optimizer (Local & Global Optimizer)

Scheduling & Register(Memory Address) Assignment

## 5. Target Code Generator (exe file, Machine Code)

```
var a = 1;  
var b = 2;  
var c = a + b;  
if (c >= 3)  
    console.log('Big!');  
else  
    console.log('Small!')
```

Token  $\Rightarrow$  Tree

```
import * as ts from "typescript";

const program = ts.createProgram(files, opts);
const checker = program.getTypeChecker();
program.emit();
```

- 1) **Read TSConfig (tsconfig.json)** setup the program & gets starting files
- 2) **Pre-process Files** follow imports (all possible files)
- 3) **Tokenize(Scan) and Parse** convert text to syntax-tree
- 4) **Binder** convert identifiers in syntax-tree to symbols
- 5) **Type Check** checks the types use Binder and syntax-tree
- 6) **Transform** changes syntax-tree to match tsconfig target options
- 7) **Emit** prints syntax-tree to \*.js, \*.d.ts and other files

code to data  
Annotated AST

type  
checking

creating  
files

## 프로그래밍 언어의 구성 요소

변수(variable) & 상수(constant)    원시형(Primitive) vs 객체/참조형(Reference)

식별자(identifier)와 예약어(reserved word)

객체(object) → Class → 인스턴스(instance)    cf. this, instanceof

바인딩(binding)과 할당(assign)    연산자(operator)

선언(declaration)과 정의(definition)

문(statement), 표현식(expression) > 리터럴(literal)

조건문(condition)과 반복문(loop)    cf. Sync vs Async

함수(function)와 메소드(method) 그리고 클래스(class)와 객체({})

실행 컨텍스트 (Execution Context), Lexical Scope    cf. JVM

## And Other Things

### 1. DBMS(DataBase Management System)

RDBMS(MySQL, Postgresql, Oracle, MS-SQL, etc)

DocumentDB(MongoDB, DynamoDB, MySQL XDev/API, etc)

Session, Index, Partitioning, Clustering vs Replication

### 2. Network

Router(Routing Table), DNS(Domain Name System), Switch, Hub

TCP(Transmission Control Protocol), UDP(User Datagram Protocol)

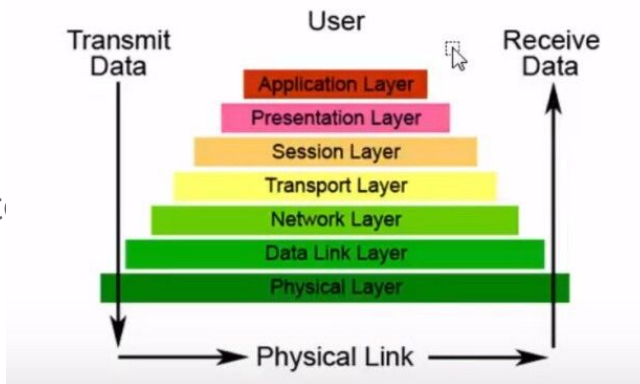
Firewall, Load-Balancer, WASWebApplicationServer vs WSWebServer

### 3. Cloud (OnDemand/OnPremise)

InfraaaS, PlatformaaS, **SoftwareaaS**GoogleApps/SalesForce

### 4. Library/Framework, Clean Architecture, Messaging Queue, etc

## The Seven Layers of OSI



## 02. Git 시작하기

# DVCS (Distributed Version Control System)

---

**Git: 2005** by **Linus** Benedict **Torvalds**

Mercurial: Python

BitKeeper: 1998, 유료

코드 공유 / 이력관리 / 협업 ⇒ 책임과 기록

cf. SCCS(1970), RCS(1980), **CVS**(1986), **SVN**(2000)

test repo: <https://github.com/indiflex/hana3>

# Settings

---

Download Git Client

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

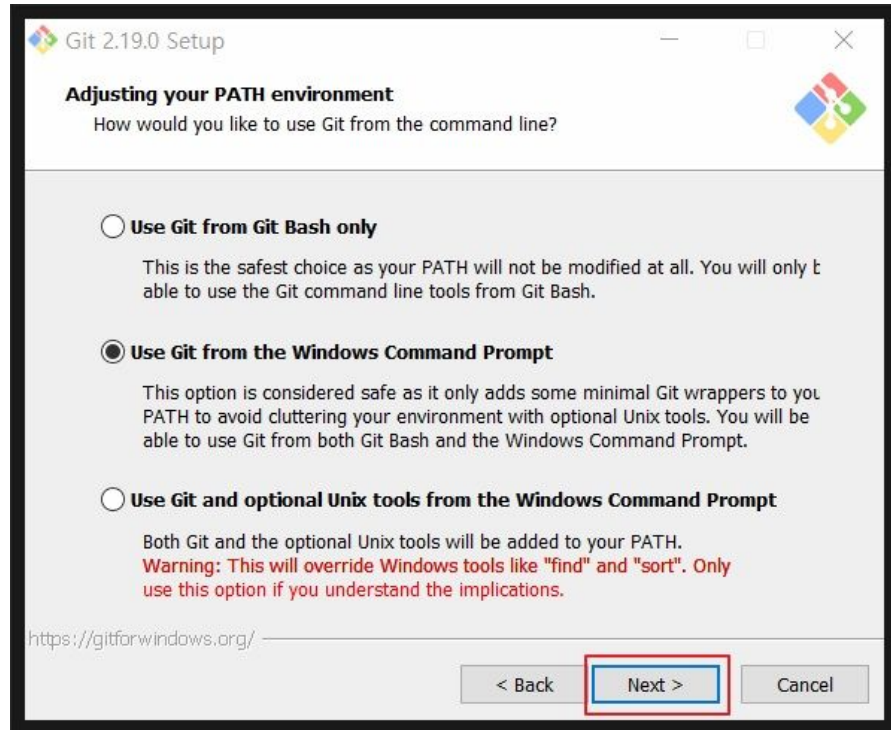
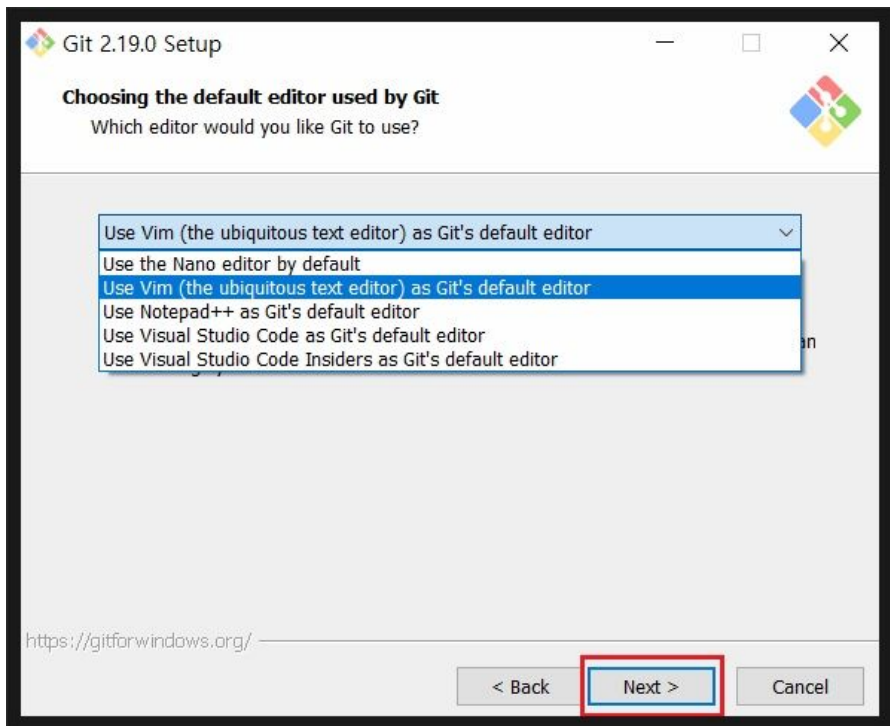
SCM Service Site

