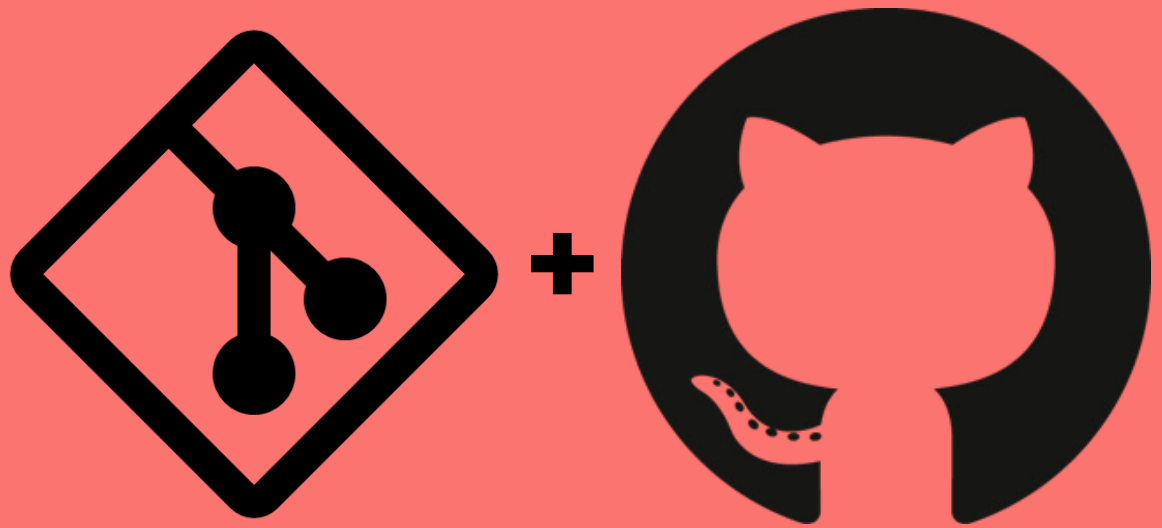
Git

In Practice

Basics



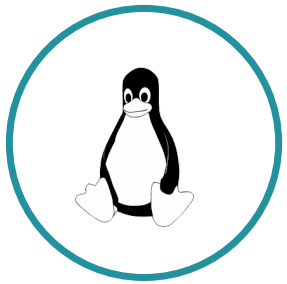
1 Git Fundamentals

2 Git States

3 Git Commits

Git – A Brief History

1992 – 2001



Linux

Software changes to the Linux Kernel were passed around as patches

2002 – 2005

BitKeeper

Linux kernel project began using a proprietary DVCs – BitKeeper

2005

Git

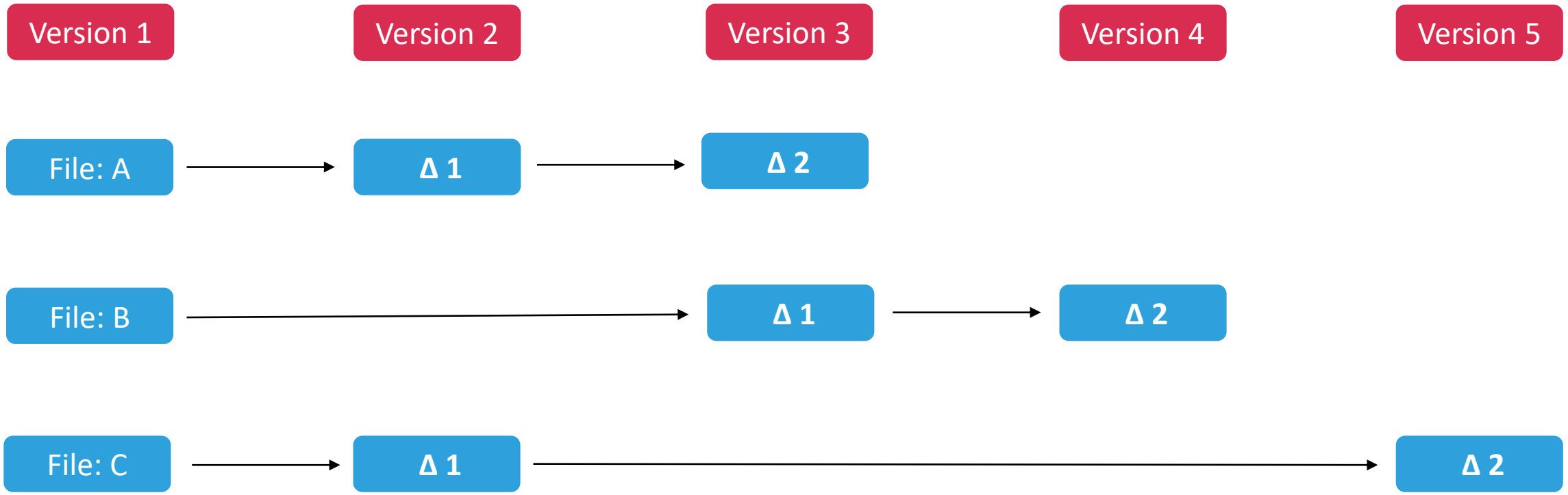
Linus Torvalds writes Git to replace BitKeeper

Goals of Git

- ❑ Fully distributed
- ❑ Simple design which handles large projects efficiently – speed and data size
- ❑ Strong support for non-linear development (thousands of parallel branches)

A Comparison

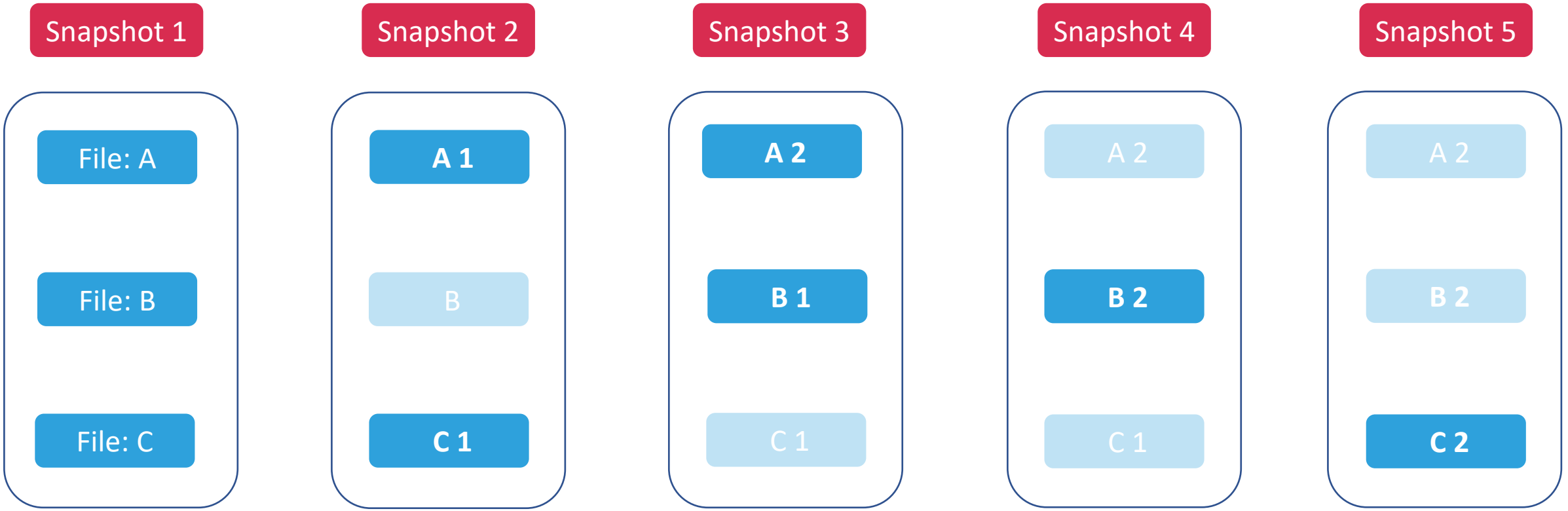
Other Version Control Systems



Delta-based Version Control

A Comparison

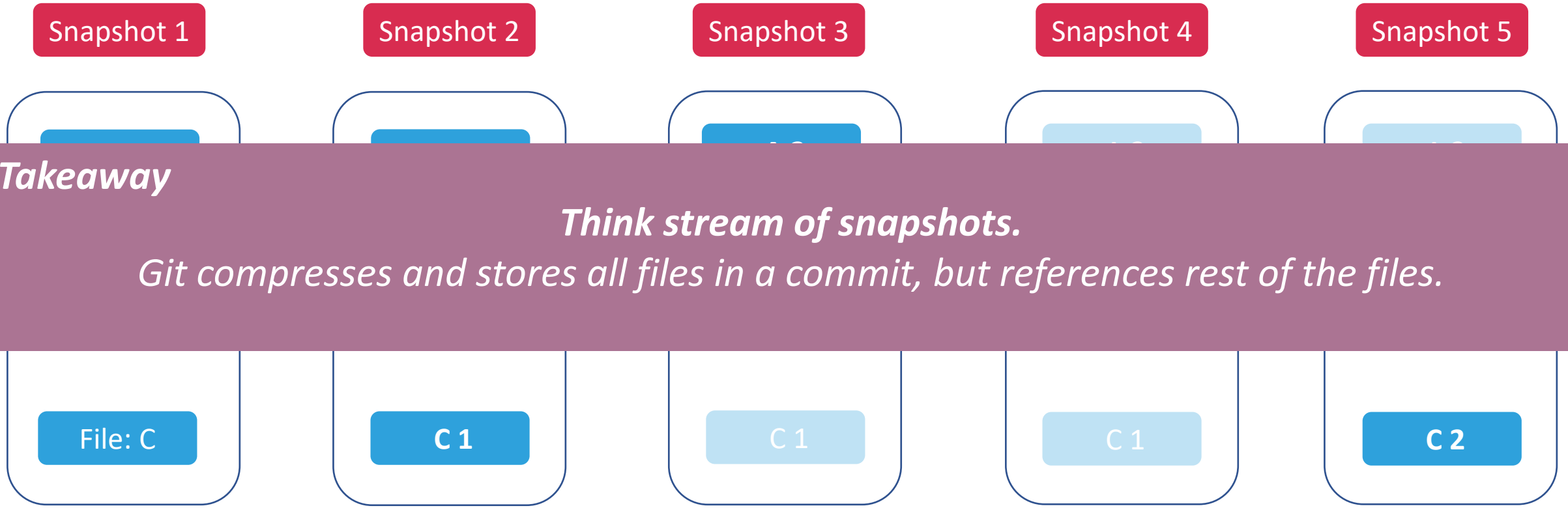
Git



Stream of **Snapshots**

A Comparison

Git



Takeaway

Think stream of snapshots.

Git compresses and stores all files in a commit, but references rest of the files.

Stream of **Snapshots**

Git Installation

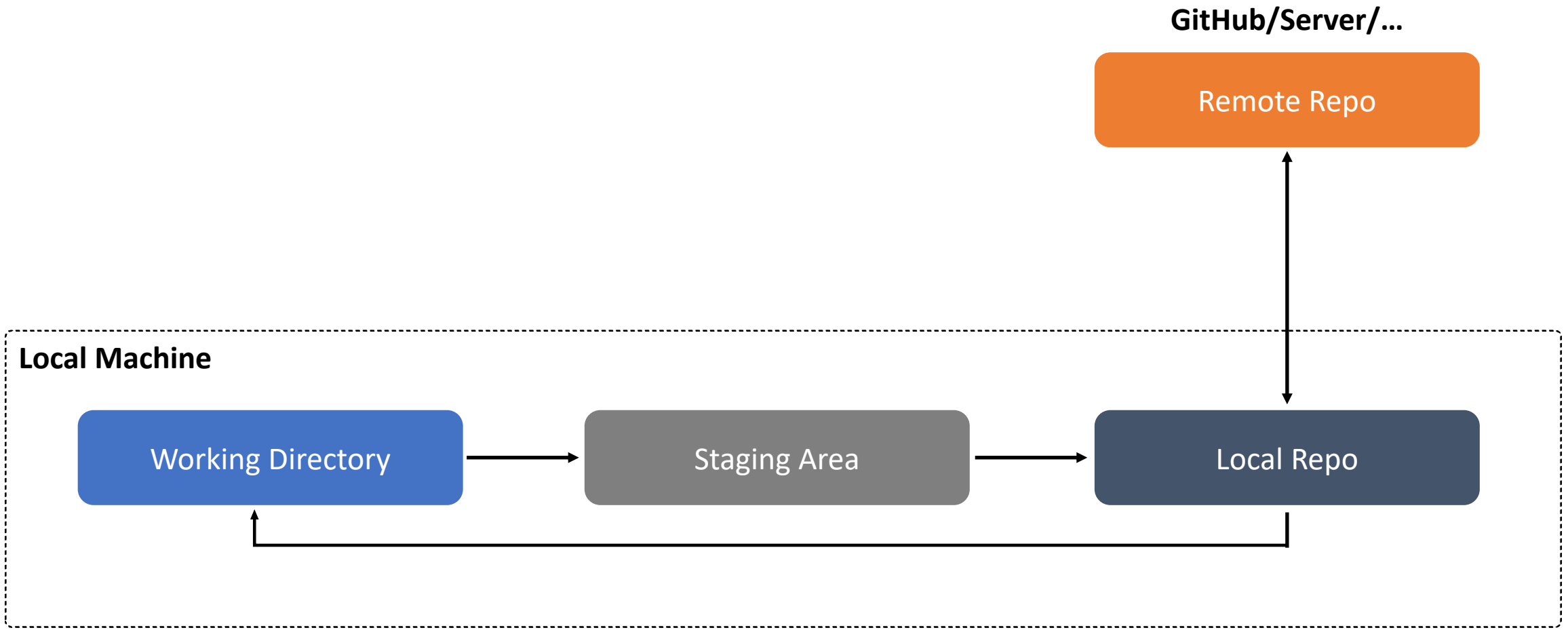
Official – git-scm

https://git-scm.com/downloads

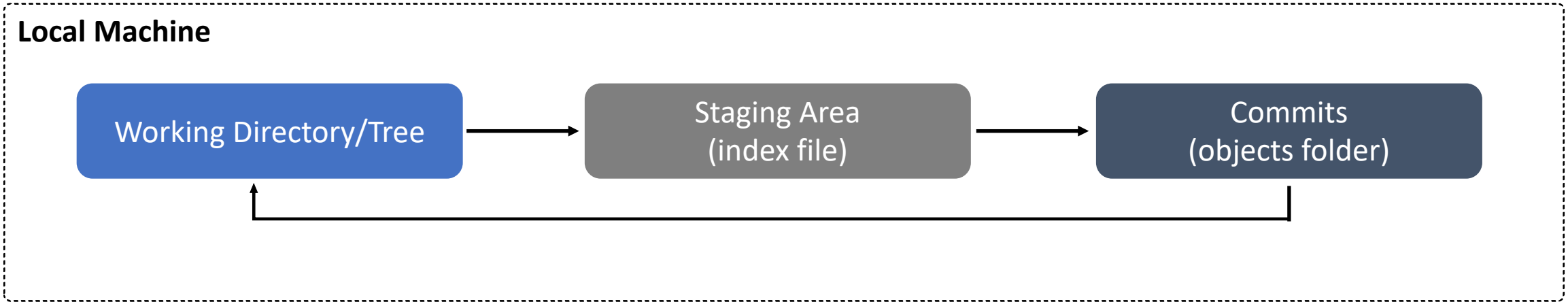
Github

https://github.com/git-guides/install-git

A Distributed Version Control System



Git Project Components



.git/

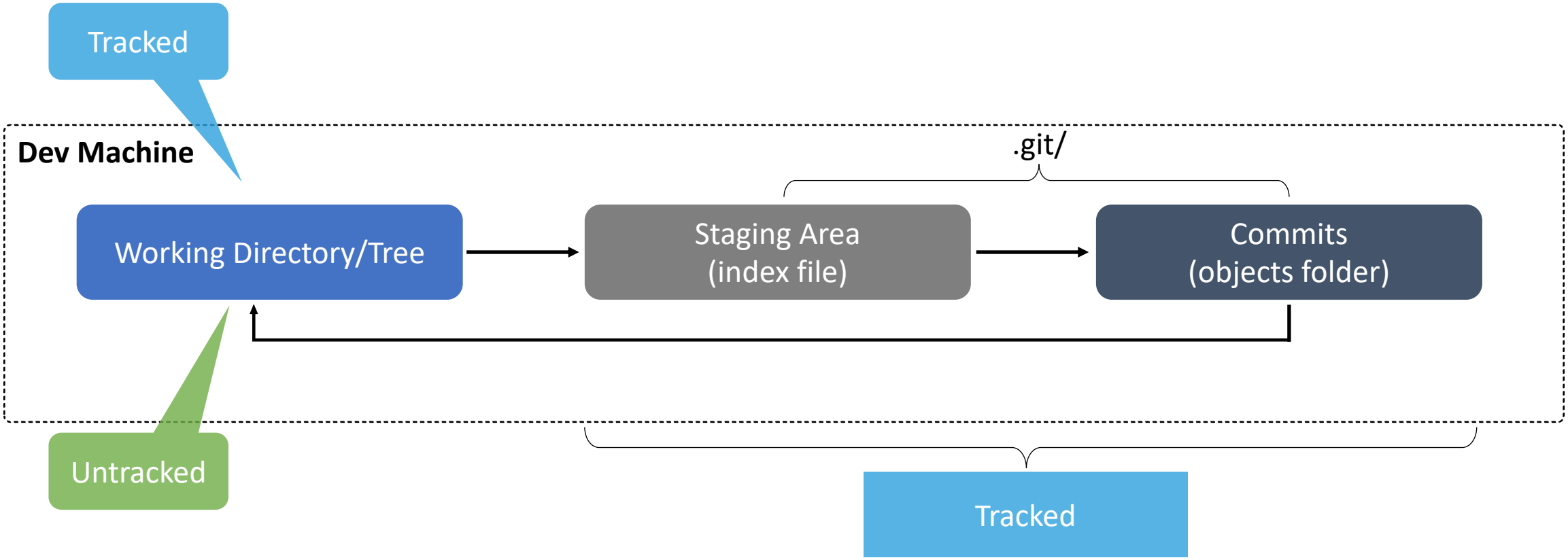
$ git init

Staging file

Commits DB

```
→ .git (main)$ tree
.
├── COMMIT_EDITMSG
├── HEAD
├── ORIG_HEAD
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── fsmonitor-watchman.sa
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-merge-commit.samp
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   ├── prepare-commit-msg.sa
│   ├── push-to-checkout.samp
│   └── update.sample
├── index
├── info
│   ├── exclude
│   └── refs
├── logs
│   ├── HEAD
│   └── refs
│       └── heads
│           ├── develop
│           └── main
├── objects
│   ├── e6
│   │   └── 9de29bb2d1d6434b8
│   ├── info
│   │   ├── commit-graph
│   │   └── packs
│   └── pack
│       ├── pack-04da0be5b947
│       └── pack-04da0be5b947
├── packed-refs
└── refs
    ├── heads
    │   └── main
    └── tags
```

Git Project Components




```
$ ls
```

```
$ git ls-files
```

```
$ git log
```

Git States

Untracked (New) – A new untracked change (file) exists in the working directory

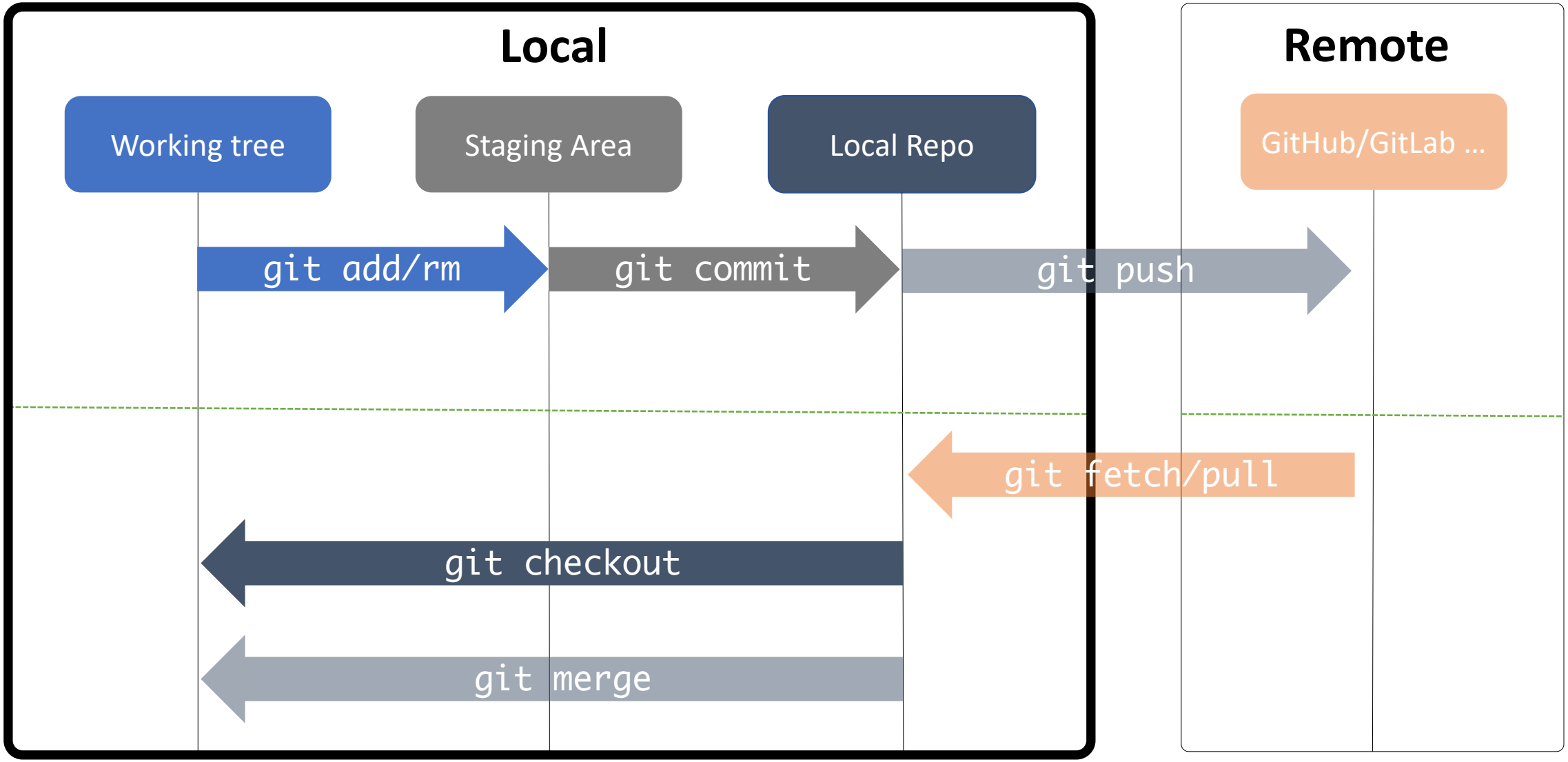
Modified – A tracked file is modified in your working directory

Staged – Changes are tracked and ready to be committed

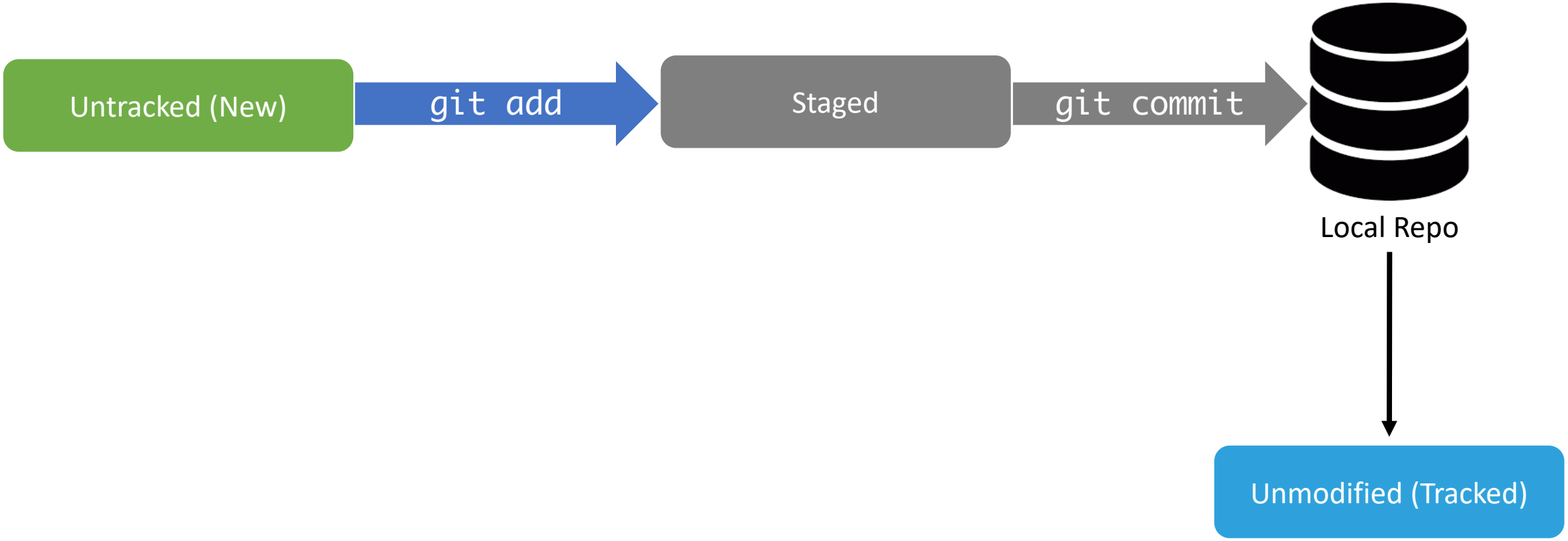
Local Repo – Changes are committed and stored in the Object Database

Unmodified (Tracked) – The working directory is clean – nothing to commit

Project Components – Commands



Scenario 1 – Add Changes



What is a Checksum?

- A checksum is the value returned from a *one-way hash algorithm*

- Checksums can be used to validate the integrity of the data, because
  - Modifying the data in any way will change the hash value – checksum

What is a Checksum?

- Git uses SHA–1 (Simple Hashing Algorithm) which generates a 40-char hash

```
3c2571e28d1269ed545572262084c13176271ce2
```

```
$ git hash-object first.txt
```

Git and Checksum

- Git checksums include meta data about the *commit* including
  - Commit message
  - Committer
  - Commit date
  - Author
  - Author create date
  - Working Directory/tree hash
  - The previous *commit's* (parent) hash

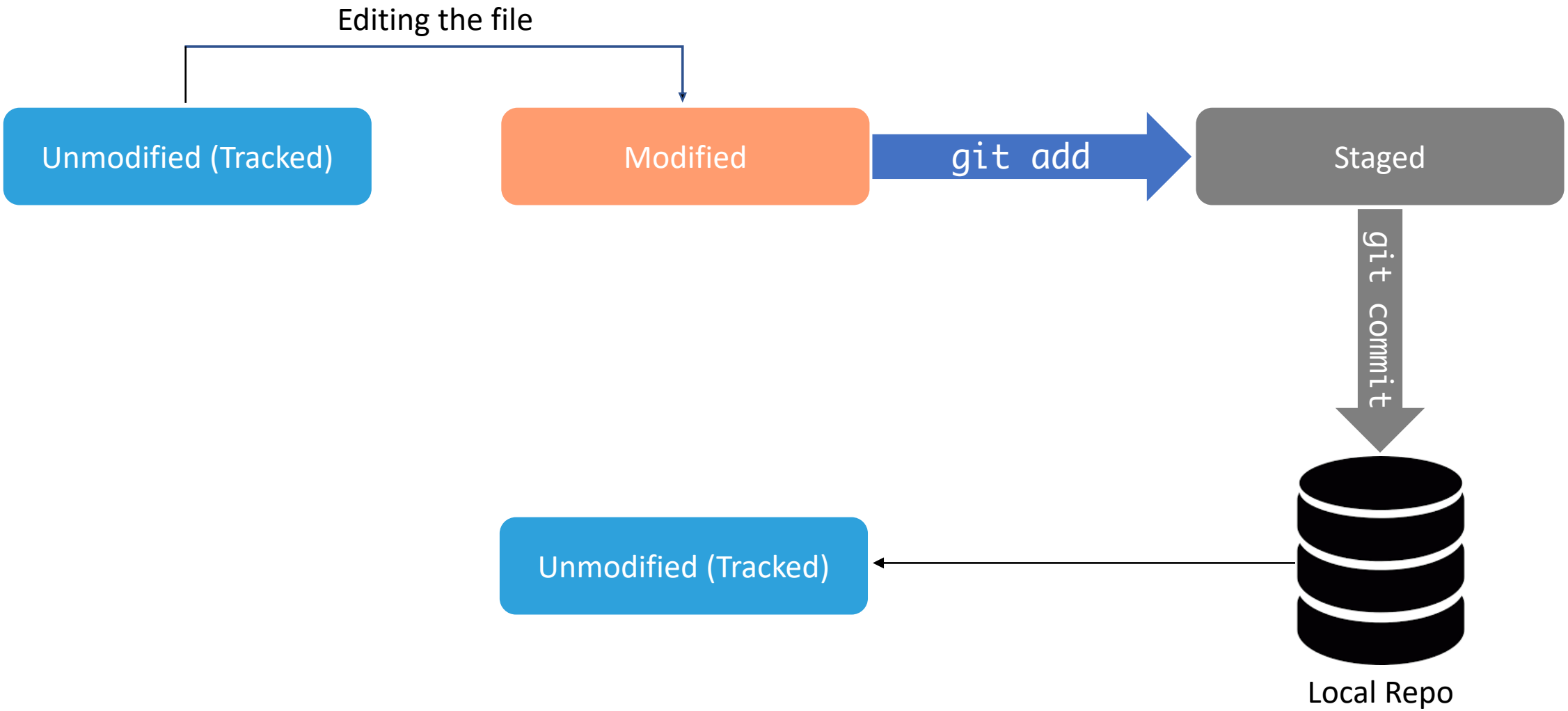
Git and Checksum

- Git assures the integrity of the stored data by using checksums as identifiers

- This way, Git also ensures historical chain of commits cannot be edited

Let us practice

Scenario 2 – Edit



Let us practice

Scenario 3 – Remove Changes – TODO split it

Action	Command	Result
To delete a tracked file	`git rm filename`	Removes files from working directory and adds a delete marker to staging area
To remove the file from the staging area	`git rm --cached filename`	Removes the file from the staging area

If a tracked file was deleted from the working directory	`git rm filename`
	OR
	`git add filename`
		Adds a delete marker to the staging area

Let us practice

Scenario 4 – Undo Working Tree

Unstaged

Modified

Discard current changes

Working Tree

index.html Tracked

...

...

```
$ git checkout index.html
```


Working Tree

index.html Tracked

...

...

`git checkout –` discards changes in the working tree and restores the file to the state of the *previous* commit

Scenario 4 – Undo Working Tree

Unstaged

`git checkout –` discards changes in the working tree and restores the file to the state of the *previous* commit

```
$ # To undo changes to all files
$ git checkout .
```

Let us practice

Scenario 5 – Undo Staging Area

Staged

- Unstaging staged changes is a two-step process
  - Step 1 – Unstage file from the staging area
  - Step 2 – Discard changes from the working tree

Scenario 5 – Undo Staging Area

Staged

- New command in git v2.23+
- Two ways to unstage and discard changes
  - `git restore --staged` with `git restore`
  - `git reset` with `git checkout`

New

Scenario 5 – Undo Staging Area – 1 of 2

Staged

Using restore

Case 1

Newly added

Unstages from staging area

Staging Area

index.html Tracked

…

…

2.23

```
$ git restore --staged index.html
```

Staging Area

…

…

…

`git restore --staged` – removes newly staged files from the staging area and the file becomes untracked in the working tree

Scenario 5 – Undo Staging Area – 1 of 2

Staged

Using restore

Case 2

Modified

Unstages from staging area

Staging Area

index.html Tracked

...

...

Staging Area

index.html Tracked

...

...

2.23

```
$ git restore --staged index.html
```

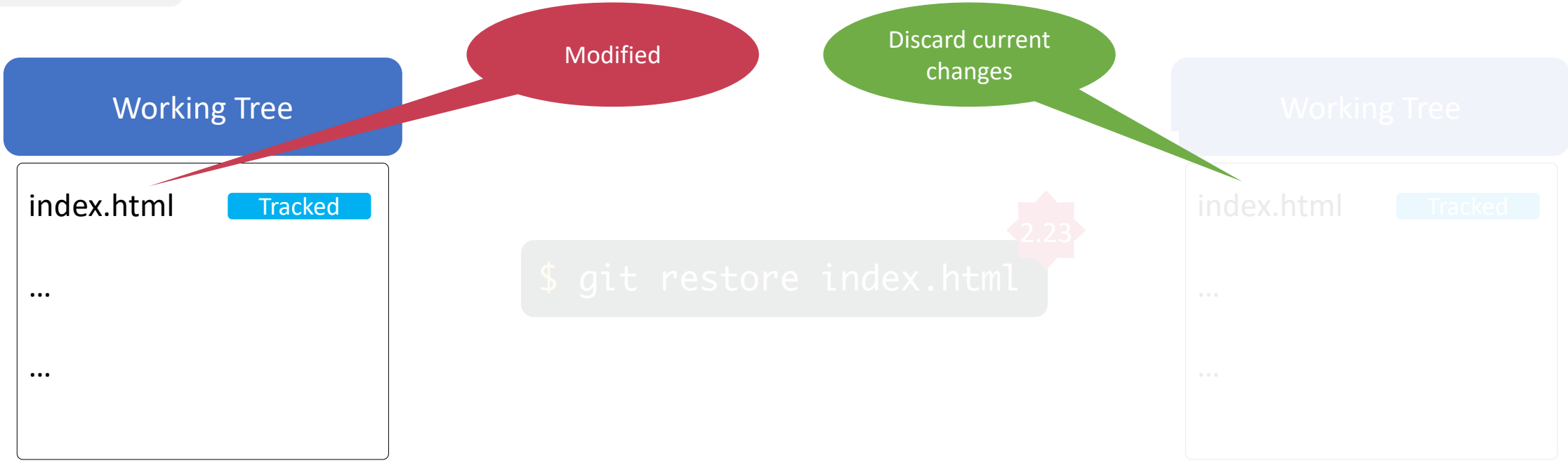
`git restore --staged` – unstages the recent git add/rm of already tracked files from the staging area.

An indication that staging area contains more than one copy of the same file

Scenario 5 – Undo Working Tree – 2 of 2

Staged

Using restore



2.23

```
$ git restore index.html
```

`git restore` – discards changes in the working tree
and restores the file to the state of the *previous* commit

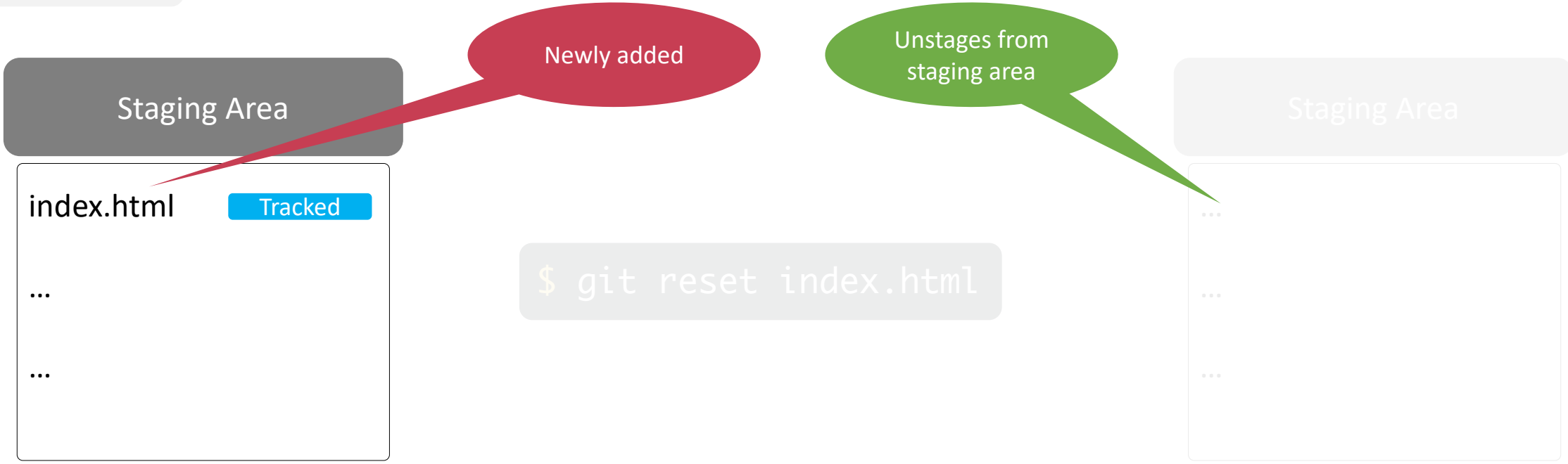
Let us practice

Scenario 5 – Undo Staging Area – 1 of 2

Staged

Using reset

Case 1



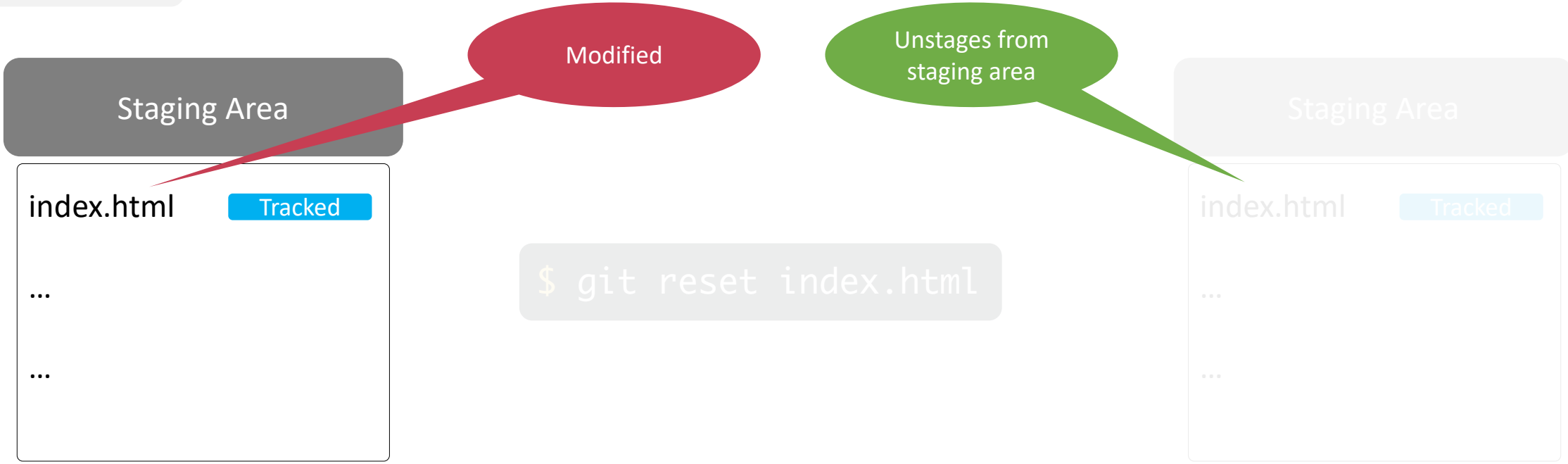
`git reset` – removes newly staged files from the staging area

Scenario 5 – Undo Staging Area – 1 of 2

Staged

Using reset

Case 2



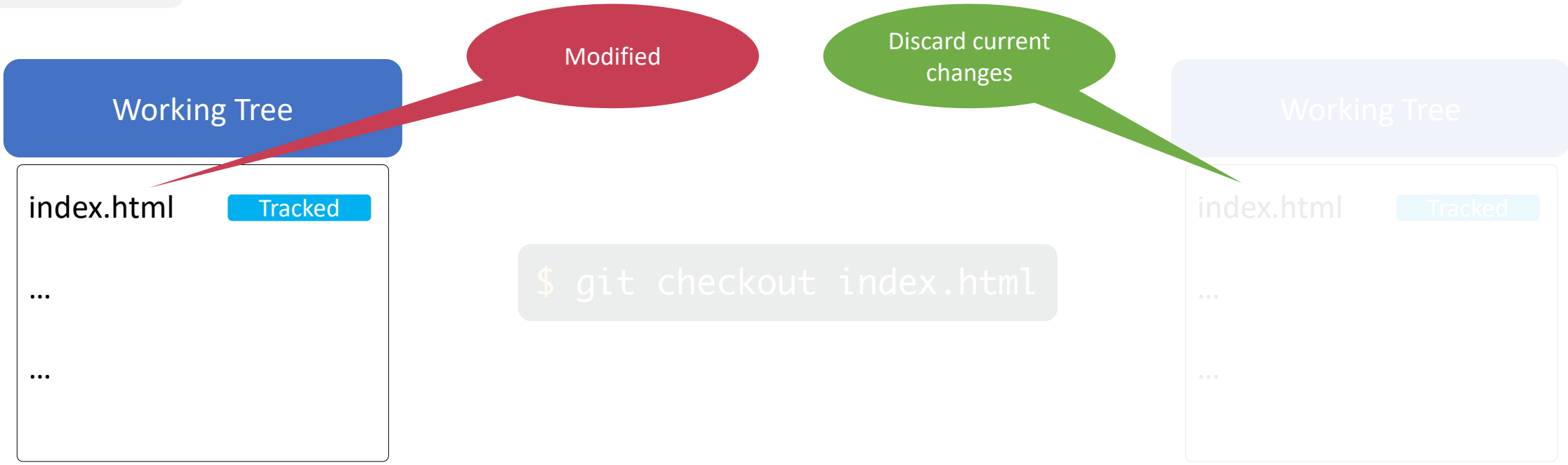
`git reset` – unstages the recent git add/rm of already tracked files from the staging area.

