Version Control (GIT)

Week 5

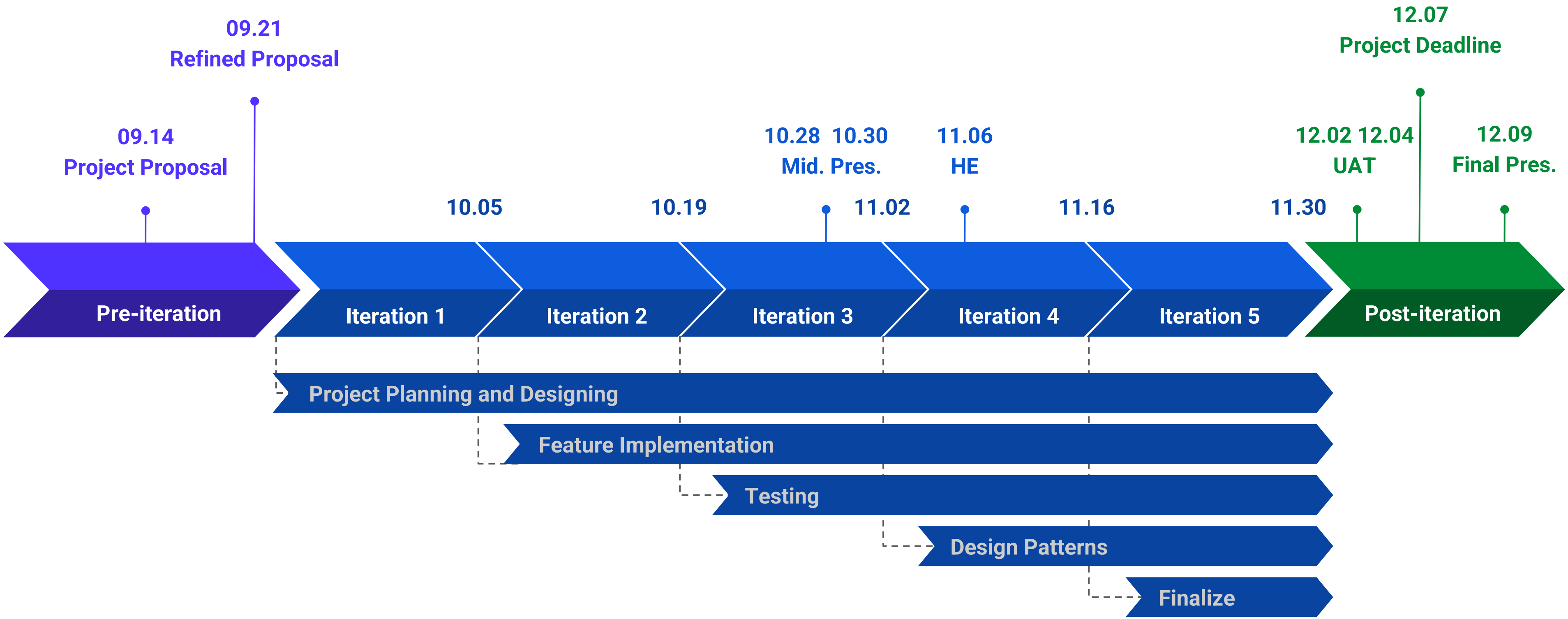
A man is more a man through the things he keeps to himself than through those he says.

— Albert Camus, The Myth of Sisyphus

Where Are We?

- We understand the importance of process
- We understand the importance of project management
- We have a good project idea
- We have initial schedule for the project
- We learn Android basics

Project Process Overview



Objectives

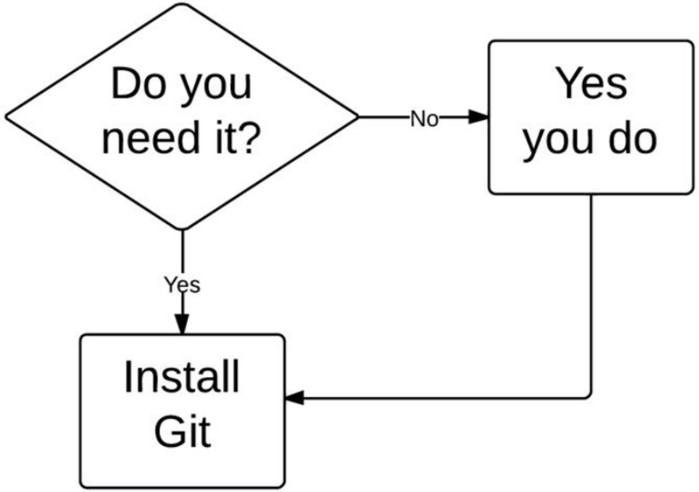
- Understand the basics of version control
- Understand how to manage the development history and work with others with Git and GitHub
- Establish good Git habits

Contents

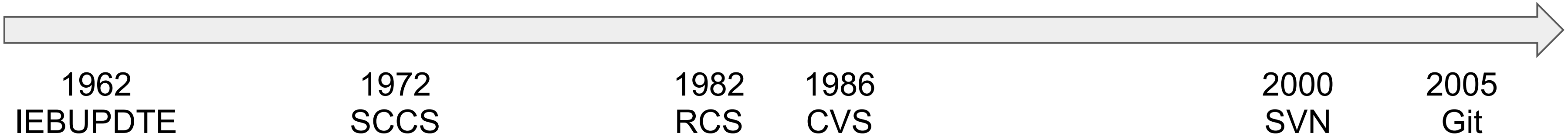
- Version control overview
- Basics of Git
- Remote with Git
- GitHub
- Collaborating with Git
- Git branching strategies

Version Control

- Tracking changes of files as snapshots
- Why version control?
  - Checkpoint and track changes
  - Collaborative development
- How version control?
  - Use GIT!



History of Version Control Systems (1/2)



1962 IEBUPDTE

1972 SCCS

1982 RCS

1986 CVS

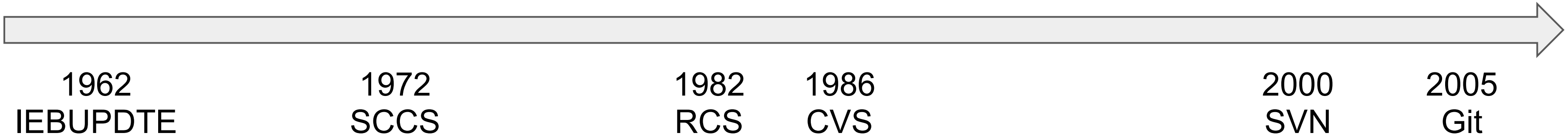
2000 SVN

2005 Git

- IEBUPDTE
  - Precursor of modern version control systems
  - Use punch cards to store data
- Source Code Control System (SCCS)
  - Create, edit, track changes
  - Single user only
- Revision Control System (RCS)
  - Reverse delta-based efficient implementation
  - Single user only