- [GitHub](#) : 무료는 public만 가능, 안정적(2022~ 모두 무료)
- [BitBucket](#) : public/private 모두 무료, 안정적인 편
- 기타: [GitLab](#), [Codebase](#), [CloudForge](#), etc

(참고) windows에서 GitHub에 한글 깨질때

- cmd"명령프롬프트"> **chcp 65001**

# Settings



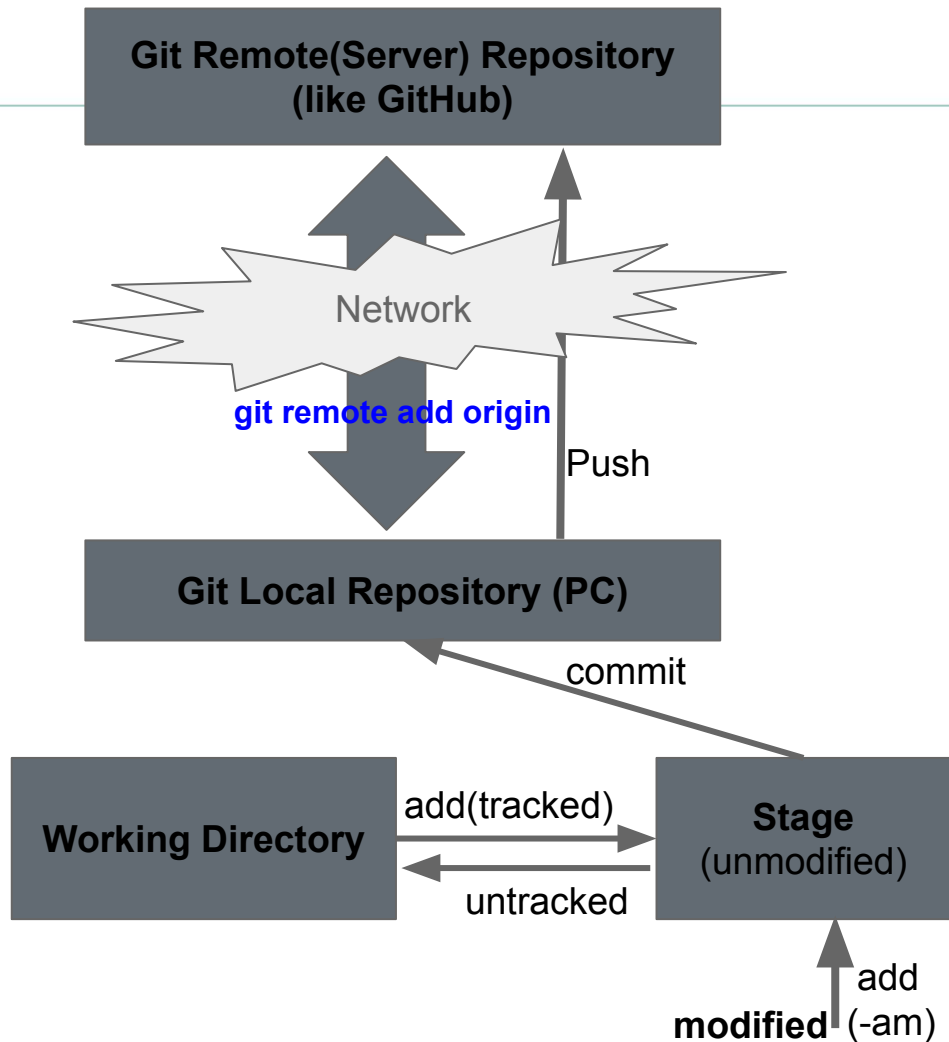
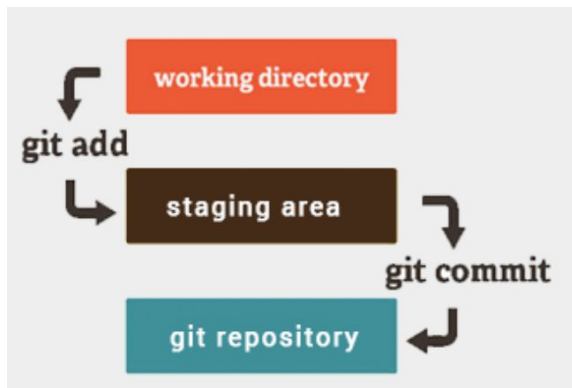


# Git Config

---

1. # 바탕화면 **git bash** 실행
2. **git config --list**
3. `git config user.name`  
`git config user.name "xxx"`  
**git config --global user.name <github-name>**
4. `git config user.email`  
**git config --global user.email <email>**
5. # git config 한눈에 보기 (git 활성화된 디렉터리)  
**cat .git/config**

# git structure



## **workingDirectory - stage - repository**

---

**working directory** (working tree)

**stage:** 임시 저장 공간 (git status 또는 git ls-files --stage)

**repository** (local / remote)

cf. **untracked** vs **tracked**

**staged** vs **unstaged**

**modified** vs **unmodified**

(빈폴더 / .gitignore)

git rm --cache 파일

now working

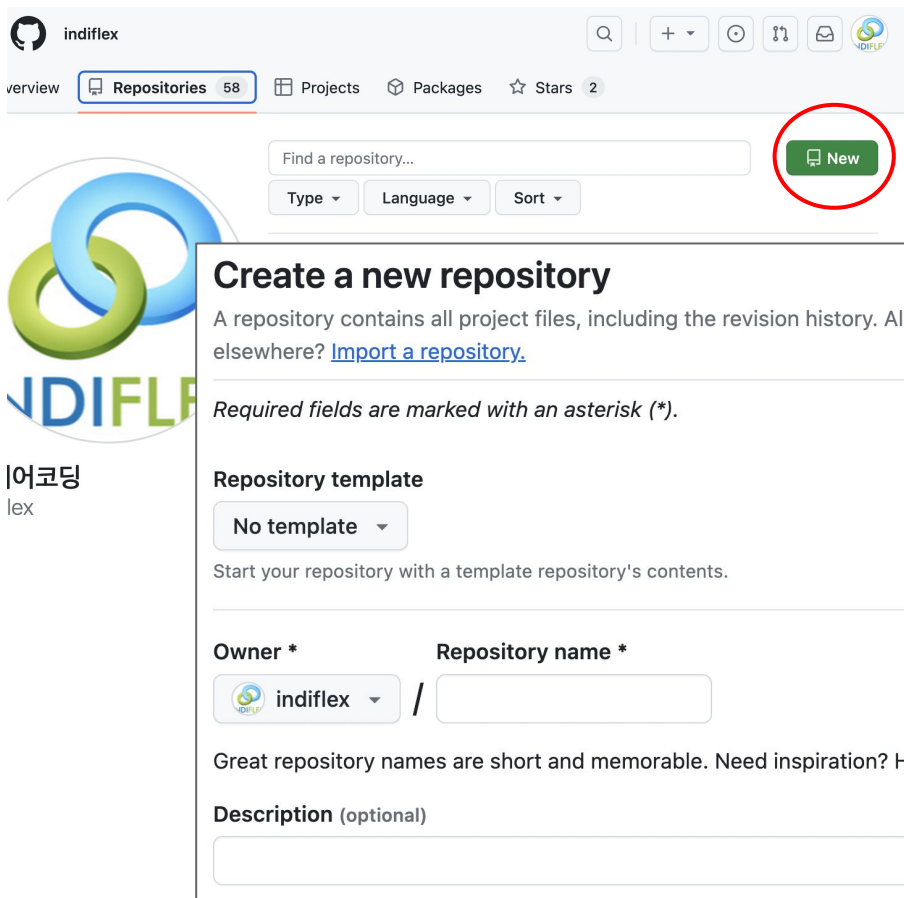
\* 파일수정 → **modified(WD)** → **add** → **modified(Stage)** → **commit** → **unmodified(Repo)**

## 로컬 저장소(Local Repository) 만들기

---

1. `cd <work-dir>`
2. `git init`                    # .git 디렉터리 생성
3. `vi .gitignore`            # git 제외 대상 파일 (`touch .gitignore && code .`)
4. `echo "ttt" > test.txt`
5. `git ls-files`            # test.txt 없음(:untracked)
6. `git add --all`            # working dir ⇒ stage    `git add -A` 또는 `git add .`  
cf. `git add <file명>`    또는    `git add .`
7. `git ls-files`            # test.txt 보임(**tracked file list**)  
`git status`
8. `git commit -m "first commit message"`
9. `git log`            # commit logs
10. `git status`    # staging 상태 확인

# 리모트 저장소 (Remote Repository)



indiflex

view Repositories 58 Projects Packages Stars 2

Find a repository...

