# An Introduction To 

David.meredith@stfc.ac.uk

Adrian.coveney@stfc.ac.uk

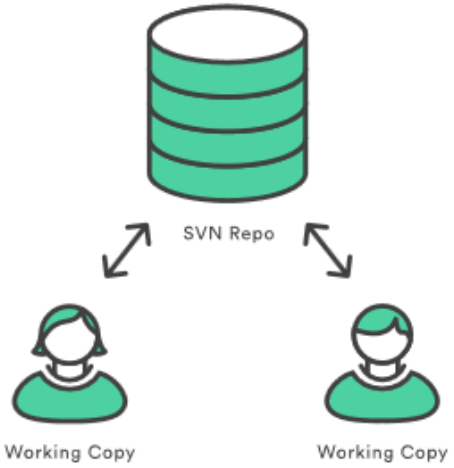
Some favourite Git tag lines

"everything-is-local"
"local-branching-on-the-cheap"
"fast-version-control"
"distributed-is-the-new-centralized"

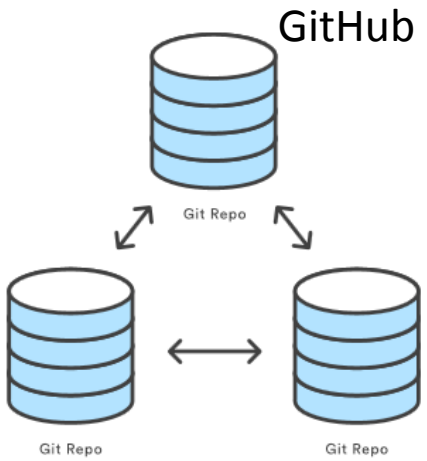
# Git vs Svn

- SVN: Centralised VCS (CentralRepo-to-WorkingCopy)
  - Svn checkout revisions, not a full copy of the repository
- Git: Distributed VCS (Repo-to-Repo)
  - Unlike SVN, Git makes no distinction between the working copy and the central repo, both are all full-fledged Git repositories.
    - No single point of failure
  - Convention is to designate (bless) one repo as central, eg GitHub

SVN

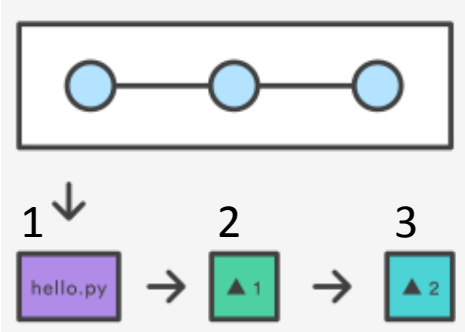



GIT



# Repository Revisions / History

- **Svn commit**: Svn stores file changes as a set of DIFFs

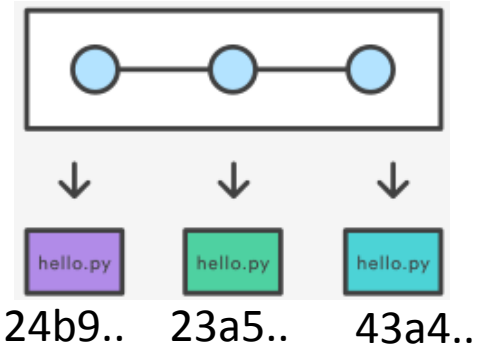



- **Git commit**: Git stores *state of all files in repo* as a FS snapshot
  - For efficiency, if a file is unchanged a pointer is stored to previous version
  - A SHA-1 checksum is calculated over whole snapshot before it is stored:

    '24b9da6552252987aa493b52f8696cd6d3b00373'

    - a) Ensures the integrity of the entire snapshot
    - b) Serves as a unique ID for the commit

