# Git

## Concepts and Workflows

(for Googlers: go/git-explained)

Edwin Kempin
Google Munich
ekempin@google.com

This presentation is based on a Git/Gerrit workshop that was developed by SAP.
Credits go to sasa.zivkov@sap.com, matthias.sohn@sap.com and christian.halstrick@sap.com

***PUBLIC - SHARED WITH GERRIT COMMUNITY***

# Target Audience

**This presentation is for:**

- Git beginners
- Advanced Git users that want to consolidate their Git knowledge
- Git users that start working with Gerrit Code Review

**Required pre-knowledge:**

- Basic knowledge about software development and versioning systems.

# Content

YES

- Git terminology
- Git concepts
- Git commands that are needed for daily work.
- Basic Git workflows
- Explanations of Git version graphs

```
git add, git alias, git bisect, git blame,
git branch, git cherry-pick, git checkout,
git clone, git commit, git commit --amend,
git diff, git fetch, git init, git log,
git merge, git notes, git pull, git push,
git push --force, git rebase,
git rebase --interactive, git reflog,
git reset, git revert, git rm, git show,
git stash, git status, git submodule,
git tag
```

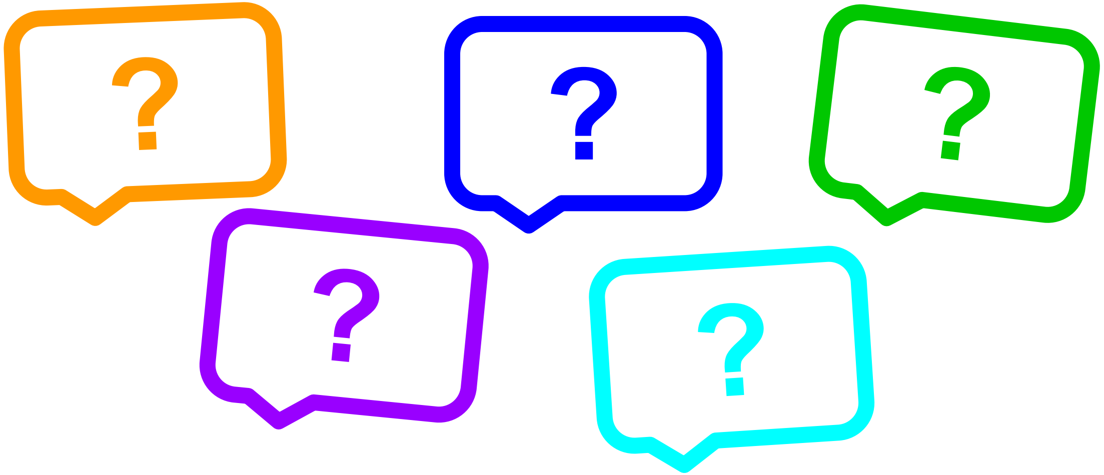
NO

- Gerrit Code Review (covered by Gerrit - Concepts and Workflows presentation)
- GitHub Pull Requests
- Git internals (like git protocol)
- Google specifics

# Agenda

- Git Repository Structure
- Making changes
- Branches
- Clone + Fetch
- Merge, Rebase, Cherry-Pick
- Push
- Interactive Rebase
- More Git

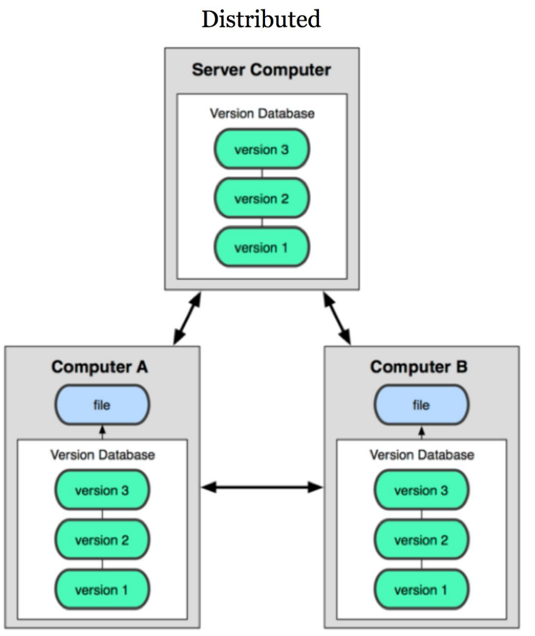
# Welcome

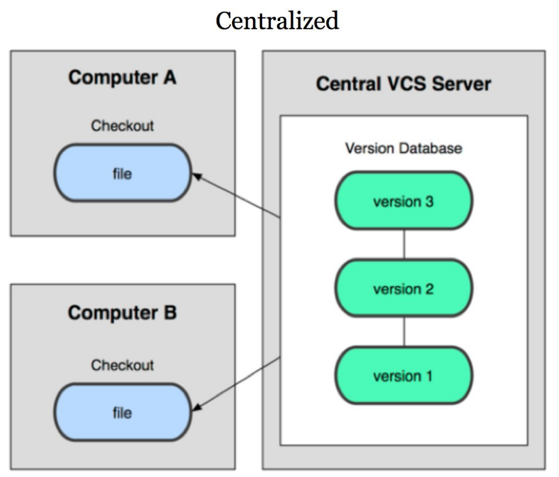


- Please ask questions immediately!
- To make the presentation more interactive you will also be asked questions :-)
- If you read through the slides and the answer to a question doesn’t get clear from the next slide, you can likely find it in the speaker notes.

*Q: What’s a distributed versioning system?*

# Distributed Versioning Systems





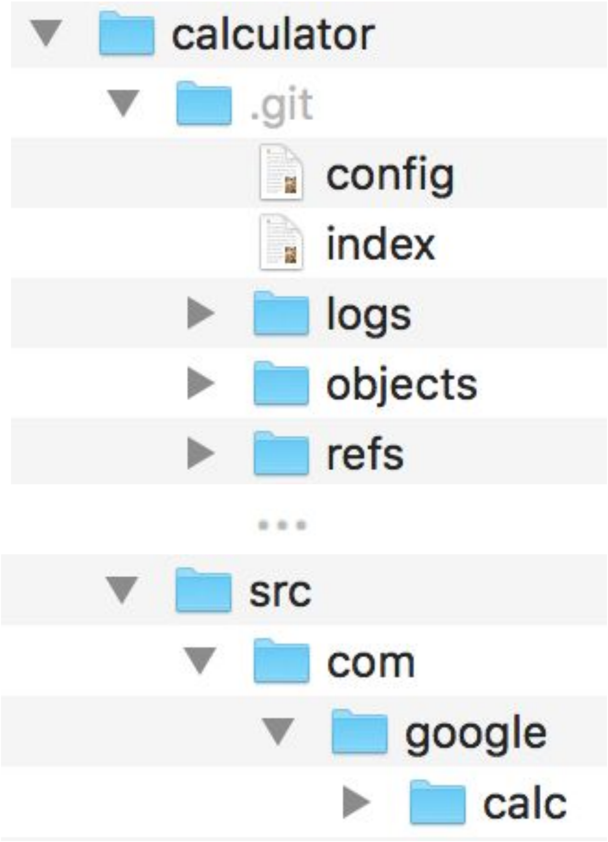
Distributed means:

- each developer has a ***complete, local repository***
- technically the central repository is not different from the local repositories
- easy offline usage
- easy to branch a project
- Examples: Git, Mercurial, Bazaar

# Git

- Created 2005 for Linux Kernel Development
- Used for Linux, Android, Eclipse
- Integration into Eclipse, Netbeans, XCode
- GitHub (popular Git hosting)
- Used at Google, SAP, Qualcomm, Ericsson, Sony, Wikimedia, Intel, NVIDIA, Twitter, Garmin, etc.

# Git Repository Structure

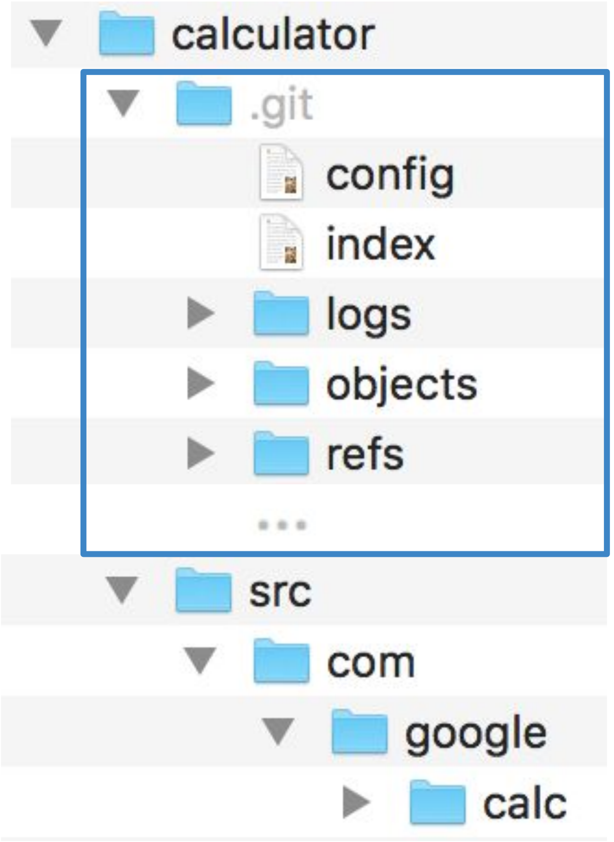


```
calculator
    .git
        config
        index
        logs
        objects
        refs
        ...
    src
        com
            google
                calc
```

A ***Git repository*** is created by:

- `git init`
- `git clone` (explained later)

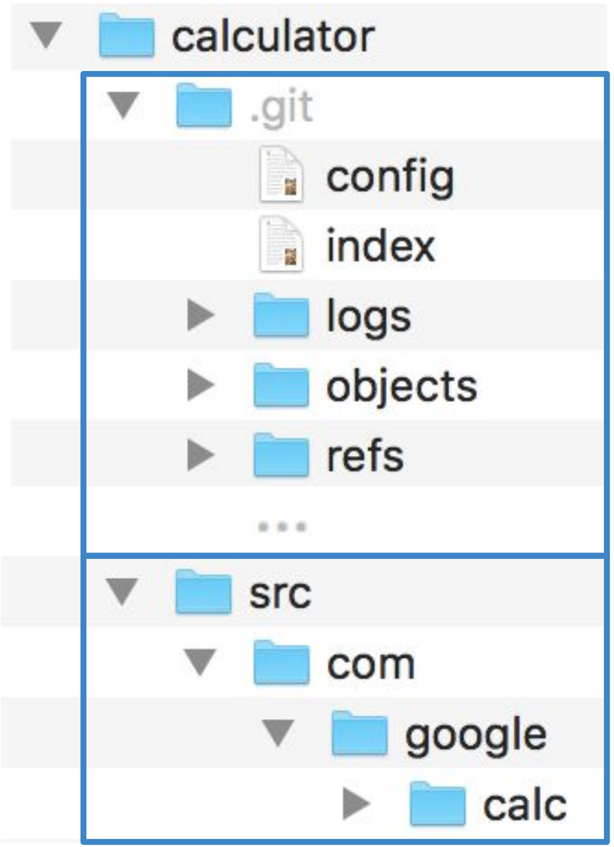
# Git Repository Structure



*.git* folder is the ***Git repository***

- The *.git* folder contains the full version database.
- Many files in the *.git* folder are human-readable.

# Git Repository Structure



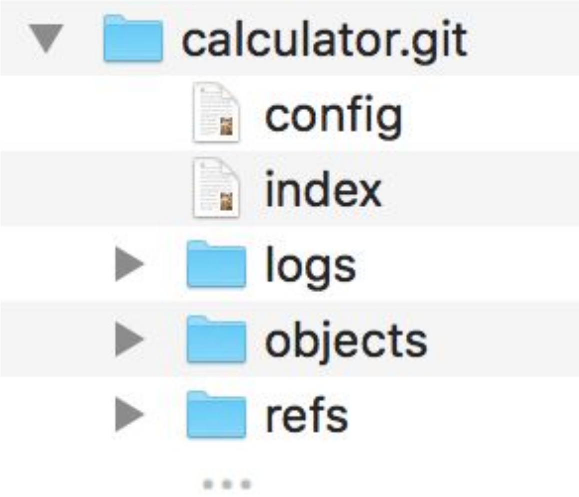
calculator
.git
config
index
logs
objects
refs
...
src
com
google
calc

*.git* folder is the ***Git repository***

files/folders next to the *.git* folder are the ***working tree***

A ***Git repository*** has at most one ***working tree***.

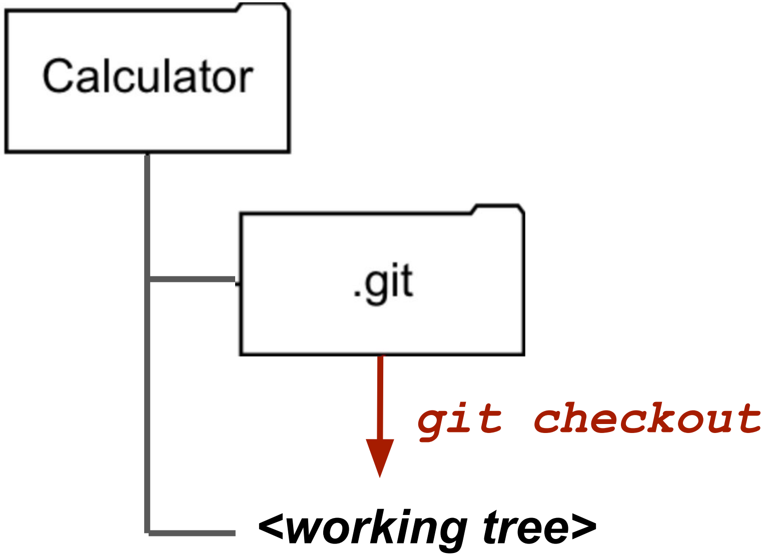
# Git Repository Structure



```
calculator.git
    config
    index
    logs
    objects
    refs
    ...
```

A ***Git repository*** without ***working tree*** is called ***bare repository*** (used on servers).

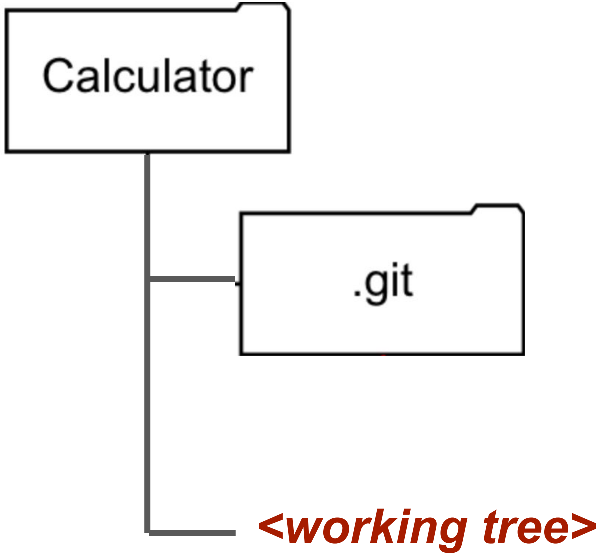
# Checkout



***Checkout***:

- populates the ***working tree*** with the ***commit*** you want to start working from

# Making Changes



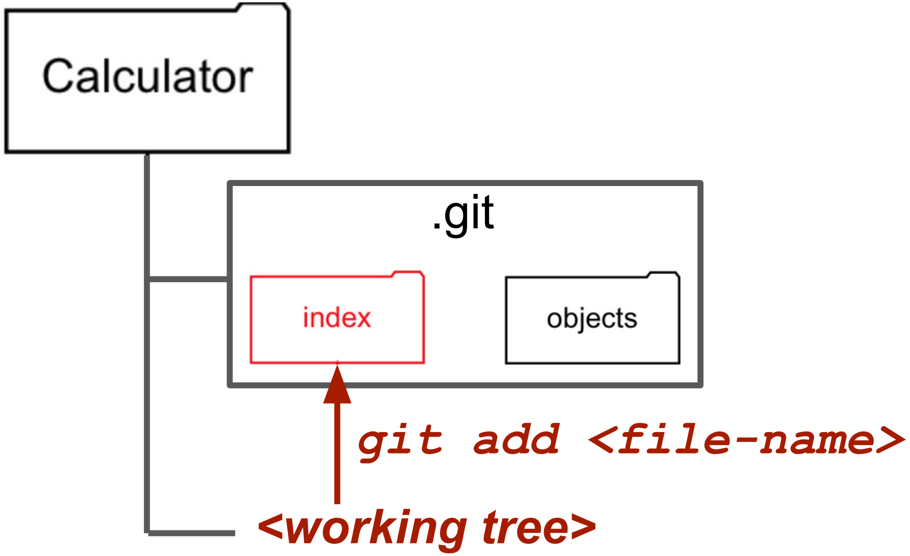
Just start doing your changes:

- modify, add, delete files
- no need to tell Git which files you want to work on
- tell Git which changes you intend to commit:

```
git add <file>
git rm <file>
```

*Q: What happens on git add?*

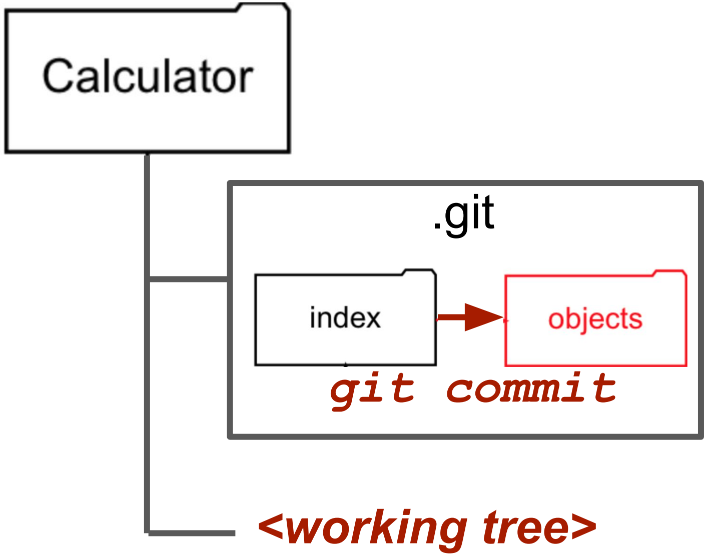
# Making Changes



***Index*** or ***Staging Area*** is where the next ***commit*** is prepared:

- `git add` and `git rm` update the ***index***
- Stage single hunks: `git add -p <file>`
- Unstage files: `git reset HEAD <file>`

# Making Changes

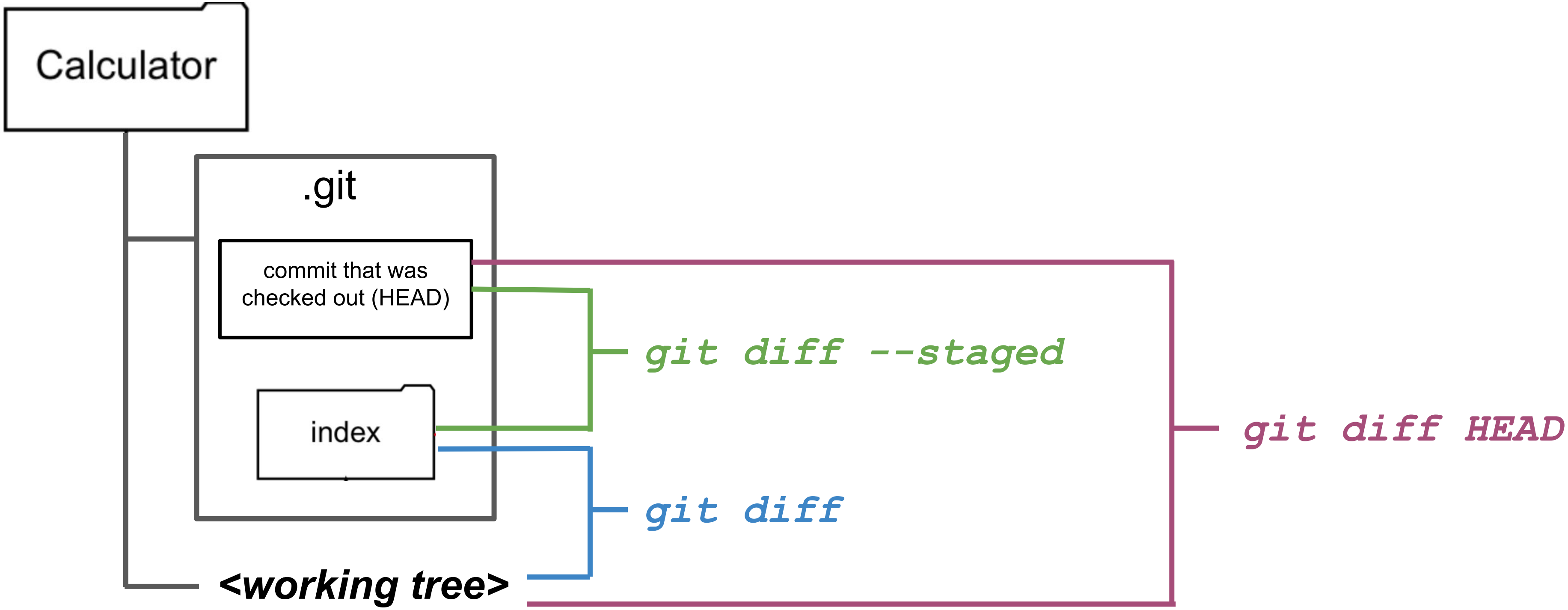


- `git commit` commits ***staged*** changes only (the ***index***)
- There can still be non-staged changes in the ***working tree*** which will not be included into the ***commit***.