Type Language Sort

**Create a new repository**

A repository contains all project files, including the revision history. Aliases are supported. [Import a repository.](#)

Required fields are marked with an asterisk (\*).

**Repository template**

No template

Start your repository with a template repository's contents.

**Owner \*** Repository name \*

indiflex /

Great repository names are short and memorable. Need inspiration? [View repository names](#)

**Description (optional)**

1. `cd <work-dir>`  
github remote repository 생성 선행
2. `git remote add origin <git-remote-url>`  
`cat .git/config`  
`git remote show [origin]`
3. `git branch -M master` # main → master
4. `git push -u origin master`  
`git push` # -u 옵션으로 파이프라인 이후!
5. 오류시 처리  
`git pull origin master` # pull 먼저  
`git pull origin master --allow-unrelated-histories`
6. `git log`
7. `git status` # staging 상태 확인  
`git ls-files -s` # `git ls-files --help`
8. `mv test.txt ttt.txt`  
`git ls-fiels -m` # -c, -i, -o  
`git ls-fiels -d`

## remote 명령어

---

Local | HTTP(s) | SSH | Git

**git remote -v**

```
$> git remote add <별칭:origin> https://github...
```

```
$> git remote rename <before> <after>           // 별칭 변경
```

```
$> git remote rm origin                          // origin 저장소 연결
```

해제

```
$> git (push|pull) [-u] origin <branch>           // -u: set-upstream
```

```
$> git remote show origin                         // origin 정보 보기
```

## git clone / push / pull

---

# Remote repo와 클론하기

**git clone** <github-remote-url> [dir-name]

#git pull : 서버의 최신 소스를 다운받기

#git push : 서버에 local repo 올리기

### 1. **git pull**

2. touch nnn.txt # modified or new  
git **diff** --name-only # diff for HEAD  
git **diff** HEAD~1 # 이전 커밋과의 diff

### 3. **git add --all** # git add -A

4. **git commit -m "message"**  
git commit -am "msg" # modified → stage → repository

5. **git push -u origin master**  
cf. git push

(참고: 서버와 연결된 repo 끊기)

**git remote rm origin**

## restore (make unstage)

---

원본 — 수정 → modified(WD)  
— **git add** → staged

staged — **restore --staged** →  
modified(WD)  
— **restore** → 원본!

1. `echo "bb" >> ttt.txt` # modified(WD)
2. `git status`
3. `git add -A` # modified(staged)
4. `git status`
5. **`git restore --staged ttt.txt`**
6. `git status` # modified(WD)  
`git ls-files`
7. **`git restore ttt.txt`** # unmodified(WD)  
(sameAs **`git checkout ttt.txt`**)  
"수정 전 상태(HEAD)로 돌아옴"

cf. `git checkout <CommitID>`



## rm (make untracked)

tracked —rm --cache→  
untracked(WD)  
& deleted(Stage)

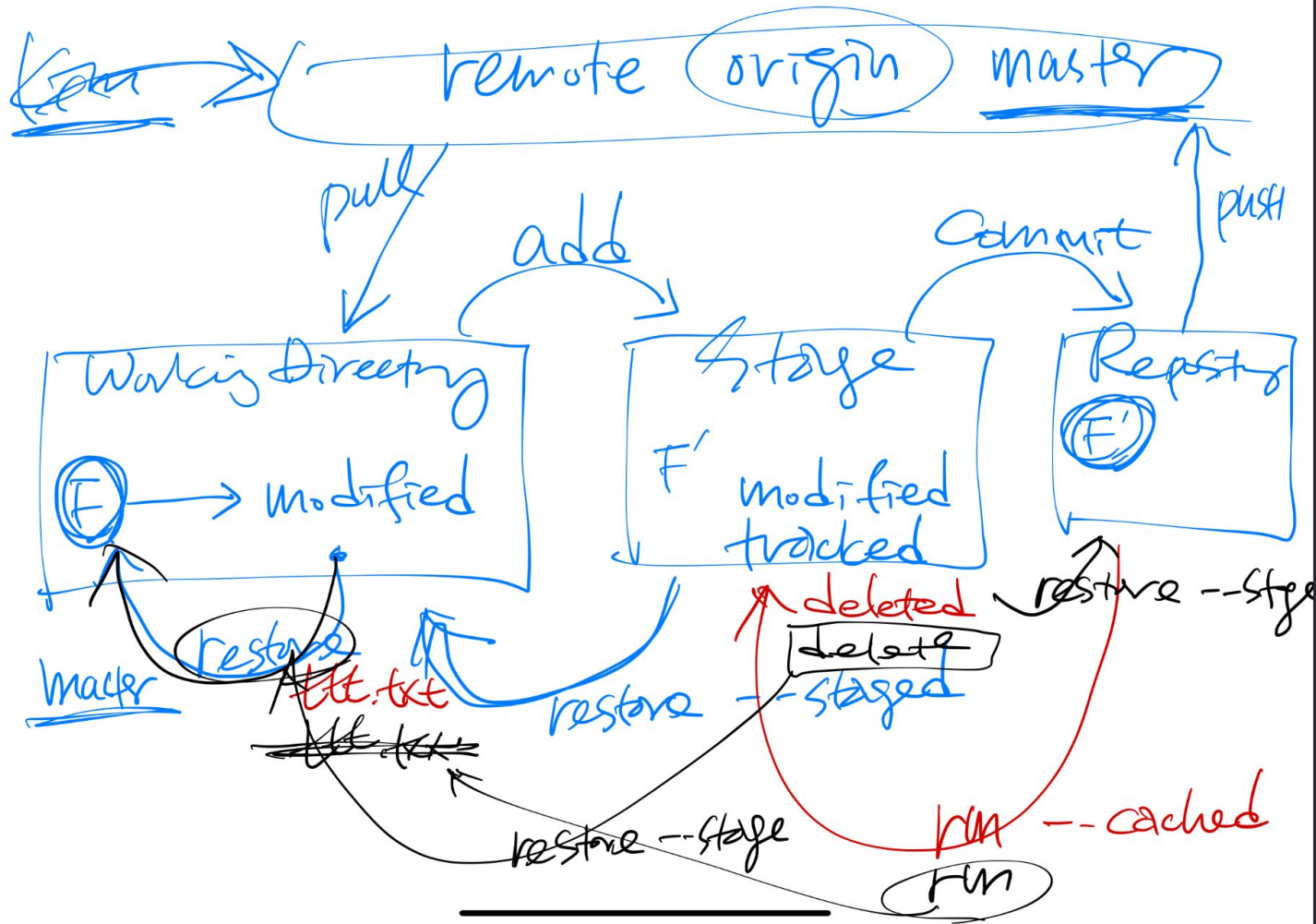
—restore --staged→  
unmodified(Stage)  
원위치(취소됨)!

1. `echo "ttt" > ttt.txt` # new file(untracked)
2. `git add .` # tracked & staged
3. `git commit -m 'ttt'` # committed
4. `git show` # last commit log
5. `git log` # commit log
- `git log --raw` # with filename
- `git log --graph` # git log --oneline
- `git reflog --raw` # with HEAD pointer
6. `git rm --cache ttt.txt`  
# 파일 원본은 나두고, **cache**에서만 삭제해줘!
7. `git status`
8. `git ls-files`

주의!) **--cache option**을 붙이지 않으면 완전 삭제됨!!  
`git rm aaa.txt` # deleted(완전삭제!)