Source: A History of Version Control

History of Version Control Systems (2/2)



1962 IEBUPDTE

1972 SCCS

1982 RCS

1986 CVS

2000 SVN

2005 Git

- Concurrent Versions Systems (CVS)
  - Support multiple users
  - Widely adopted in open source projects
- Subversion (SVN)
  - Improve and fix bugs of CVS
  - Centralized version control system (CVCS)
- Git
  - Distributed version control system (DVCS)
  - *De facto* standard

Source: A History of Version Control

Centralized vs. Distributed

- CVCS (e.g., svn) maintains a main repository on a server
  - Single main repository on a central server
  - Users must pull the latest version before editing
  - Most operations (commit, log, revert) require server communication
  - Lower fault tolerance: server failure blocks collaboration
  - Difficult to modify the same code section by multiple users

Source: What is a centralized version control system

Centralized vs. Distributed

- In DVCS (e.g., git), each user has their own copy of the repository
    - Each user has a full copy of the repository
    - Most operations work locally without server access
    - Higher fault tolerance: work continues even if the server is down
    - Easier branching and merging for collaborative development
    - Supports offline work and faster operations

SVN Usage Example

- Checkout code from central server:

  ```
  svn checkout https://server/project
  ```

- Commit requires server access:

  ```
  svn commit -m "Fix bug in module"
  ```

- View history always from server:

  ```
  svn log
  ```

- If server is down → no commit/log possible
- Collaboration depends on central server availability

Git Usage Example

- Clone full repository (local copy):

  ```
  git clone https://github.com/user/project.git
  ```

- Commit locally (no server needed):

  ```
  git commit -m "Fix bug in module"
  ```

- View history locally:

  ```
  git log
  ```

- Push to server only when ready:

  ```
  git push origin main
  ```

- Work offline and sync later, high fault tolerance

Git

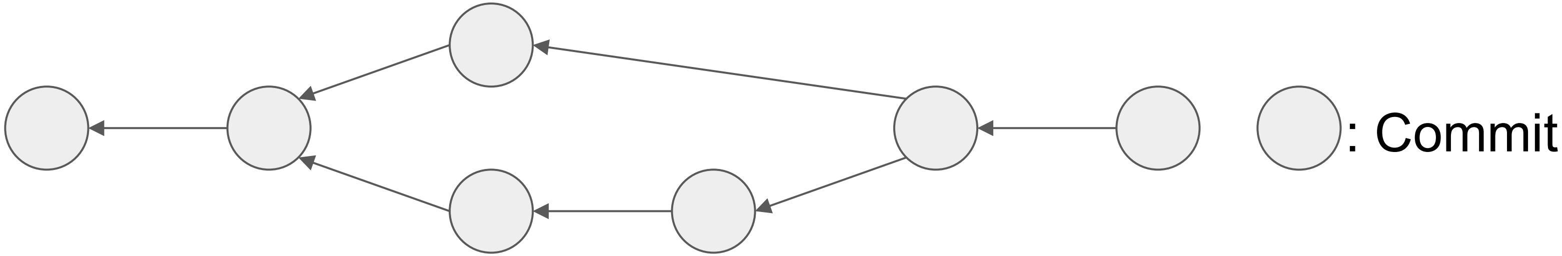
- Most popular distributed version control system
- Released on 7th April 2005, by Linus Torvalds
  - The development began on 3rd April 2005
- Latest stable version is 2.50.1, released on June 16, 2025
- Open source software

Start with Git

- `git init`
- Creates `.git` directory in your current working directory
- No sub-directory can have their own `.git` directory

Data Model of Git

- Directed Acyclic Graph (DAG) of commits



○: Commit

What is a Commit?

- Snapshot
  - Traced blobs (files) and trees (directories) at that moment

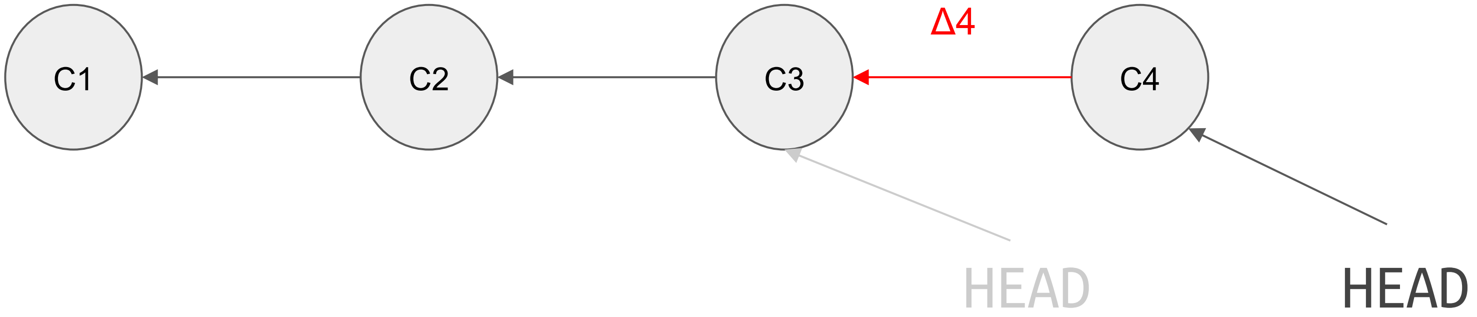
- Metadata
  - ID: **hash** of the commit (e.g. 2fa98c9320b…) ⇒ **Immutable**!
  - Author: One who committed
  - Message: Commit message written by the author
  - Etc.

Usage of Git: Basics

- Creating commits
- Checking commit history
- Navigating commits
- Managing modifications
    - Stash
    - Restore
    - Undo commits
    - Revert commits

Creating Commits

- Make a snapshot of current blobs and trees
- Store them into the git repository
- Create the metadata of a commit
- Update the `HEAD` pointer (the currently looking commit)

