Advanced git Course

How to git good!

Simone Rossi Tisbeni

Who am I

Simone Rossi Tisbeni

PhD Data Science and Computation
University of Bologna, INFN

simone.rossitisbeni@unibo.it

https://github.com/rsreds

Table of Content

Part 1

- Recap of git basics
  - The three areas
  - commits, branches and references
  - Typical workflow
- Why advanced git
- Useful commands
  - The stash
  - checkout, switch and restore
- Fixing mistakes
  - restore
  - reset
  - revert

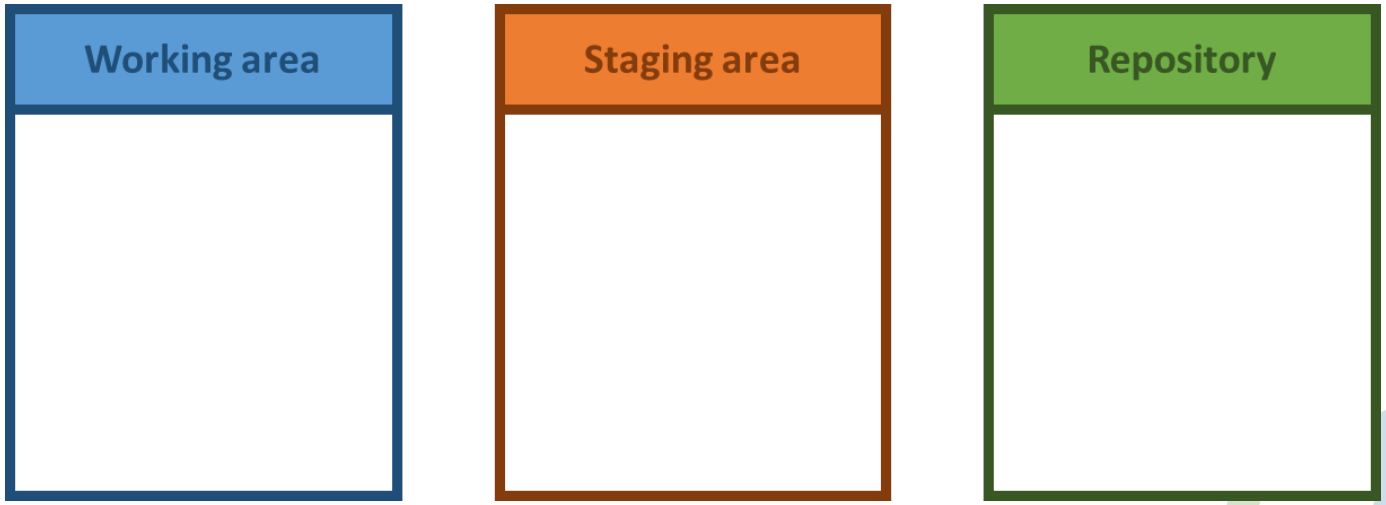
Part 2

- Advanced merging
  - rebase
  - cherry-pick
- Reflog
- Rewriting History
  - amend
  - rebase
  - filter-repo
- Git Hooks
- Advanced debugging
  - log
  - blame
  - bisect

PART 1

Recap: the three areas

1. Working area
2. Staging area
3. Repository

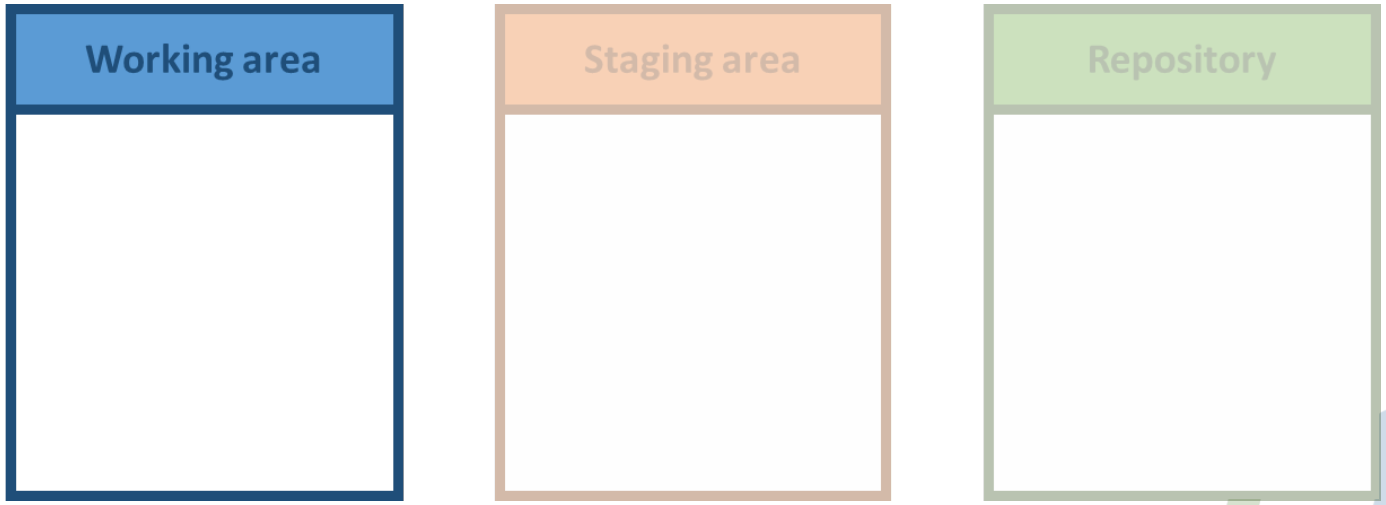


Recap: working area

Single **checkout** of one version of the project

Contains

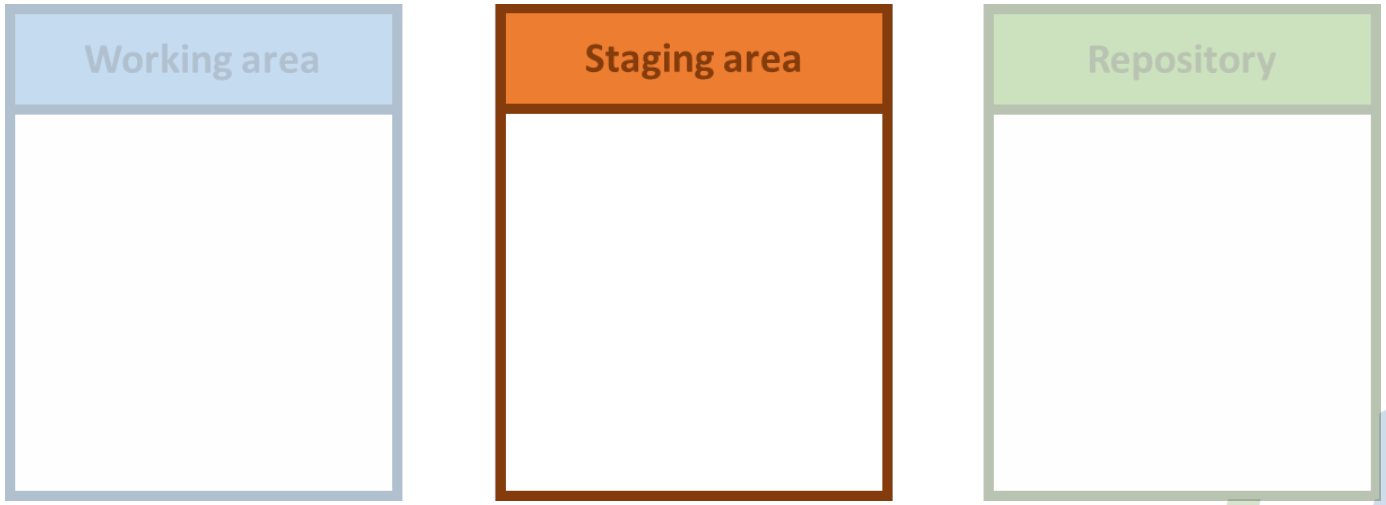
- Untracked files
- Tracked and unchanged files



Recap: staging area

Also known as **index**

Files that will be part of your next commit



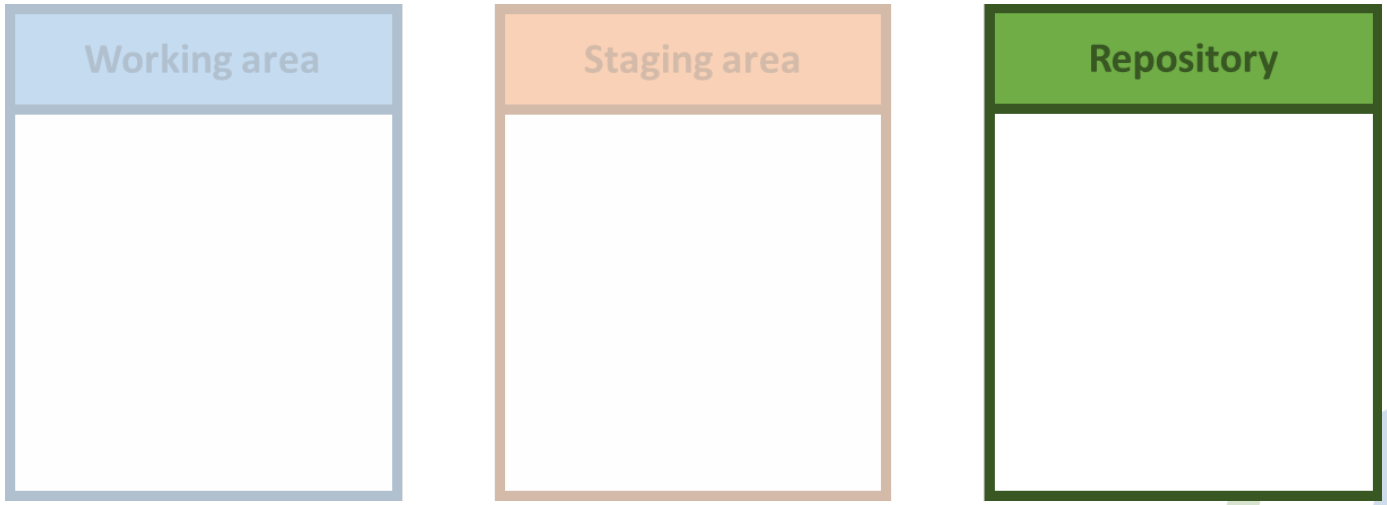
Recap: repository

The `.git` directory

Contains the files that git knows:

- Metadata
- Objects data

All ever **committed** file versions



Recap: what is a commit?

A commit points to:

- The packaged data

Actually, it points to a tree pointing to data blobs

And contains metadata:

- Author and committer
- Date(s)
- Commit message
- Parent(s) commit

```
$ git cat-file -p 9c098
tree a2985758940c9c8eb1fe1e483006cd3e…
parent 701dcc4e2eadbc3054c2585098305b…
author Simone Rossi Tisbeni 1707835791 +0100
committer Simone Rossi Tisbeni 1707836247 +0100

Add new feature
```

commit	tree	parent
9c098…	a2985…	701dc…
author	**date**	**message**
Simone Rossi Tisbeni	1707835791 +0100	Add new feature

Recap: you can't "change" commits

The hash of this information is the commit's **SHA-1**

If you change any data, the commit will have a new hash

The old one is not going to be removed (for some time)

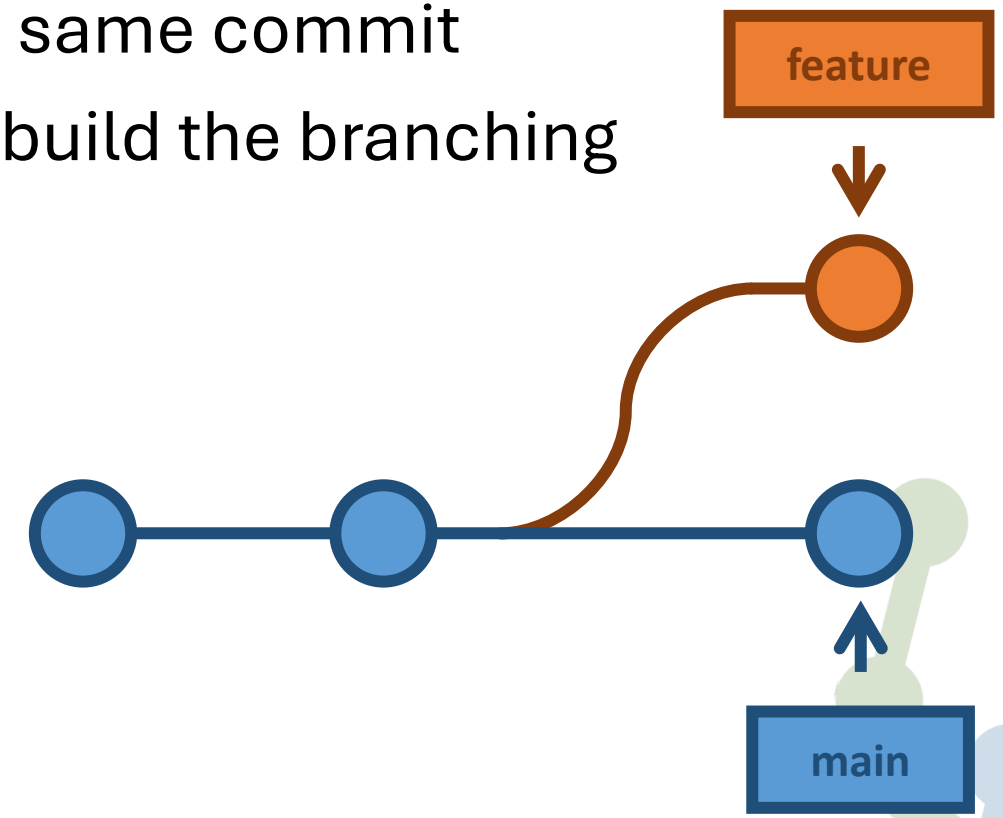
! The refs to that commit will still point to the old one

commit	tree	parent
9c098...	a2985...	701dc...
author	**date**	**message**
Simone Rossi Tisbeni	1707835791 +0100	Add new feature

Recap: branches

- A branch is a **movable** pointer to a commit
- Multiple branches can point to the same commit
- Commits pointing to their parents build the branching

```
$ git branch feature
$ git commit -am "main commit"
$ git switch feature
$ git commit -am "feature commit"
```



Recap: references

Commits in git are indexed with their SHA-1 values

They are stored in files under simple names so that they can be more easily **referenced**

4 types of references:

- Heads ← Branches reside here
- Tags
- Stash
- Remotes

```
$ find .git/refs --maxdepth 1
.git/refs
.git/refs/remotes
.git/refs/tags
.git/refs/stash
.git/refs/heads
```

Recap: heads

- Points to the last commit in a branch
- New commits update the heads

`git branch <branch_name>`

Create pointer to the SHA-1 of the last commit of the current branch

HEAD

- `HEAD` file is a symbolic reference to the current branch
- It contains a pointer to a head pointer to a commit

Recap: tags

- It points to a commit
- Like a branch reference but it never moves

`git tag -a`

annotated tag: also contains a tagger, a date, a message

```
$ git tag v.1.4
$ git tag -a v1.5 -m "version 1.5"
$ tree .git/refs/tags/
.git/refs/tags/
├── v1.4
└── v1.5
$ git show v1.5
tag v1.5
Tagger: Simone Rossi Tisbeni
<simone.rossitisbeni@unibo.it>
Date:   Wed Mar 13 12:02:07 2024 +0100

version 1.5

commit a2844e7b5… (HEAD->main, tag:v1.5)
…
```

Recap: remotes

- Remote repositories are saved as `refs` in `.git`
- `refs/remotes`

  store a pointer to the last known change for each branch
- Same as `refs/heads` but “read-only”:

  You can switch to a remote ref, but commits will be **dangling** and will be (at some point) removed

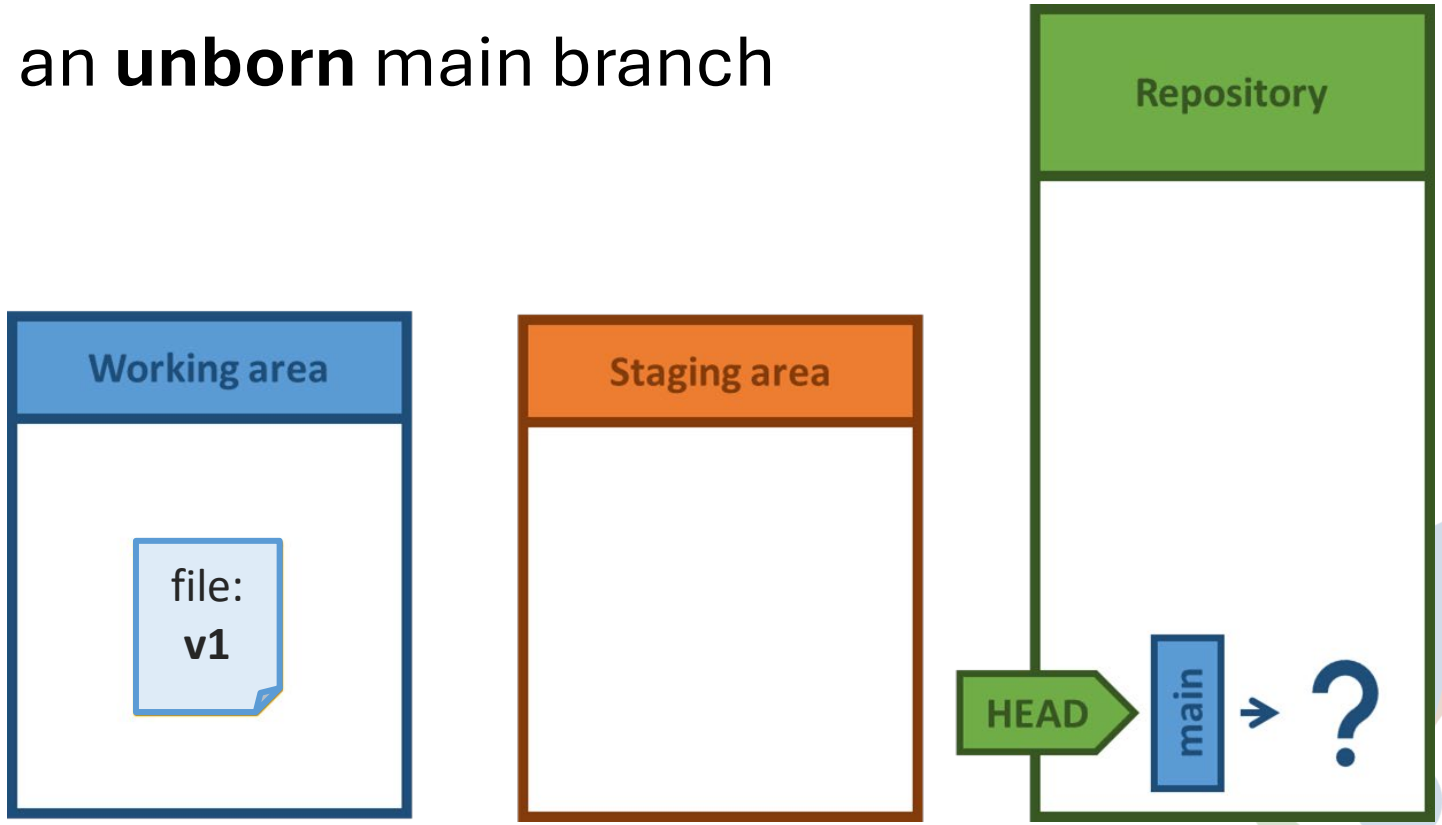
```
$ tree .git/refs/remotes/
.git/refs/remotes/
├── fork
│   ├── fix-log-url
│   ├── master
│   └── script-handler
└── origin
    ├── HEAD
    └── main
```

More on dangling commits and garbage collection in backup slides

Typical workflow: `init`

- When initialized the repo doesn't track files by default
- The HEAD will point to an **unborn** main branch

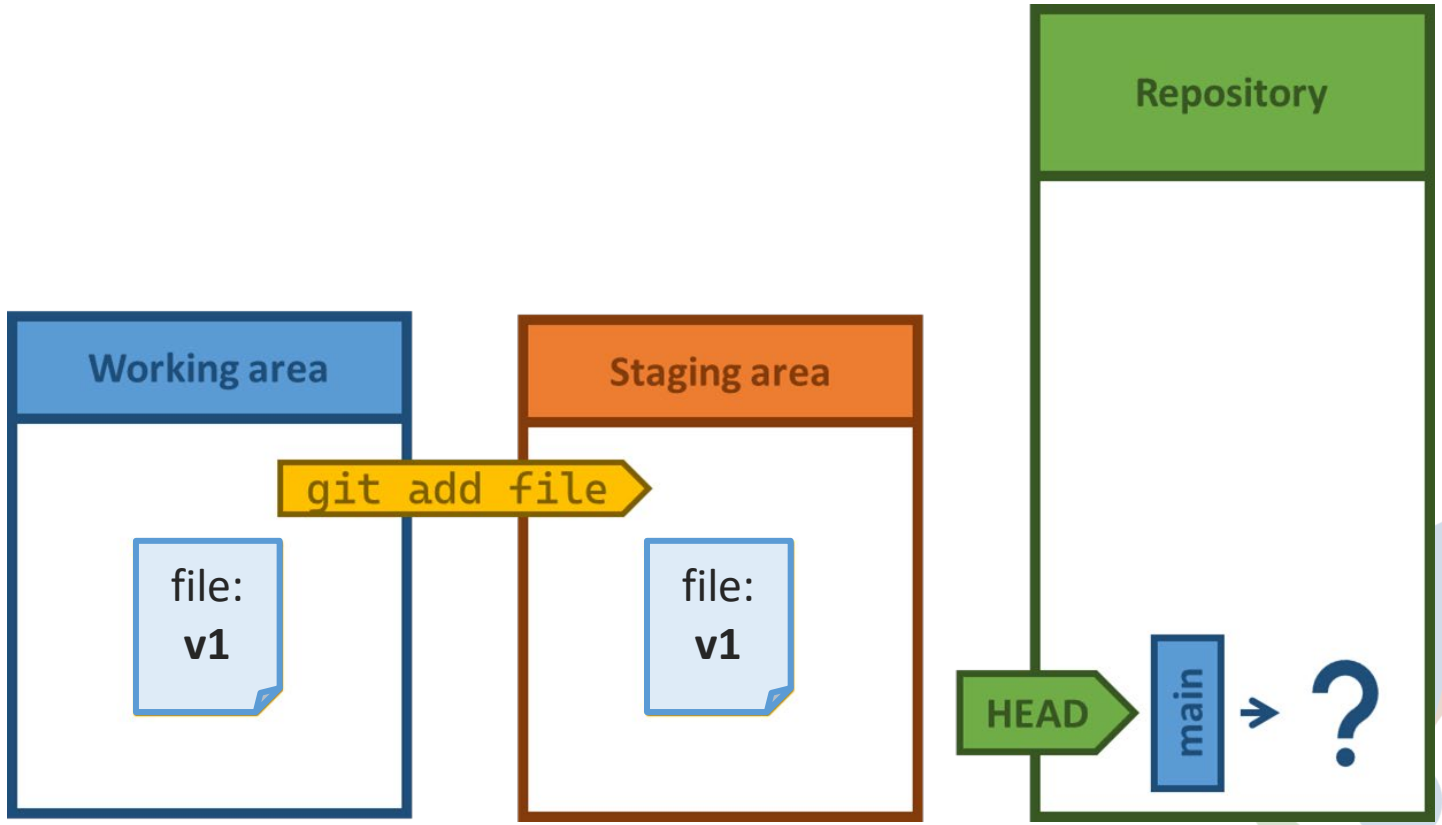
```
$ echo A > file
$ git init
Initialized empty Git
repository in ~/icsc/.git/
$ cat .git/HEAD
ref: refs/heads/main
$ tree .git
.git
├── HEAD
:
└── refs
    ├── heads
    └── tags
```