복구 방법)

**restore --stage** ⇒ deleted(WD) ⇒ **restore**



# 연습문제

## Try This -restore

실수로 **tracked** 파일을  
수정했다.  
**restore**를 사용하여,  
다음 각 상황별로 원위치해보자.

1. 아직 **add**를 하지 않았다.
2. **add** 까지 했다(staged)

1. `echo "777" >> ttt.txt`

⇒ **modified**

⇒ 수정 취소하기

`git restore ttt.txt`

2. `echo "888" >> ttt.txt`

`git add ttt.txt`

⇒ 수정 취소하기

`git restore --staged ttt.txt`

`git restore ttt.txt`



# 연습문제

## Try This -rm

commit된 파일을  
untracked하려다  
실수로 `--cache` 옵션을  
붙이지 않아 원본이 삭제되었다.

이를 복구해보자.

`git rm --cache ttt.txt` 해야 할 것을 실수로  
`git rm ttt.txt` 하였다. 이를 복구해보자.

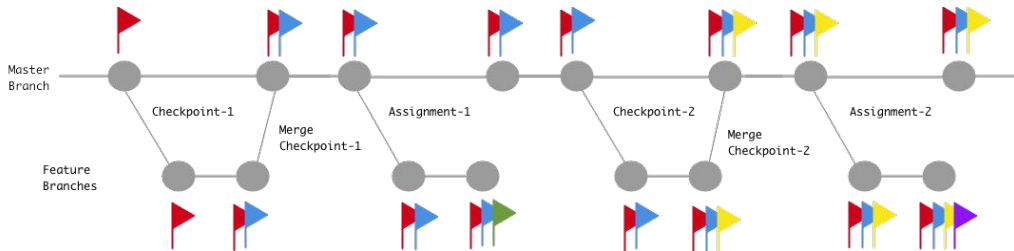
1. `git rm ttt.txt`
2. `git status`
3. `git restore --staged ttt.txt`
4. `git restore ttt.txt`



## 03. 깃 브랜치(Branch)

# Branch

- \* 기본(head) branch는 master(or main)  
(`git config --list` ⇒ `init.defaultbranch`)
- \* 가상 폴더 생성
  - 특정 commit pointer로 사본(가상폴더) 생성
  - master branch는 HEAD pointer 유지
- \* 독립적인 작업 공간(merge하지 않는 한 별개의 소스)
  - checkout(switch)하면 WD도 해당 브랜치로 변경!
- \* 빠른 동작(41B Blob - Binary Large Object)

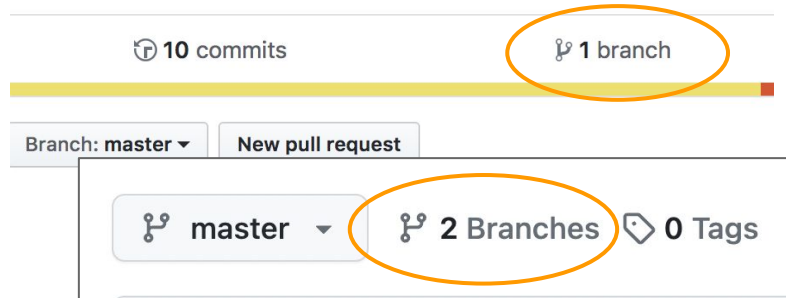


## branch 생성

**git branch <Branch> [CommitId]**

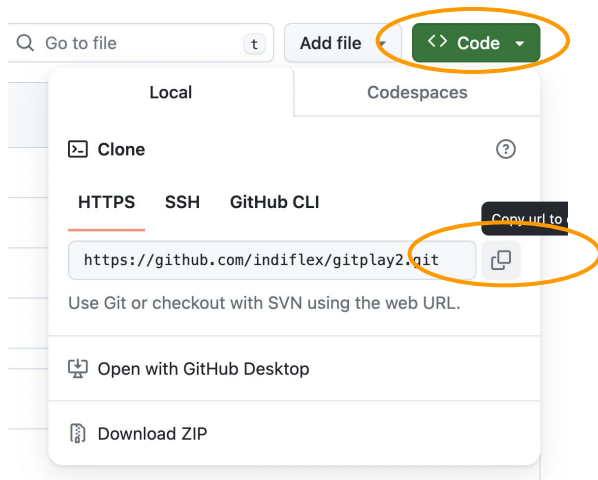
test code

[Manage topics](#)



1. `git branch <branch>` # branch 만들기
2. `git branch` # branch 전체 보기
3. `git checkout <branch>` # 전환  
`git switch <branch>` # 전환  
`git checkout -` # 이전 브랜치
4. `git checkout -b <branch>` # 생성 + 이동  
`git switch -c <branch>` # 생성 + 이동
5. 소스 수정 & add & commit # branch 실제로 생성!
6. `git push origin <branch>`  
`git push -u origin <branch>`
7. `git branch -v` # 또는 `-vv`
8. `git branch -r` # remote
9. `git branch -a` # local + remote
10. `git log <branch>`  
`git log <branch> --graph [--all]`
11. `git rev-parse <branch>`

## branch with remote



1. `git remote -v` # 원격 저장소 링크 + tracking  
`git remote show origin` # 링크 + 정보  
`git branch -avr`
2. `git push -u origin <branch>`  
`git push origin <org-name>:<new-name>`
3. `git config pull.rebase false` # merge def.
4. 다른폴더> `git clone <remote-url>` # master only

OR

다른동료폴더> `git pull origin <branch>`  
`git branch <branch> origin/<branch>`  
`git checkout -b <branch> origin/<branch>`  
`git checkout --track origin/<branch>`

cf. `git clone -b <branch> <remote-url>`

ex) A 브랜치 내용을 로컬의 B 브랜치로 merge

B> `git merge A` # local A → B

B> `git fetch` # fetch before merge!!

B> `git merge origin/A` # remote A → B



## commit & log

---

**HEAD:** 변화 내용에 대한 snapshot

**AHEAD**

vs

**BHEAD** (behind)

(local before push)

(remote before pull)

```
$> git commit -m [또는 -am] "message"
```

```
$> git commit --allow-empty-message -m ""
```

```
$> git log [--pretty=[short|online] | -p | --stat | --graph --all]
```

```
$> git (show|diff) <CommitID 또는 head>
```

## branch 삭제

```
git branch -d <branch>  
git branch -D <branch>
```

```
git push origin --delete <br>  
git fetch -p
```

1. **git checkout -b xx** # xx 브랜치 생성 + 이동  
**git branch xx** # 생성
2. **git branch -d xx** # xx에서 xx를 지울 수 없음!
3. **git checkout -** # git checkout master
4. **git branch -d xx** # 삭제 됨
5. **git branch -D xx**  
# merge 또는 commit 할게 남았다면 -D로 강제 삭제!!

ex) push를 이용하여 remote branch 삭제하기

```
$> git push origin --delete <branch>
```

```
$> git push origin -d <branch>
```

```
$> git fetch -p # 다른 폴더에서는 fetch -p해야
```

적용!

## cf. github bug repo history 완전 삭제

# BGF 설치

- <https://rtyley.github.io/bfg-repo-cleaner/>

# 삭제 작업

1. 파일 삭제
2. add & commint & push
3. git clone --mirror <url>
4. bfg --delete-files
5. push

```
$> git clone --mirror git@github.com:indiflex/hello.git
$> cd hello.git
$> java -jar ../bfg.jar --delete-files bigquery.json .
$> git reflog expire --expire=now --all && git gc
--prune=now --aggressive
$> git push
```

# 소스 저장소로 이동후!!

```
$> git pull --allow-unrelated-histories
```

## cf. git flow에서 branch 관리

---

# git flow feature start X

# 작업

1. git checkout develop
2. git pull
3. git checkout -
4. git rebase develop
5. merge & add
6. git rebase --continue
7. 작업...
8. git rebase develop
9. merge & add & commit
10. git rebase --continue
11. git flow feature finish X

# 연습문제

## Try This -branch#1

다음 두 개의 브랜치를 만들고  
각각 `ttt.txt`를 수정한 다음  
`remote`에 `push` 하시오.