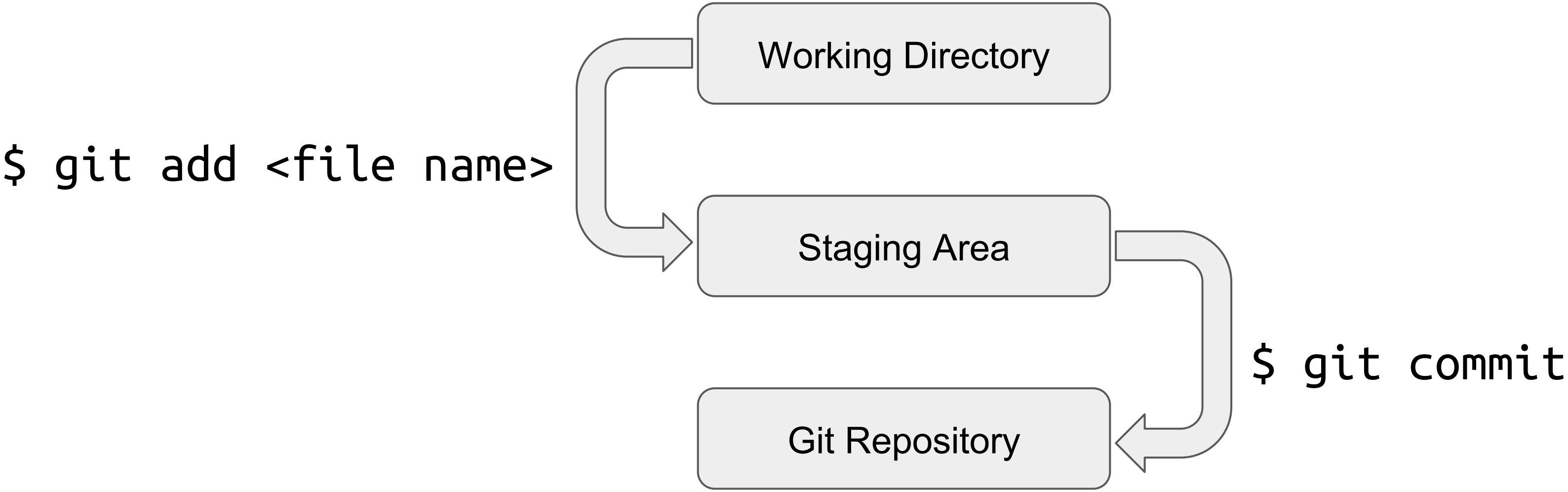


Creating Commits: HEAD Pointer

- Indicates a commit that a user currently looks at
- `HEAD` can directly point a **commit**
- `HEAD` usually points a **branch** (more details later)
  - `HEAD` directly pointing a commit is known as detached head
  - Not recommended

Creating Commits: Staging Area

- Updates not directly committed but go through staging area
  - Intermediate area before adding commits
  - Control blobs to commit among all modified blobs



Working Directory

$ git add <file name>

Staging Area

$ git commit

Git Repository

Creating Commits: Three States

- Modified blobs
  - Git traces all blobs that are different from parent commit’s
- Staged blobs
  - User mark modified blobs to go into the next commit
  - `git add <file name>`
- Committed blobs
  - Snapshot staged blobs
  - `git commit`
- Check them with `git status`

Checking Commit History

- `git log`
  - History of commits
  - Metadata
  - `HEAD`

- Several useful options
  - --all, --graph, --oneline, etc.

Navigating Commits

- Change the commit that you look (`HEAD` changes)
- Make sure your working directory is clean (no modified or staged files)
- Instructions
  - `git checkout <commit id>`
  - `git switch -d <commit id>`
  - `HEAD` directly points the commit
  - The “detached `HEAD`” state

Managing Modifications: Stashing

- Stashing, unlike committing, does not remain in the history but provides a mechanism to safely store and restore changes in the working directory and staging area.
- Usually for navigating while work is not enough to commit
- Instructions
  - `git stash`: save modifications in a stack
  - `git stash list`: show the stack
  - `git stash pop`: apply saved modifications

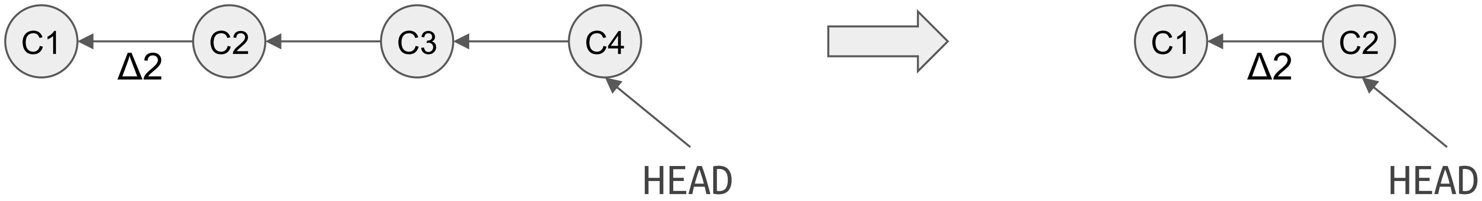
Managing Modifications: Restoring

- Restore modified blobs
  - Overwrite with blobs from the `HEAD` commit
  - Modifications are lost
  - User can select blobs to overwrite
- Instructions
  - `git checkout <file name>`
  - `git restore <file name>`

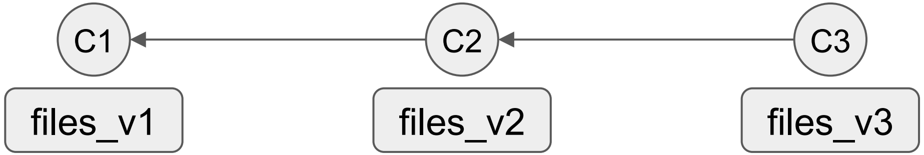
Managing Modifications: Undo Commits (1/2)

- Use `git reset` to
  - Simply changing the commit pointed by branch and `HEAD`
- Delete all commits after the determined commit
- You will **lose all your edits** with `git reset --hard`

```
$ git reset C2
```



Managing Modifications: Undo Commits (2/2)



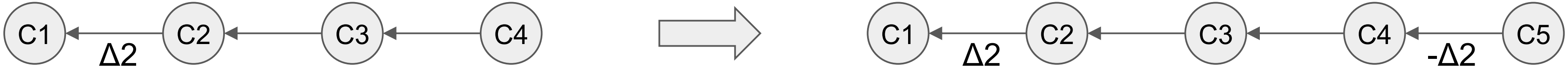
```
$ git reset --soft C1

$ git reset --mixed C1

$ git reset --hard C1
```

HEAD	Staging area	Modified blobs
C3	-	-
C1	files_v3	files_v3
C1	-	files_v3
C1	-	-

Managing Modifications: Revert Commits

- `git revert <commit id>`
- **Create** a new commit by applying the **reverse patch (diff)**
- No deletion of commits

```
$ git revert C2
```