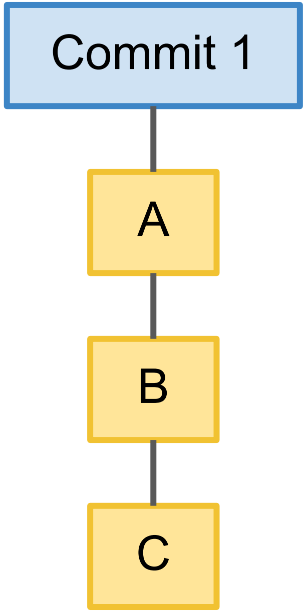
# See what was changed

- 3 states:
  - commit that was checked out (***`HEAD`***)
  - ***index*** (staged modifications)
  - ***working tree*** (unstaged modifications)
- `git status` shows changed paths
  - between ***index*** and commit that was checked out (***`HEAD`***)
  - between ***working tree*** and ***index***
- `git diff` shows file modifications
  - details on next slide

# See what was changed

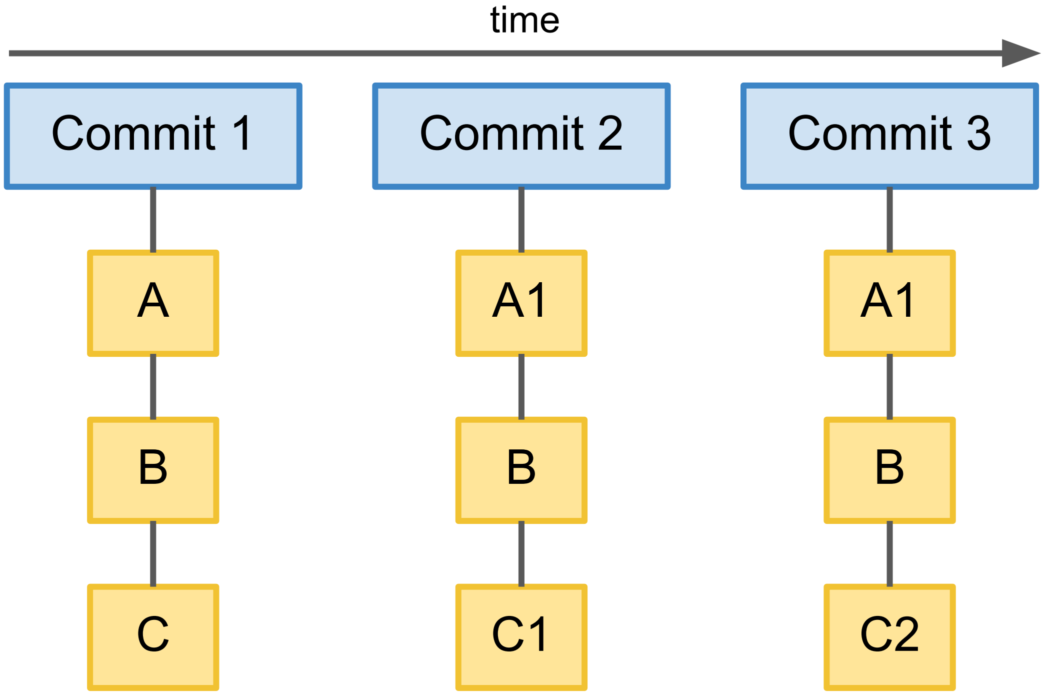


# Commits



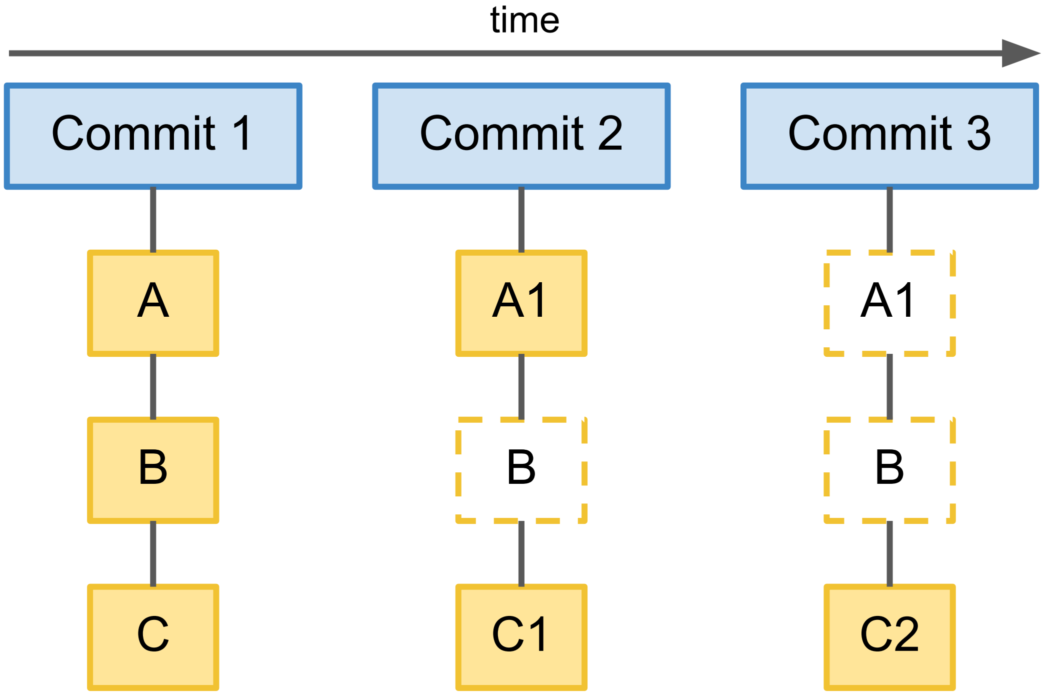
- Imagine a project that contains 3 files: ***A***, ***B*** and ***C***

# Commits



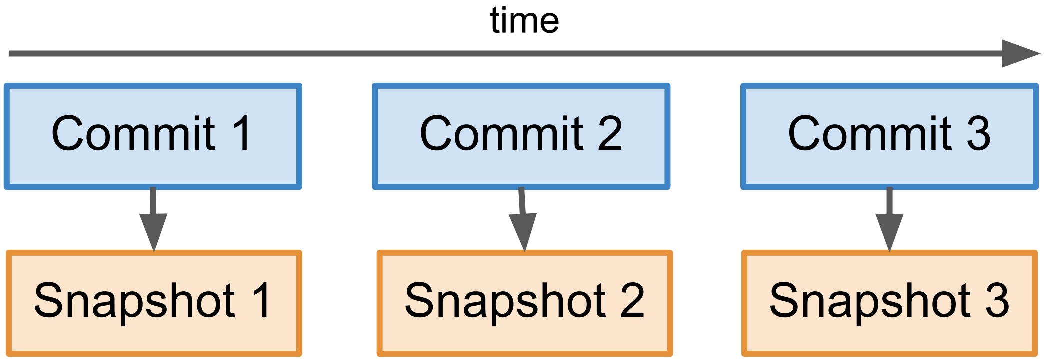
- Each time changes are committed a new commit is created: ***Commit 1***, ***Commit 2***, ***Commit 3***
- Every commit is a ***full snapshot*** of the whole project.

# Commits



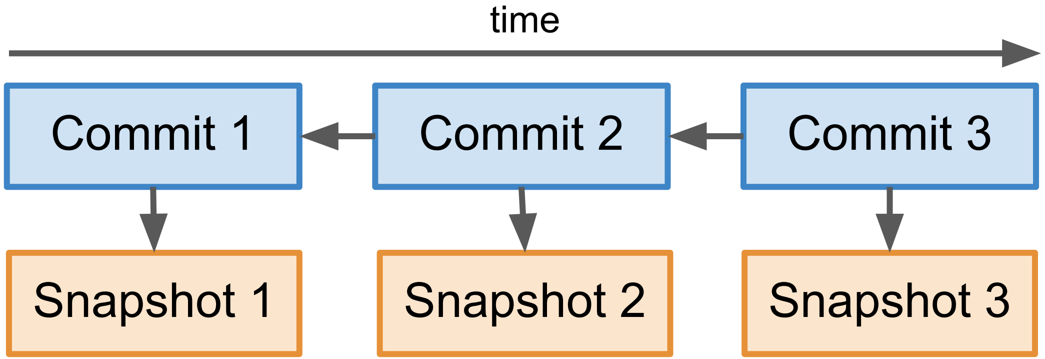
- Git optimizes the storage and will not create copies of non-modified files.

# Commits



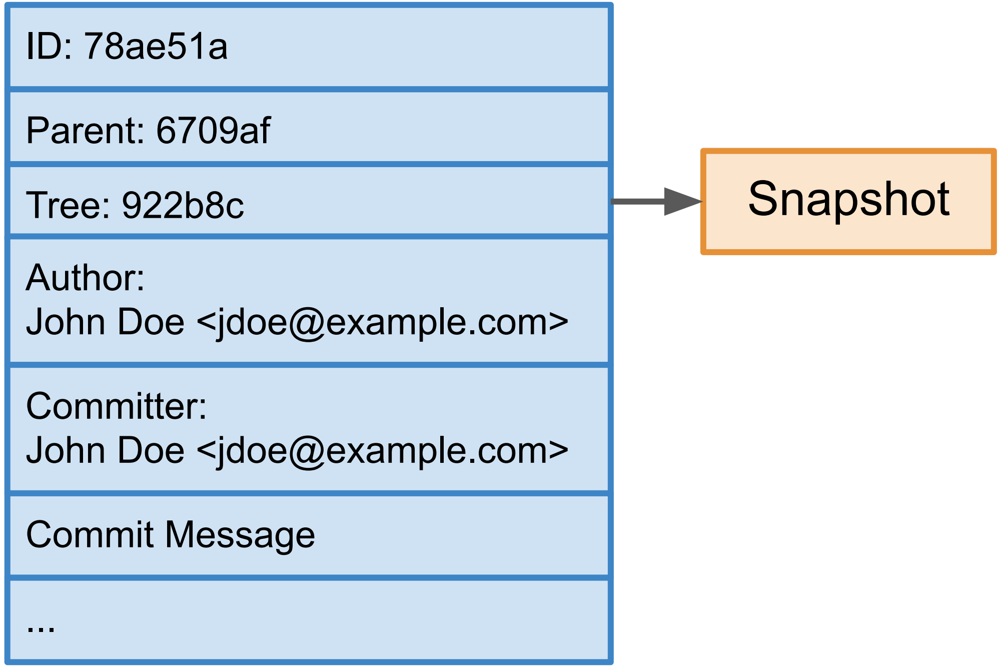
- Same as previous slide, only the files are collapsed into snapshots now.

# Commits



- Each commit knows its parent.

# Commit Object Structure



ID: 78ae51a

Parent: 6709af

Tree: 922b8c

Author:
John Doe <jdoe@example.com>

Committer:
John Doe <jdoe@example.com>

Commit Message

...

***SHA1:***

- globally unique commit ID
- 40-digit hexadecimal number
- function of the commit object content
- shown in *`git log`* output etc.

To inspect a commit use:

- *`git show <SHA1>`*
- *`git show --format=fuller <SHA1>`*

Once created commits are immutable.

*Q: What's the difference between author and committer? When do they differ?*

# Author vs. Committer

- ***Author***:
  - person who wrote the patch
- ***Committer***:
  - person who created the commit,
    e.g. project maintainer who applied the patch

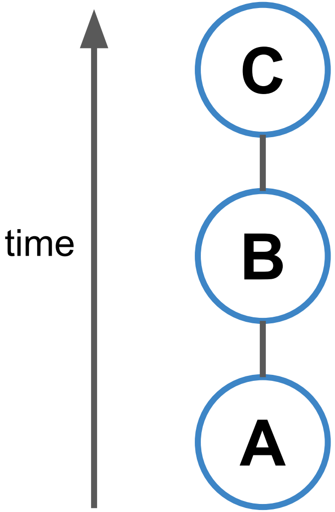
- Git sets ***author*** and ***committer*** based on the `user.name` and `user.email` config values.
- Author can be explicitly set on commit:
  `git commit --author=<author>`

# Commit Message

```
First line is the subject, should be shorter than 70 chars

Separate the body from the subject by an empty line. The
commit message should describe why you are doing the
change. That's what typically helps best to understand what
the change is about. The details of what you changed are
visible from the file diffs.

The body can have as many paragraphs as you want. Lines
shouldn't exceed 80 chars. This helps command line tools to
render it nicely. Paragraphs are separated by empty lines.

Bug: Issue 123
```

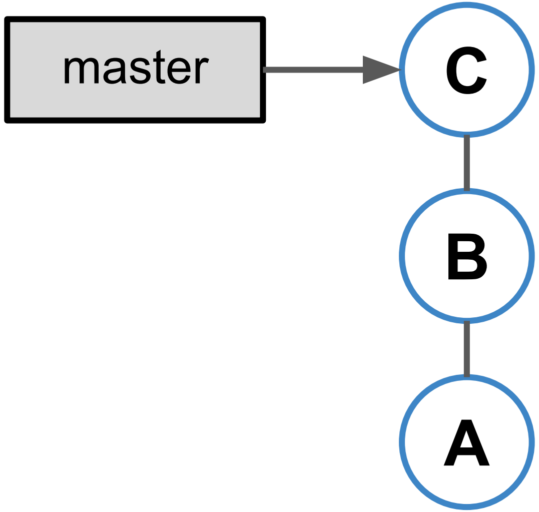
- First line is the subject.
- Separated by a blank line follows the body.
- The last paragraph is for metadata (key-value pairs). The metadata is intended to be interpreted by tools.

# Commit History



- ***C*** is a successor of ***B***
- ***B*** is a successor of ***A***
- The lines between the commits represent parent relationships, the arrows for parent relations are omitted.
- Can be seen by:
  - `git log` (with file diffs)
  - `git log --oneline` (with subject only)
  - `git log --graph` (as graph)
  - `gitk` (as graph in Git repository browser)

# Branches

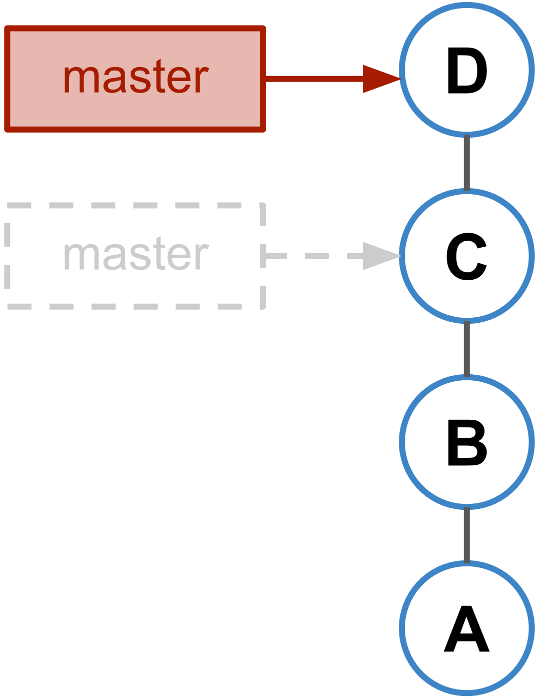


A ***branch*** is a named pointer to a ***commit***:

- example: *`master`*
- full name: *`refs/heads/master`*
- All commits that are reachable from a branch following the parent pointers form the ***branch history***.
- The commit to which the branch points is also called ***branch tip***.

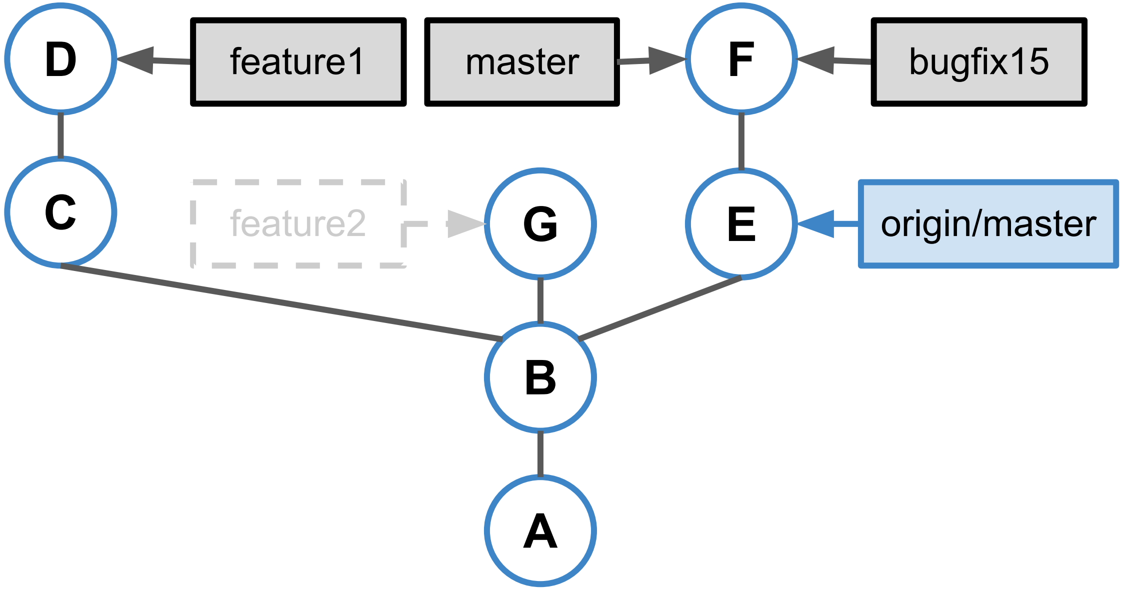
*Q: What happens when new changes are committed?*

# Branches



- A new commit ***D*** gets created.
- The branch is moved to point to the new commit.

# Branches



Usually there are many branches in a Git repository:

- Branches can point to the same commit.
- Branches can be deleted: *`git branch -D <branchname>`*
- There is nothing special about *`master`*, it's just a normal local branch
- *`origin/master`* is a remote tracking branch (explained later).

Branch creation is done by:

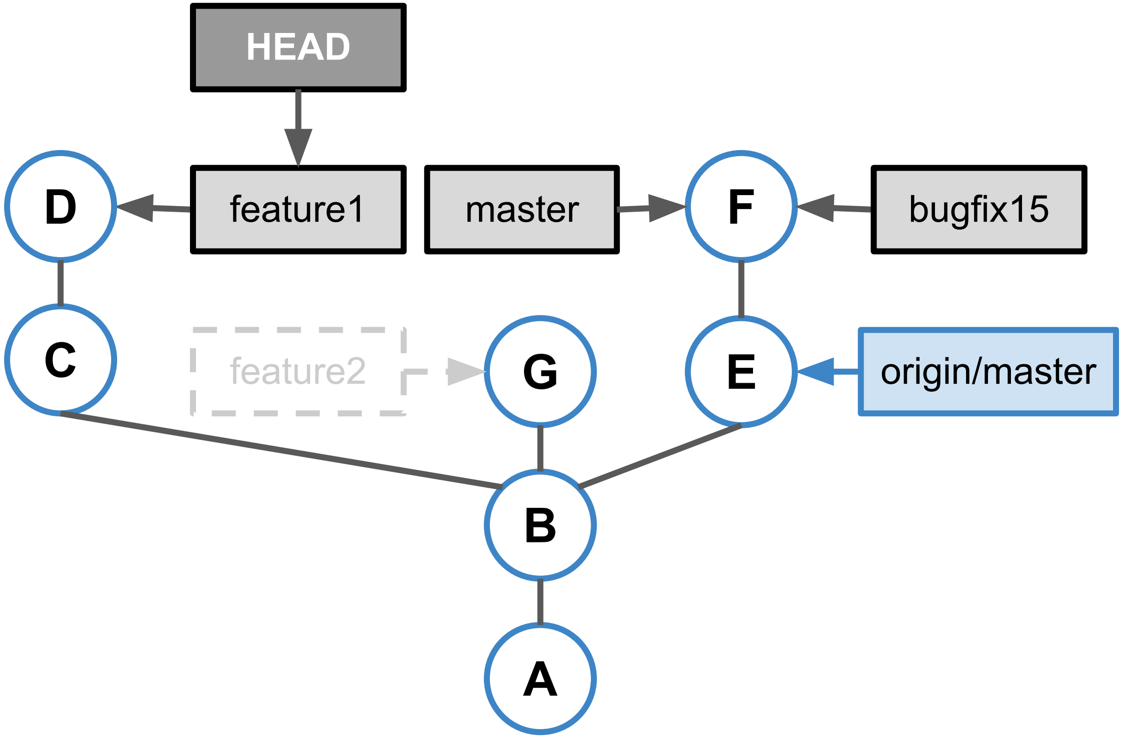
- *`git branch <branchname>`*
- *`git checkout -b <branchname>`*

List all branches:

- *`git branch -a`*

*Q: How does Git know which branch should be updated on the next commit operation?*

# HEAD

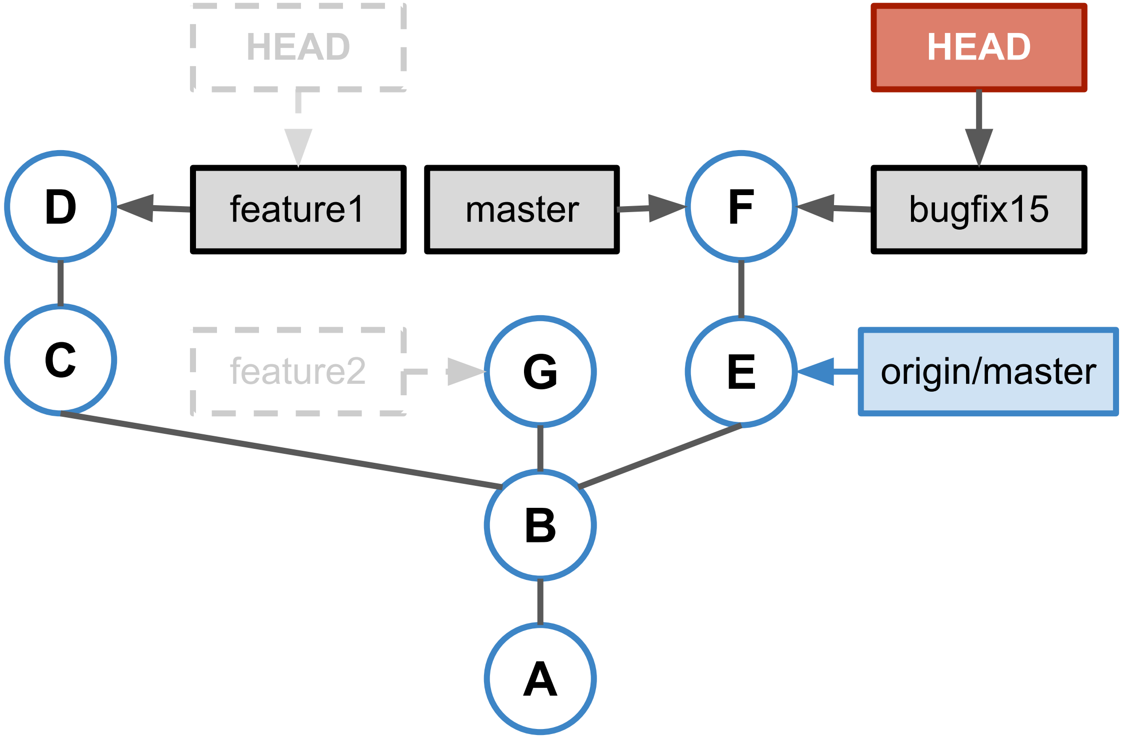


***HEAD*** points to the ***current branch***:

- `git commit` updates the current branch
- `git checkout` sets the current branch

*Q: What happens on* `git checkout bugfix15` *?*

# HEAD



`git checkout`:

- moves ***HEAD***
- also updates the ***working tree***

*Q: What if you started to make changes but you forgot to checkout the correct branch?*

# Checkout with changes in working tree

If you started to make changes to the working tree but the wrong branch is checked-out:

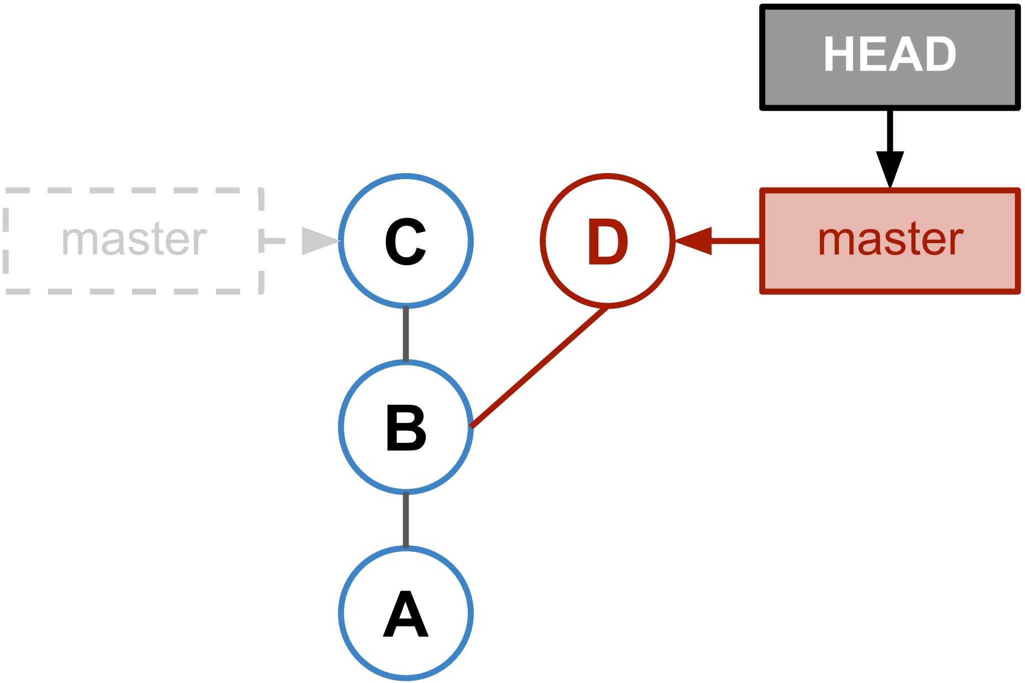
- just try to checkout the correct branch, if there are no conflicts this will just work
- if there are conflicts the checkout fails, in this case you can do:

  ```
  $ git stash
  $ git checkout <correct-branch>
  $ git stash pop
  $ <resolve conflicts>
  $ git stash drop
  ```

- a ***working tree*** with modifications is called ***dirty***
- a ***working tree*** without modifications is called ***clean***
- `git stash` puts changes in a ***dirty working tree*** aside, with `git stash pop` they can be applied somewhere else (more about conflict resolution later)

*Q: What if you have created a commit but you want to include additional changes?*