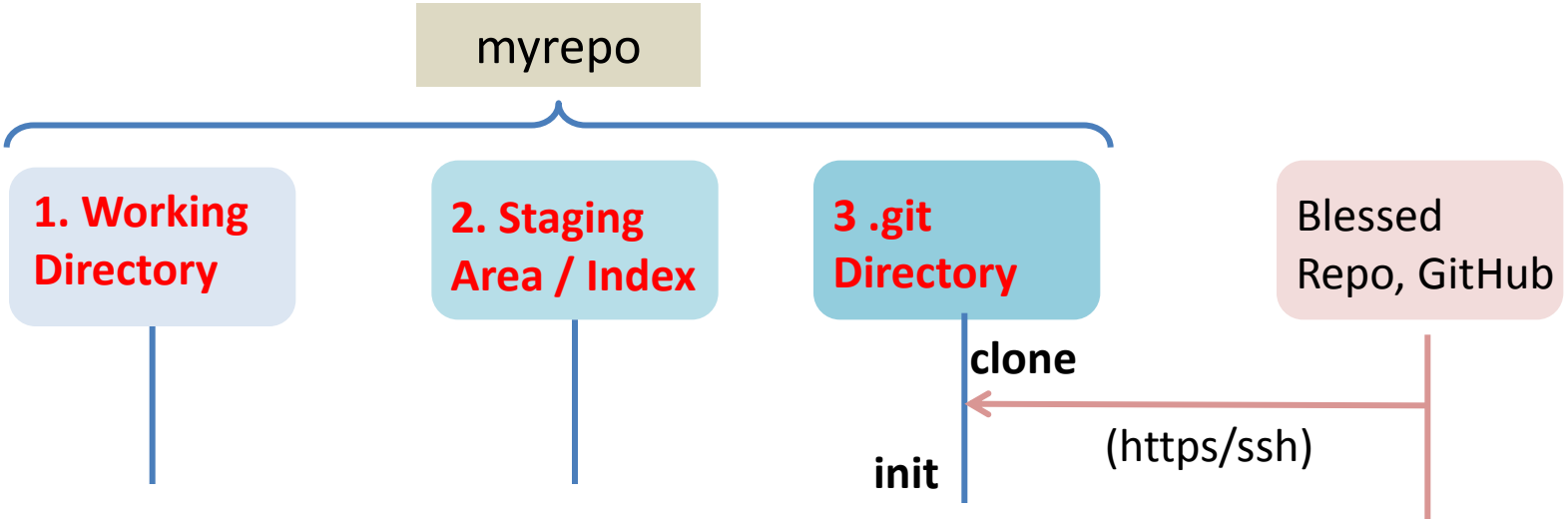
Git operations faster than SVN, as a specific version of a file doesn't have to be "re-assembled" from its diffs.



# Repository Structure

- Establish repo: a) clone an existing repo or b) init new/empty
- Useful to then think of repo as having 3 different file system trees:
  1. Working Directory
  2. Staging Area / Index
     - A buffer between the WorkDir and the .git repository
     - Used to prepare files for your next commit
  3. .git Directory
     - Located in top level repository root directory
     - Holds the internal Git DB (snapshots, Index, checksums…)

No .git sub-dirs unlike .svn



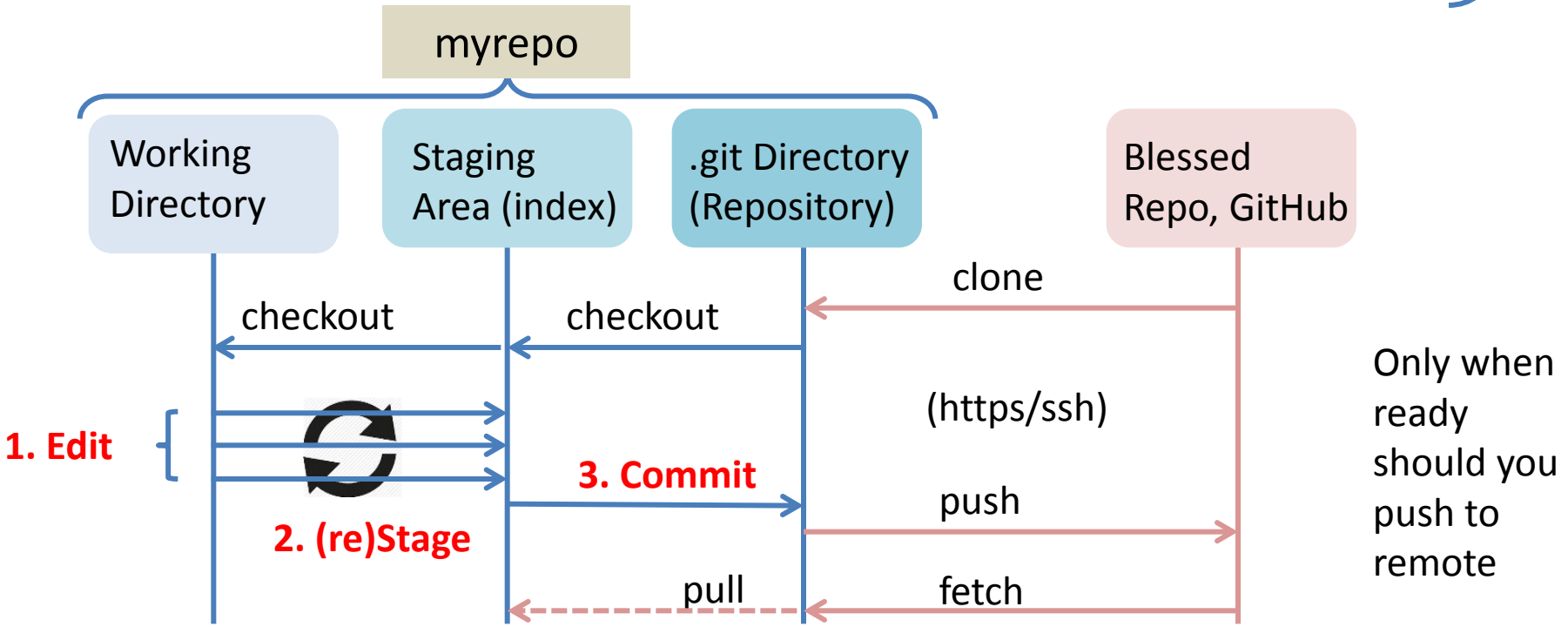
# Basic Workflow (edit, stage, commit)