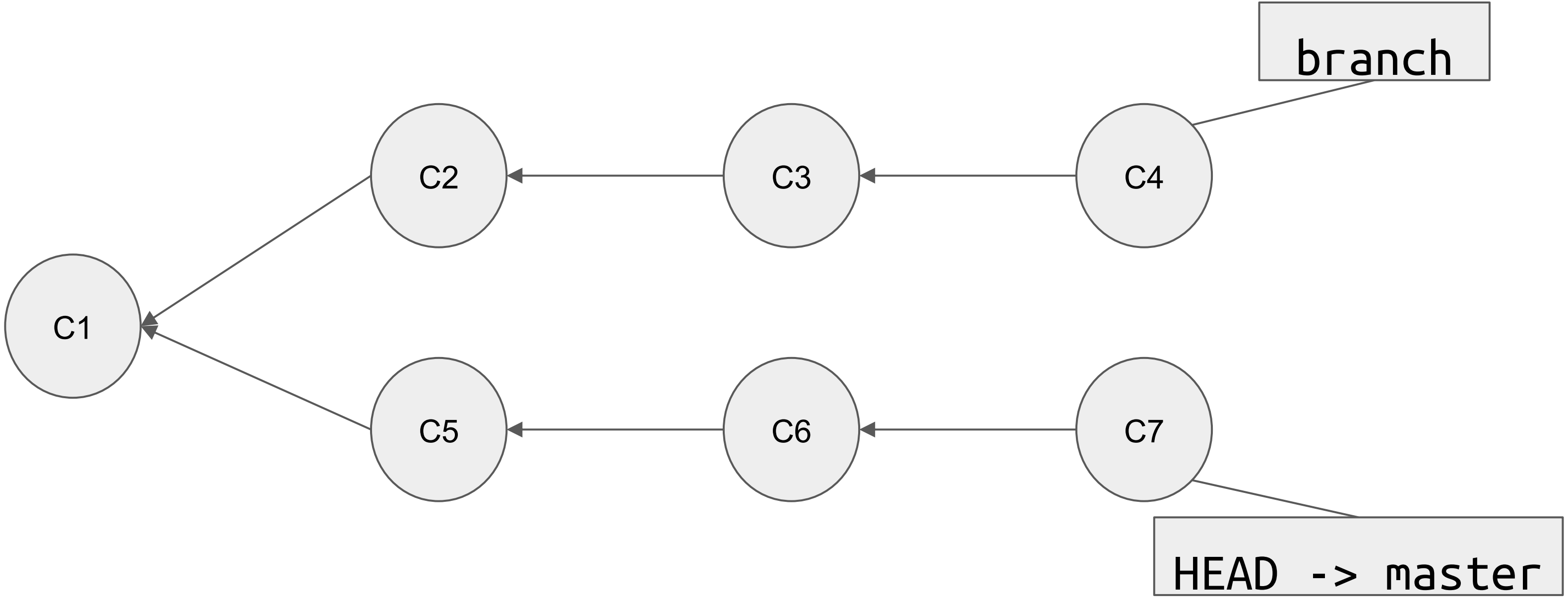


Be Careful about Deleting Commits!

- Do not delete commits pushed to the shared repository
  - `rebase` → `merge`
  - `reset` → `revert`
- If you delete a commit but someone was working on it, that commit will revive!
  - Multiple commits with same commit message
  - Mess up the commit history
  - See the example
- **Never** do `git push --force`

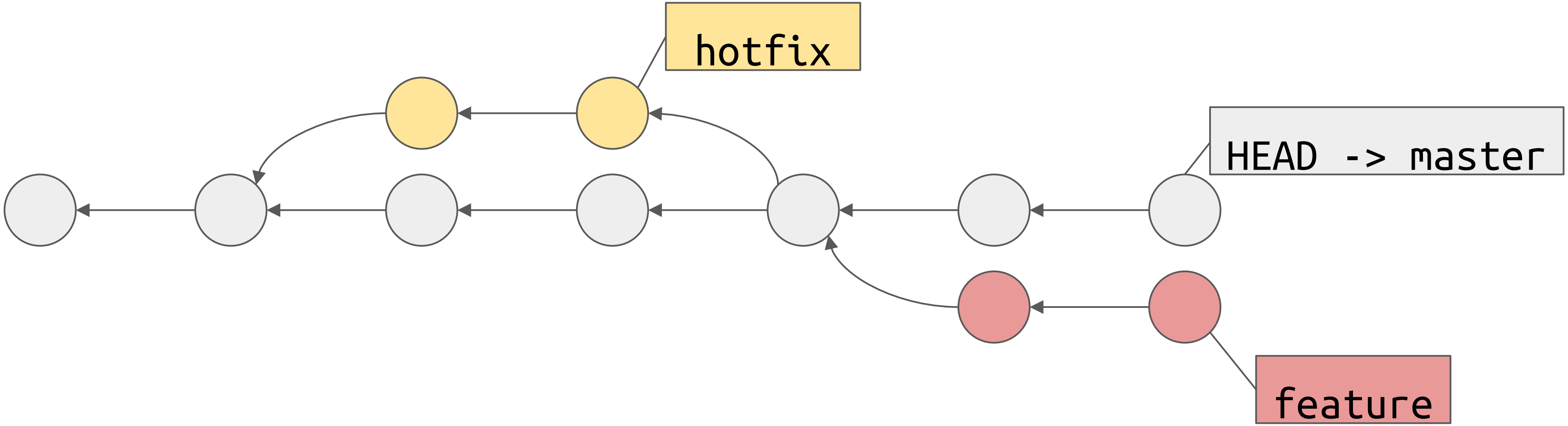
Advanced Usage: Branch

- Can we manage multiple commit paths to
  - Separate the implementations of multiple features?
  - Work with other people in parallel?



Branch

- Separations of concerns by having multiple paths
- Diverge from the main branch and continue work
- You can create and merge branches from any commit!



Branch

- **Pointer** to a commit
- `master` branch (`main` branch)
    - Default branch when you initialize Git
    - Convention: Keep `master` branch stable!
- If you create a new commit,
    - `HEAD` and the branch pointed by `HEAD` point the **created commit**
- `git branch`
    - Show branches and indicate current branch

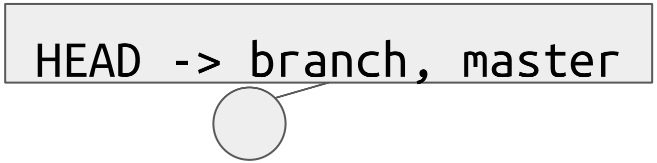
Creating Branch

- Create a new branch from the current commit
- Instructions
  - `git branch <branch name>`
    - Create a new branch only (does not switch to it)
  - `git switch -c <branch name>`
    - Create a new branch **and** switch to it
    - Move HEAD to the new branch
  - `git checkout -b <branch name>`
    - Legacy form, same as switch -c, but switch is now recommended

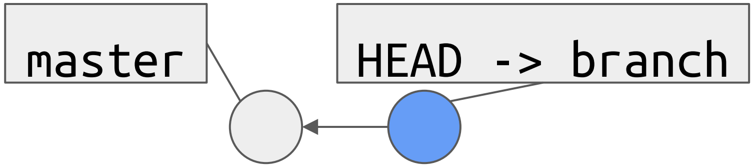
Merging Branches

- Multiple commits can share same parent
- Such divergence is managed by branches
- We can merge branches to **integrate changes**
- `git merge <branch name>`