`br11`: `master`의 `HEAD`

`br22`: `master`의 `HEAD~1`

### 1. `br11` 생성

```
master> git checkout -b br11
```

```
br11> echo "br11" >> ttt.txt
```

```
br11> git push -u origin br11
```

### 2. `br22` 생성

```
master> git log --oneline
```

```
master> git branch br22 HEAD~1
```

```
master> git checkout br22
```

```
br22> ...
```



# 연습문제

## Try This -branch#2

이전 페이지에서 만든  
br11, br22 브랜치를  
다른 폴더에서 clone 하시오.

**git clone : master**

**git checkout -b br11 origin/br11**

**git checkout -b br22 origin/br22**



# 연습문제

## Try This -branch#3

br33 브랜치를 만들고,  
br11과 br22를 merge하시오.

1. 

```
br11> git pull
```

```
br11> git checkout -b br33
```

```
br33> git merge br11
```
2. 

```
br33> git merge origin/br22
```



## 04 - 깃 스택시(stash)와 클린(clean)



## stash (임시 저장)

작업중인데 갑자기 긴급  
수정사항이?!  
(아직 commit하면 안되는데...)

작업중....

**git stash save** (임시저장,  
작업내역제거)

긴급 작업 & commit

**git stash pop** (임시저장내역 다시적용)

\* Cannot switch branch before working-tree  
clean! (commit전에는 branch switch 안됨)

\* commit하기 전에 갑자기 긴급 수정사항이 있다면?!  
⇒ 현재 작업중인걸 임시 저장하고 패치하자!

\* **git stash [save "label"]** // STACK 구조  
Saved working directory and index state WIP on  
<branch>: <CommitID> <Last Commit Message>  
⇒ 작업중이던 내용은 **stash**로 임시저장되고 사라짐!  
⇒ 이제 switch 가능!

\* 여러개의 **stash**가 **stack**에 저장(save & pop)  
WIP(Work In Progress)는 **stash@{num}**형태

\* **cat .git/refs/stash**  
**git stash list**

## stash (Cont'd)

---

```
git stash list
      show
      -p <stash WIP>
      pop
      drop
```

```
$> git stash [save "label"]    # stash 저장
```

```
$> git stash [--include-untracked]
```

```
$> git stash list                # stash 목록
```

```
$> git stash show                # stash log
```

```
$> git stash show -p <WIP>      # 상세 log
```

이제 패치(`switch & edit & commit ... 모두`) 가능

```
$> git stash pop    # LIFO (auto-merge!)
```

```
$> git stash drop    # conflict시 삭제안된 내역까지
삭제 (Last In First Drop)    "stash 취소"
```

## stash branch

(안전한 복구를 위해 갈아타기!)

`git stash branch <branch>`

현재 작업중인 내역으로 새로운 브랜치 생성

```
$> git stash # stash 저장

$> git stash branch <new-branch-for-stash>

wbr> git stash branch st1 # stash로 st1 만들
st1> git commit -am "working..." # wbr의
작업내역 사라짐!

st1> git branch -vv
st1> git stash list # 저장된 stash drop됨

st1> git switch wbr # 작업할 br로 이동
wbr> edit & commit... # 작업

wbr> git switch st1
st1> # commit하지 못한 작업 이어서 하기
```

## stash apply

---

(특정 작업중 상태로 복원!)

**git stash apply [stashID]**

어느 브랜치건 **stash**로 임시 저장해 놓은  
작업중인 상태로 복원하기  
(커밋된 내역은 유지하고, 다만 작업중이던  
내역만 복원)

```
$> git stash [save "label"]    # stash 저장
```

⇒ 작업 전(마지막 commit)상태로 돌아감

(주의: git stash 할 때마다 stashID는 변경됨)

```
$> git stash list
```

```
$> git stash apply stash@{0}    # label 상태로 복원
```

⇒ stash는 사라지지 않고 그 상태로만 복원

```
$> git stash drop stash@{0}    # 특정 stash 삭제
```

## git clean

### (untracked file 삭제)

**git clean -n** (내역만 확인)

**git clean -d** (untracked files only)

**git clean -X** (with ignore too)

**git clean -X** (only ignore files)

```
$> git clean -h # clean 도움말(옵션 보기)
```

```
$> git clean -n # clean 시 삭제 될 파일 목록 출력  
⇒ .gitignore에 정의된 파일은 대상이 아님
```

```
$> git clean -di # 안전하게 interactive하게!  
⇒ ignore file와 untracked 모두 삭제  
(하위폴더포함)
```

```
$> git clean -df # 묻지도 따지지도 않고 삭제
```

```
$> git clean -xi # 안전하게 interactive하게!  
⇒ ignore file(x)포함 모든 untracked 삭제
```

```
$> git clean -xf # 묻지도 따지지도 않고 삭제
```

## 05. 깃 머지(merge)와 리베이스(rebase)

## merge (병합)

---

- **Fast-Forward merge**
- **3-Way merge**
- **Rebase**

### 1. **Fast-Forward merge**

시간의 흐름대로 커밋된 내역을 병합

충돌(conflict) 발생 없고 100% Auto-merge  
merge 후 모든 commit이 복제

### 2. **3-Way merge**

두 개 이상의 브랜치로 파생된 커밋을 병합

충돌(conflict) 가능성 있음

병합 메시지(merge commit) 존재

### 3. **Rebase**

공통조상(base) 병합

3-Way ⇒ Fast-Forward 화

# Fast-Forward merge

master> **git checkout -b br11** # 작업 branch

br11> **edit & commit** # 작업(커밋1)

br11> **edit & commit** # 작업(커밋2)

br11> **git switch master** # 기준 branch

master> **git merge br11** # merge

⇒ master로 커밋1, 커밋2 모두 복제

cf. **git merge --abort** # 병합 취소

master> **git log --graph** # 확인

master> **git branch --merged** # merge 확인

```
git5 (master) $ git branch --merged  
br11  
* master
```

master> **git branch -d br11** # 완료 제거

```
git5 (master) $ git log --graph  
*   commit a7cc21b5b7f1ad4e9986d5c4c9705ccda8e990dd (HEAD -> master)  
    Merge: b7533b9 3eb52b1  
    Author: indiflex <indiflex1@gmail.com>  
    Date:   Tue Jan 2 14:15:23 2024 +0900  
  
        Merge branch 'br11'  
  
*   commit 3eb52b1e3a4b5d6da4532ea7fb79598898c6ed1e (br11)  
    Author: indiflex <indiflex1@gmail.com>  
    Date:   Thu Dec 28 19:08:53 2023 +0900  
  
        aaa  
  
*   commit b7533b92d1916c23ce5097067c4e2d2b117fb913 (origin/master)  
    Author: indiflex <indiflex1@gmail.com>  
    Date:   Thu Dec 28 19:16:00 2023 +0900  
  
        aa
```



## 3-Way merge

- 공통 조상(base) [1]
- 작업 브랜치(feature) [2]
- 또 다른 작업 브랜치(master) [3]

```
master> git checkout -b feature # 작업 branch
```

```
feature> edit & commit # 작업(커밋F1)
```

```
feature> edit & commit # 작업(커밋F2)
```

```
feature> git switch master # 기준 branch
```

⇒ 아직 feature를 merge하지 않아 master는 수정 전.

```
master> edit & commit # 작업(커밋M)
```

⇒ master에서도 작업

```
master> git merge feature [-e | --edit]
```

# 3-way(F1 + F2 + M) merge

```
master> git log --graph # merge 확인
```

```
master> git ls-files -u # 충돌 확인
```

```
master> git branch -d feature # 완료 제거
```

# conflict & merge

```

i README.md ! x
i README.md > abc # 11111
You, 24 seconds ago | 2 authors (You and others)
1  ## git test
2  11111
3  22222
Accept Current Change | Accept Incoming Change
4  <<<<<< HEAD (Current Change)
5  33-local
6  =====
7  333-remote
8  >>>>>>
c4b83aaa34a455a050931d25d13f3418d