After repo is established, we 'checkout' a snapshot:

- Index is updated to mirror snapshot and files copied from index into workingDir

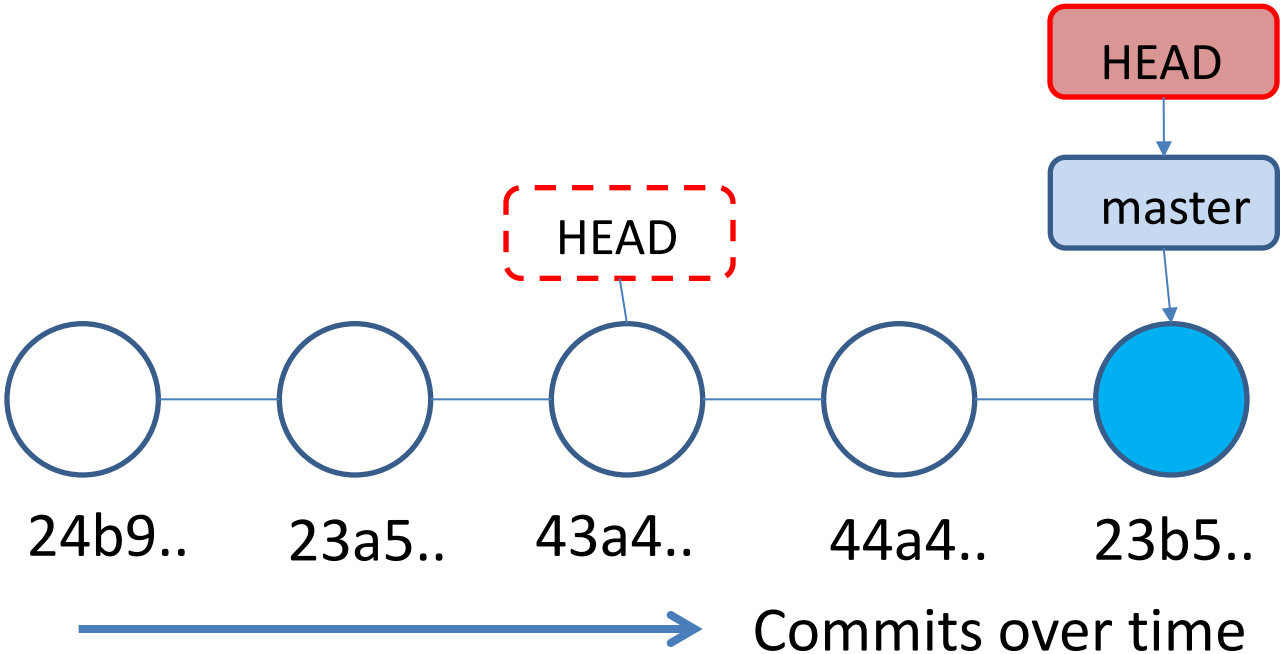
1. Edit local Working Dir files
   - Modify tracked files, create new untracked files, del/rename files
2. (re)Stage modifications into Index to prepare for next commit
3. Commit a set of modifications that are *logically related*