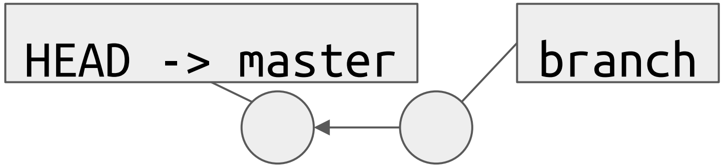
Merging Branches: Example



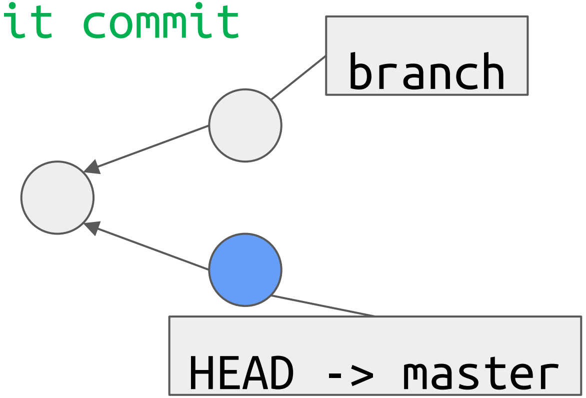
```
$ git commit
```



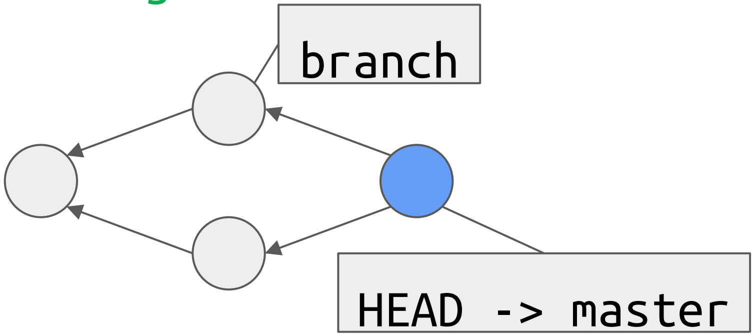
```
$ git switch master
```



```
$ git commit
```

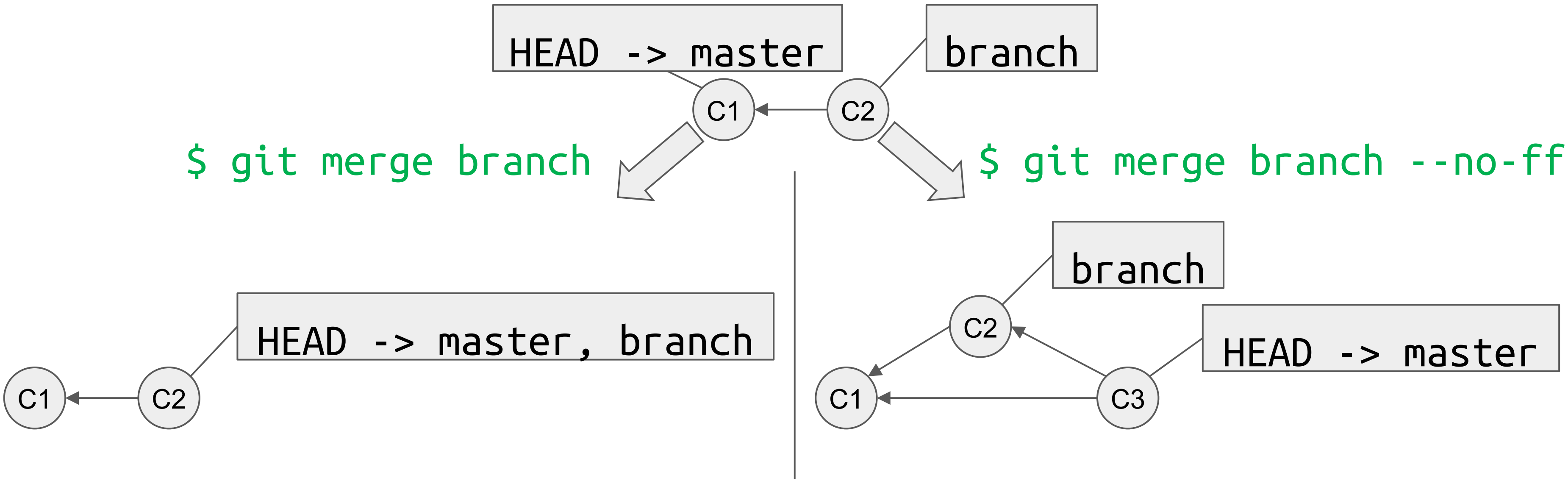


```
$ git merge branch
```



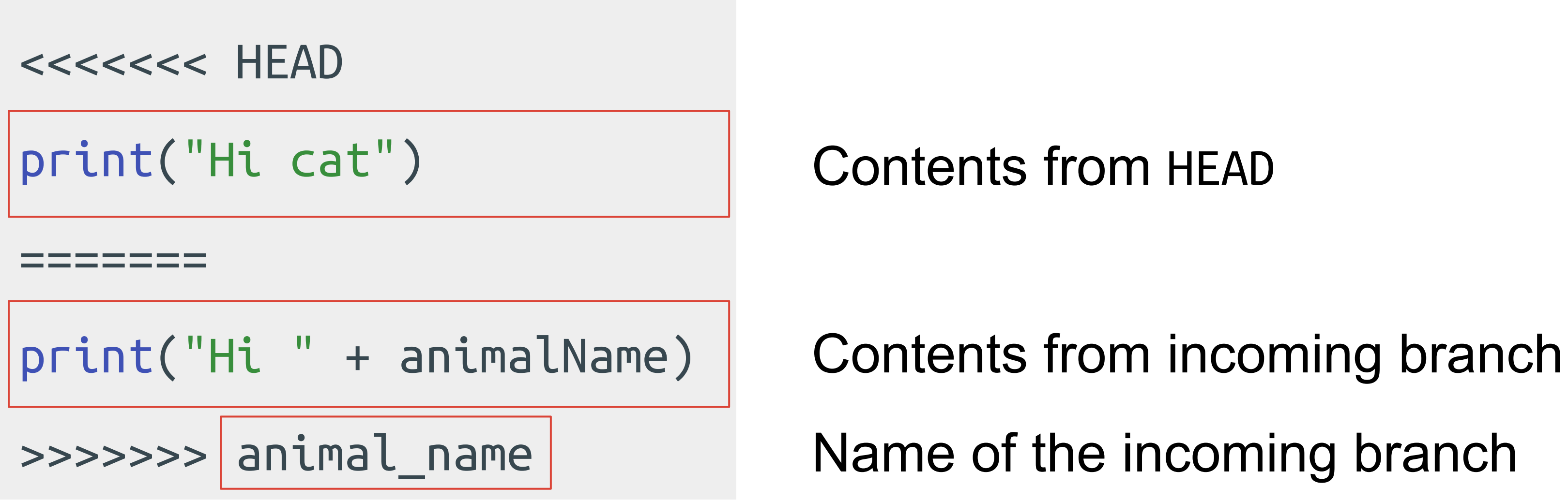
Merging Branches: Fast Forward

- If incoming branch is ahead of `HEAD`, **fast-forward**
- Use `--no-ff` option to create a merge commit