Typical workflow: add

- Take the content in the working directory and add it to the stage

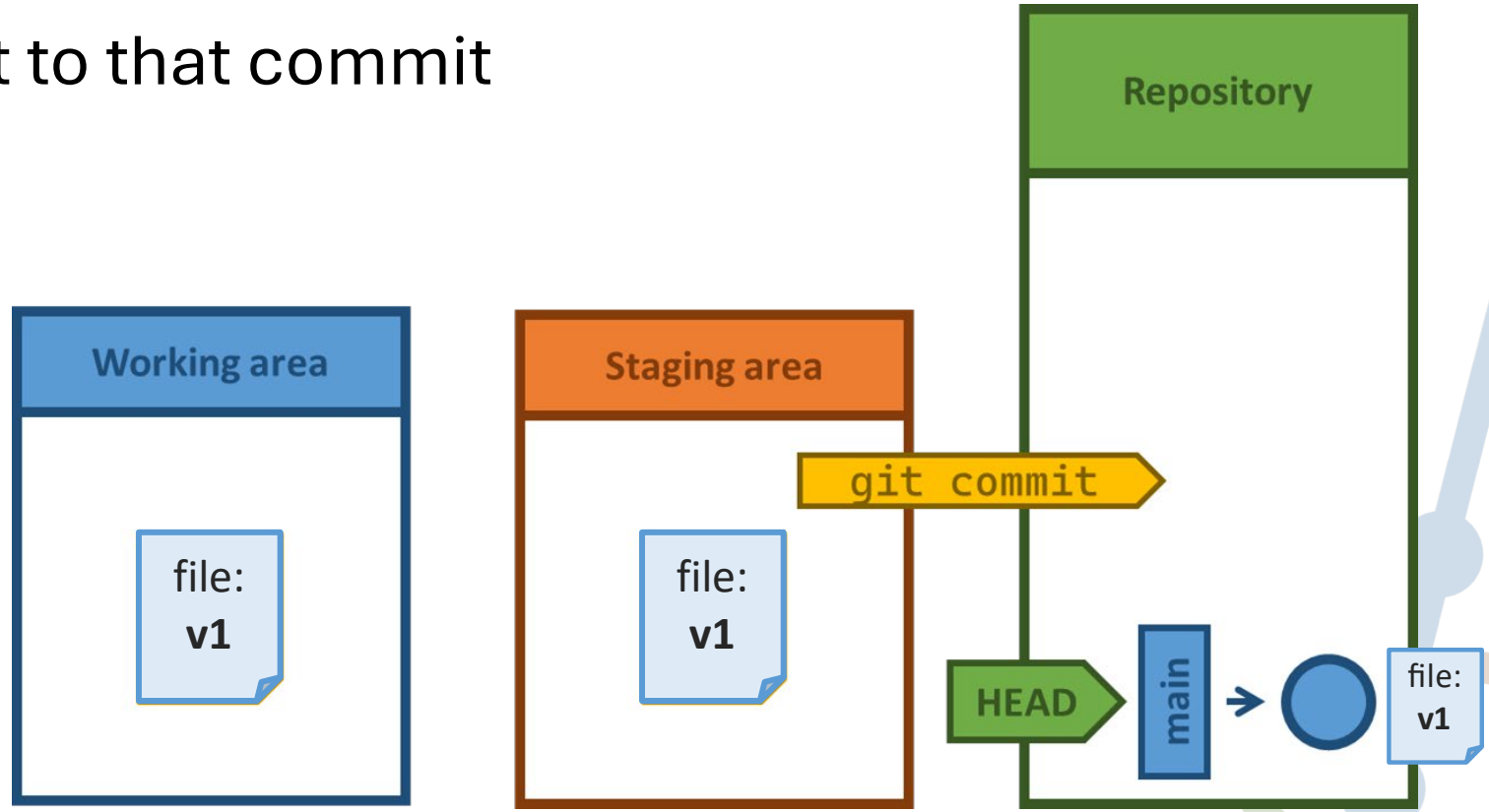
```
$ git add file
$ git status
On branch main

No commits yet

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



Typical workflow: `commit`

- Content of the index saved a permanent snapshot.
- Updates main to point to that commit

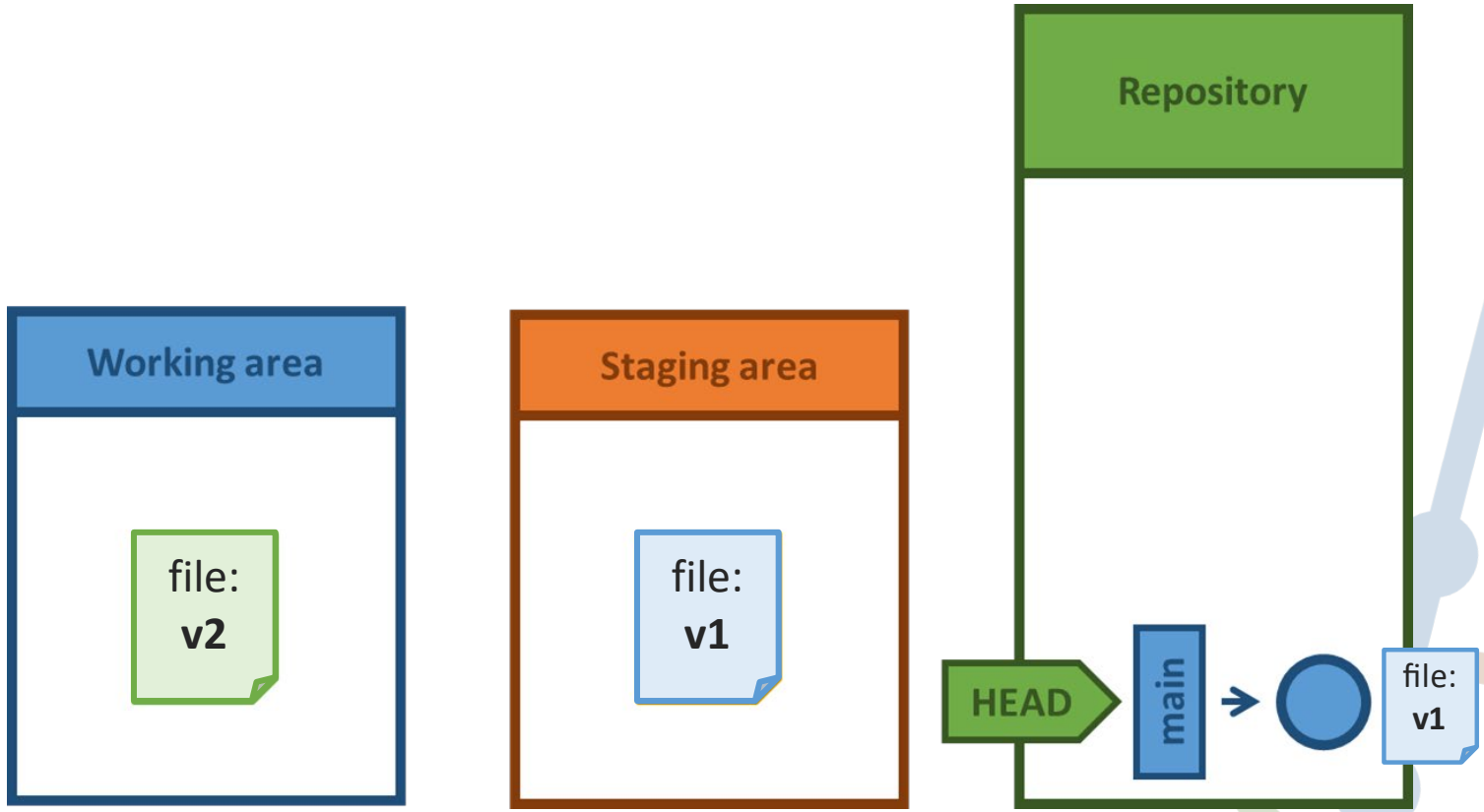
```
$ git commit -m "Initial"
[main (root-commit) 9c59da4]
Initial
 1 file changed, 1
insertion(+)
 create mode 100644 file
$ cat .git/refs/heads/main
9c59da48dac108f810a629084…
```



Typical workflow: editing tracked files

- Changes in tracked files are not automatically added to index

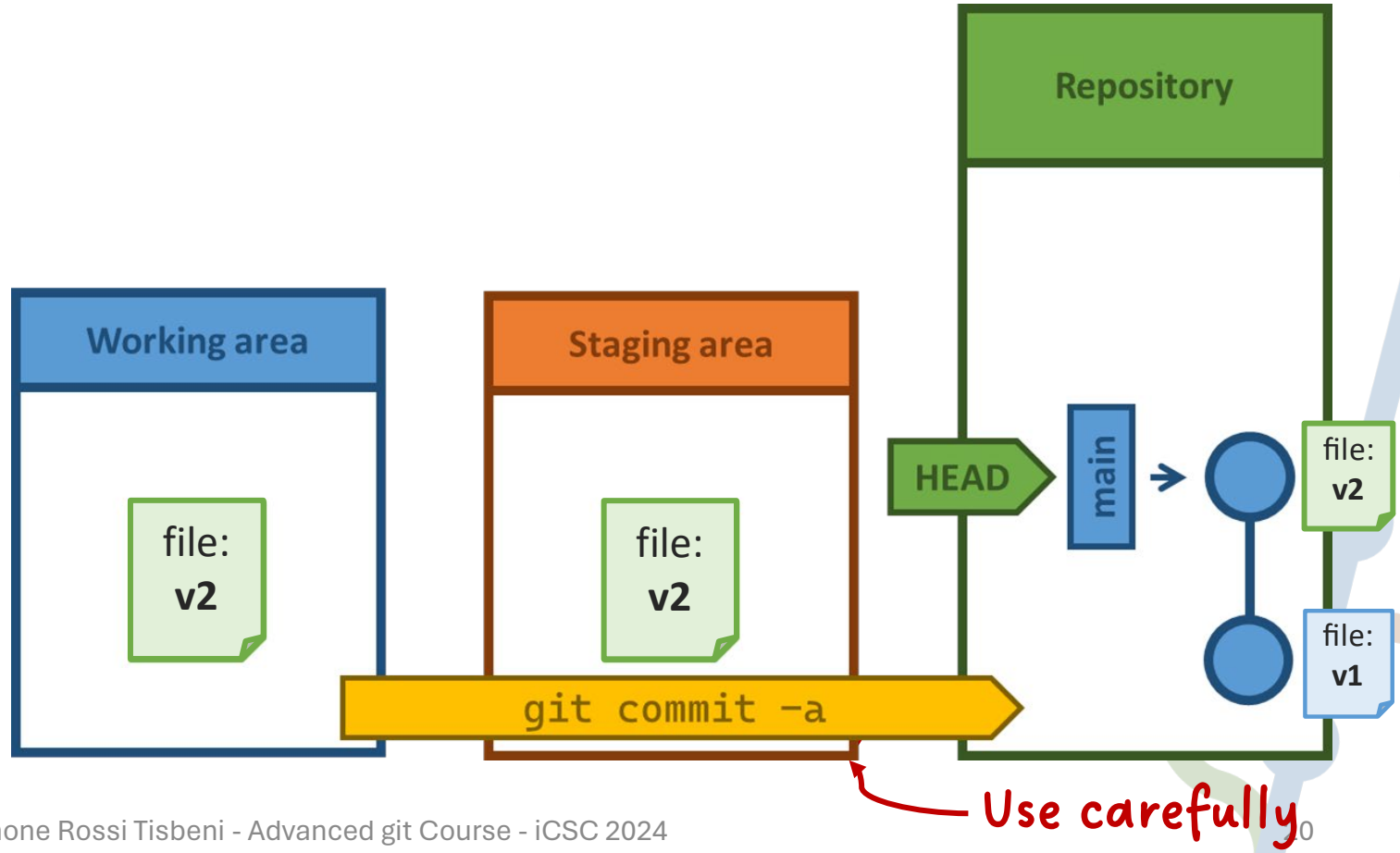
```
$ echo B > file
$ git status
On branch main
Changes not staged for
commit:
…
    modified:   file

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



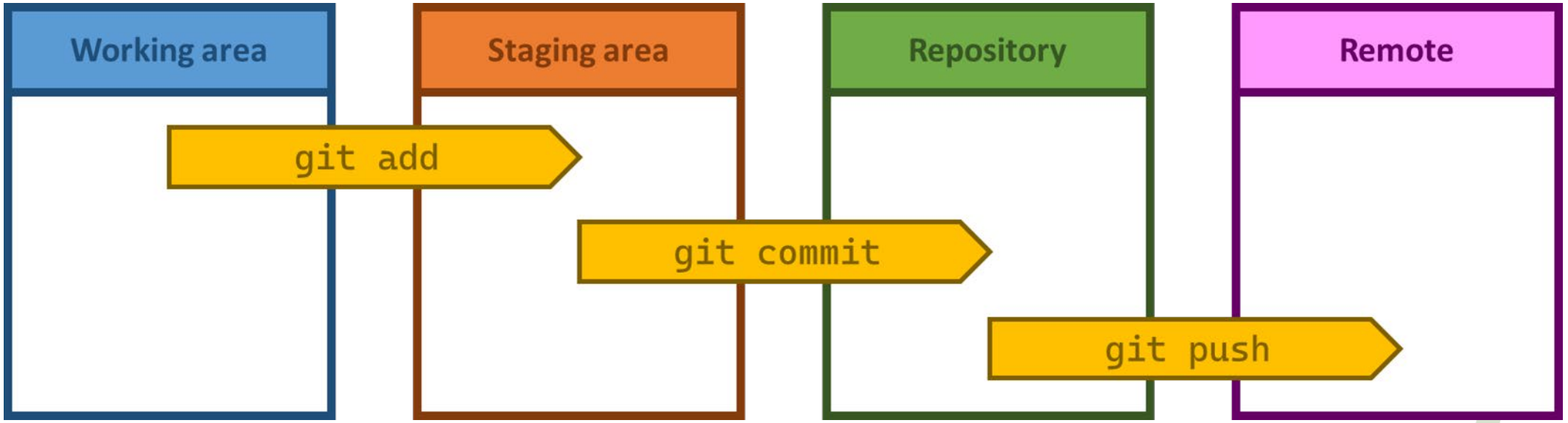
Typical workflow: skip the staging

- Changes in tracked files are not automatically added to index

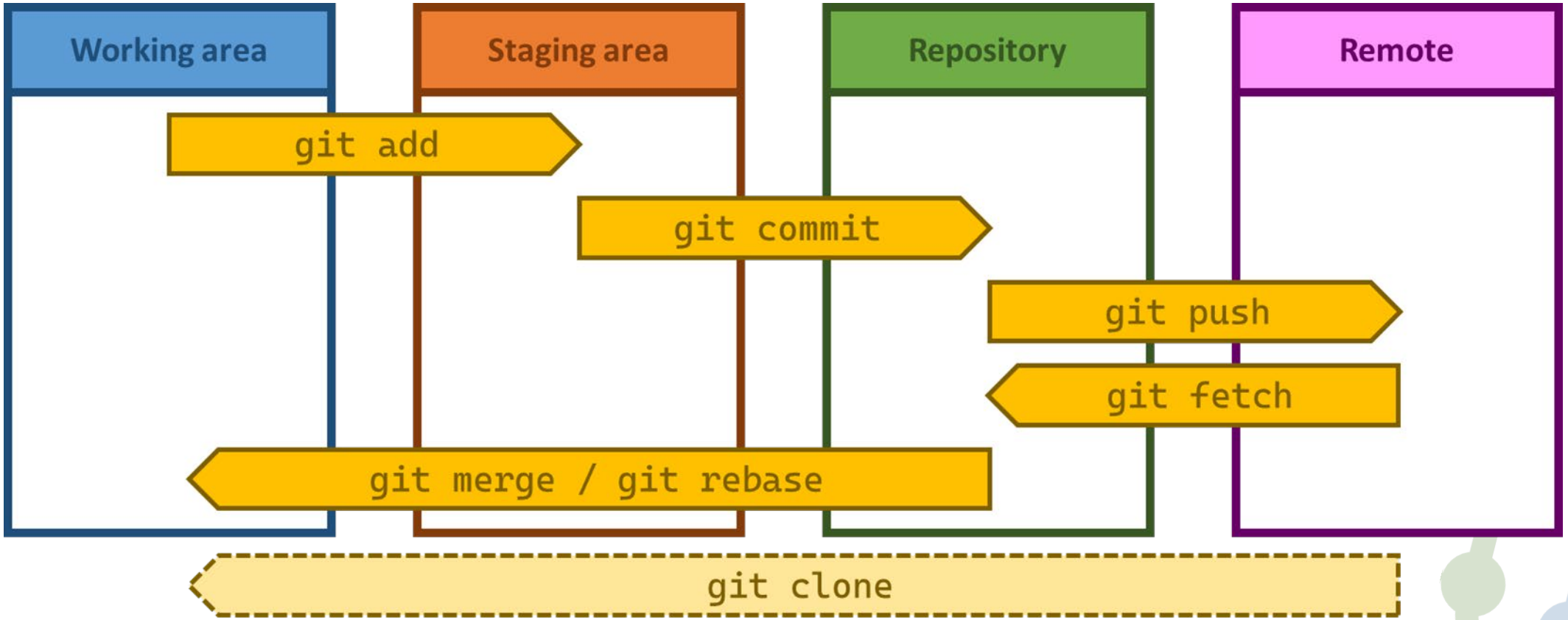
```
$ git commit -a -m
"Modify file"
[main 2520012] Modify
file
 1 file changed, 1
insertion(+), 1
deletion(-)
```



Typical workflow: remote repository



Typical workflow: all the pieces



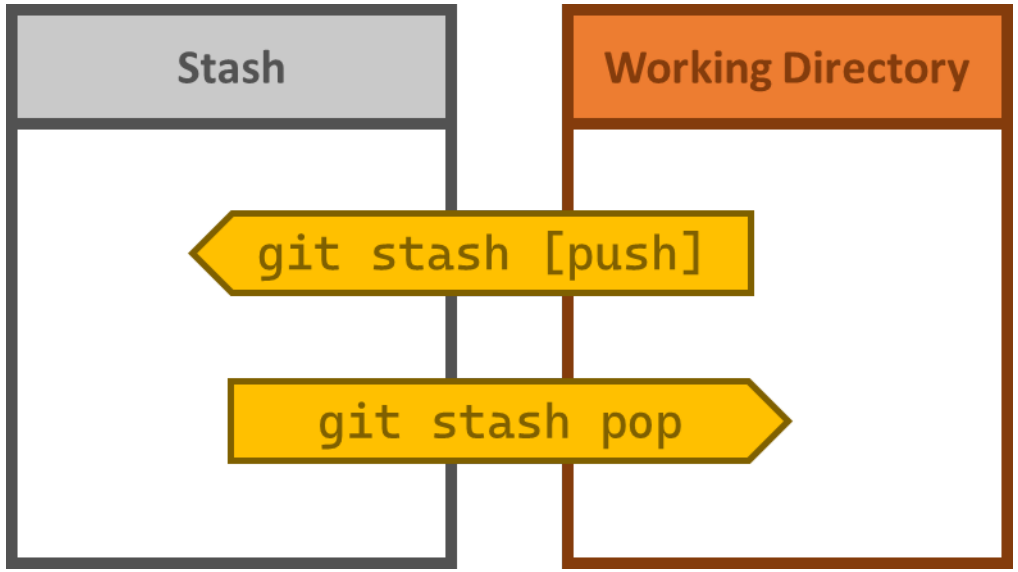
Why advanced git?