Localhost



# BranchRef and HEAD

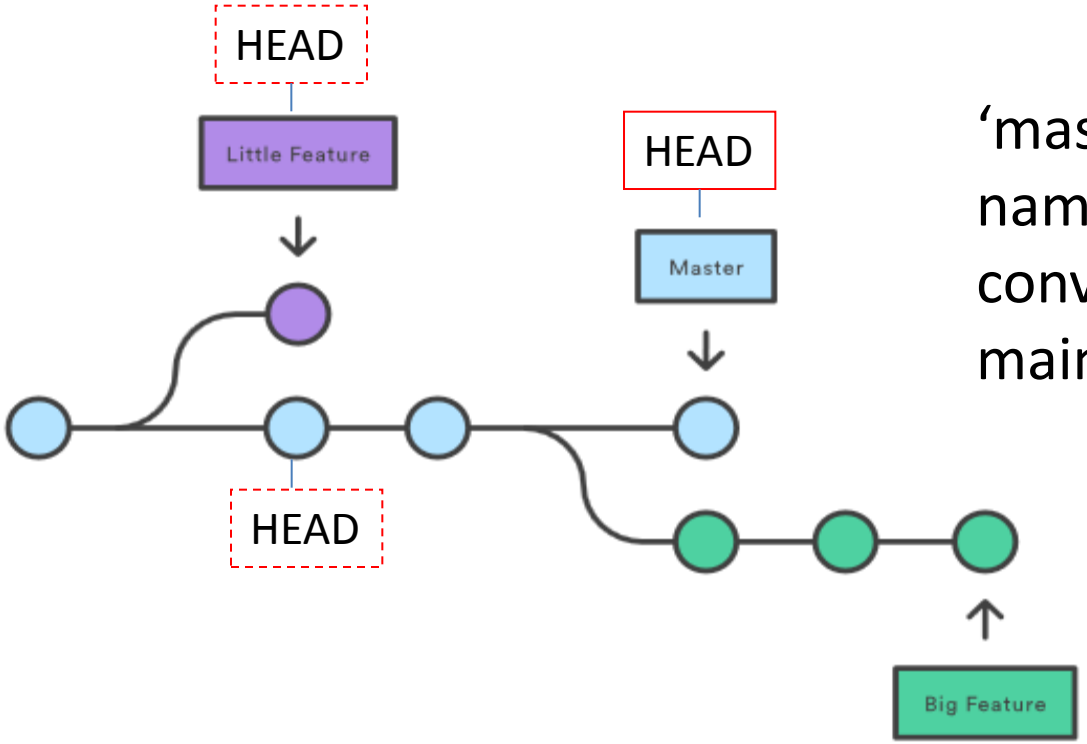
- BranchRef: a pointer to a named branch e.g. ‘master’
  - BranchRef usually points to the last commit on the branch
- HEAD: a pointer to our current position
- HEAD is moved using ‘git checkout’ so that it points to:
  - A branchRef or a historical commit (detached-head-state)




Demos 1+2

# Branching

- (Topic)Branch: represents an independent line of development for the edit/stage/commit process
- Purpose: to isolate the changes to a branch without affecting other branches (‘myFix,’ ‘dev,’ ‘preRelease’)
- Branch early, branch often (cheap to create branches)
- Only when ready, should branches be combined




‘master’ is just naming convention for our main branch

# Combining Branches

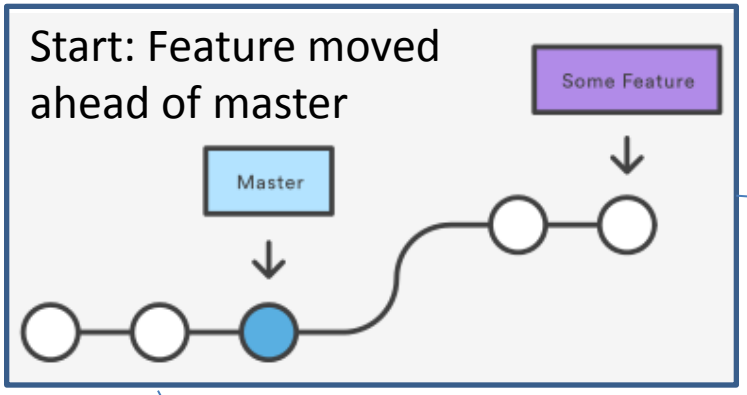
2 main ways to combine the changes from different branches:

1. Merging – Merge changes between branches and keep branch history
2. Rebasing – Move a branch to the tip of another branch to flatten branch history