Merge Conflicts

- Conflict occurs if automatic merge is impossible,
    - User must **manually** solve it



```
<<<<<<< HEAD
print("Hi cat")
=======
print("Hi " + animalName)
>>>>>>> animal_name
```

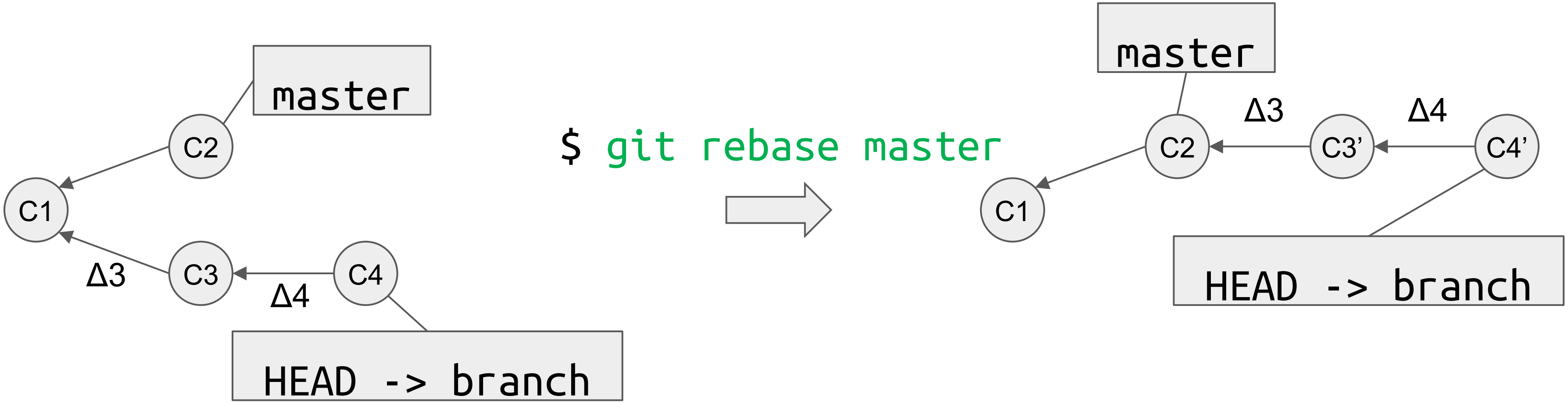
Contents from HEAD

Contents from incoming branch

Name of the incoming branch

Rebasing

- Similar behavior with `git merge`
    - Extract **patch** (difference) from the current branch
    - **Delete** commits from `HEAD`
    - Make new commits by **applying** the patch to the base branch
    - Make the branch to point the new commit



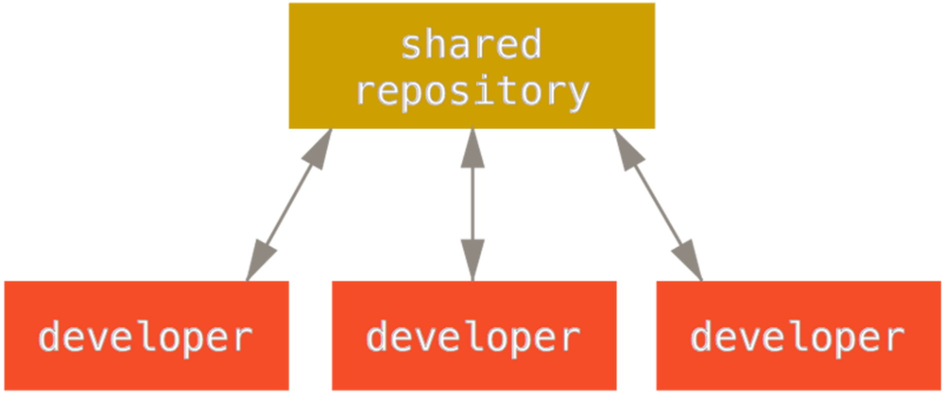
```
$ git rebase master
```

Rebase Vs. Merge

	Pro	Con
`rebase`	Simple linear history	Deleting commits
`merge`	Commit metadata = Actual work (Time, Author, etc.)	Hard to track history

Git Remote

- Multiple users interact with shared repository
    - The largest host is GitHub
    - Others: GitLab, Bitbucket, etc
- `git remote add <remote name> <url>`
    - `origin` is widely used for the `<remote name>`



Git Remote Commands

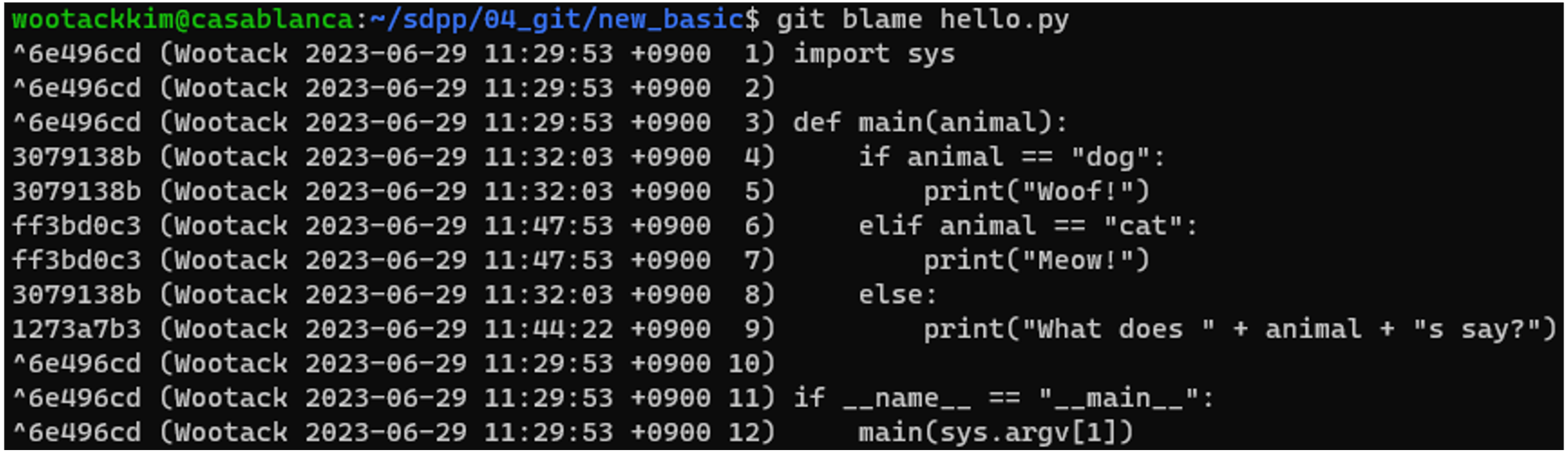
- `git clone <url> <directory name>`
  - Copy the entire history from remote repository into new directory
- `git push <remote name> <branch>`
  - Reflect changes on the branch to the remote repository
- `git fetch <remote name> <branch>`
  - Download changes from the remote repository
- `git pull <remote name> <branch>`
  - `fetch` + `merge`