An indication that staging area contains more than one copy of the same file

Scenario 5 – Undo Working Tree – 2 of 2

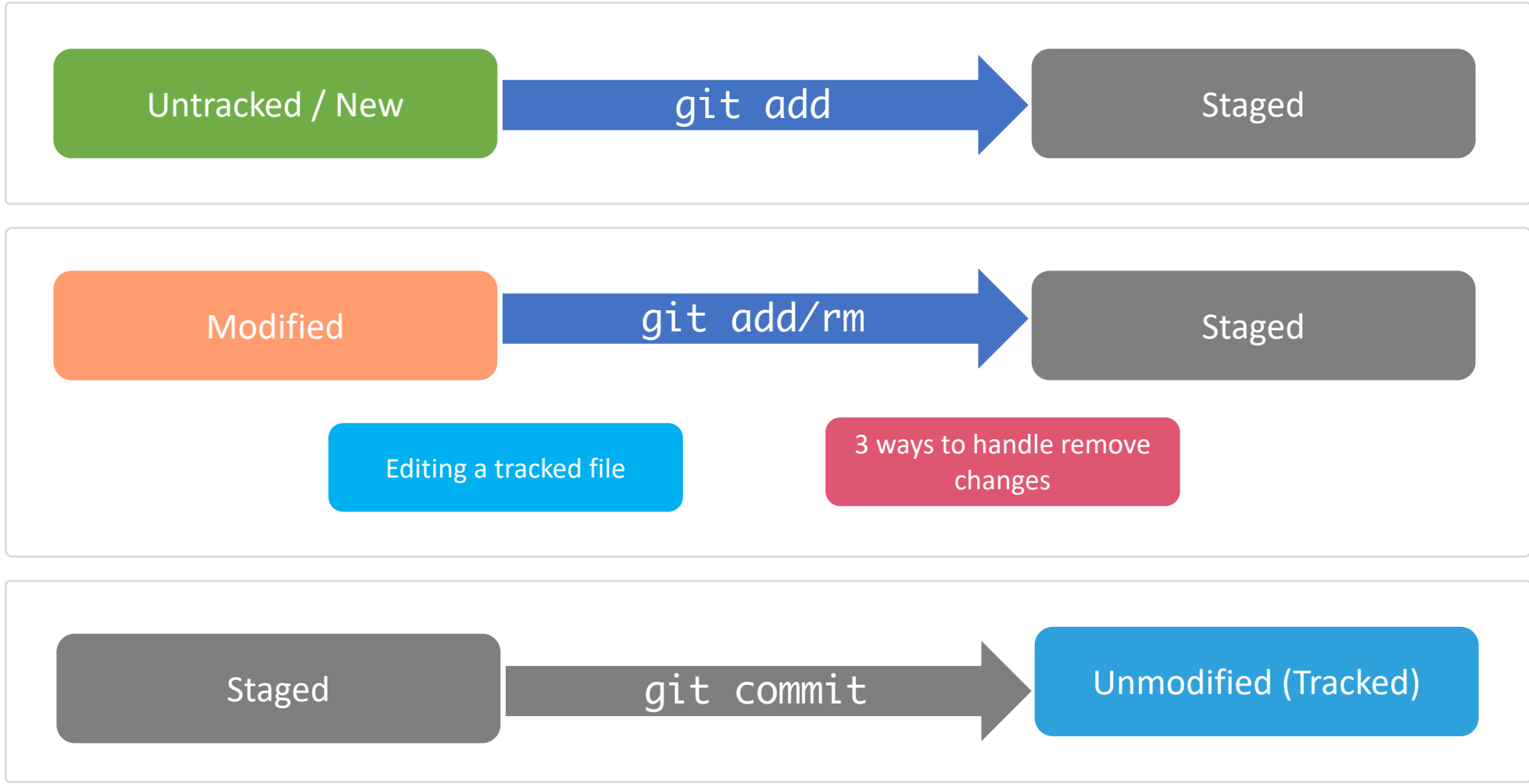
Staged

Using checkout

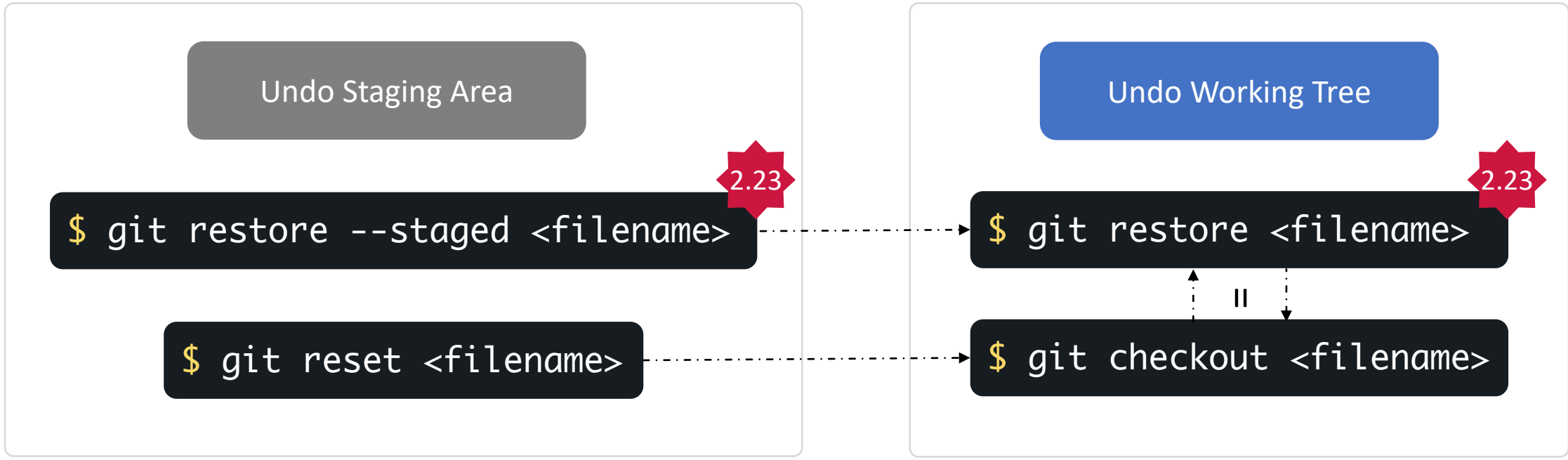


`git checkout` – discards changes in the working tree and restores the file to the state of the *previous* commit

Scenarios – Summary



Scenarios – Summary



- Unstaging staged changes is a two-step process
  - Use `git restore --staged` with `git restore`
  - Use `git reset` with `git checkout`

Takeaways

- ✓ Stream **of snapshots**
  - Git doesn’t store data as a series of changesets or differences
  - Instead, it stores data as a series of *snapshots*
  - You can consider each commit as a “mini file system”

Talk about the three git states and trees

- ✓ Stream **of snapshots**
  - Git doesn’t store data as a series of changesets or differences
  - Instead, it stores data as a series of *snapshots*
  - You can consider each commit as a “mini file system”

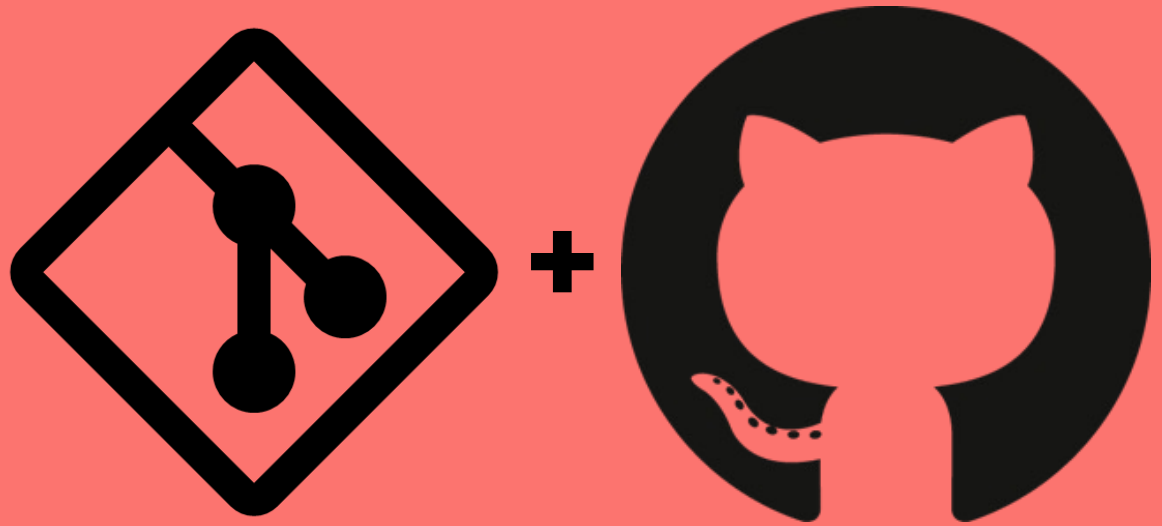
Takeaways

- ✓ Commands
    a. `git init`
    b. `git status`
    c. `git add`
    d. `git rm` and `git rm --cached`
    e. `git ls-files`
    f. `git checkout`
    g. `git commit`
    h. `git log`
    i. `git restore --staged` with `git restore`
    j. `git reset` with `git checkout`

Git

In Practice

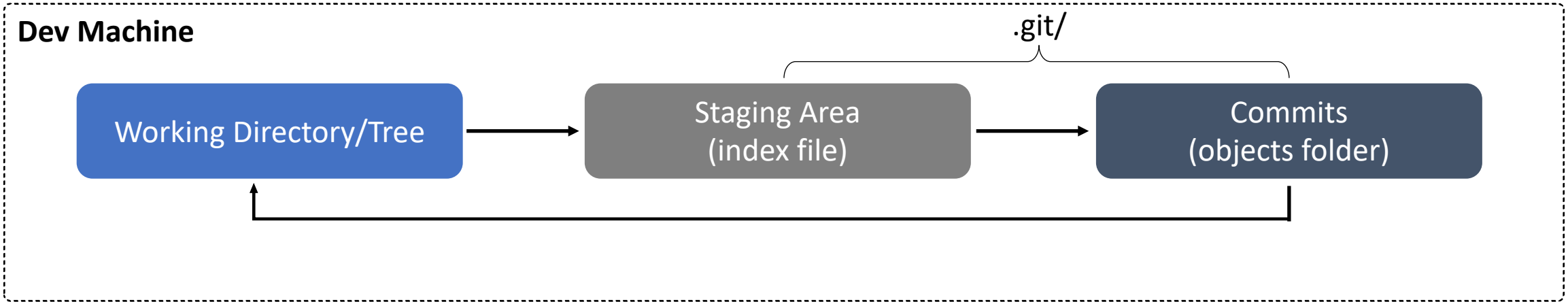
Basics



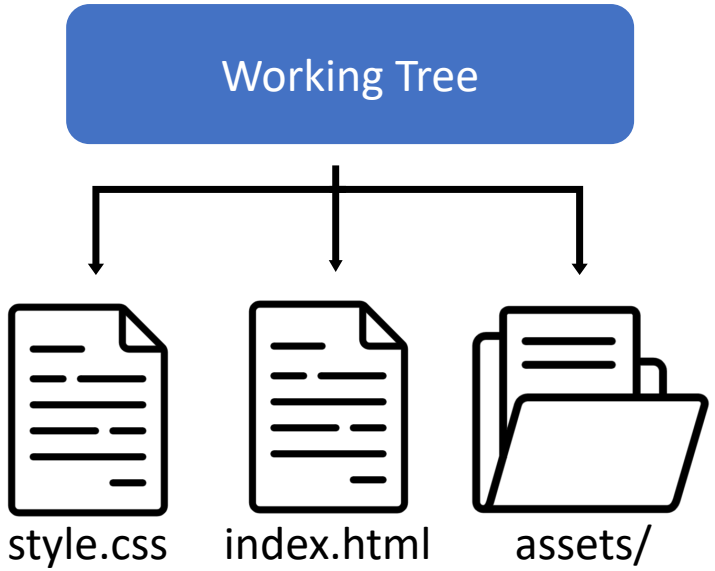
1 Git Branches

2 Remote Branches

The Three Trees In Git



Trees in Git



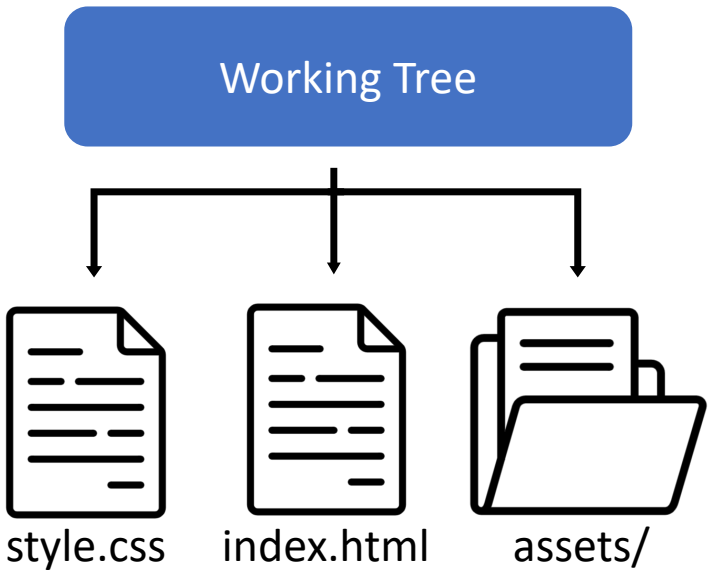
Staging Index (Tree)

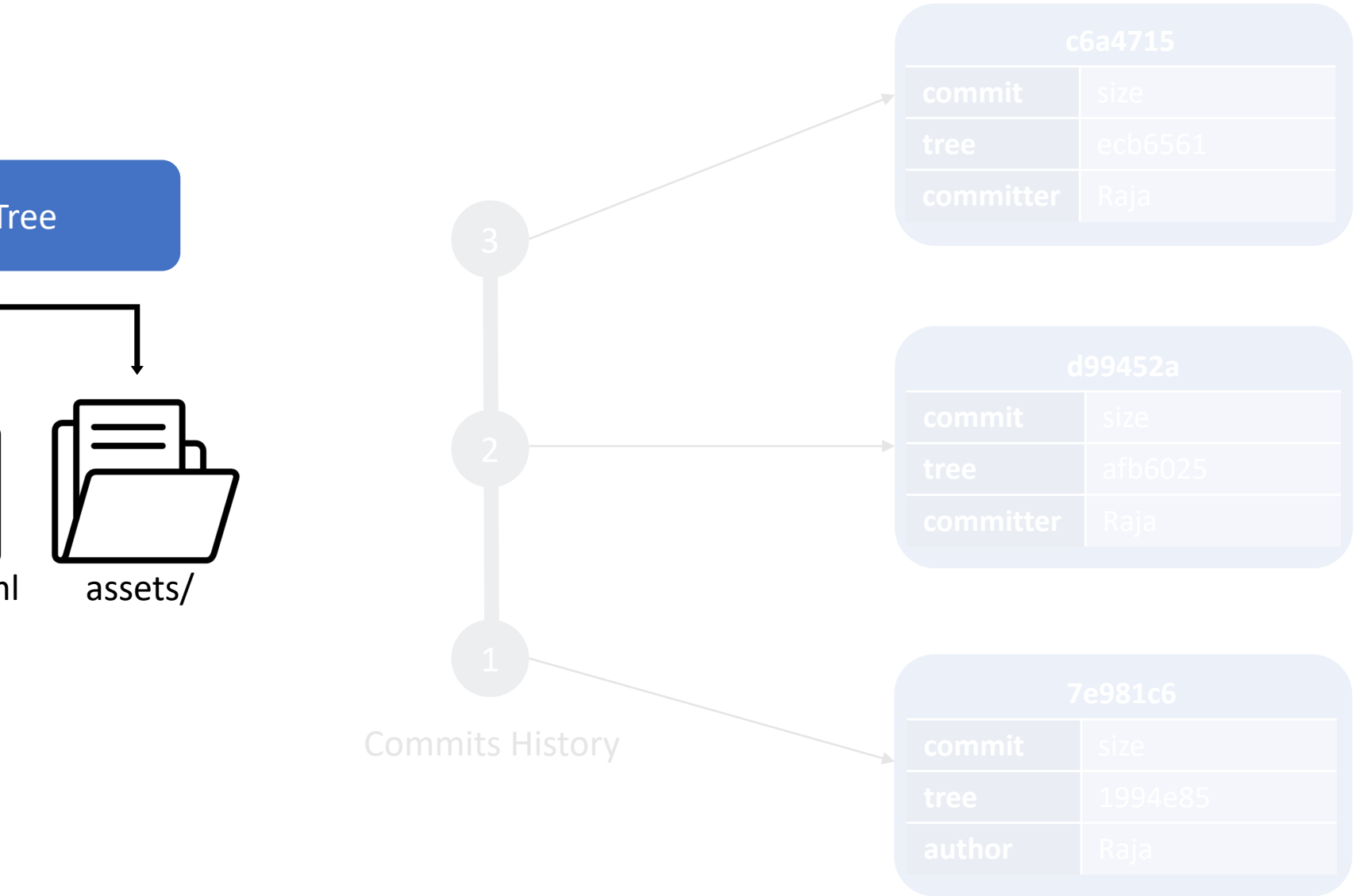
1. Staging Index tracks the Working Directory changes, that have been promoted with `git add`
2. This tree is a complex internal caching mechanism
3. Git generally tries to hide the implementation details of the Staging Index from the user

Staging Index (Tree)

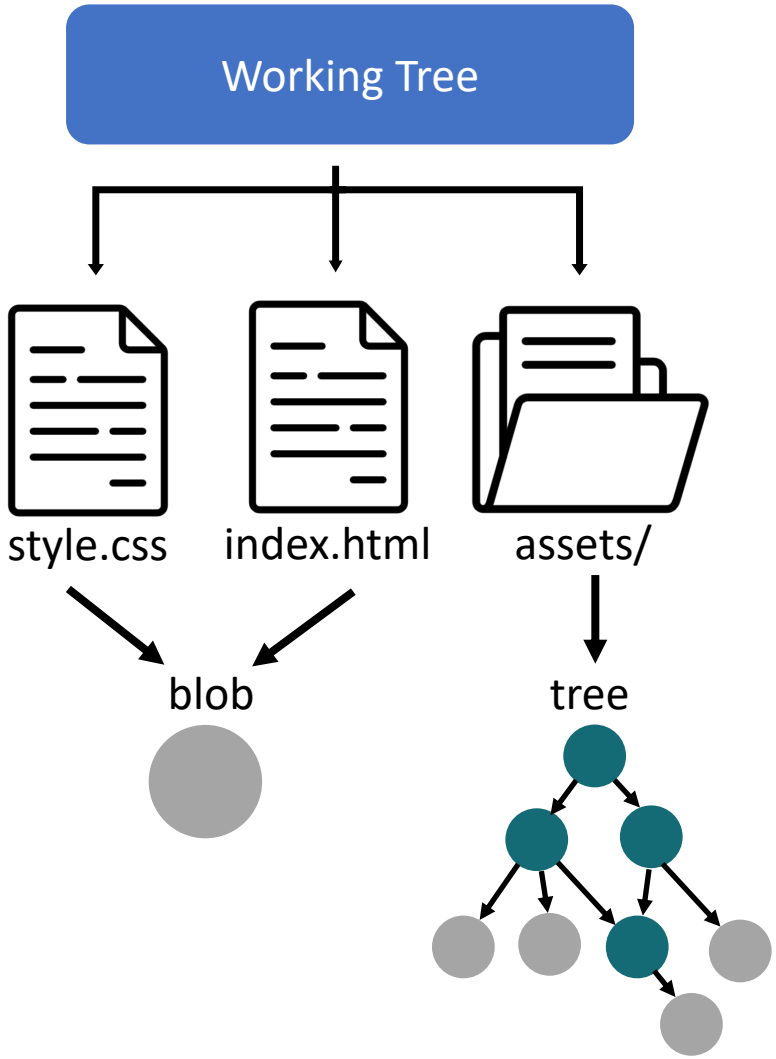
1. `git ls-files` output without the `-s` lists file names and paths that are currently part of the index
2. `-s` lists more metadata about the file such as mode bits, object name, stage number and file name

Commit Tree


Working Tree
style.css
index.html
assets/

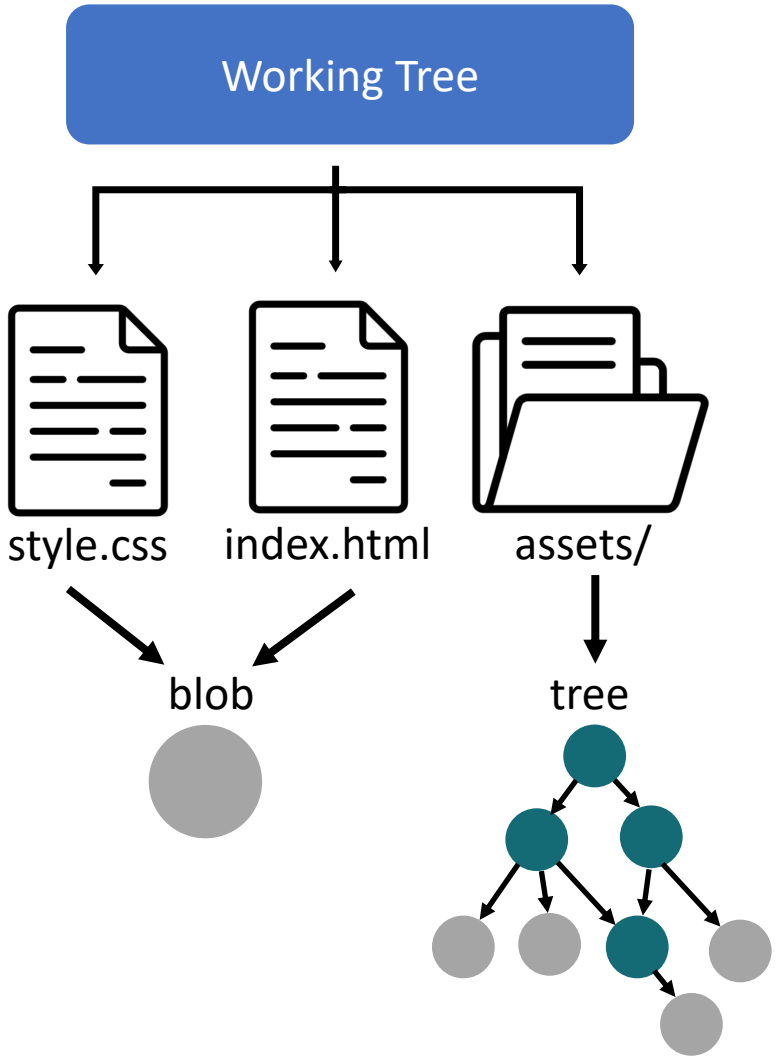

3
2
1
Commits History
c6a4715
commit
size
tree
ecb6561
committer
Raja
d99452a
commit
size
tree
afb6025
committer
Raja
7e981c6
commit
size
tree
1994e85
author
Raja

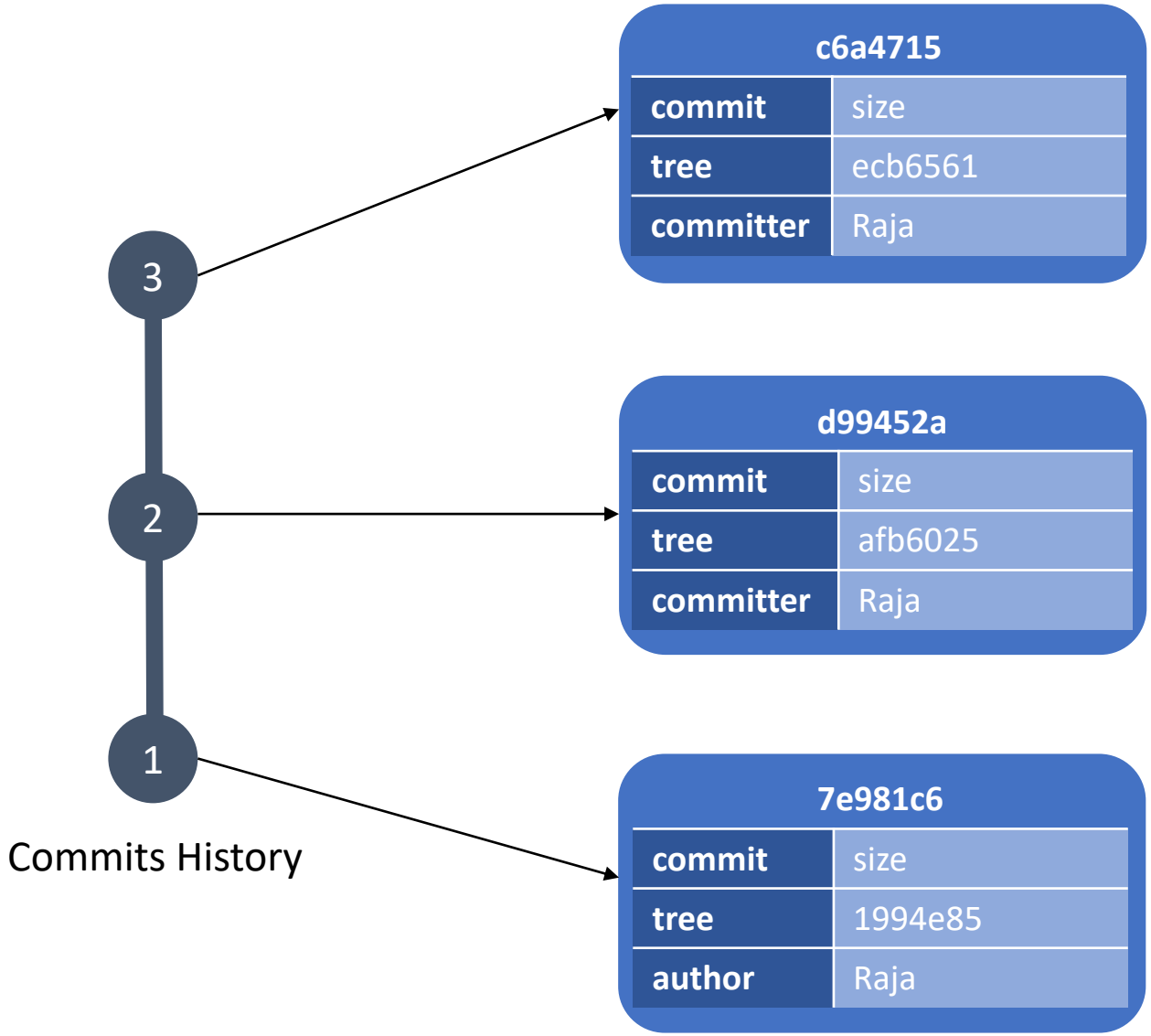
Commit Tree


Working Tree
style.css
index.html
assets/
blob
tree

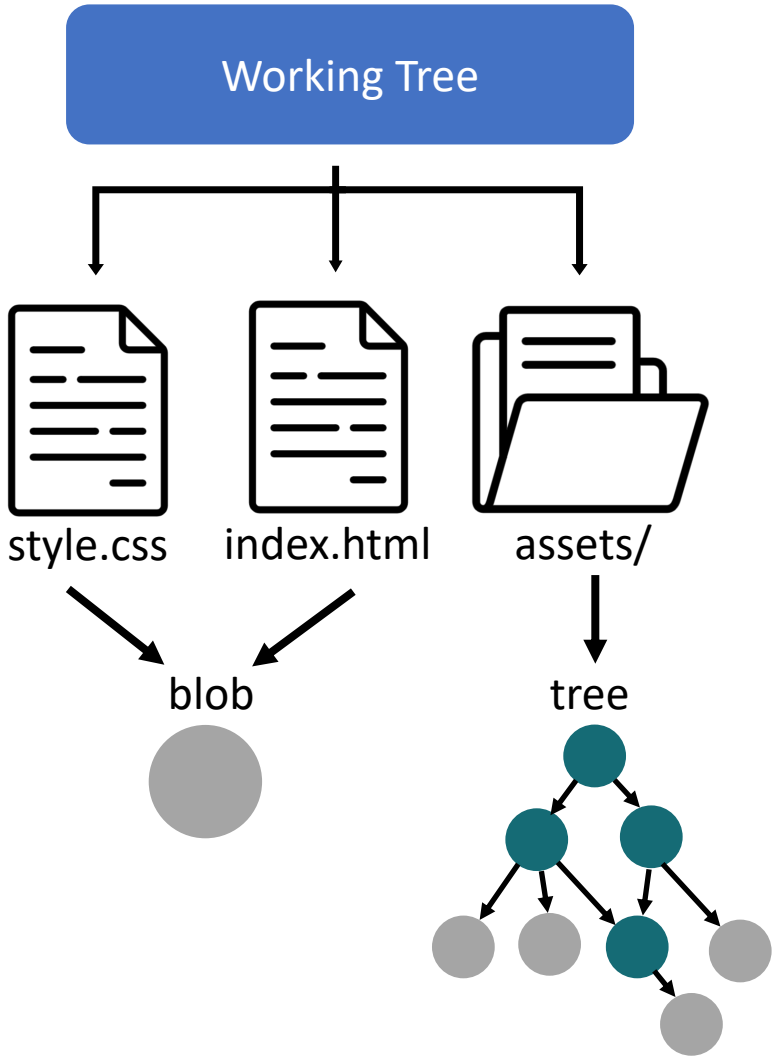
3
2
1
Commits History
c6a4715
commit
size
tree
ecb6561
committer
Raja
d99452a
commit
size
tree
afb6025
committer
Raja
7e981c6
commit
size
tree
1994e85
author
Raja

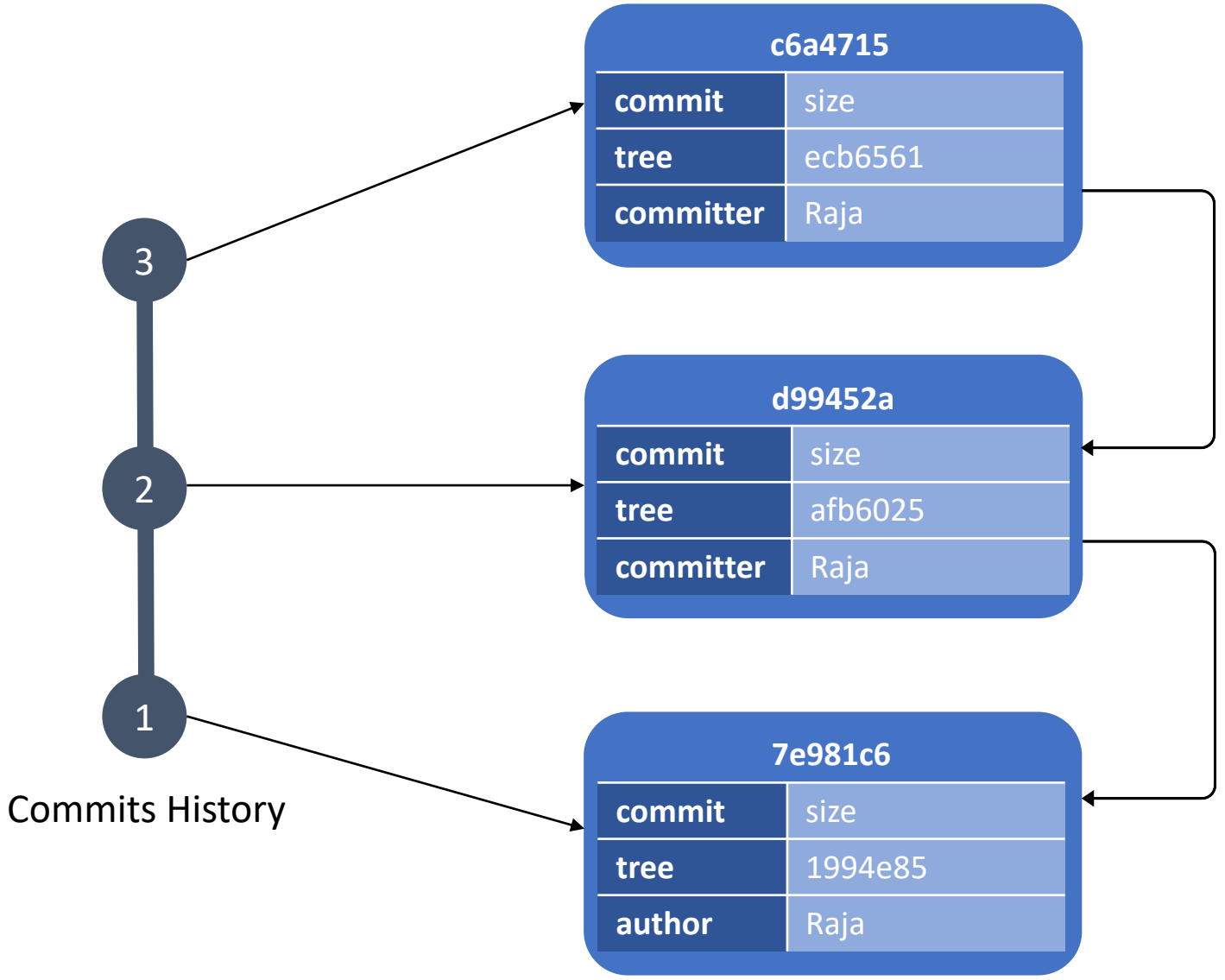
Commit Tree


Working Tree
style.css
index.html
assets/
blob
tree

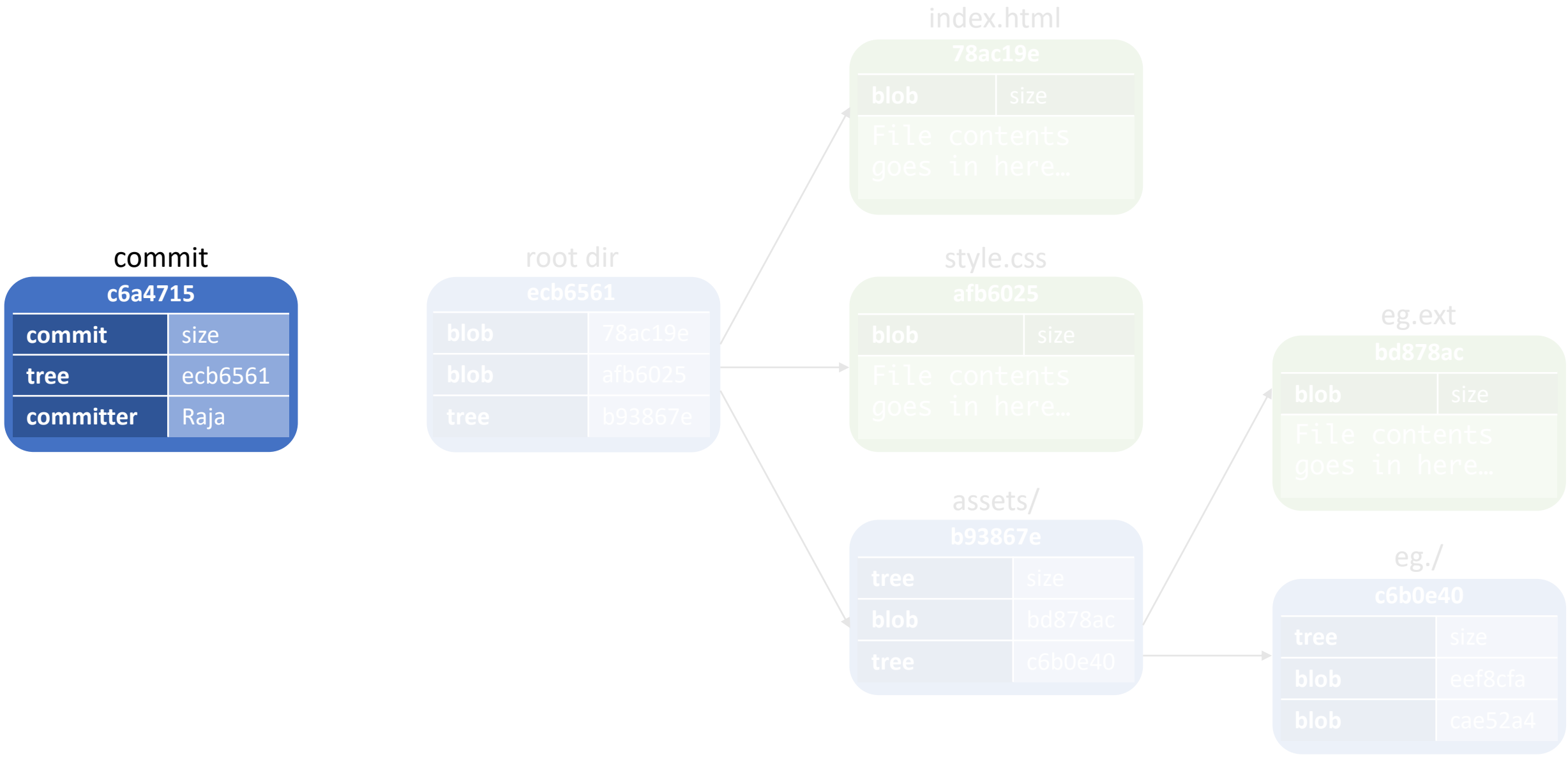

3
2
1
Commits History
c6a4715
commit
size
tree
ecb6561
committer
Raja
d99452a
commit
size
tree
afb6025
committer
Raja
7e981c6
commit
size
tree
1994e85
author
Raja

Commit Tree

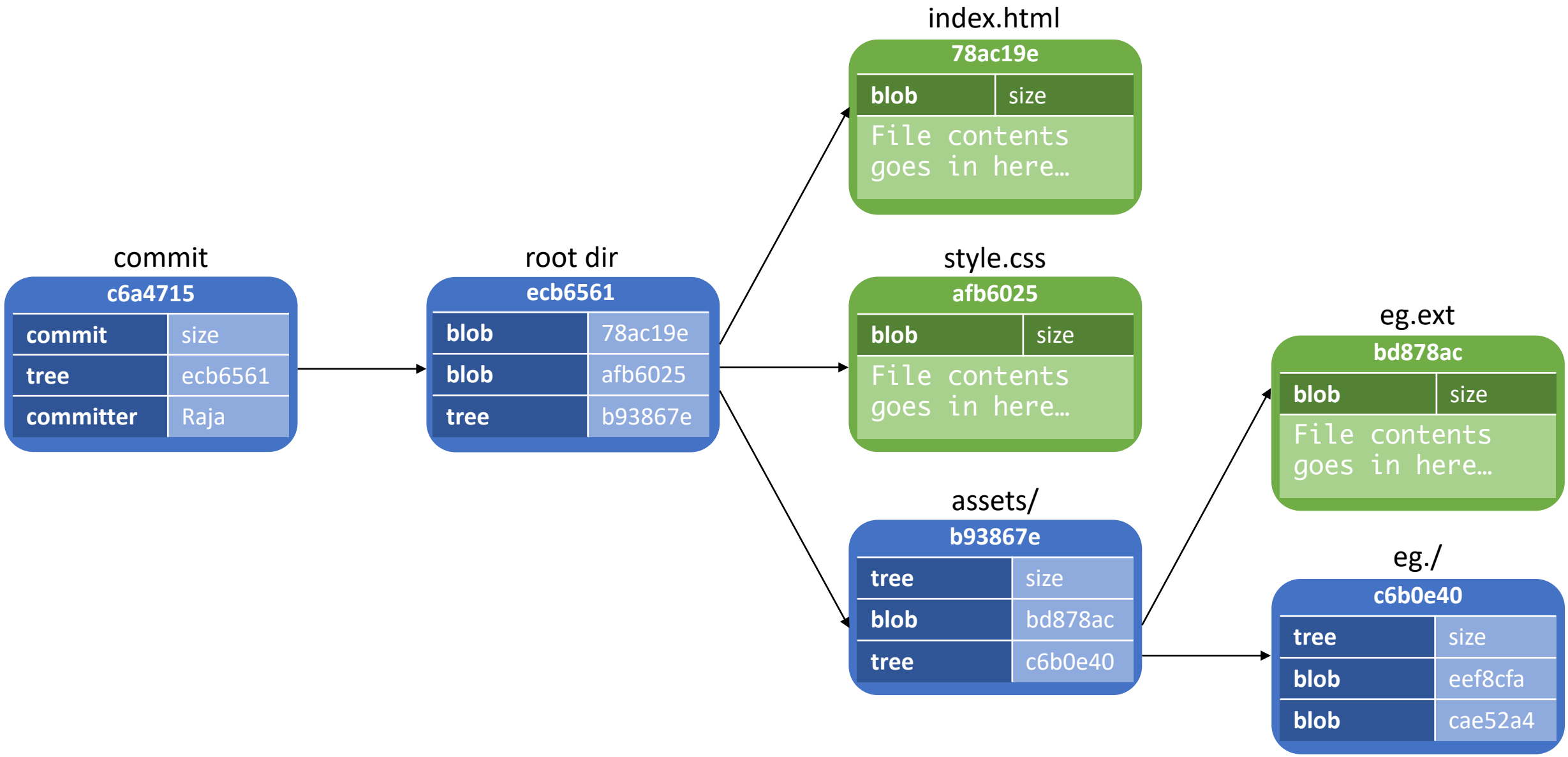

Working Tree
style.css
index.html
assets/
blob
tree


3
2
1
Commits History
c6a4715
commit
size
tree
ecb6561
committer
Raja
d99452a
commit
size
tree
afb6025
committer
Raja
7e981c6
commit
size
tree
1994e85
author
Raja

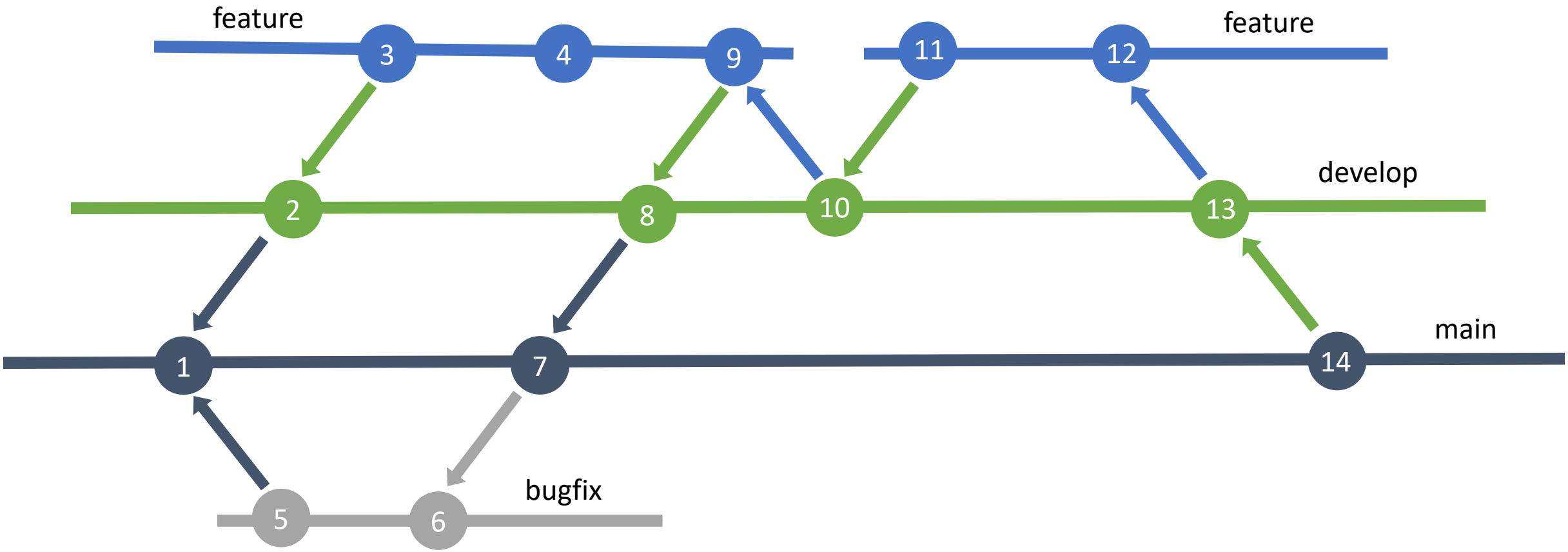
Commit Tree


commit
c6a4715
commit
size
tree
ecb6561
committer
Raja
root dir
ecb6561
blob
78ac19e
blob
afb6025
tree
b93867e
index.html
78ac19e
blob
size
File contents goes in here…
style.css
afb6025
blob
size
File contents goes in here…
assets/
b93867e
tree
size
blob
bd878ac
tree
c6b0e40
eg.ext
bd878ac
blob
size
File contents goes in here…
eg./
c6b0e40
tree
size
blob
eef8cfa
blob
cae52a4

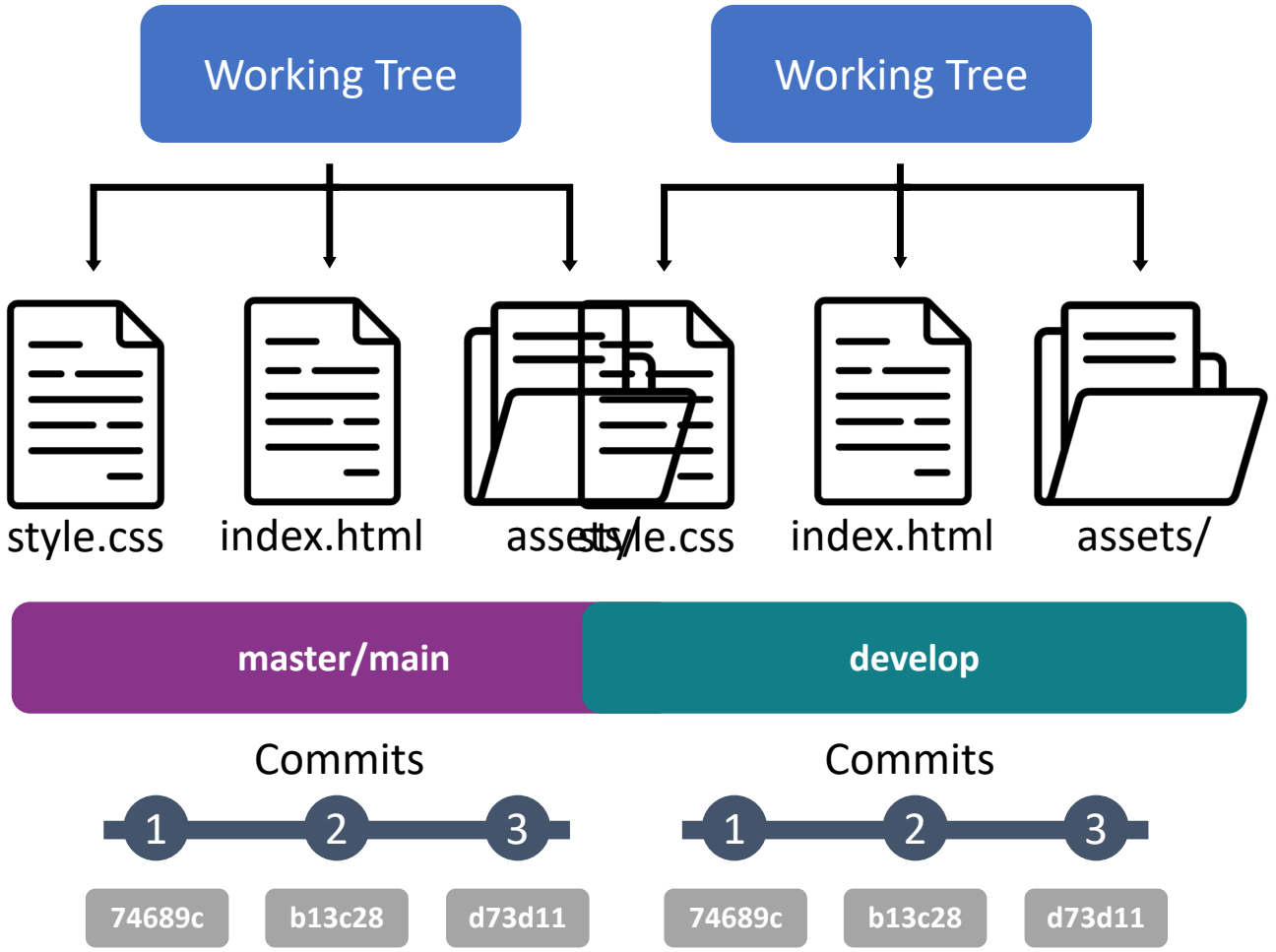
Commit Tree


index.html
78ac19e
blob
size
File contents goes in here…
commit
c6a4715
commit
size
tree
ecb6561
committer
Raja
root dir
ecb6561
blob
78ac19e
blob
afb6025
tree
b93867e
style.css
afb6025
blob
size
File contents goes in here…
eg.ext
bd878ac
blob
size
File contents goes in here…
assets/
b93867e
tree
size
blob
bd878ac
tree
c6b0e40
eg./
c6b0e40
tree
size
blob
eef8cfa
blob
cae52a4

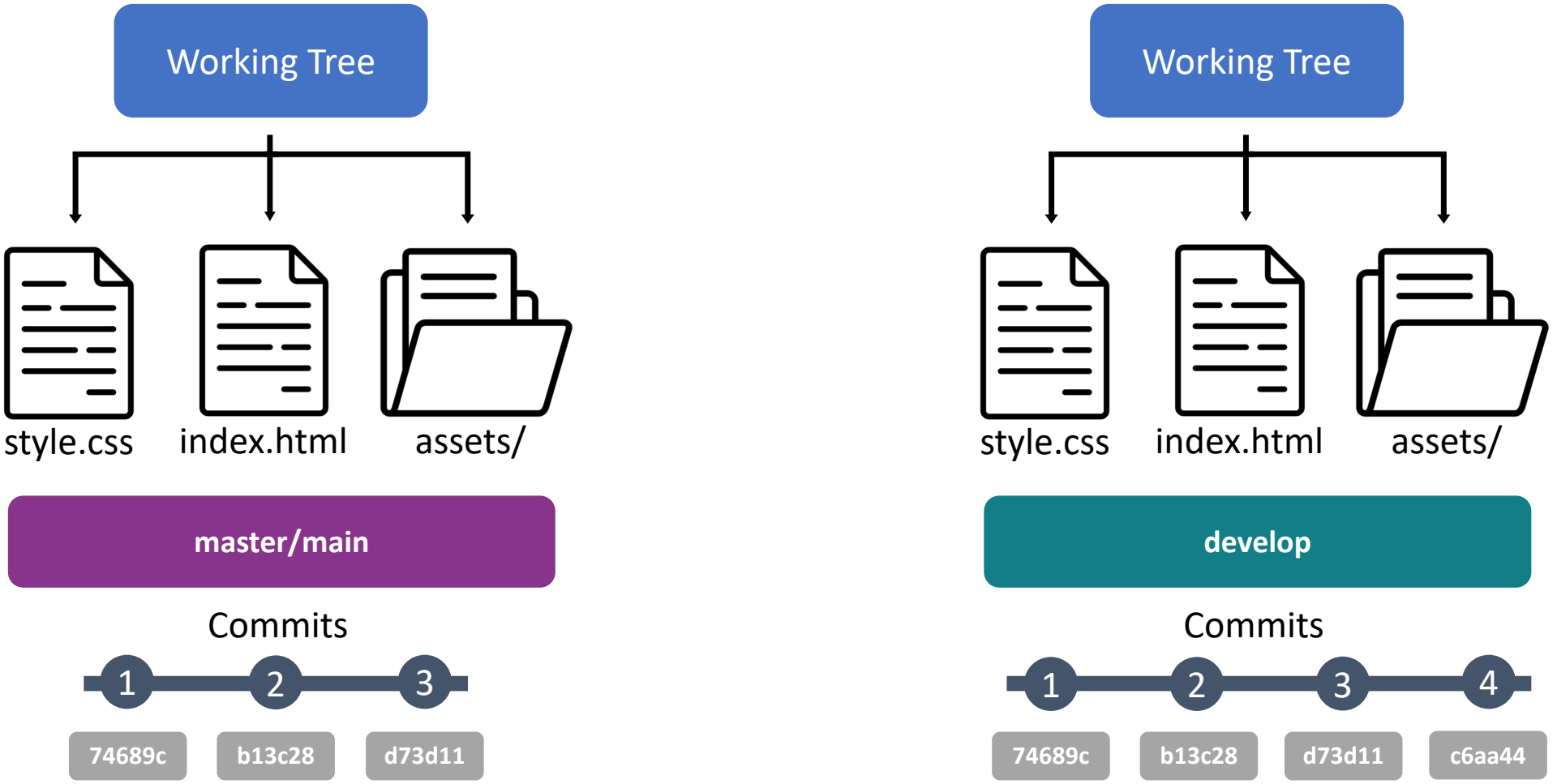
A Typical Branched Tree



What Is A Branch?



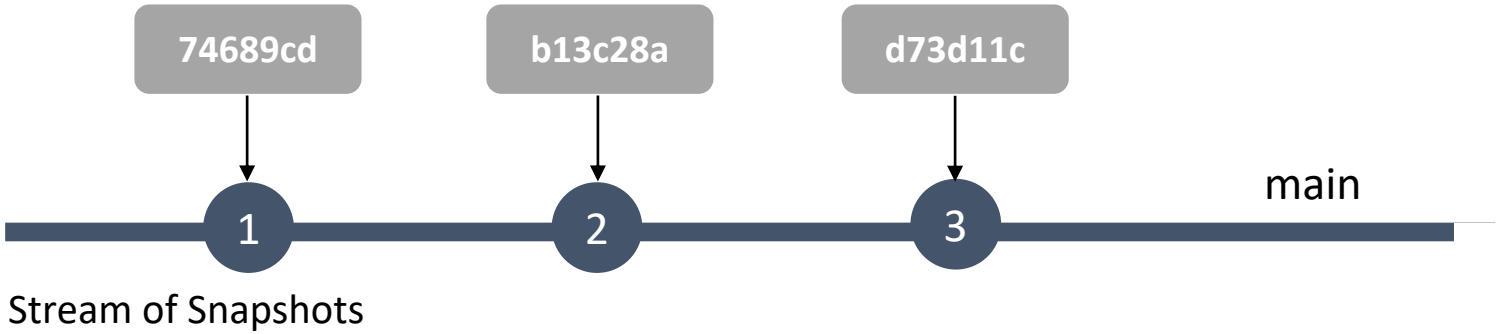
What Is A Branch?