```

- local과 remote 동시 수정  
git pull # commit before merge!!  
git commit -am "333-local"  
git pull # conflict!!  
cf. **git config pull.rebase false**

- **conflict**

<<<< HEAD

local(me) 변경 내역

=====

remote(others) 변경 내역

>>>> sha1 id

- **git ls-files -u**
- **editor**에서 수정 & 저장 후 아래 명령 수행  
git commit -am "merge!"
- git log --graph --oneline

```

*   cfd4413 (HEAD -> master) mege!
|  \
|   * c4b83aa (origin/master) Update README.md
|  /
* | 29b1262 3-local
| /

```

## cf. merge by fetch

---

- pull by auto-merge
- fetch by manual-merge

### pull vs fetch

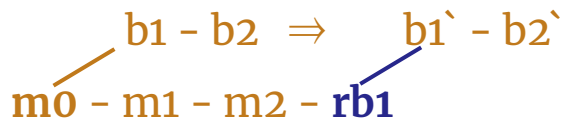
1. github에서 README.md 추가(또는 수정)
2. **git pull** # pull & auto-merge  
cat README.md
3. github에서 README.md 수정(변경)
4. **git fetch** # 임시 브랜치로 pull  
cat README.md # 변경 내역 안보임!
5. git log **origin/master** README.md  
# show remote log (fetch이후에만 보임!)
6. **git merge origin/master**
7. git log README.md  
cat README.md

## Rebase merge

(주의) push 전에 rebase하자!

(내 base를 master HEAD로!)

- re-base(base를 변경)

$b1 - b2 \Rightarrow b1' - b2'$   
  
 $m0 - m1 - m2 - rb1$

- 3-way → fast-forward 화
- master
- CommitID 변경
- br2는 최종적으로 merge 해야 함!

(주의) br을 push했다면 revert사용(:ID 변경)

**git rebase --abort / skip**  
**--continue**

```
master> git checkout -b br # 작업 branch (m0)
```

```
br> edit & commit # 작업 (커밋b1)
```

```
br> edit & commit # 작업 (커밋b2)
```

```
br> git switch master # 기준 branch
```

```
master> edit & commit # 작업 (커밋m1)
```

```
master> edit & commit # 작업 (커밋m2)
```

```
master> git checkout br # 작업 br로 이동
```

```
br> git rebase master # rebase
```

⇒ m2까지는 그대로, br1과 br2는 새로운 ID로 변경!!

```
br> git add -A # mark merge
```

```
br> git rebase --continue # rebase 마무리
```

```
br> git checkout master
```

```
master> git merge br # b1, b2 병합
```

```
master> git branch -d br
```

## cf. rebase의 또 다른 기능

- **git rebase -i HEAD~n**  
(n: 작업할 포인트 개수)

**m0** - **m1** - **m2** - **m3** - **m4**  
(11) (22) (33)

\$> **git rebase -i HEAD~3**  
**pick m2**

**s m3**

**s m4**

⇒ **m0** - **m1** - ~~**m2**~~ - ~~**m3**~~ - ~~**m4**~~ - **m5**  
(11) (22) (33) (11)

1. 마지막 n개의 커밋을 하나로 **합치기**

**git log -5 --oneline**

2. **git rebase -i HEAD~3** # interactive모드

**squash(s)**      **합칠 커밋ID**      # HEAD포함 3개

**pick**              **남길 커밋ID**

cf. **rebase the remote branch**

br> **git fetch -p**

br> **git rebase origin/master**

no-br> **merge editing for conflict**

no-br> **git add -A**

no-br> **git rebase --continue**

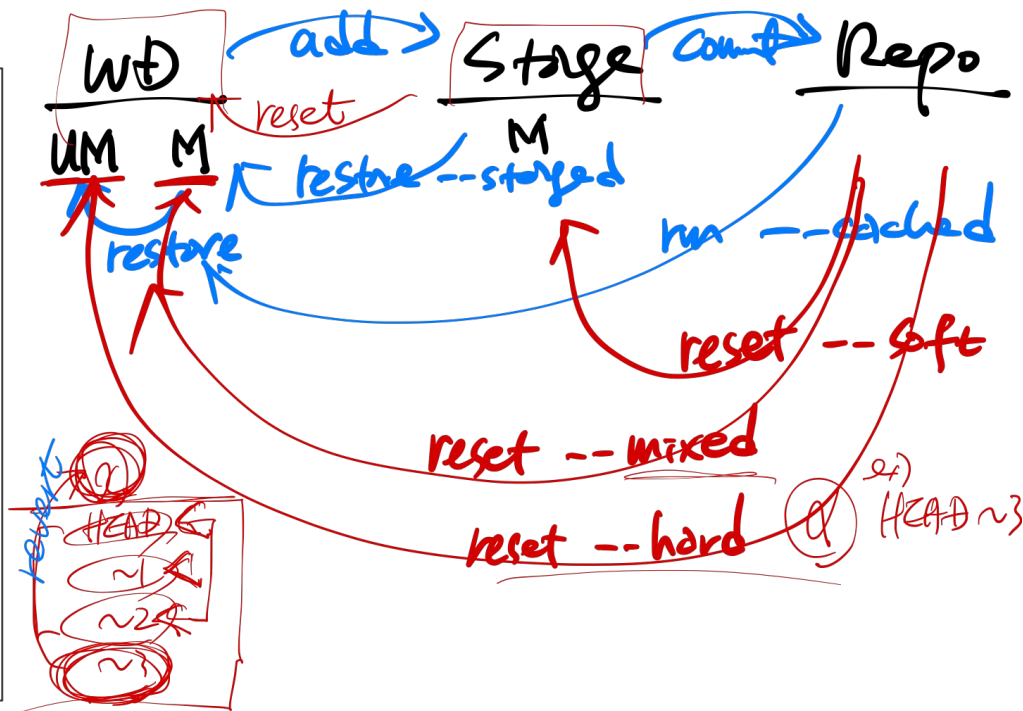
br> **git status**

## 06. 깃 리셋(reset), 리버트(revert), 태그(tag)

## reset (복귀, 되돌리기)

- soft : repo → stage
- mixed : repo → WD(M)
- hard : repo → 초기화! (수정전)
- merge : merge 취소

(주의) Commit Log(ID)를  
변경하므로, push된 내역은 reset을  
사용하면 안됨!!  
(push된 커밋은 reset후 pull 요구)



## reset options

Repo —reset --soft→ Stage

Repo —reset --mixed→  
WD(Modified)

Repo —reset --hard→  
WD(Unmodified)  
: 수정내역 삭제!!

\$> git reset [options] <target-HEAD>

1. `vi ttt.txt` # 수정
2. `git commit -am "c1"` # commit (C1)
3. `edit & commit` # commit (C2)  
`edit & commit` # commit (C3)
4. `git reset --soft HEAD~1` # Repo → Stage
5. `git log -5 --oneline`  
`git status` # C3 제거되고 staged
6. `git reset --mixed HEAD~1` # Repo → WD(M)
7. `git reset --hard HEAD~1` # Repo → WD(UM)  
⇒ 수정내역 모두 사라짐!!

cf.

alias gl5='git log --oneline -5'



## reset의 또 다른 기능

- stage(add) 초기화
- merge(병합) 취소

### 1. Stage 초기화

```
edit ttt.txt & aaa.txt
```

```
git add -A # make staged
```

```
git reset # --mixed HEAD 생략
```

```
⇒ ⇒ ⇒ stage → WD(modified)
```

```
git reset --hard # 수정 내역까지 삭제!
```

### 2. merge 취소

```
master> git checkout -b hotfix # 생성
```

```
hotfix> edit & commit # 작업
```

```
hotfix> git checkout master
```

```
master> git merge hotfix # 병합
```

```
master> git reset --merge HEAD~1 # 취소!
```

```
master> git log # merge log
```

## revert

기존 Commit 남겨두고 되돌리기.  
즉, **push**된 내역 되돌리기!