We ~~should~~ ~~must~~ want to use Git

- Your colleagues use it
- They expect you to use it
- You work on codebase too large to maintain without
- More than one version of your codebase must be supported
- Your work gets faster and easier
- You understand what you are doing

stash

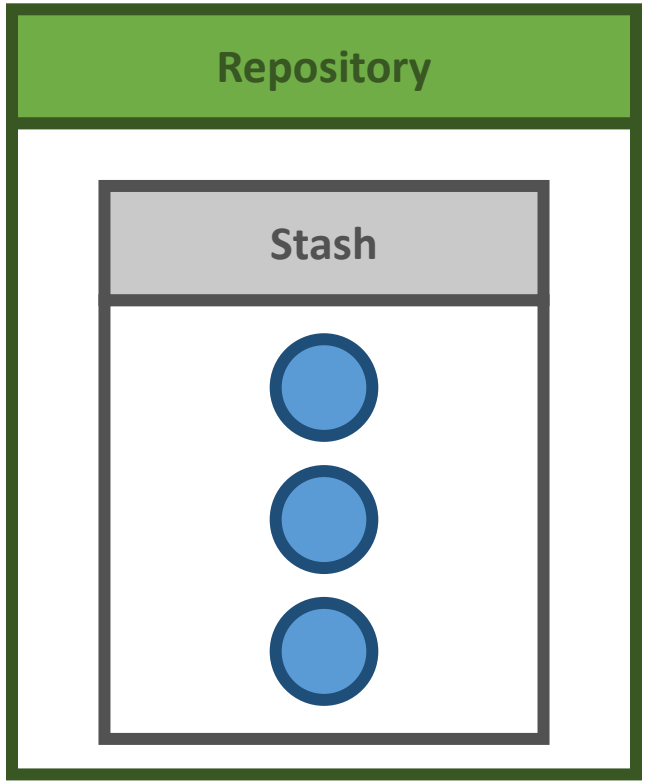
- Stack of independent changes
- Save uncommitted work
- Keep stashed code safe from destructive operation




```
$ git stash push
Saved working directory and index state
WIP on new_branch: 0e70d3b Add repo
description to README.md
```

```
$ git stash pop
On branch new_branch
Changes not staged for commit:
  (use "git add <file>..." to update
what will be committed)
  (use "git restore <file>..." to
discard changes in working directory)
        modified:   README.md

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

stash list



- Stash items are encoded as commits
- Reference to the most recent item in the stack in `.git/refs/stash`

`git stash list`

Show previous stashes:
listed in a **references log**

```
$ git stash list
stash@{0}: WIP on new_branch: 0e70d3b
Add repo description to README.md
stash@{1}: WIP on main: 0e70d3b Add repo
description to README.md
stash@{2}: WIP on main: 5c19be9 Initial
commit
```

`stash` options

git provides multiple options for every command

Simple and complete documentation

`git stash apply`

Like pop, but doesn't remove the item from the stash

`git stash drop`

Removes from the stash without applying to working area

`git stash --keep-index`

Save changes to stash without removing them from the staging area

`git stash --include-untracked`

Add to stash everything in the working area

`git stash branch <branch_name>`

Create new branch starting from stashed changes

patch(-p)

- Allows to stash in hunks
- Interactively!
- Usable also in staging (`git add -p`)

- Useful when there is too much in the same commit

```
$ git stash -p
…
diff display
…
(1/1) Stash this hunk [y,n,q,a,d,e,?]? ?
y - stash this hunk
n - do not stash this hunk
q - quit; do not stash this hunk or any
of the remaining ones
a - stash this hunk and all later hunks
in the file
d - do not stash this hunk or any of the
later hunks in the file
e - manually edit the current hunk
? - print help
```

Close

0 response submitted

git checkout

Scan the QR or use link to join

https://forms.office.com/e/3HPMphsXL3

Copy link

Don't kno

I know it and ometimes use it

I use it regularly

Treemap Bar

1 of 3

Saying goodbye to `checkout`

git checkout is one of the most used command from git

It performs more than one operation

can be difficult to learn: will it do what I want?

Set file content `git checkout <filename>`

discards changes and restore state to index

Changes HEAD `git checkout <branch_name>`

! ambiguous if branch name = filename

sets HEAD to point to different branch

Set file content `git checkout <branch_name> -- <filename>`

discards changes and restore state to that in branch

Changes HEAD `git checkout <commit_sha1>`

sets HEAD to point to that commit, can be DETACHED

`restore` and `switch`

`git restore`

Set file content

`git restore <filename>`

discard changes and restore state to index

`--source <branch_name> <filename>`

discard changes and restore state to the content of the branch

More on restore in backup slides

`git switch`

Changes HEAD

`git switch <branch_name>`

set HEAD to point to a branch

`--detached <commit_sha1>`

set HEAD to point to a commit

Can’t happen unintentionally

`-c <branch_name>`

create a new branch and switch HEAD

restore and switch: Reference sheet

checkout	Change HEAD to:	Which files are changed:	switch/restore
`git checkout filename`	No change	filename	`git restore filename`
`git checkout branch filename`	No change	filename	`git restore --source branch filename`
`git checkout branch`	branch	All files in working dir	`git switch branch`
`git checkout commit`	commit	All files in working dir	`git switch --detach commit`
`git checkout -b branch`	branch	All files in working dir	`git switch -c branch`

Fixing mistakes

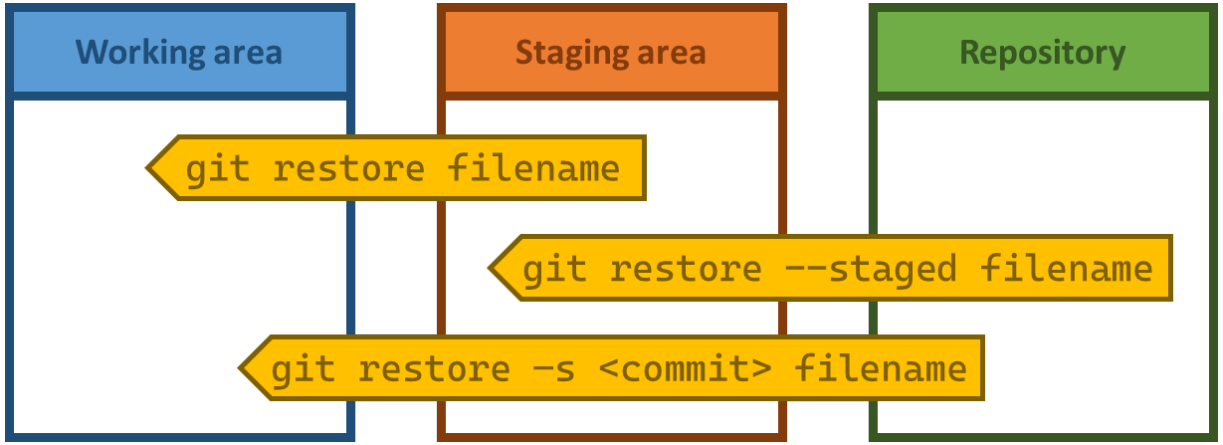
What to do if we staged the wrong files?

What if we already committed some changes?

How do we turn back our repository to a previous state?

- `git restore`
- `git reset`
- `git revert`

Fixing mistakes: `restore`



Warning!
These operations overwrite files in the staging and/or working area without asking for confirmation!

Remove changes not yet committed

`git restore <filename>`

replace the working area copy with the copy from the index

`git restore --staged <filename>`

unstage file, replacing with the copy from HEAD

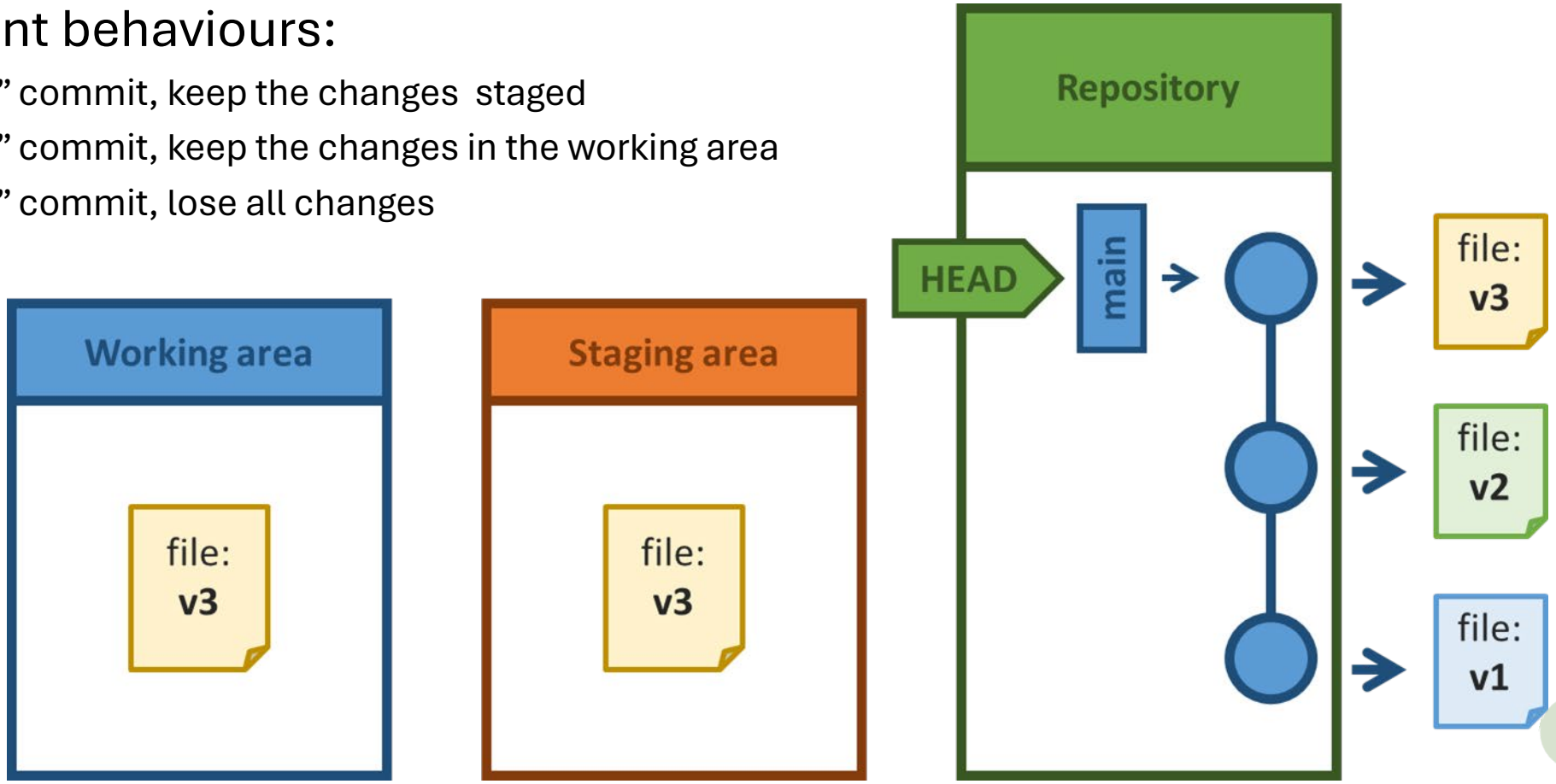
`git restore -s <tree> <filename>`

replace working area copy with the copy from the specified commit, branch or tag

Fixing mistakes: `reset`

different behaviours:

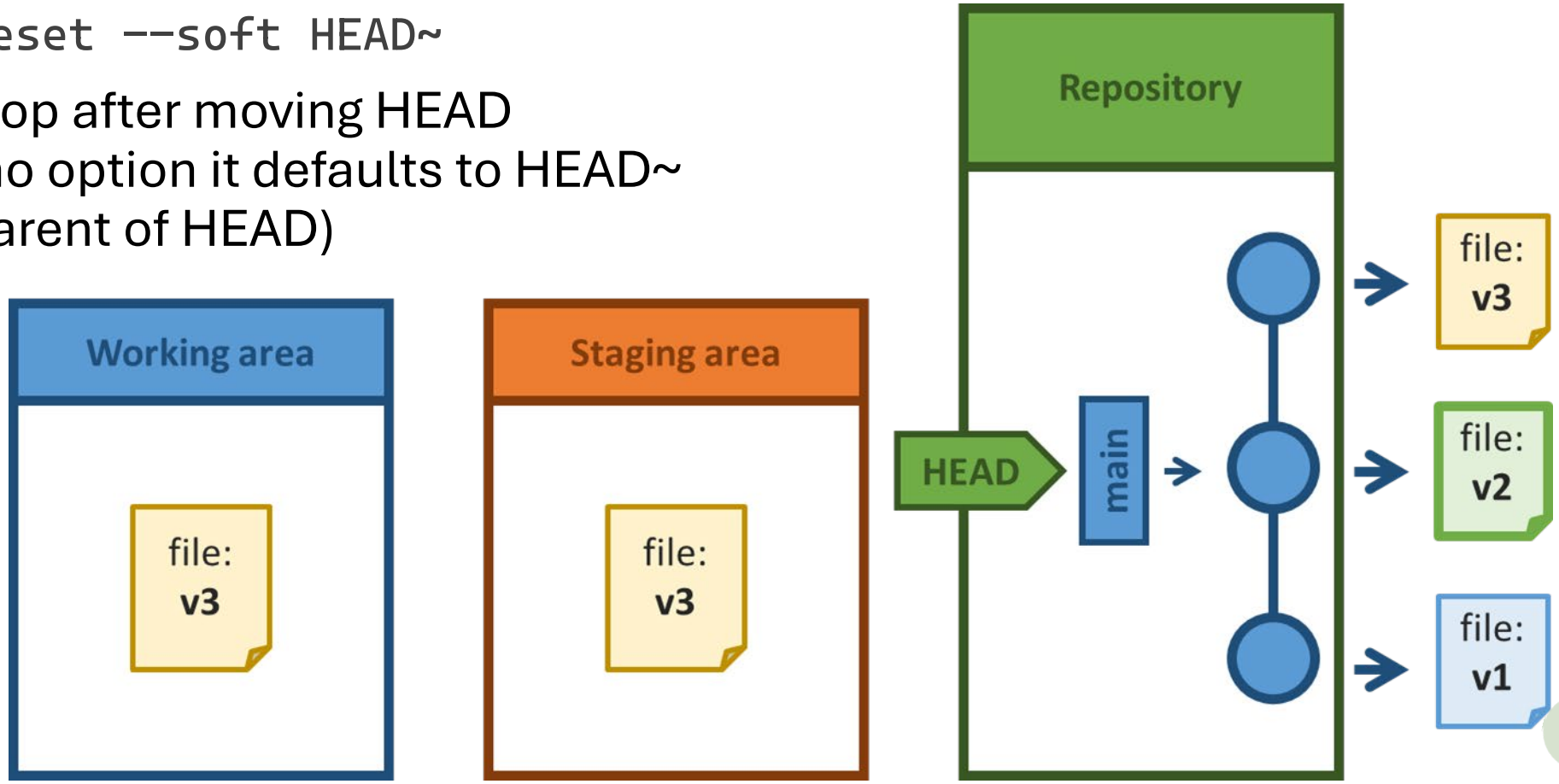
- “undo” commit, keep the changes staged
- “undo” commit, keep the changes in the working area
- “undo” commit, lose all changes



Fixing mistakes: `reset --soft`