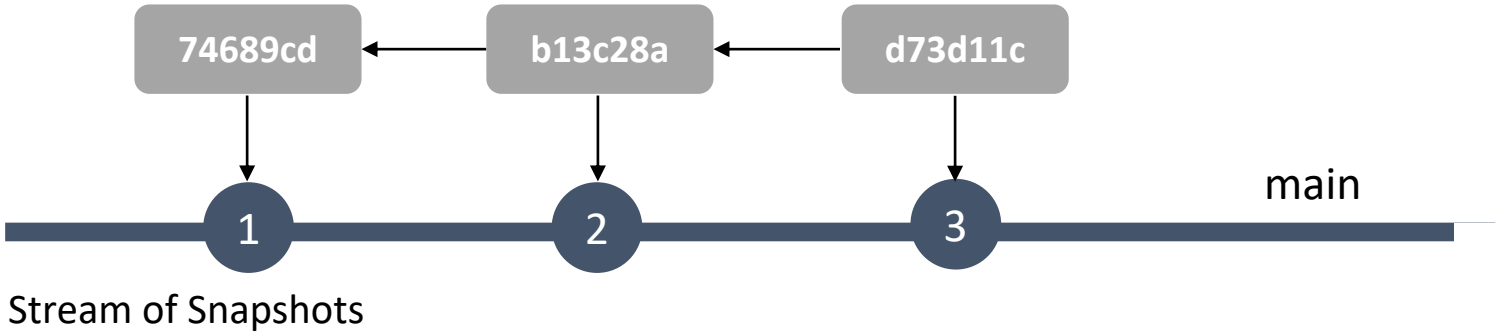
A Git Branch

A Git Branch

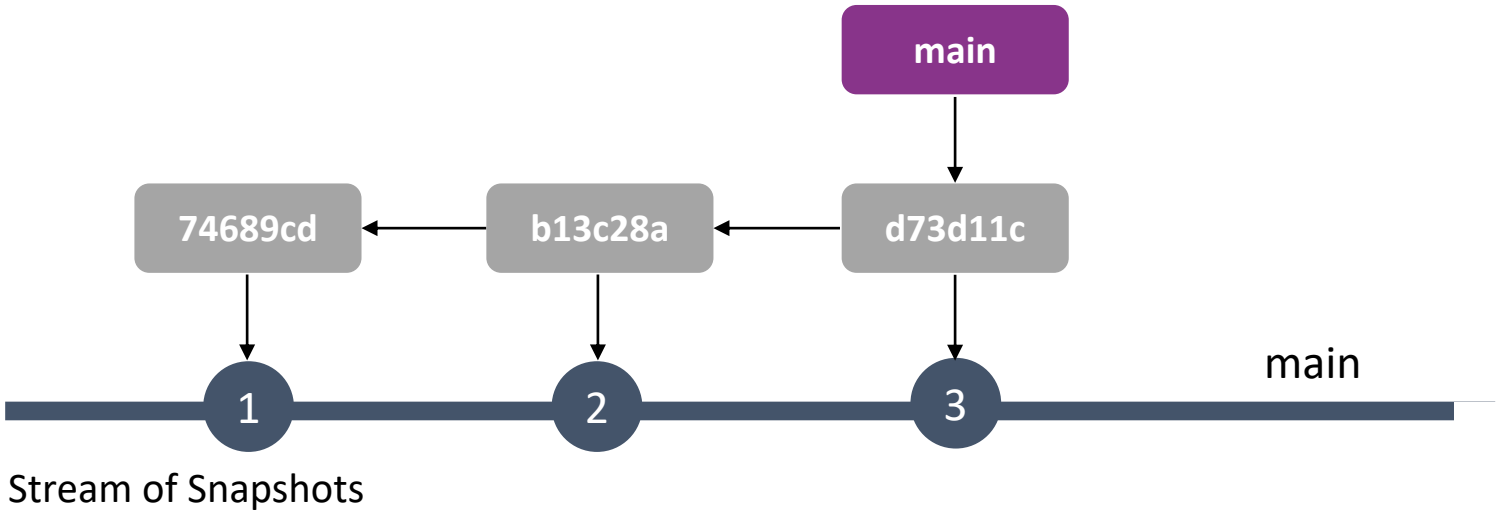
A Git Branch



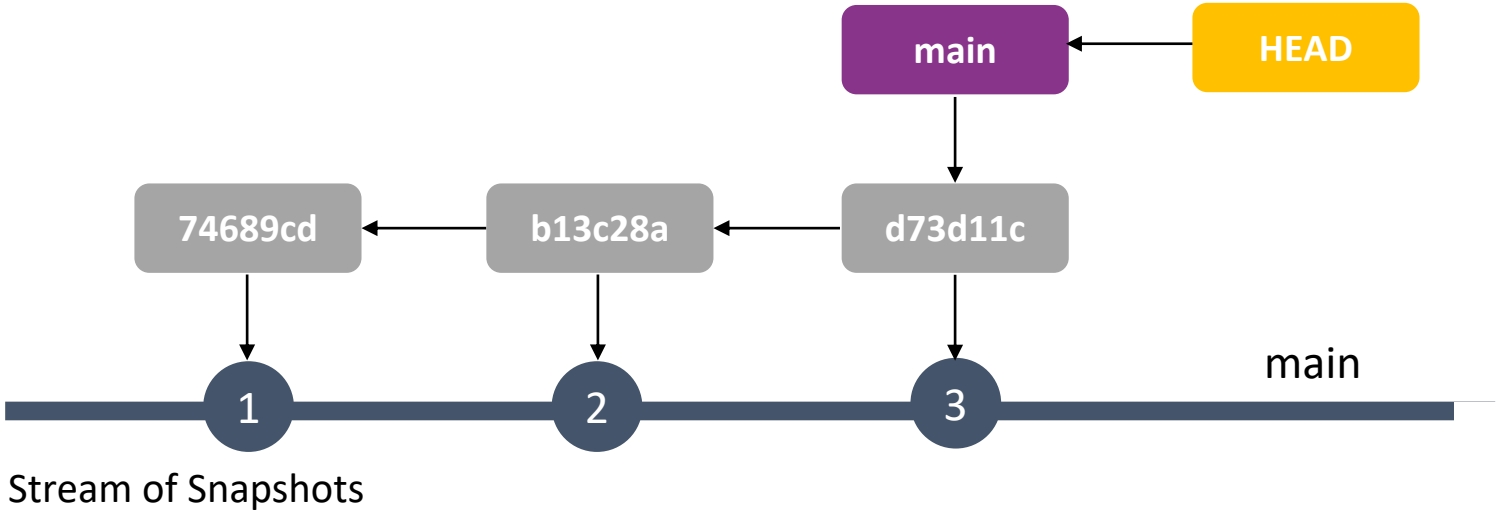
A Git Branch



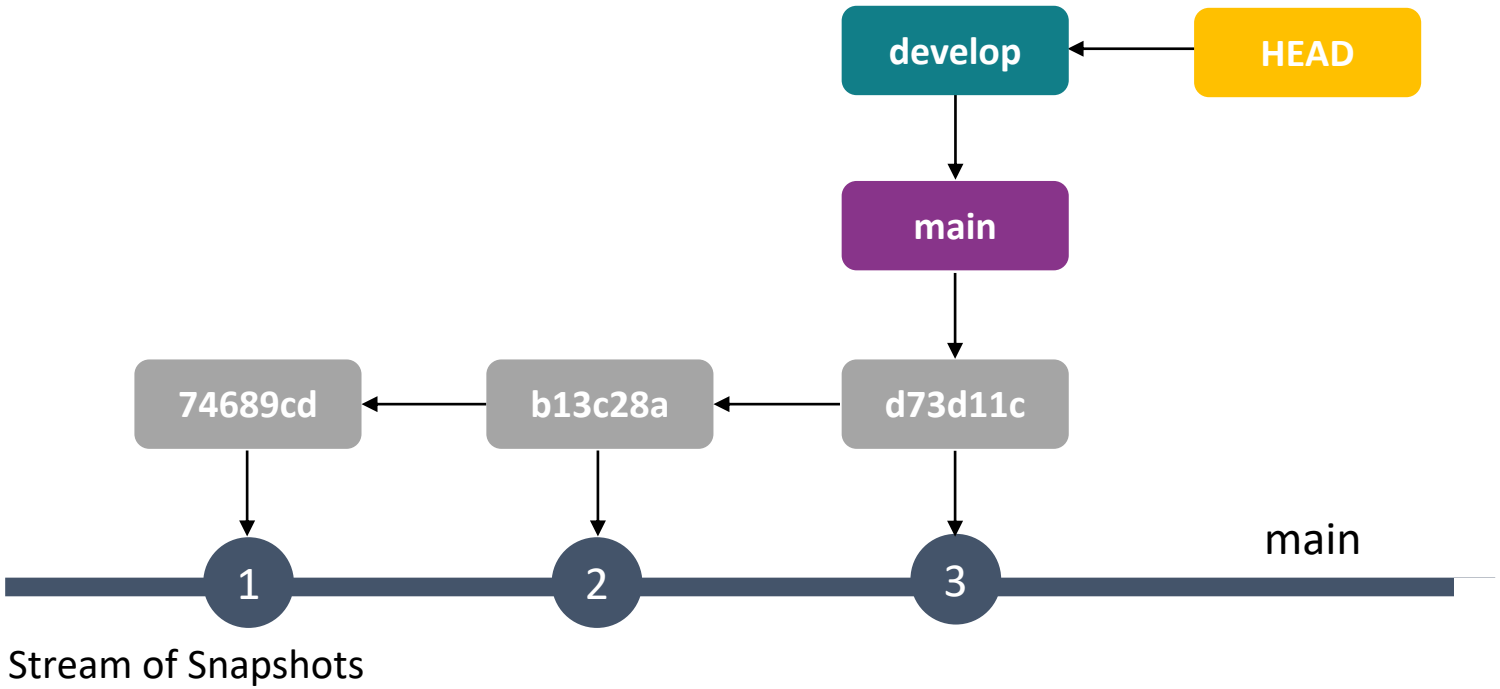
A Git Branch



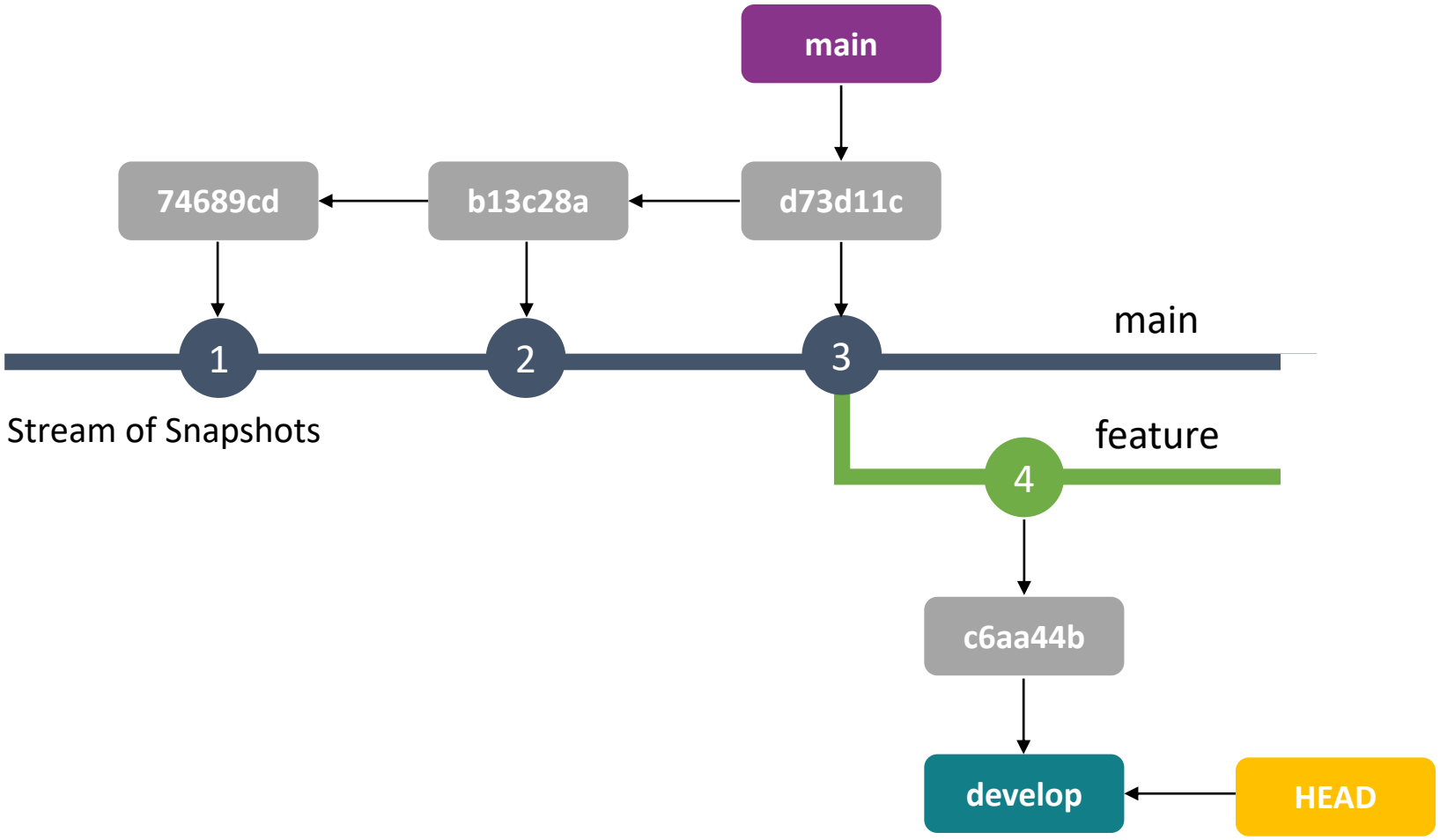
A Git Branch


main
HEAD
74689cd
b13c28a
d73d11c
main
1
2
3
Stream of Snapshots

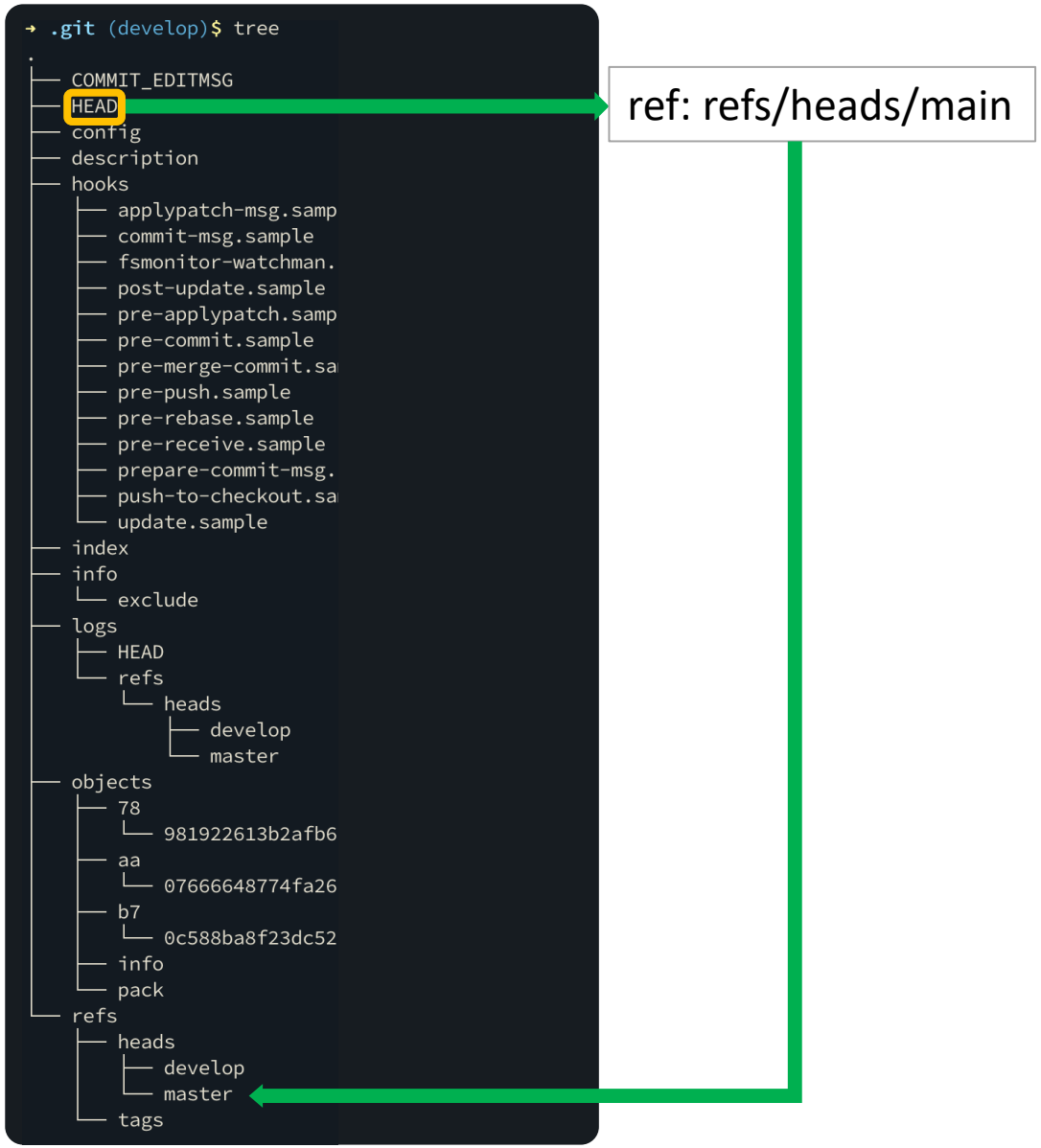
A Git Branch


develop
HEAD
main
74689cd
b13c28a
d73d11c
1
2
3
main
Stream of Snapshots

A Git Branch


main
74689cd
b13c28a
d73d11c
1
2
3
main
Stream of Snapshots
feature
4
c6aa44b
develop
HEAD

HEAD / branch refs



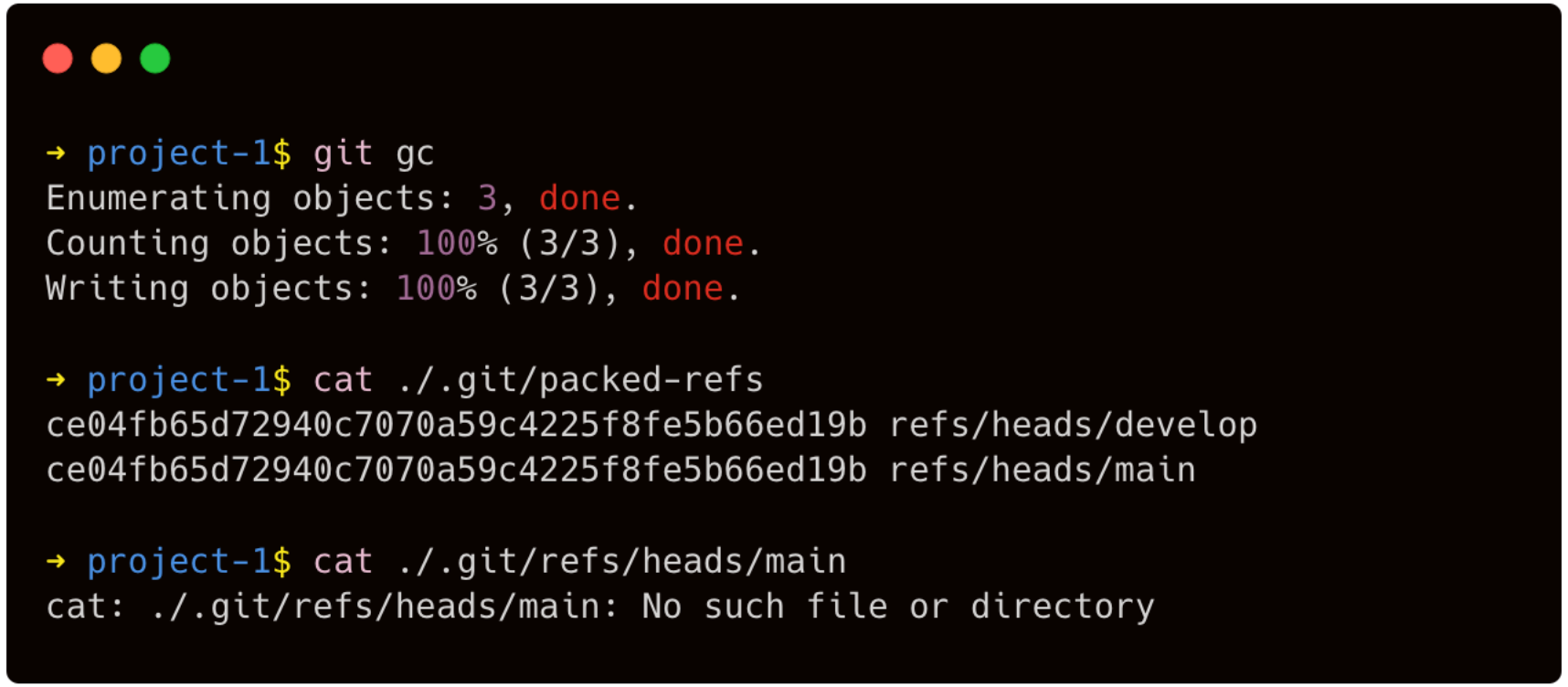
HEAD / branch refs

```
➜ project-1$ ls -1 ./.git
COMMIT_EDITMSG
HEAD
config
description
hooks
index
info
logs
objects
refs

➜ project-1$ ls -1 ./.git/refs/heads
develop
main
```

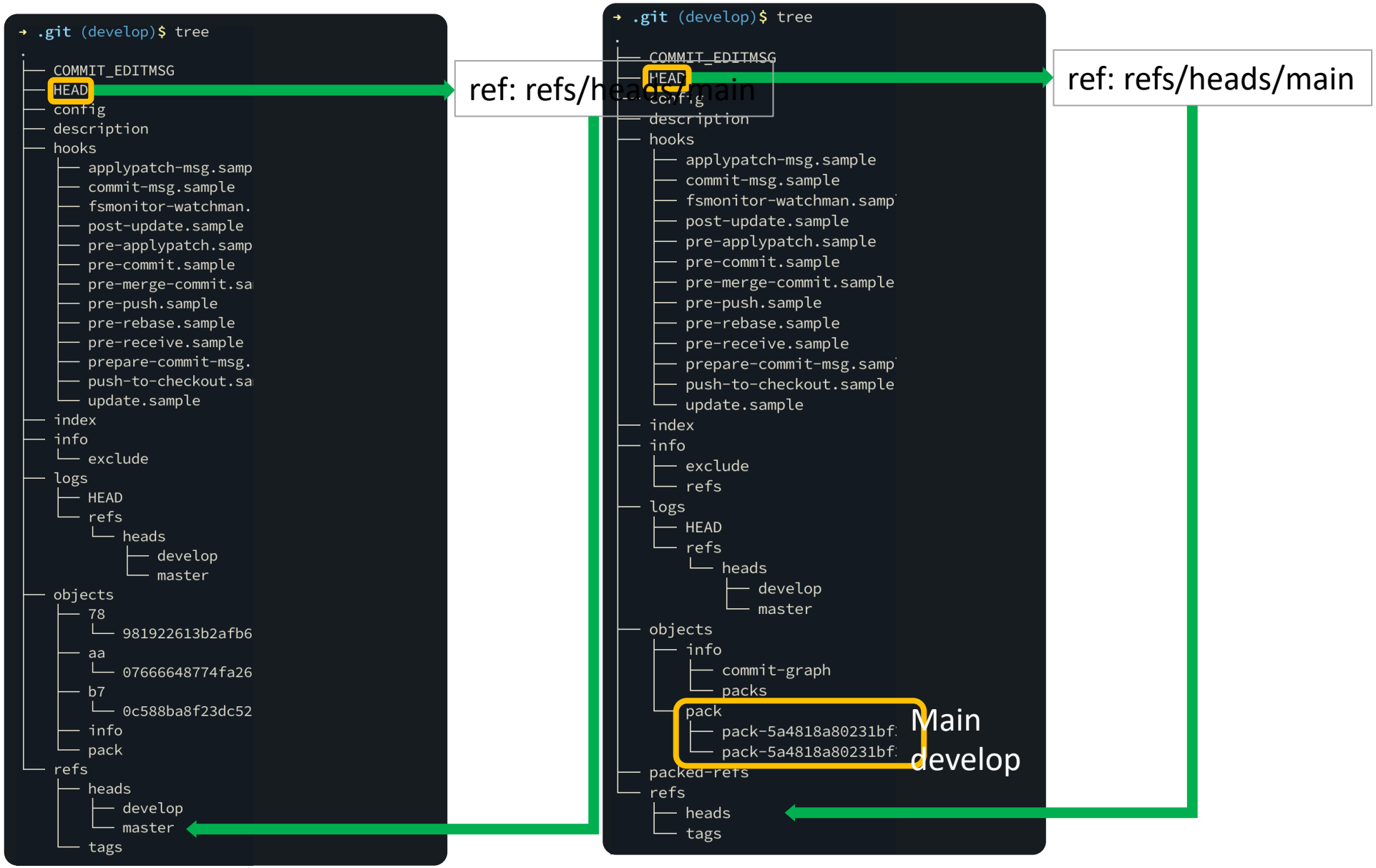
HEAD / branch refs

```
➜ project-1$ git branch
* main
develop

➜ project-1$ cat ./.git/HEAD
ref: refs/heads/main

➜ project-1$ cat ./.git/refs/heads/main
4e27ff85b94ff8dd773186bfa0e7e923e3346ce4  # commit ID
```

Garbage Collect



After Garbage Collect



Creating A Branch

```
$ git branch develop
$ git checkout develop
```

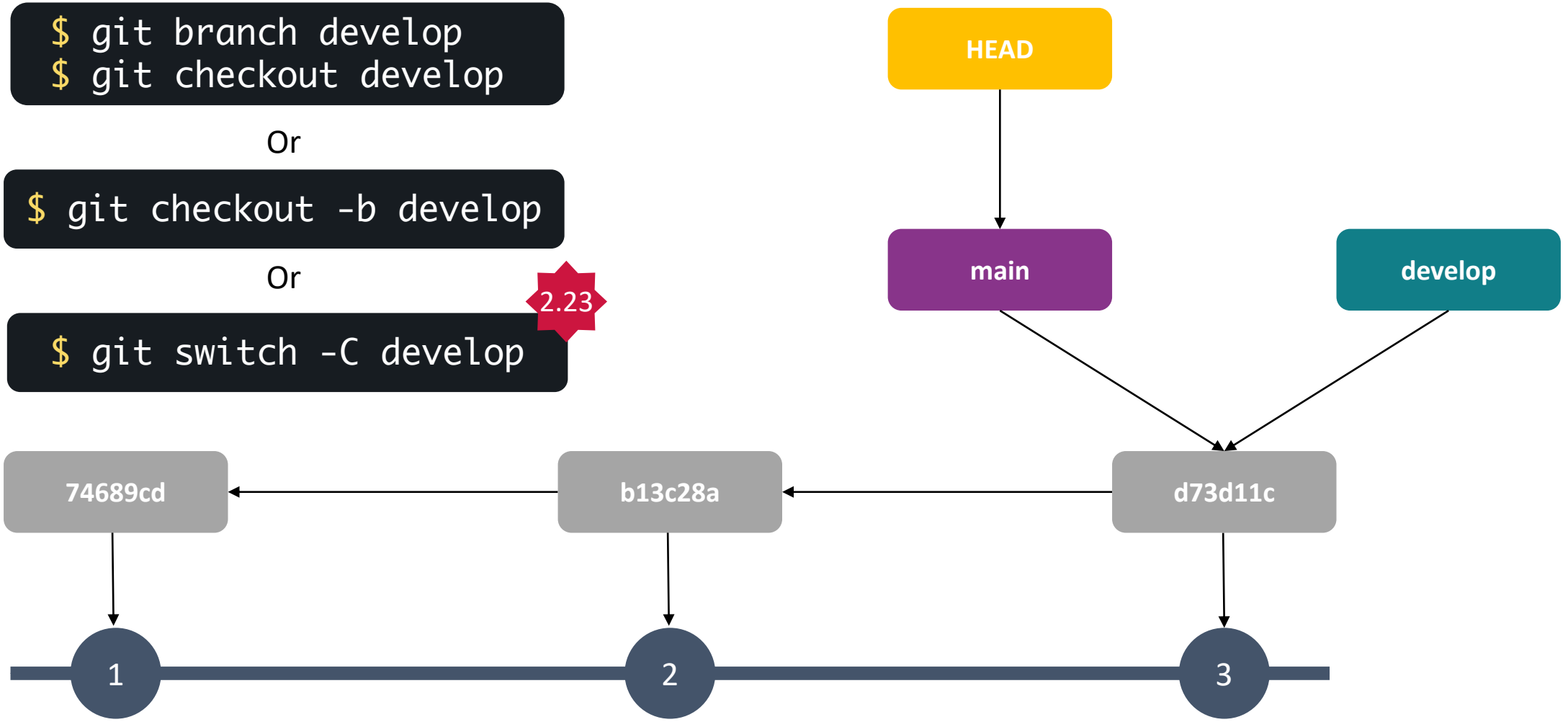
Or

```
$ git checkout -b develop
```

Or

```
$ git switch -C develop
```

2.23



Let us practice

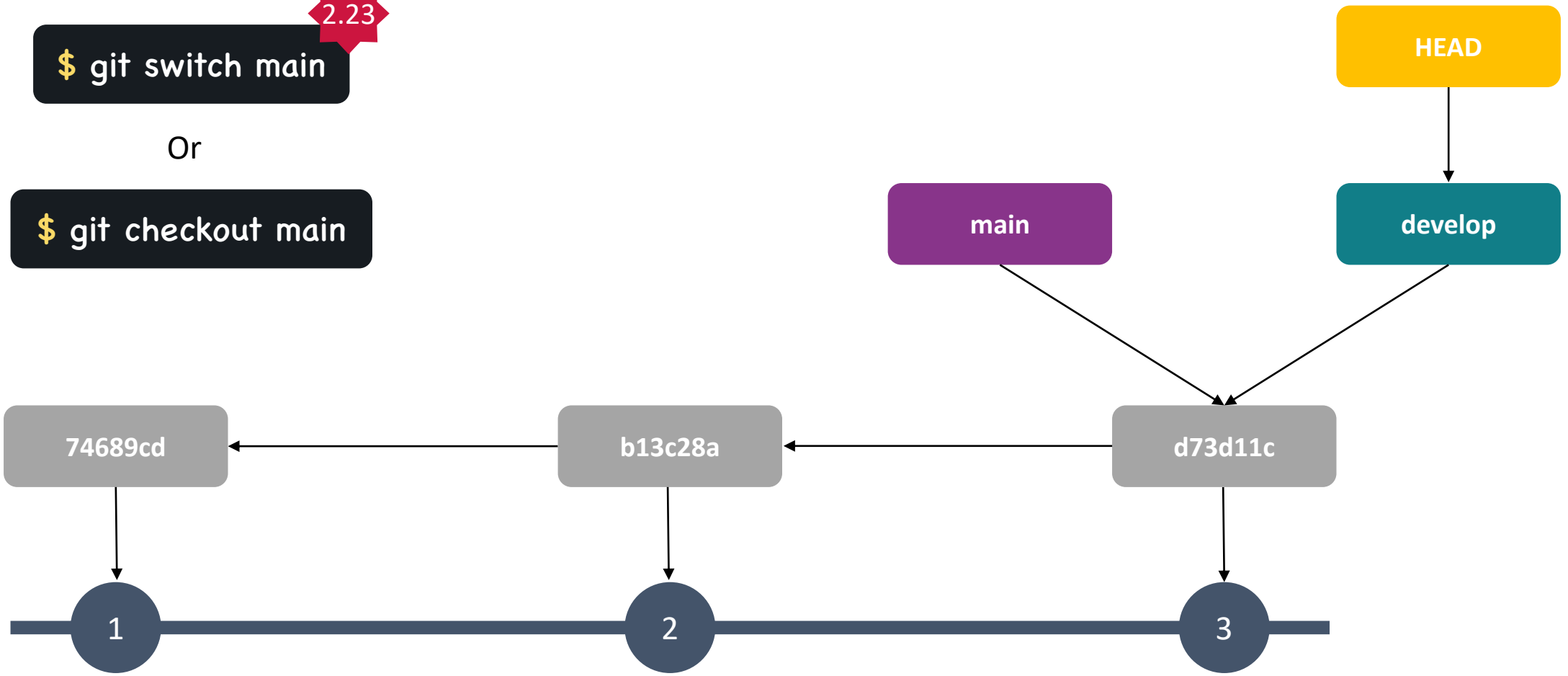
Switch Branch

2.23

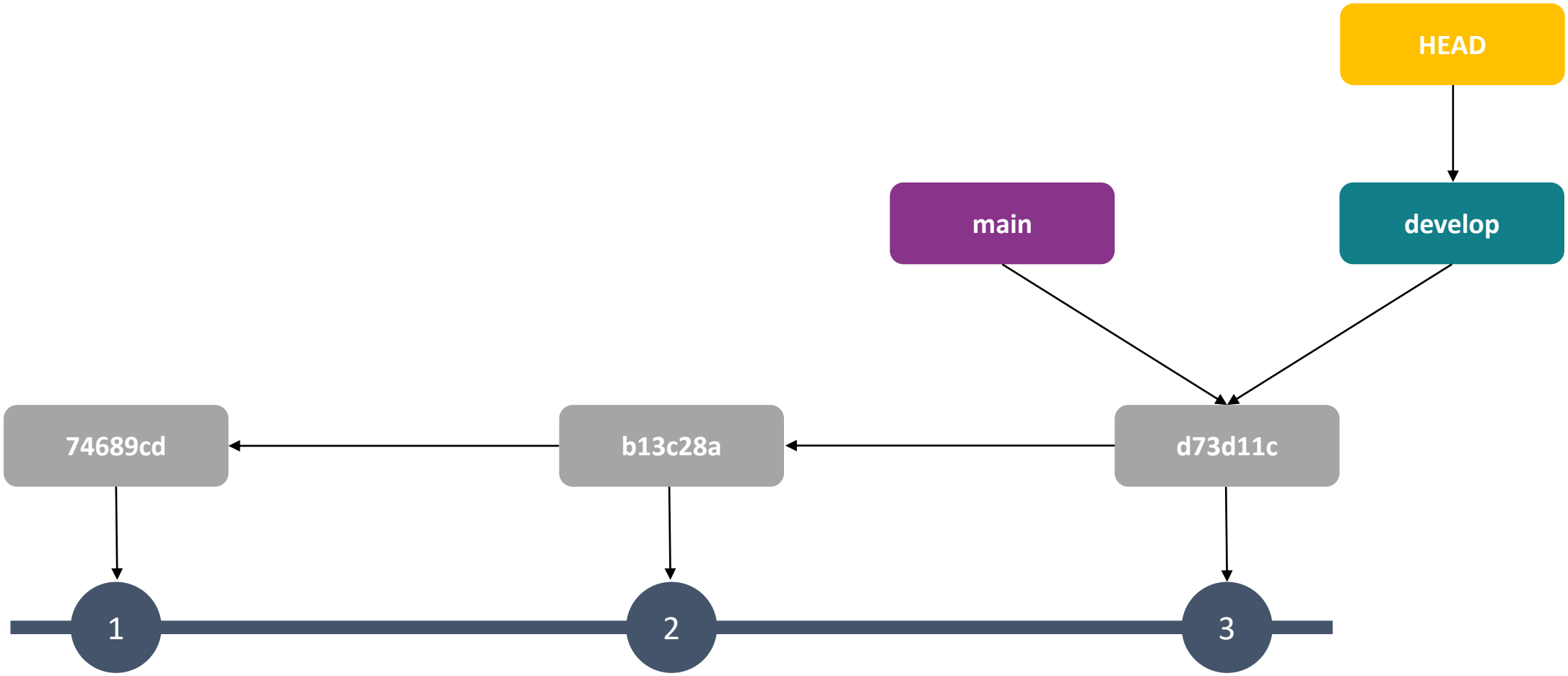
```
$ git switch main
```

Or

```
$ git checkout main
```

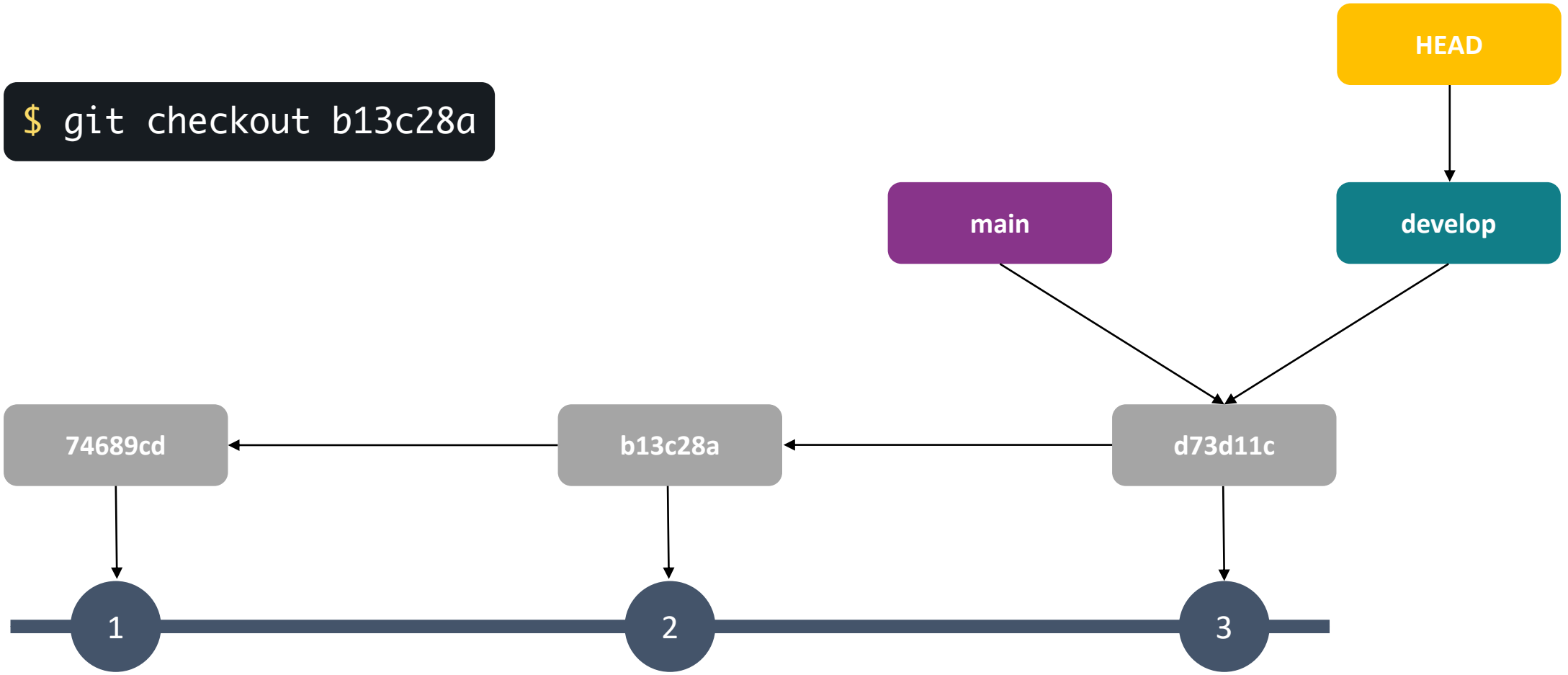


Detached HEAD

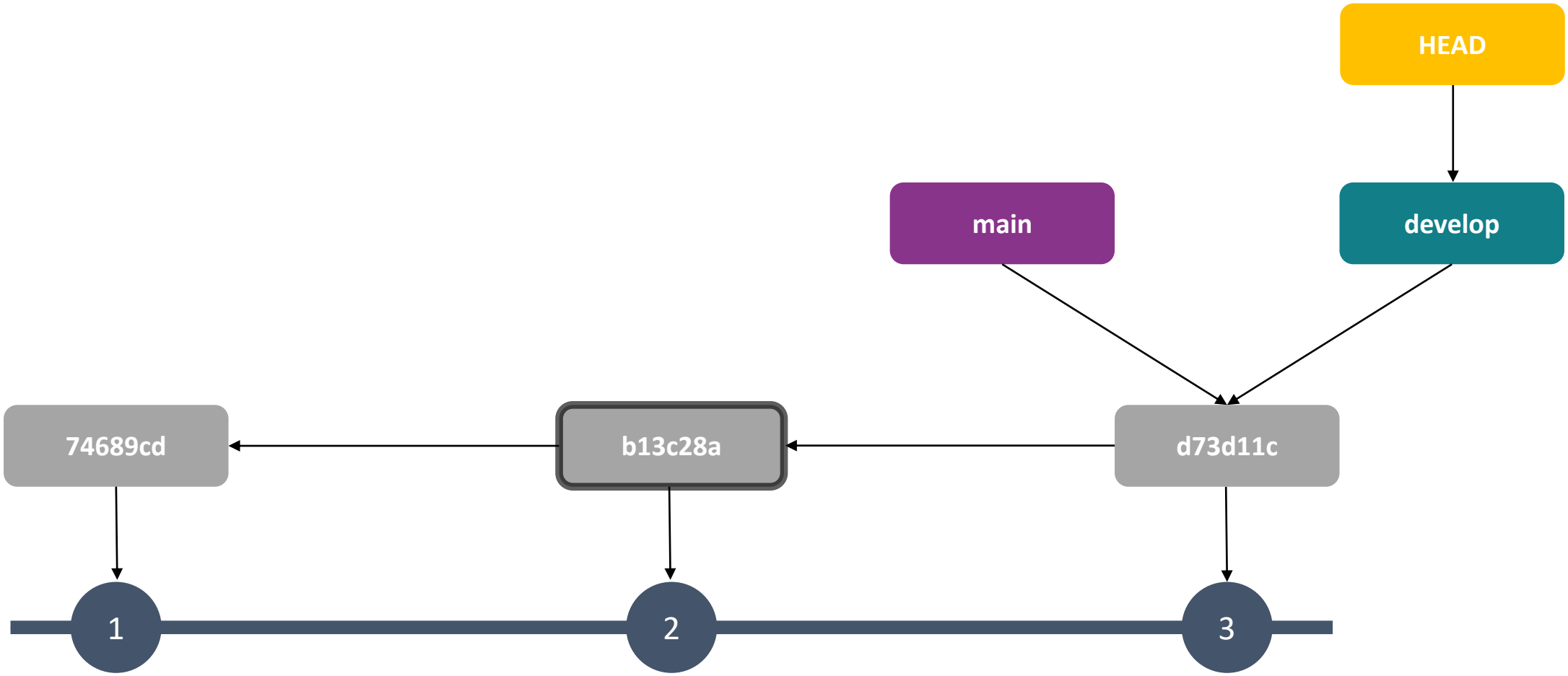


Detached HEAD

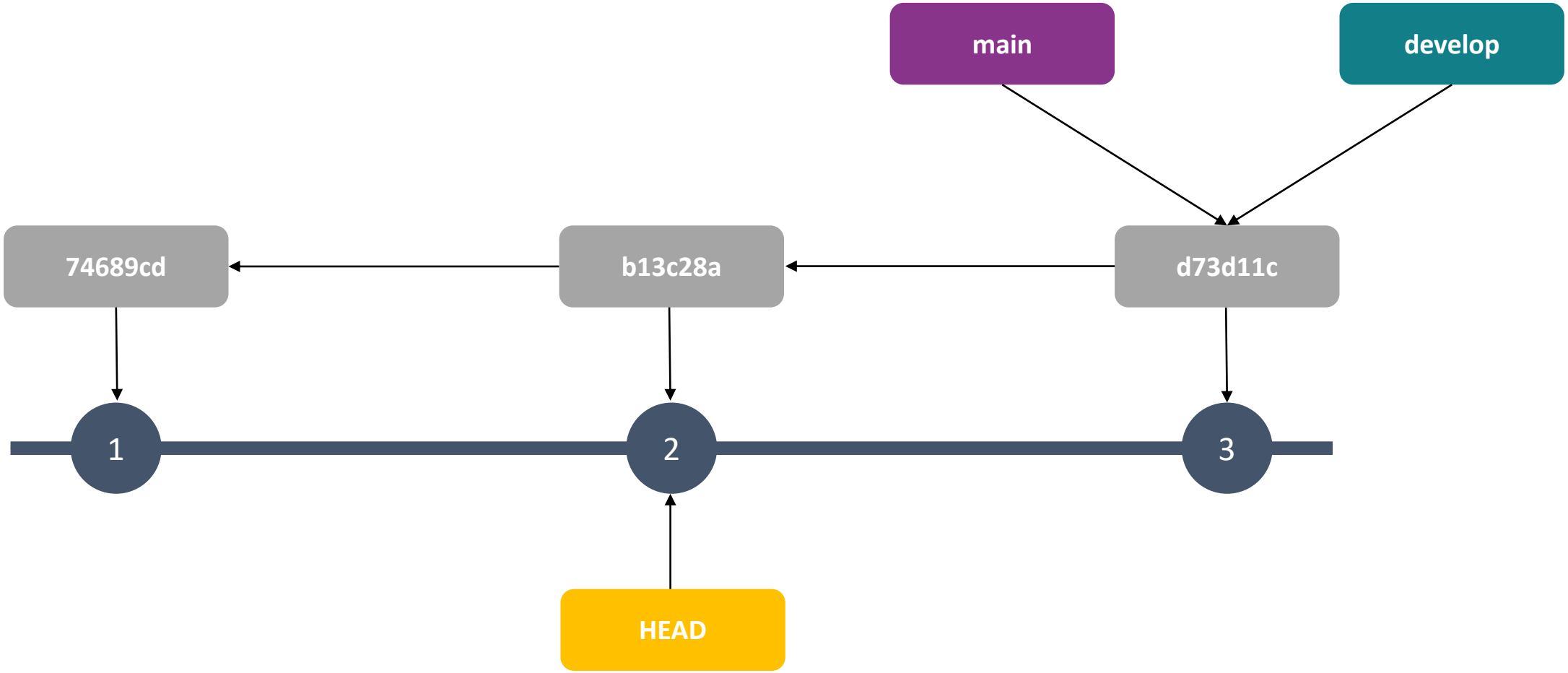
```
$ git checkout b13c28a
```



Detached HEAD

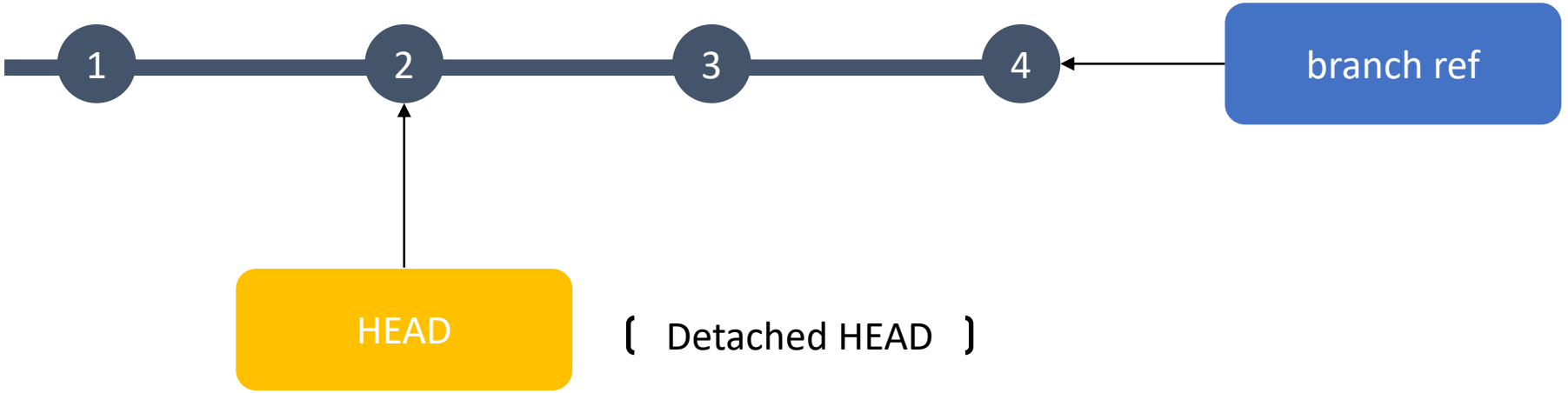


Detached HEAD



git checkout <commit ID>


```
git checkout 2 # commit ID 2
```



Detached HEAD – When?

1. `git checkout HEAD~1`
2. `git checkout 53ec1 (commit ID)`

We will look into the below commands in the later videos

1. `git checkout tags/v1.1`
2. `git rebase (when conflicts happen)`

Detached HEAD

1. Is Detached head good or bad?
2. What do you do when you get a detached head?
3. How to get away from a detached head?