cf. 마지막 커밋 수정 (CID 변경)  
**git commit --amend -m 'xx'**

1. `vi ttt.txt`
2. `git commit -am "before revert"`
3. `git log -3 --oneline` # 지난 3 commits
4. `git revert HEAD` # 방금 전 commit만 취소
5. `edit & commit 2 times`
6. `git revert HEAD~2`  
# 두 커밋 전과 conflict! (**both-modified**)
7. `vi ttt.txt` # 수동 merge
8. `git add ttt.txt` # make staged(merged)  
# modified (in stage)
9. `git revert --continue` # 수동 merge

```
You are currently reverting commit b9aba2f.
(all conflicts fixed: run "git revert --continue")
(use "git revert --skip" to skip this patch)
(use "git revert --abort" to cancel the revert operation)
```

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   ttt.txt
```

```
~/workspace/gitplay/gitpp2 (master) $ git revert --continue
[master fb1e8ab] Revert "Revert "5555555""
 1 file changed, 4 insertions(+)
~/workspace/gitplay/gitpp2 (master) $ git log --oneline -3
fb1e8ab (HEAD -> master) Revert "Revert "5555555""
dc6c4d0 (origin/master) r22
```

## cf. VSCode GitLens after revert

```
dsfdsakfjkals;dsfdsakfjkalssfdas
<<<<<< HEAD
a
sdf
dsfdsakfjkalssfdasdfasd
dsfdsakfjkalsdsa
f
dsfdsakfjkalssfdasdfasf
as
dsfdsakfjkals
dsfdsakfjkalsdsa
f
ads
d
sasdfa
=====
DATE=`date +%Y-%m-%d` "%H:%M`
MSG="$DATE lesson"
if [ $# -gt 0 ]; then
    MSG="$DATE - $1"
fi

git add --all
git commit -am "${MSG}"

git push
>>>>>> parent of cacf34a... LAST
```



GitLens — Git supercharged v14.7.0

GitKraken [gitkraken.com](https://gitkraken.com) | 28,855,755 | ★★★★★ (761)

Supercharge Git within VS Code — Visualize code authorship at a glance ...

[Disable](#) [Uninstall](#) [Switch to Pre-Release Version](#) ⚙

This extension is enabled globally.

현재 변경 사항 수락 | 수신 변경 사항 수락 | 두 변경 사항 모두 수락 | 변경 사항 비교

<<<<<< HEAD (현재 변경 사항)

```
a
sdf
dsfdsakfjkalssfdasdfasd
dsfdsakfjkalsdsa
f
dsfdsakfjkalssfdasdfasf
as
dsfdsakfjkals
dsfdsakfjkalsdsa
f
ads
d
sasdfa
=====
```

DATE=`date +%Y-%m-%d` "%H:%M`

MSG="\$DATE lesson"

if [ \$# -gt 0 ]; then

MSG="\$DATE - \$1"

fi

git add --all

git commit -am "\${MSG}"

git push

>>>>>> parent of cacf34a... LAST (수신 변경 사항)

## tag (배포 꼬리표, 특정 커밋 태깅)

\* Annotated: 태그 + 정보

\* Lightweight: 태그 이름만!

- `git tag -a <tag>`

- `git tag -l/d`

cf. SemVer(Semantic Versioning)

1. `git tag [-l]` # tag list
2. `git tag -a <tag> [ID]` # make tag  
annotated-tag  
release note 작성 (edit tag info)
3. `git log` # tag 정보 보임  
`git tag -l` # tag list
4. `git show <tag>` # 해당 tag 변경 사항
5. `git tag -d <tag>` # 특정 tag 삭제
6. `git checkout <tag>` # 특정 tag로 이동  
`git checkout -b <br> <tag>` # br 생성
7. `git push origin <tag>` # push

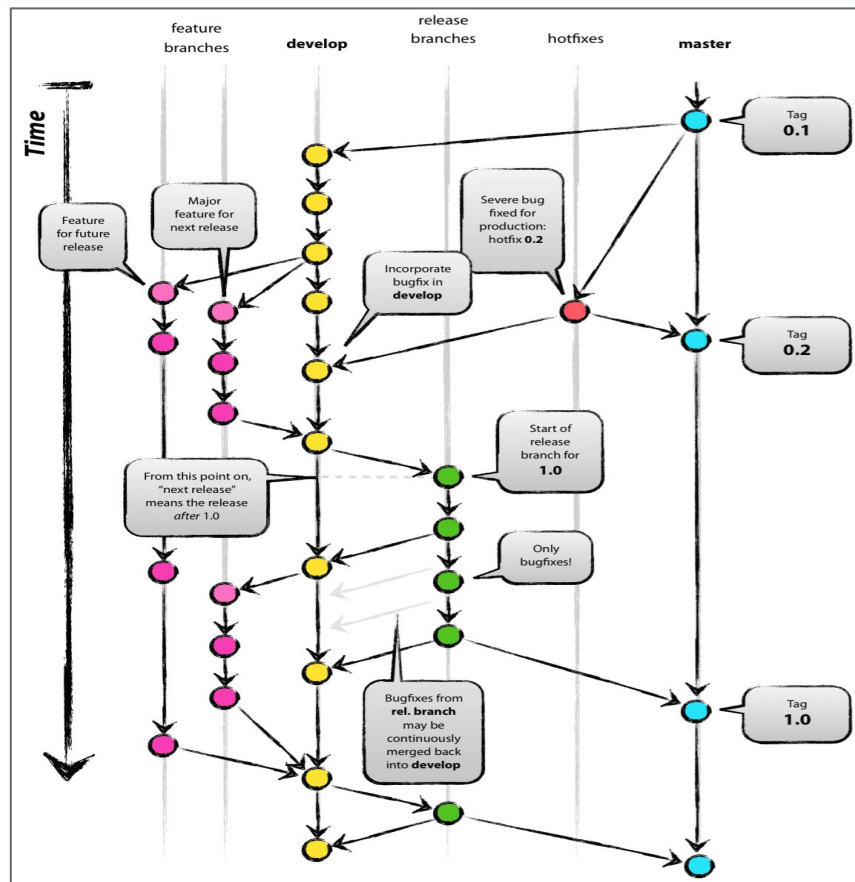
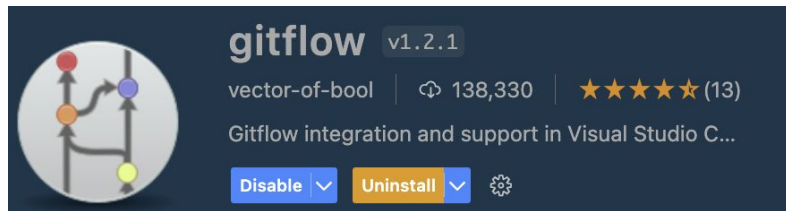
## 07. 깃플로우(Git-Flow)

# Git Flow Concepts

- **master**
- **develop**
- **feature/**
- **release/**
- **hotfix/**
- **support/** (only git flow cli)

강의영상

<https://youtu.be/VY8K9FguM-E>



<https://nvie.com/posts/a-successful-git-branching-model/>

# Install Git Flow CLI

## # Git Flow Settings

1. Install **gitflow** VSCode extension
2. <https://github.com/nvie/gitflow>



### ## Windows

[guide](#)

1. Install MSysGit  
<https://gitforwindows.org/>
  2. download dll files  
libiconv-1.9.2-1-bin/bin/libiconv2.dll  
libintl-0.14.4-bin/bin/libintl3.dll  
util-linux-ng-2.14.1-bin/bin/getopt.exe
  3. copy to msysgit's bin  
(ex. C:\Program Files (x86)\Git\bin)
- ```
$> git clone --recursive git://github.com/nvie/gitflow.git
$> cd gitflow
$> git config --global url."https://github".insteadOf git://github
```

```
C:\gitflow> contrib\msysgit-install.cmd "[path to git installed folder]"
$> ln -s "/C/Program Files (x86)/Git/bin/git-flow" git-flow
```

### ## Mac

[guide](#)

```
$> brew install git-flow
```

### ## Linux

[guide](#)