Fallacies

- Git must be used with GitHub
  - No! GitHub is no more than a host of Git
  - Git is helpful in local usage
- Only urls are allowed for remote repositories
  - No! You can select your local directories as remote repositories
  - `git clone existing_directory new_directory`

Blame

- `git blame <blob name>`
  - Show last modified commit and author of each line
- Main purpose is to understand why such line is written
  - Not “blaming” your colleagues



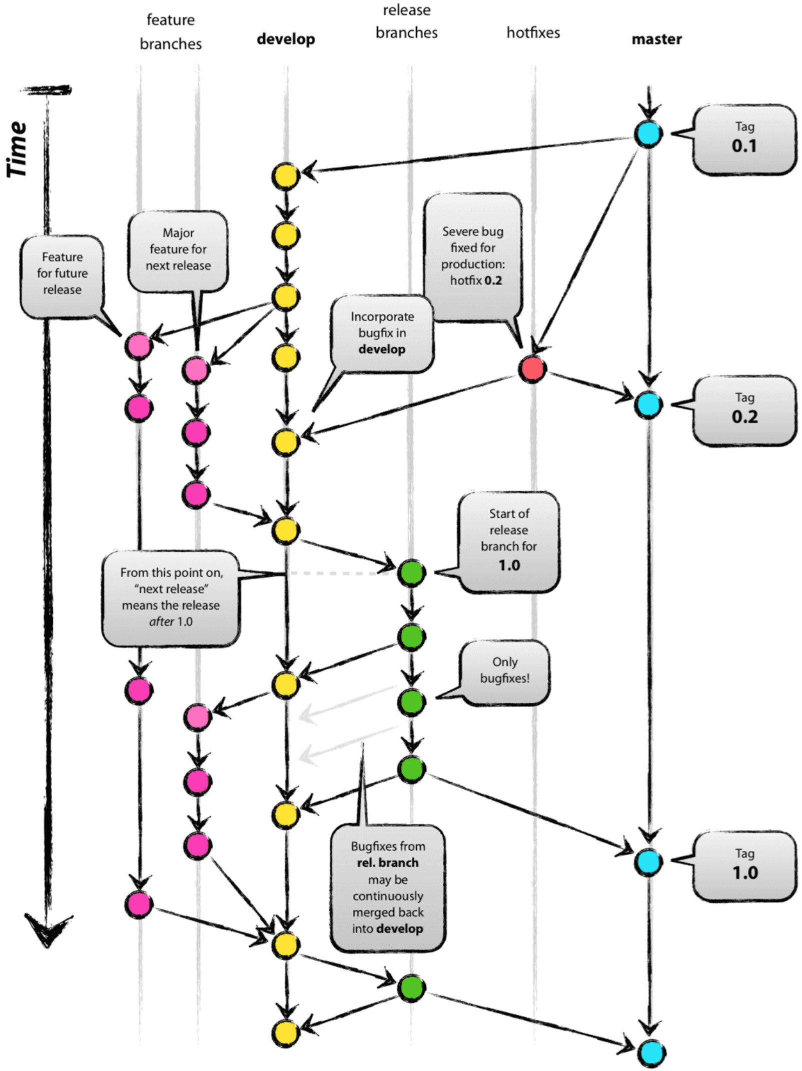
```
wootackkim@casablanca:~/sdpp/04_git/new_basic$ git blame hello.py
^6e496cd (Wootack 2023-06-29 11:29:53 +0900  1) import sys
^6e496cd (Wootack 2023-06-29 11:29:53 +0900  2)
^6e496cd (Wootack 2023-06-29 11:29:53 +0900  3) def main(animal):
3079138b (Wootack 2023-06-29 11:32:03 +0900  4)     if animal == "dog":
3079138b (Wootack 2023-06-29 11:32:03 +0900  5)         print("Woof!")
ff3bd0c3 (Wootack 2023-06-29 11:47:53 +0900  6)     elif animal == "cat":
ff3bd0c3 (Wootack 2023-06-29 11:47:53 +0900  7)         print("Meow!")
3079138b (Wootack 2023-06-29 11:32:03 +0900  8)     else:
1273a7b3 (Wootack 2023-06-29 11:44:22 +0900  9)         print("What does " + animal + "s say?")
^6e496cd (Wootack 2023-06-29 11:29:53 +0900 10)
^6e496cd (Wootack 2023-06-29 11:29:53 +0900 11) if __name__ == "__main__":
^6e496cd (Wootack 2023-06-29 11:29:53 +0900 12)     main(sys.argv[1])
```

.gitignore

- Some blobs you will never want to track with Git
  - Gigantic raw data
  - Compiled objects
  - Private information (e.g., password, API key)
  - OS-specific blobs and trees (e.g., _MACOSX, .DS_Store)
- Git does not trace blobs stated in `.gitignore`
- .gitignore files can be managed hierarchically
- You can start with templates

Git Flow (1/3)

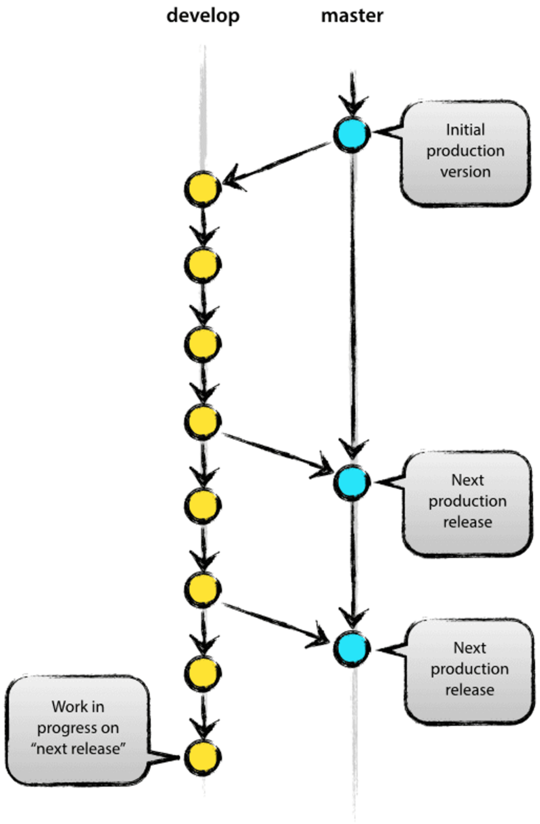
- Major branches
  - Single and permanent
  - `master`
  - `develop`
- Supporting branches
  - Short-living
  - Created on-demand
  - `feature`
  - `release`
  - `hotfix`