```
git reset --soft HEAD~
```

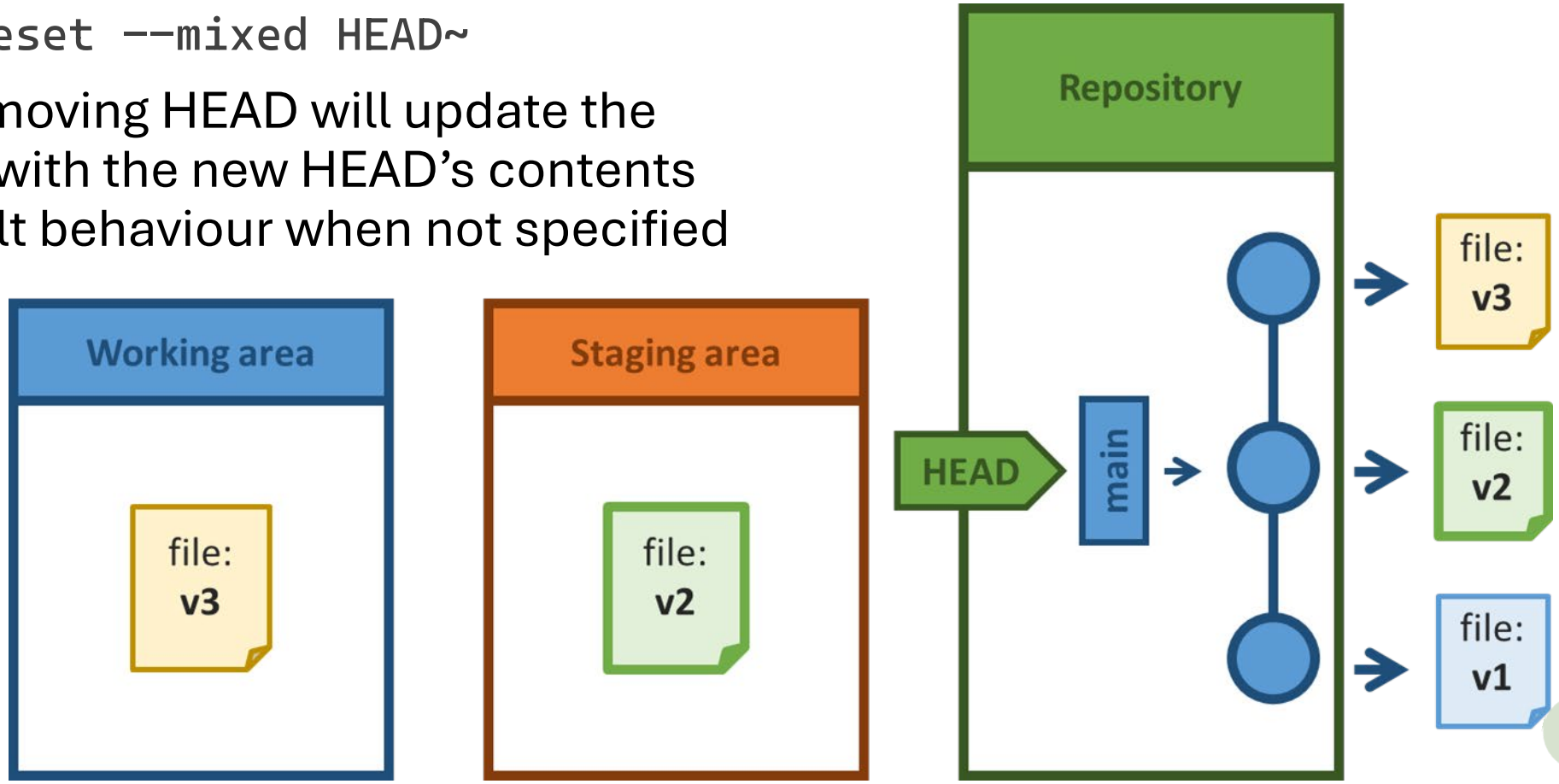
Will stop after moving HEAD
With no option it defaults to HEAD~
(the parent of HEAD)



Fixing mistakes: `reset --mixed`

`git reset --mixed HEAD~`

After moving HEAD will update the index with the new HEAD's contents
Default behaviour when not specified

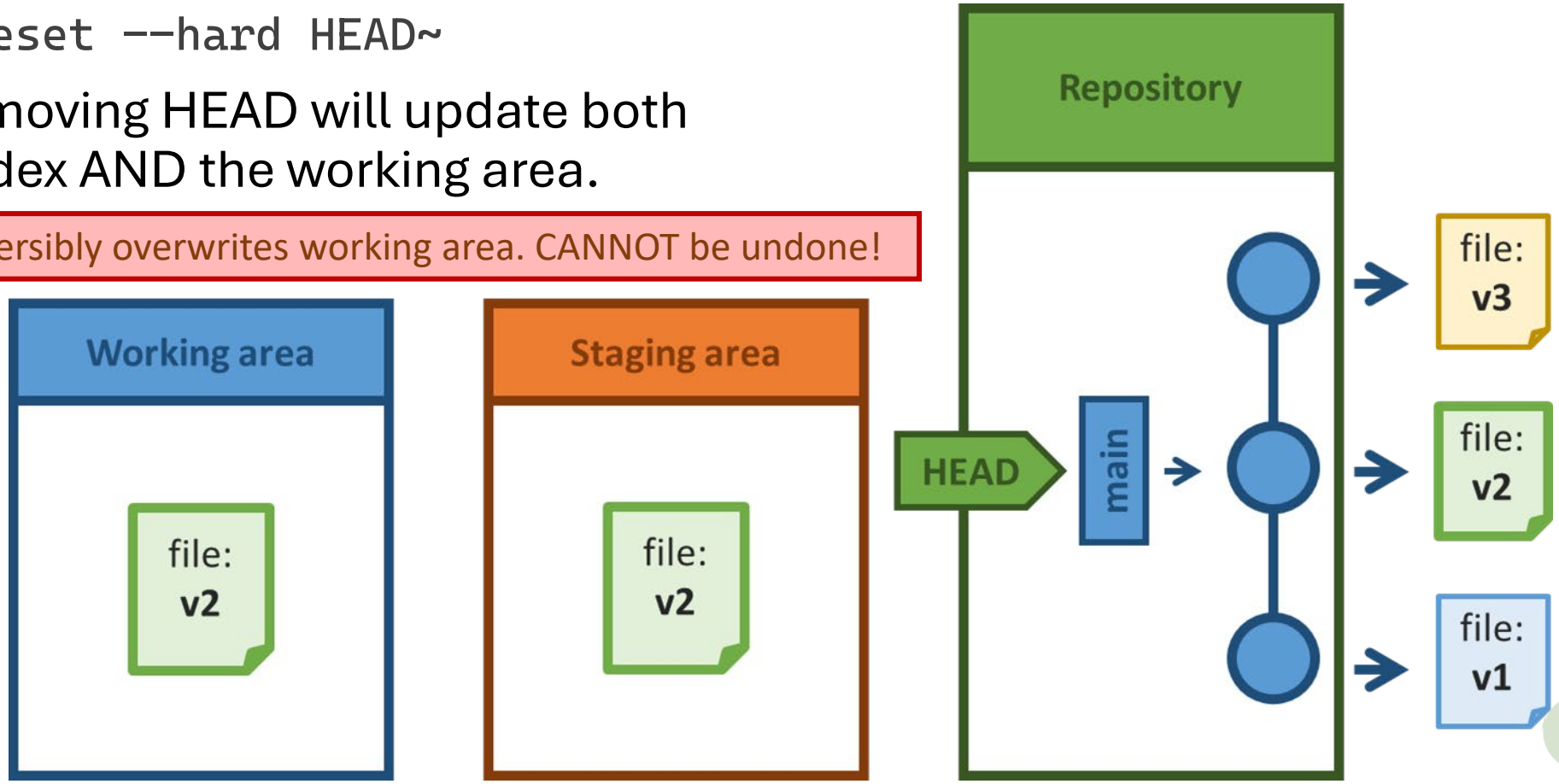


Fixing mistakes: `reset --hard`

```
git reset --hard HEAD~
```

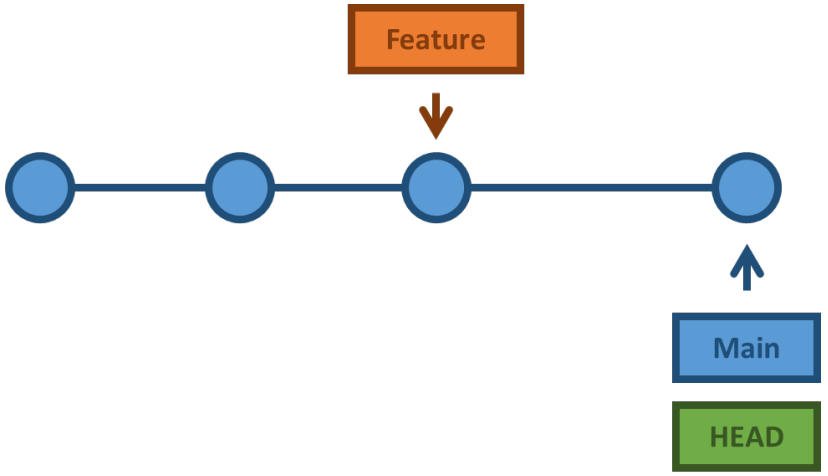
After moving HEAD will update both the index AND the working area.

! Irreversibly overwrites working area. CANNOT be undone!

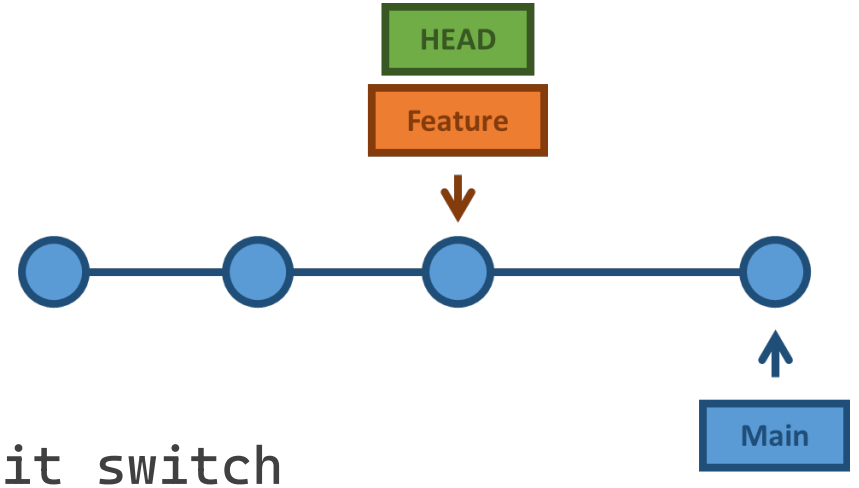


More on reset in backup slides

Moving HEAD

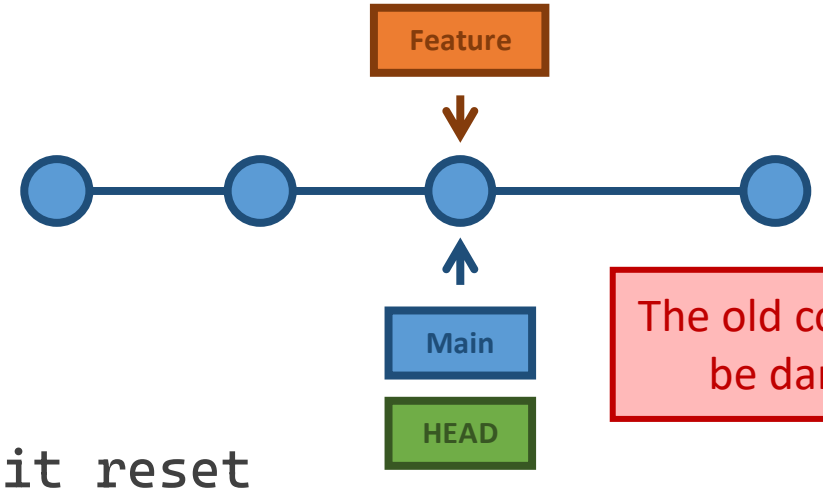


Before the command



`git switch`

moves the HEAD but the branch stays



The old commit will be dangling!

`git reset`

moves the HEAD and the branch ref

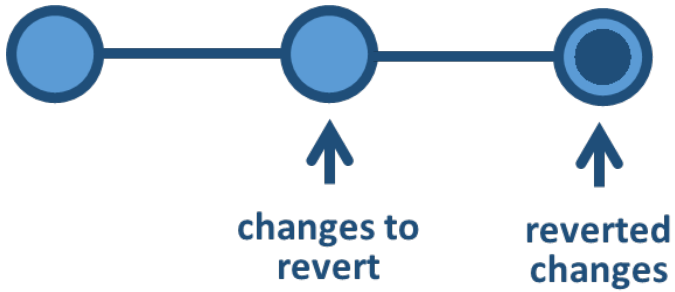
Fixing mistakes: `revert`

```
b39cc8f (HEAD -> main) Add tests
918d81e Add new file to repo
$ git revert 918d
[main 02af80e] Revert "Add new file to repo"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 file
$ git log --oneline
02af80e (HEAD -> main) Revert "Add new file to repo"
b39cc8f Add tests
918d81e Add new file to repo
```

A safer way to undo changes

`git revert`

creates a **new** commit that applies the opposite of the change introduced in a commit



The original commit persist!

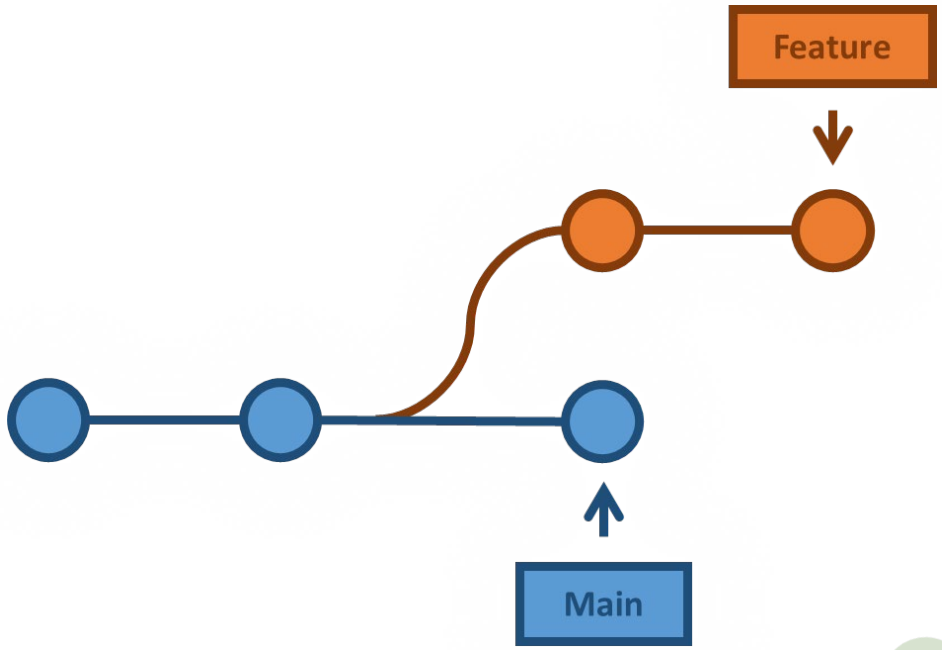
Revert **does not** change history

PART
2

Merge

Merging is Git's way of putting a forked history back together again.