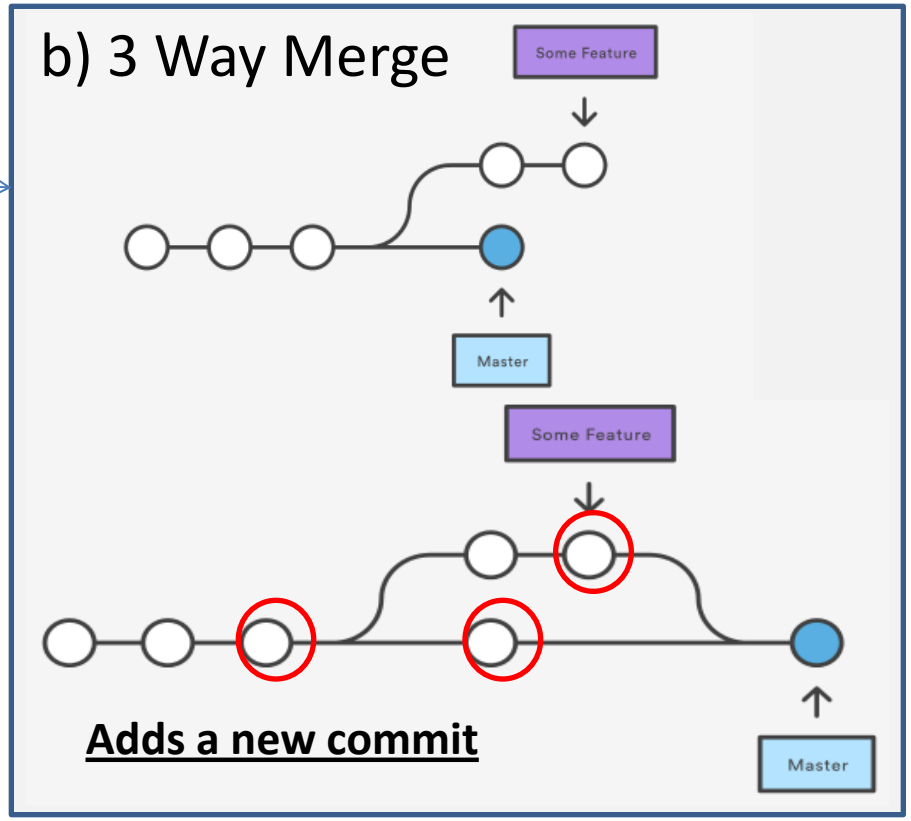
# Merging

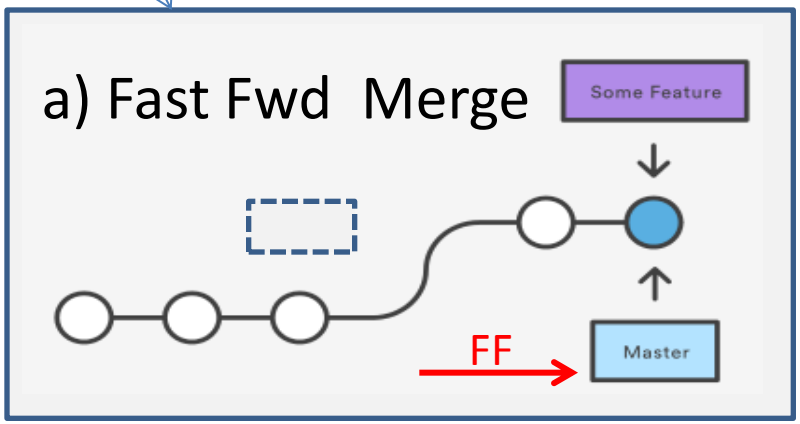
- Merge combines changes from specified branch **into** current branch
- 2 types of merge depending on branch history
  - A) Fast Forward Merge
  - B) 3 Way Merge

```
git checkout master
git merge Feature
```
(merges Feature branch **into** master branch)




**+ Fork** (master also moved)



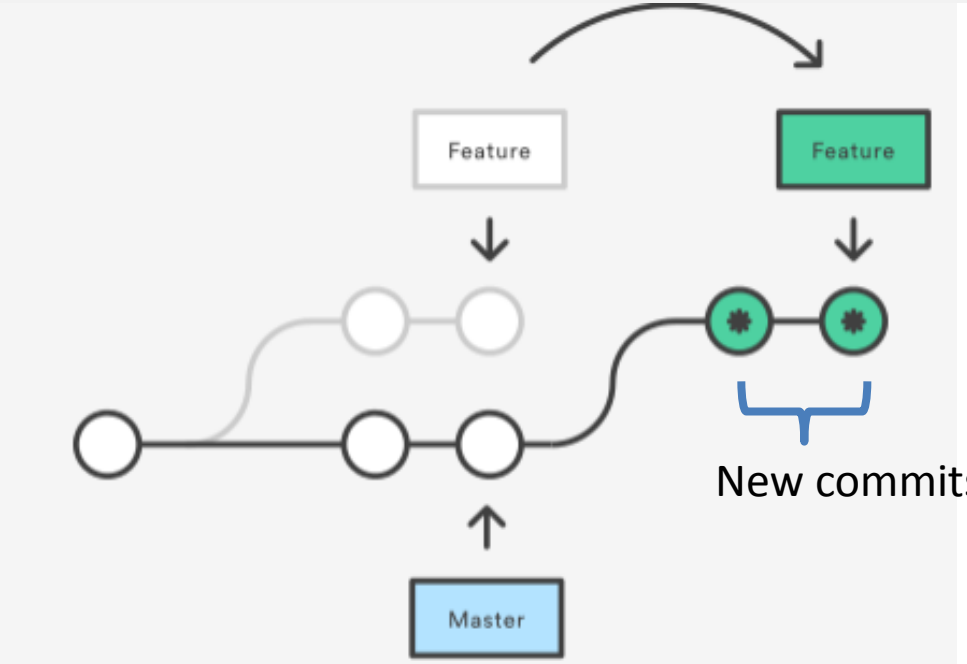




Demos 3+4

# Rebasing

- Moves a branch onto the end of another branch by adding a series of patches to reproduce the changes
- This flattens a forked branch
  - Produces a simple linear history
  - Often to clean-up several local topic branches before pushing to a remote

```
git checkout Feature
git rebase master # rebase onto master
```




Note, don't rebase branch that has already been pushed to a public repo – you need to preserve shared branch history.