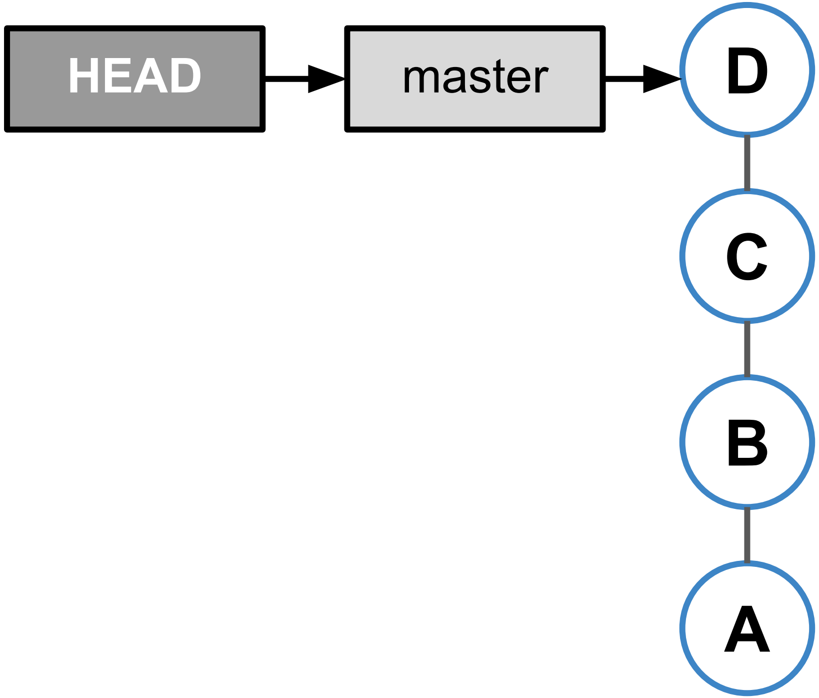
# Amend Commit



`git commit --amend` rewrites the last commit:

- creates a sibling commit ***D*** of the last commit ***C*** and ***resets*** the current branch to it
- the old commit message is preserved, but can be edited if wanted
- the old commit ***C*** is still available in the repository
- rewrites the history of the branch (you should never rewrite commits that have already been shared with others, since others may already have used it as base for other work)

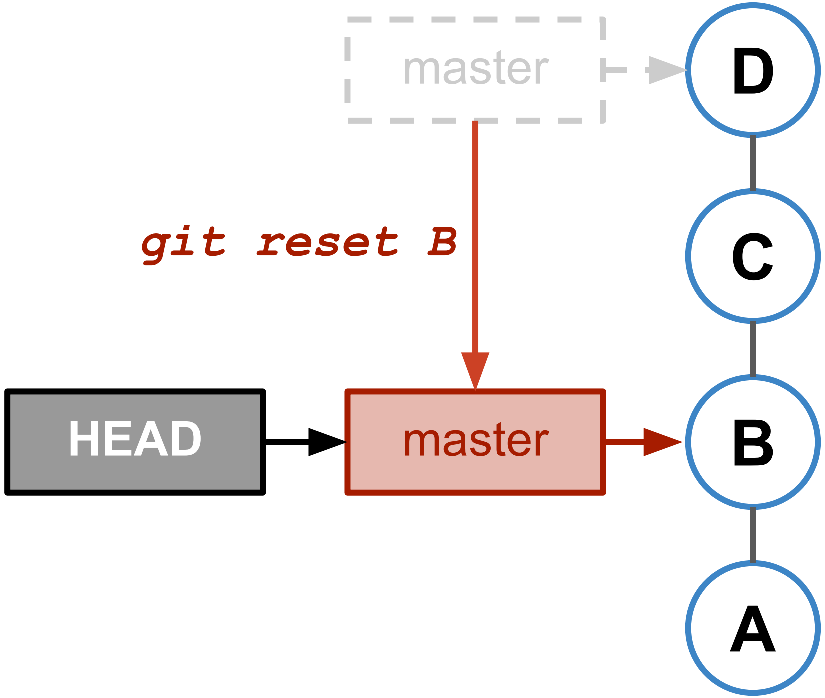
# Resetting Branches



- Branches can also be moved “manually” by `git reset`.

*Q: What happens when master is reset to commit B?*

# Resetting Branches

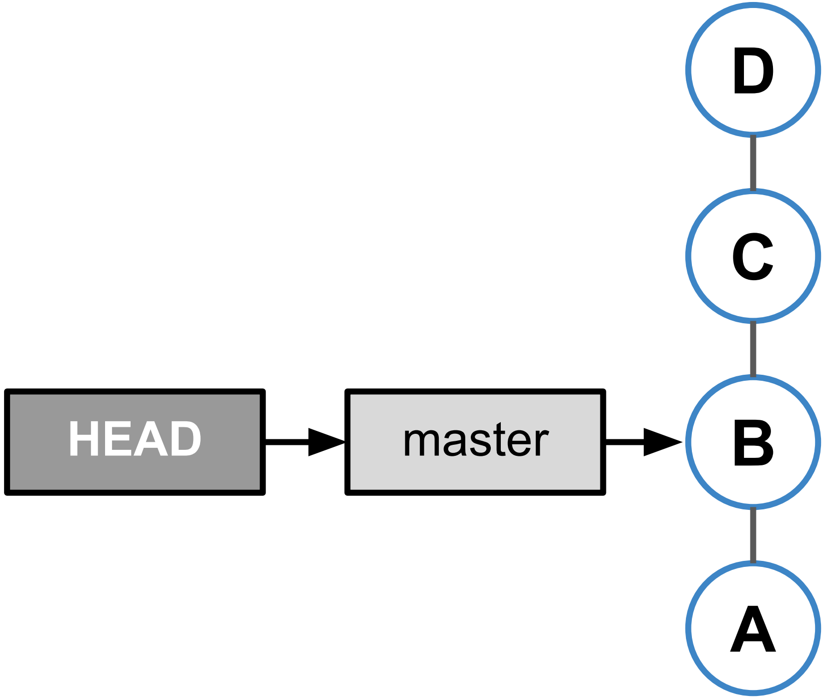


```
git reset B
```

- Updates the ***current branch*** to point to commit ***B***.
- Commit ***C*** and ***D*** are no longer part of the history of the master branch.

*Q1: What happens to the non-reachable commits C and D?*

# Non-reachable Commits

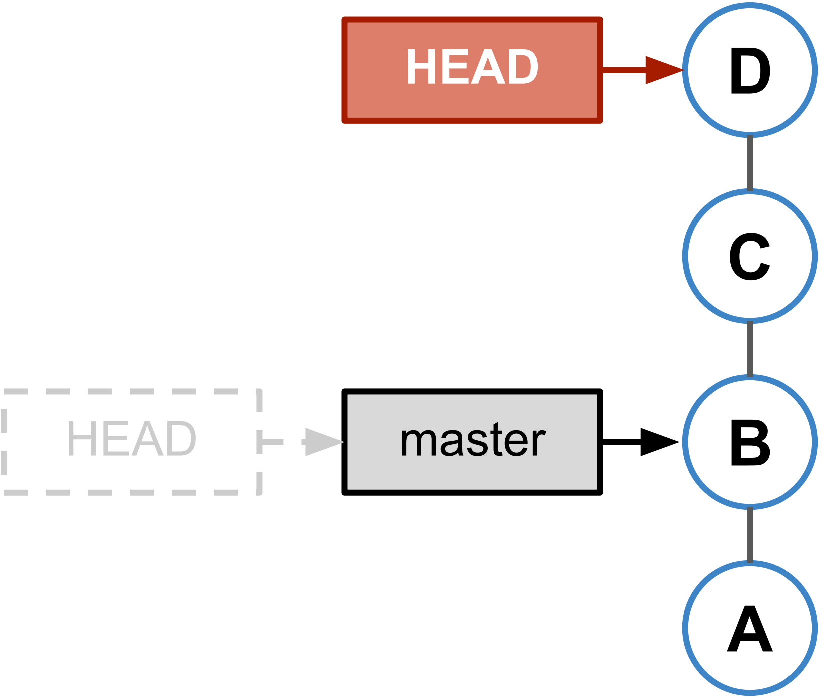


Non-reachable commits

- are by default kept for 2 weeks
- are garbage collected after the expiry time has passed and when `git gc` is run
- Can be checked out by SHA1.

*Q: What happens if a non-reachable commit is checked-out?*

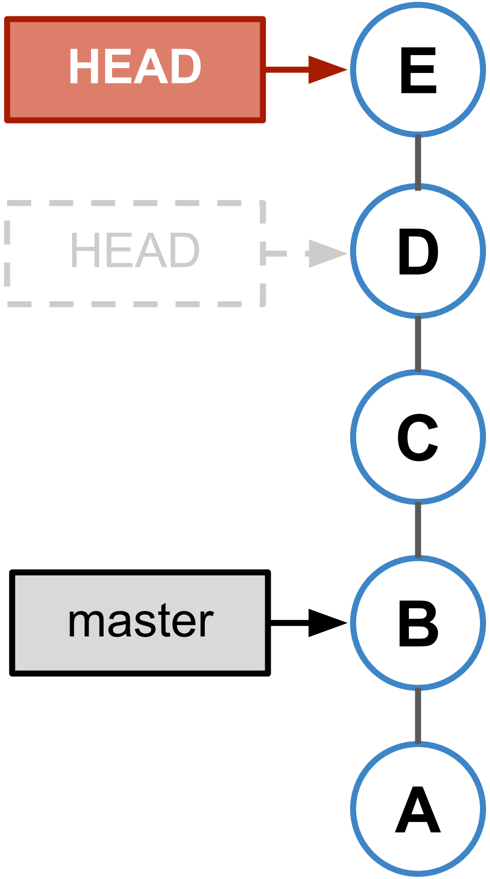
# Detached HEAD



If *HEAD* points directly to a commit (instead of pointing to a branch) it's called ***detached HEAD***.

*Q: What happens if a new commit is created now?*

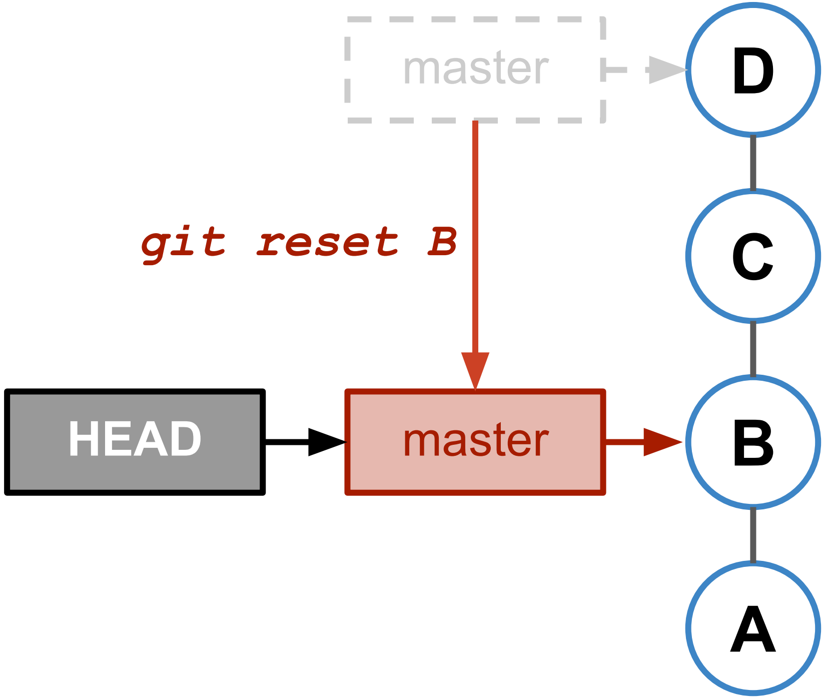
# Detached HEAD



New commits can be created even if *HEAD* is detached:

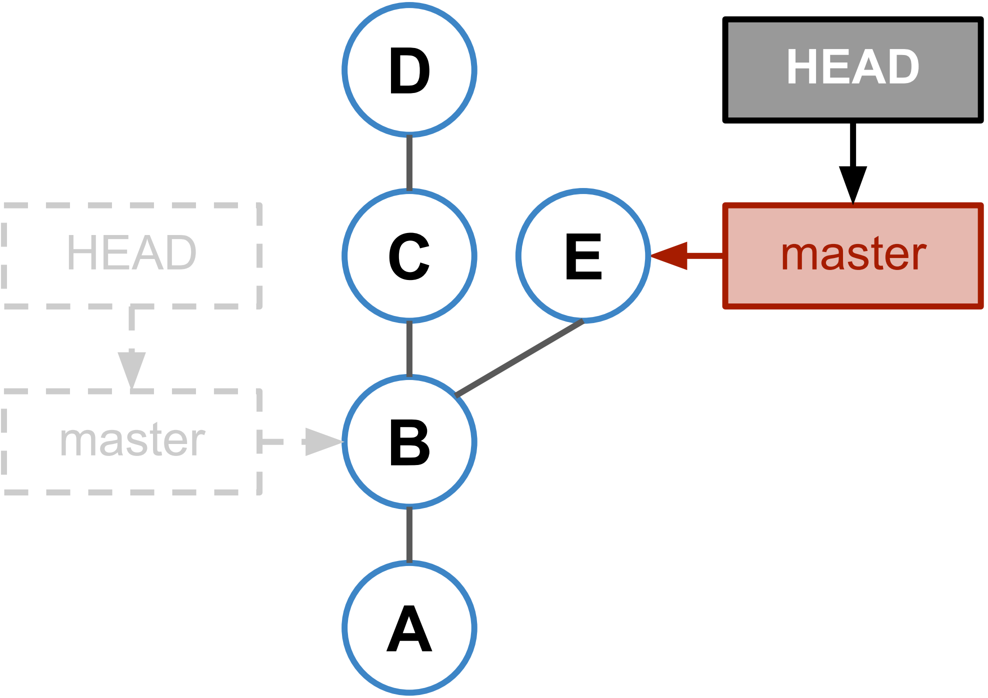
- If you checkout something else now the new commit gets unreachable (but you may still access it if you know its SHA1).

# Resetting Branches



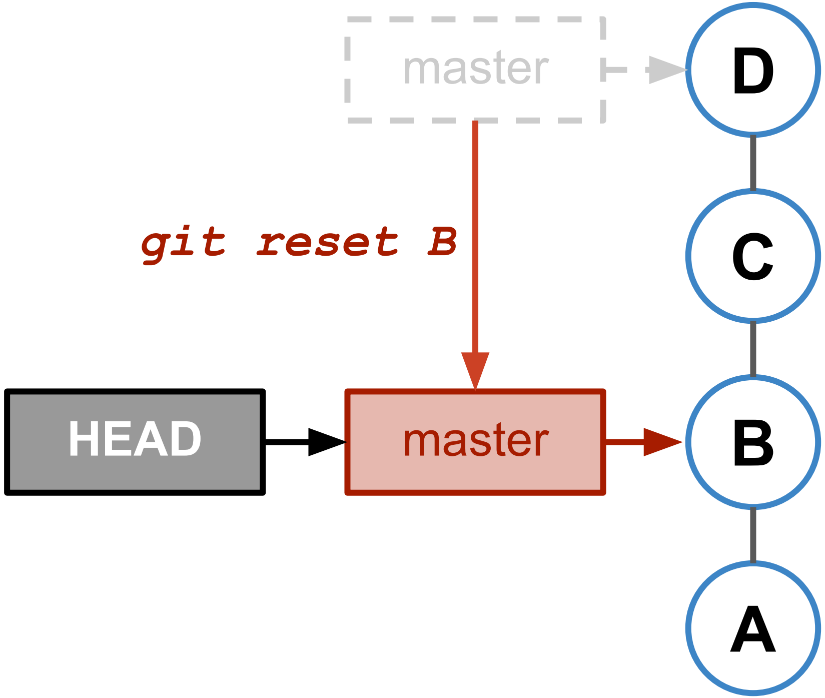
*Q2: Where is the new commit created on git commit after reset?*

# New Commit after Reset



- The new commit becomes successor of the commit to which the current branch points.
- The current branch is updated.

# Resetting Branches



*Q3: What happens to the working tree and the index on branch reset?*

# Resetting Branches

| `git reset` | branch | index | working tree |
|---|---|---|---|
| `--soft` | **Yes** | **No** | **No** |
| `--mixed` (default) | **Yes** | **Yes** | **No** |
| `--hard` | **Yes** | **Yes** | **Yes** |

The ***branch*** is always reset, whether the ***index*** and ***working tree*** are reset depends on the reset mode (`soft`, `mixed`, `hard`).

With `git reset --hard` local modifications in the working tree are lost.

*Q: What are use cases for the different reset modes?*

# Resetting Branches

| *git reset* | branch | index | working tree |
|---|---|---|---|
| *--soft* | **Yes** | **No** | **No** |
| *--mixed* (default) | **Yes** | **Yes** | **No** |
| *--hard* | **Yes** | **Yes** | **Yes** |

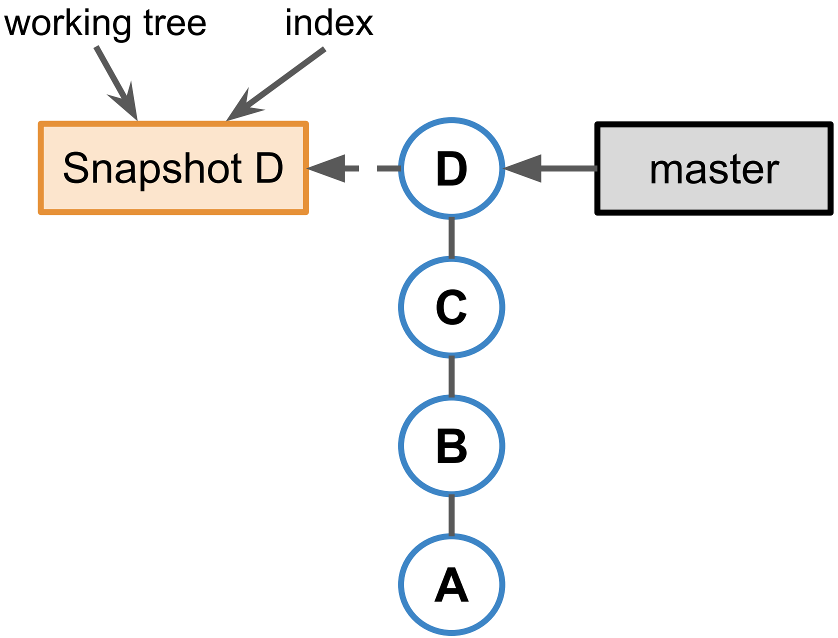
Use cases:

- *git reset --hard*: Discard all local modifications.
- *git reset --soft*: Squash commits.
- *git reset --mixed*: Split commits.

*Q: How is squashing and splitting commits by reset working?*

# Squash commits by soft reset

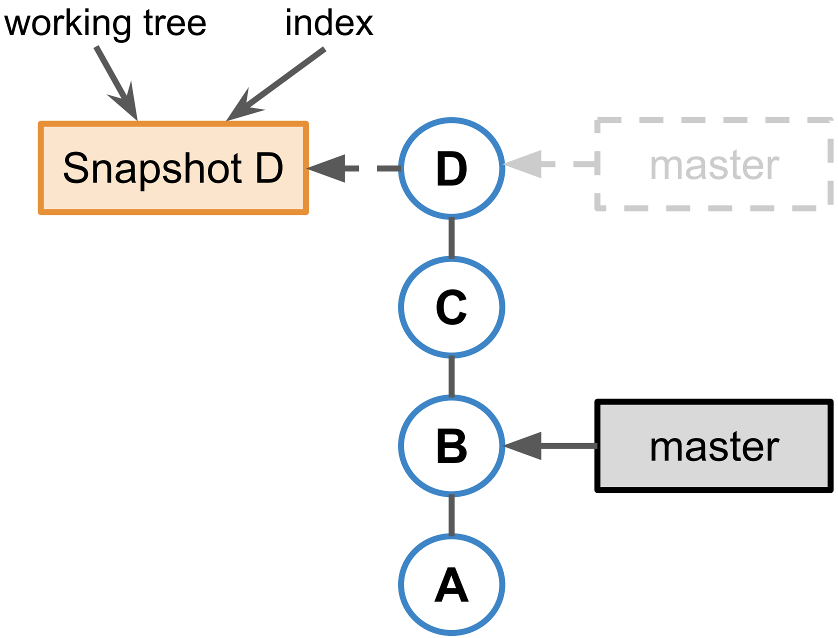
| `git reset` | branch | index | working tree |
|---|---|---|---|
| `--soft` | **Yes** | **No** | **No** |



1. *`git checkout D`*: current branch points to D, index and working tree contain snapshot of D

# Squash commits by soft reset

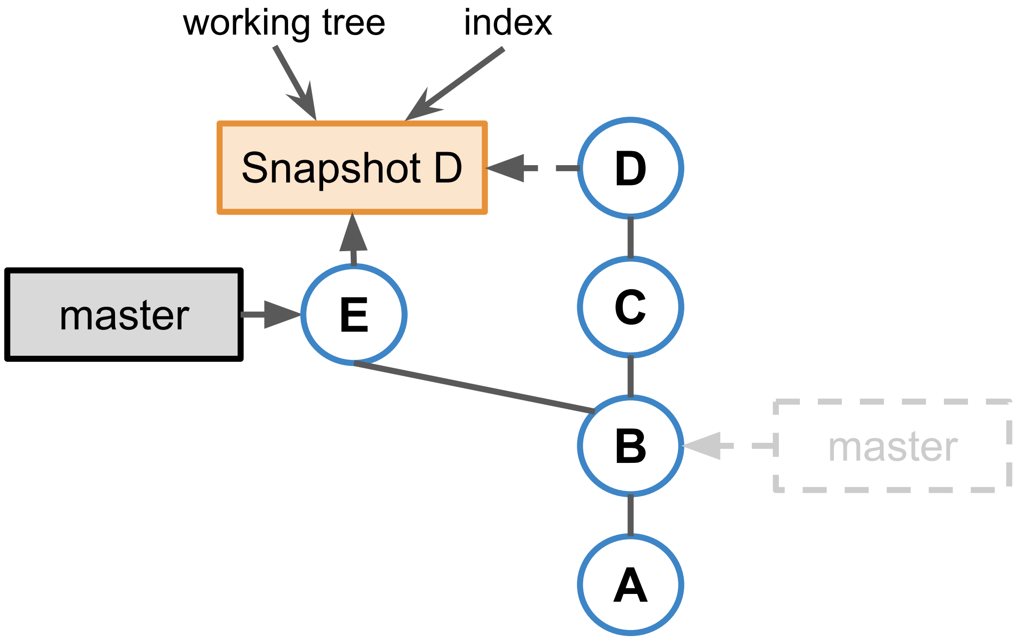
| *git reset* | branch | index | working tree |
|---|---|---|---|
| *--soft* | **Yes** | **No** | **No** |



1. *git checkout D*: current branch points to D, index and working tree contain snapshot of D
2. *git reset --soft B*: current branch points to B, index and working tree still contain snapshot of D

# Squash commits by soft reset

| *git reset* | branch | index | working tree |
|---|---|---|---|
| *--soft* | **Yes** | **No** | **No** |

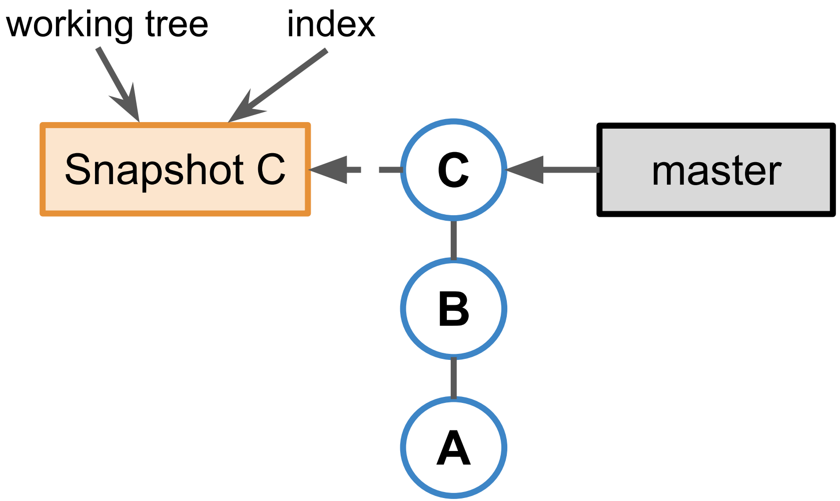


1. *git checkout D*: current branch points to D, index and working tree contain snapshot of D
2. *git reset --soft B*: current branch points to B, index and working tree still contain snapshot of D
3. *git commit*: new commit E is created from index state (snapshot of D)

Git interactive rebase is a better way to squash commits (explained later).