Takeaways

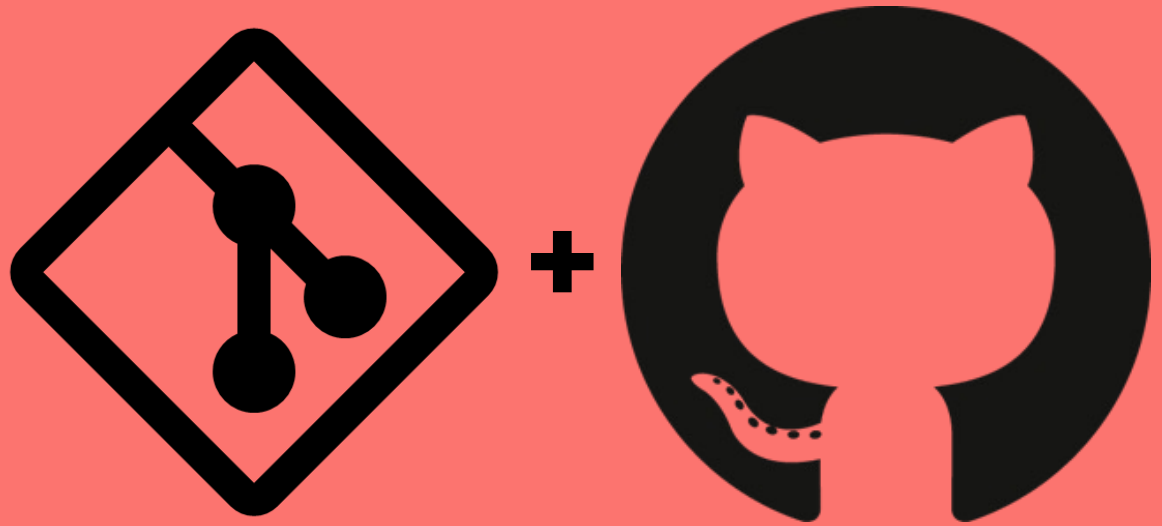
1. A branch is a set of code changes grouped together with a unique name
2. A branch is **a pointer to a specific commit**
3. The branch pointer moves along with each new commit you make, and only diverges in the graph if a commit is made on a common ancestor commit
4. The branch is also a type of a reference
5. Detached head…TODO
6. Commands
    a. `git branch – to List`
    b. `git branch <branchname> – to create`
    c. `Git checkout <branchname> – to change HEAD`
    d. `git checkout –b <branchname> – to create and change HEAD`
    e. `git switch <branchname> – to switch between branches`

Let us practice

Git

In Practice

Basics



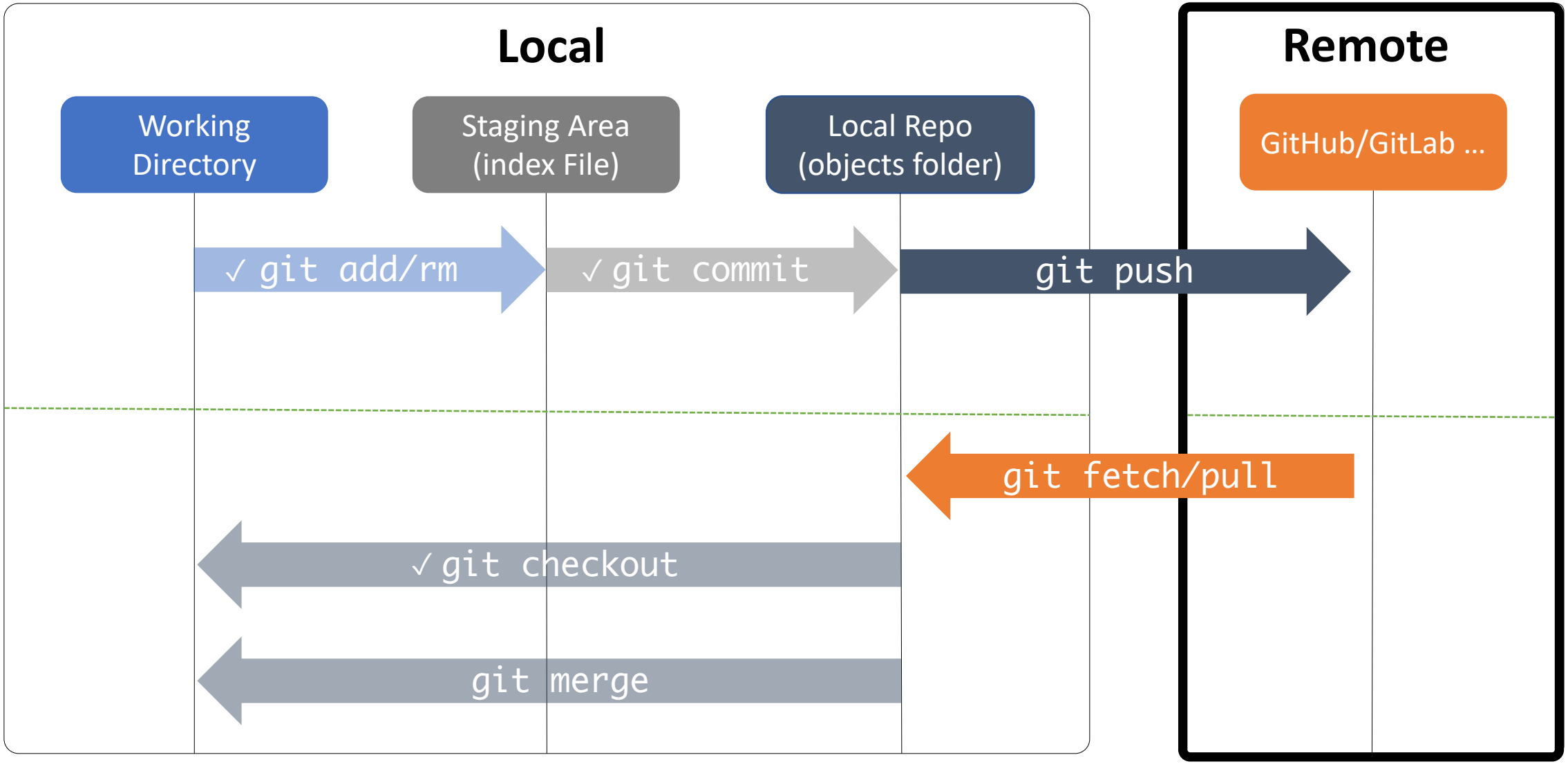
1 Git Branches

2 Remote Branches

What Is Remote?

1. TODO

The Remote State



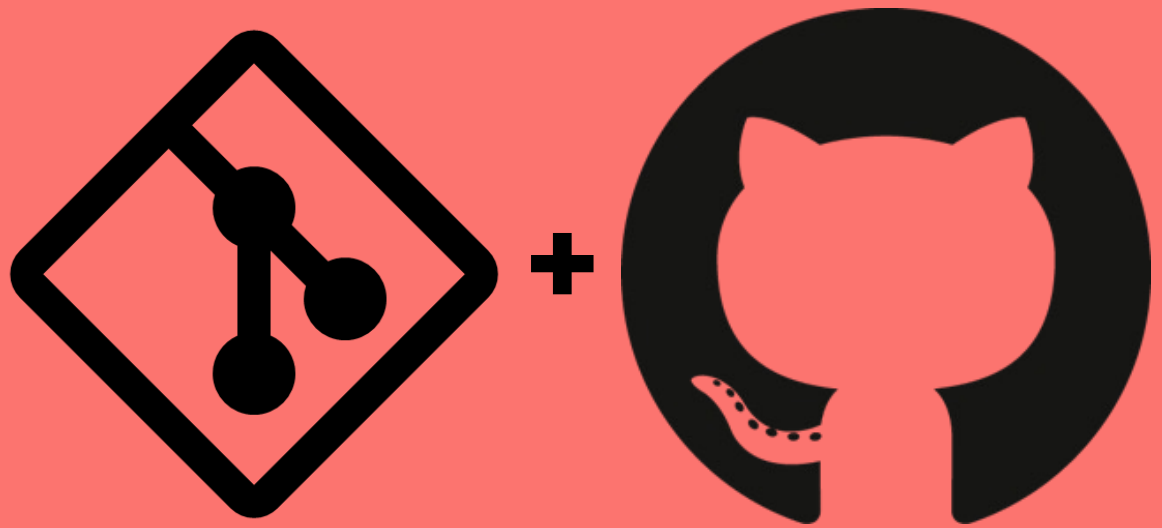
Remote Branches

1. `echo "# github-slides" >> README.md`
2. `git init`
3. `git add README.md`
4. `git commit -m "first commit"`
5. `git branch -M main`
6. `git remote add origin git@github.com:r-raman/github-slides.git`
7. `git push -u origin main`

Git

In Practice

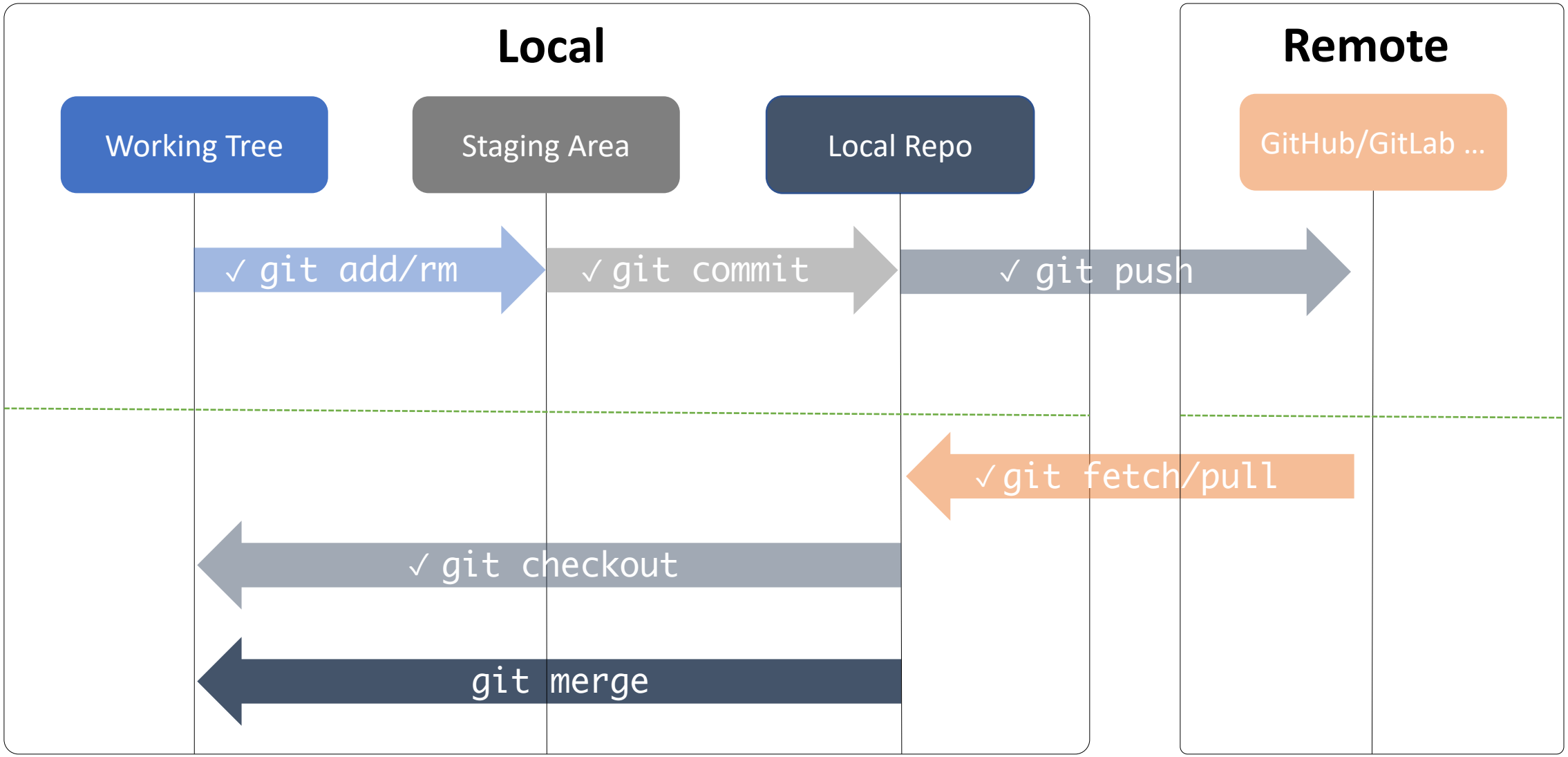
Basics



1. git merge
2. git rebase
3. merge vs rebase

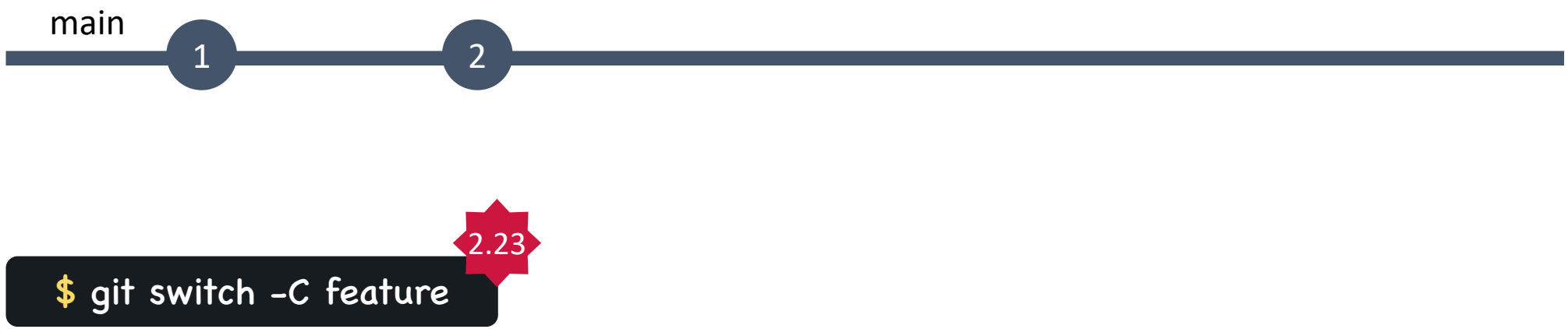
Basic Merges

Project Components – Merge



What is a merge?

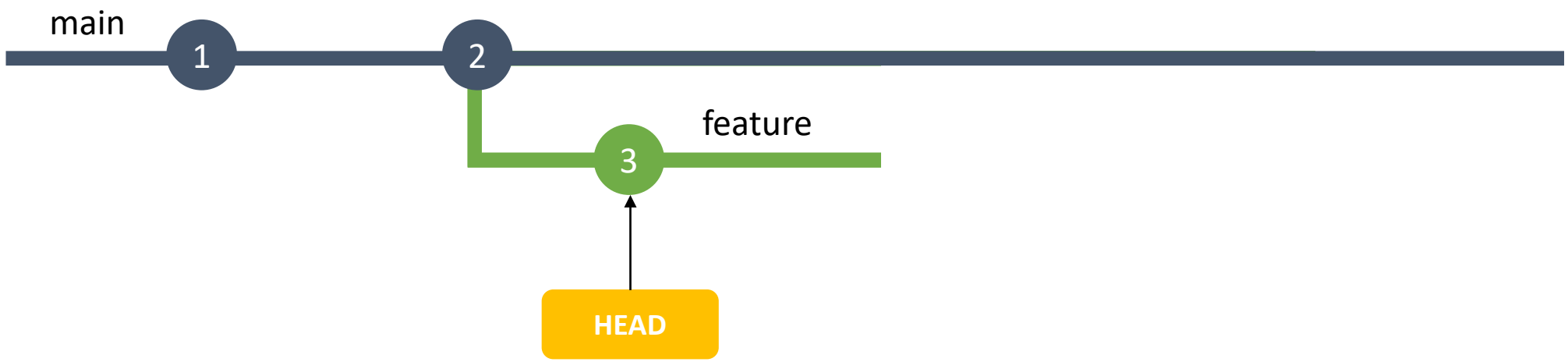
What is a merge?



What is a merge?

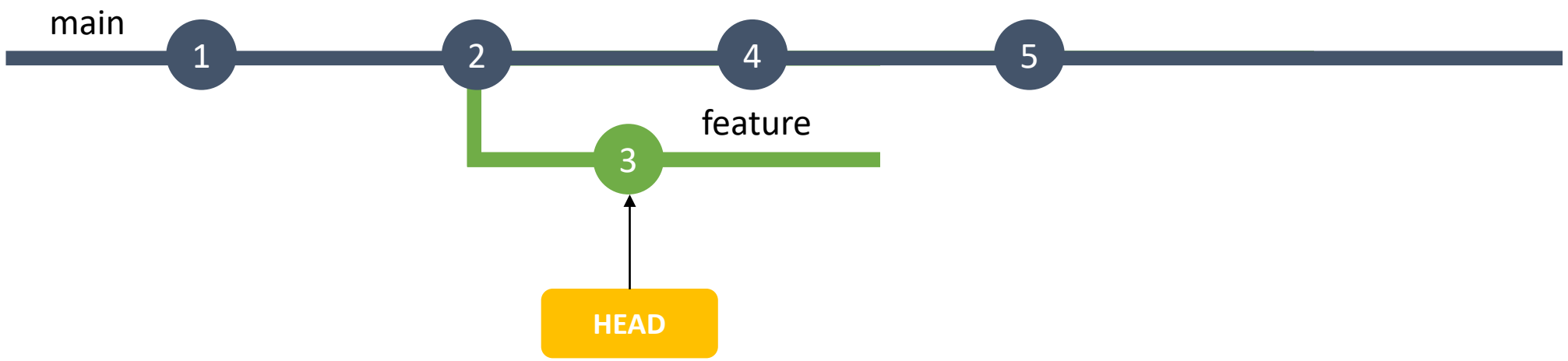
What is a merge?

New commit in feature



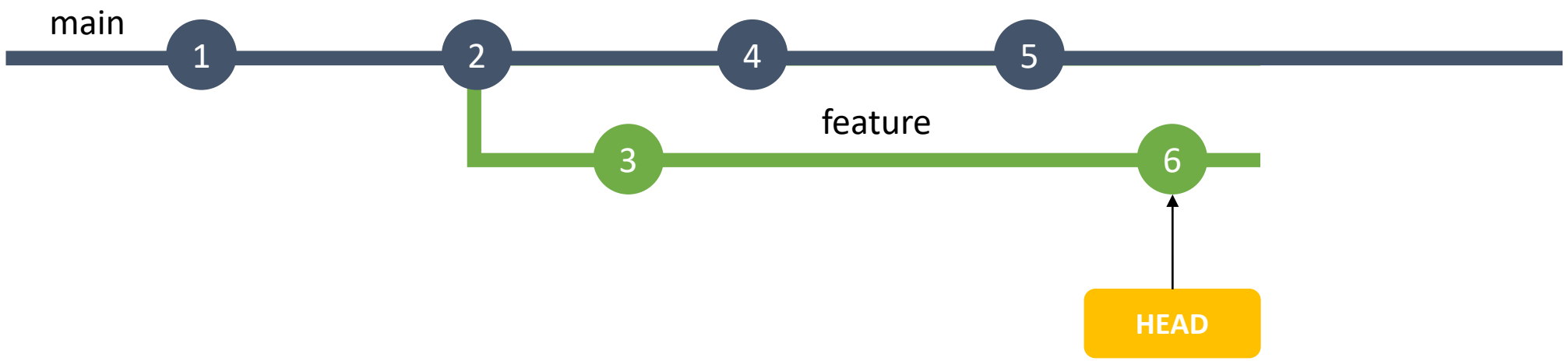
What is a merge?

More commits in main



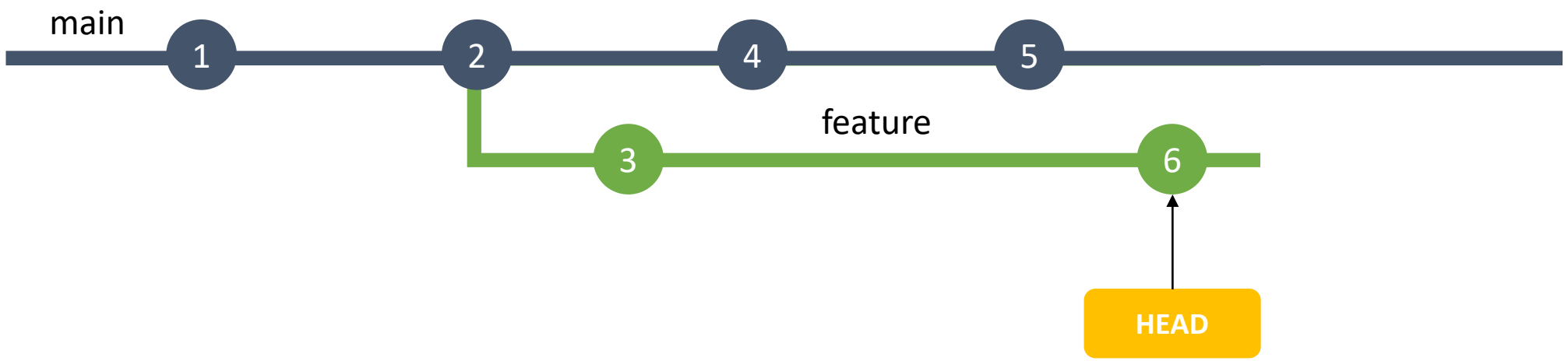
What is a merge?

One more commit in feature

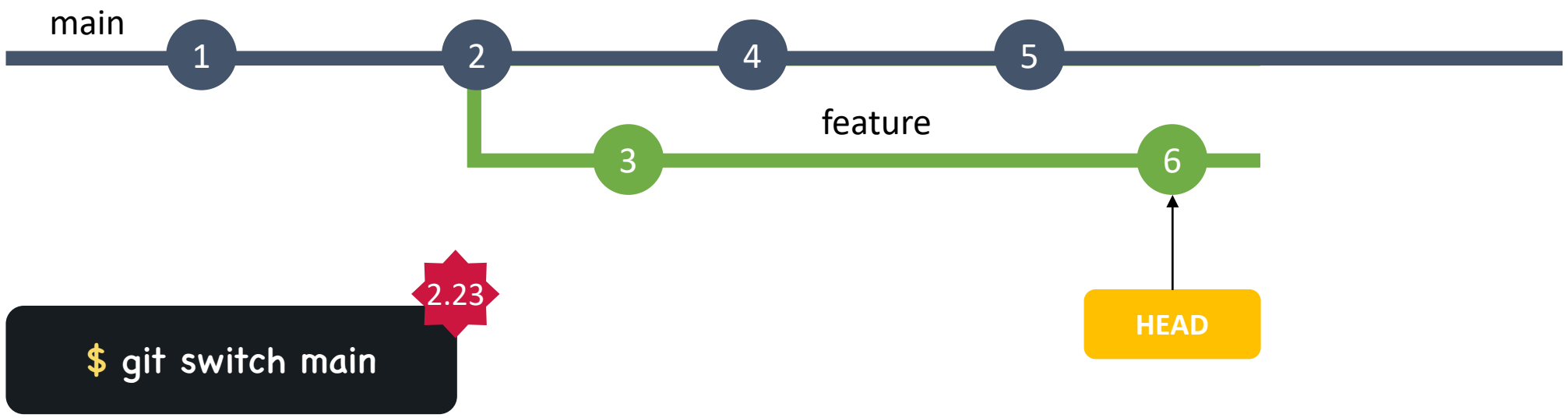


What is a merge?

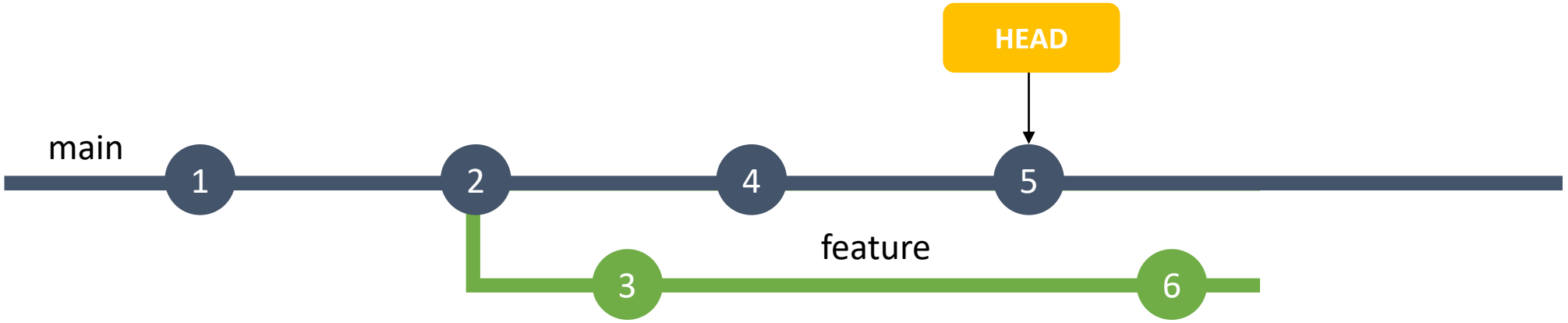
Merge develop to main



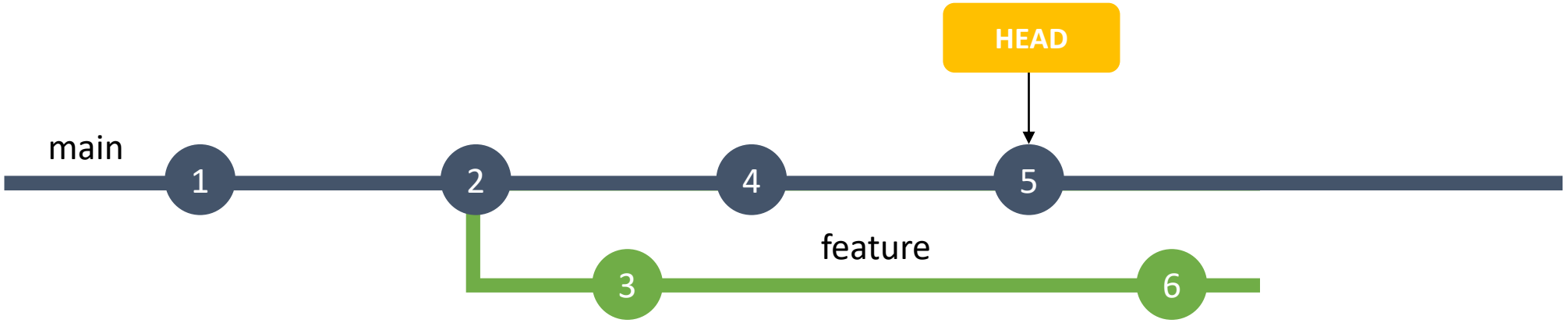
What is a merge?



What is a merge?



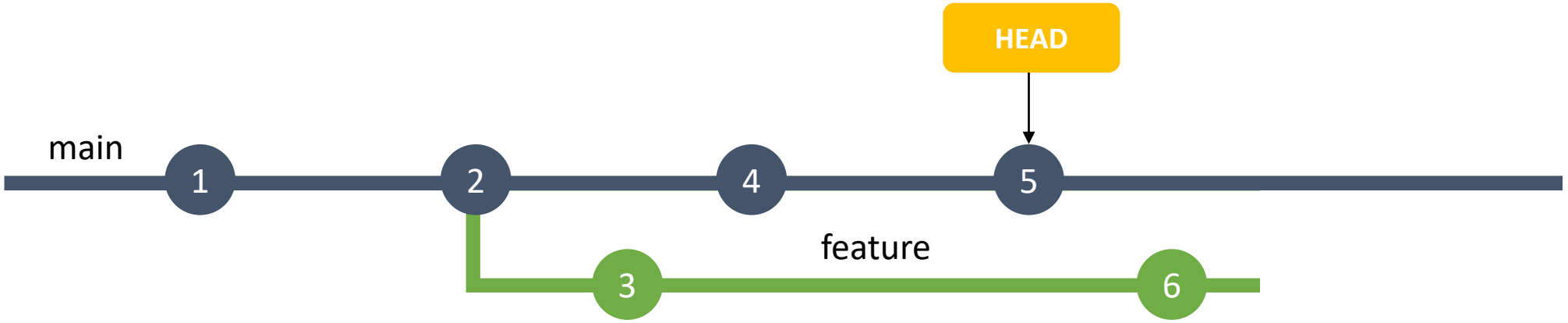
What is a merge?



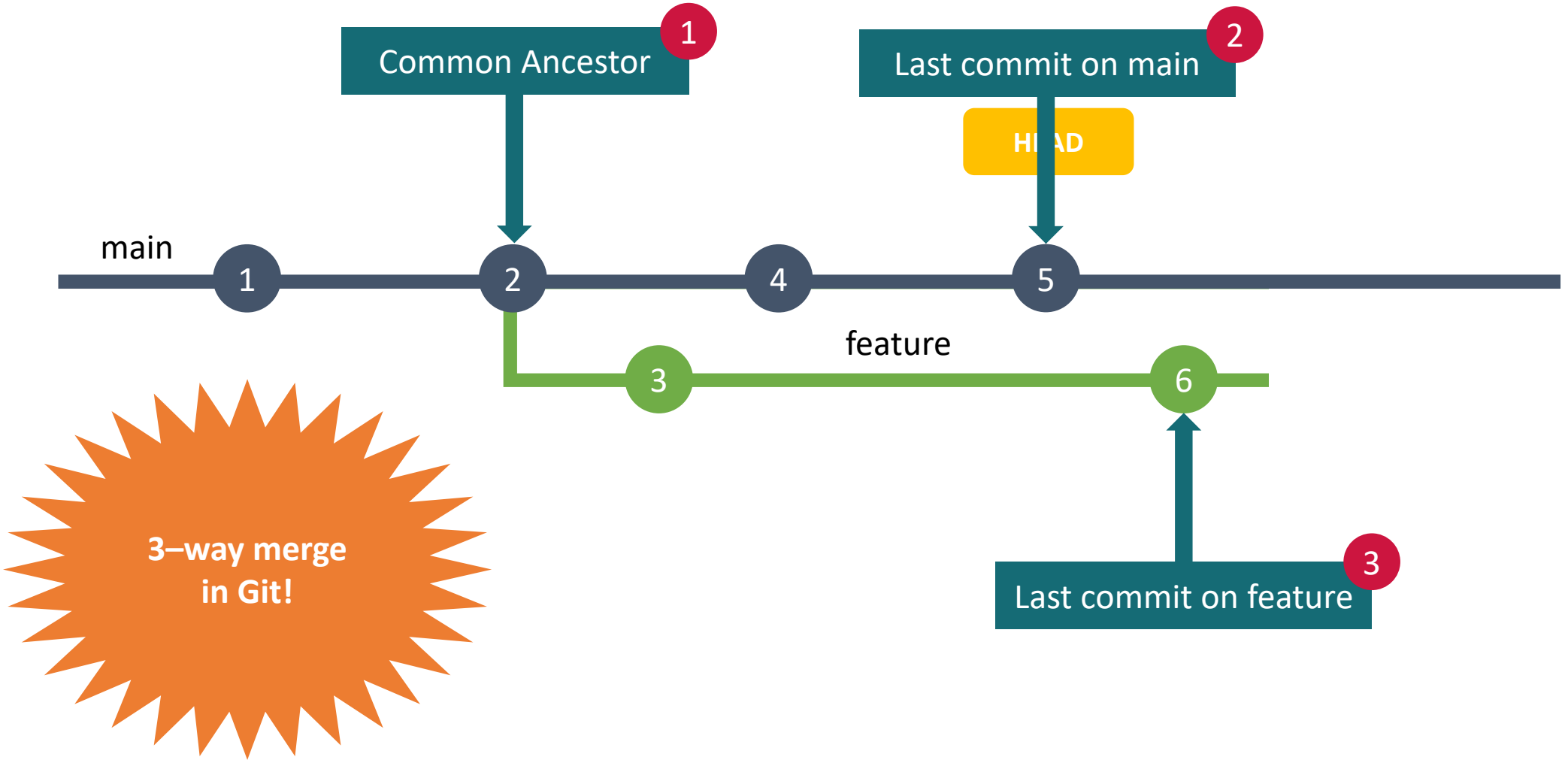

```
$ git fetch feature
TODO
```

What is a merge?

```
$ git merge feature
```

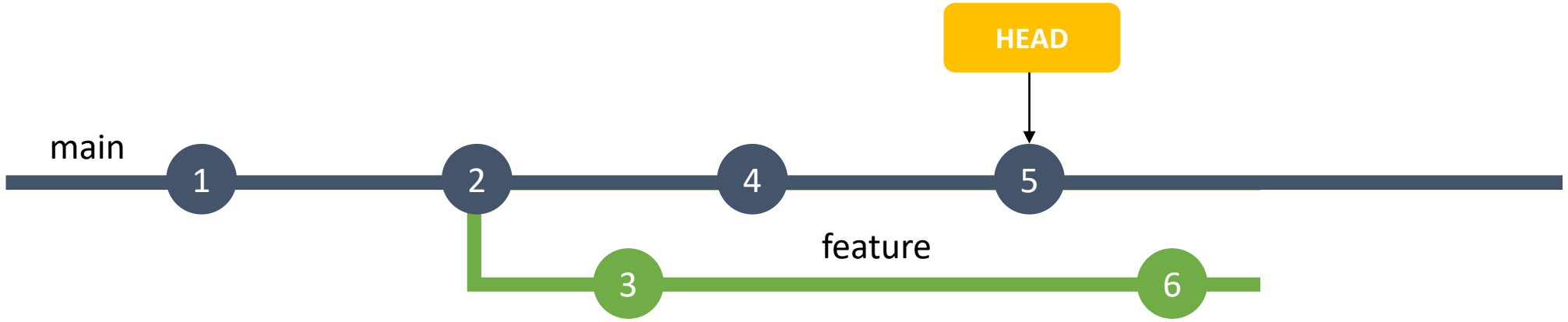


What is a merge?

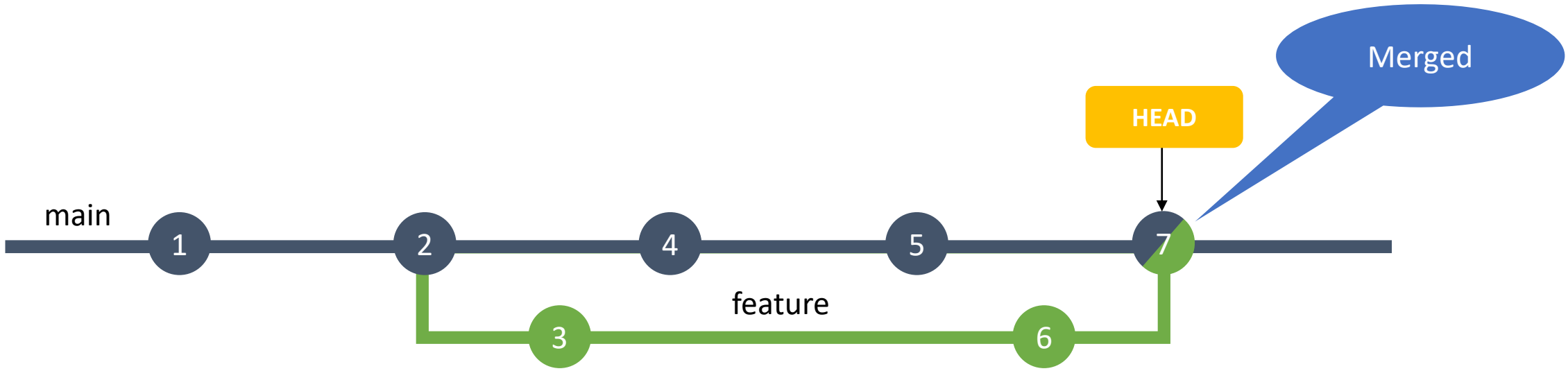


What is a merge?

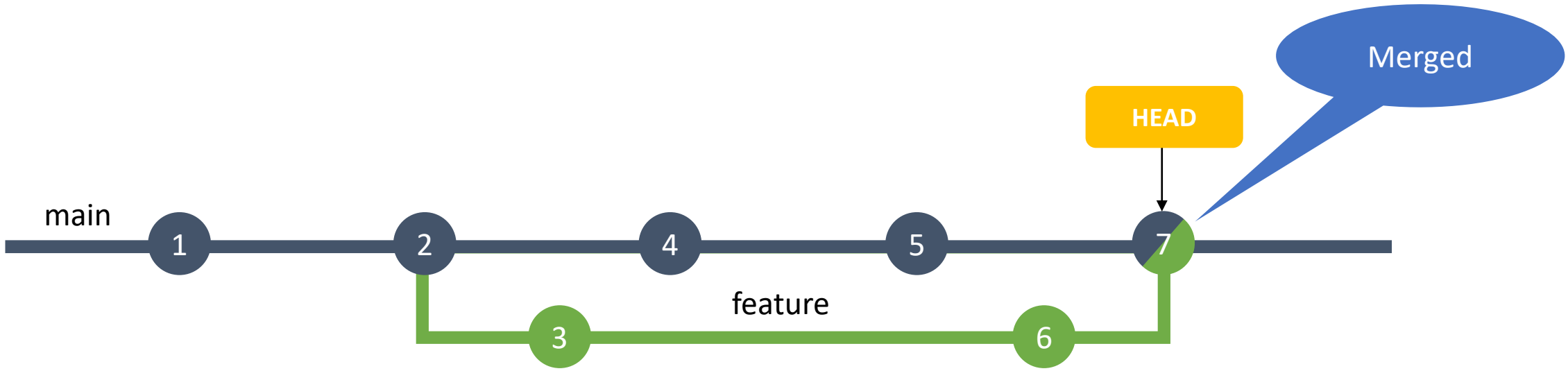
Merge develop to main



What is a merge?



What is a merge?




```
$ git branch -d feature
```

What is a merge?

What is a merge?

Takeaway

*Git supports **merge into** feature and not merge from!*

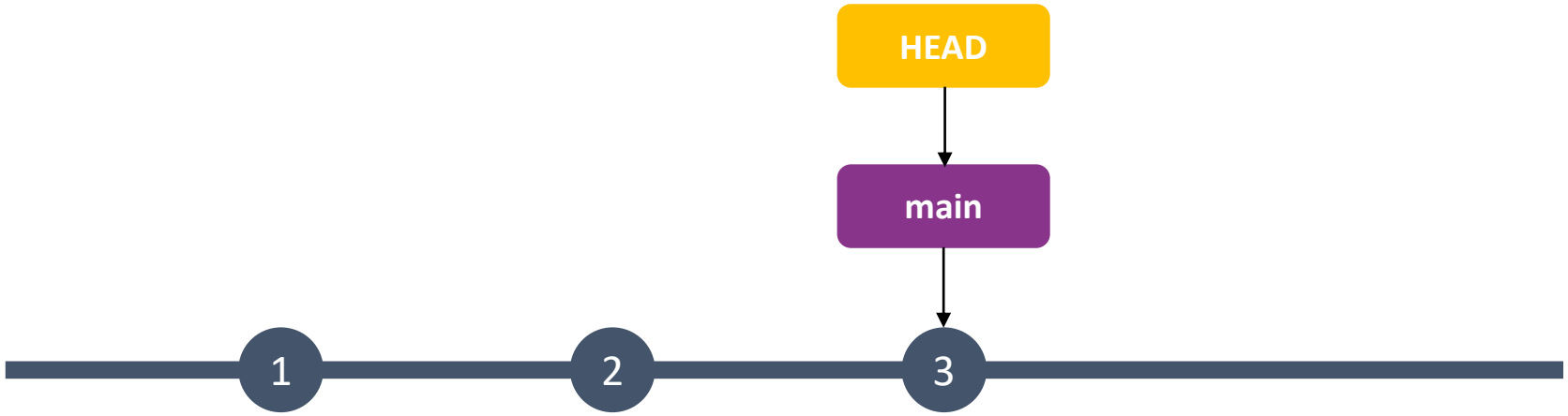
Always switch to the branch you want to merge into first, before merging

Types of merges

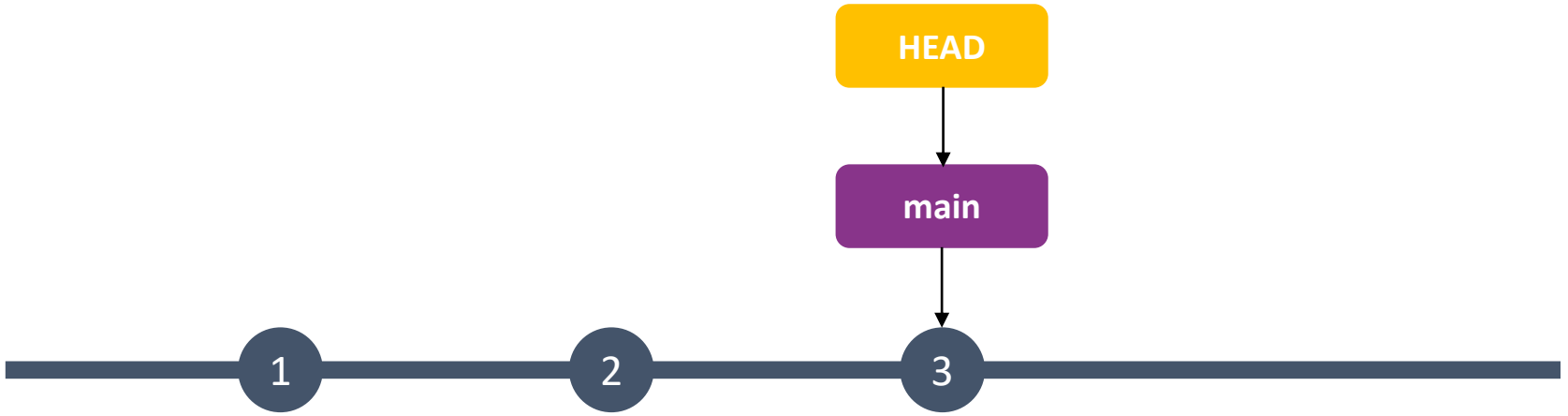
1. Fast Forward Merges – Merges that don’t create a new snapshot
2. Merge commits – Merges that create a new snapshot
   - Merge without conflict
   - Merge with conflict