```
$ git merge feature
```



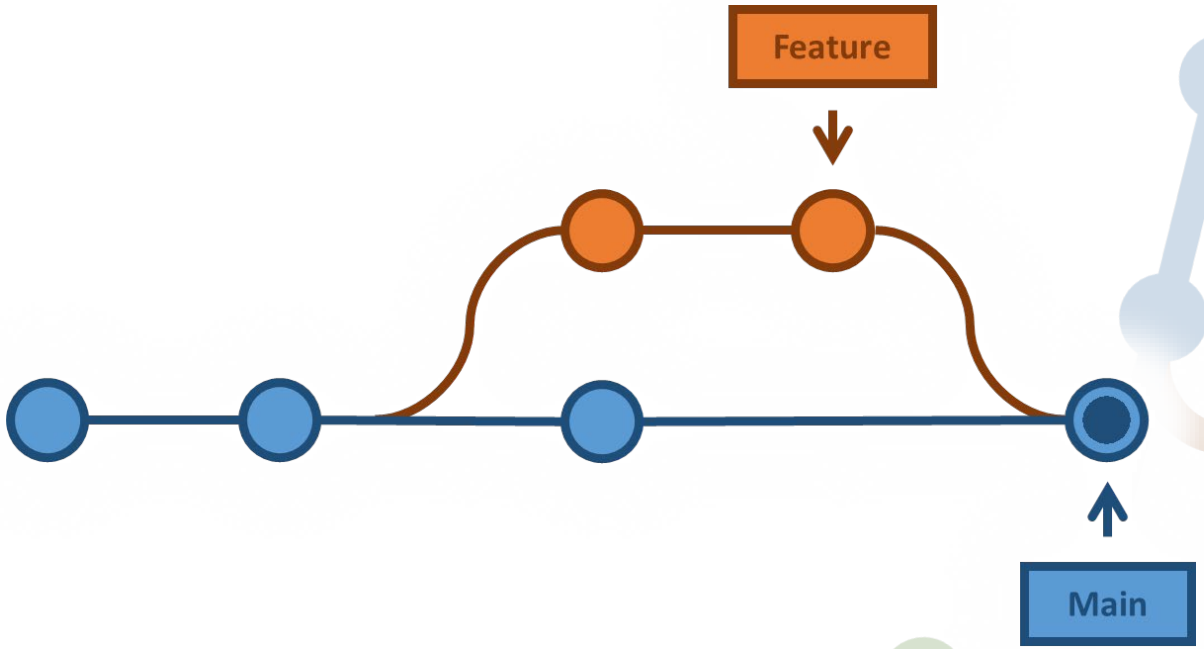
Merge (contd.)

Can merge more than 2 branches

Performs a 3-way merge between the two latest snapshots

Creates a new commit

```
$ git merge feature
Merge made by the 'ort' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
$ git log -n1
commit 5d1870609ce76… (HEAD -> main)
Merge: e63e713 fd8dc20
…
    Merge branch 'feature'
```



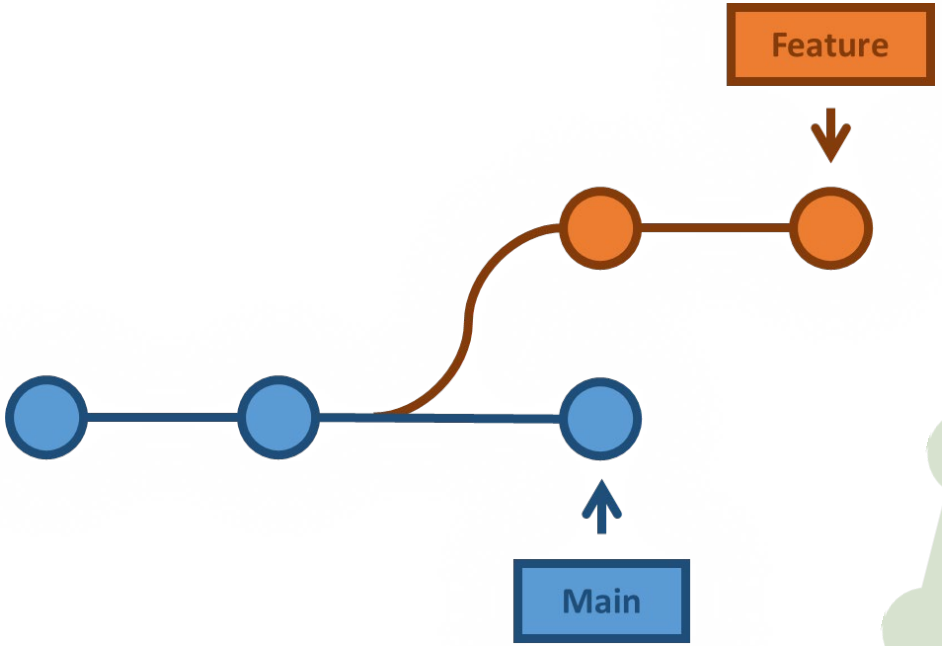
! The new commit will point to 2 parents

Rebase

- git rebase

  Takes the commit from one branch and replays them on a different branch

```
$ git log --oneline --all --graph
* e2610ab (HEAD->feature) Add second file
* 58e5b2c Add first file
| * 8eb64bc (main) Add file to main
|/
* 0e70d3b Add repo description
* 5c19be9 Initial commit
```

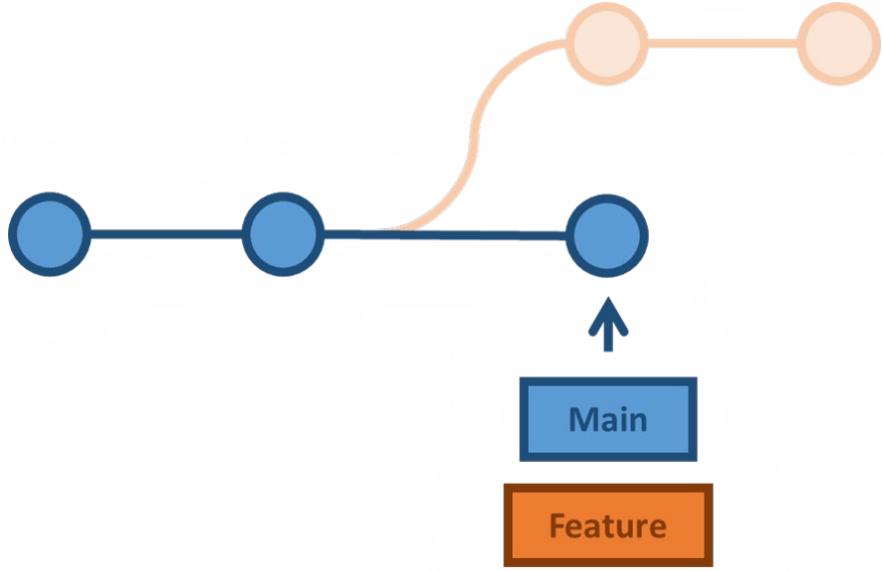


Rebase

- git rebase

  Takes the commits from one branch and replays them on a different branch

```
$ git rebase main

First, rewinding head to replay your work
on top of it...
```

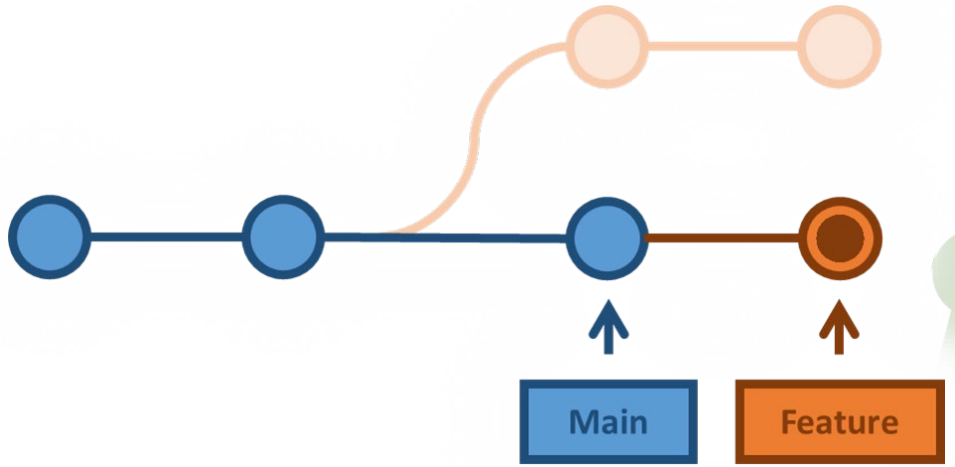


Rebase

- git rebase

  Takes the commits from one branch and replays them on a different branch

```
$ git rebase main

First, rewinding head to replay your work
on top of it...

Applying: Add first file
```



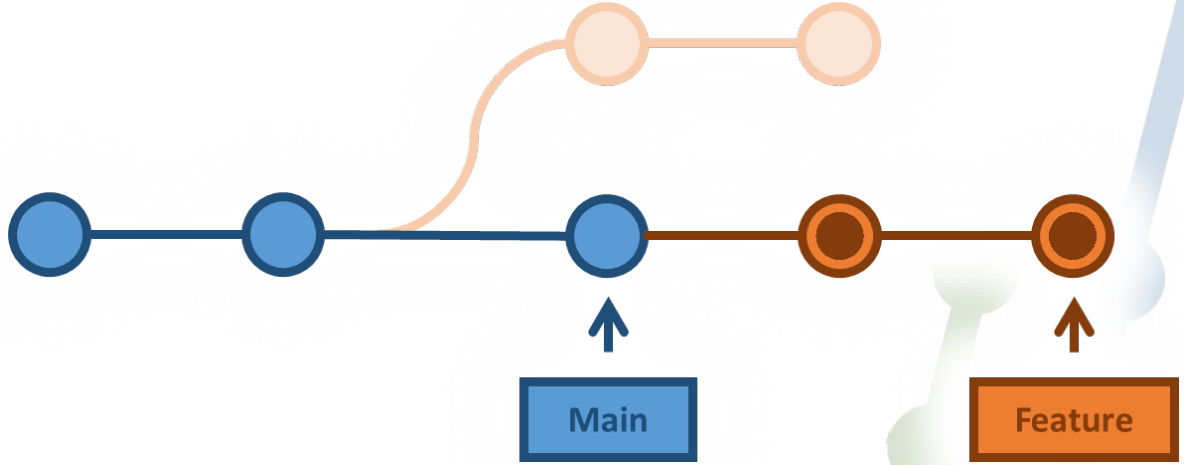
Rebase

- git rebase

  Takes the commits from one branch and replays them on a different branch

```
$ git rebase main
First, rewinding head to replay your work
on top of it...
Applying: Add first file
Applying: Add second file
```

! The old commits now are dangling

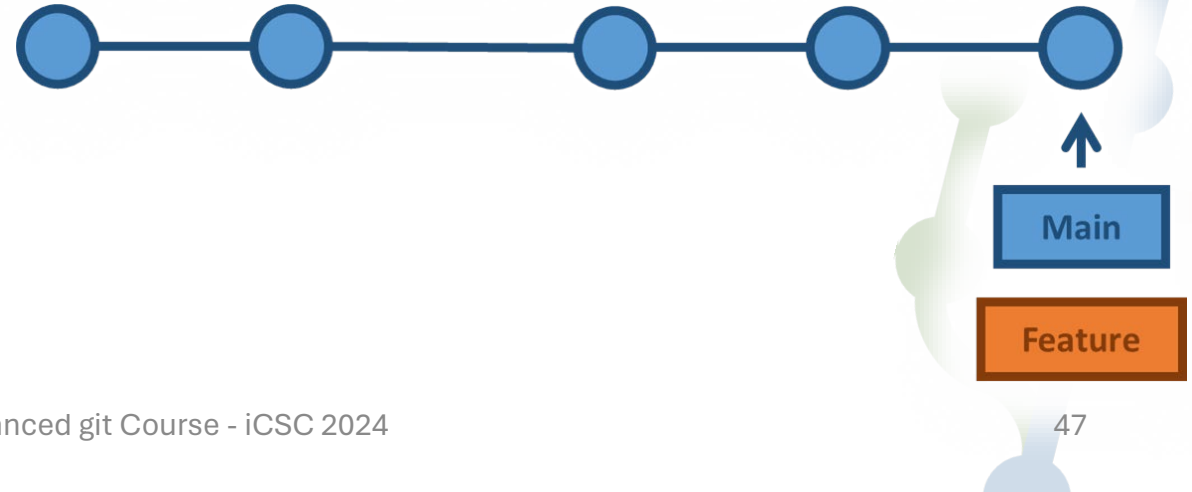


Fast-forward

When no divergent work exists

- Moves the branch pointer to the other's commit location
- Doesn't create a merge commit