# Split commits by mixed reset

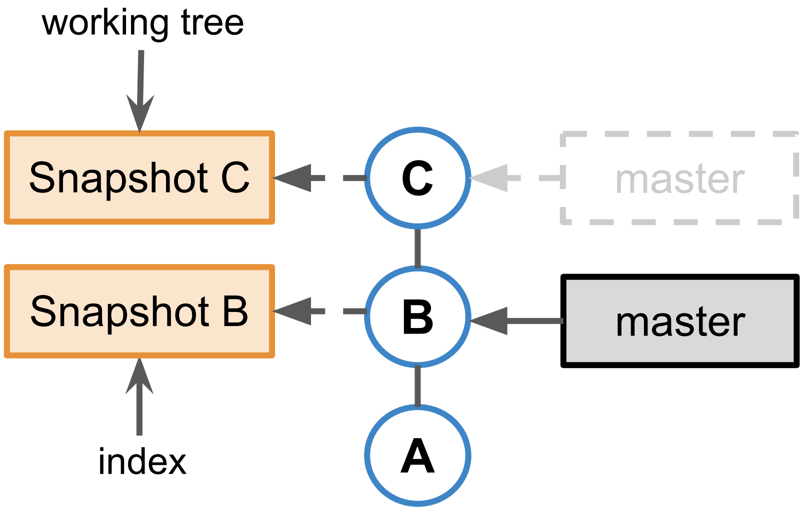
| `git reset` | branch | index | working tree |
|---|---|---|---|
| `--mixed` (default) | **Yes** | **Yes** | **No** |



1. `git checkout C`: current branch points to C, index and working tree contain snapshot of C

# Split commits by mixed reset

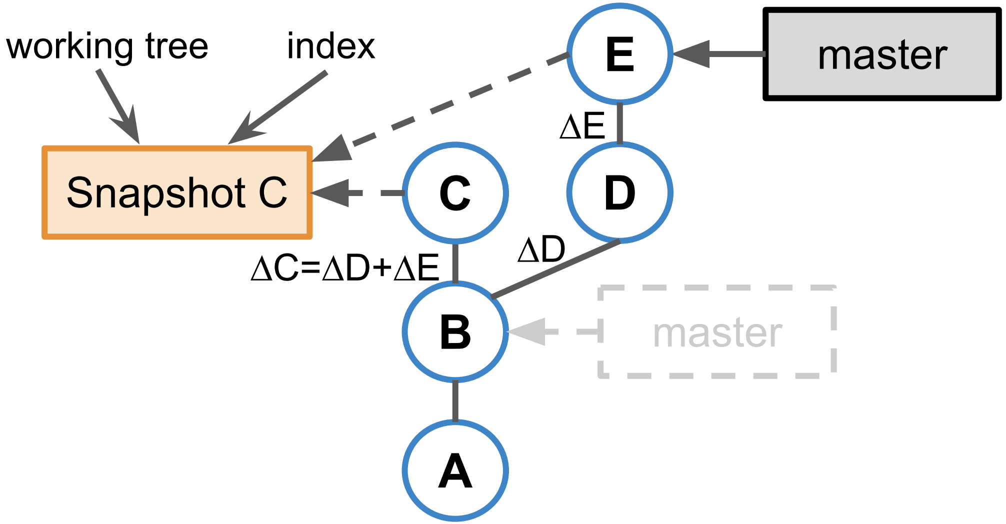
| `git reset` | branch | index | working tree |
|---|---|---|---|
| `--mixed` (default) | **Yes** | **Yes** | **No** |



1. `git checkout C`:
   current branch points to C, index and working tree contain snapshot of C
2. `git reset --mixed B`:
   current branch points to B, index contains snapshot of B, working tree still contains snapshot of C

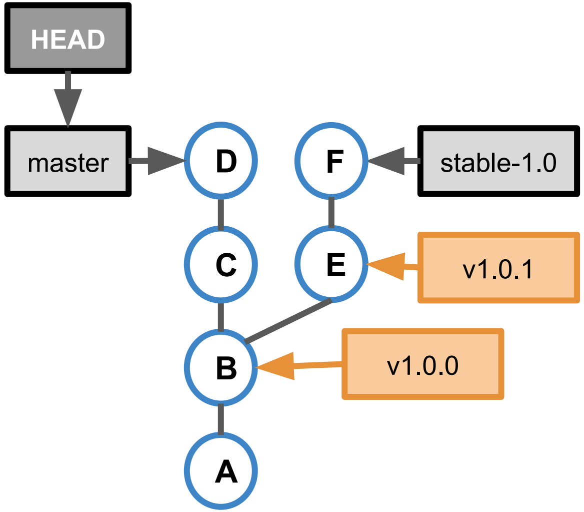
# Split commits by mixed reset

| `git reset` | branch | index | working tree |
|---|---|---|---|
| `--mixed` (default) | **Yes** | **Yes** | **No** |



1. `git checkout C`:
   current branch points to C, index and working tree contain snapshot of C
2. `git reset --mixed B`:
   current branch points to B, index contains snapshot of B, working tree still contains snapshot of C
3. `git add <file> && git commit`:
   Stage some modifications and commit.
4. `git add <file> && git commit`:
   Stage rest of modifications and commit.

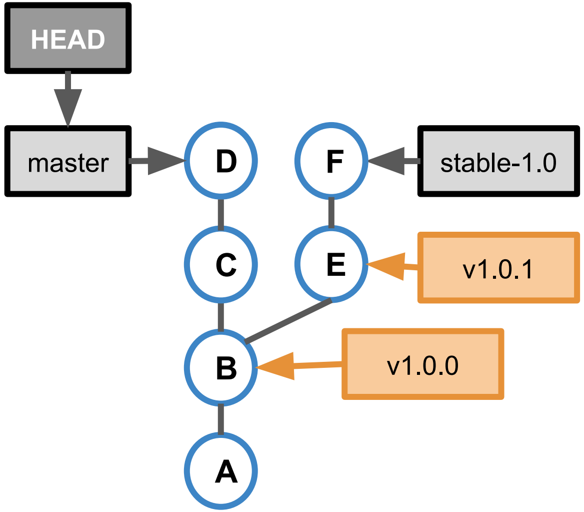
# Tags



A *tag* allows to tag a specific point in the version history:

- normally used to tag important versions such as releases
- in contrast to ***branches tags*** are immutable (well you can delete and recreate tags, but you really should not do this once they have been shared with others)
- example: `v1.0.0`
- full name: `refs/tags/v1.0.0`

# Tags



There are 3 kind of ***tags***:

- ***lightweight tags*** (just a pointer to a commit)
- ***annotated tags*** (full Git object, allows tags to have a message)
- ***signed tags*** (tag with signature)

Tag creation:

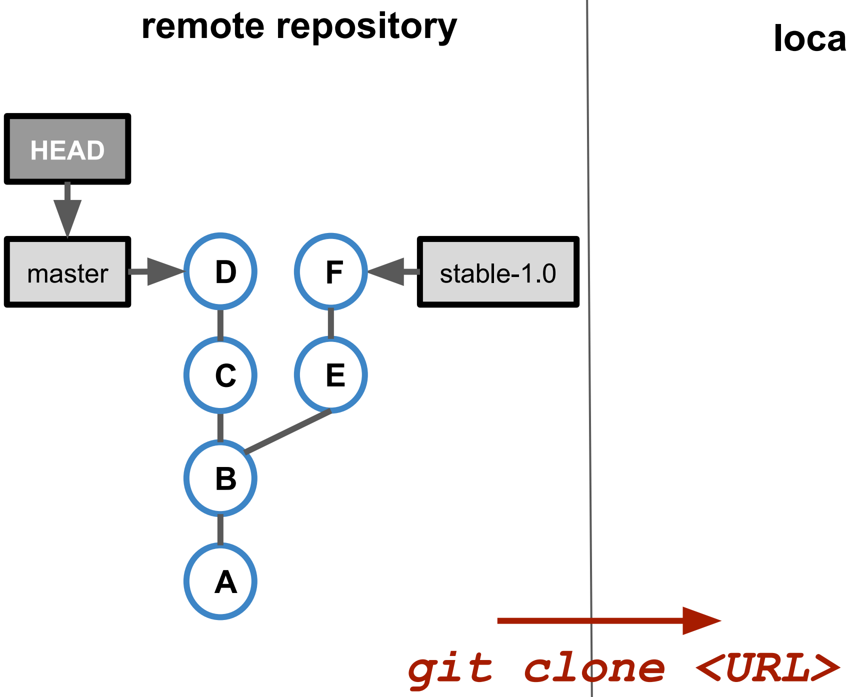
- `git tag <tagname>`
- `git tag -a <tagname>`
- `git tag -s <tagname>`

List tags:

- `git tag`

# B-R-E-A-K

# Clone

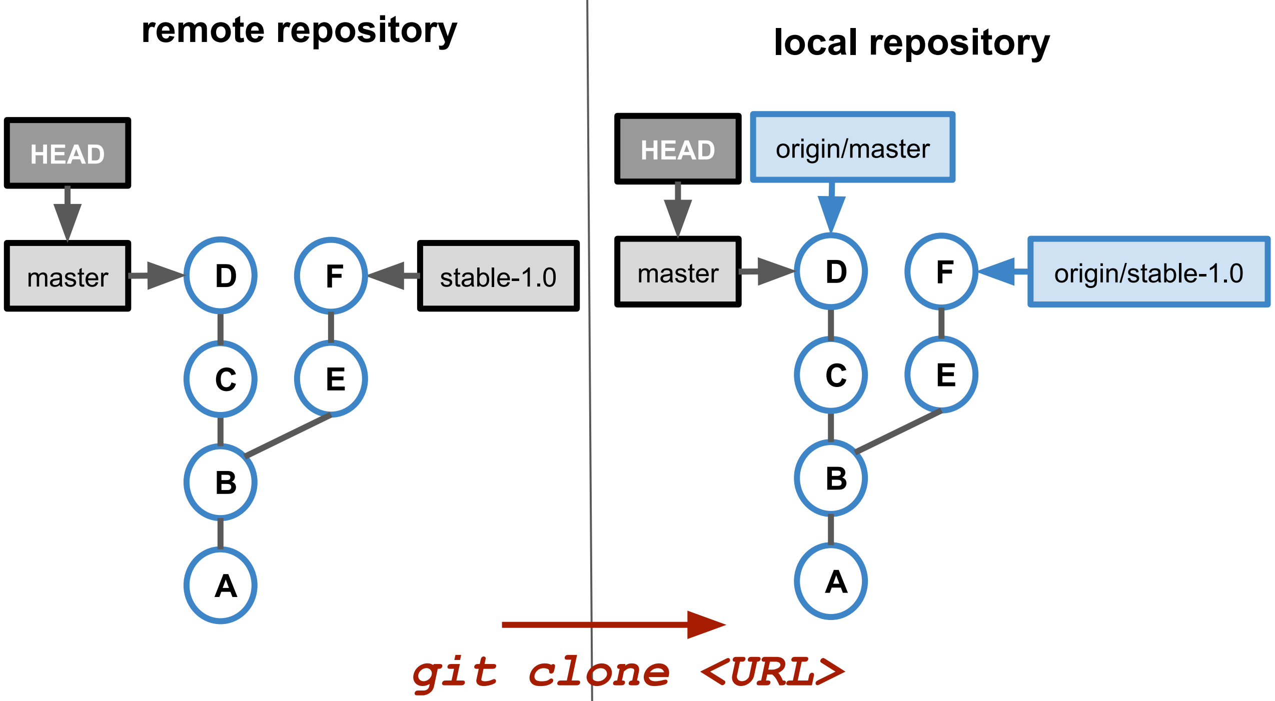


A remote repository can be cloned to a client by

`git clone <URL>`

*Q: What happens on git clone? Which commits does the client get? Which branches?*

# Clone

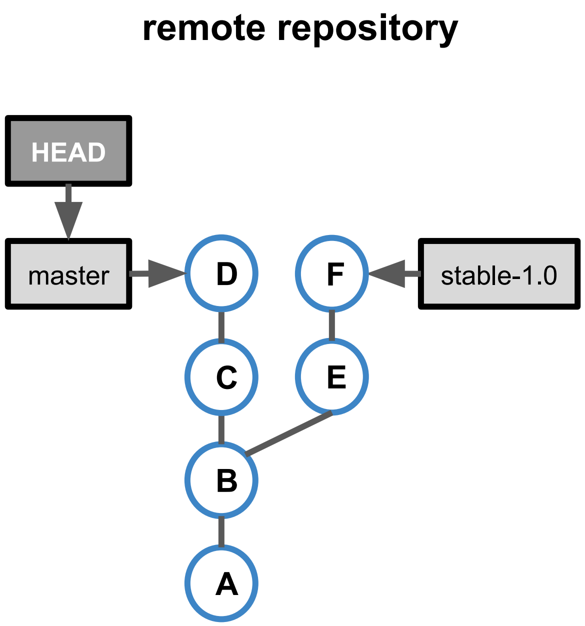


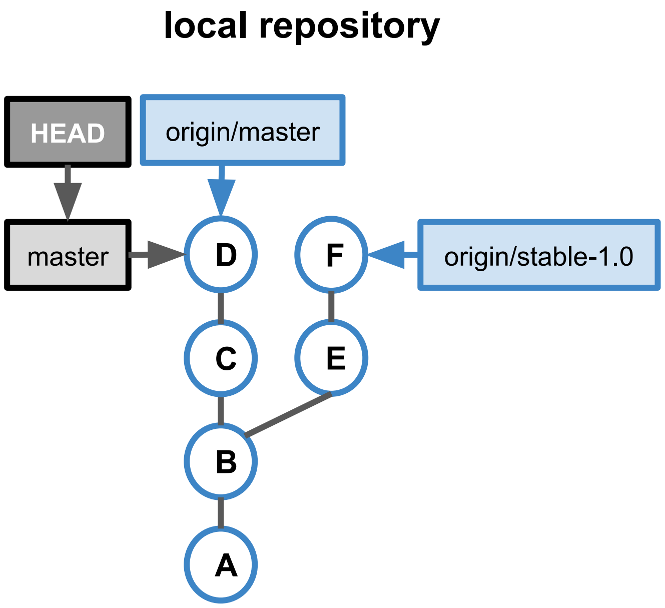
```
git clone <URL>
```

- The client gets all (reachable) commits.
- From the client's perspective the branches in the remote repository are ***remote branches***.
- For each remote branch a ***remote tracking branch*** is created in the local repository (e.g. *`origin/master`* and *`origin/stable-1.0`*).
- For the remote branch to which *`HEAD`* points a ***local branch*** is created and checked out (normally *`master`*).
- The repository URL is stored in the git config under a name, by default the remote repository is called *`origin`*.

*Q: How are remote tracking branches different from local branches?*

# Clone



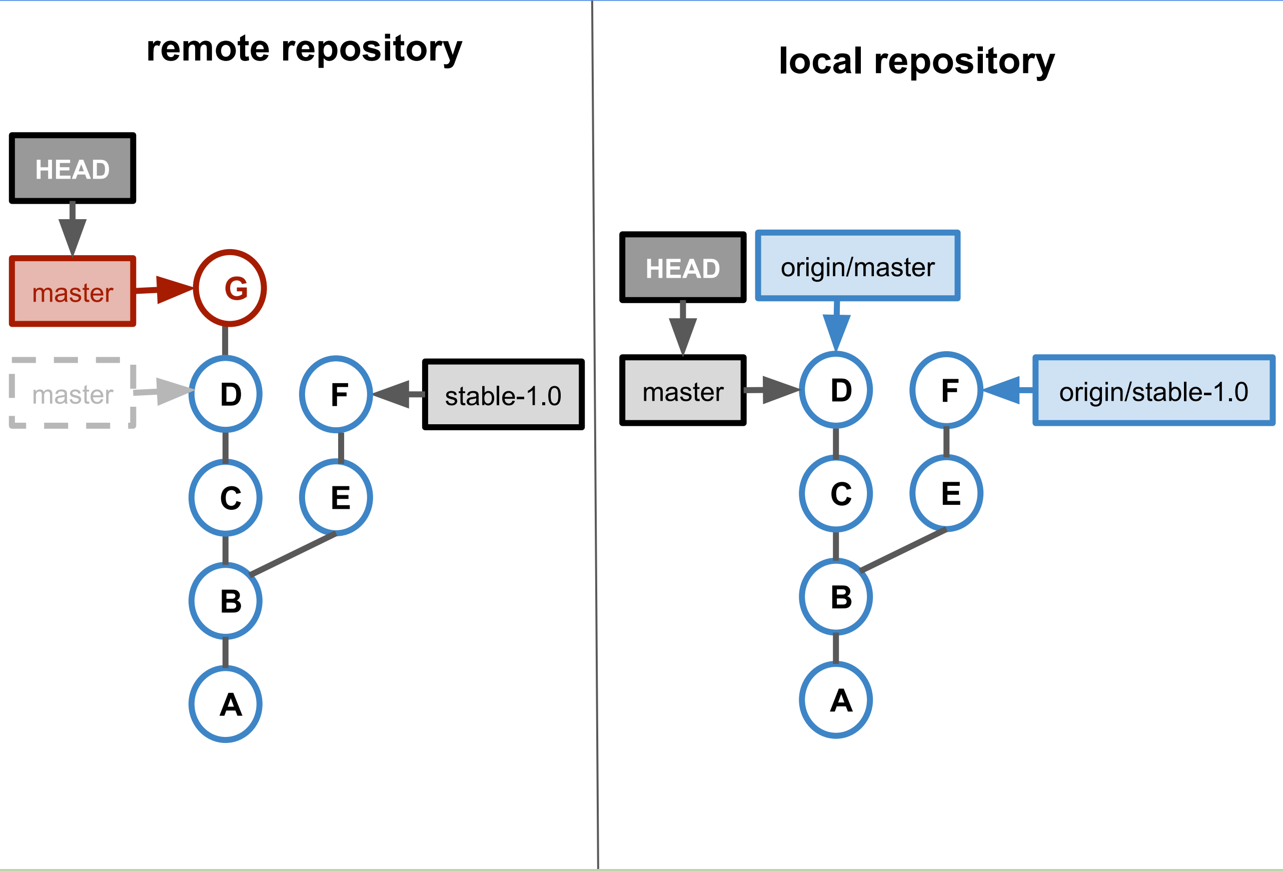


A ***remote tracking*** branch

- tracks the state of a remote branch in the local repository
- is only updated by `git fetch` and is otherwise read-only

*Q: What happens when a remote branch is updated?*

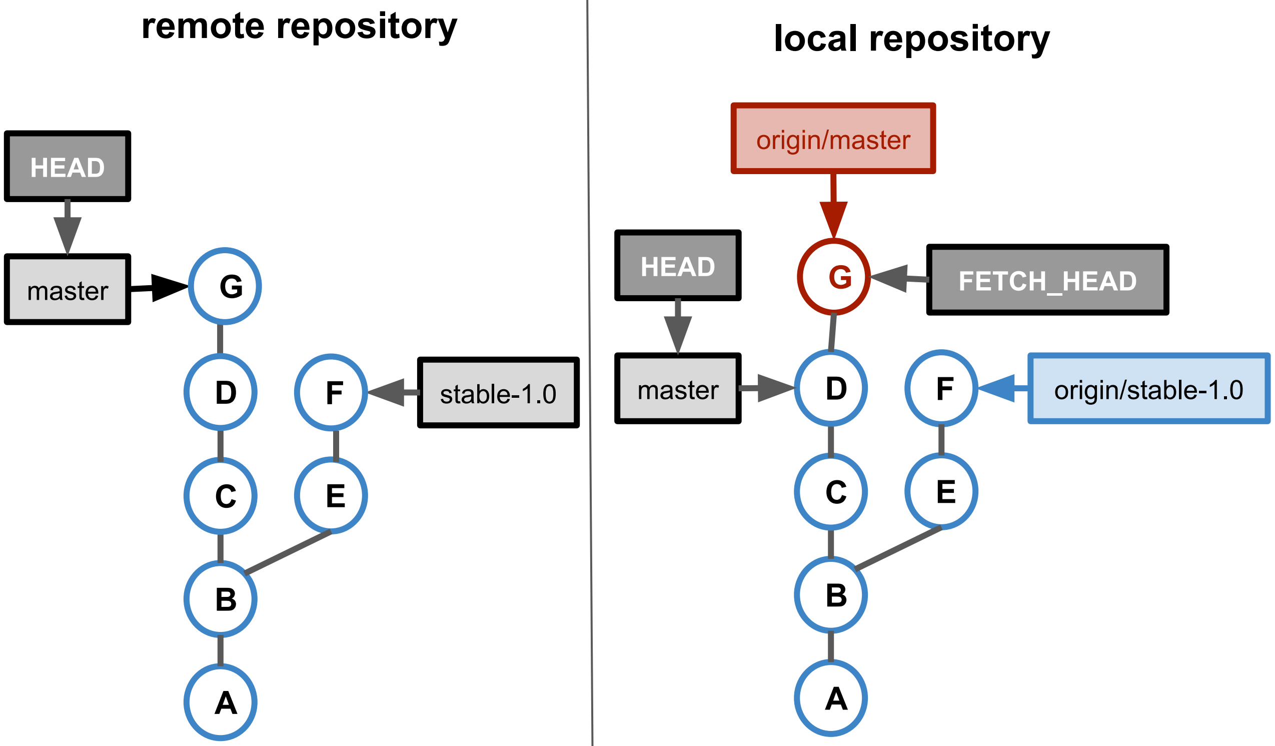
# Clone



- changes in the remote repository are only reflected in the local repository on `git fetch`

*Q: What happens on fetch?*

# Fetch

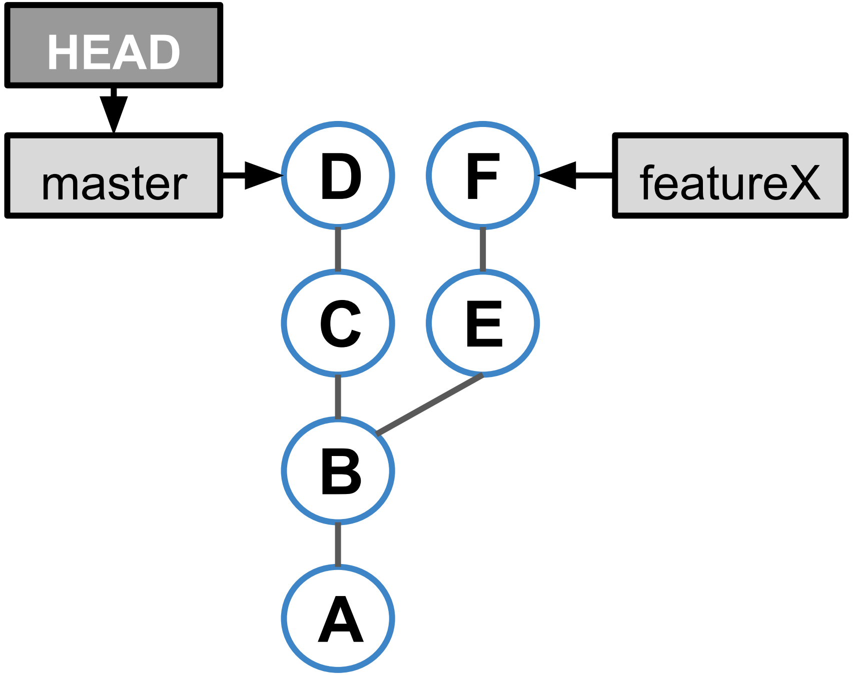


`git fetch`

- fetches new commits
- updates the ***remote tracking branches***
- never updates ***local branches***
- never changes the ***working tree***
- is always safe to do
- *`FETCH_HEAD`* points to the commit that was fetched last

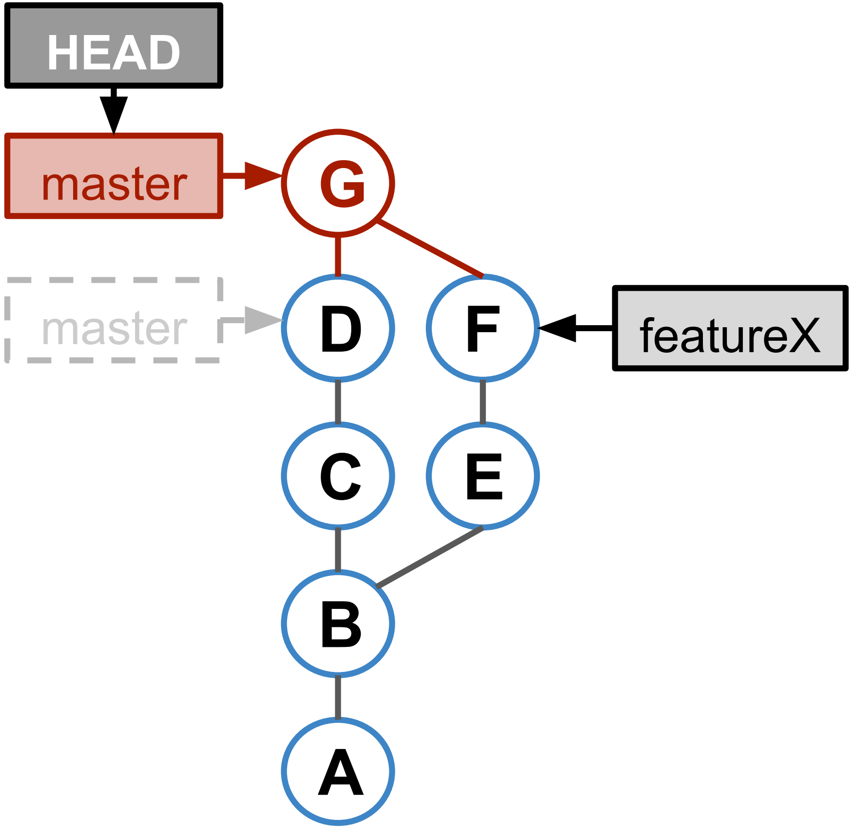
*Q: How do local branches get updated?*

# Merge



*Q: What is the result of merging the featureX branch into the master branch? Which branch is updated?*

# Merge

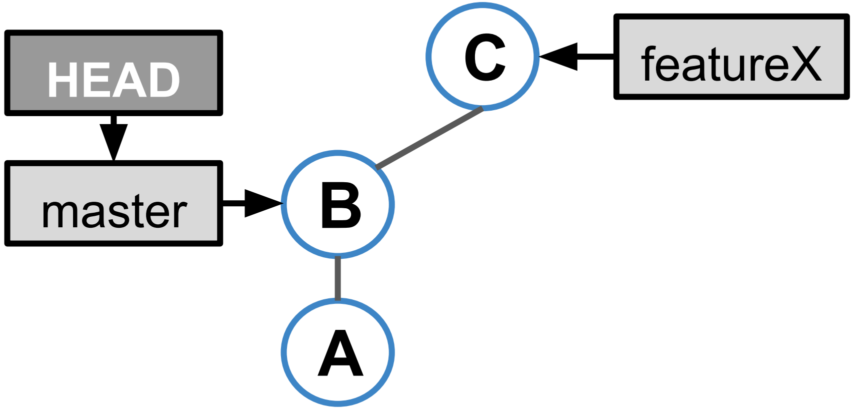


`git merge featureX`:

- Merges *`featureX`* into the current branch (*`master`*).
- Creates a ***merge commit*** (commit with more than one parent).
- The current branch is updated.