Fast Forward Merge

Scenario 1 – Fast Forward Merge



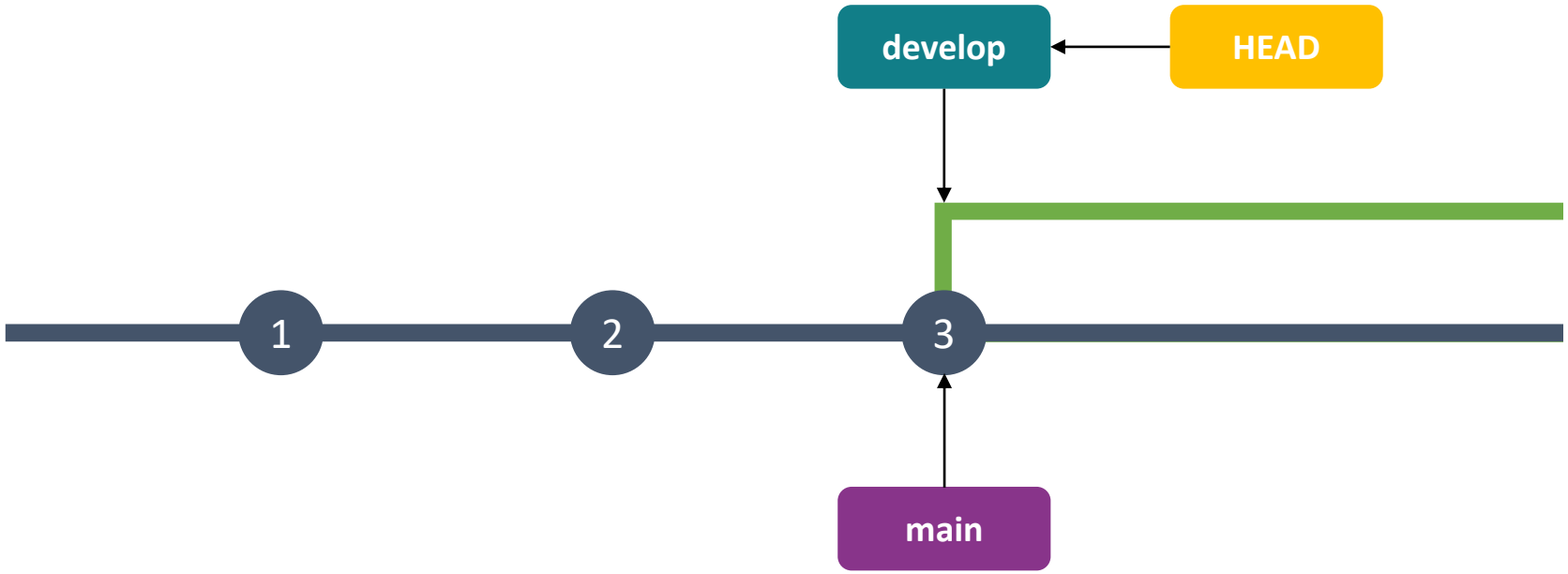
Scenario 1 – Fast Forward Merge



2.23

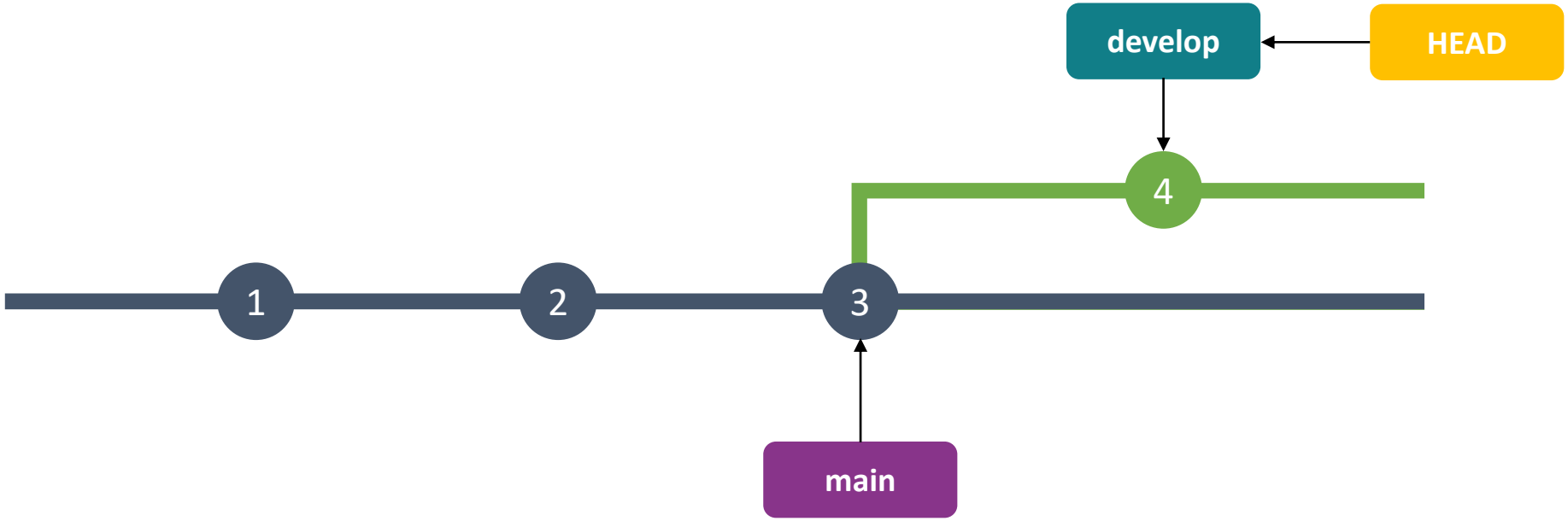
```
$ git switch -C develop
```

Scenario 1 – Fast Forward Merge

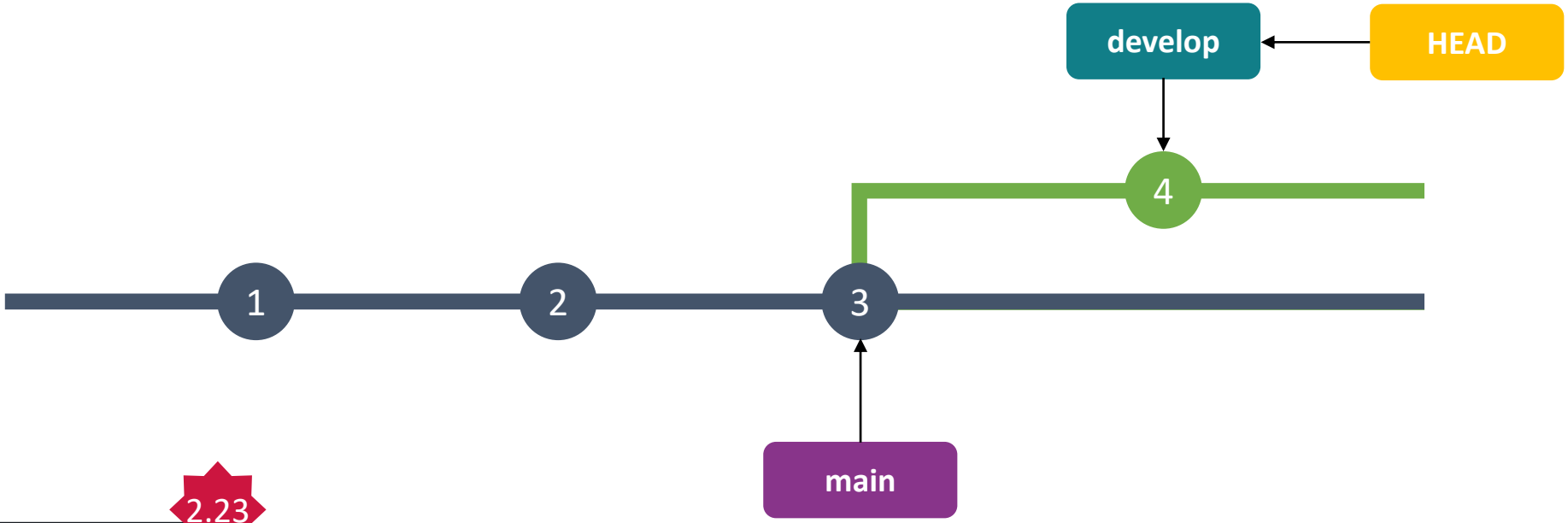


Scenario 1 – Fast Forward Merge

New commit in develop



Scenario 1 – Fast Forward Merge

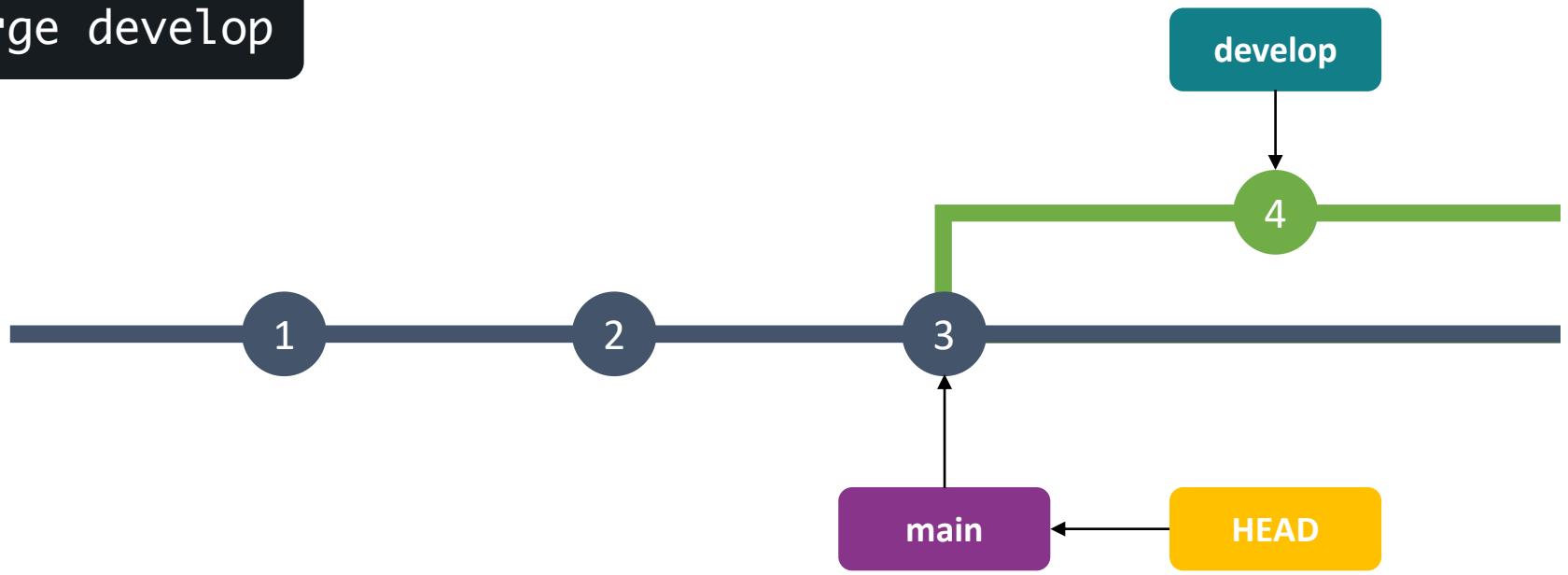


2.23

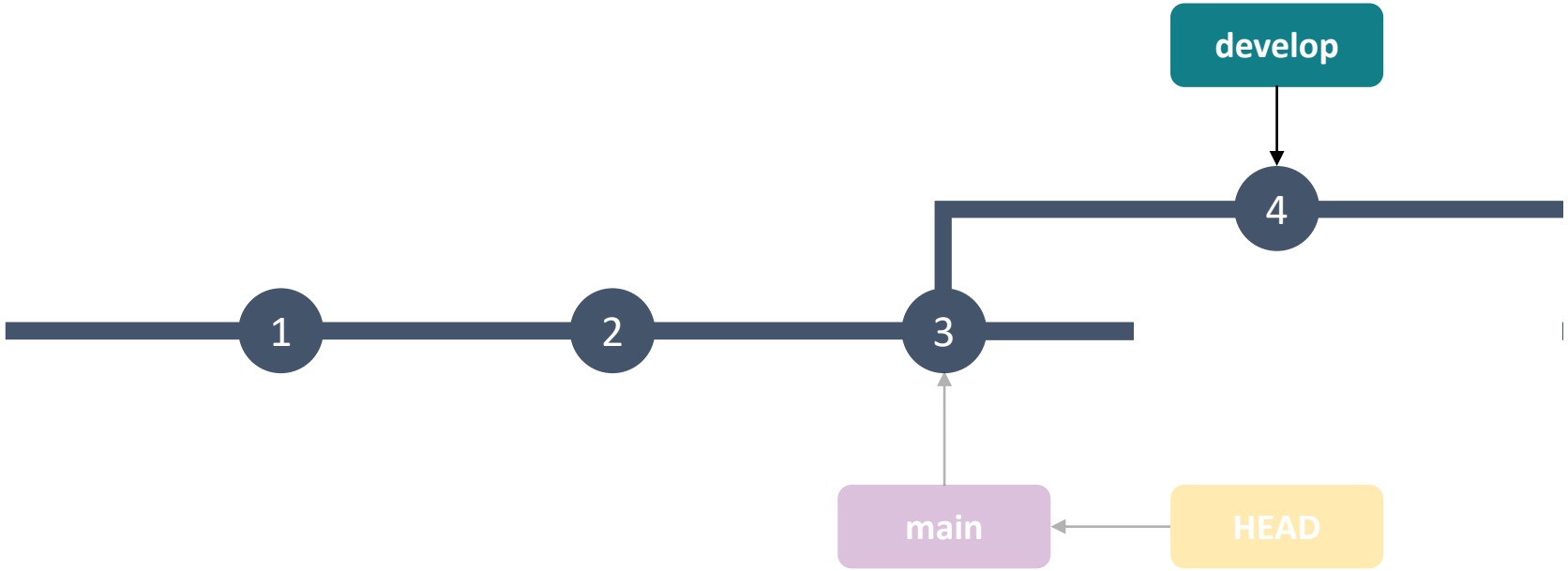
```
$ git switch main
```

Scenario 1 – Fast Forward Merge

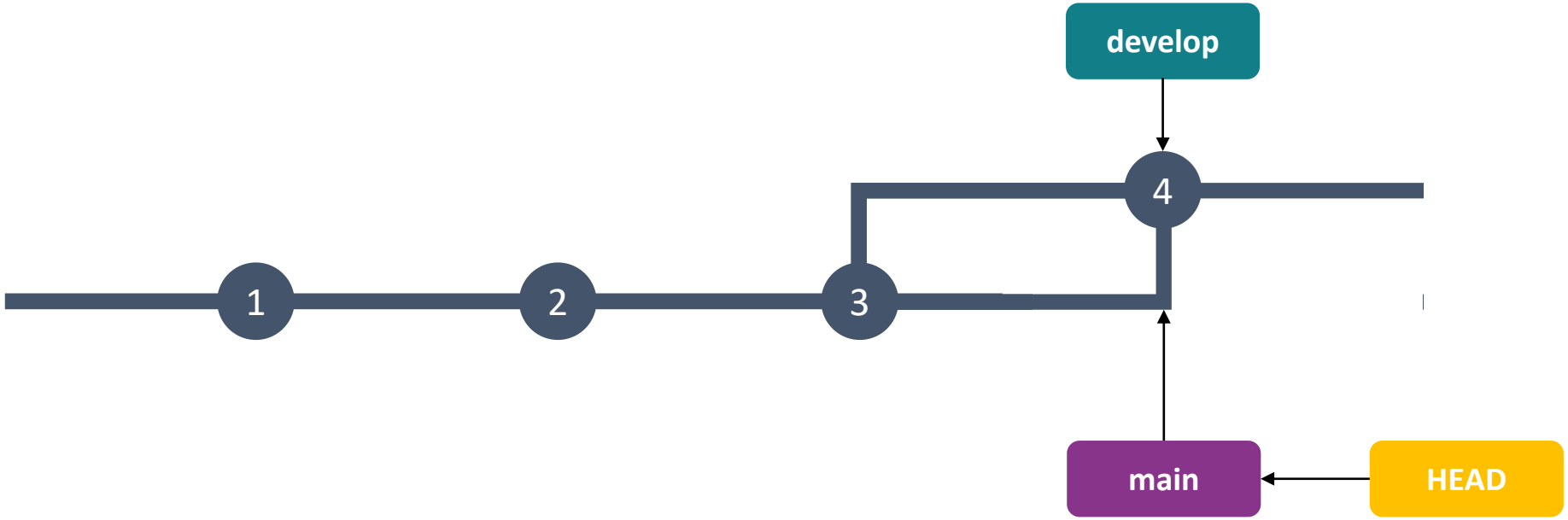
```
$ git merge develop
```



Scenario 1 – Fast Forward Merge

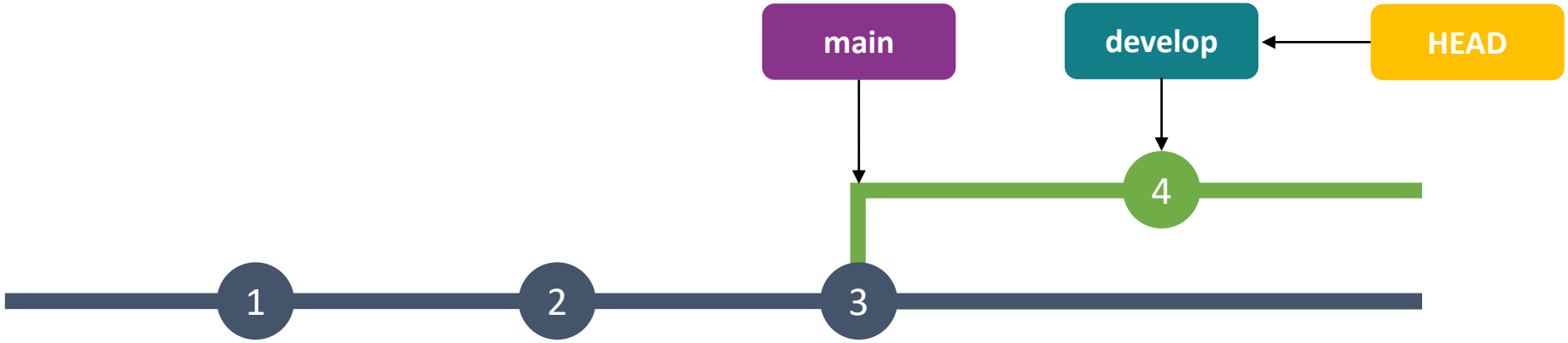


Scenario 1 – Fast Forward Merge

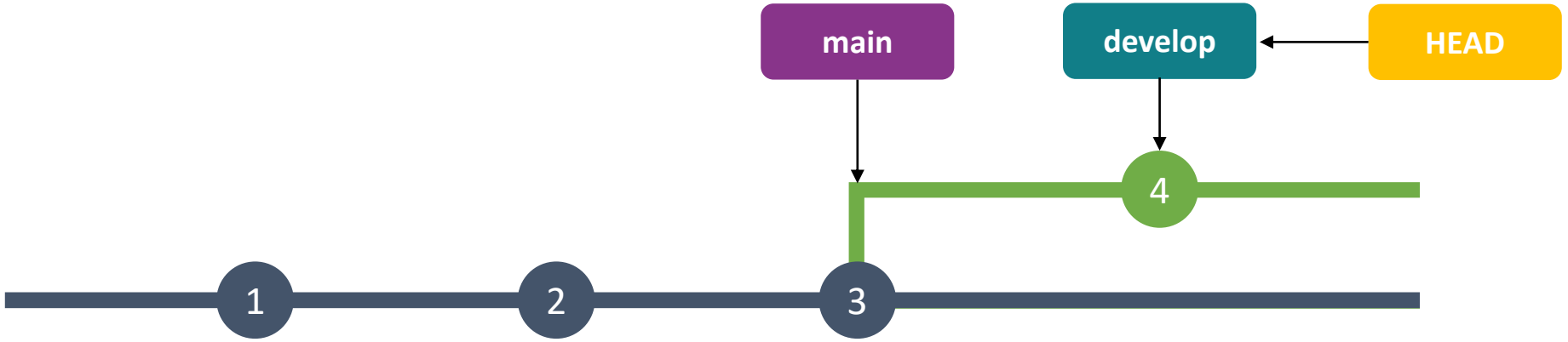


Let us practice

Scenario 2 – Fast Forward Merge



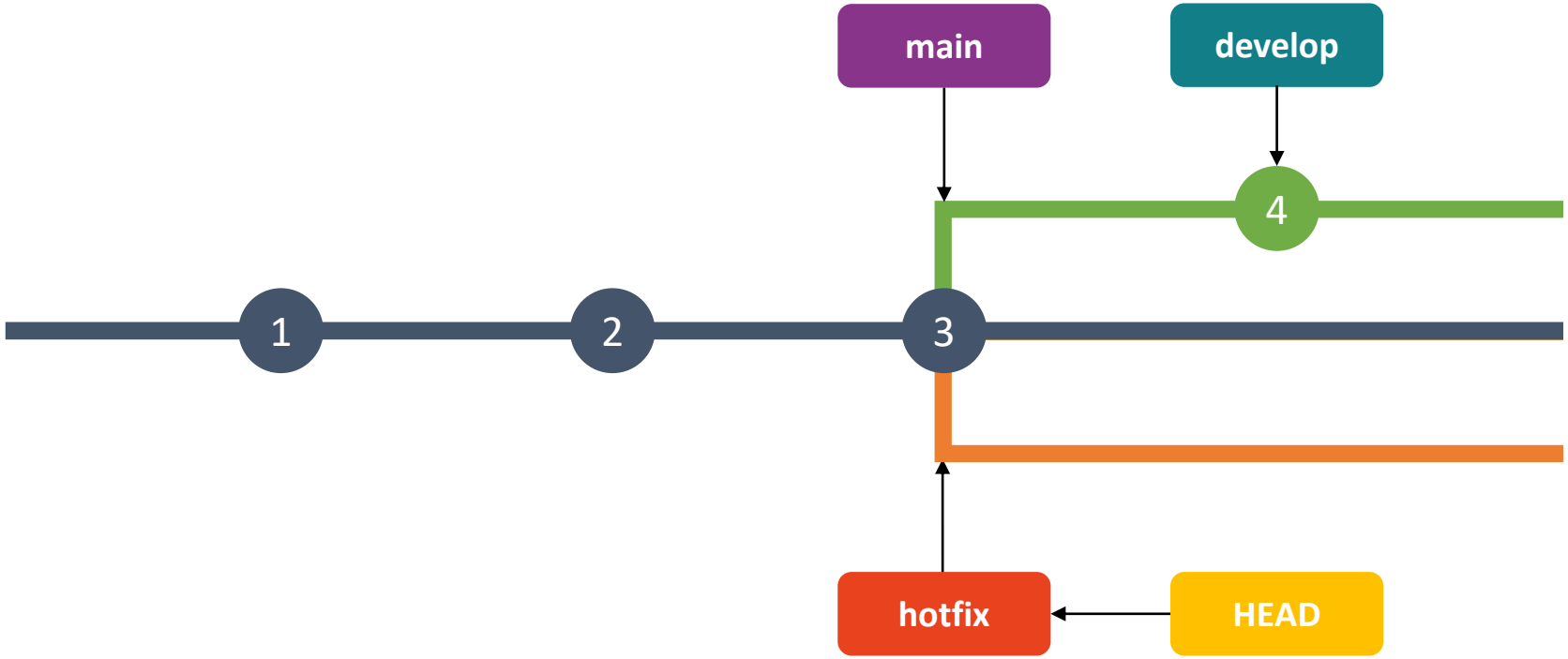
Scenario 2 – Fast Forward Merge



2.23

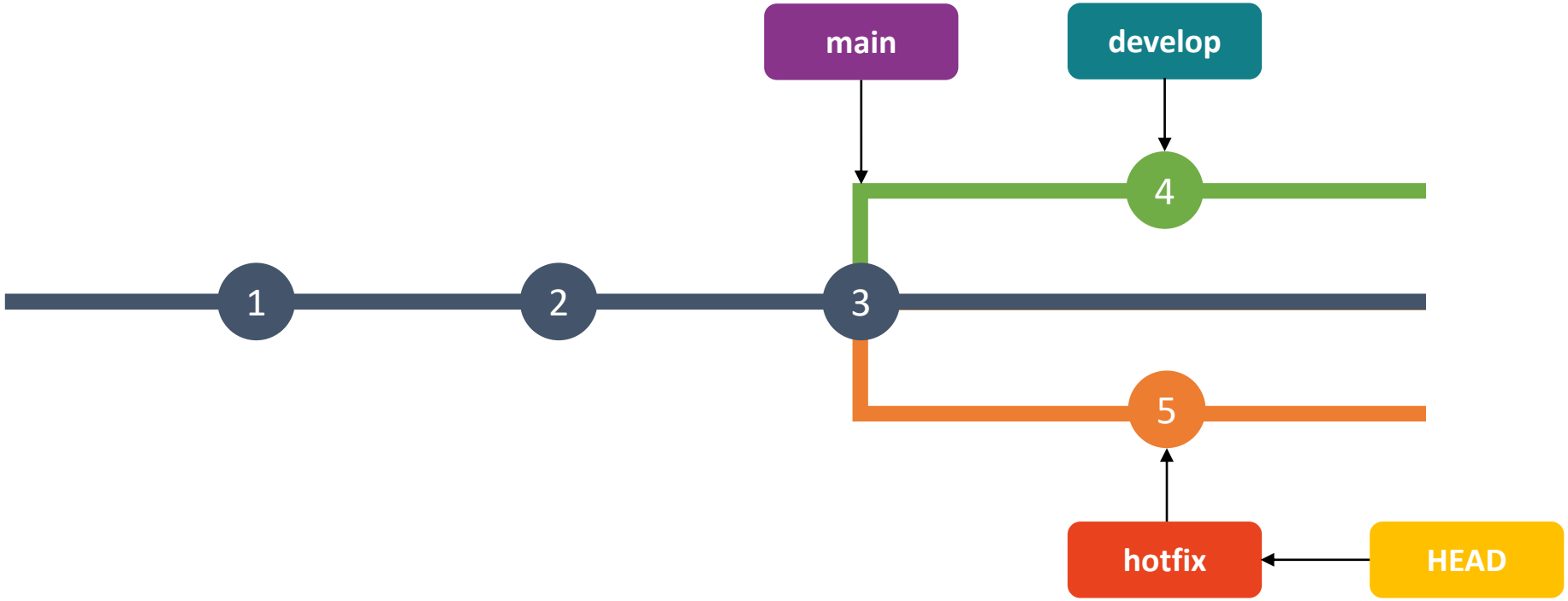
```
$ git switch -C hotfix
```

Scenario 2 – Fast Forward Merge

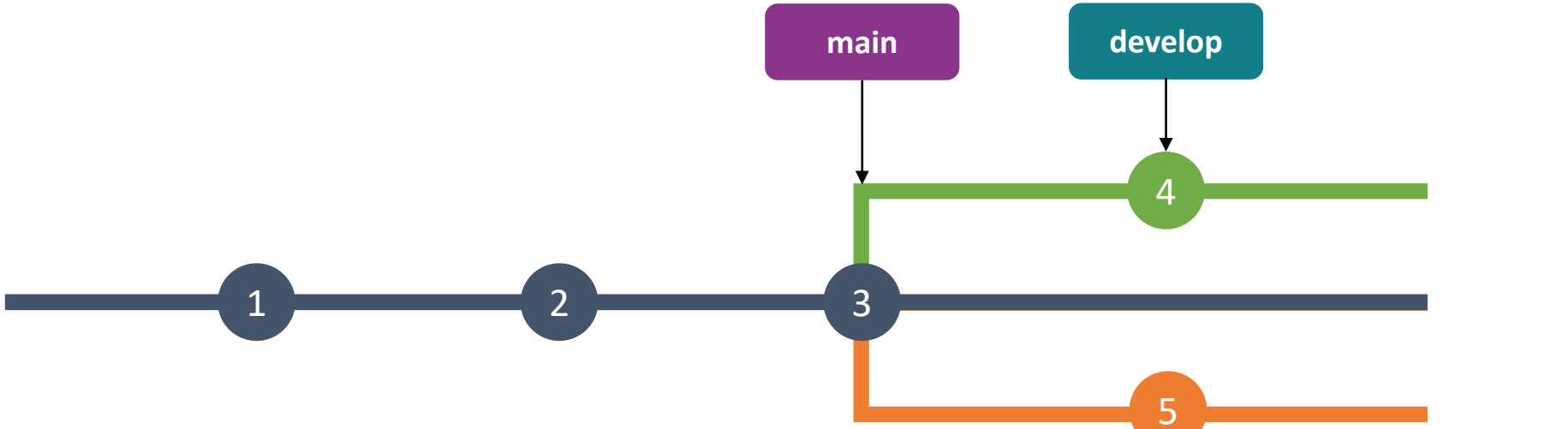


Scenario 2 – Fast Forward Merge

New commit in hotfix



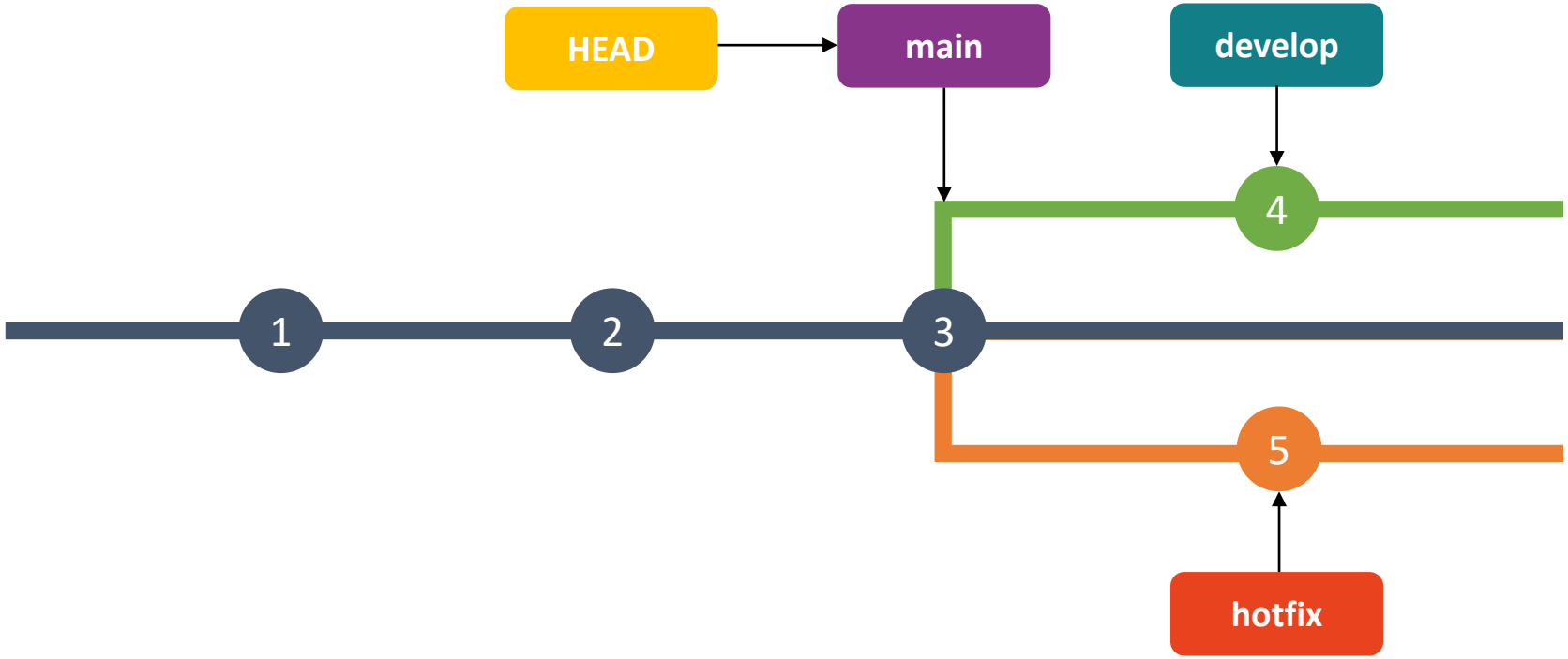
Scenario 2 – Fast Forward Merge



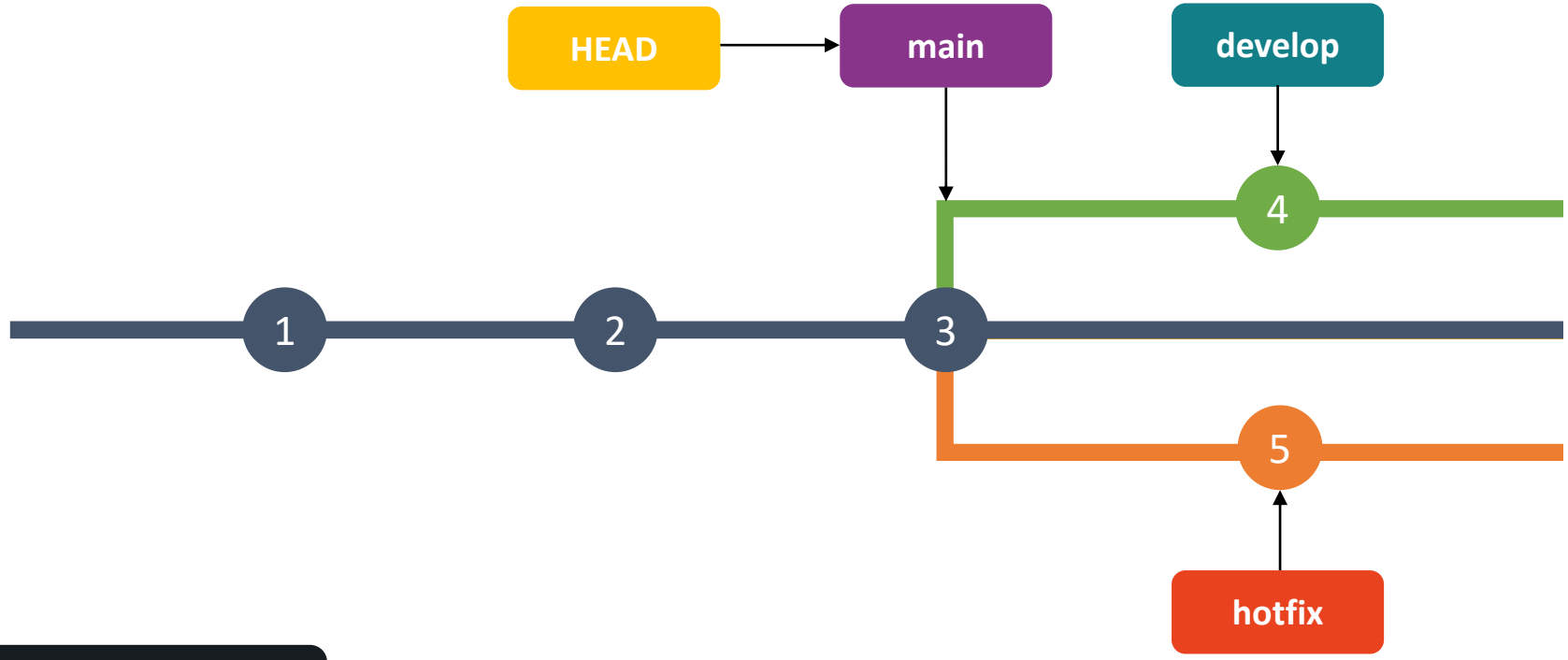
2.23

```
$ git switch main
```

Scenario 2 – Fast Forward Merge

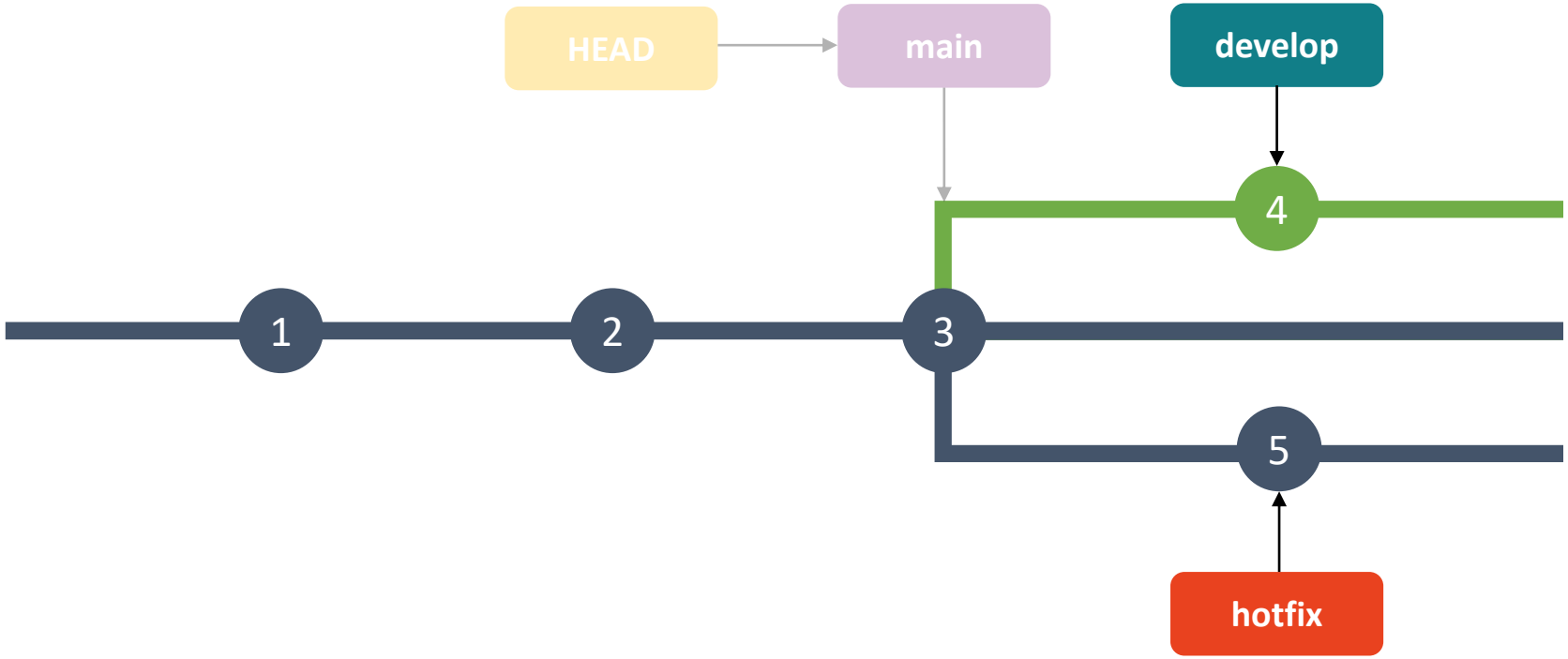


Scenario 2 – Fast Forward Merge

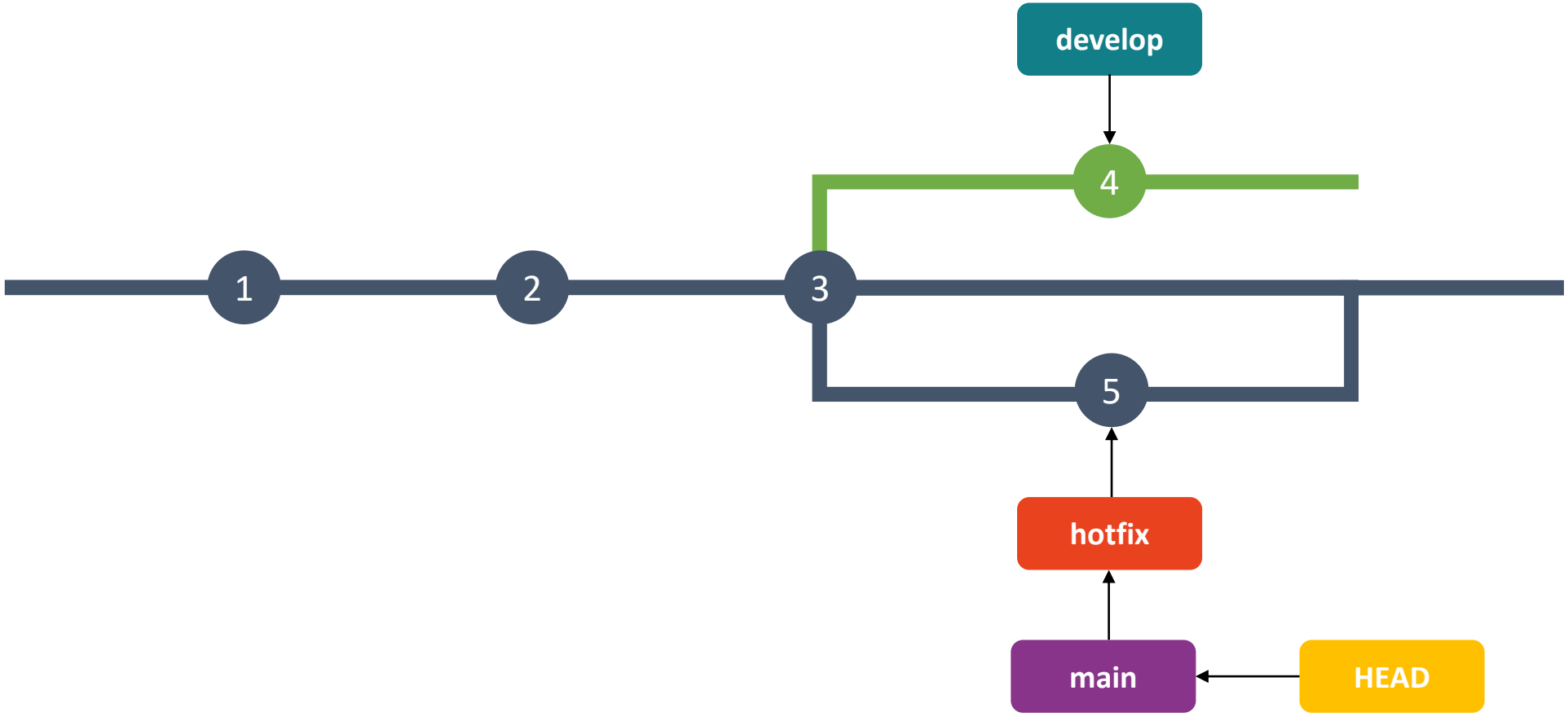



```
$ git merge hotfix
```

Scenario 2 – Fast Forward Merge



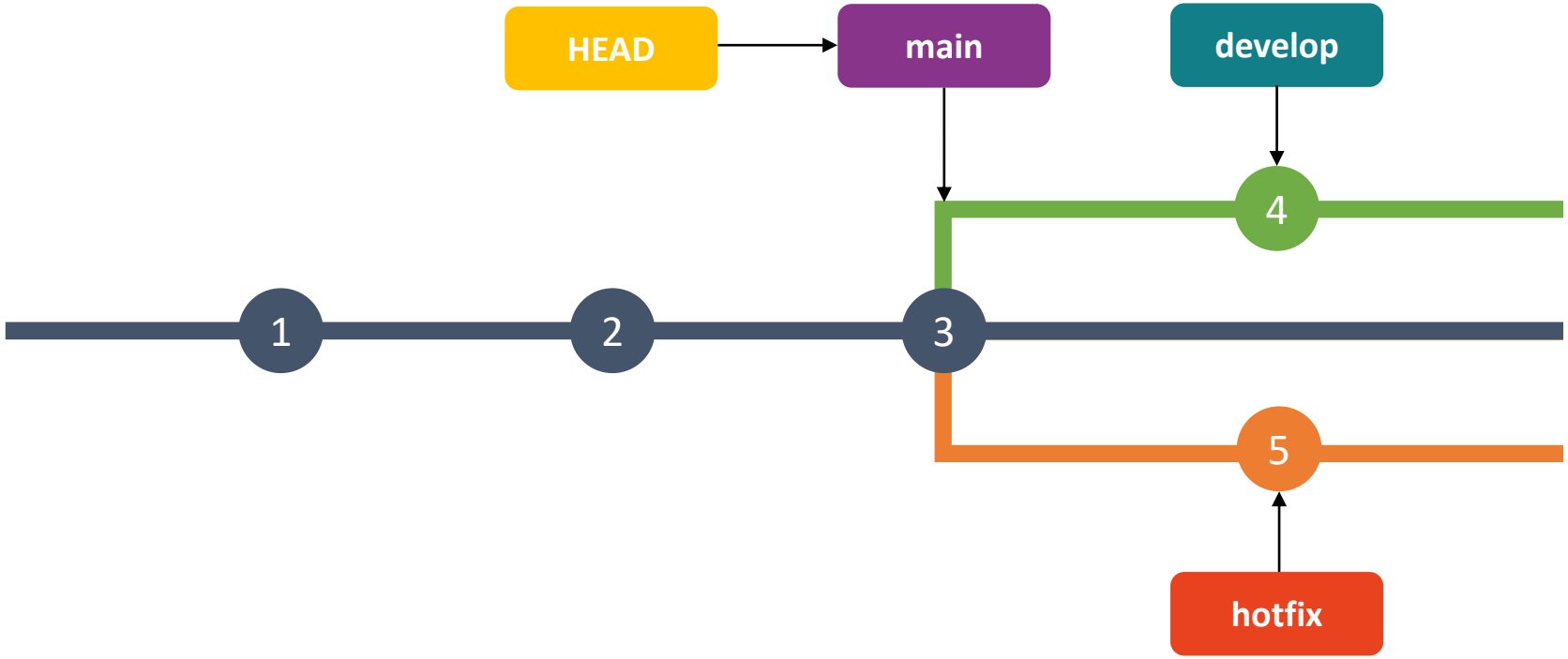
Scenario 2 – Fast Forward Merge



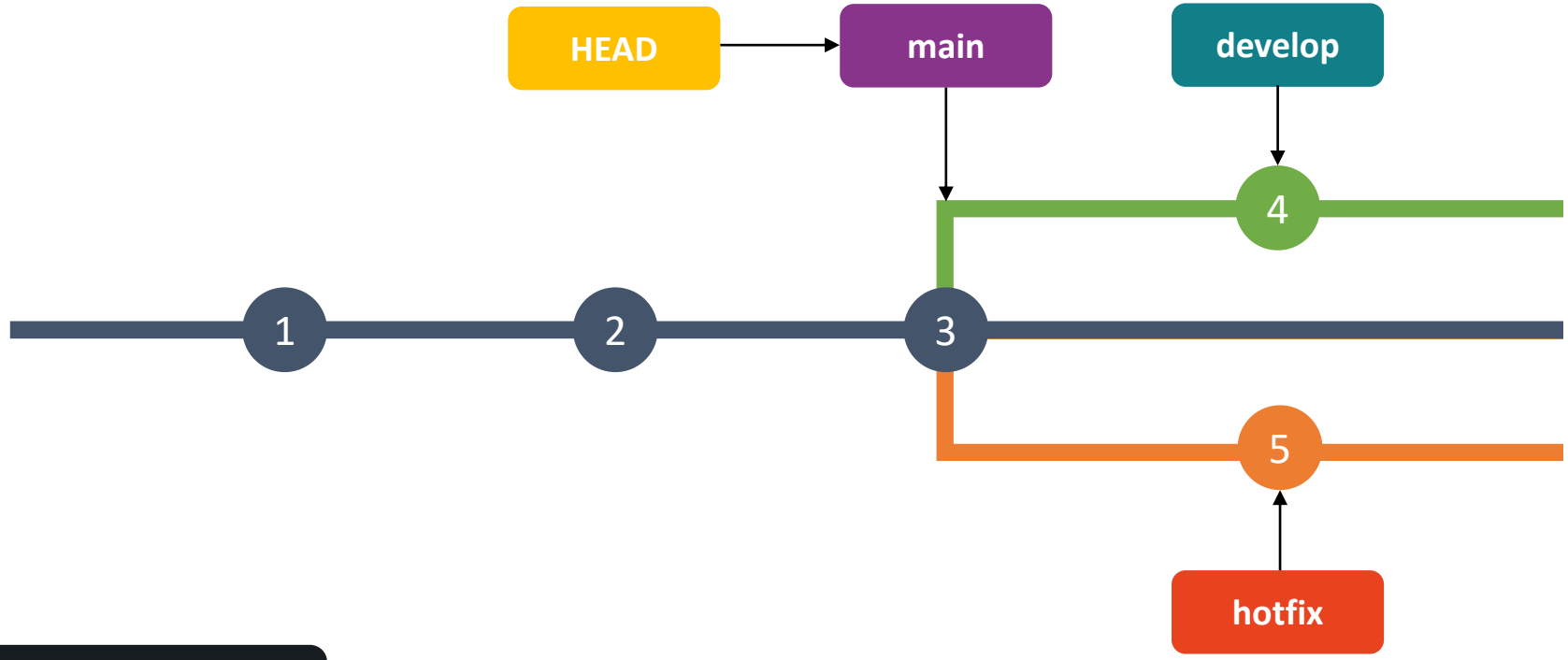
Let us practice

Merge Commits – Without conflicts

Scenario 1 – Initial State

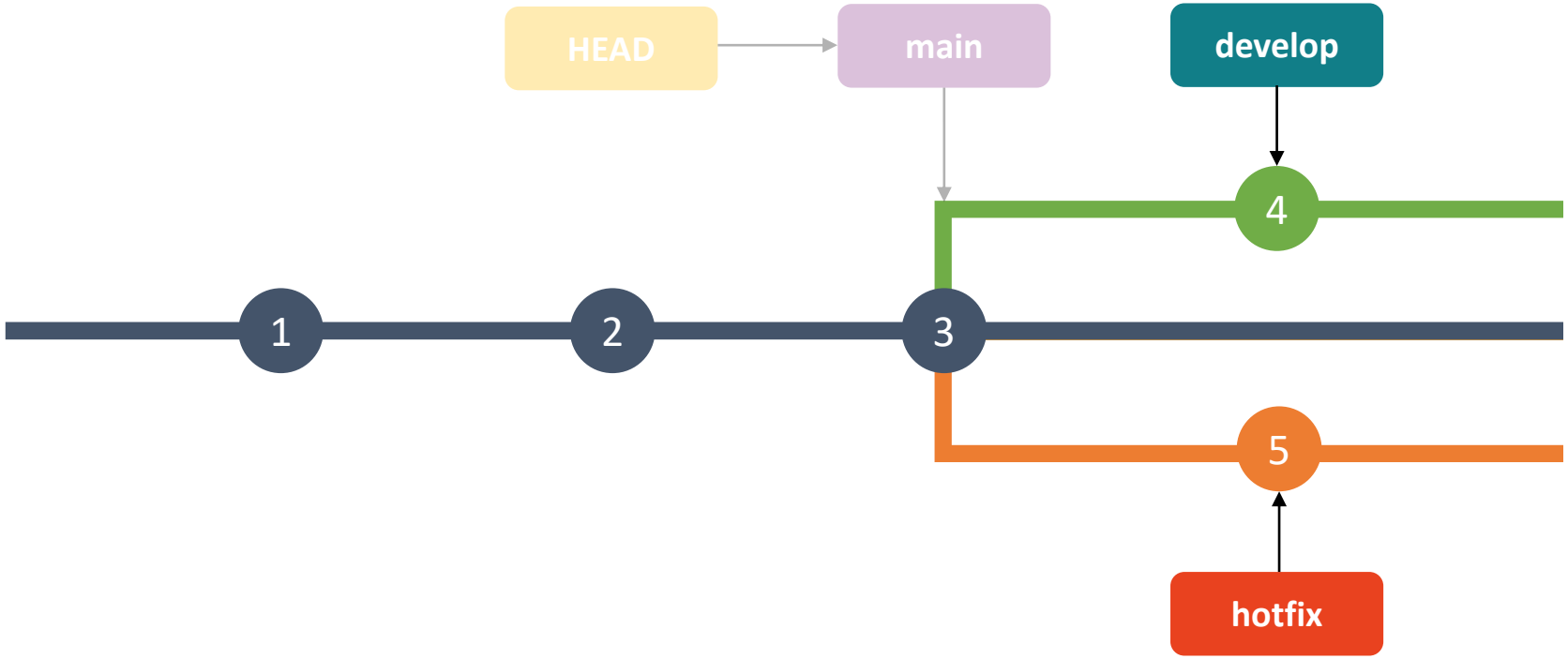


Scenario 1 – Merge hotfix With main

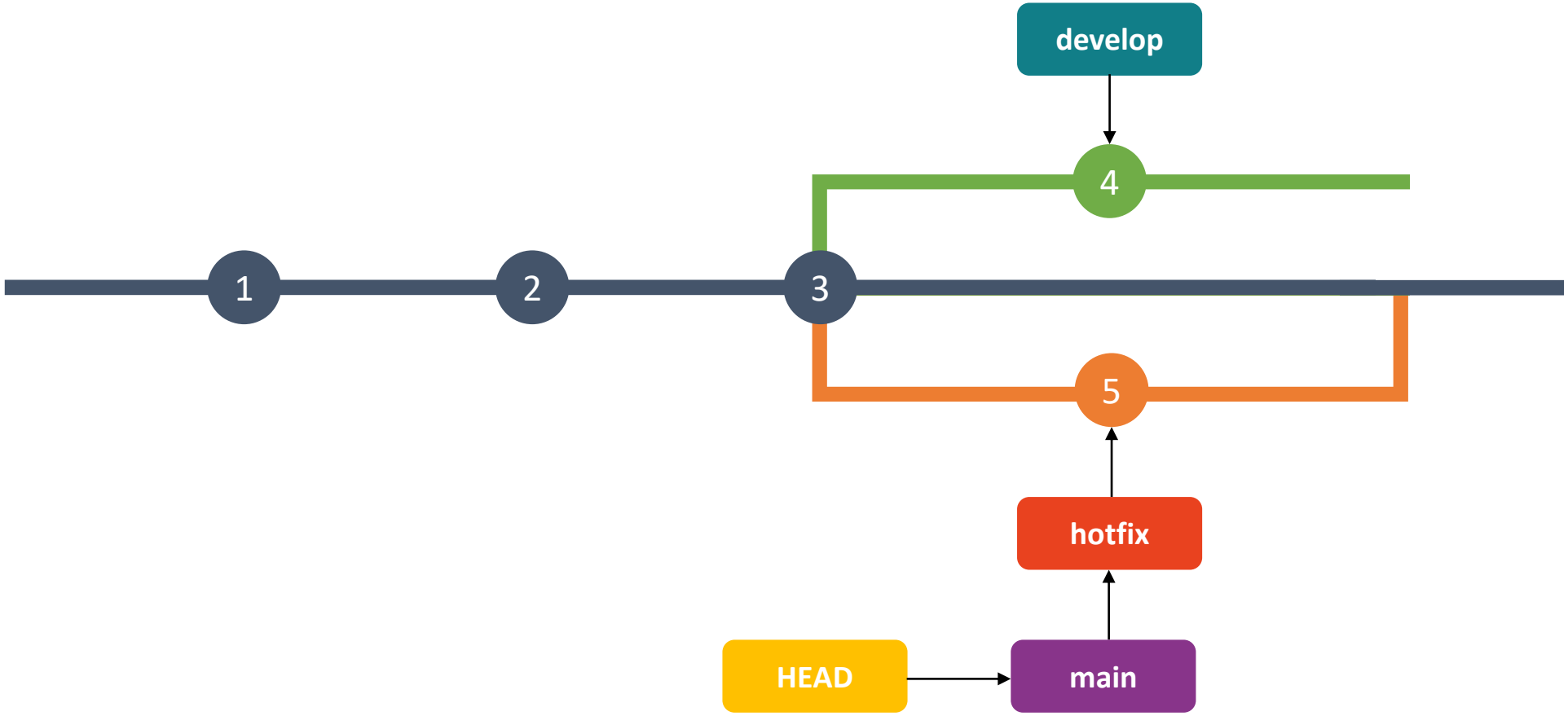



```
$ git merge hotfix
```

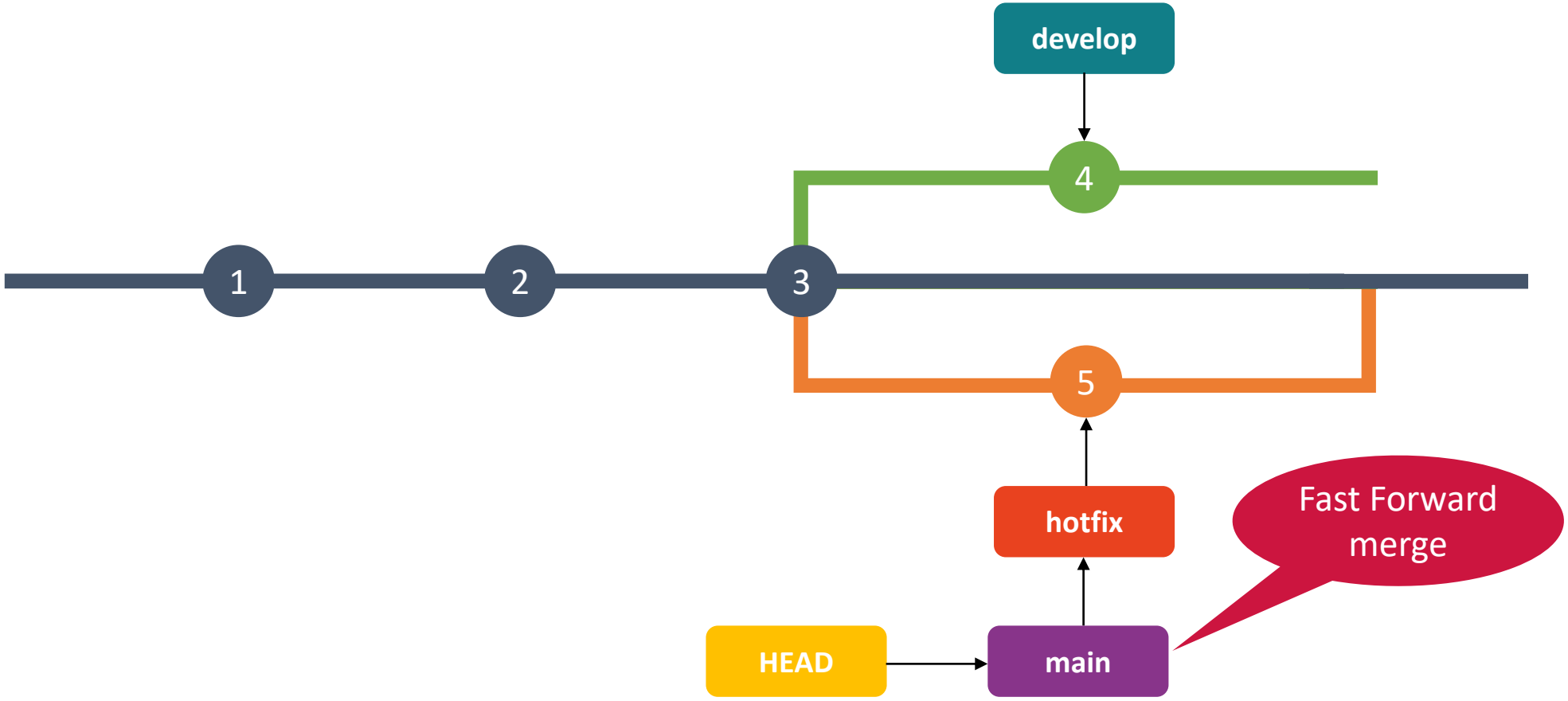
Scenario 1 – Merge hotfix With main



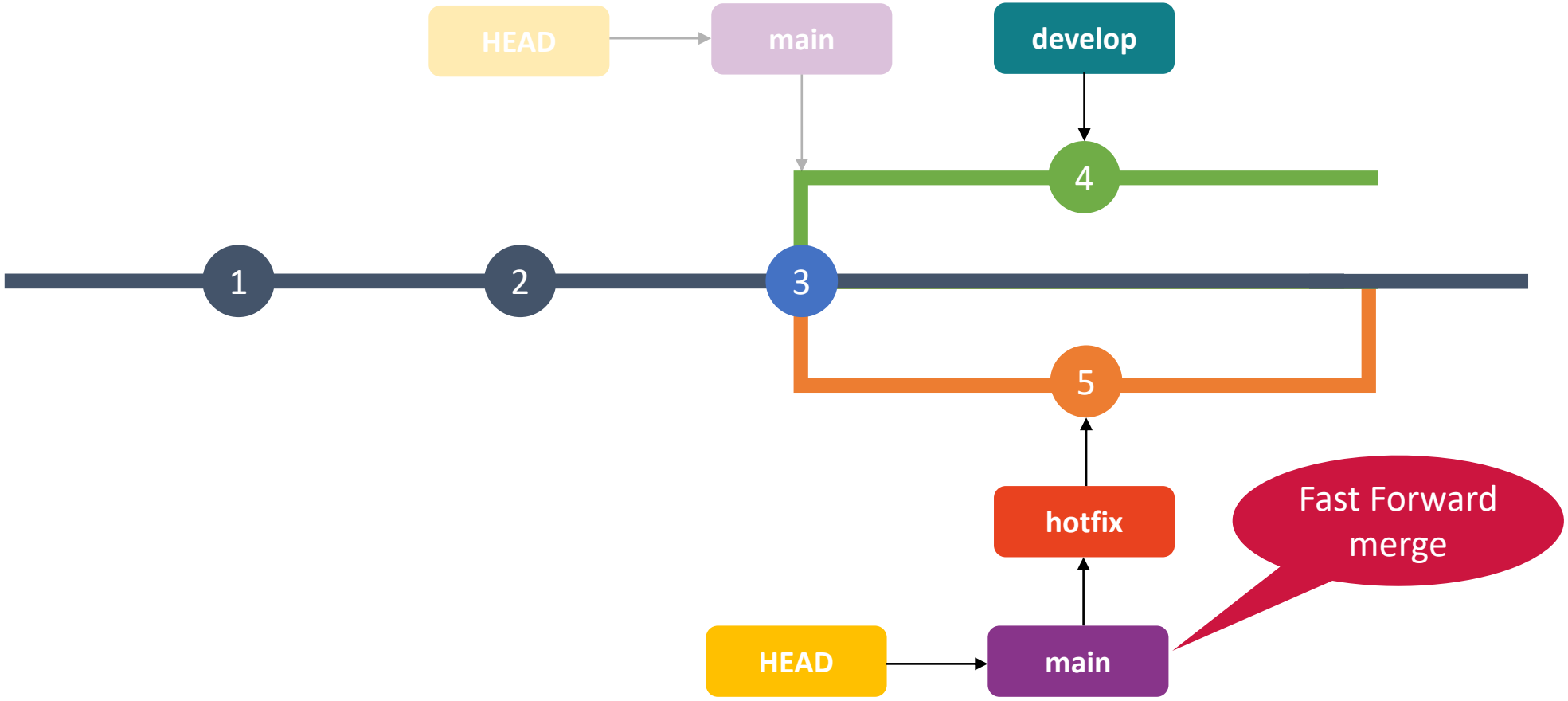
Scenario 1 – After Merge hotfix With main