```
$ git switch main
$ git merge feature --ff
Updating 8eb64bc..9c0984f
Fast-forward
 file 1 | 0
 file 2 | 0
 2 files changed, 0 insertions(+), 0
deletions(-)
```



Merge vs Rebase

When to use merge

- When you want a clear record of what happened in your repository
- When you want to manually address conflicts
- For long-term branches

! **When you are collaborating and already pushed changes**

When to use rebase

- When you want to show a streamlined history of changes
- When you have short (feature) branches
- When you need to merge the changes back to original branch

! **Rebased remote branches requires** `push --force`

cherry-pick

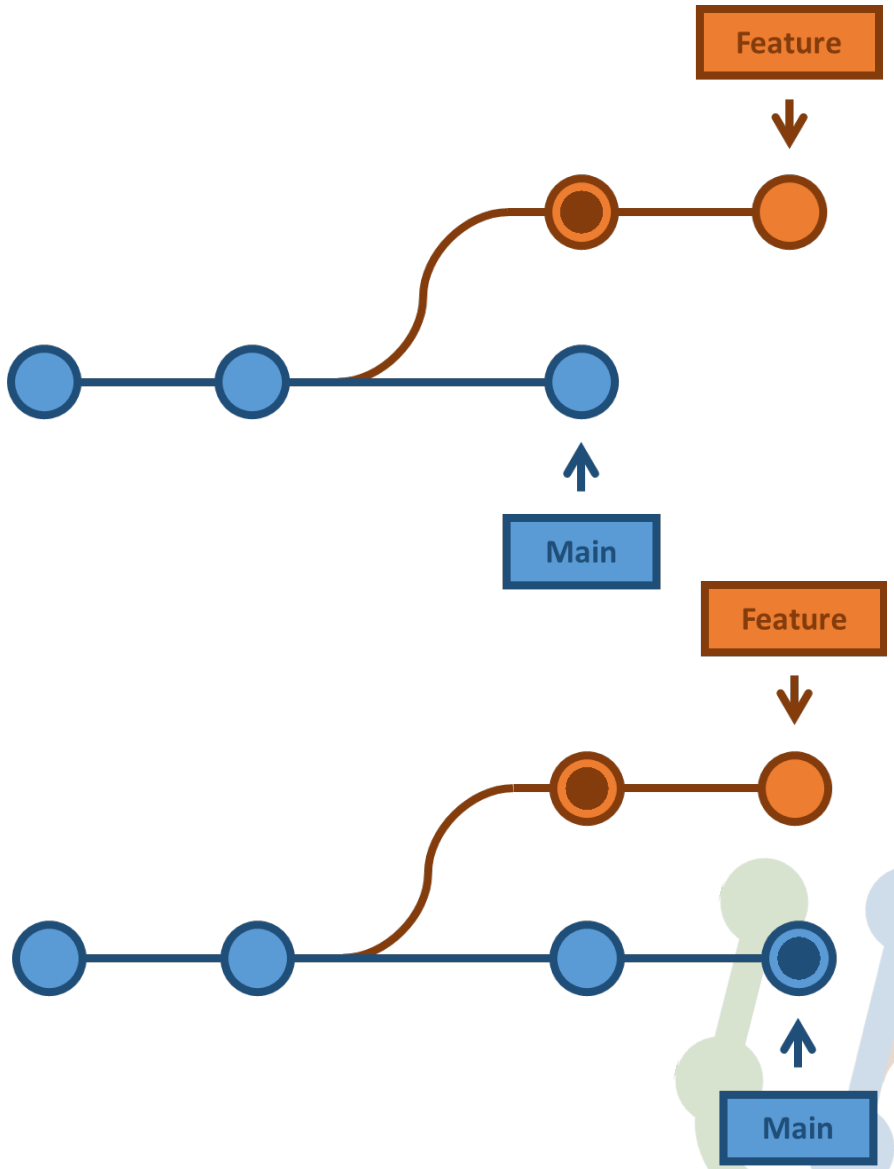
`git cherry-pick`

take the change introduced in a commit and try to re-introduce it on the current branch

- Allows to keep a linear history when merging small changes
- It creates a new commit

!same patch, different date = different SHA-1

- Practically the opposite of revert



reflog

- Background log of HEAD and branches' references
- Few months of history
- Resides in .git/logs/
- Local only!
- Alias for
  `git log -g --oneline`

```
$ git reflog
9c0984f (HEAD -> main, feature) HEAD@{0}:
merge feature: Fast-forward
8eb64bc HEAD@{1}: checkout: moving from
feature to main
9c0984f (HEAD -> main, feature) HEAD@{2}:
rebase finished: returning to
refs/heads/feature
9c0984f (HEAD -> main, feature) HEAD@{3}:
rebase: Add new feature
701dcc4 HEAD@{4}: rebase: Add file B
8eb64bc HEAD@{5}: rebase: checkout main
e2610ab HEAD@{6}: commit: Add new feature
58e5b2c HEAD@{7}: commit: Add file B
5d18706 HEAD@{8}: checkout: moving from
main to feature
e63e713 HEAD@{9}: commit (amend): Add
content to test
```

`reflog`: missing references

- reflog keeps track of commits without references
  - Deleted branches
  - Following reset
  - Orphaned commits
- Why only changes to working area are truly irreversible

```
88207e5 (HEAD -> unstable) Add file_2
57cbedc Add file_1
9c0984f (main) Add new feature
8eb64bc Add file to main
$ git branch -D unstable
Deleted branch unstable (was 88207e5).
$ git log --oneline
9c0984f (HEAD -> main) Add new feature
8eb64bc Add file to main
$ git reflog
88207e5 HEAD@{1}: commit: Add file_2
57cbedc HEAD@{2}: commit: Add file_1
9c0984f (HEAD -> main) HEAD@{3}: checkout:
moving from main to unstable
```

@{…} reference

HEAD@{2}

the value of HEAD 2 steps prior

You can use to see where a branch was some time ago:

main@{yesterday}

HEAD@{2.months.ago}

```
$ git show main@{1.week.ago}
commit e63e713b280daa51edf2549d20fd4f0…
Author: Simone Rossi Tisbeni
<simone.rossitisbeni@unibo.it>
Date:   Wed Feb 7 14:07:23 2024 +0100

    Add content to test
```

Rewriting history: `amend`

When you realize you have missed something in the latest commit

`git commit --amend`

Create new version of the most recent commit. Old one stays dangling.

- Loads the previous commit message in editor
- If present, will add staged changes to commit

! It will ALWAYS change the SHA-1 of the commit

`git commit --amend --no-edit`

For trivial commit that don’t need changes in message

Rewriting history: `rebase`

When you need to rewrite multiple commits

- `rebase` allows you to replay commits

`git rebase -i`

launches rebase interactively

- Can stop after each commit, to allow you to edit it

Accepts as argument the **parent** of the last commit you want to edit

`git rebase -i HEAD~3`

Rebase 3 commits: in the range `HEAD~3..HEAD`

`git rebase -i 8eb64^`

Rebase 3 commits: from `8eb64`'s parent (excluded)

```
$ git log HEAD~3..HEAD --oneline
5856c7c (HEAD -> main) Add new feature
701dcc4 Add file B
8eb64bc Add file A to main
```

Rewriting history: `rebase` todo list

- Interactive rebase opens the editor with a list of commands

`<command> <commit> <message>`

- “script” of commands to be played
- Commits listed in the opposite direction of log
- Save and close the editor to run the script

```
pick 8eb64bc Add file A to main
pick 701dcc4 Add file B
pick 5856c7c Add new feature

# Rebase 5d18706..5856c7c onto 5d18706 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
...
```

Rewriting history: `rebase` options

- `pick`
  use (replay) this commit
- `reword`
  use commit, but stop to edit message
- `edit`
  use commit, but stop to amend
- `squash`
  use commit, but combine with previous, stop to edit message
- `fixup`
  like squash, but keep the previous commit message
- `exec <command>`
  run an arbitrary shell command
- `break, drop, label, reset, merge`

Rewriting history: `rebase` interrupted

- You can put a break in a rebase to drop you to the command line
- Stops also when a command fails or when there is a conflict

`git rebase --continue`

will proceed through the list of commands after the break

`git rebase --abort`

will interrupt and return the repo to the state it was before

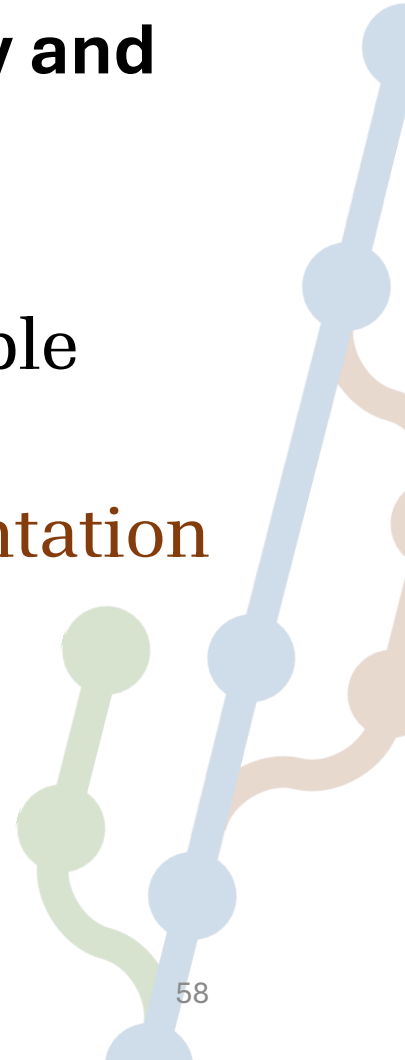
`git rebase --edit-todo`

allows to make changes to the todo list during rebase

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

- the official git documentation



Close

0 response submitted

Have you ever pushed on main after rewriting it's history?

Scan the QR or use link to join

https://forms.office.com/e/LmpByiXkve

Copy link

Sir, I'd ne

uilty as charged

I do it and I don't regret it!

Treemap Bar

1 of 1

Force push with lease

A safer alternative to traditional force push

`git push --force-with-lease`

Pushes only if the remote ref has not changed

! Still better not to force push: it will not consider other people's work that has been based on the changes to be overwritten

Rewriting history: the hard way

Git provides an in-built tool to alter the repo's history in drastic way

`git filter-branch`

- rewrite a large number of commits in a scriptable way
- can easily modify metadata

! no longer recommended by Git

Replaced by the `git-filter-repo` python tool

https://github.com/newren/git-filter-repo

Rewriting history: `git filter-repo`

Single file python script

Faster and safer* than git filter-branch for large repositories

Still a highly destructive operation!

Allows to

- Filter commits based on author, date, file path,...
- Move files or directories, rename branches
- Remove data completely from the entire history

Rewriting history: `filter-repo` warnings

! Rewriting history can break builds, references, and integrations

! Rewritten history cannot be easily reverted

! Don’t use it if other people have based work off of the repo!

Backup your work

Work on a clean copy of the repo

Test your rewrites rigorously before applying to original

```
$ git filter-repo --replace-text replacements.txt

Aborting: Refusing to destructively overwrite repo history since this does not look like a fresh clone.

  (expected at most one entry in the reflog for HEAD)

Please operate on a fresh clone instead. If you want to proceed anyway, use --force.
```

Rewriting history: removing sensitive data

```
2a0f46a (HEAD -> main) Removed secret file
diff --git a/secret b/secret
deleted file mode 100644
…
-token: ABC123DEF456
-username: srossiti
b1a8053 Removed my password
diff --git a/secret b/secret
…
 token: ABC123DEF456
 username: srossiti
-password: dolphin
```

```
6af6473 Add token to secret file
diff --git a/secret b/secret
…
+token: ABC123DEF456
 username: srossiti
 password: dolphin
ef2850 Add secret file
diff --git a/secret b/secret
new file mode 100644
…
+username: srossiti
+password: dolphin
```

Rewriting history: removing sensitive data

```
$ echo dolphin > replacements.txt
$ git filter-repo --replace-text replacements.txt
Parsed 18 commits
New history written in 0.04 seconds; now repacking/cleaning...
Repacking your repo and cleaning out old unneeded objects
HEAD is now at a2844e7 Removed secret file
Enumerating objects: 40, done.
Counting objects: 100% (40/40), done.
Delta compression using up to 8 threads
Compressing objects: 100% (33/33), done.
Writing objects: 100% (40/40), done.
Total 40 (delta 14), reused 0 (delta 0), pack-reused 0
Completely finished after 0.10 seconds.
```

```
a509aab Removed my password
diff --git a/secret b/secret
…
 token: ABC123DEF456
 username: srossiti
-password: ***REMOVED***
…
ef2850f Add secret file
diff --git a/secret b/secret
new file mode 100644
…
+username: srossiti
+password: ***REMOVED***
```

--replace-message to modify commit's messages

Rewriting history: removing sensitive data

```
$ cat replacements.txt
dolphin
foo==>bar
glob:*666*==>
regex:\bdriver\b==>pilot
literal:MM/DD/YYYY==>YYYY-MM-DD
regex:([0-9]{2})/([0-9]{2})/([0-
9]{4})==>\3-\1-\2
```

Replace `dolphin` with `***REMOVED***`

Replace `foo` with `bar`

Replace lines containing `666` with a blank line

The word `driver` with `pilot` (but not `drivers`)

Replace the exact text `MM/DD/YYYY` with `YYYY-MM-DD`

Replace date of the form `MM/DD/YYYY` with the form `YYYY-MM-DD`

Rewriting history: changing author

- Replace wrong email and/or username
- Uses a mailmap file

```
$ cat mailmap
Name <email@addre.ss>
<new@ema.il> <old1@ema.il>
New Name <new@ema.il> <old2@ema.il>
New Name <new@ema.il> Old Name <old3@ema.il>
```

```
$ git log --pretty="%h %an <%ae>"
a3f82d7 Simone Rossi Tisbeni
<simone.rossitisbeni@unibo.it>
49a6711 Simone Rossi Tisbeni
<private@email.it>
82ac0e8 Simone Rossi Tisbeni
<private@email.it>
ef2850f Simone Rossi Tisbeni
<simone.rossitisbeni@unibo.it>
5856c7c Simone Rossi Tisbeni
<simone.rossitisbeni@unibo.it>
```

Rewriting history: changing author

```
$ git filter-repo --mailmap mailmap
Parsed 18 commits
New history written in 0.02 seconds; now repacking/cleaning...
Repacking your repo and cleaning out old unneeded objects
HEAD is now at a2844e7 Removed secret file
Enumerating objects: 40, done.
Counting objects: 100% (40/40), done.
Delta compression using up to 8 threads
Compressing objects: 100% (20/20), done.
Writing objects: 100% (40/40), done.
Total 40 (delta 14), reused 36 (delta 13), pack-reused 0
Completely finished after 0.06 seconds.
```