```
$> sudo -
$> dnf install git -y
$> curl -OL
https://raw.githubusercontent.com/nvie/gitflow/devel/contrib/gitflow-installer.sh
$> chmod +x gitflow-installer.sh
$> ./gitflow-installer.sh
```

# Git Flow Commands

```
$> git flow init
$> git push origin --all
$> git flow feature start <feature-branch>
$> git flow feature list
$> git add .
$> git commit -m
# push & pull-request(to review)
$> git flow feature finish <feature-branch>
$ other-feature> git merge --no-ff develop
$> git flow release start <tag>
$> git flow release finish <tag>
$> git tag -l
$> git push origin --all --follow-tags
$> git flow hotfix start <tag>
$> git flow hotfix finish <tag>
```

```
## example scripts
$> git flow init
$> git push origin --all
$> git flow feature start login
$> git merge --no-ff develop
```

```
~/workspace/worki/admx (main) $ git flow init

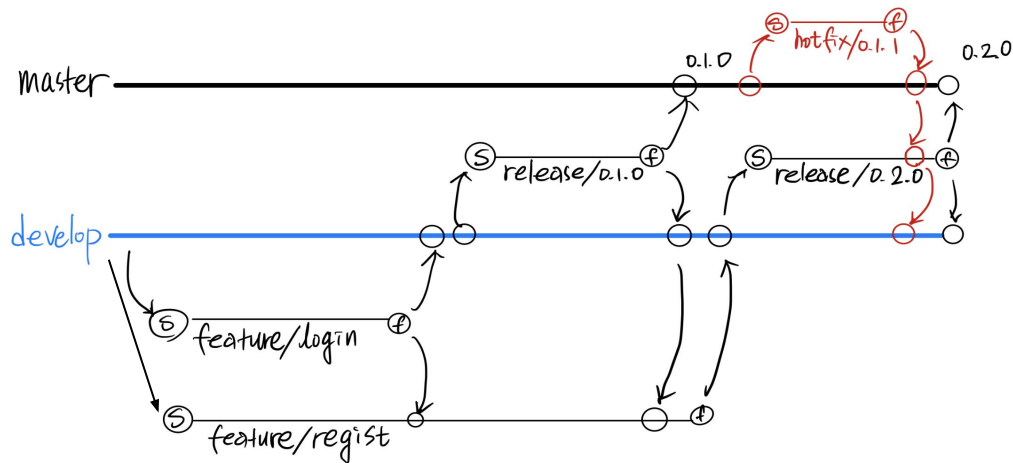
Which branch should be used for bringing forth production releases?
- main
Branch name for production releases: [main]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
```



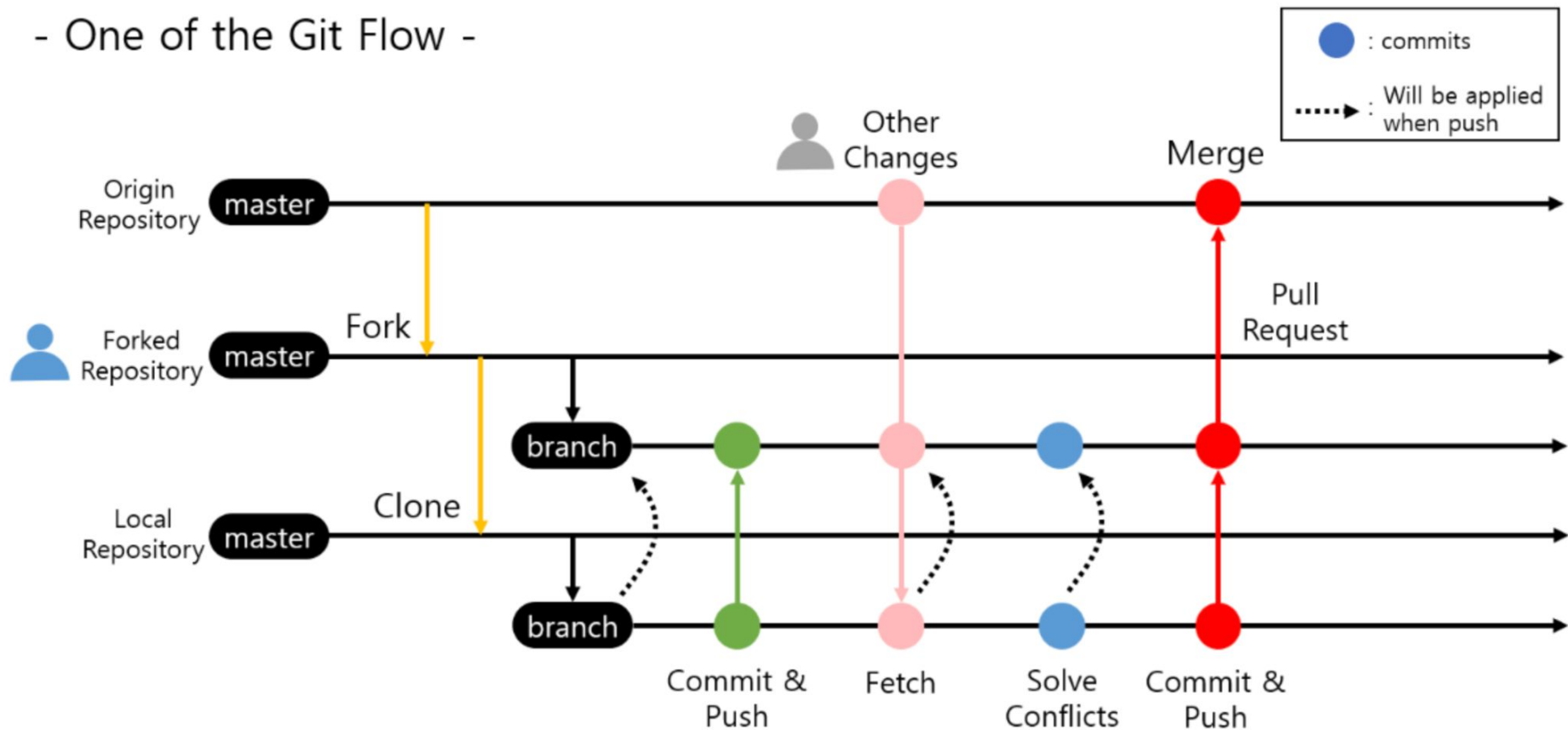
# 연습문제

1. 2개 feature 시작 (login, regist)
2. login, regist 각각 코딩
3. login 작업 완료
4. login 코드 리뷰
5. login 릴리즈 & 배포 (0.1.0)
6. regist 작업완료
7. regist 릴리즈 (0.2.0)
8. **0.1.0에서 버그 발생!!** → hotfix 시작
9. hotfix 0.1.1 작업완료
10. hotfix 0.1.1 코드리뷰
11. hotfix 0.1.1 배포(0.1.1)
12. regist 릴리즈 마무리 & 배포(0.2.0)



## 08. Fork & Pull-Request

## - One of the Git Flow -



 **develop** had recent pushes 21 minutes ago

Compare & pull request

### Existing forks

 indiflex/test11-do

+ Create a new fork

2 Commits

Write

Preview

H

B

I

≡

<>

🔗

≡

≡

🔗

🔗

🔗

🔗

🔗

🔗

🔗

Add your description here...

Markdown is supported

Paste, drop, or click to add files

Create pull request

## 승희님 테스트 PR이에요 #1

Open

indiflex wants to merge 1 commit into [doSeung11:main](#) from [indiflex:fetch-ttt](#)

Conversation 0

Commits 1

Checks 0

Files changed 1



indiflex commented 2 minutes ago

No description provided.



ttt

ec8f3bf

# 연습문제

옆 사람의 리포를  
Fork하고  
수정 후 PR을 달려보자.

1. 옆 사람의 Repo로 이동
2. 해당 repo에서 create fork
3. fork한 repo를 PC에 clone
4. pc에 원본(옆사람) 리포도 remote로 등록  
`git remote add <별칭> https://github.com/<owner>/<repo>`
5. <작업 branch> 생성
6. `commit & push <작업branch>`
7. PR 생성



**Thank  
you**