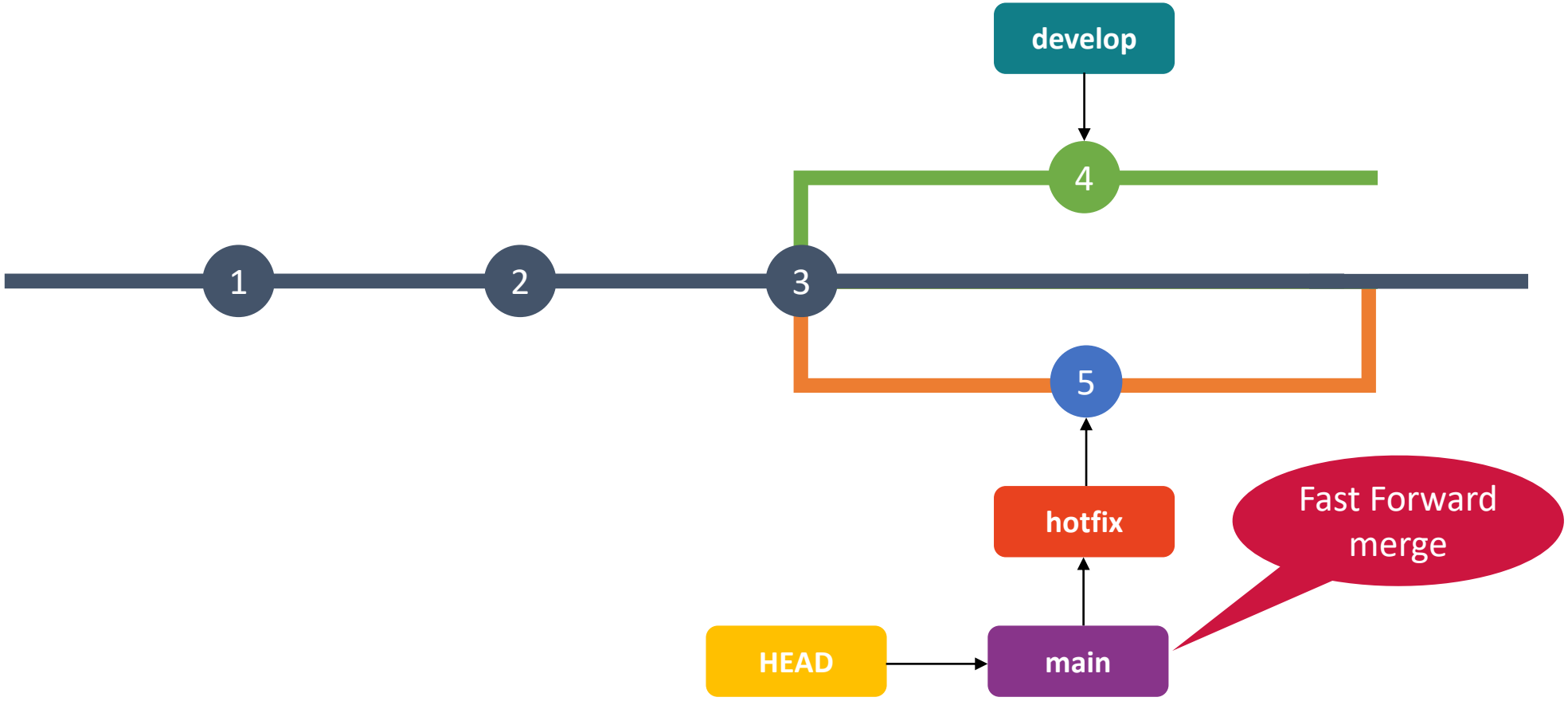
Scenario 1 – After Merge hotfix With main



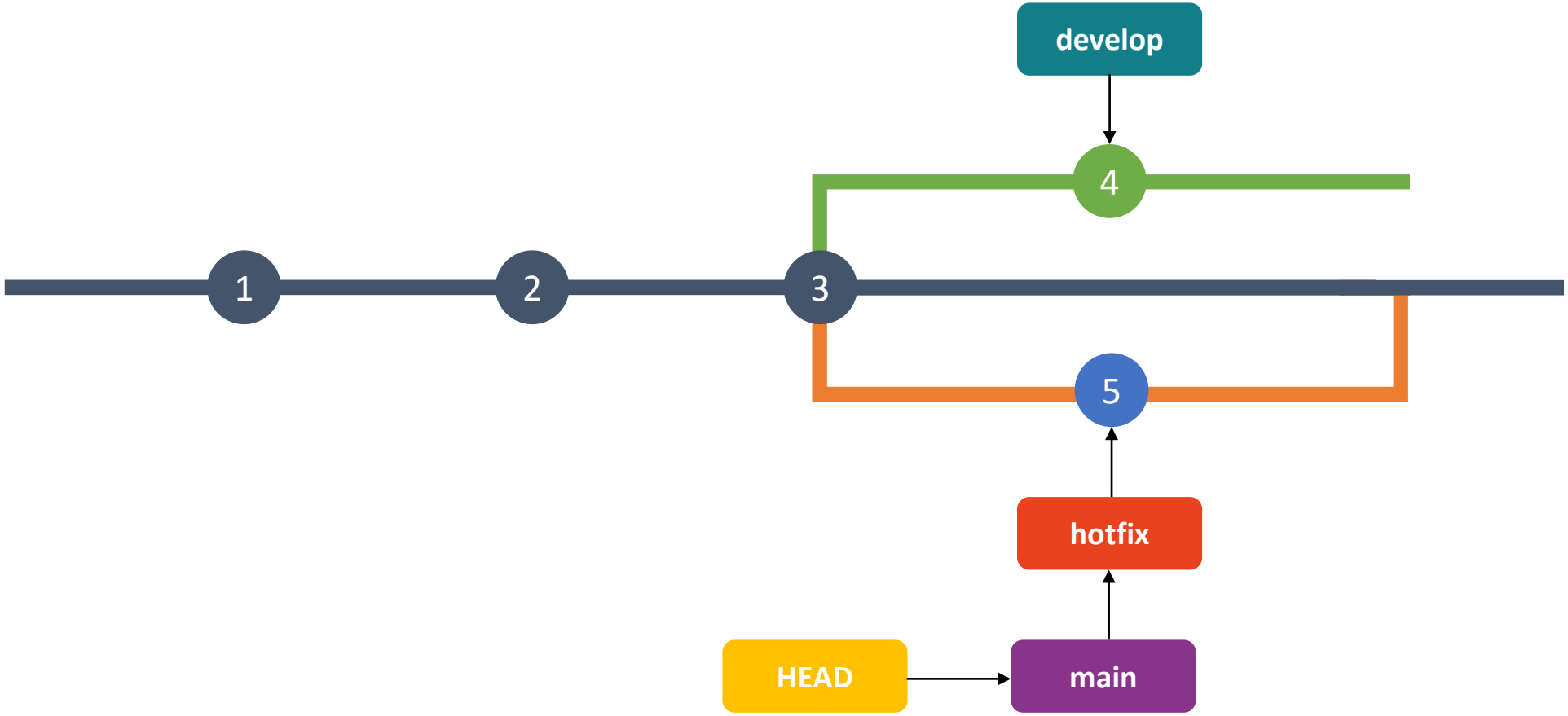
Scenario 1 – After Merge hotfix With main



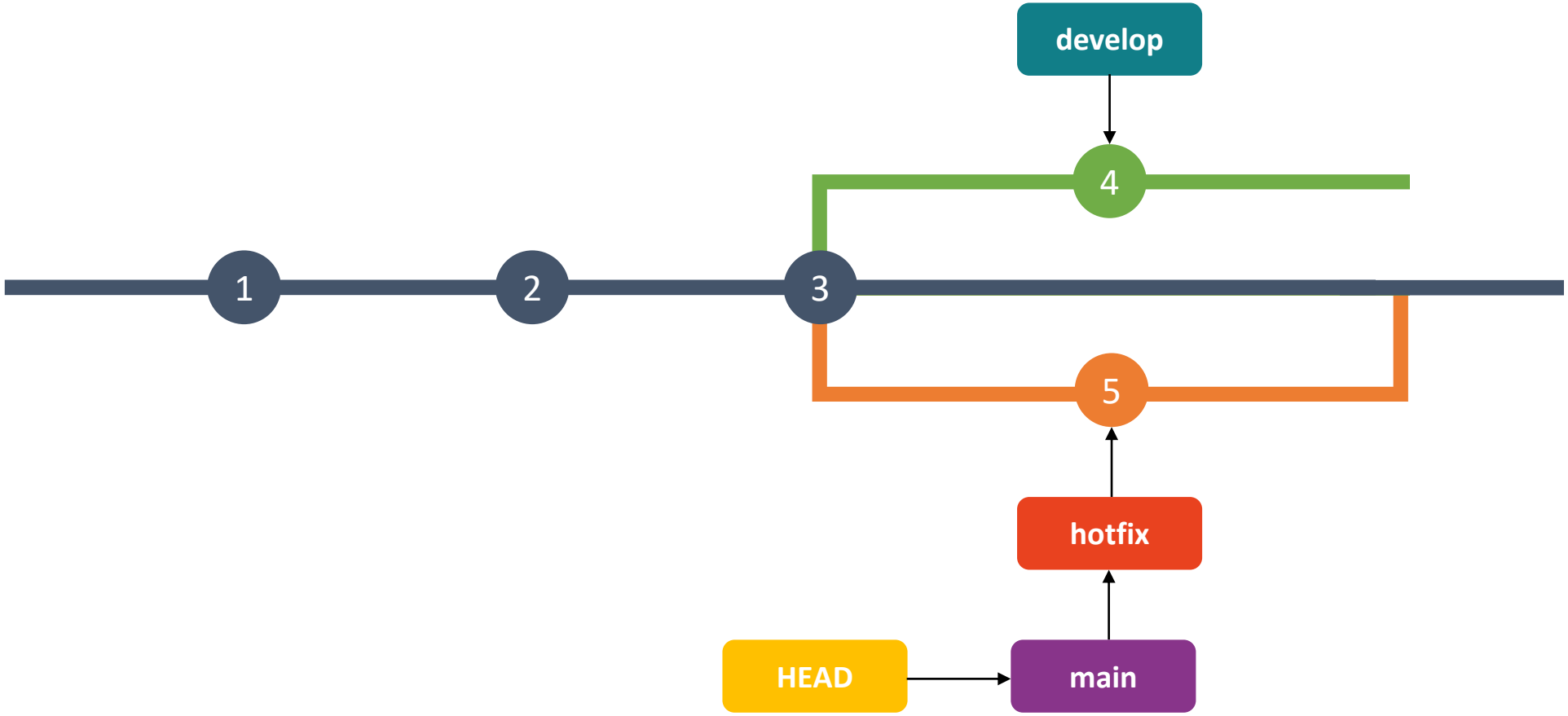
Scenario 1 – After Merge hotfix With main



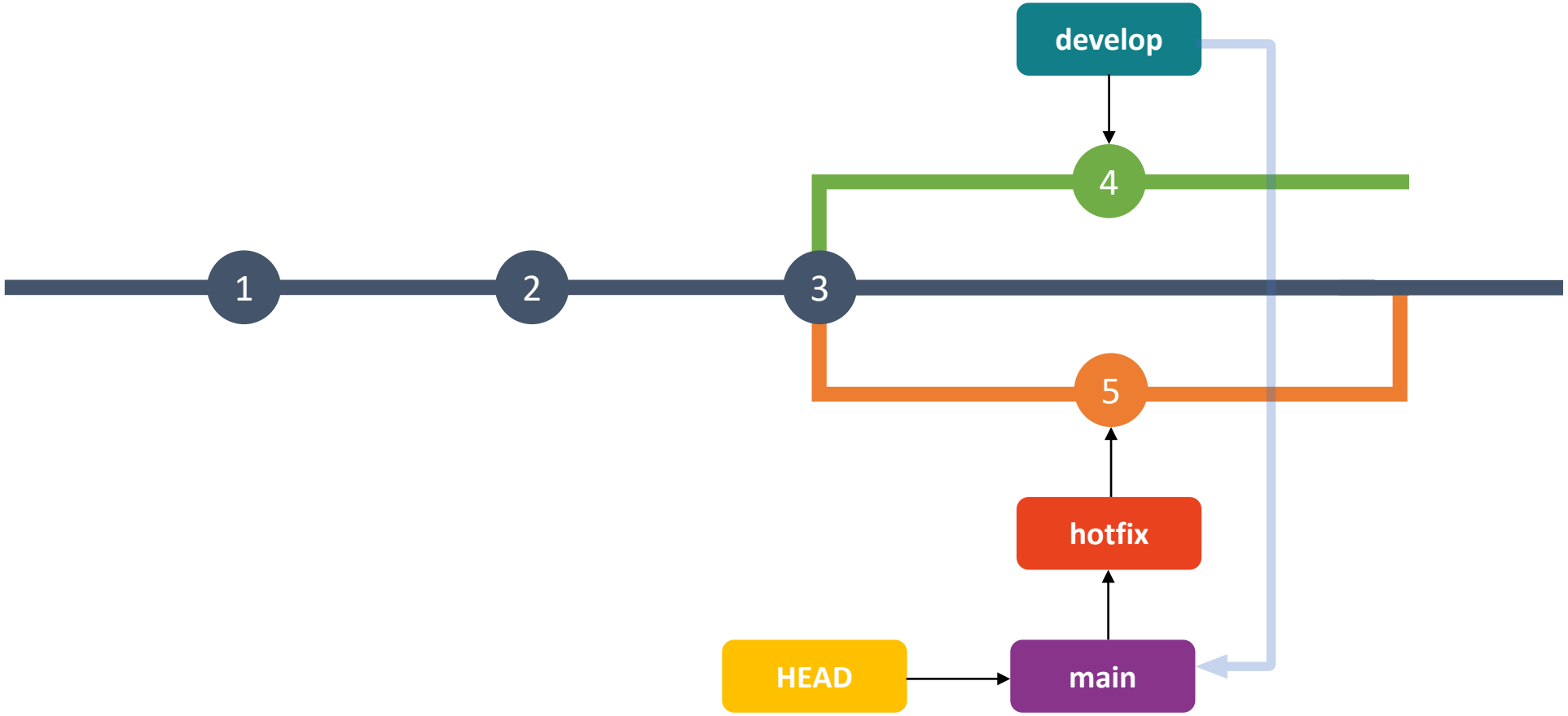
Scenario 1 – After Merge hotfix With main



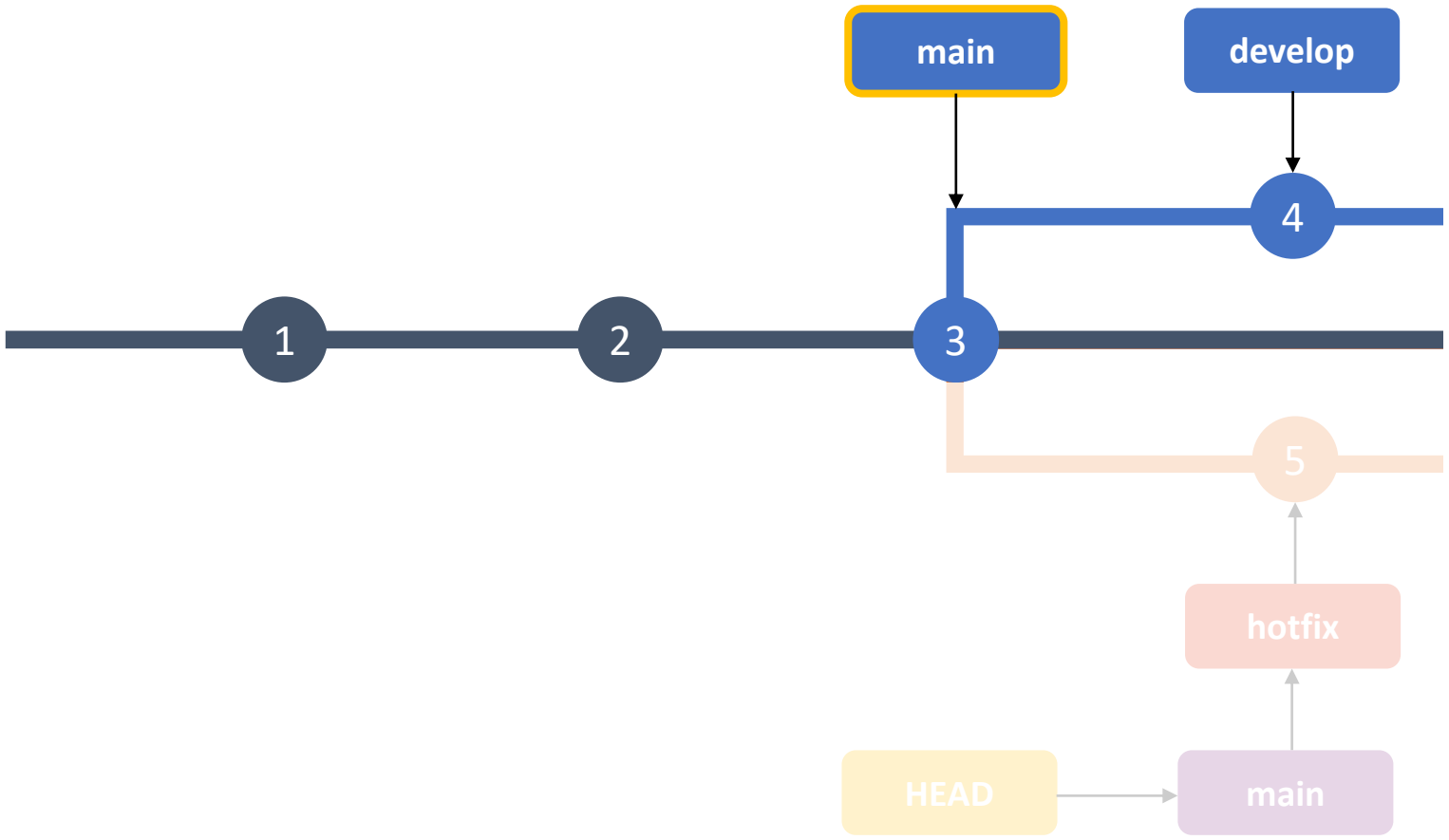
Scenario 1 – Next Step



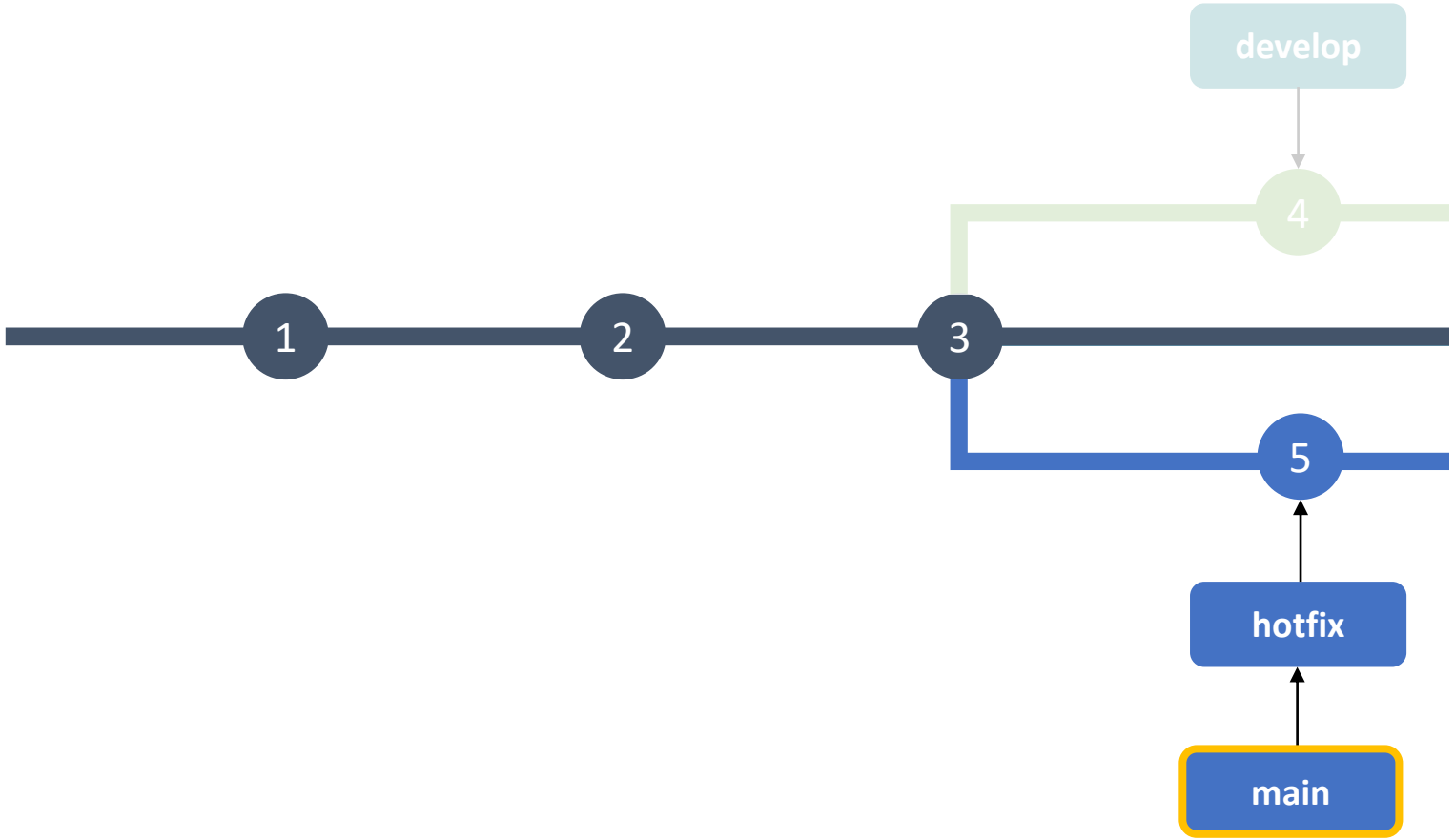
Scenario 1 – Goal: To Merge develop With main



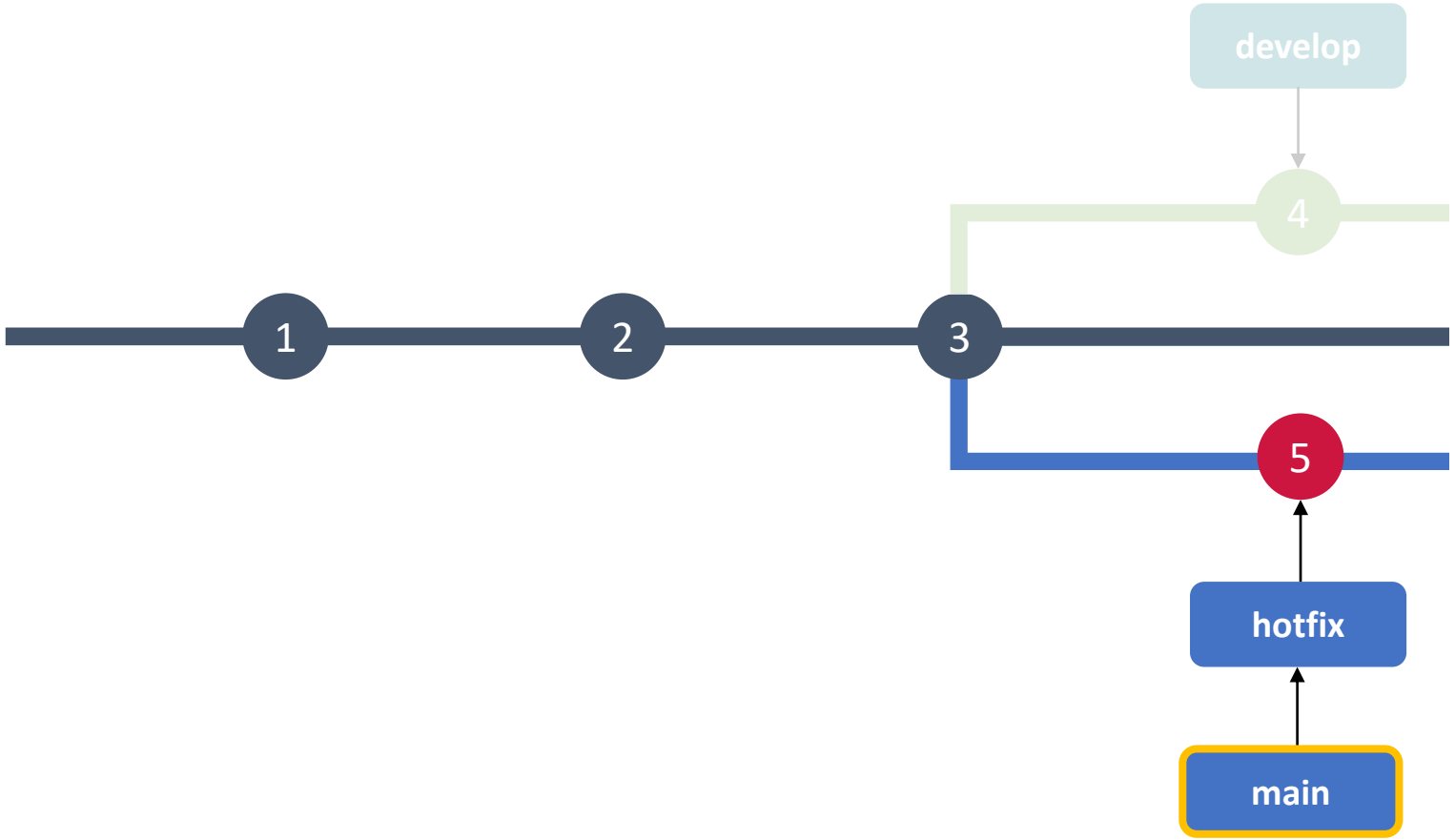
Scenario 1 – The Issue: Previous State



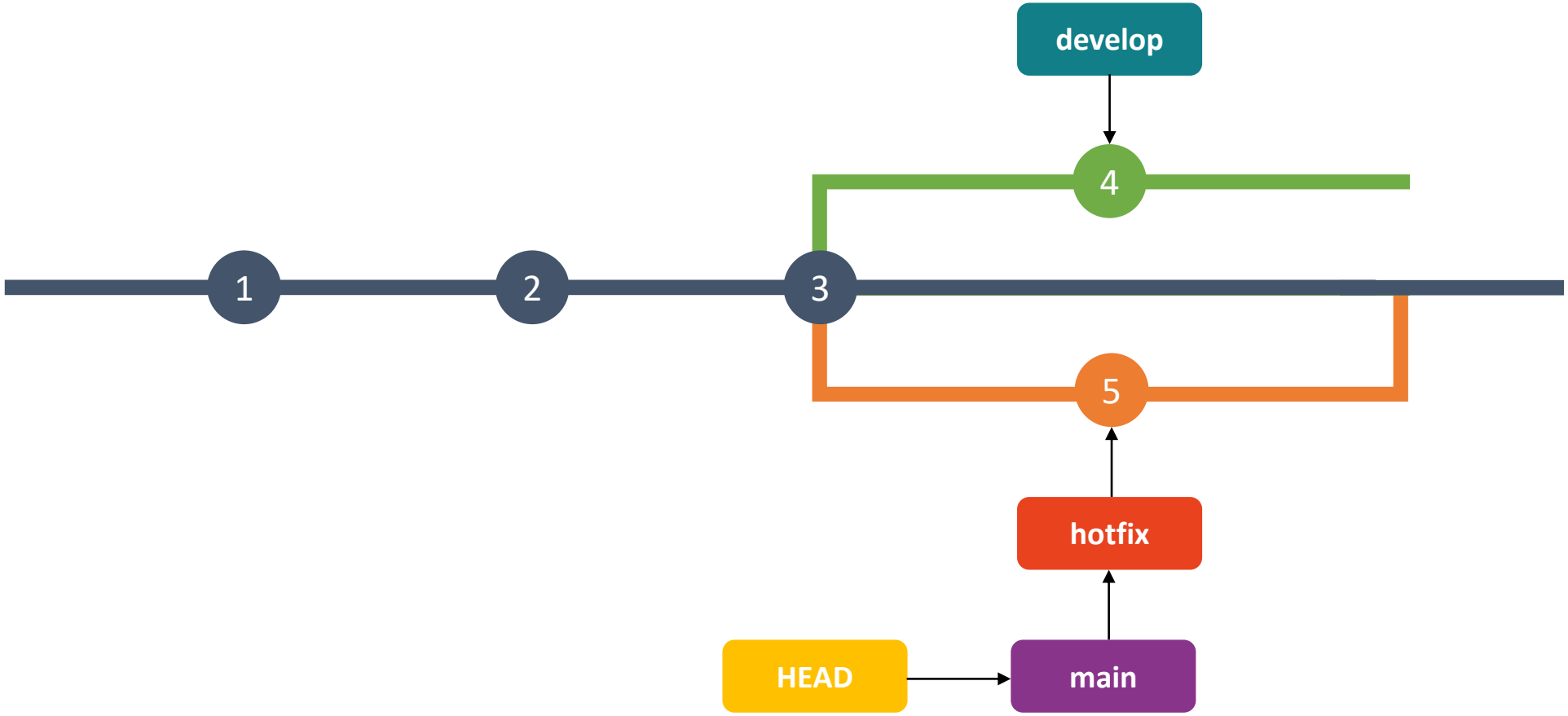
Scenario 1 – The Issue: Current State



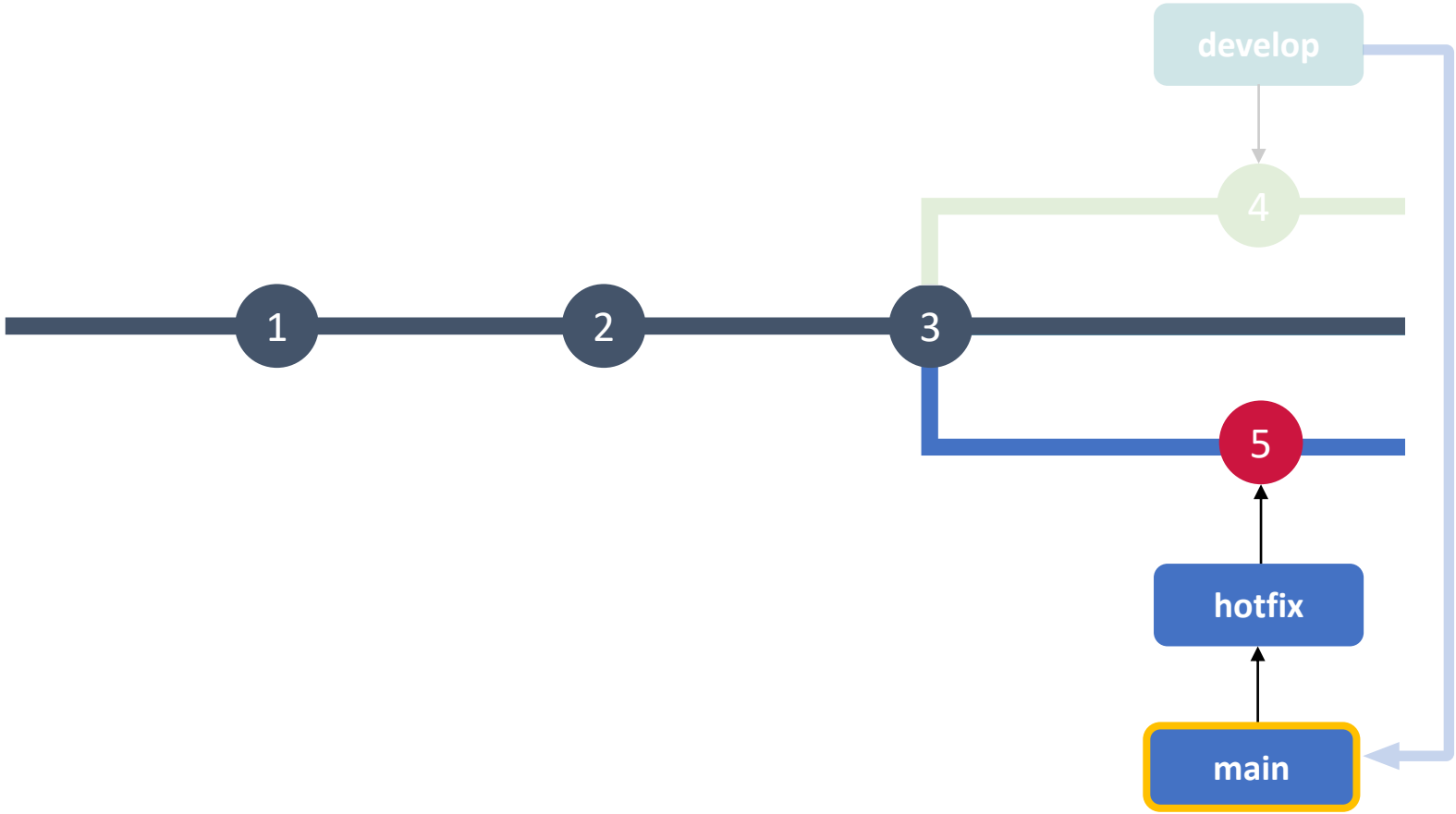
Scenario 1 – The Issue



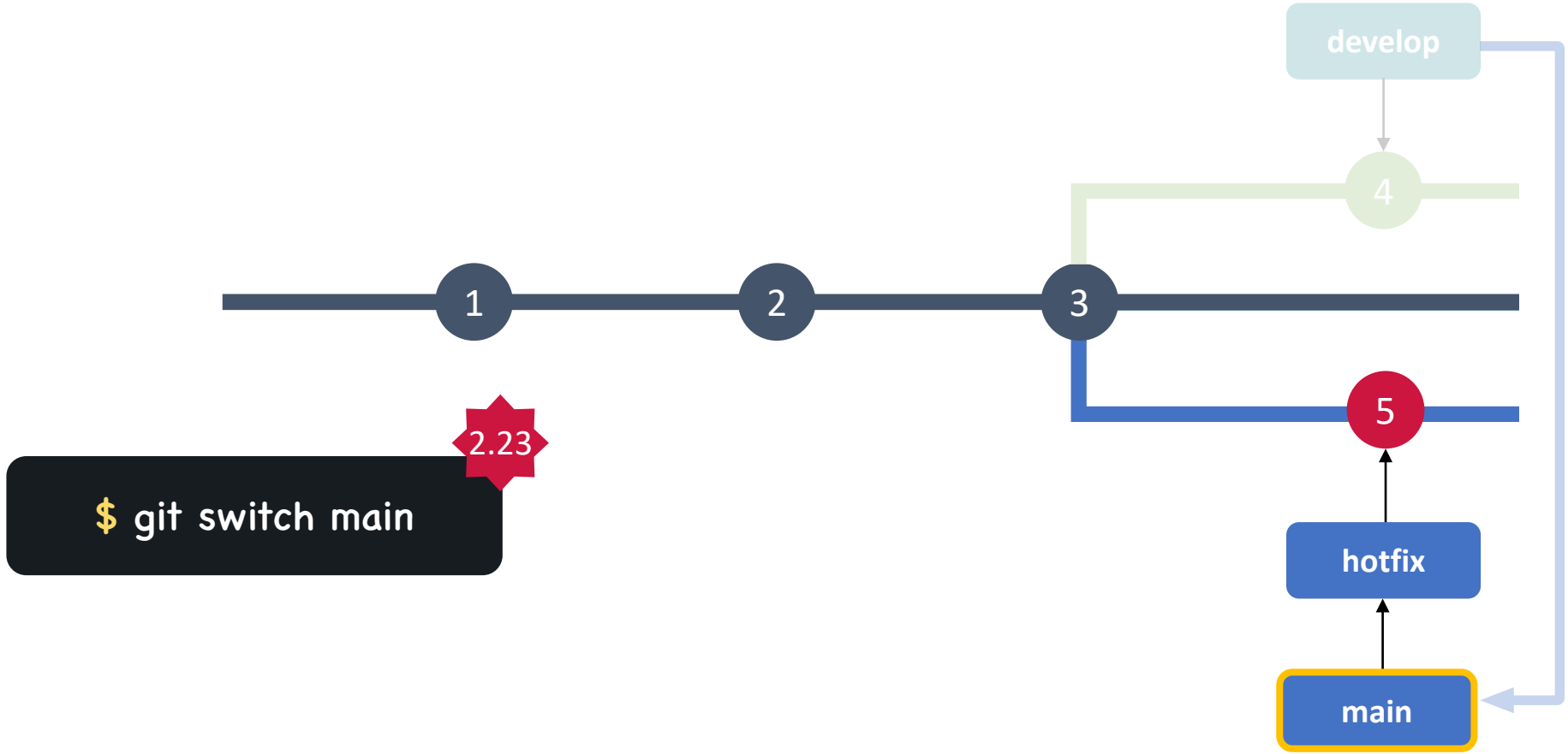
Scenario 1 – Next Step



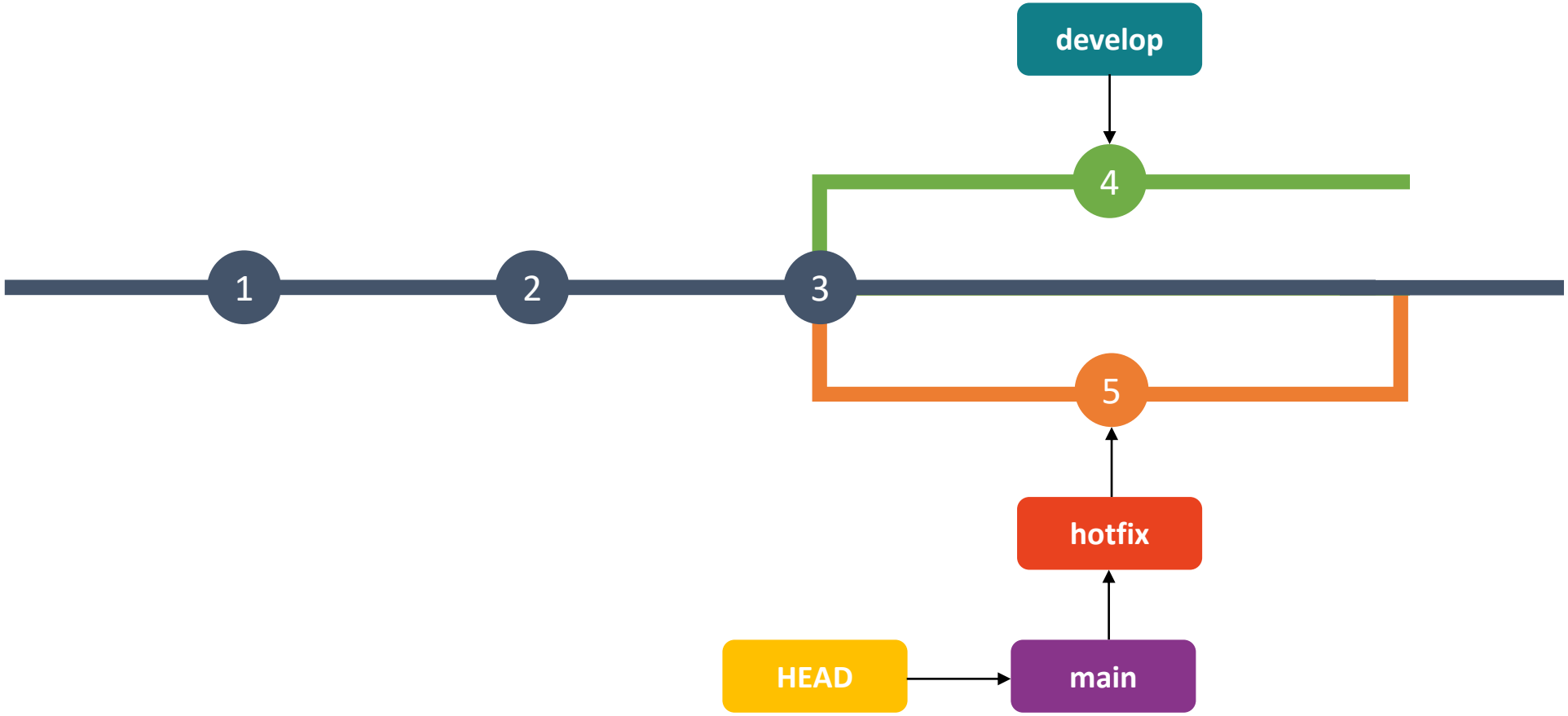
Scenario 1 – Goal: To Merge develop With main



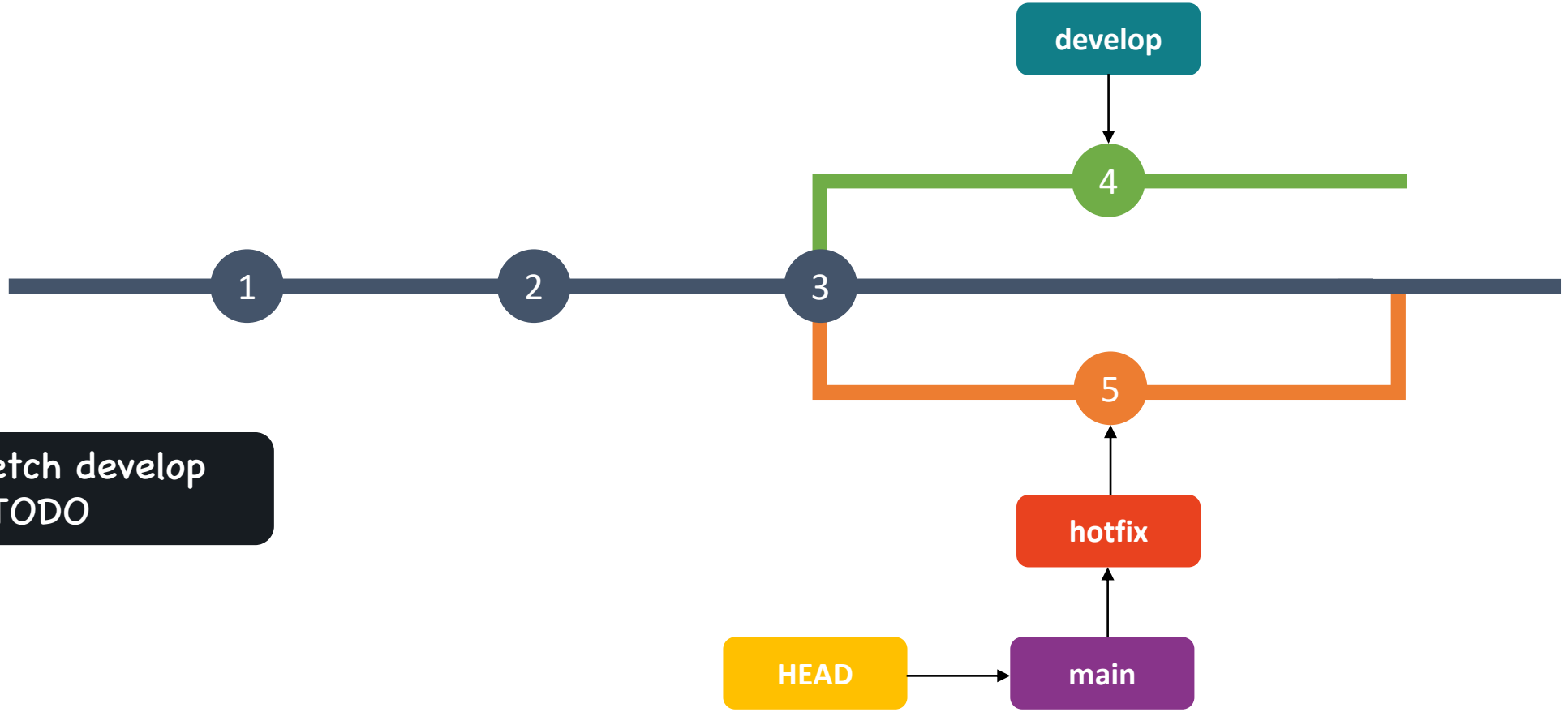
Scenario 1 – Goal: To Merge develop With main