```
$ git log --pretty="%h %an <%ae>"
a2844e7 Simone Rossi Tisbeni <simone.rossitisbeni@unibo.it>
a509aab Simone Rossi Tisbeni <simone.rossitisbeni@unibo.it>
60c4622 Simone Rossi Tisbeni <simone.rossitisbeni@unibo.it>
ef2850f Simone Rossi Tisbeni <simone.rossitisbeni@unibo.it>
5856c7c Simone Rossi Tisbeni <simone.rossitisbeni@unibo.it>
```

Rewriting history: removing large binaries

```
$ git log --oneline
54275a3 (HEAD -> main) Ops, removed large binary
2a20afa Add a new thing
701dcc4 Add file B
8eb64bc Add file A to main
$ git filter-repo --analyze
Processed 8 blob sizes recursively: ".git/filter-repo/analysis"
Processed 20 commits
Writing reports to .git/filter-repo/analysis...done.
```

```
$ head .git/filter-repo/analysis/path-all-sizes.txt
=== All paths by reverse accumulated size ===
Format: unpacked size, packed size, date deleted, path name

  10485760  45791 2024-02-21 large.bin
       116    120 <present>  README.md
         0      9 <present>  A
         0      9 2024-02-07 feature
         0      9 <present>  B
```

Rewriting history: removing large binaries

```
$ git filter-repo --invert-path --path large.bin
Parsed 20 commits
New history written in 0.01 seconds; now repacking/cleaning...
Repacking your repo and cleaning out old unneeded objects
HEAD is now at a2844e7 Removed secret file
Enumerating objects: 40, done.
Counting objects: 100% (40/40), done.
Delta compression using up to 8 threads
Compressing objects: 100% (19/19), done.
Writing objects: 100% (40/40), done.
Total 40 (delta 14), reused 40 (delta 14), pack-reused 0
Completely finished after 0.06 seconds.
```

```
$ git log --oneline
701dcc4 (HEAD -> main) Add file B
8eb64bc Add file A to main
$ head .git/filter-repo/analysis/path-all-sizes.txt
=== All paths by reverse accumulated size ===
Format: unpacked size, packed size, date deleted, path name

        116    120 <present>  README.md
          0      9 <present>  A
          0      9 2024-02-07 feature
          0      9 <present>  B
```

Git Hooks

Scripts that runs when a **git event** occurs

- Located in `.git/hooks`
- With `git init` some .sample hooks are created by default
- Any executable script will work
- Can be manually run with

```
git hook run <hook-name> -- <hook-arguments>
```

- Hooks are all run from the root of the working area

Git Hooks: committing hooks

- `pre-commit`
  Run first, before the message is typed. Exiting non-zero aborts the commit.
  i.e. Check code format, lint, tests…
- `prepare-commit-msg`
  Before the editor, after the default message is created.
  i.e. programmatically edit templated commit (merge, squash, amends…)
- `commit-msg`
  After the commit message is written. Exiting non-zero aborts the commit.
  i.e. validate the commit message format
- `post-commit`
  After the entire commit process is completed.
  i.e. notification, logging…

Git Hooks: client-side hooks

- `pre-rebase`
  Runs before any rebase. Exiting non-zero aborts the rebase.
  i.e. disallow rebase on unsafe conditions
- `post-merge`
  Runs after a successful merge.
  i.e. restore data untracked by git, permissions…
- `post-checkout`
  Runs after checkout and switch.
  i.e. auto show diffs, move data untracked by git…
- `pre-push`
  Runs during push, after the remote has been update, but before any transfer. Exiting non-zero aborts the push.
  i.e. run tests before push, prevent force push

Git Hooks: server-side hooks

These hooks are handled from the receiving repository (server)

- `pre-receive`
  Run when handling a push from a client. Exiting non-zero aborts the push
- `update`
  As with `pre-receive` but once for every branch pushed. Exiting non-zero rejects only one reference at a time
- `post-receive`
  After the entire push is completed.

Sample Hook: disallow unsafe rebase

```
$ cat .git/hooks/pre-rebase
#!/bin/sh
branch="$2"
[ -n "$branch" ] || branch=`git rev-parse
 --abbrev-ref HEAD`

if git config init.defaultBranch > /dev/null; then
  main_branch=$(git config init.defaultBranch)
else
  main_branch="master"
fi

if [ "$branch" = "$main_branch" ]; then
    echo “Rebase on $main_branch branch is not
allowed."
    exit 1
fi
```

```
$ chmod -x .git/hooks/pre-rebase

$ git rebase -i HEAD~2

Rebase on main branch is not allowed.

fatal: The pre-rebase hook refused to
rebase.
```

`log`

`git log`

the basic command that shows you the history of your repository

Its functionalities are extended by many options

`git log --pretty=<format>`

Allows to print logs with different formats

i.e. `oneline`, `full`, `reference`, custom...

`git log --graph`

draw a text-based graphical representation of the branches

`log`: partial display

Revision range

Defaults to show the entire history up to `HEAD`

Specify a range as argument:

- `HEAD~2..HEAD`
- `origin..HEAD`
- `main..feature`

…

Commit limiting

By default, shows all commit visible in range

- `-n 5`
- `--since=yesterday`
- `--before=2.weeks.ago`
- `--author="Simone Rossi"`
- `--grep "#\d+"`

…

`log`: searching

If you are interested in **when** code was introduced or changed

`git log -S string`

will show all commits where string was added (or removed)

`git log -L 10,20:file`

will show the evolution of lines 10 to 20 of file

`git log -L:myFunction:file`

will show the evolution of the function that matches the regex in file.
It will try to find the boundaries of the function.

blame

`git blame filename`

Used to tell the author and date of the last changes in a file

`git blame HEAD~2 -- filename`

You can specify a point in the history (commit, branch, ...)

`-L`

restrict the changes to specific lines

`-M`

Tracks line moved within the file, blaming the original author

`-C -C -C`

Tracks lines moved between files, in the same commit
In the first commit the file appear
In any other commit

bisect

`git bisect start HEAD known_good_commit`

To start a binary search to find an issue

Git will checkout the middle commit test and keep looping until it finds the first bad commit.

`git bisect bad`

To mark a non-working commit as bad

`git bisect good`

To mark a known working commit as good

`git bisect reset`

Will stop the cycle and reset your HEAD

`git bisect run test.sh`

Will automatically run the test.sh script until it finds the first that exit non-zero.

Useful link

Pro Git:

https://git-scm.com/book/en/v2

Git reference docs:

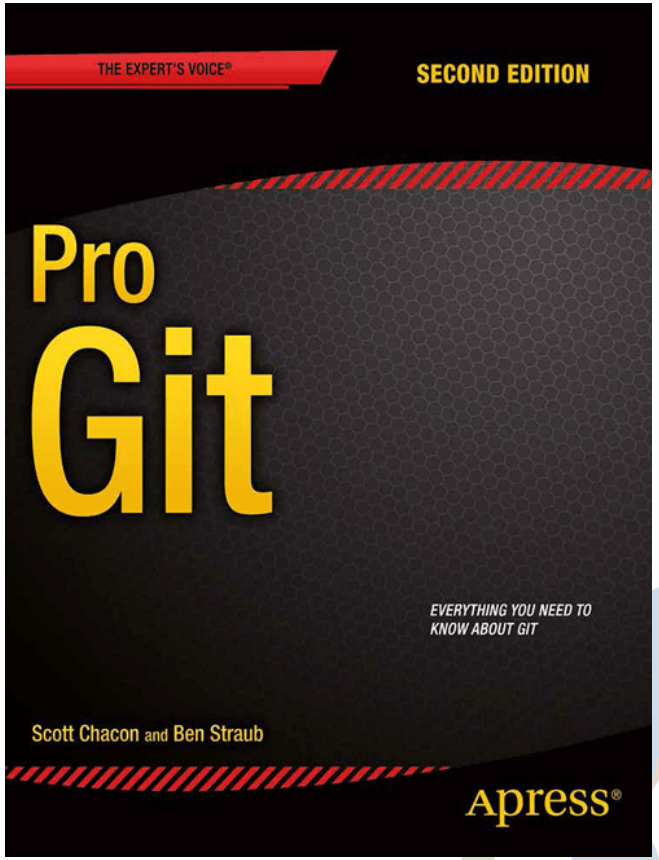
https://git-scm.com/docs

HSF Git guide:

https://hsf-training.github.io/analysis-essentials/git

Git cheat-sheet

https://ndpsoftware.com/git-cheatsheet.html



Exercise session

Tomorrow the 18th , from 15:15 to 16:15

You will need:

- Your own laptop, with `git` and an internet connection
- You will download (if you don't have it yet) `git-filter-repo`
- Optionally `c++ compiler`, `cmake`

  (if you want to compile and execute the sample code in the repository)

https://github.com/rsreds/git-good

Thank you!

More on restore

`git restore --worktree`

- Default behaviour. Changes go into your working copy

`git restore --staged`

- Changes go into your index

- you can pass both to combine the behaviour

`git restore --source <tree>`

If not specified, the contents are restored from the staging area otherwise, they're restored from the specific tree.

https://github.blog/2019-08-16-highlights-from-git-2-23/

More on reset: `reset filename`

Passing a path to reset will not move the HEAD but replace content.

`git reset --mixed filename`

replace the index copy of filename with the copy from the HEAD

Effectively unstaging the file: the inverse of `git add filename`

As for `add`, it accepts the `--patch` option

More on reset: ORIG_HEAD

Git keep the previous value of HEAD in variable called ORIG_HEAD.

To go back to the way things were:

```
git reset ORIG_HEAD
```

Allows you to avoid using the reflog to undo a rebase or a merge

```
git reset --hard ORIG_HEAD
```

Reset history to status before merge

```
$ git reflog

5d18706 (HEAD -> feature) HEAD@{0}:
rebase (finish): returning to
refs/heads/feature

5d18706 (HEAD -> feature) HEAD@{1}:
rebase (start): checkout HEAD~3

9c0984f HEAD@{2}: checkout: moving from
main to feature

…

$ git show ORIG_HEAD

commit 9c0984f…
```

More on reset: `squashing`

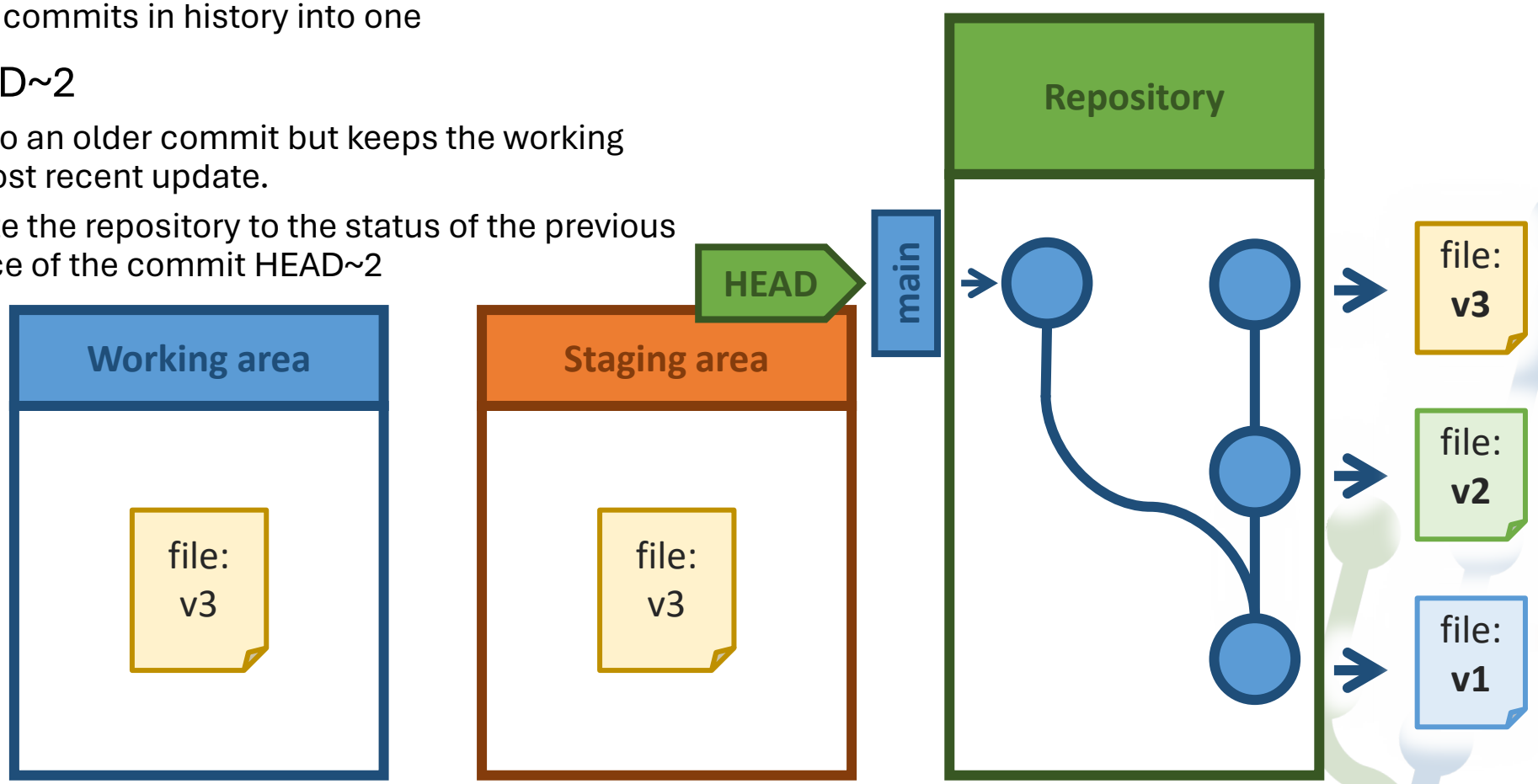
- Squashing commits

  Combine multiple commits in history into one

git reset --soft HEAD~2

Move HEAD back to an older commit but keeps the working directory to the most recent update.

Commit will update the repository to the status of the previous HEAD, with no trace of the commit HEAD~2



Dangling commit and GC

Git automatically runs a command called `auto gc`

- Packs loose objects
- Removes object not reachable from any commit
- Removes **dangling** commit: not referenced by anything

It does this when you have more than 6700 loose object!

It prunes dangling commit older than 90 days!

The garbage collector most often does nothing!