Demo 5

# Dealing with Merge Conflicts

- When merging/rebasing, conflicts are possible if there are changes to the **same lines** in two different versions of the **same file**
- Git adds standard conflict markers to the affected file(s) for you to manually resolve

  <<<<, =====, >>>>>
- After fixing conflict markers, you need to restage affected files and re-commit to resolve conflict

Demo 6

# Stashing

- Consider following common scenario:
  - Working on ‘dev’ branch with local changes,
  - A bug fix is reported and you need to immediately switch (checkout) to ‘master’ branch to apply a fix

- If a **file has diverged between two branches**, and you have **local modifications in same file**, checkout is not allowed (git prevents clobbering local changes), options:
  - Commit changes on dev (but may not by ready yet!) ☹
  - Stash changes ☺

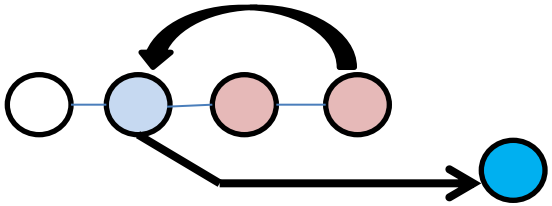
- Stashing allows you to checkout a diverged branch and come back to stashed changes later on

Demo 7

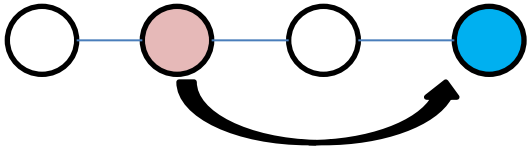
# Undoing History

2 main ways to undo the changes of former commits to undo mistakes

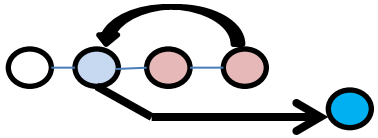
1. Reset – rollback over a full sequence of commits



2. Revert - undoes a specified commit(s)

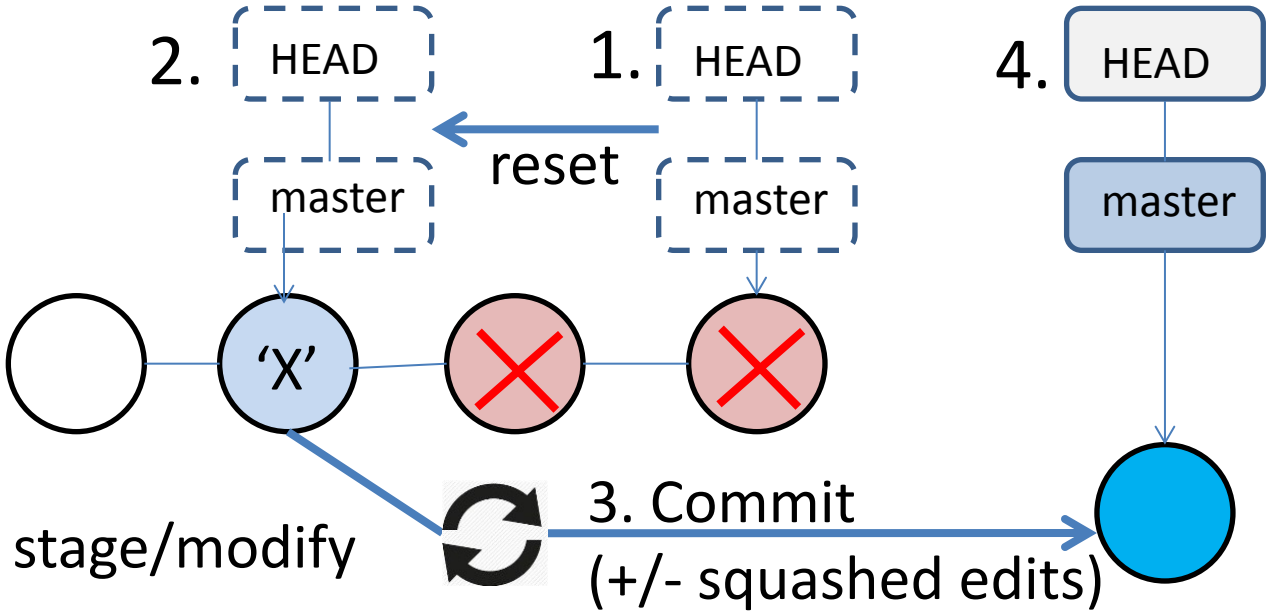


# Reset



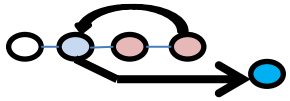
- Undoes a full **sequence of commits**
- Moves **branchRef** + **HEAD** back to commit ‘X’
- Optionally *squash* all of the changes of the reset commits into the Index/WorkDir to edit before next commit:

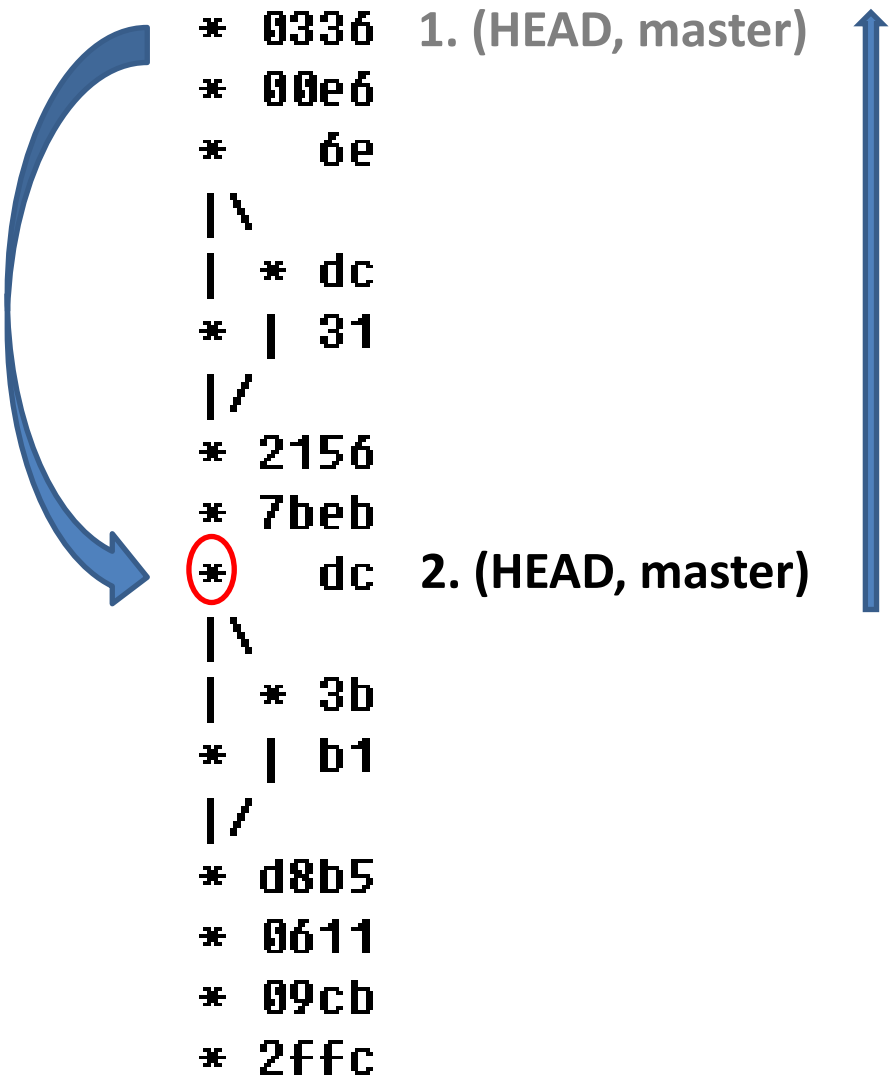
| | | Rolled back |
|---|---|---|
| --soft | **BranchRef** + **HEAD** (not Index/WrkDir) | => changes remain in Index+WrkDir |
| --mixed | **BranchRef** + **HEAD** + **Index** (not WrkDir) | => changes remain in WrkDir |
| --hard | **BranchRef** + **HEAD** + **Index** + **WorkDir** | = > no changes remain ⚠ |



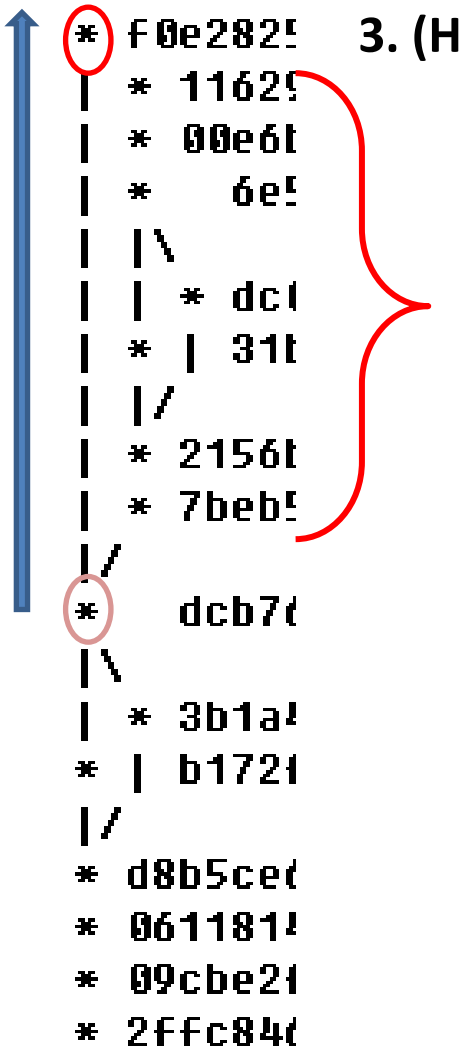

Never reset commits that have been shared remotely (reset local commits only)