Scenario 1 – Switched To main

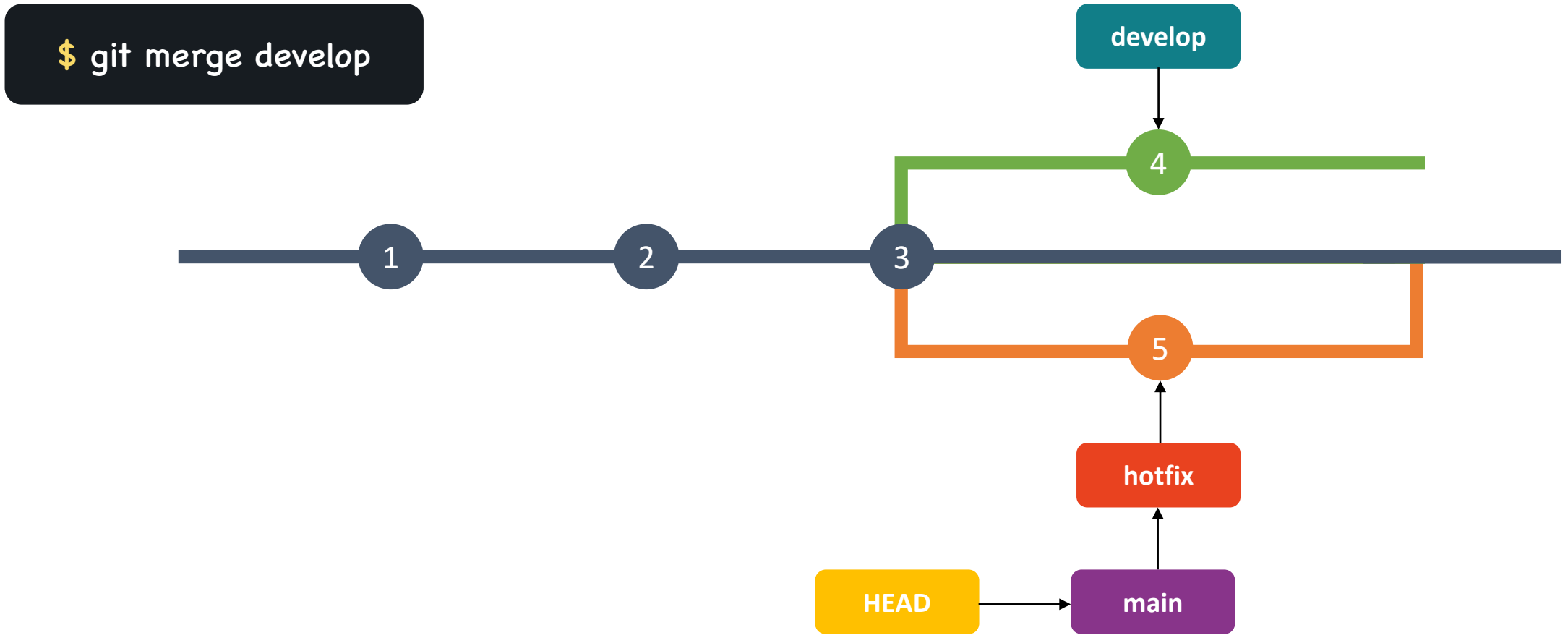


Scenario 1 – Switched To main

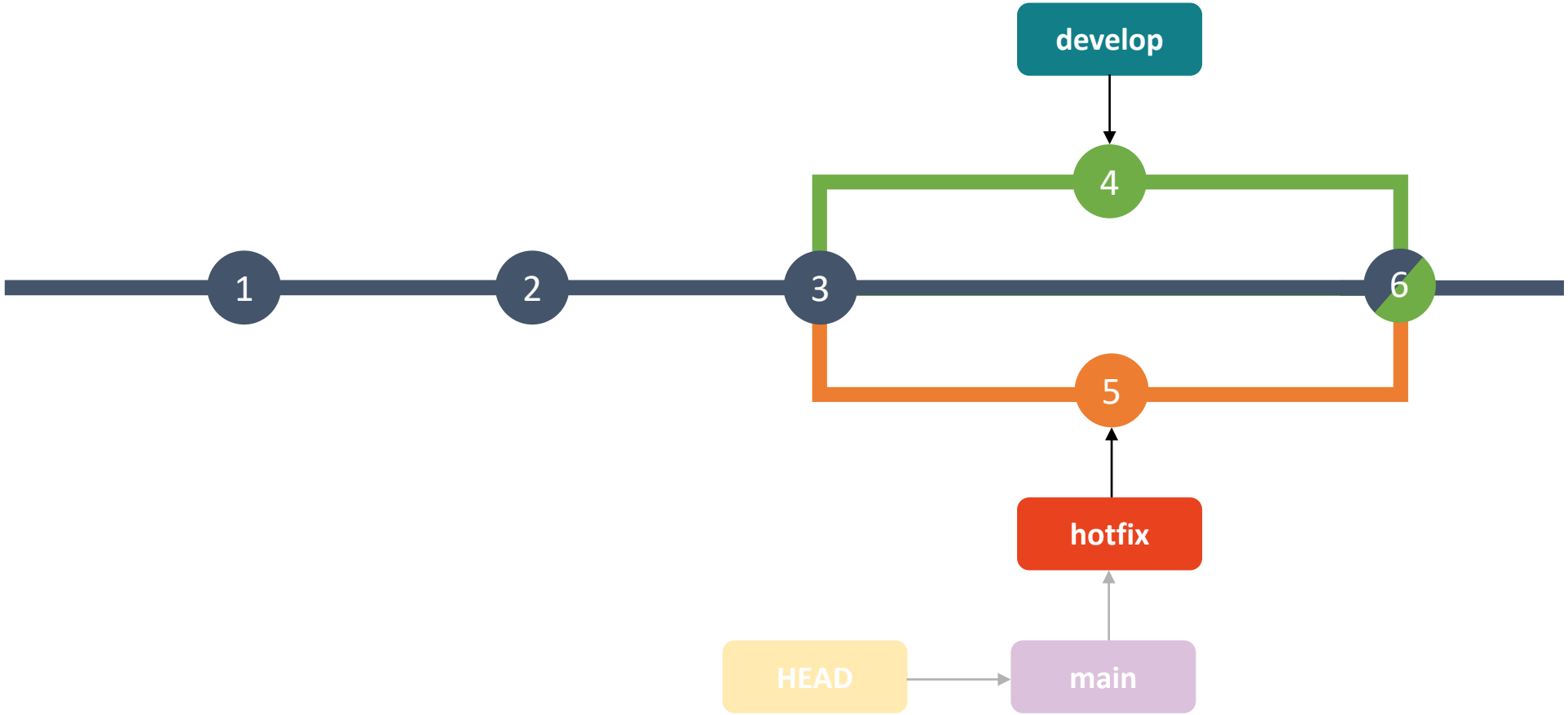



```
$ git fetch develop
TODO
```

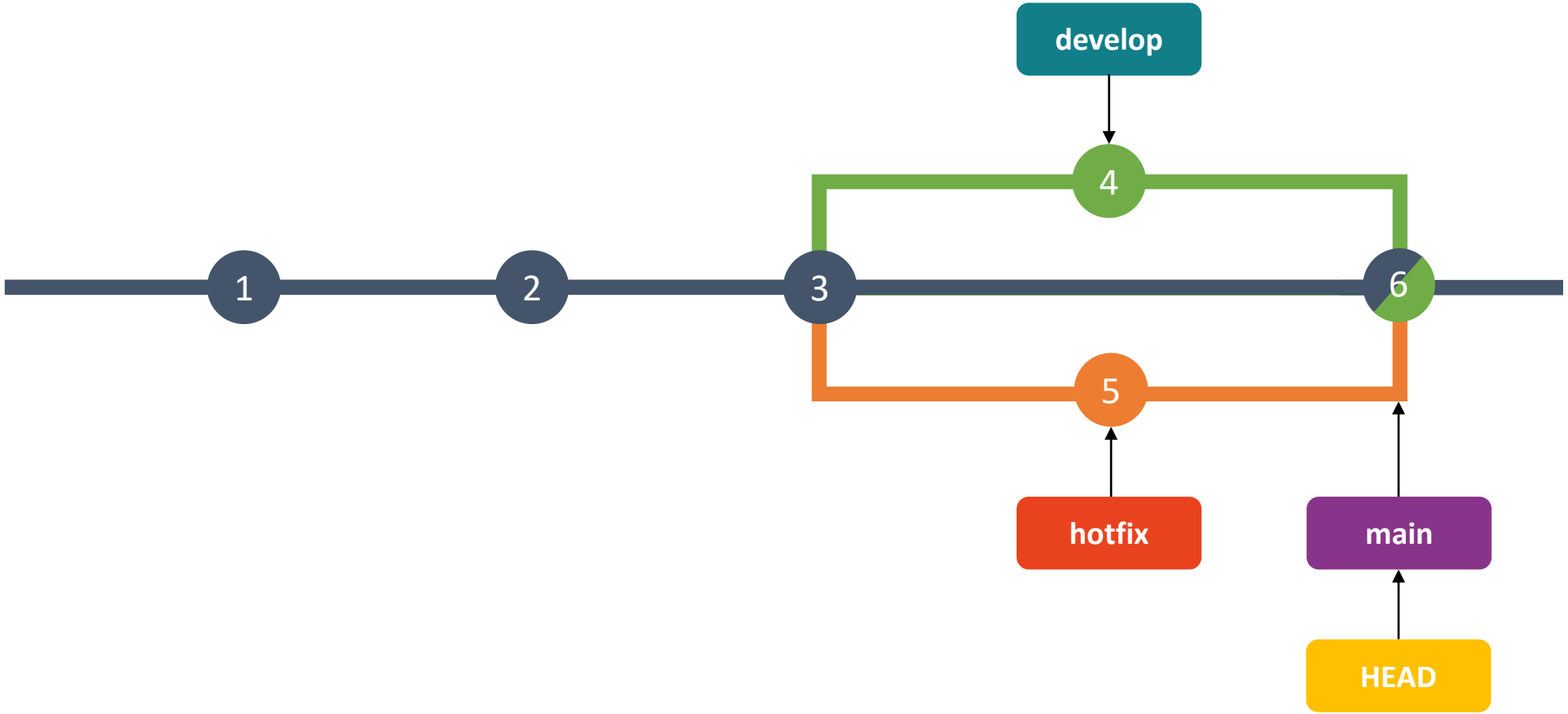
Scenario 1 – Switched To main



Scenario 1 – Merge Commit Created



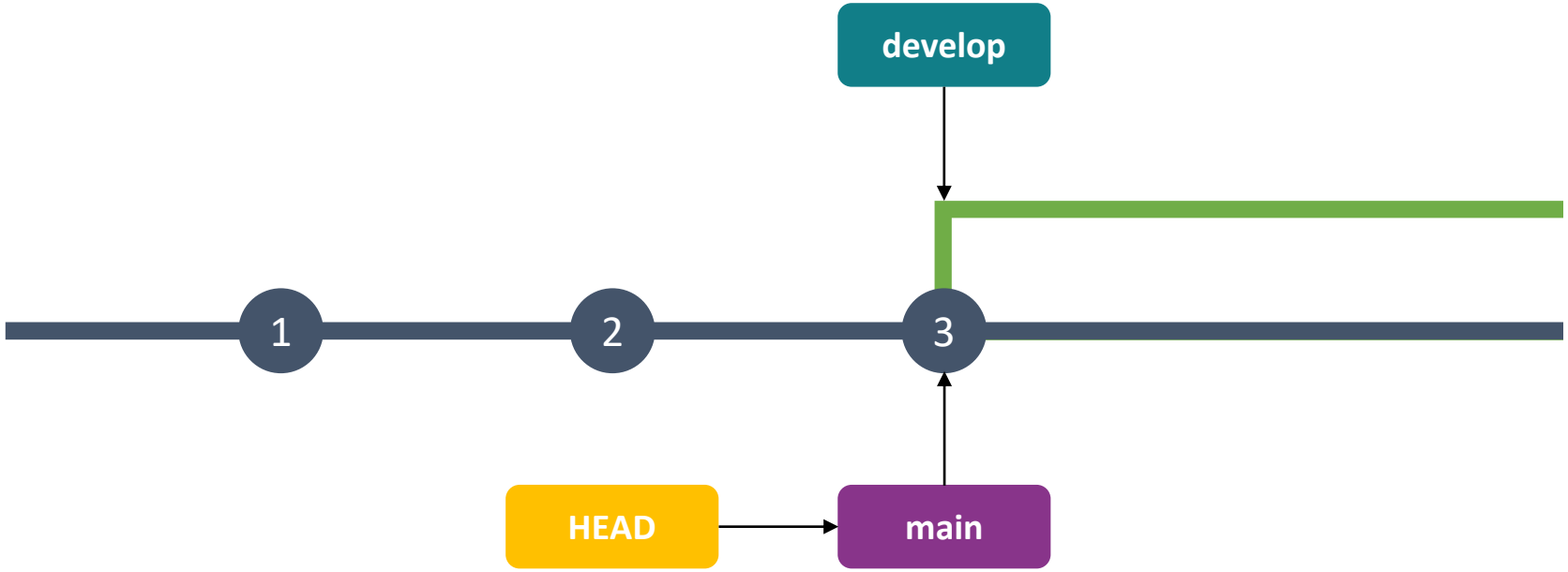
Scenario 1 – Merge Commit Created



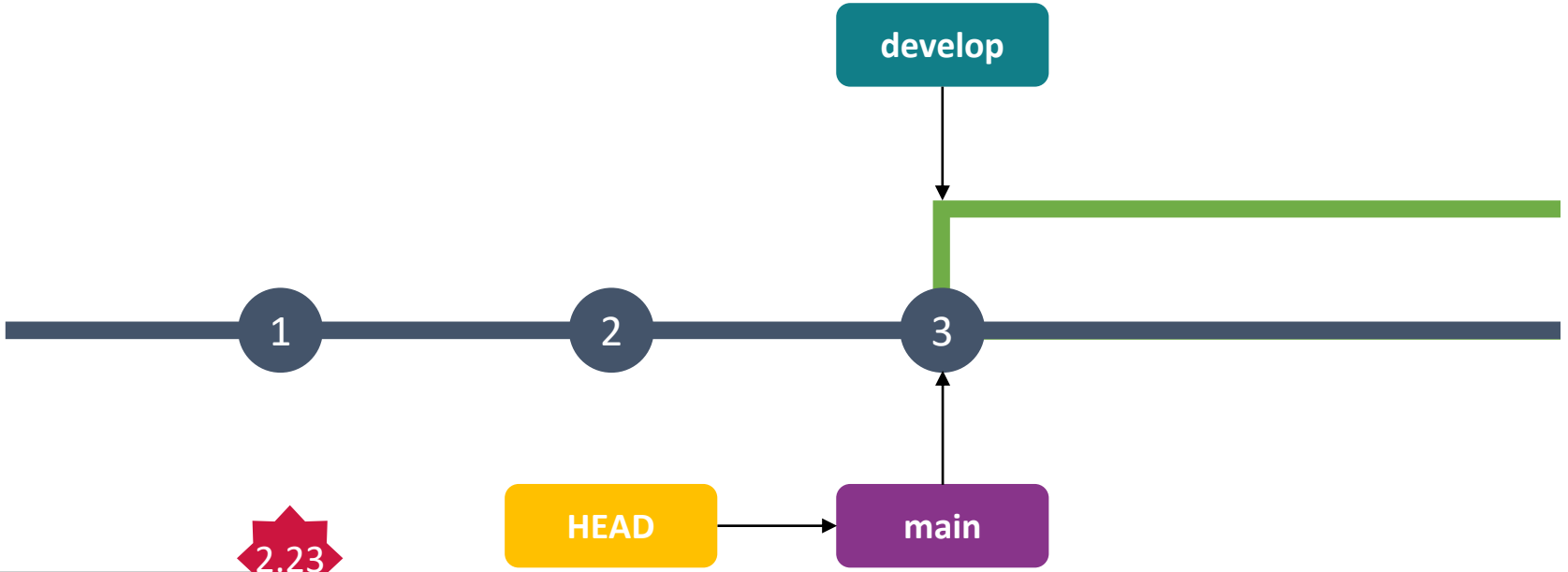
Let us practice

Merge Commits – With conflicts

Scenario 2 – Initial State



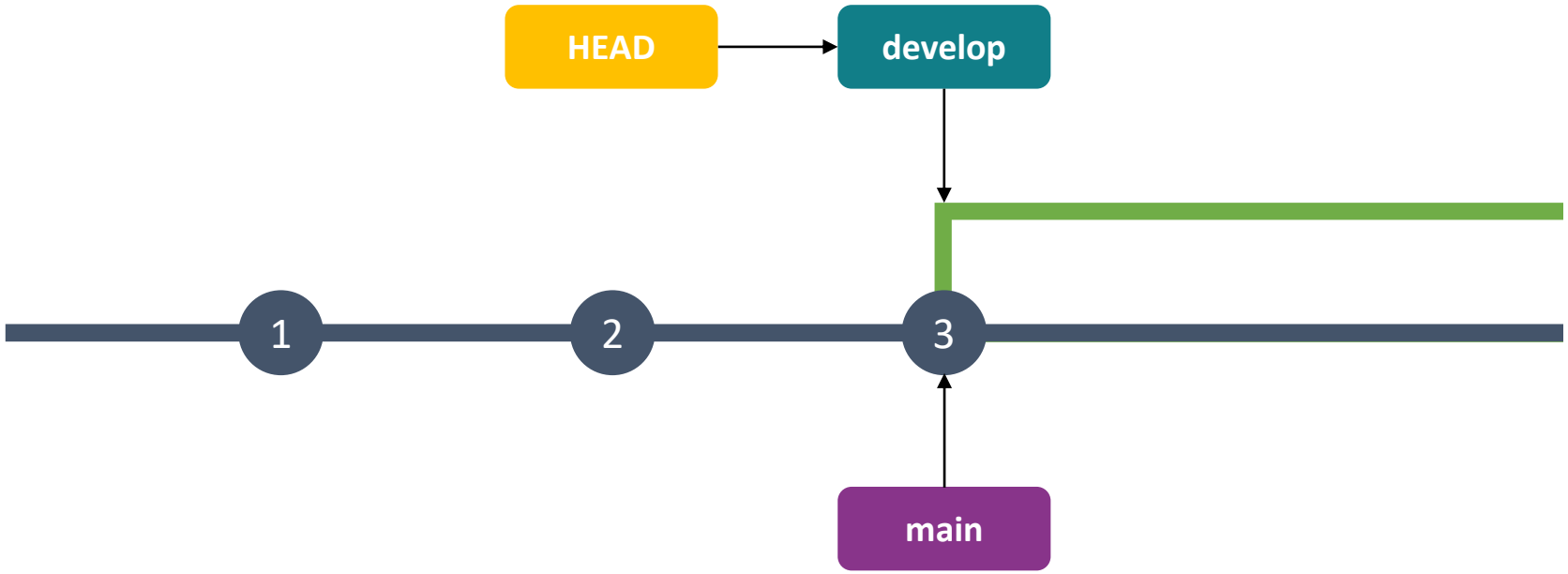
Scenario 2 – Initial State



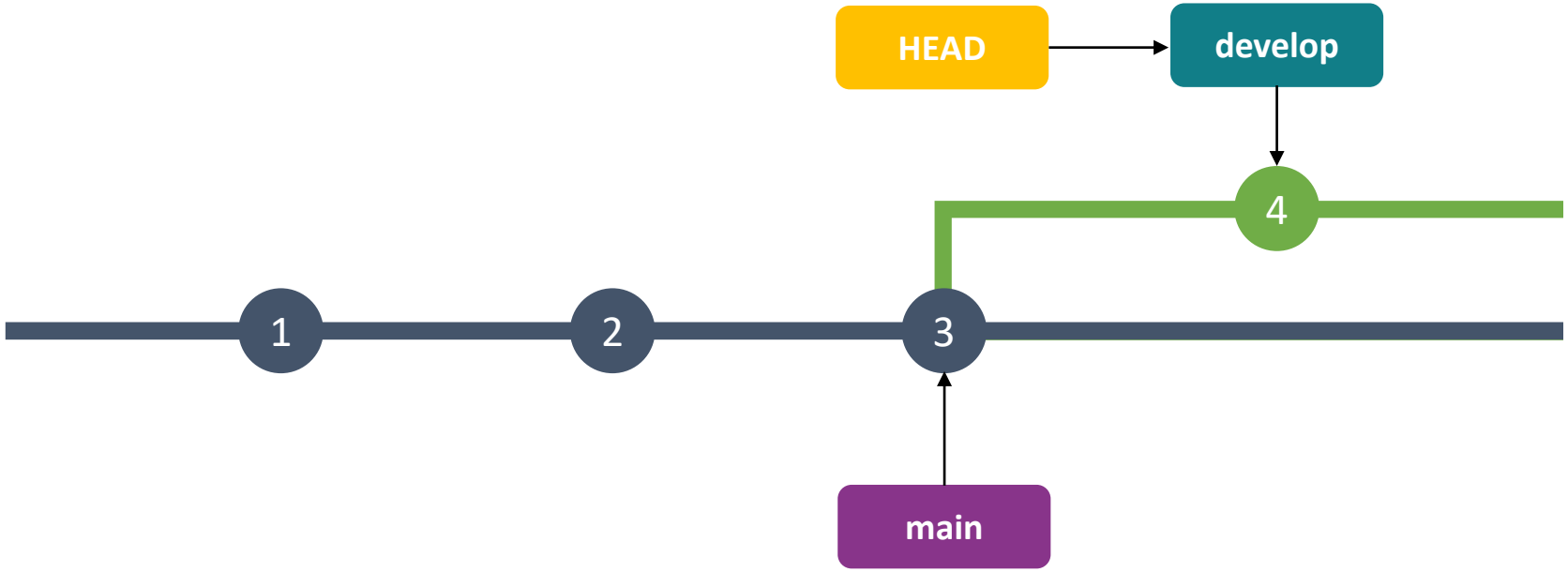
2.23

```
$ git switch develop
```

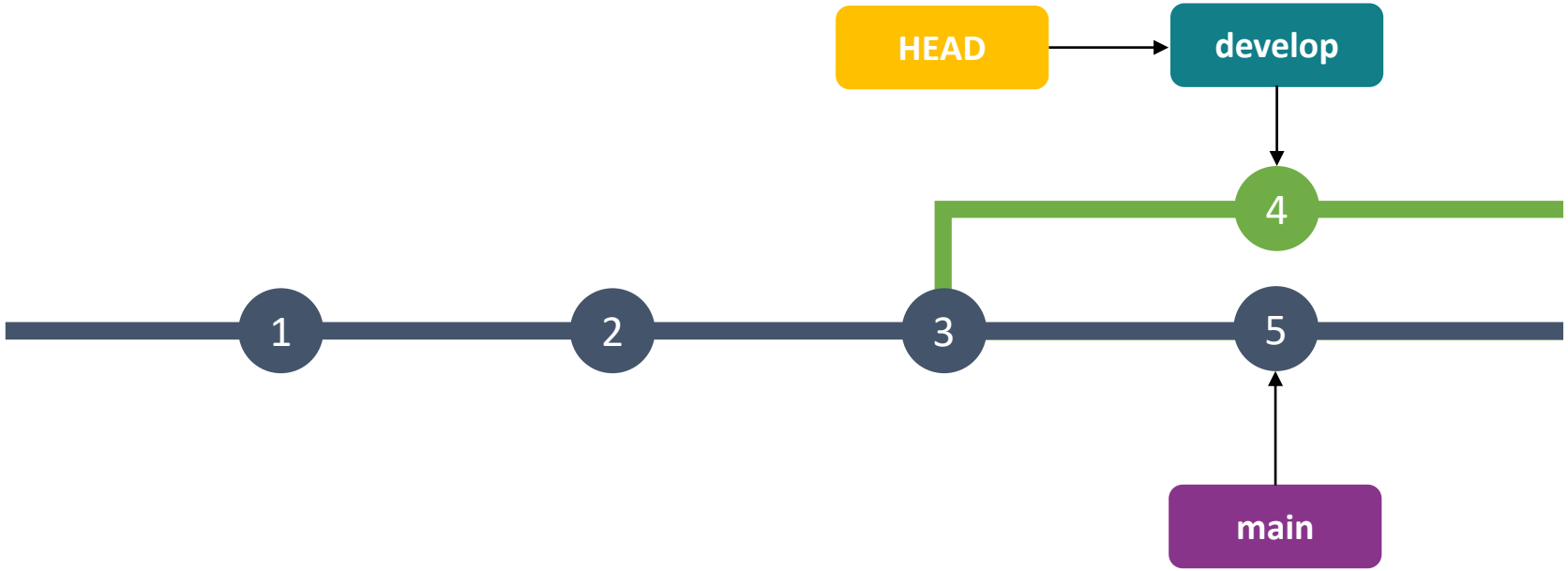
Scenario 1 – Switched To develop



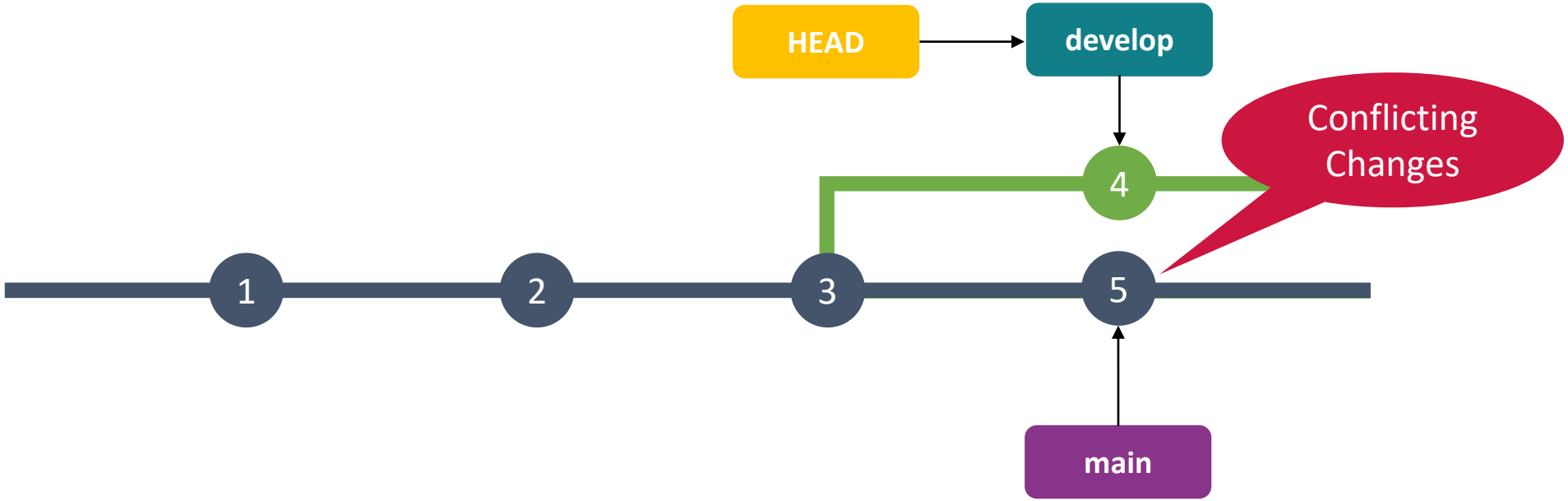
Scenario 1 – New Commit In develop



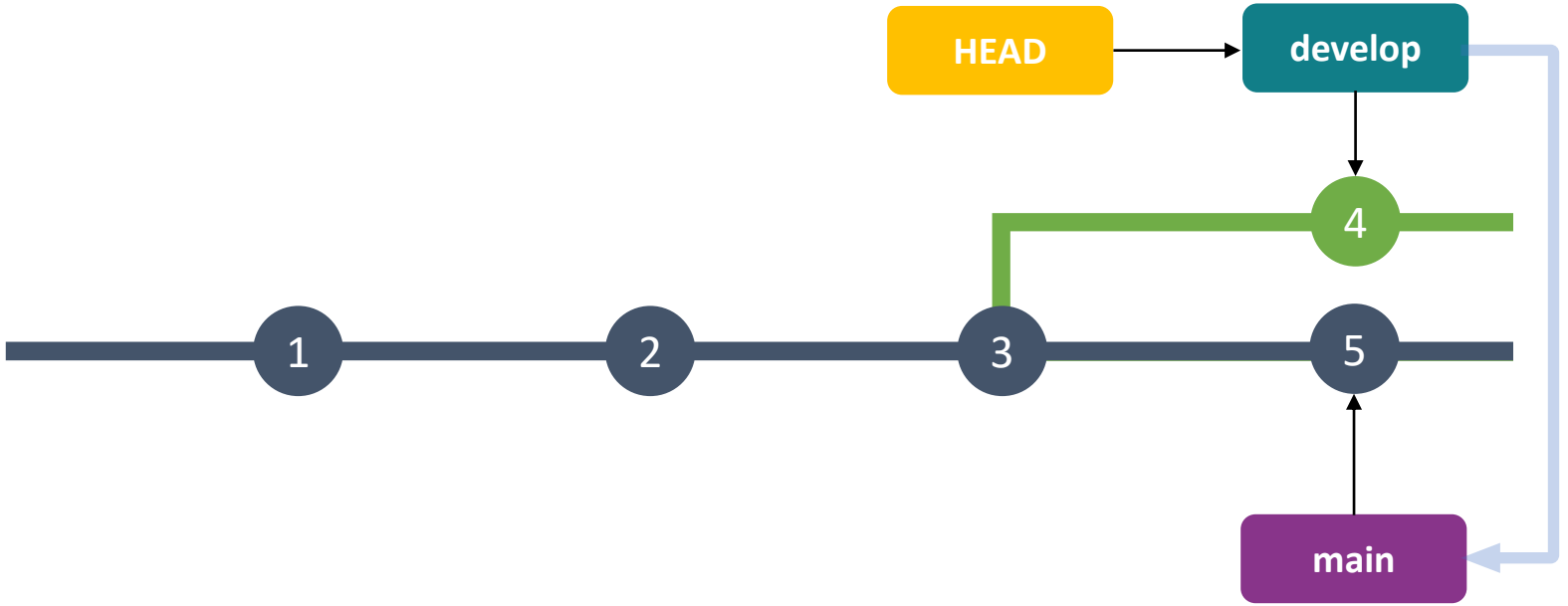
Scenario 1 – New Commit In main



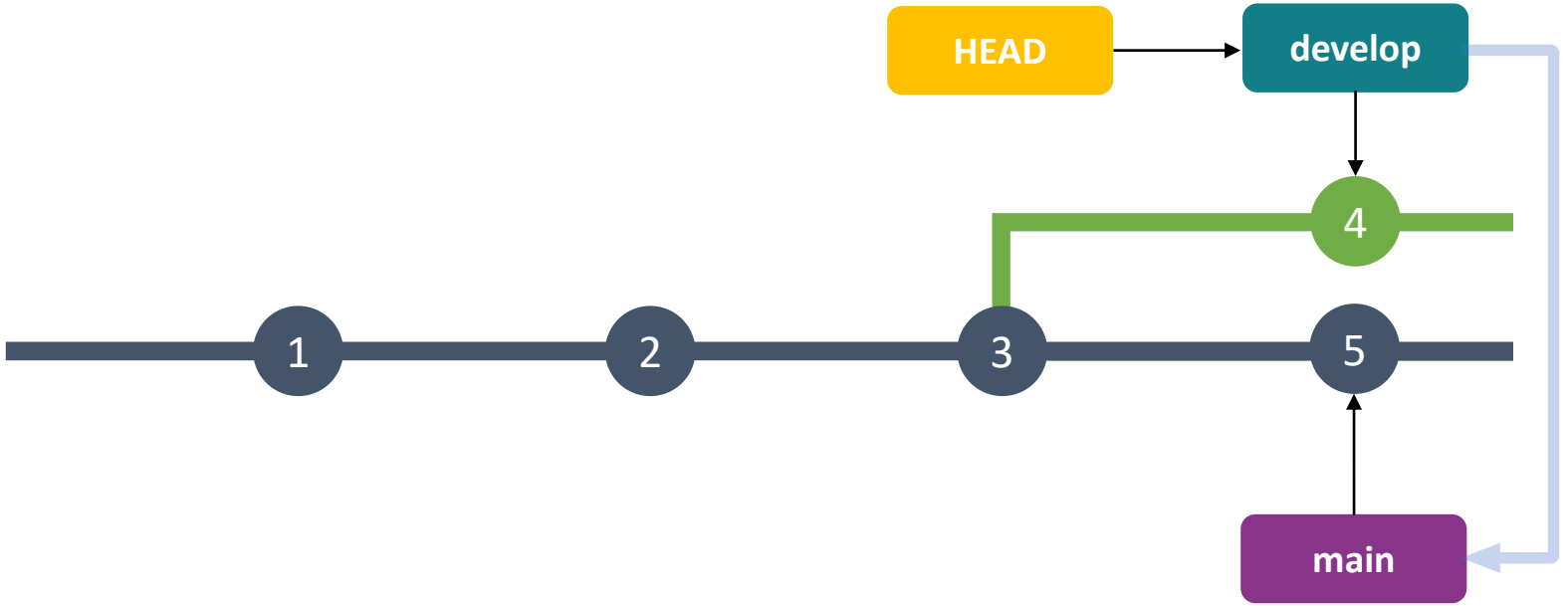
Scenario 1 – New Commit In main



Scenario 1 – Goal: Merge develop into main



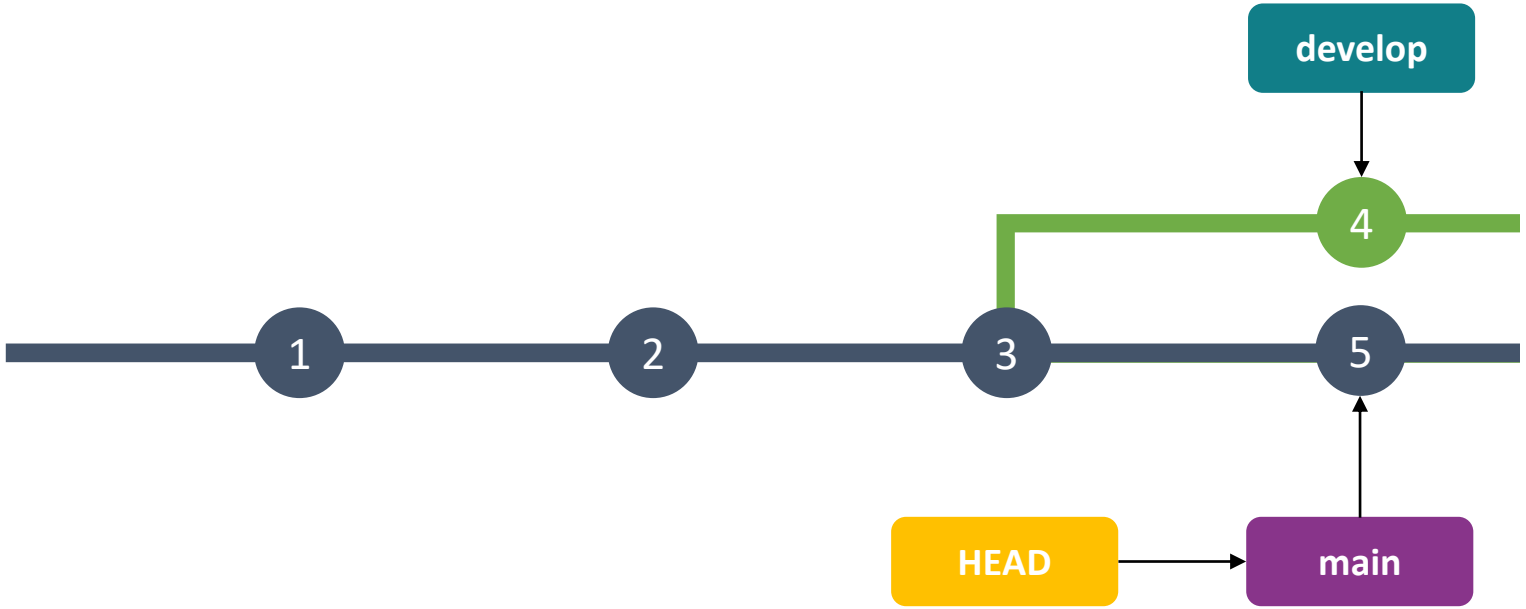
Scenario 1 – Goal: Merge develop into main



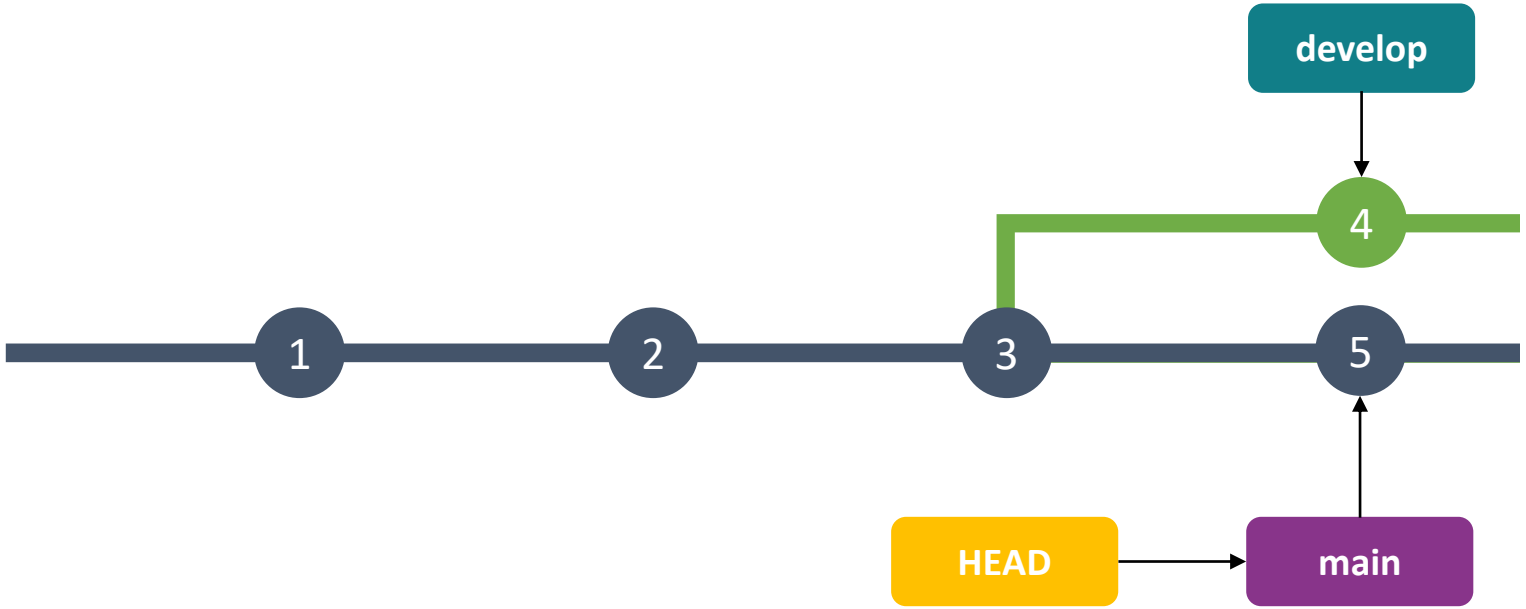
2.23

```
$ git switch main
```

Scenario 1 – Switched To main



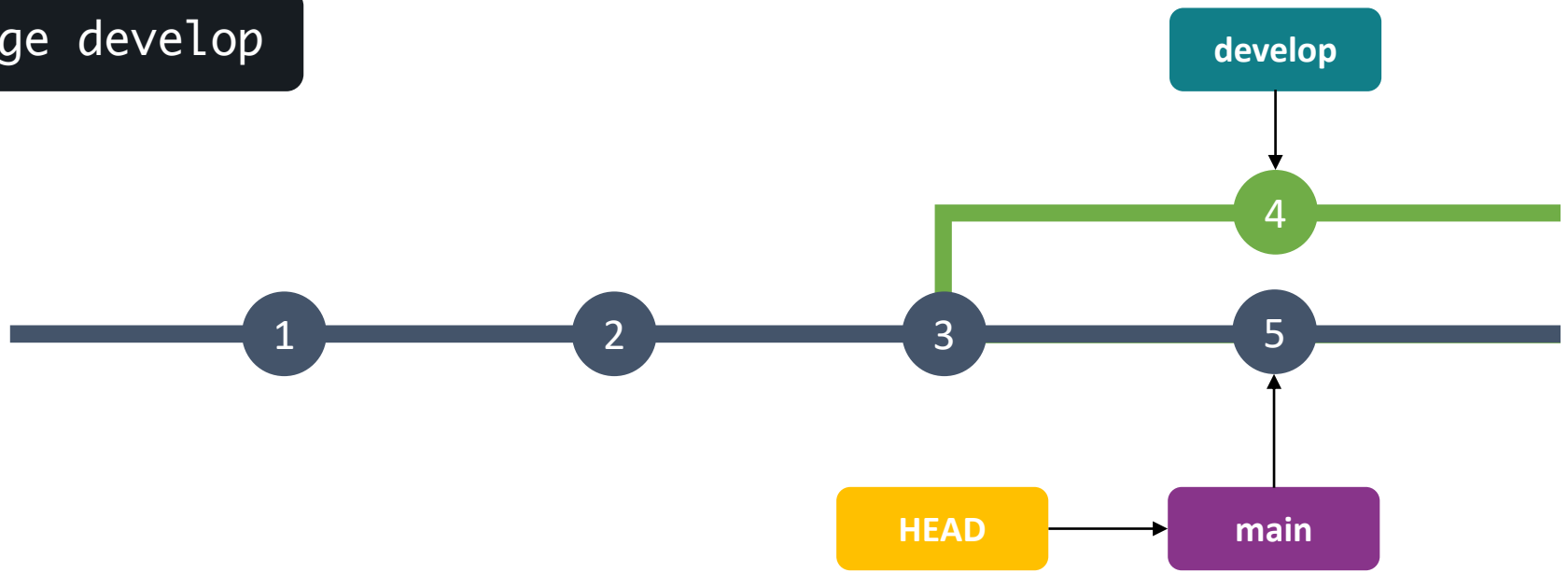
Scenario 1 – Switched To main



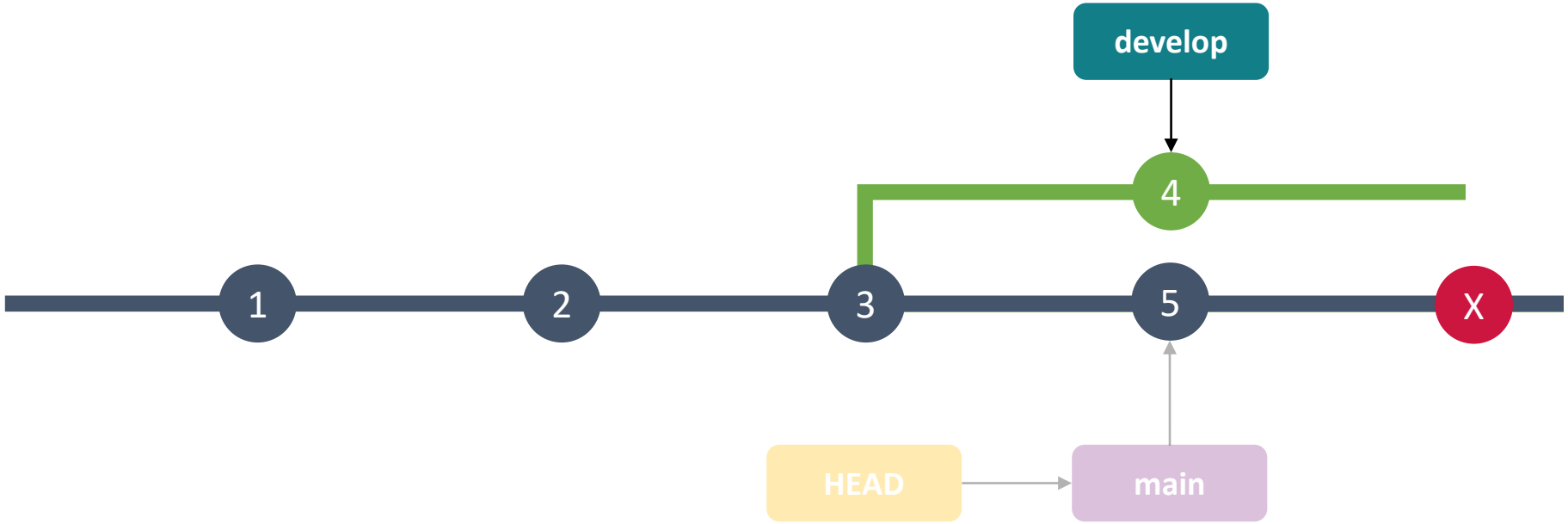

```
$ git fetch develop
```

Scenario 1 – Merge develop To main

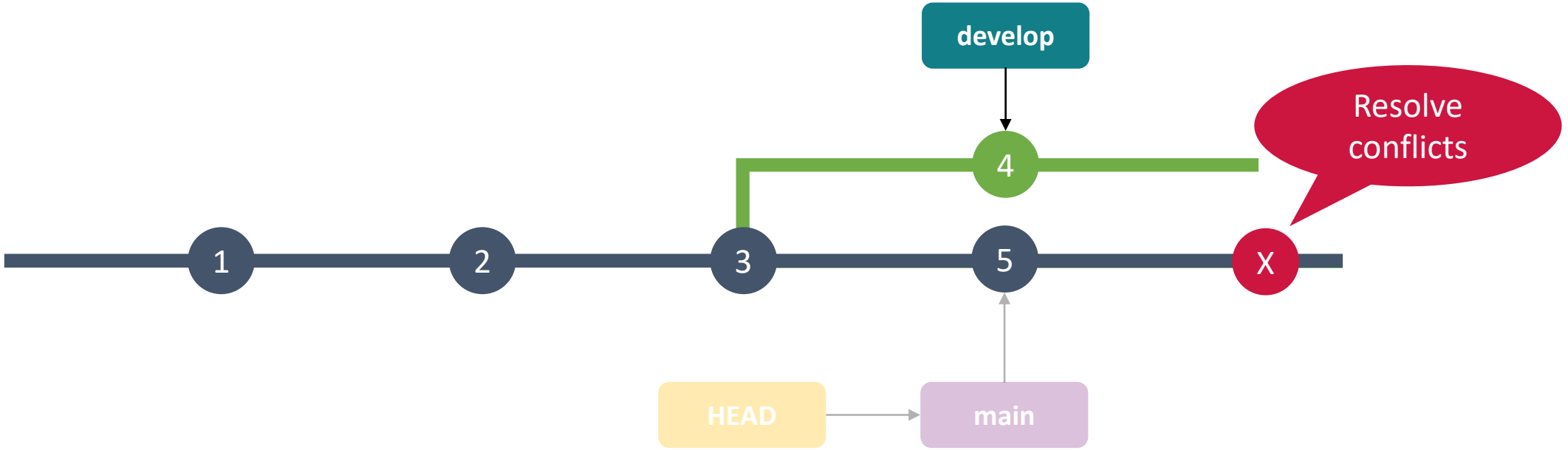
```
$ git merge develop
```



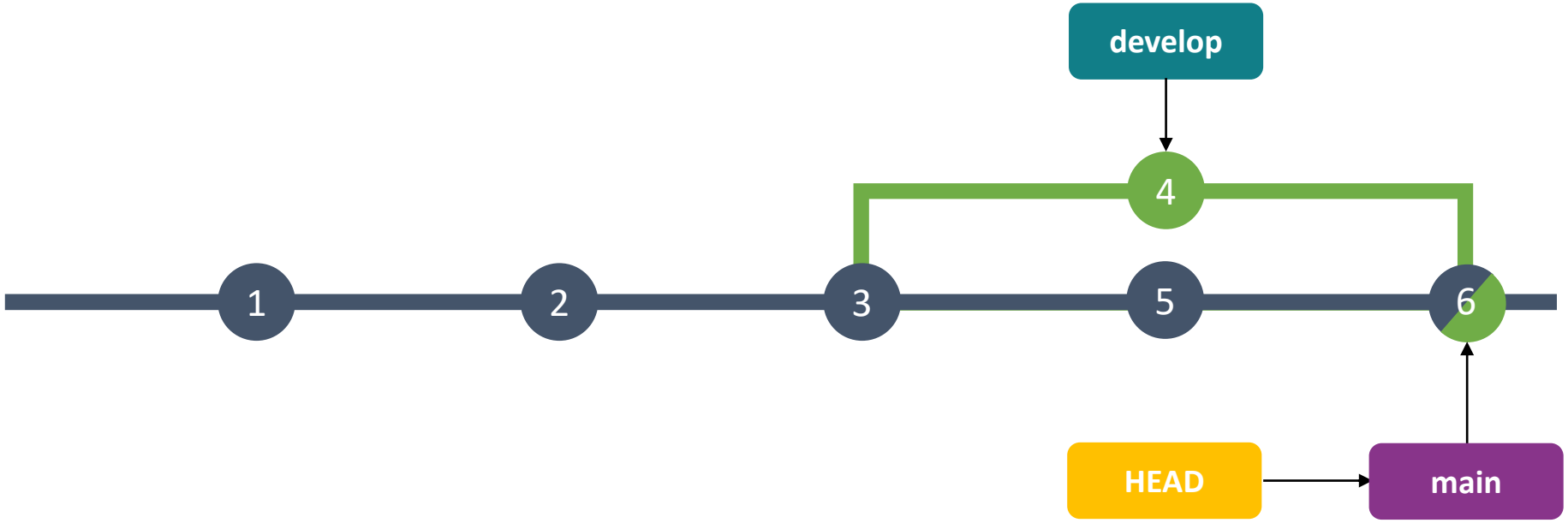
Scenario 1 – Merge develop To main



Scenario 1 – Merge develop To main



Scenario 1 – Merge Commit Created



Let us practice

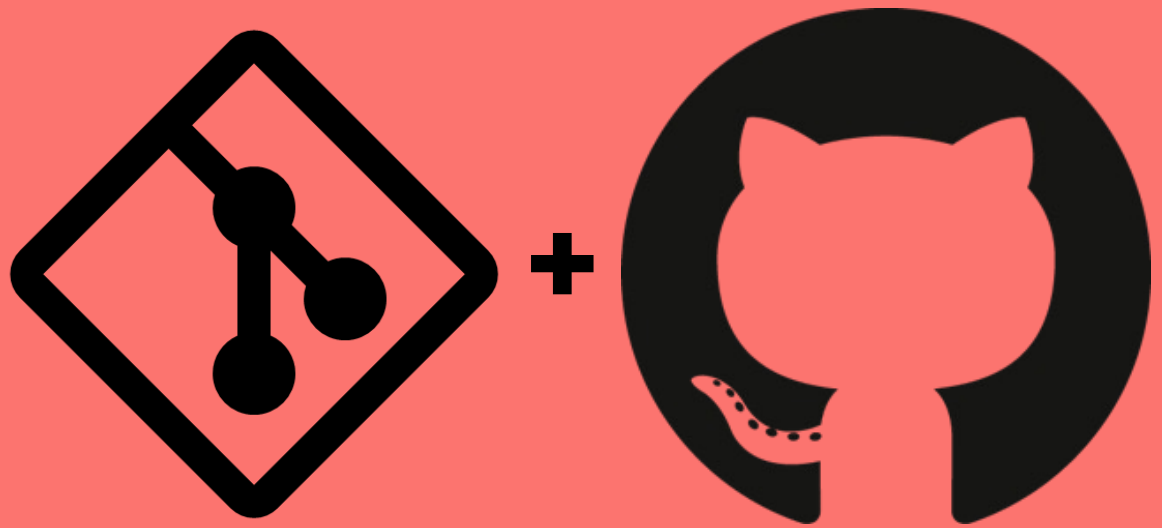
Takeaways

1. Git supports **merge into** feature and not merge from
2. Switch to the branch you want to merge into first, before merging
3. Commands
    a. `git switch <branchname> – to switch between branches`
    b. `git merge <branchname>`