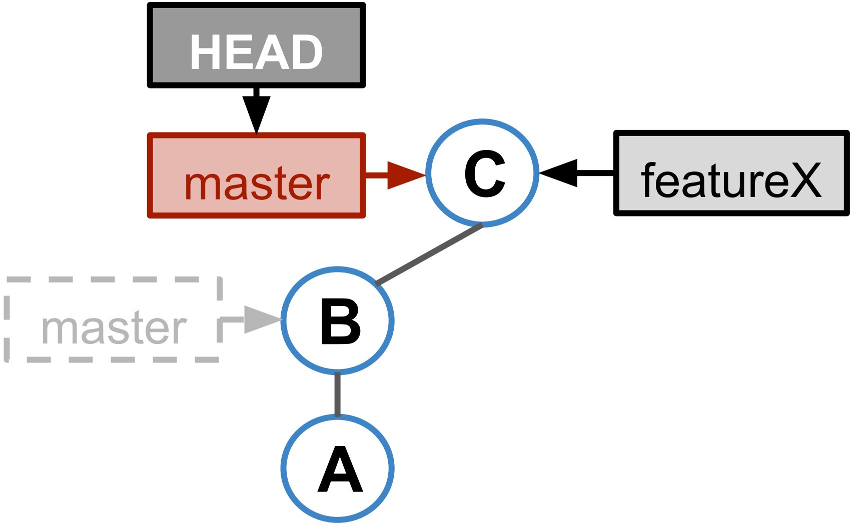
*Q: Which commits belong to the history of the master branch? What if master was merged into featureX?*

# Merge



*Q: What will be the result of merging featureX into master?*

# Merge - Fast-Forward

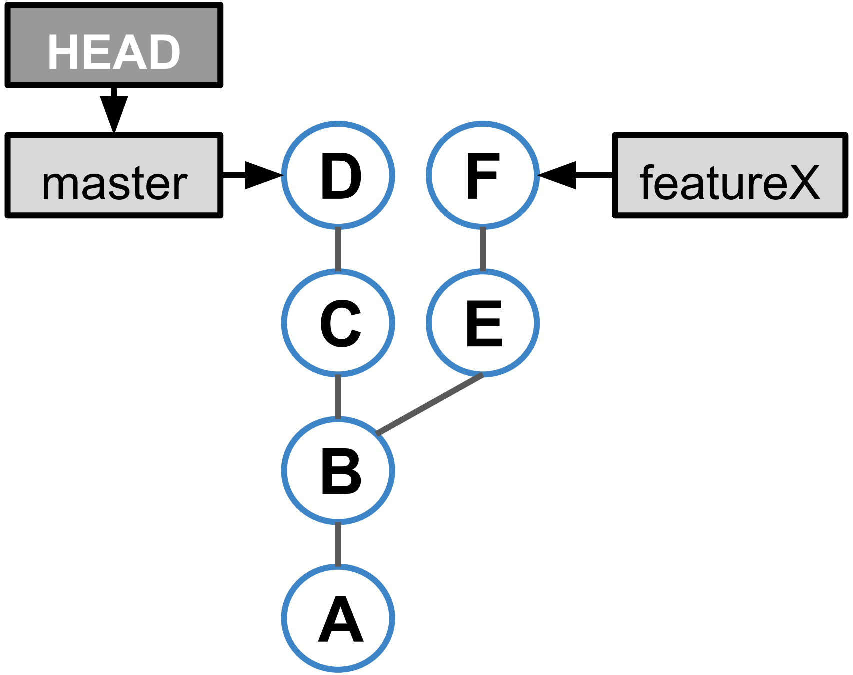


***Fast forward merge***:

- Just moves the branch pointer.
- No creation of a merge commit.
- The creation of a merge commit can be enforced by:

  *`git merge --no-ff`*

# Merge



On ***merge*** Git automatically tries to do a content merge and reports conflicts only if the same lines (+ some context) in the same file have been touched.

*Q: What if there are conflicting modifications in master and featureX?*

# Merge - Conflict Resolution

## Conflicts markers (default):

```
<<<<<<< HEAD
foo
=======
bar
>>>>>>> featureX
```

## Conflicts markers with common ancestor version:

```
<<<<<<< HEAD
foo
||||||| merged common ancestors
baz
=======
bar
>>>>>>> featureX
```

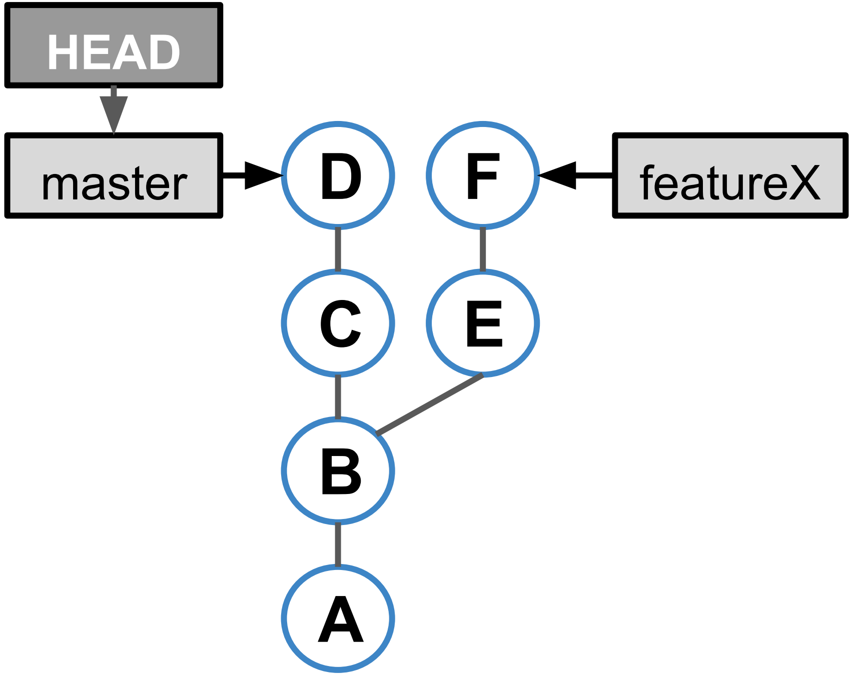
If there are conflicts:

- the merge operation stops and leaves you with a ***dirty working tree***, the conflicting files contain ***conflict markers***
- Use `git status` to see which files have conflicts.
- Resolve the conflicts manually by editing the files or accept either version by:
  - `git checkout --ours <file>`
  - `git checkout --theirs <file>`
- after resolving the conflicts the conflict resolution must be staged by `git add`
- once all conflict resolutions are staged continue with `git commit`
- to abort the merge do `git reset --hard`

Tip:

- To see the common ancestor version use `merge.conflictstyle=diff3` (git config setting)

# Cherry-Pick

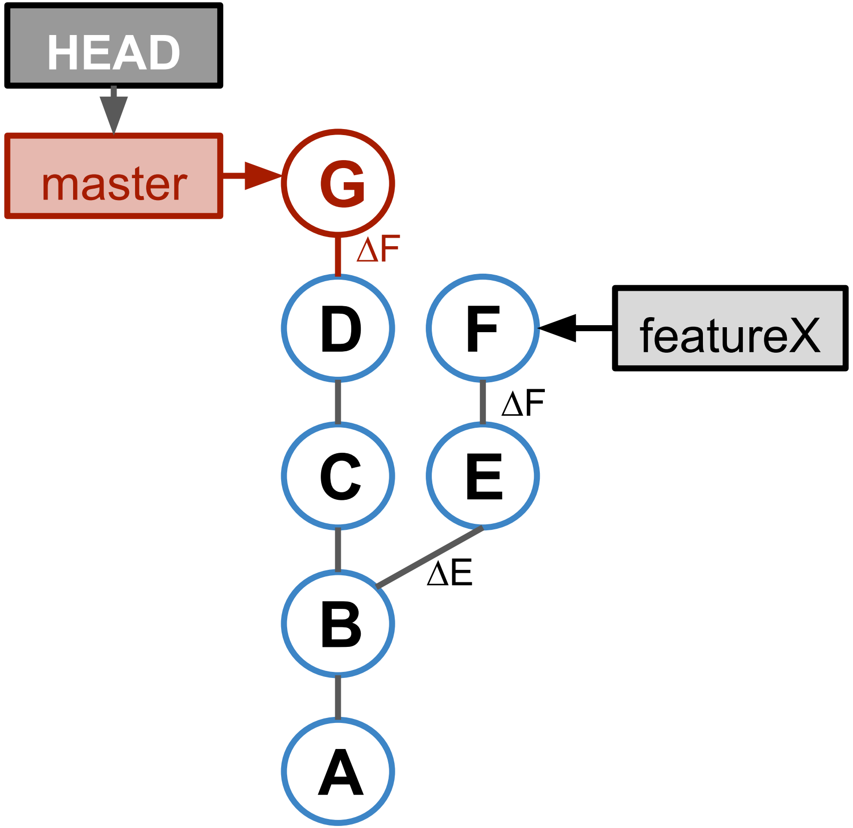


Imagine that:

- commit ***E*** implements a feature
- commit ***F*** is bug-fix
- the bug-fix ***F*** is needed in `master`

*Q: How can only the bug-fix be brought into master? Why does merge not work?*

# Cherry-Pick

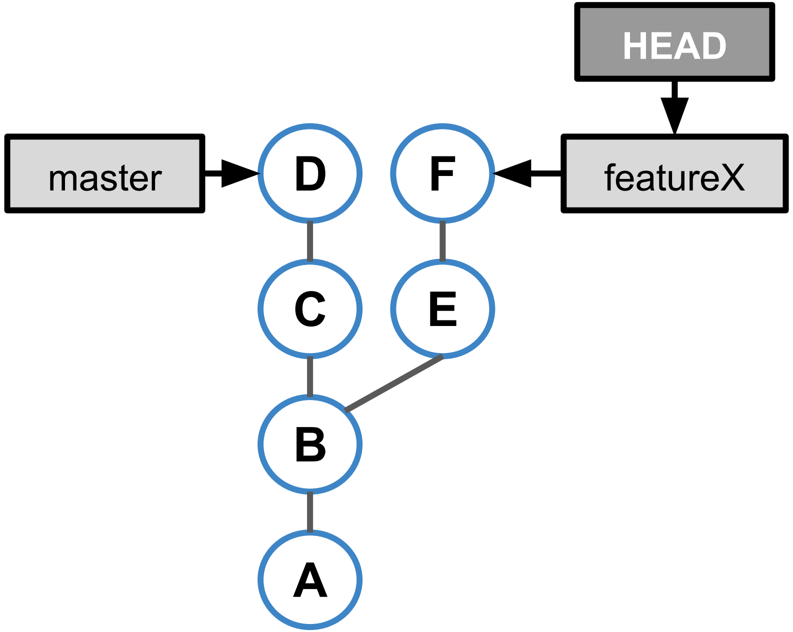


Cherry-Pick:

- Applies the modifications that were done by the commit that is cherry-picked (the commit delta) to a new base.
- The commit message is preserved.
- The new commit has no parent relation to the commit that was cherry-picked.
- The cherry-pick can fail with conflicts. The conflict resolution is done the same way as for conflicts on merge.

*Q: What is git rebase?*

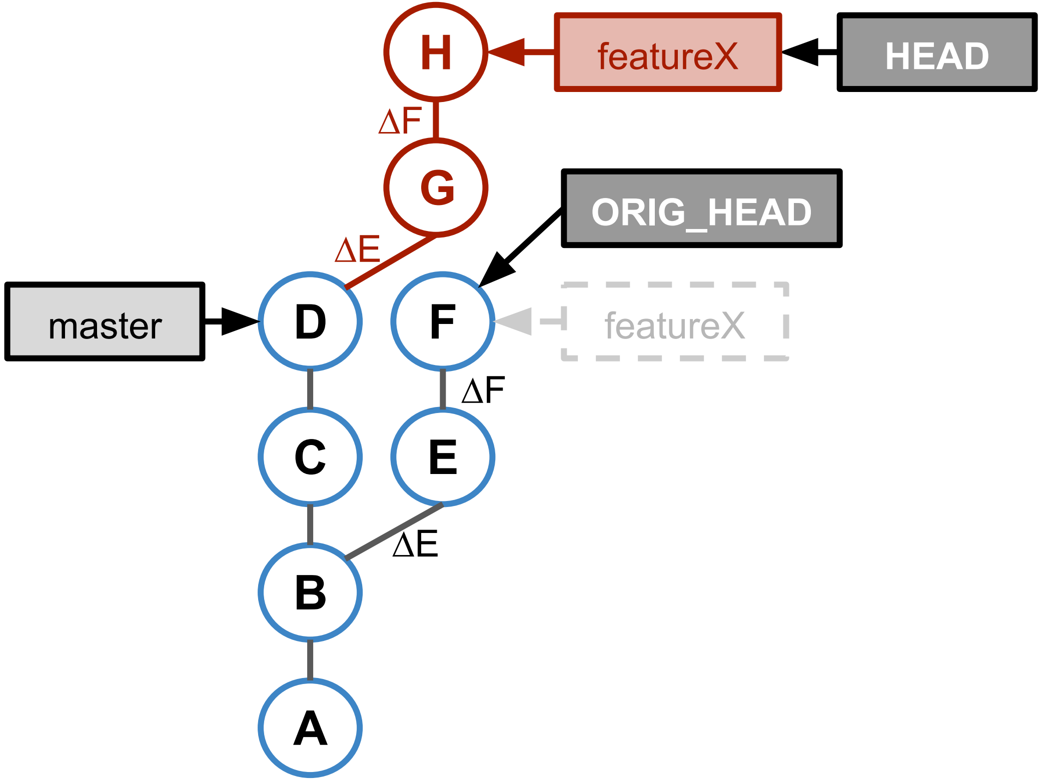
# Rebase



***Rebase***:

- redo the work that was done in the `featureX` branch on top of the `master` branch

# Rebase

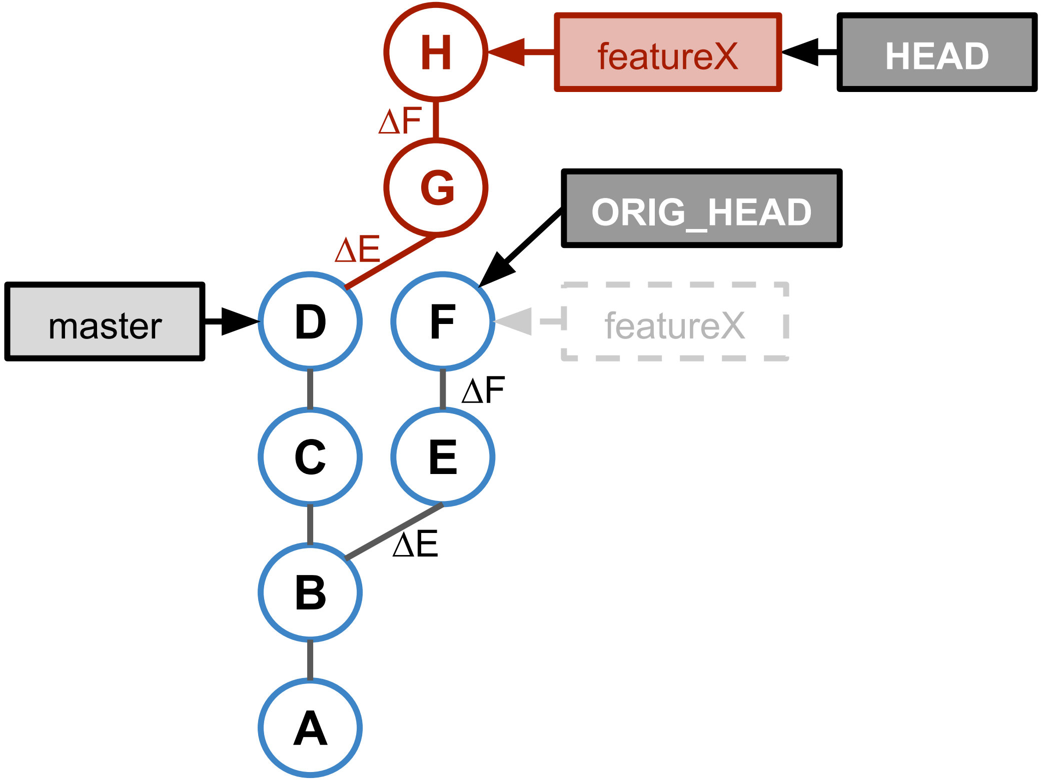


***Rebase***:

- rebases the current branch to a another ***base*** commit
- rebase = series of cherry-picks
- *`git rebase master`* rebases all commits of the *`featureX`* branch (which is currently checked out) onto the *`master`* branch.
- the commit messages are preserved
- the history of the *`featureX`* branch is rewritten, this is bad if the *`featureX`* branch is shared and others have based work on top of it
- the old commits ***E*** and ***F*** still exist in the repository
- after the rebase a ***fast-forward*** of *`master`* is possible
- Linear history
- *`ORIG_HEAD`* points to the old *`HEAD`*

*Q: How is conflict resolution on rebase different from resolving conflicts on merge?*

# Rebase - Conflict Resolution



For each commit that is rebased there can be conflicts:

- the rebase operation stops at the commit that has conflicts and leaves you with a ***dirty working tree***, the conflicting files contain ***conflict markers***
- Use `git status` to see which files have conflicts.
- Resolve the conflicts manually by editing the files or accept either version by:
  - `git checkout --ours <file>`
  - `git checkout --theirs <file>`
- after resolving the conflicts the conflict resolution must be staged by `git add`
- once all conflict resolutions are staged continue with `git rebase --continue`
- to abort the rebase do `git rebase --abort`

*Q: What is git pull?*

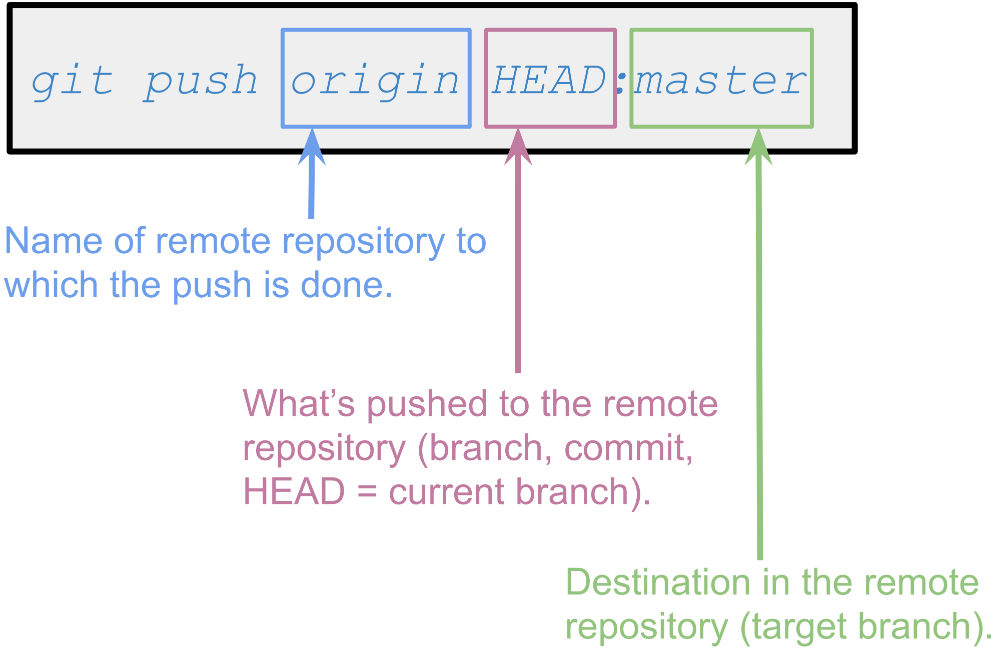
# Pull

*`git pull`* = *`git fetch`* + *`git merge FETCH_HEAD`*

***Pull***:

- *`git pull`* can be configured to do *`git fetch`* + *`git rebase`* instead (config parameter *`pull.rebase=true`*)

# Push



```
git push origin HEAD:master
```

Name of remote repository to which the push is done.