# Reset






git reset <dc..>

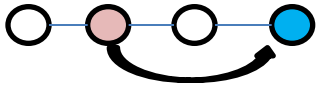



Reset/rolled-back commit sequence is preserved

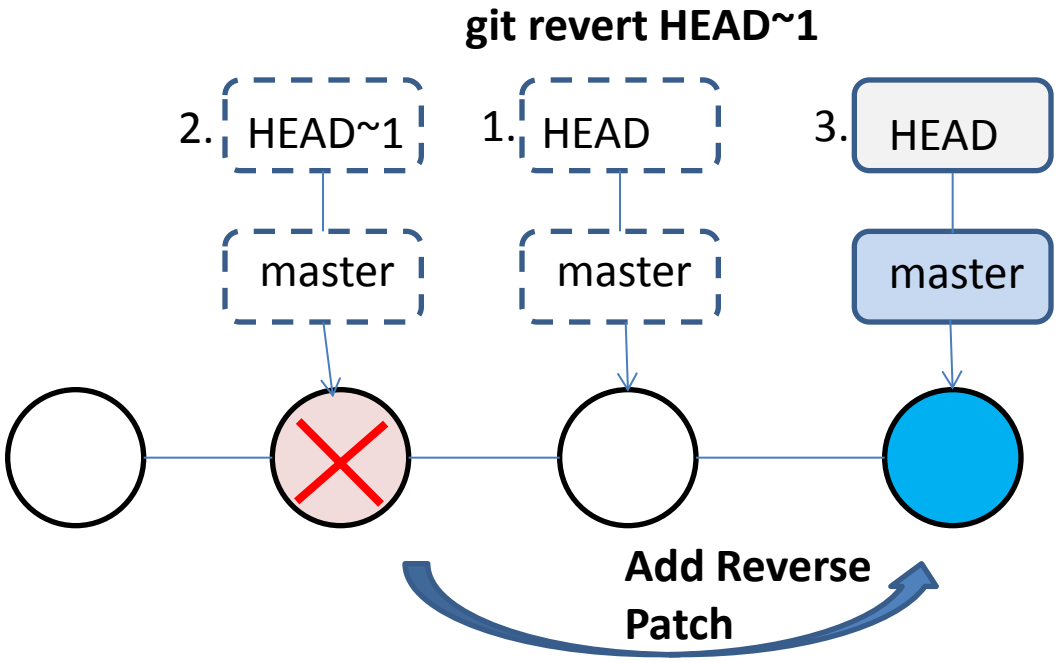
git commit

Demo 8

# Revert



- Undoes all of the changes introduced by **specified commit**(s) by adding a new commit to the branch tip that applies a set of **reverse patches**
- Only undoes the specified commit(s), it **does not rollback** a full sequence of commits like reset does
- E.g. a bug was introduced by a specific commit, we can revert just that commit to remove its changes




Demo 9

# Working With Remote Repos (Setup)

Optional: First create a remote repo on e.g. GitHub / BitBucket (usually via web interface)

Two ways to work with a remote repo:

1. For existing remote repo: Clone the repo

```
git clone https://github.com/owner/repoName.git
```

2. For existing local repo: Add a reference to a remote repo and push local branches up to the remote:

```
git remote add origin https://github.com/owner/repoName.git
git push –u origin <localBranch>
```

# Working With Remote Repos
## (local and remote branches)

- 2 types of branch in your local repo:
  - **Local** branch e.g. *master*, *dev*
  - **Remote** branch that are prefixed with name of remote e.g. *origin/master*, *origin/dev*
    - 'origin' is default tag name for a remote repo's URL
    - Read only, can't directly edit/commit a remote branch
  - Important: Local + Remote branches are **distinct**:

```
[gocdb]$git branch -a
  dev
  master
* rolelog
  remotes/origin/dev
  remotes/origin/master
  remotes/origin/rolelog
```

Local branches

Remote branches

# Workflow: fetch, merge, (edit,stage,cmt), push

1. **Fetch** remote branch into local-repo

```
git fetch <remoteRepoName> <remoteBranch>
```

2. **Merge** changes in remote branch into local branch

```
git checkout master
git merge origin/master
```

Same as merging between 2 local branches

Local branch edit, stage, commit

3. **Push** local changes to remote branch

```
git push [-u] <remoteRepoName> <remoteBranch>
```

Can assign local branch as a remote-tracking-branch with [-u]

Simplify steps 1+2 via **git pull**
(automatic merging into local B)

```
[gocdb]$git branch -vv
  dev      6976f08 [origin/dev: ahe
  master   62462eb [origin/master]
* rolelog d3dc85f [origin/rolelog:
```

# Create a new personal Repo in GitHUB and then push to it

Create new empty repo via portal + copy repo URL

GitHub then tells us how to create a new repo and push from the command line


Demo 10

# Resources

- http://git-scm.com/book/en/v2
- https://www.atlassian.com/git/
- http://pcottle.github.io/learnGitBranching/