Git

In Practice

Basics



1. git merge
2. git rebase
3. merge vs rebase

git rebase

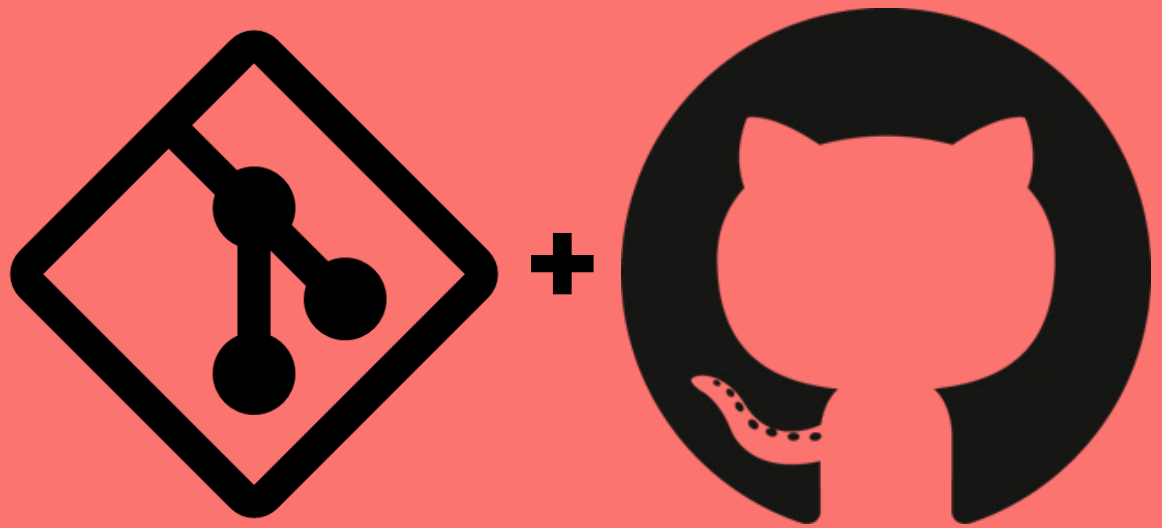
1. Fast Forward Merges
2. Merge commits – Merges that create a new snapshot

Let us practice

Git

In Practice

Basics



1. git merge
2. git rebase
3. merge vs rebase

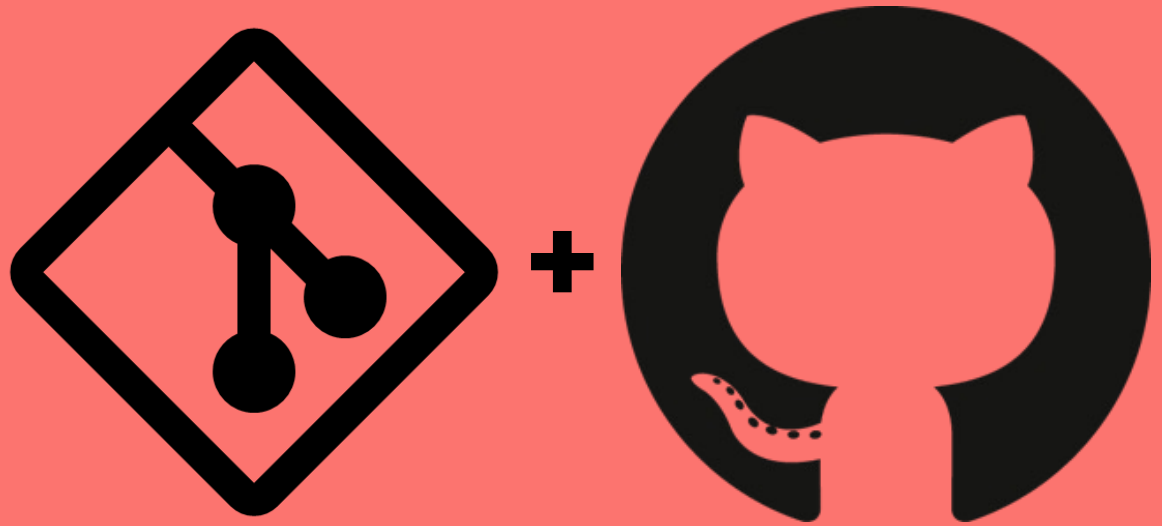
merge vs rebase

1. Fast Forward Merges
2. Merge commits – Merges that create a new snapshot

Git

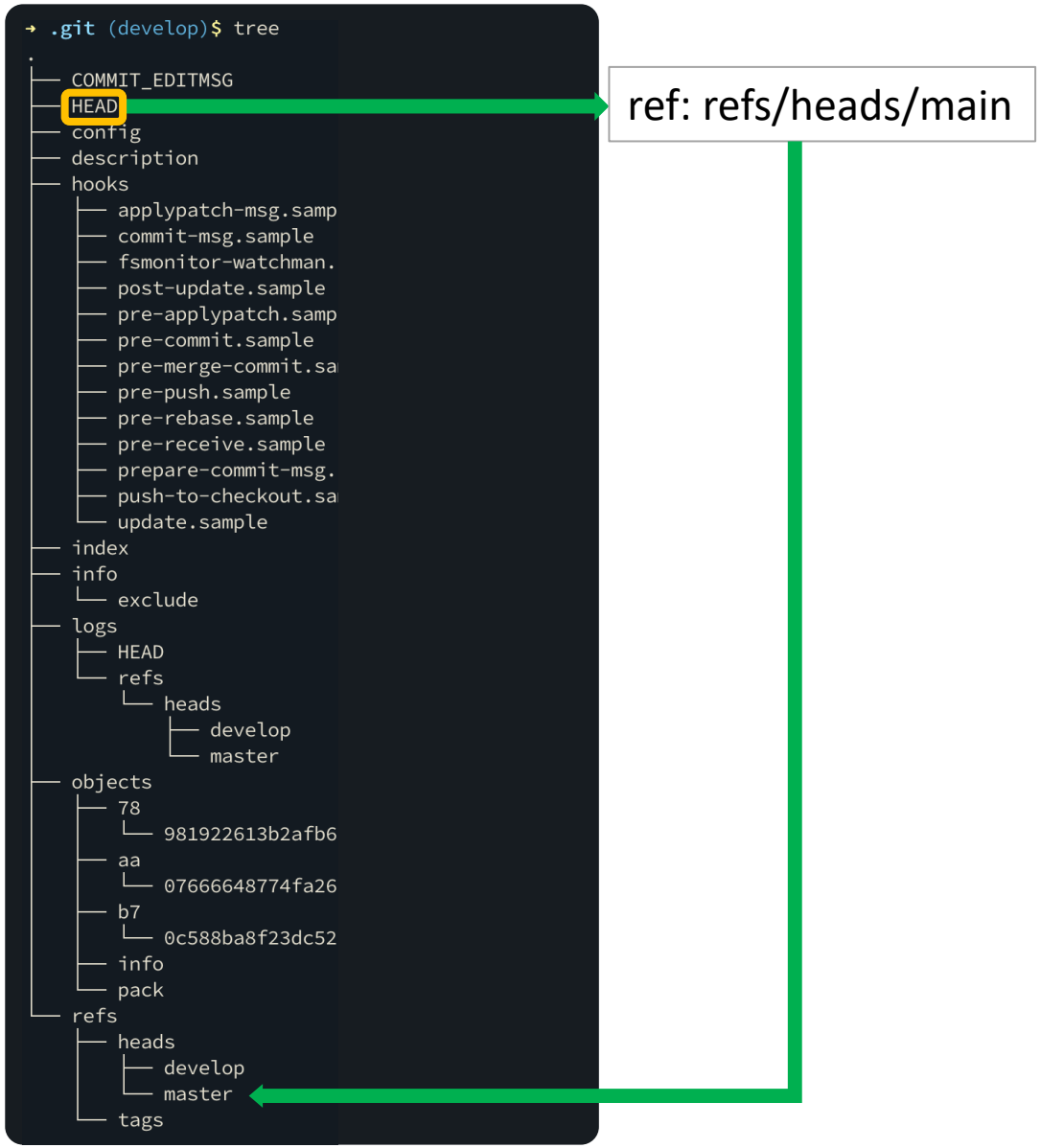
In Practice

Basics



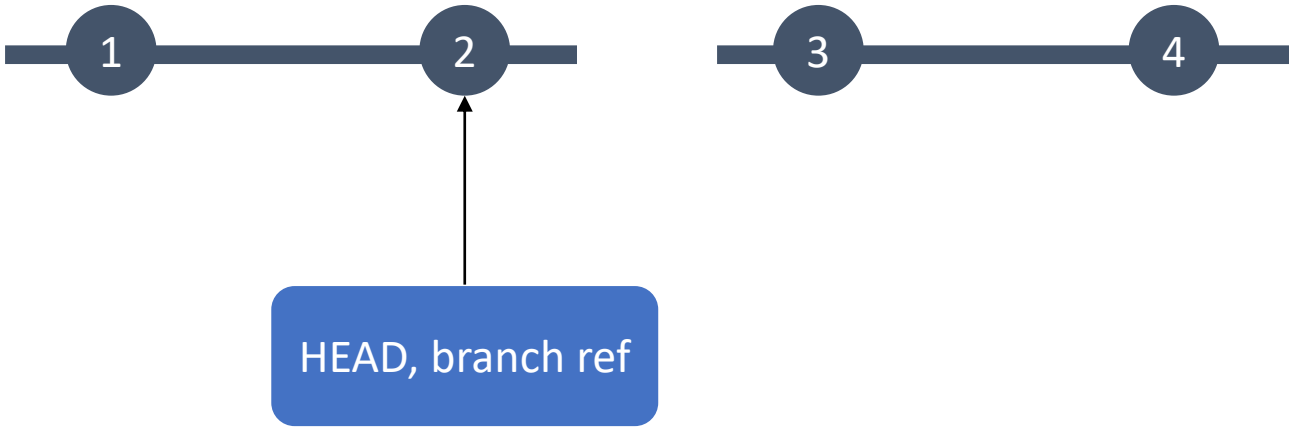
1. git reset
2. git revert
3. Comparisons
   git checkout vs reset vs revert

HEAD / branch refs

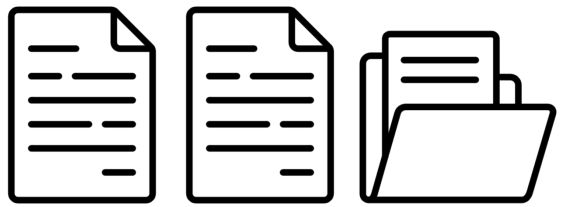


git reset


```
git reset 2 # commit ID 2
```



git reset – Types

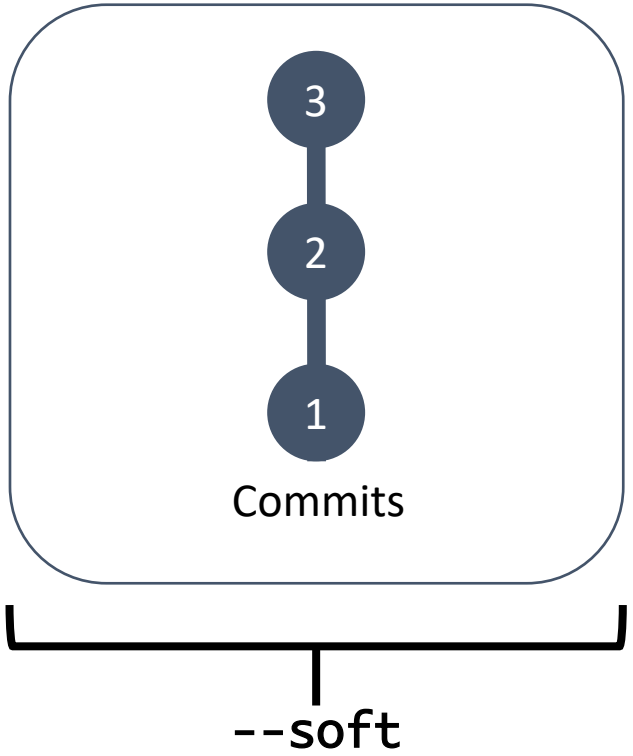


Working Tree

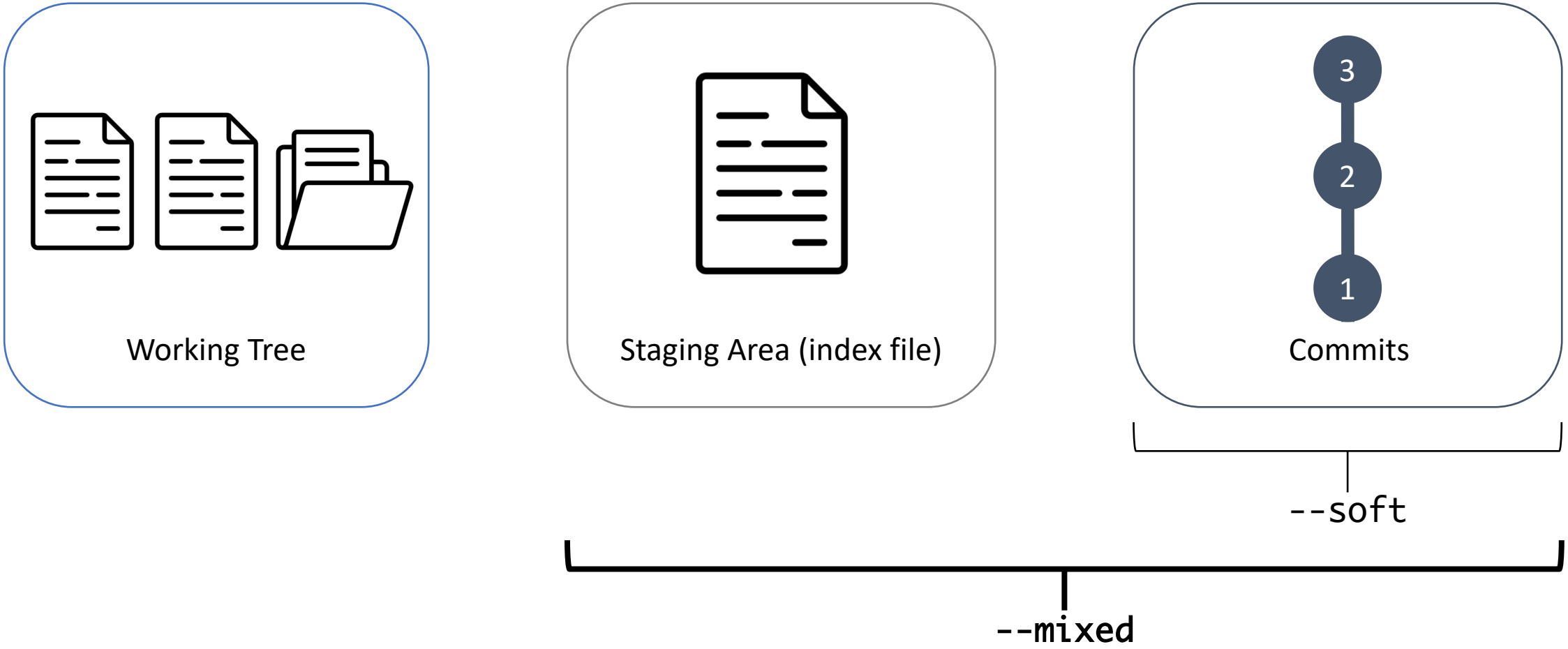
Staging Area (index file)

Commits

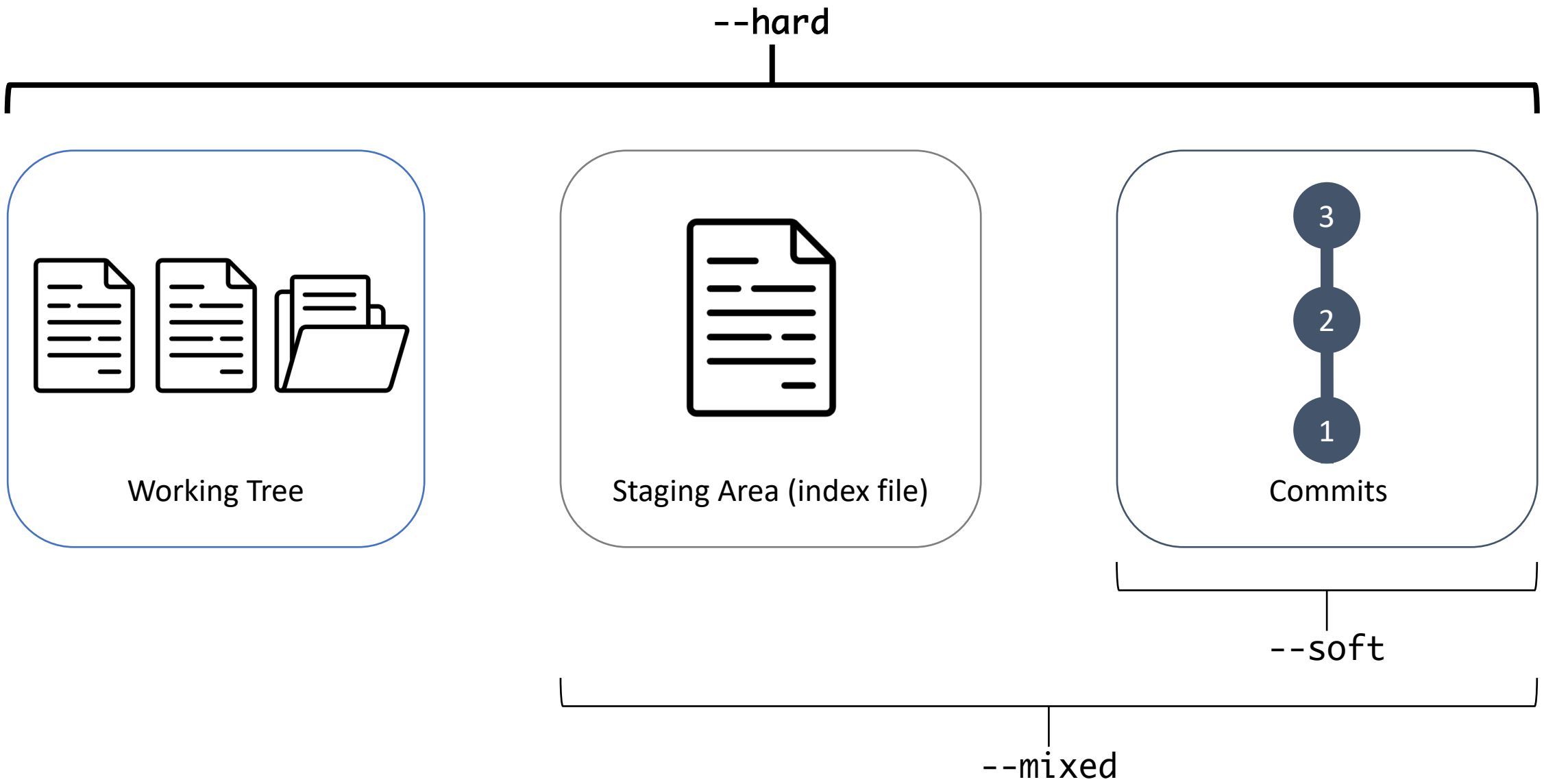
git reset – Types



git reset – Types



git reset – Types

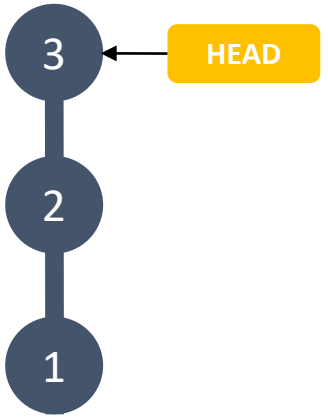


git reset – Types

--hard

Working Tree

Staging Area (index file)



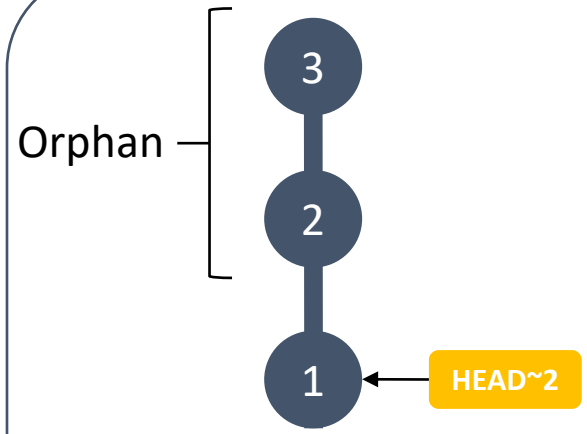
Commits

git reset – Types

--hard

Working Tree

Staging Area (index file)



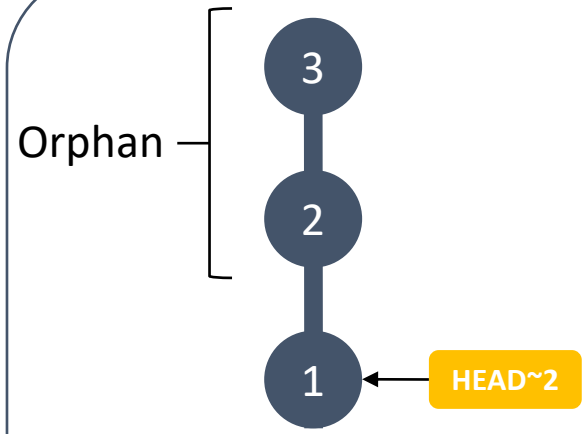
Commits

USE CASES for 3 types – TODO

--hard

Working Tree

Staging Area (index file)

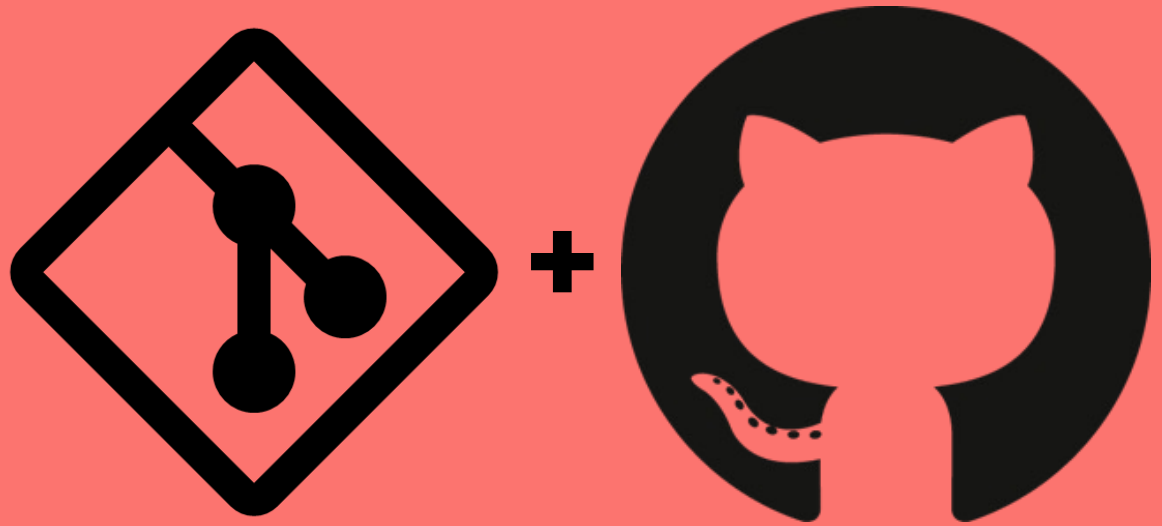


Commits

Git

In Practice

Basics



1. git reset
2. git revert
3. Comparisons
   git checkout vs reset vs revert

git revert

1. Revert is used to apply the inverse of a commit to the project history
2. The git revert command can be considered as an “undo” type command, however, it is not a traditional undo operation
3. Instead of removing the commit from the project history, revert inverts the changes introduced by a commit and appends a new commit for the resulting inversed content

git revert


```
git revert 2 # commit ID 2
```

Scenario 1 – Reverting a Fast-Forward commit

1. x

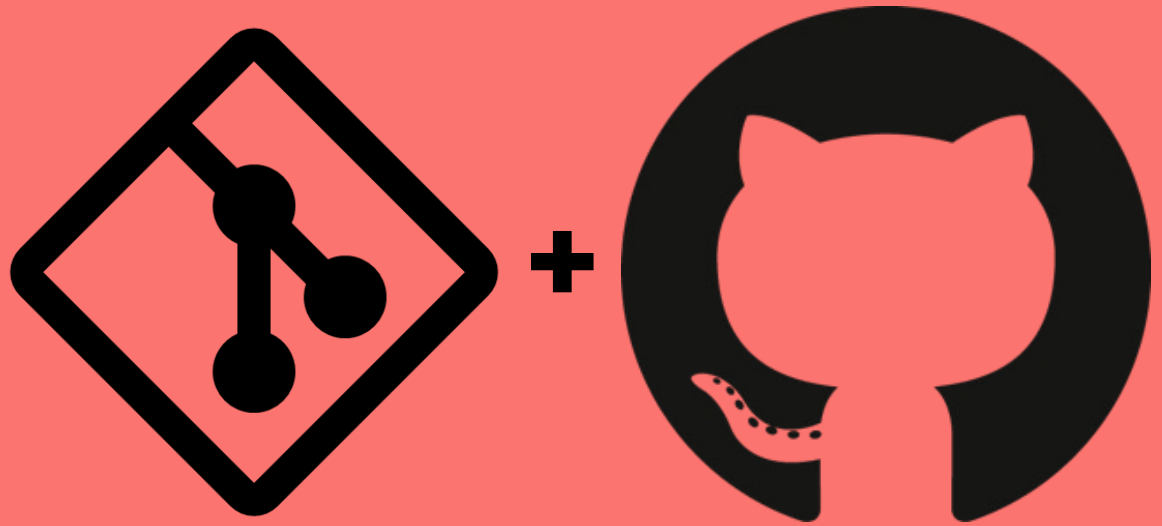
Scenario 2 – Reverting a Merge commit

1. x

Git

In Practice

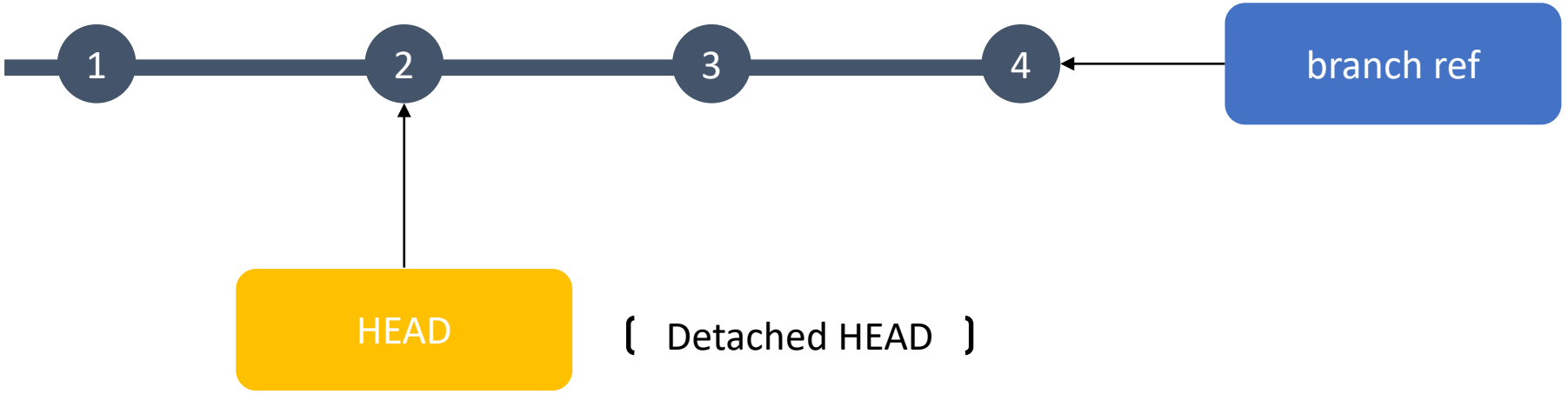
Basics



1. git reset
2. git revert
3. Comparisons
   git checkout vs reset vs revert

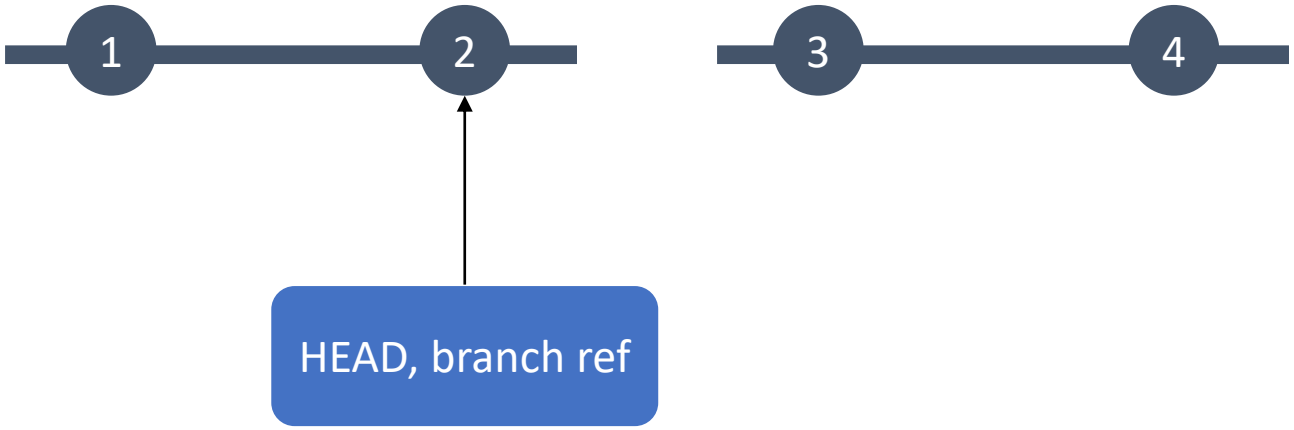
git checkout


```
git checkout 2 # commit ID 2
```



git reset


```
git reset 2 # commit ID 2
```



git revert


```
git revert 2 # commit ID 2
```

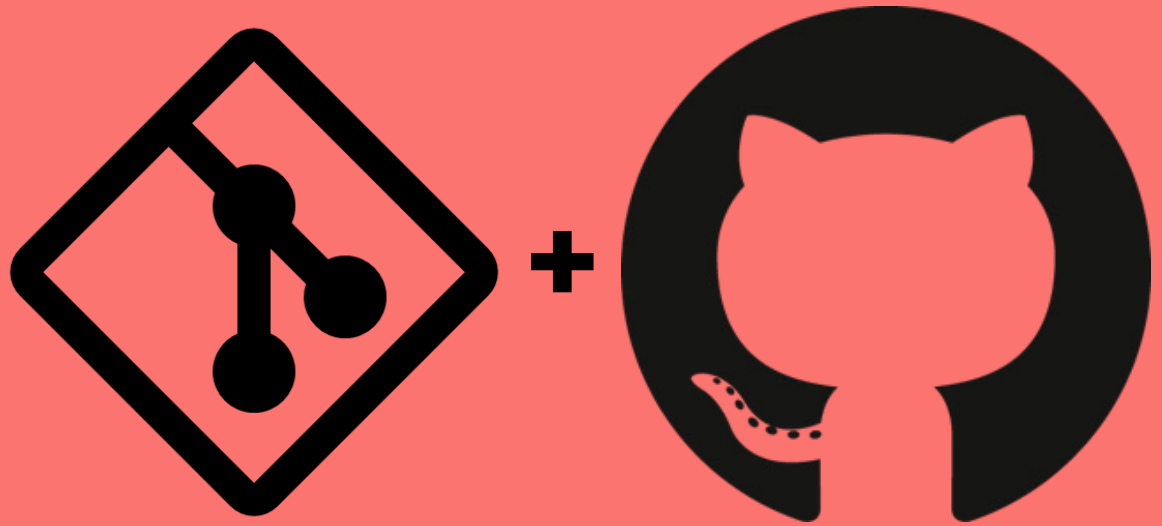
revert vs reset

revert	reset
Can revert a single branch	Can revert multiple branches
Will create a new commit	Will not

Git

In Practice

Intermediate



1 Branching Strategies

2 OTHERS

Branching Strategies

1. Integrating changes and structuring releases
    1. Mainline development – always integrate
    2. State, release and feature branches

2. Mainline development
    1. Few branches
    2. Relatively small commits
    3. High-quality testing and QA standards

3. State, release and feature branches
    1. Here, Branches enhances structure and workflows
        1. Different types of branches
        2. Fulfill different types of jobs

Branching Strategies

1. Branching conventions in a team depends on its size, the type of project and how releases are handled by your team / organization
2. git is very good when creating branches – but it doesn't tell how to use them
3. So, it is better to think about how work should be structured in your team

Branching Types

1. Long running
2. Short–lived
3. Remote Branches?

Long Running Branches

Every repository has at least one long running branch

1. They exist through the life-time of the project – main / master

Integration branches are often long running and they are designed to mirror the “stages” in a development life cycle – develop / test / staging / production / hotfix

- A common convention of an integration branch is
  1. Typically, commits are not added directly to these branches
  2. Commits to these branches only happen through merge or rebase

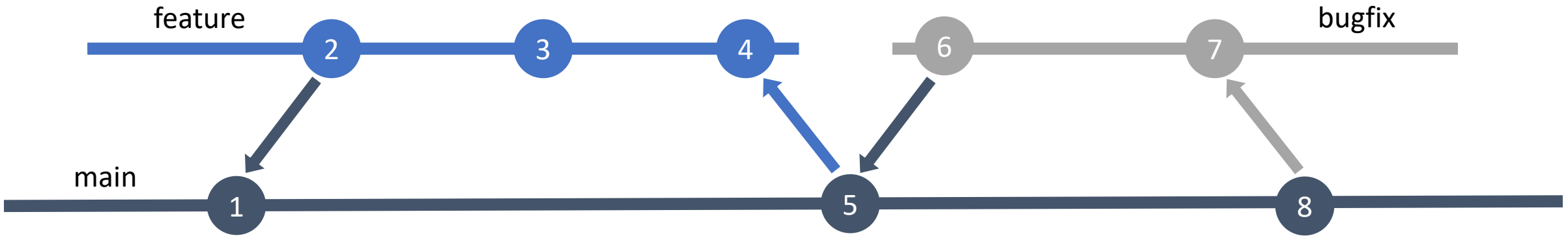
Short-Lived Branches

1. These branches are created for new features, bug fixes, experiments etc
2. Usually, short-lived branches are based on a long running branch
3. Generally, these branches are deleted after an integration (merge / rebase)

Popular Branching Strategies

GitHub Flow

- Very lean and simple: only one long running branch (main/master) and feature branches

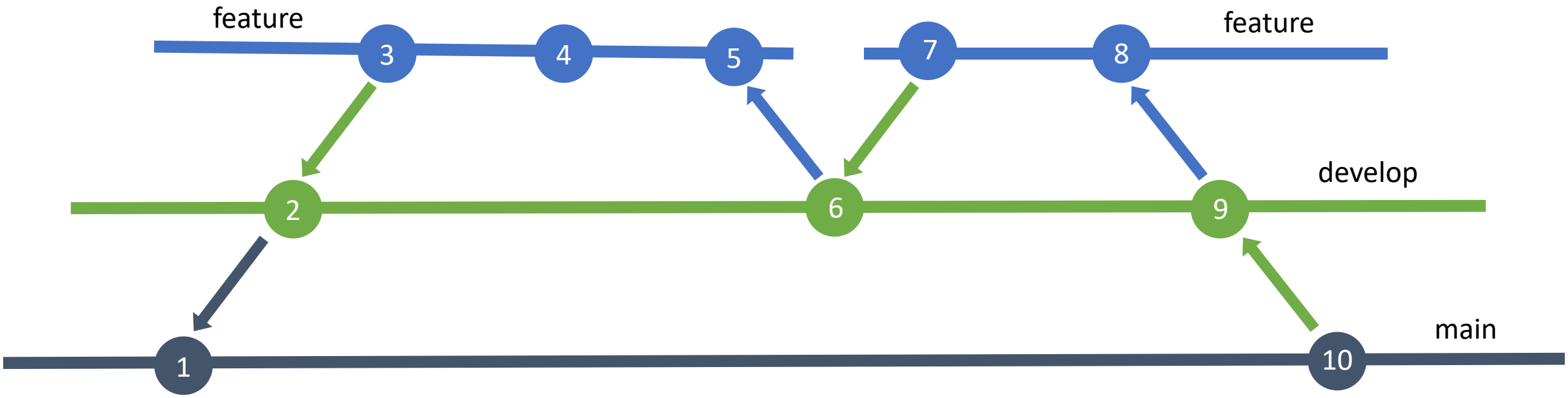


Popular Branching Strategies – GitFlow

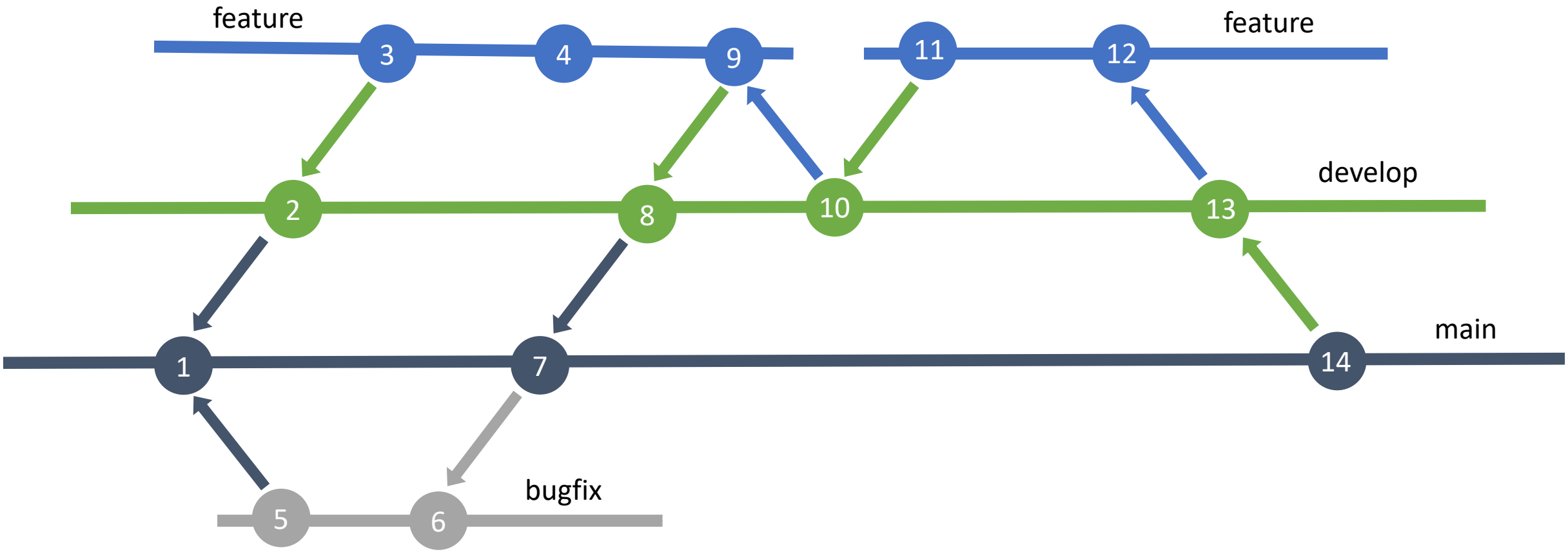
GitFlow

1. Main – reflection of current state of production
2. Feature branches are created from develop and feature branches merge back into develop
3. Develop is also the starting point of any new releases
4. Open a release branch, commit any bug fixes. Once confidence merge it back to main. Add tags to release commit on main and close the release branch

Popular Branching Strategies – GitFlow



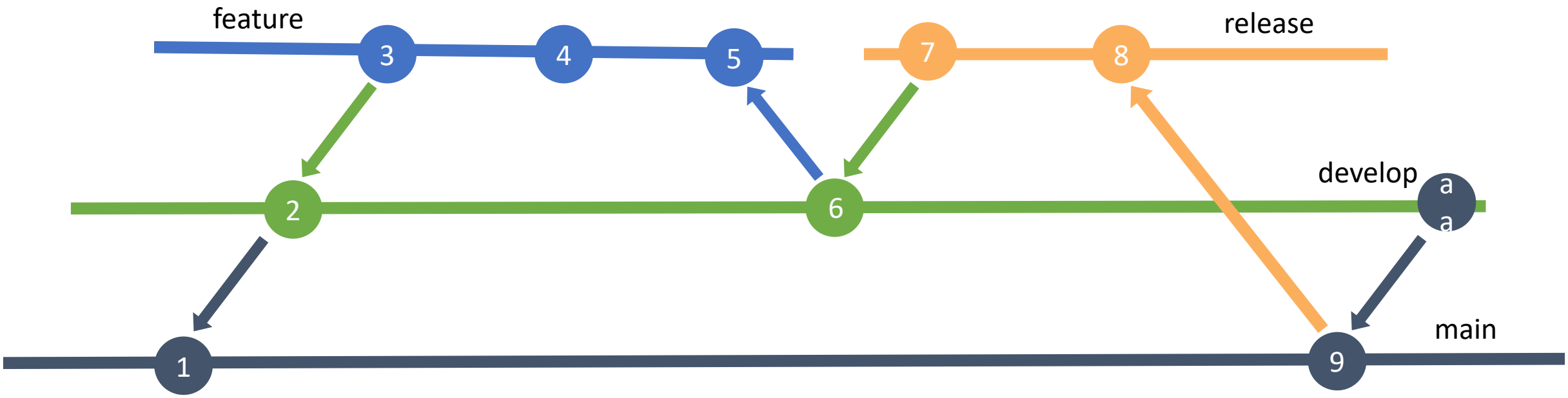
Popular Branching Strategies – GitFlow



Popular Branching Strategies

1. GitFlow
    1. More structure and rules
    2. Long-running: main and develop branches
    3. Short-Lived: features, releases, hotfixes
    4. GitFlow introduces a number of steps and tasks in the process

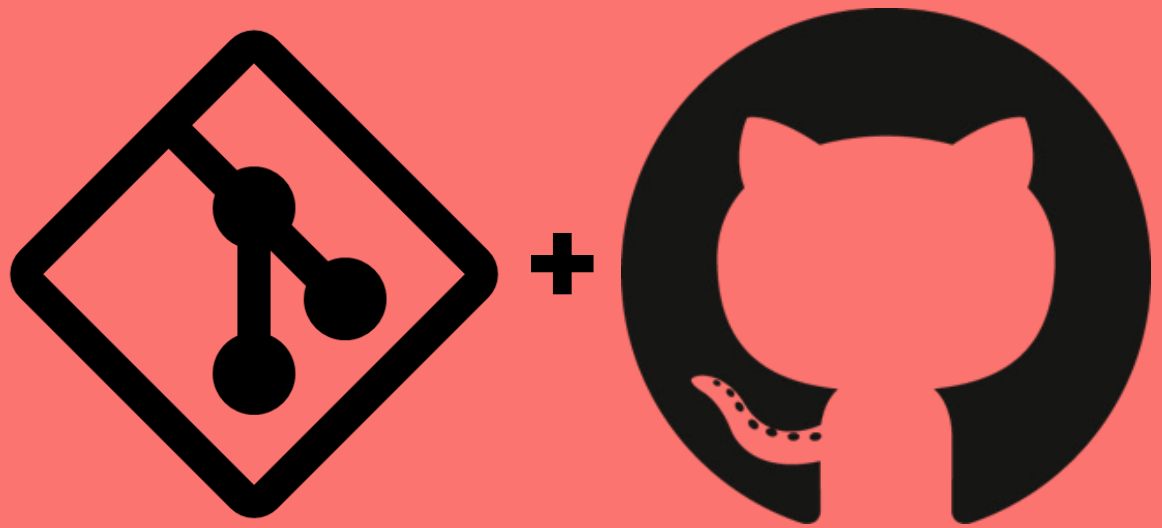
Popular Branching Strategies – GitFlow Release



Let us practice

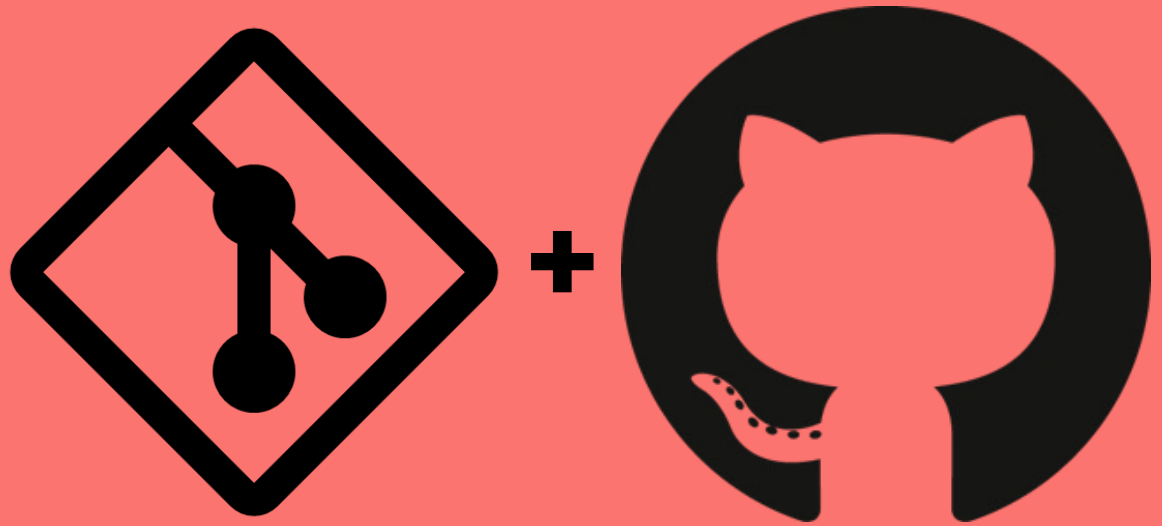
Git

In Practice



Git

In Practice



1. Git Branches
2. Branching Strategies