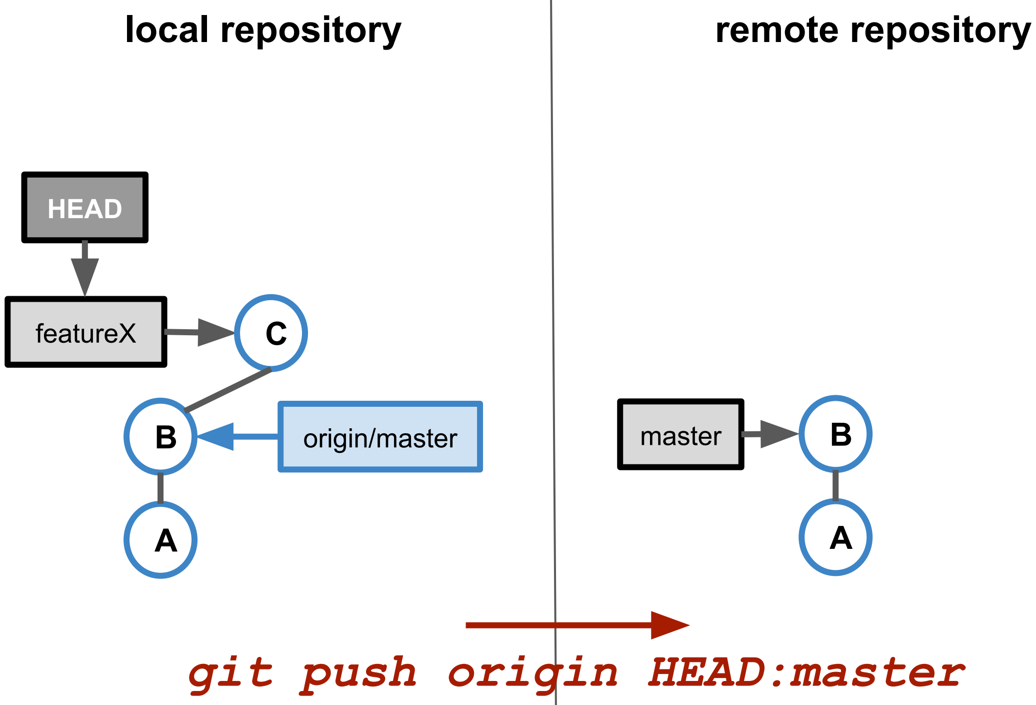
What’s pushed to the remote repository (branch, commit, HEAD = current branch).

Destination in the remote repository (target branch).

***Push***:

- pushes commits from the ***local repository*** to a ***remote repository*** (more precisely from a ***local branch*** to a ***remote branch***)
- *`git push origin HEAD:master`* is equivalent to *`git push origin HEAD:refs/heads/master`*
- *`git push origin master`* is equivalent to *`git push origin refs/heads/master:refs/heads/master`*

# Push



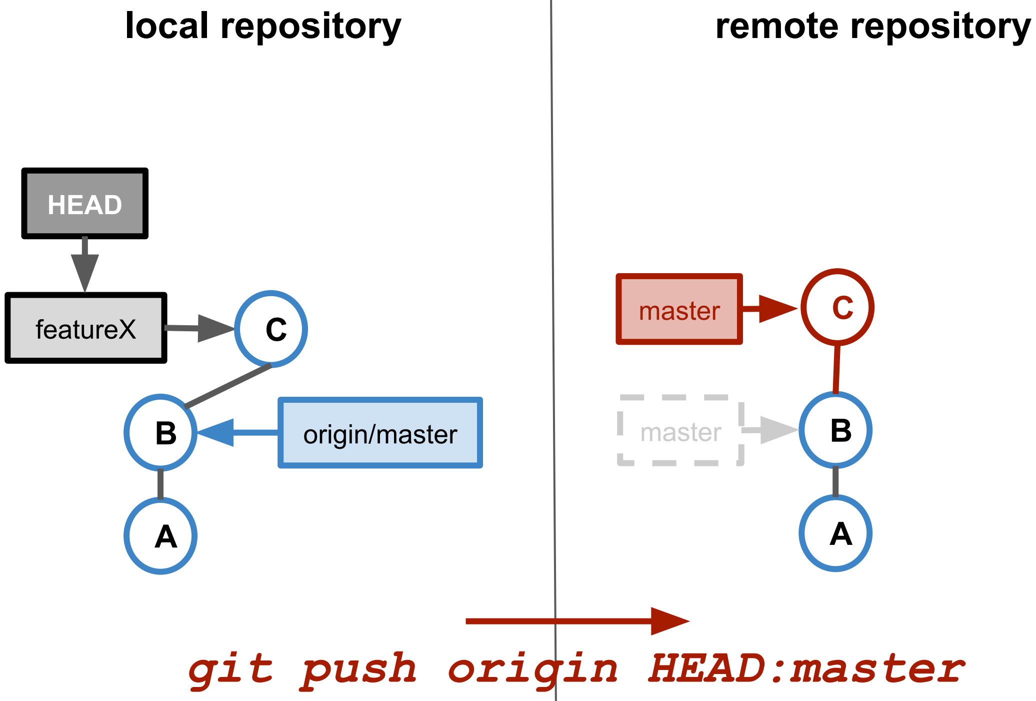
```
git push origin HEAD:master
```

Situation:

- The remote repository was cloned, a local `featureX` branch was created and in this branch a commit ***C*** was created.

*Q: What happens on git push?*

# Push

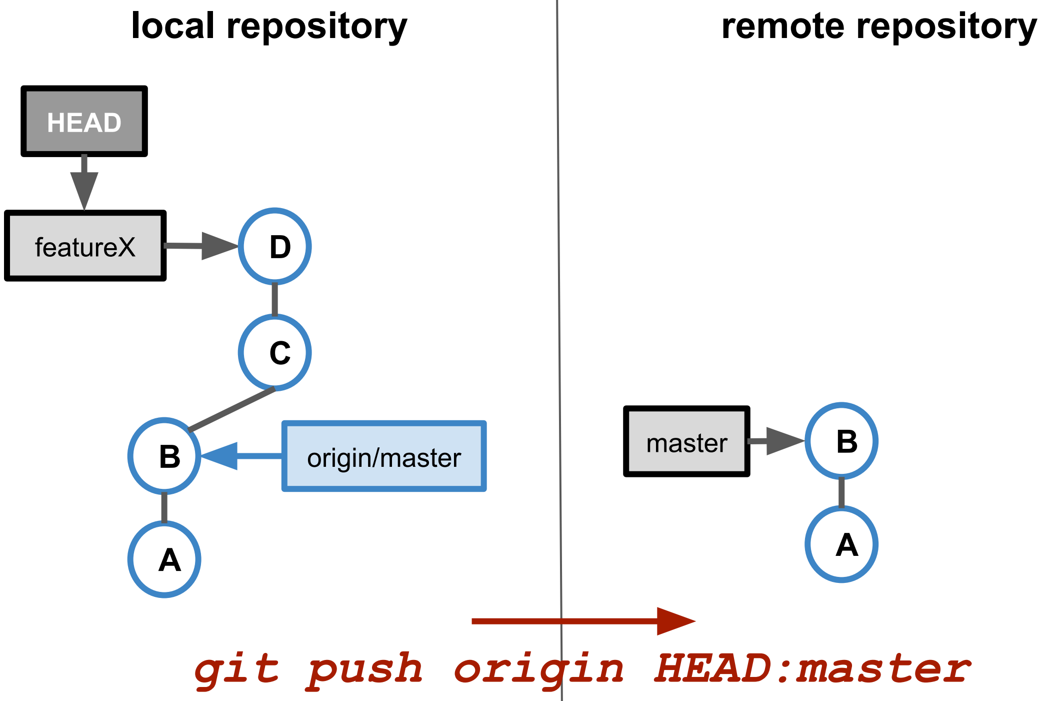


`git push origin HEAD:master`

Push:

- pushes commit ***C*** to the remote repository
- updates the `master` branch in the remote repository
- The local branch name is never transferred to the remote repository.

# Push



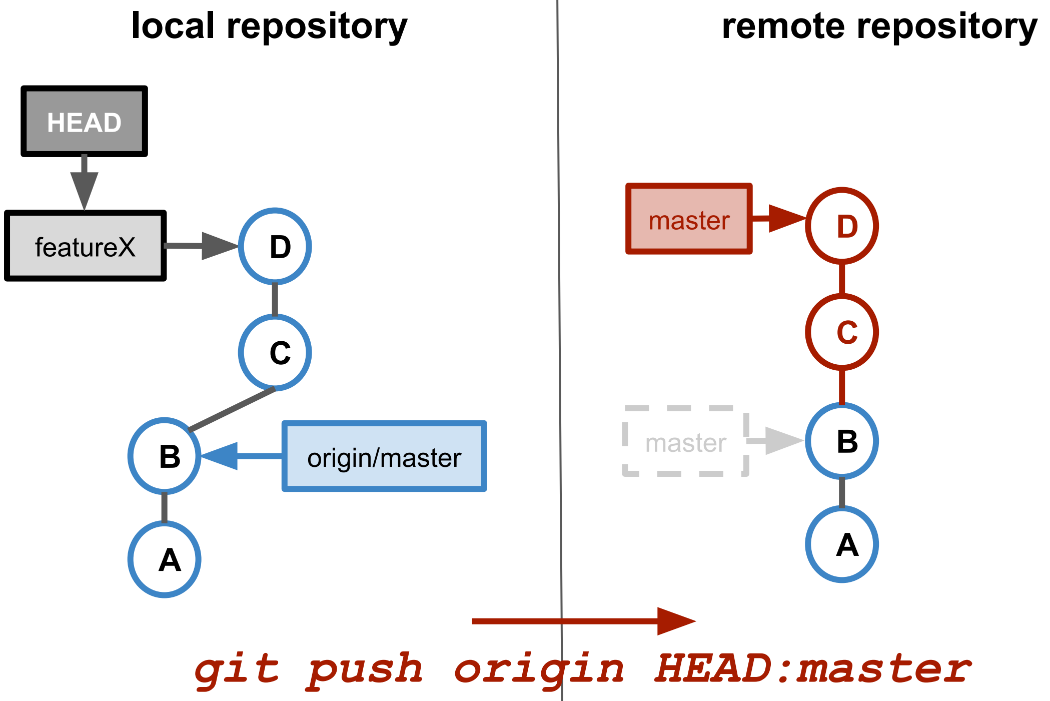
*git push origin HEAD:master*

Situation:

- The remote repository was cloned, a local *featureX* branch was created and in this branch two commits, ***C*** and ***D***, were created.

*Q: Which commits get pushed?*

# Push

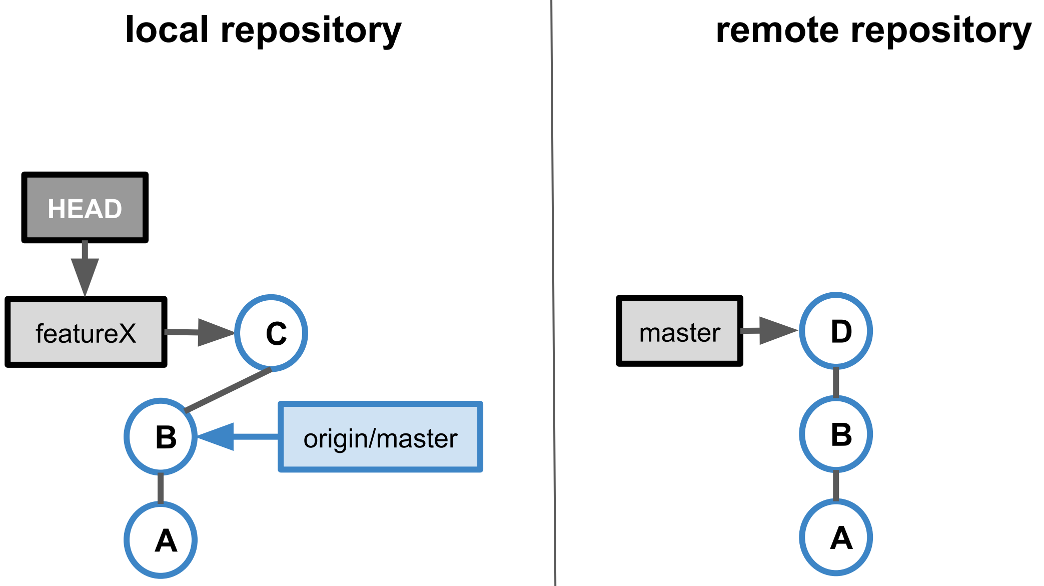


```
git push origin HEAD:master
```

Push:

- pushes **all** commits which are reachable from the pushed commit and which are not available in the remote repository

# Push

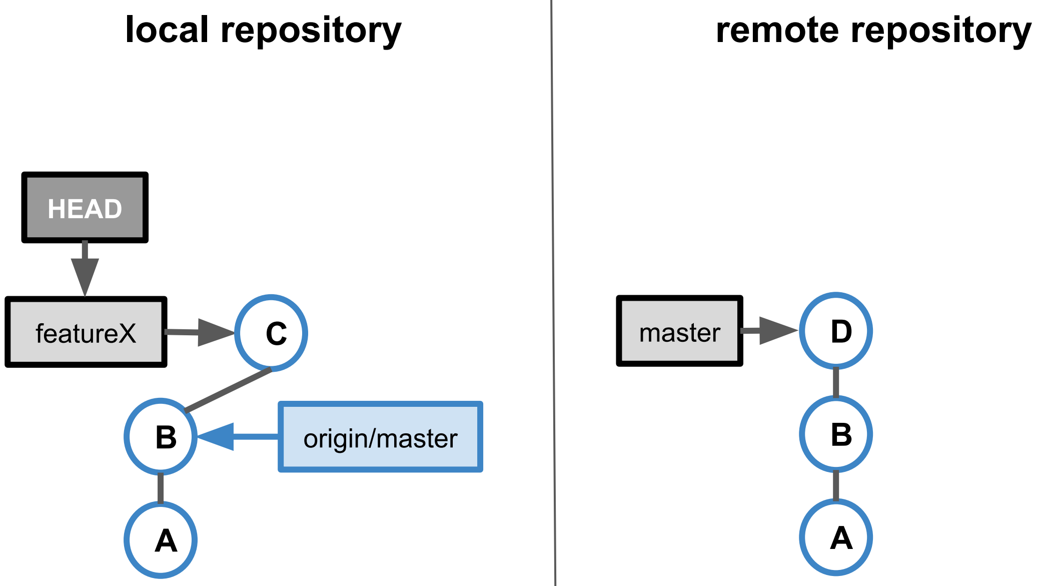


Situation:

- The remote repository was cloned, a local `featureX` branch was created and in this branch a commit ***C*** was created. In the meantime `master` in the remote repository was updated to commit ***D***.

*Q: What happens on push?*

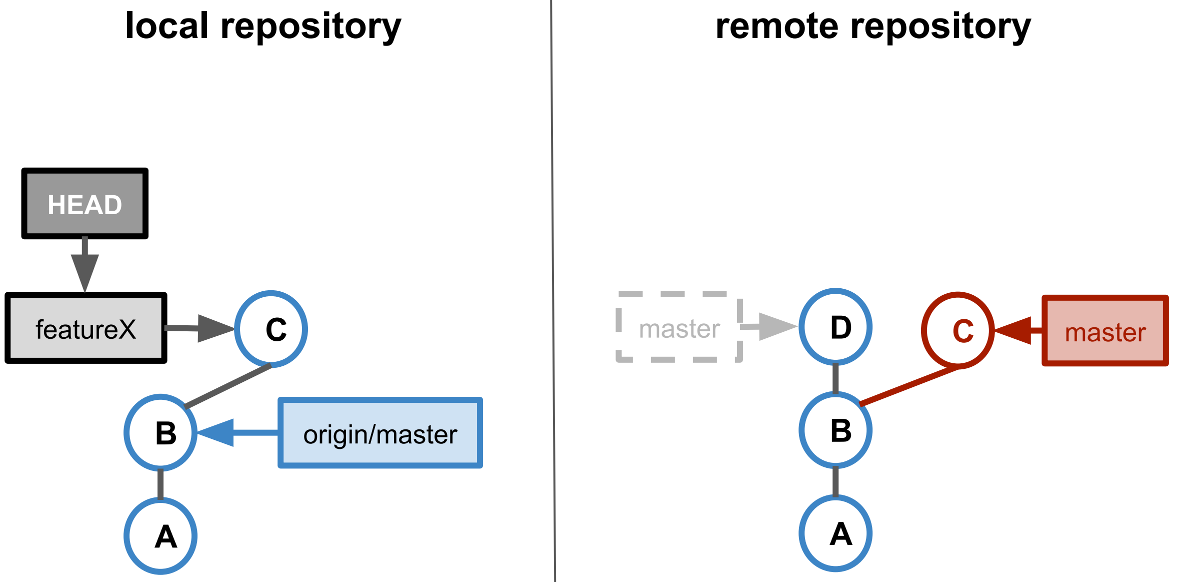
# Push



`git push` fails if the target branch cannot be fast-forwarded to the pushed commit.

*Q: What happens on force push?*

# Force Push



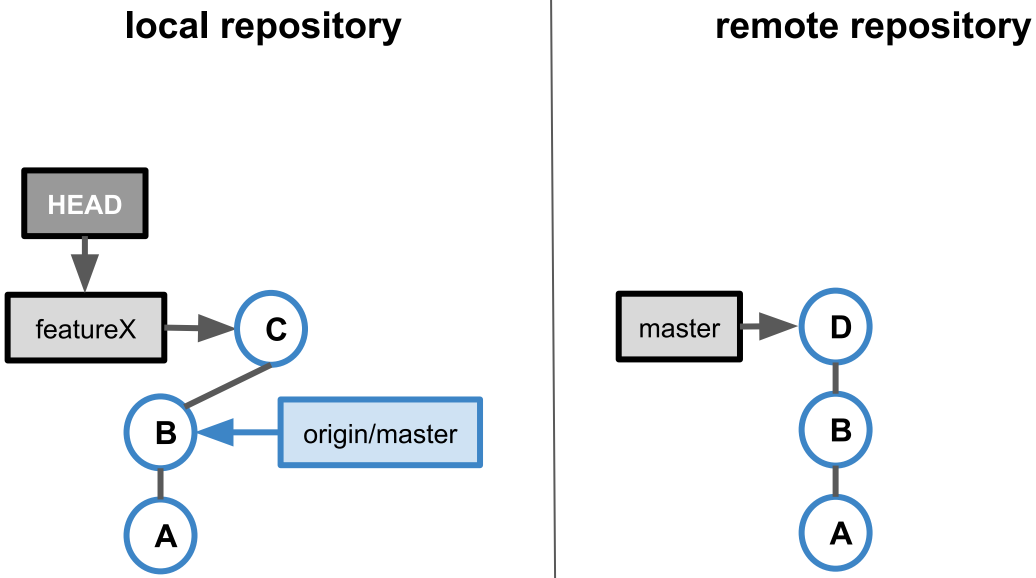
```
git push --force origin HEAD:master
```

Force push:

- Makes the push succeed even if the target branch cannot be fast-forwarded.
- The target branch is updated to the pushed commit, conflicting commits are removed from the history of the target branch!
- After the force push commit ***D*** is no longer contained in the history of the *`master`* branch.

*Q: How can the push succeed without discarding commit D?*

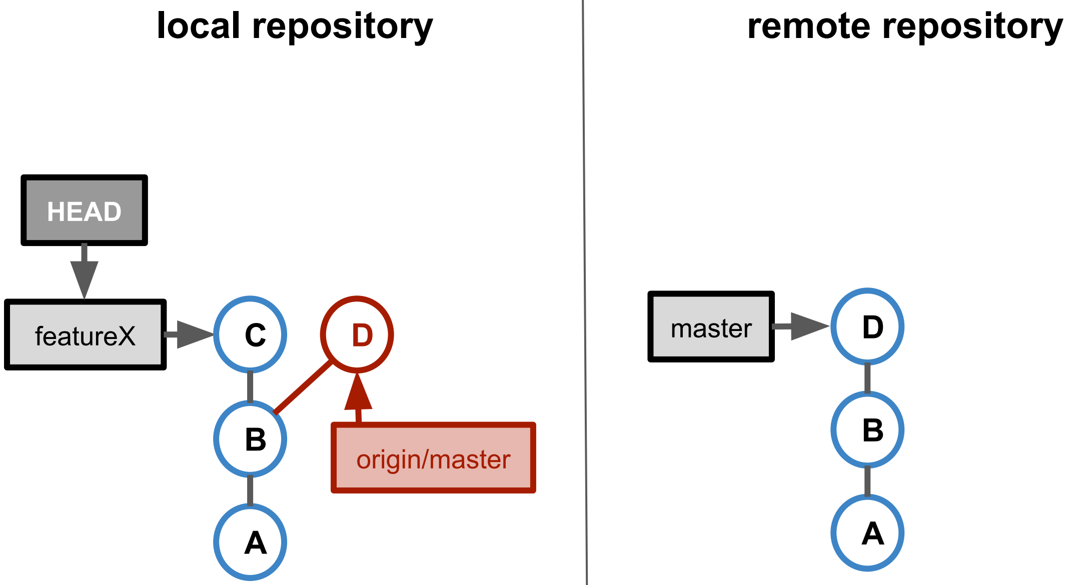
# Possibility 1: fetch, merge, push



Situation:

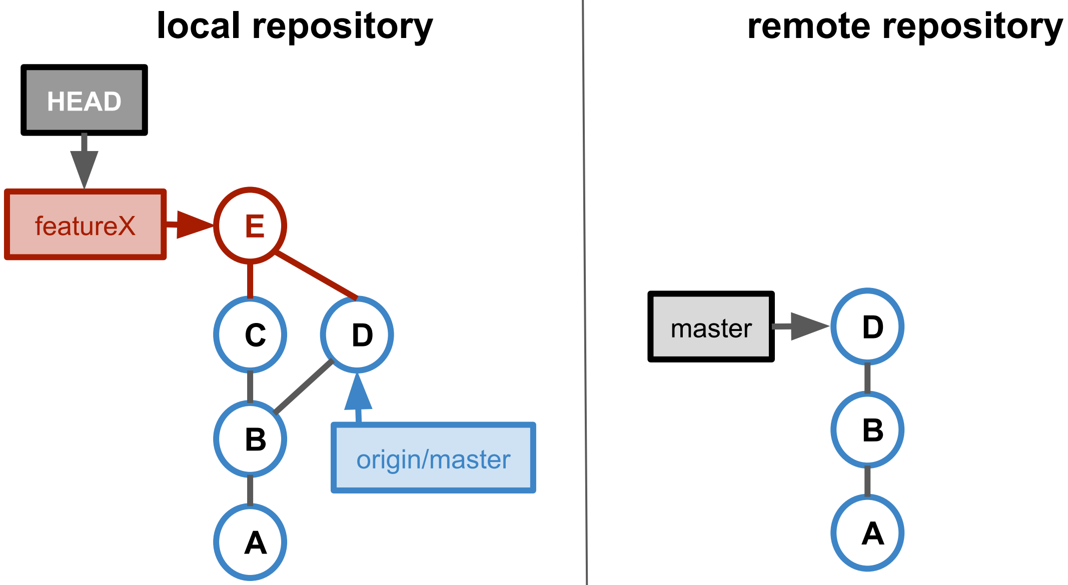
- Commit ***C*** cannot be pushed because the *`master`* branch in the remote repository cannot be fast-forwarded to it (it conflicts with commit ***D***).

# Possibility 1: fetch, merge, push



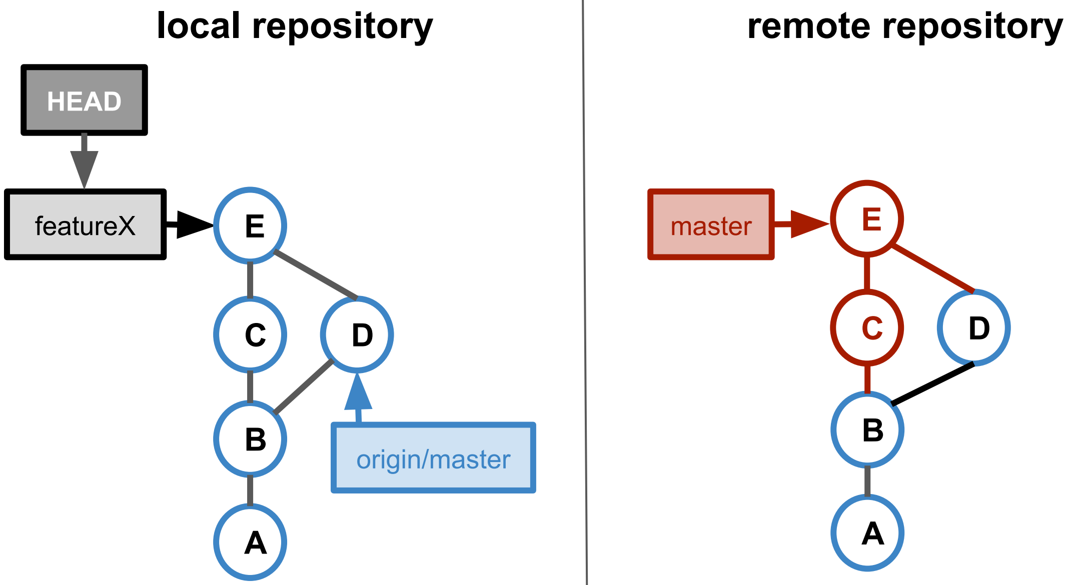
1. *`git fetch origin`*: Retrieves commit ***D*** and updates the remote tracking branch.