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Feature for future release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Source: A successful Git branching model

Git Flow (2/3)

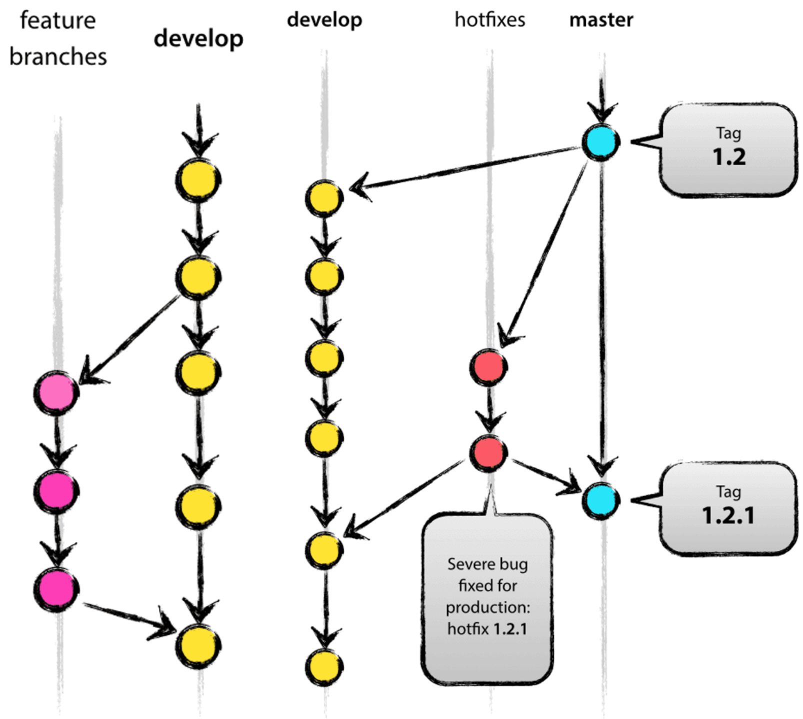
- `master` branch
  - Always in **production-ready** state
  - Tagged with release version
- `develop` branch
  - Development changes for next release
- Do not fast-forward to these branches
  - `git merge --no-ff`


develop
master
Initial production version
Next production release
Next production release
Work in progress on "next release"

Source: A successful Git branching model

Git Flow (3/3)

- `feature` branches
  - Where we implement new features
  - Merged into `develop`
- `release` branches
  - Detailed check for release
  - Bug fix, meta-data, etc.
  - Merged into `master` & `develop`
- `hotfix` branches
  - Fix urgent bugs from master
  - Merged into `master` & `develop`



Source: A successful Git branching model

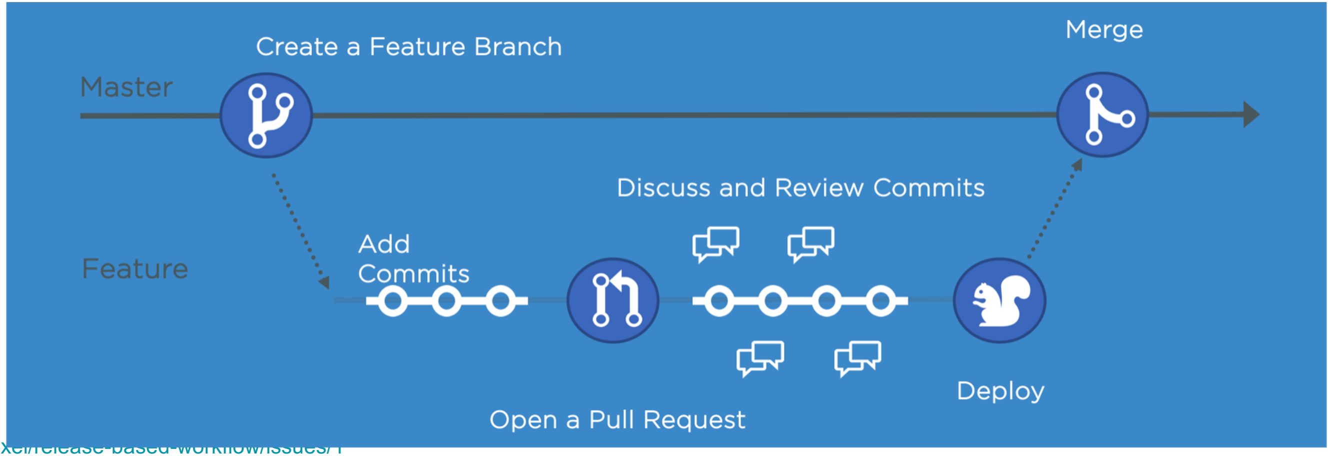
GitHub Flow (1/2)

- Simpler and faster Git branching strategy
  - Better for teams with short release interval
- Principles
  - Anything in the `master` branch is deployable
  - Create descriptive branches off of `master`
  - `push` to named branches constantly
  - Open a pull request at any time
  - `merge` only after pull request review
  - Deploy immediately after review

Source: GitHub Flow - Scott Chacon

GitHub Flow (2/2)

- `master` branch
  - Always stable and safe to deploy
- All other branches are created on-demand
  - Feature implementation, review, and testing
  - Hotfixes



[1]

[1] https://github.com/SvanBoxel/release-based-workflow/issues/1

Collaborating with Git (1/4)

- Keep commits and branches concise
  - Single commit is a “**logically separate changeset**”
  - Single branch handles **single feature**
  - Merge after each feature is done (frequently!)
    - Delete merged branches
  - Keep the `master` stable

Collaborating with Git (2/4)

- Write commit messages well
  - Use imperative form
  - Let others know changes without looking the source code
  - Explain “why”, “for what”, and “how”
  - We recommend the Conventional Commits specification

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Collaborating with Git (3/4)

- Write good issues
  - Avoid redundant issues
    - Search before report
  - One feature per issue
  - Reproduction steps for bugs
  - Describe problem, rather than your solution
  - Use proper titles, labels, assignees, etc.
  - Applying templates may help

Collaborating with Git (4/4)

- Several popular Git branching strategies
  - Git flow
  - GitHub flow
  - GitLab flow
  - Etc.
- In this course, we will cover Git flow[1] and GitHub flow[2]
- For the term project, you will use GitHub flow

1. A successful Git branching model
2. GitHub Flow

Summary

- Commits are immutable
- Branches are pointers
- Never use `git push --forced`
- Not knowing advanced features of Git is fine…
  - Following rules of commit / branch / PR is more important!

Supplementary Materials

- MIT Missing Semester - Version Control (Git)
- Pro Git (2nd)
- Writing a proper GitHub issue

Thank You.
Any Questions?