# Possibility 1: fetch, merge, push



1. *`git fetch origin`*: Retrieves commit ***D*** and updates the remote tracking branch.
2. *`git merge origin/master`*: Merges commit ***D*** into the *`featureX`* branch.

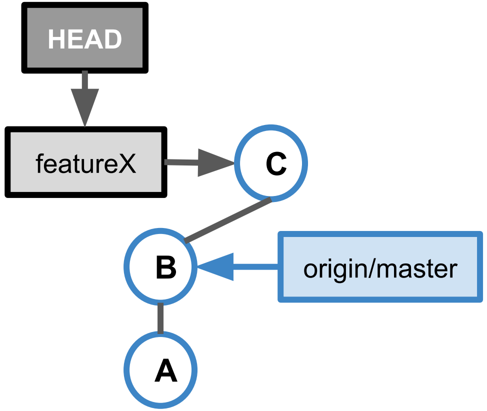
# Possibility 1: fetch, merge, push



1. *`git fetch origin`*: Retrieves commit ***D*** and updates the remote tracking branch.
2. *`git merge origin/master`*: Merges commit ***D*** into the *`featureX`* branch.
3. *`git push origin HEAD:master:`* Push of commit ***E*** succeeds now because the *`master`* branch can be fast-forwarded to it.

# Possibility 2: fetch, rebase, push

**local repository**



HEAD

featureX

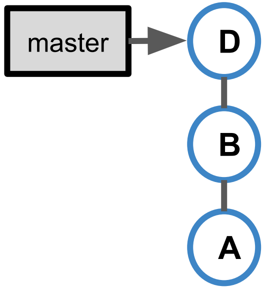
C

B

origin/master

A

**remote repository**



master

D

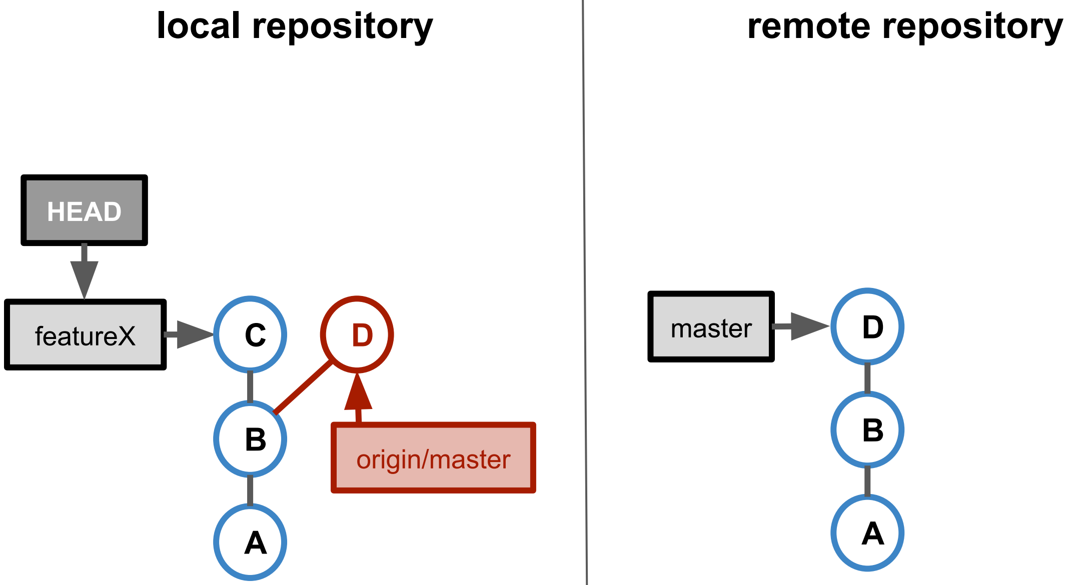
B

A

Situation:

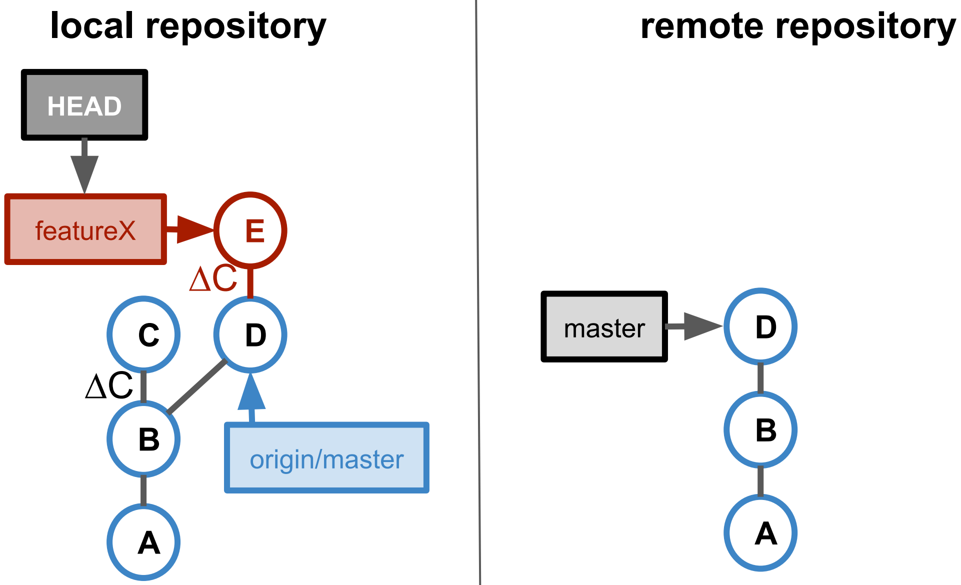
- Commit ***C*** cannot be pushed because the `master` branch in the remote repository cannot be fast-forwarded to it (it conflicts with commit ***D***).

# Possibility 2: fetch, rebase, push



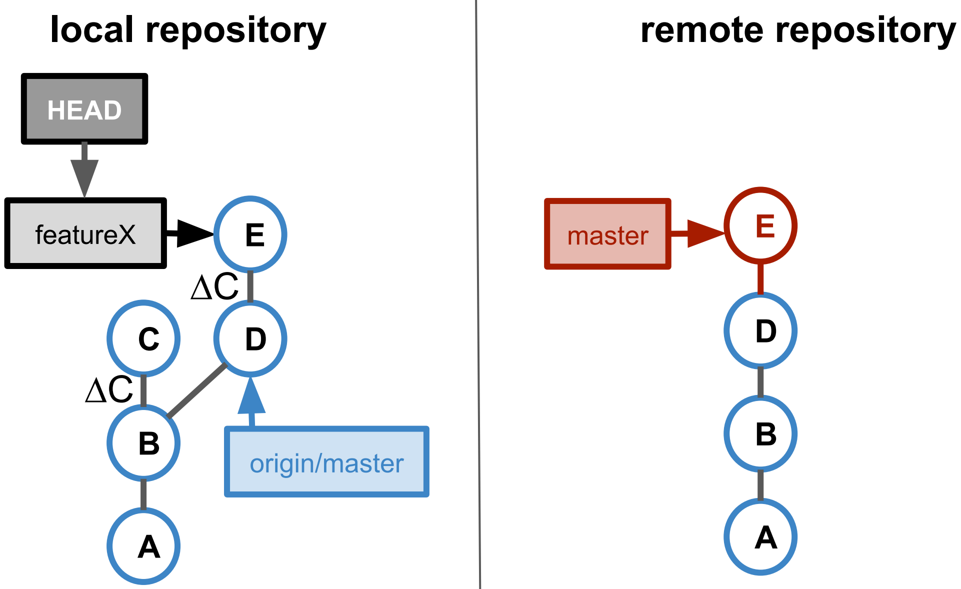
1. *`git fetch origin`*: Retrieves commit ***D*** and updates the remote tracking branch.

# Possibility 2: fetch, rebase, push



1. *`git fetch origin`*: Retrieves commit ***D*** and updates the remote tracking branch.
2. *`git rebase origin/master`*: Rebases commit ***C*** onto the commit ***D*** which creates commit ***E***.

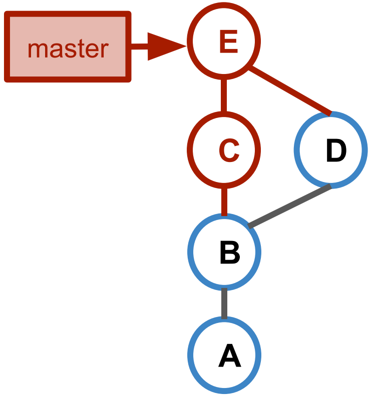
# Possibility 2: fetch, rebase, push



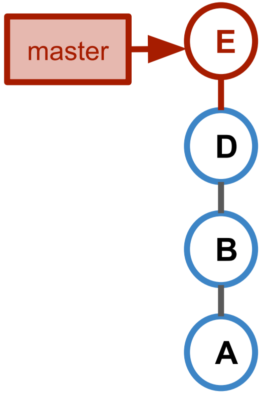
1. *`git fetch origin`*: Retrieves commit ***D*** and updates the remote tracking branch.
2. *`git rebase origin/master`*: Rebases commit ***C*** onto the commit ***D*** which creates commit ***E***.
3. *`git push origin HEAD:master:`* Push of commit ***E*** succeeds now because the *`master`* branch can be fast-forwarded to it.

# Differences between merge and rebase

**remote repository**
**after fetch, merge, push**



**remote repository**
**after fetch, rebase, push**



Content-wise the result is exactly the same in both cases. In both cases the same number of conflicts needs to be resolved.

With ***merge***:

- Two commits, commit ***C*** which implements the feature and the merge commit ***E*** (may contain conflict resolution).
- Diverged history.
- From the history one can see that commit ***C*** was originally developed based on commit **B**.

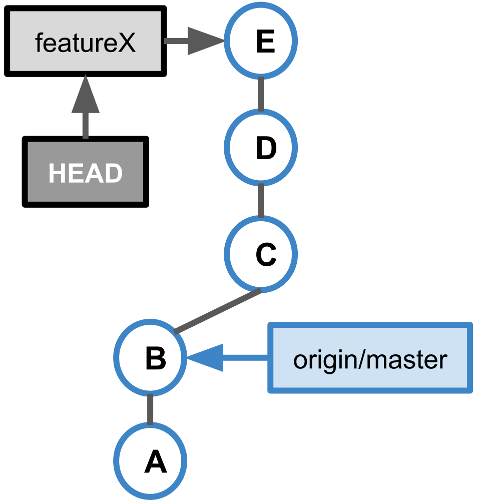
With ***rebase***:

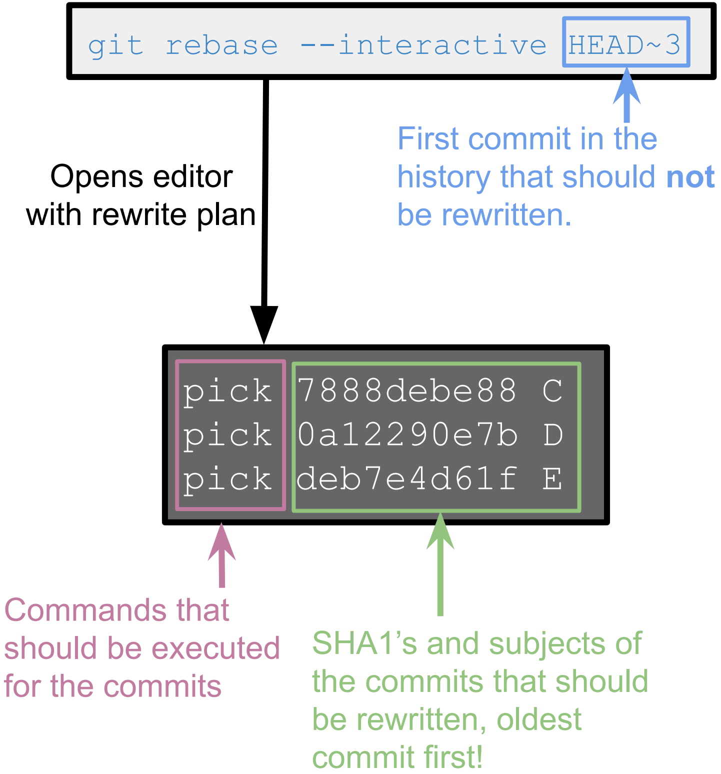
- Single commit, it looks like commit ***C*** was developed based on commit ***D***.
- Linear history.

*Q: What is better, merge or rebase?*

# B-R-E-A-K

# Interactive Rebase





```
git rebase --interactive HEAD~3
```

Opens editor with rewrite plan

First commit in the history that should **not** be rewritten.

```
pick 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

Commands that should be executed for the commits

SHA1's and subjects of the commits that should be rewritten, oldest commit first!

Situation:

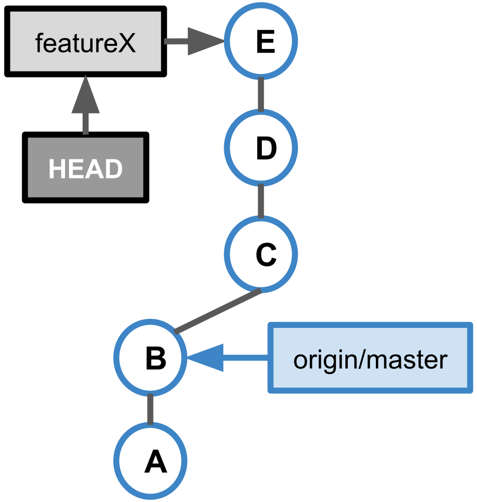
- in the local branch *featureX* three commits, ***C***, ***D*** and ***E***, have been done to implement a feature
- the commits have not been pushed yet

***Interactive rebase***

- rewrites the last n commits
- allows to update, squash, split commits while they are rewritten
- rewrites the history of the branch (you should never rewrite commits that have already been shared with others)

# Interactive Rebase

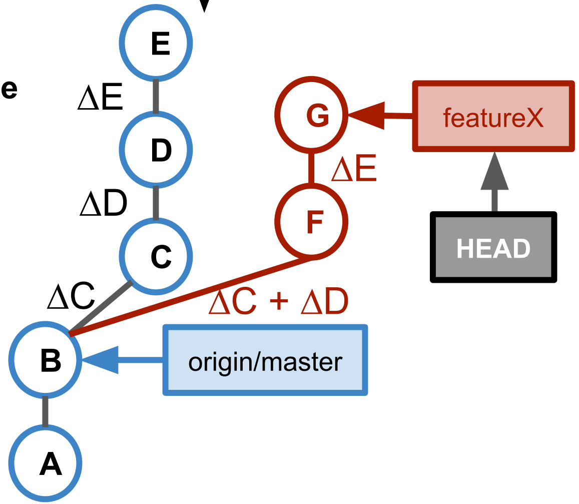
**Before interactive rebase:**



```
git rebase --interactive HEAD~3
```

```
pick   7888debe88 C
squash 0a12290e7b D
pick   deb7e4d61f E
```

**After interactive rebase:**

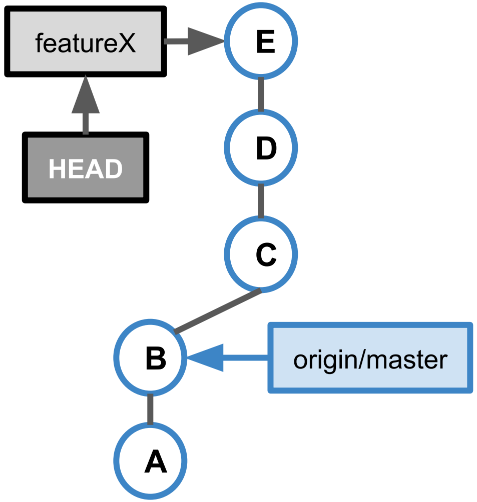


Possible commands:

- *pick* (default):
  Cherry-pick the commit unchanged.
- *reword*:
  Rewrite the commit message of the commit.
- *edit*:
  Stop at this commit so that it can be rewritten by
  *git commit --amend*, continue rebase by
  *git rebase --continue*
- *squash*:
  Squash the commit into the predecessor commit. Allows to edit the combined commit message.
- *fix*:
  Same as *squash*, but automatically takes the commit message of the predecessor commit.

# Interactive Rebase

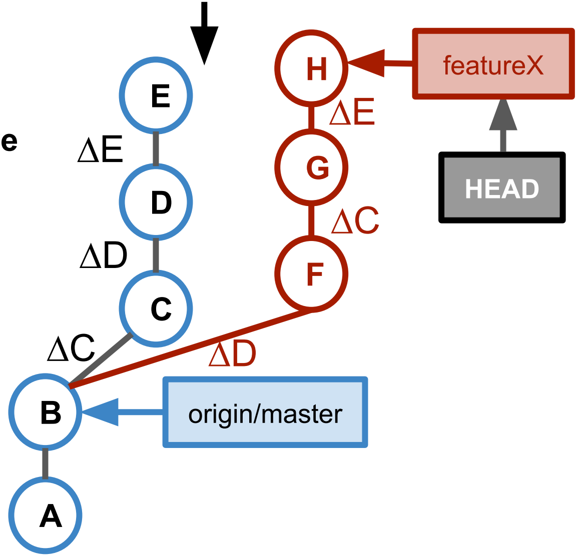
**Before interactive rebase:**



```
git rebase --interactive HEAD~3
```

```
pick 0a12290e7b D
pick 7888debe88 C
pick deb7e4d61f E
```

**After interactive rebase:**



***Interactive rebase*** also allows to reorder and drop commits. For this simply change the order of the lines in the interactive rebase editor, deleting a line means that this commit is dropped. Additional commits can be inserted by using `edit` on a commit and then creating new commits with `git commit` (rather than amending the commit with `git commit --amend`).

***Conflicts*** can appear on each stage of the interactive rebase:

- if applying a commit results in conflicts the interactive rebase stops at this point, conflicting files have conflict markers, after the conflict resolution is staged you can continue the rebase by: `git rebase --continue`
- the interactive rebase can be aborted by: `git rebase --abort`

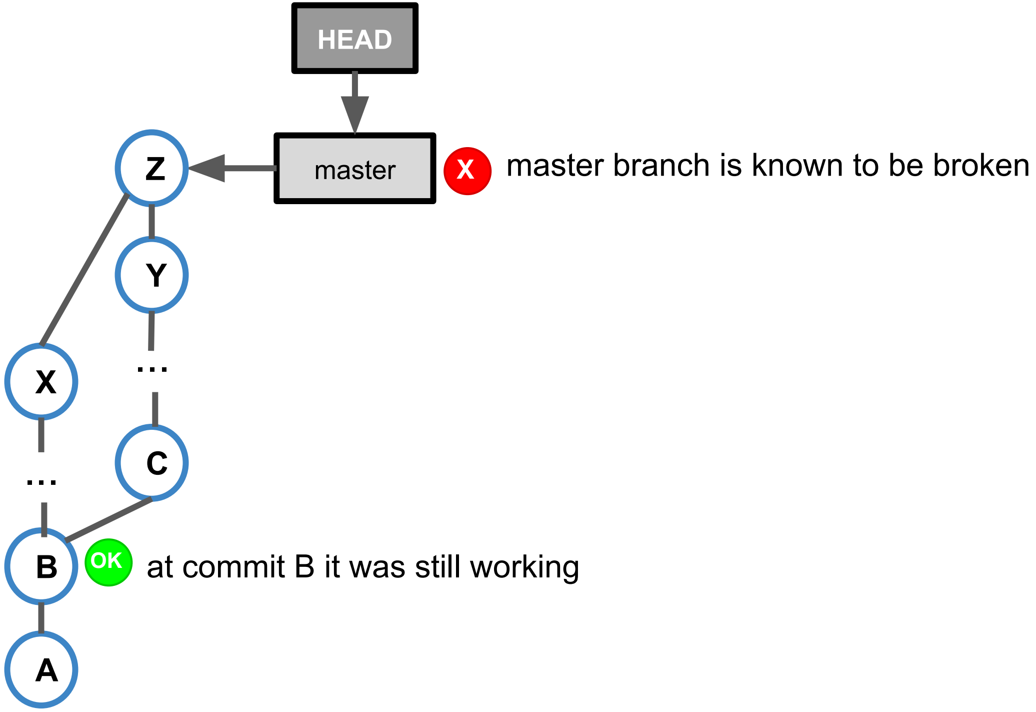
# Blame

```
...
07952c069ab (Edwin Kempin     2016-04-07 14:00:17 +0200  81)      SshKeyCache sshKeyCache,
07952c069ab (Edwin Kempin     2016-04-07 14:00:17 +0200  82)      AccountCache accountCache,
07952c069ab (Edwin Kempin     2016-04-07 14:00:17 +0200  83)      AccountByEmailCache byEmailCache,
54ba43a51dc (Dave Borowitz    2014-11-25 14:41:05 -0500  84)      AccountLoader.Factory infoLoader,
2461265e486 (Michael Ochmann  2016-02-12 17:26:18 +0100  85)      DynamicSet<AccountExternalIdCreator> extIdCreators,
07952c069ab (Edwin Kempin     2016-04-07 14:00:17 +0200  86)      AuditService auditService,
744d2b89671 (Edwin Kempin     2017-02-15 11:10:59 +0100  87)      ExternalIdsUpdate.User externalIdsUpdateFactory,
07952c069ab (Edwin Kempin     2016-04-07 14:00:17 +0200  88)      @Assisted String username) {
...
```

`git blame <file>` shows for each line in the file when it was last modified, by whom and by which commit.

If a bug is spotted `git blame` can help to find out by which commit the bug was introduced. Once the bad commit is identified you can use `git branch -r --contains <bad-commit>` to find all branches which contain the bug and may need to be fixed.

# Bisect



master branch is known to be broken

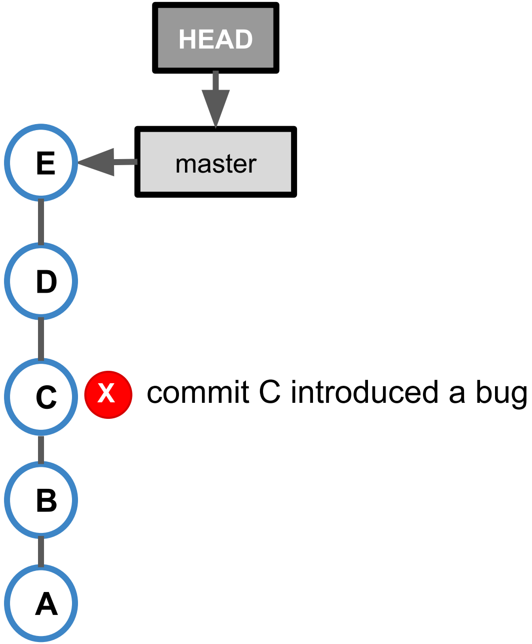
at commit B it was still working

`git bisect` finds the commit that has introduced a bug by doing a binary search over the git history:

```
$ git bisect start
$ git bisect bad
$ git bisect good <last-known-good>
```

Now `git bisect` will checkout different commits and for each of them you should probe whether the bug is present and then tell `git bisect` whether the commit is good (`git bisect good`) or bad (`git bisect bad`). If a commit cannot be probed skip it (`git bisect skip`). The probing of commits can be automated by a script. At the end `git bisect` will present you the first bad commit that introduced the bug.

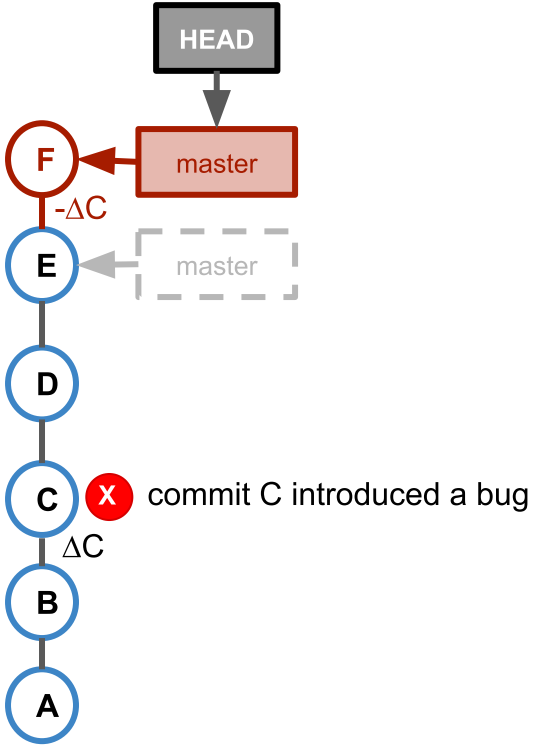
# Revert



commit C introduced a bug

`git revert` undos the changes that have been done by a commit by creating a new commit that applies the inverted changes.

# Revert



commit C introduced a bug

`git revert` may fail due to conflicts. After resolving the conflicts and staging the conflict resolution you can create the revert commit by `git commit`.

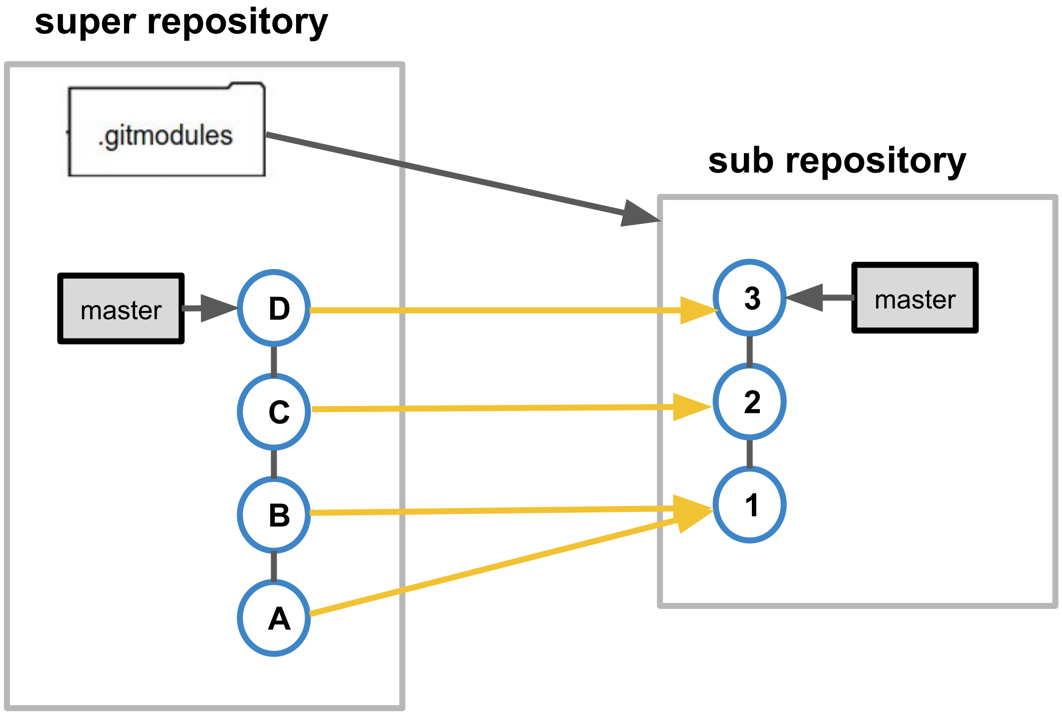
# Reflog

```
a9456bf (HEAD, origin/master, origin/HEAD) HEAD@{0}: checkout: moving from
5ff36200b29567118b3aede8e49ba0b6c6b1adb1 to a9456bfdb862dfa7197583decac3c22149ae8109
5ff3620 (origin/stable-2.16) HEAD@{1}: checkout: moving from 5d607a4193fb41b4ad8fe01622f7e002fd4208c0 to
5ff36200b29567118b3aede8e49ba0b6c6b1adb1
5d607a4 HEAD@{2}: checkout: moving from a9456bfdb862dfa7197583decac3c22149ae8109 to
5d607a4193fb41b4ad8fe01622f7e002fd4208c0
a9456bf (HEAD, origin/master, origin/HEAD) HEAD@{3}: merge origin/stable-2.16: Merge made by the 'recursive' strategy.
5d607a4 HEAD@{4}: checkout: moving from 5ff36200b29567118b3aede8e49ba0b6c6b1adb1 to
5d607a4193fb41b4ad8fe01622f7e002fd4208c0
5ff3620 (origin/stable-2.16) HEAD@{5}: merge origin/stable-2.15: Merge made by the 'recursive' strategy.
53333b3 (tag: v2.16.2, tag: v2.16.1) HEAD@{6}: checkout: moving from 5d607a4193fb41b4ad8fe01622f7e002fd4208c0 to
53333b3e3f70dfe14ce4c937246a00a2e4bfa3a0
5d607a4 HEAD@{7}: checkout: moving from 951d84b32e4f2393dbcf7c319e0d3f617838948c to
5d607a4193fb41b4ad8fe01622f7e002fd4208c0
...
```

*`git reflog <branch>`* shows the log for a branch:

- From the log you can see how the branch pointer was changed over time and by which commands.
- It allows you to find commits to which you have lost reference.
- Commits that are referenced by a reflog are not garbage-collected.
- You can also see the reflog for *`HEAD`*:
  *`git reflog`*

# Submodules



***Submodules*** are used to embed ***sub*** repositories into a ***super*** repository:

- For each submodule the `.gitmodules` file in the super repository contains the URL of the sub repository and the local path to where it should be checked out.
- The commits in the super repository contain ***git links*** to a commit in the sub repository.
- The submodule commit is checked out at the local path that is defined in the `.gitmodules` file.

*Q: What are use cases for submodules?*

# Submodules

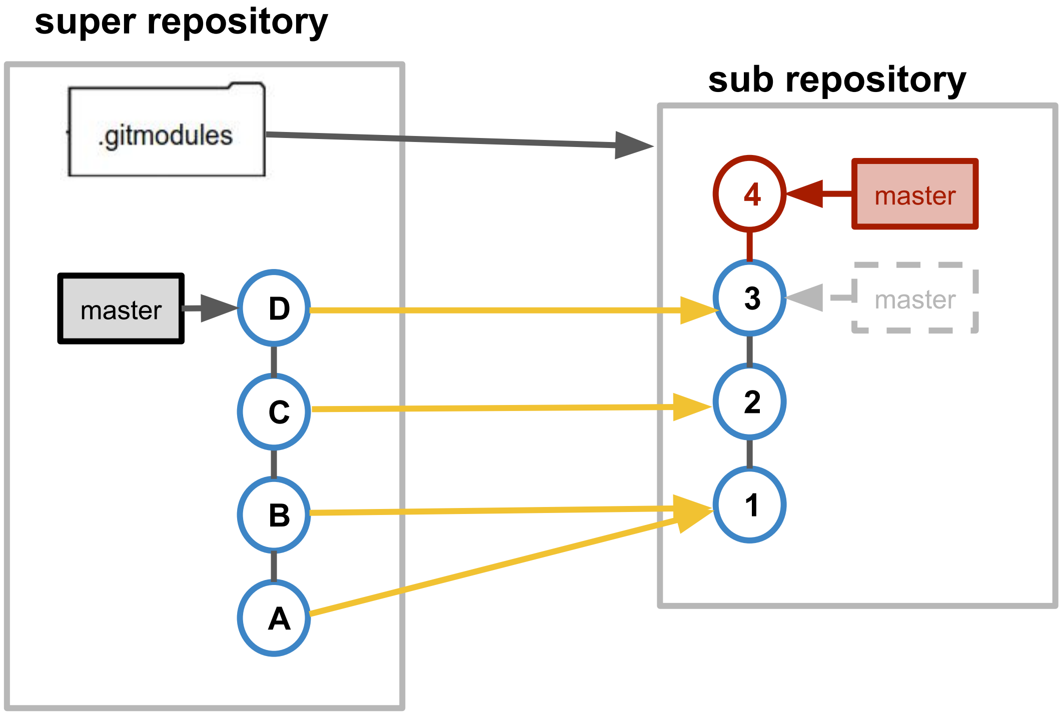
- **Separate code into different repositories:**
  - E.g. create submodules for components of a project.
  - Cleaner Git history since commits are specific to a certain submodule/component.
  - Different maintainers, release cycles etc.
- **A submodule can be added to multiple repositories:**
  - Multiple projects can share the same components.

A ***submodule*** is added by
`git submodule add <URL> <local-path>`

- Checks out the submodule repository at `local-path` (by default its `master` branch).
- Creates/updates the `.gitmodules` file and creates a ***git link*** at local-path.
- The ***git link*** and the `.gitmodules` file should be committed and pushed.
- Users who fetch a commit with a new submodule must initiate the submodule by `git submodule init`

*Q: How is a submodule updated?*

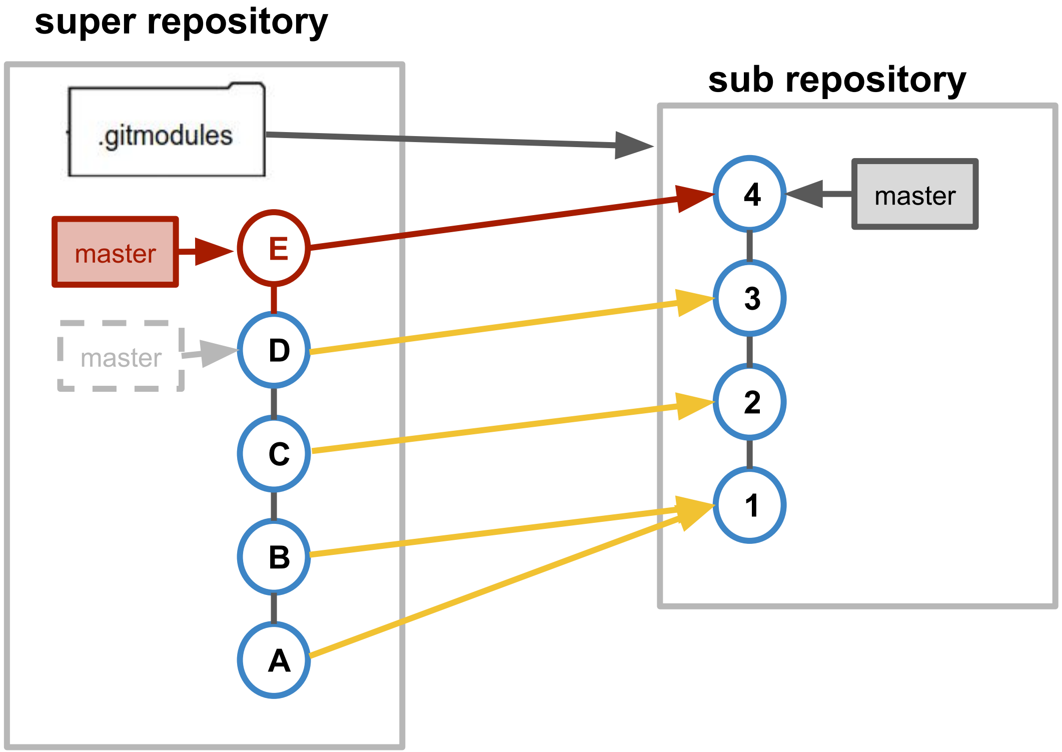
# Submodule Update



***Submodule*** update:

1. Create a new commit in the sub repository.

# Submodule Update



***Submodule*** update:

1. Create a new commit in the sub repository.
2. Update the ***git link*** in the super repository to point to the new commit ***4*** in the sub repository and create a new commit ***E***.
3. Push the changes of both repositories. Users that fetch commit ***E*** in the super repository must update their checked out submodule by *`git submodule update`*

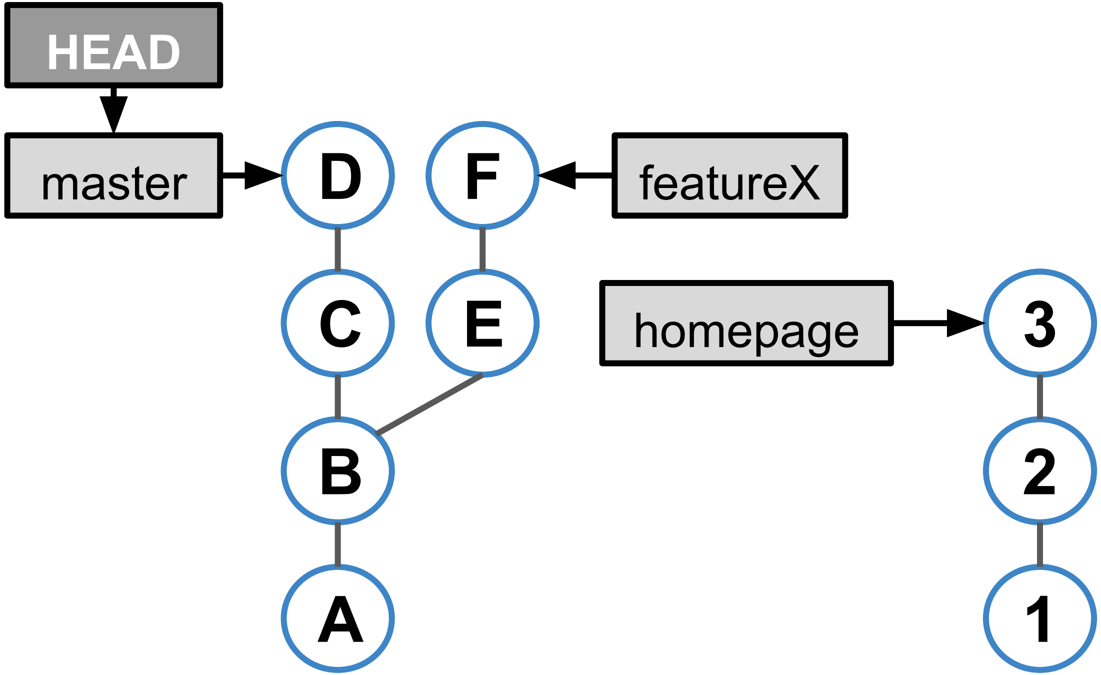
# Submodules

Tips:

- Use `git submodule update --init` to initiate and update your submodules at once
- Submodules may contain submodules themselves. In this case the submodule initiation and update can be done recursively by

  ```
  git submodule update --init --recursive
  ```

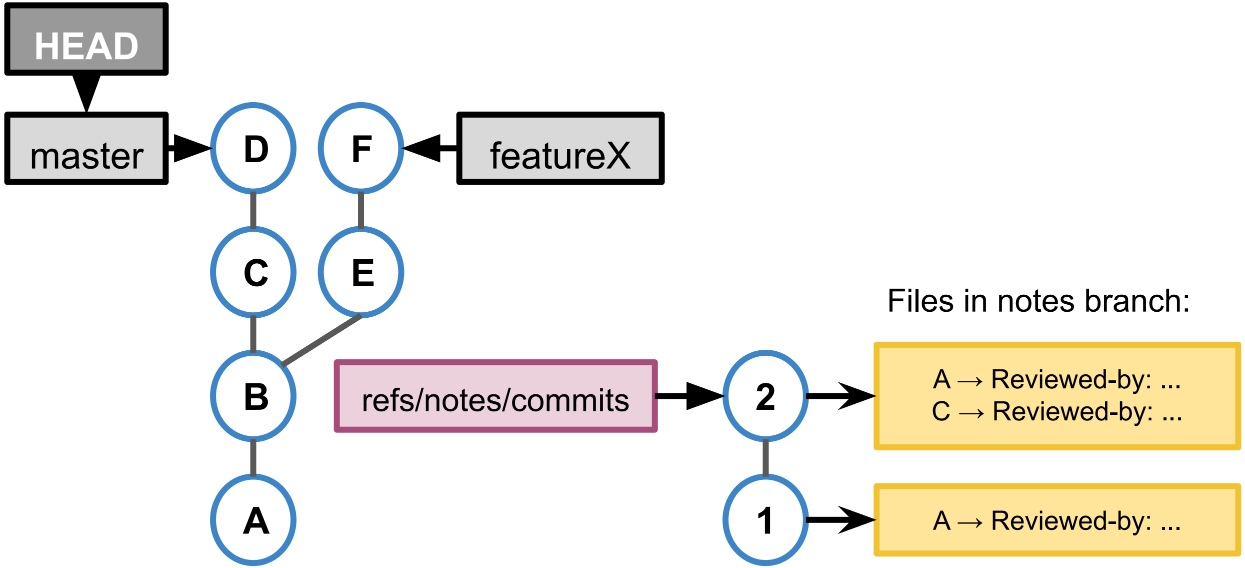
# Branches without common ancestor



Within one repository it is possible to have branches that don't share any common ancestor (e.g. contain a completely different set of files):

- An ***orphan*** branch can be created by `git checkout --orphan <branch-name>`

# Notes



***Notes*** allow to attach metadata to commits:

- can be created/updated without touching the commit (and hence without changing its SHA1)
- are stored in a separate ***notes branch*** (by default `refs/notes/commits`)
- notes branches live in the `refs/notes/` namespace and don't share ancestors with normal branches
- a notes branch contains a list of notes, the file name of a note is the ***SHA1*** of the commit to which the note belongs, the metadata is stored as content in the note file (note that files in a notes branch may be sharded over multiple directories)
- `git notes` allows to list, create and modify notes, but notes branches can also be checkout and updated like any other branch
- you can see notes in the git history by `git log --show-notes=<notes-branch>`

# Git Configuration

List configuration options (overlayed, user global, system wide):

- `git config -l`
- `git config -l --global`
- `git config -l --system`

Set an option (for current repository, user global, system wide):

- `git config user.name “John Doe”`
- `git config --global user.name “John Doe”`
- `git config --system user.name “John Doe”`

Open the config file for edit (for current repository, user global, system wide):

- `git config -e`
- `git config -e --global`
- `git config -e --system`

3 levels of configuration:

- system wide:
  `<git-inst>/etc/gitconfig`
- user global:
  `$HOME/.gitconfig`
- repository specific:
  `.git/config`

# Alias

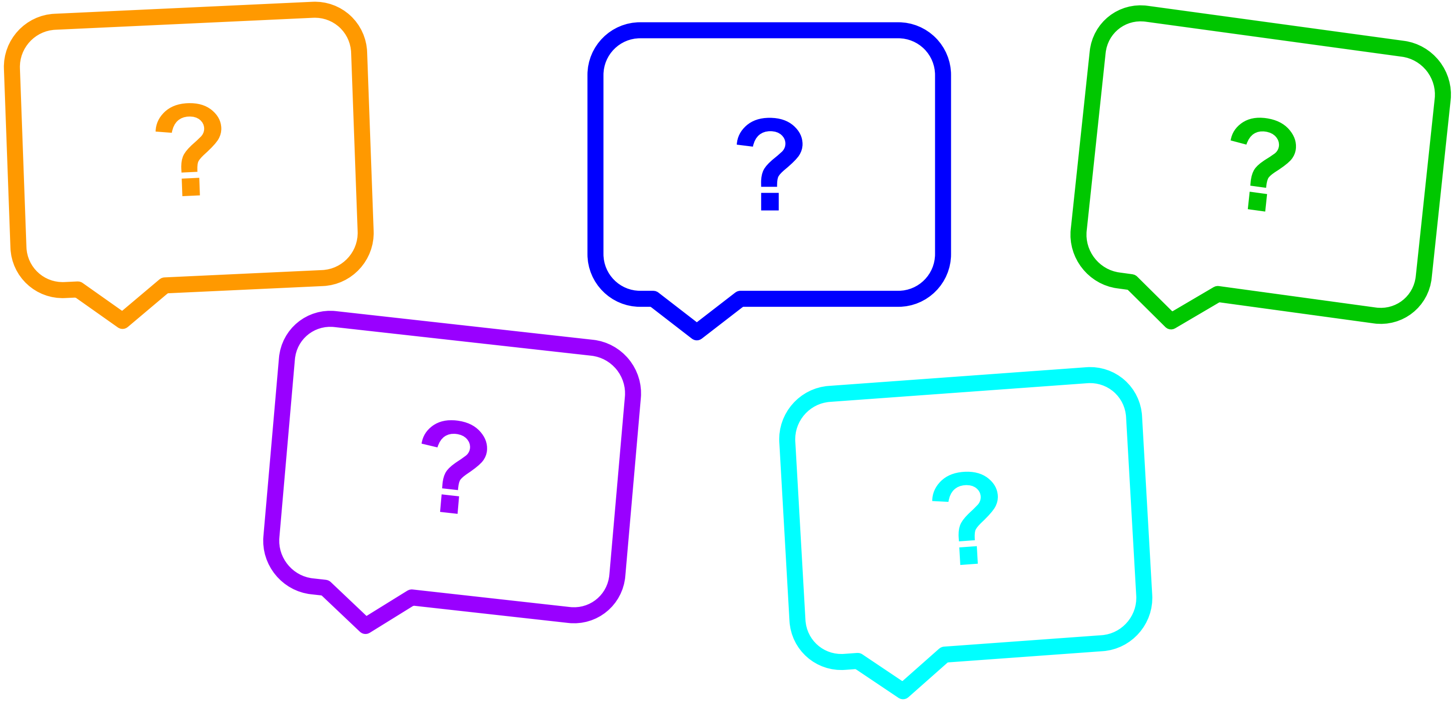
Examples:

1. *`git config --global alias.co checkout`* makes *`git co`* the same as *`git checkout`*
2. *`git config --global alias.unstage 'reset HEAD --'`* makes *`git unstange <file>`* the same as *`git reset HEAD -- <file>`*
3. *`git config --global alias.visual '!gitk'`* makes *`git visual`* the same as *`gitk`*

*`git alias`*

- allows to define own git commands
- ‘*`--`*’ is a bash feature to signify the end of command options, after which only positional parameters are accepted
- use ‘*`!`*’ to invoke external commands

# Thank You - Questions?



# Go Links (for Googlers only)

| *TOPIC* | *GO LINK* |
|---|---|
| Alias | go/git-explained@alias |
| Amend | go/git-explained@amend<br>go/git-explained@commit-amend |
| Bisect | go/git-explained@bisect |
| Blame | go/git-explained@blame |
| Branches | go/git-explained@branches |
| Checkout | go/git-explained@checkout |
| Cherry-Pick | go/git-explained@cherry-pick |
| Clone | go/git-explained@clone |
| Commit History | go/git-explained@commit-history |
| Commit Message | go/git-explained@commit-message |
| Commits | go/git-explained@commits |

| *TOPIC* | *GO LINK* |
|---|---|
| Config | go/git-explained@config |
| Detached HEAD | go/git-explained@detached-HEAD |
| Diff | go/git-explained@diff |
| Differences between Merge and Rebase | go/git-explained@merge-vs-rebase |
| Fast-Forward Merge | go/git-explained@fast-forward,<br>go/git-explained@fast-forward-merge |
| Fetch | go/git-explained@fetch |
| Force Push | go/git-explained@force-push |
| Git Repository Structure | go/git-explained@repo-structure,<br>go/git-explained@repository-structure |
| HEAD | go/git-explained@HEAD |

# Go Links (for Googlers only)

| *TOPIC* | *GO LINK* |
|---|---|
| Interactive Rebase | go/git-explained@rebase-interactive, go/git-explained@interactive-rebase |
| Merge Conflict Resolution | go/git-explained@conflict-resolution, go/git-explained@conflicts, go/git-explained@conflict-resolution-merge |
| Merge | go/git-explained@merge |
| Notes | go/git-explained@notes |
| Orphan Branch | go/git-explained@orphan-branch |
| Pull | go/git-explained@pull |
| Push: Conflict Resolution by Merge | go/git-explained@push-conflict-resolution-merge |
| Push: Conflict Resolution by Rebase | go/git-explained@push-conflict-resolution-rebase |

| *TOPIC* | *GO LINK* |
|---|---|
| Push | go/git-explained@push |
| Rebase Conflict Resolution | go/git-explained@conflict-resolution-rebase |
| Rebase | go/git-explained@rebase |
| Reflog | go/git-explained@reflog |
| Reset | go/git-explained@reset |
| Reset Modes | go/git-explained@reset-modes |
| Revert | go/git-explained@revert |
| Stash | go/git-explained@stash |
| Status | go/git-explained@status |
| Submodule | go/git-explained@submodule |
| Tags | go/git-explained@tags |

# License