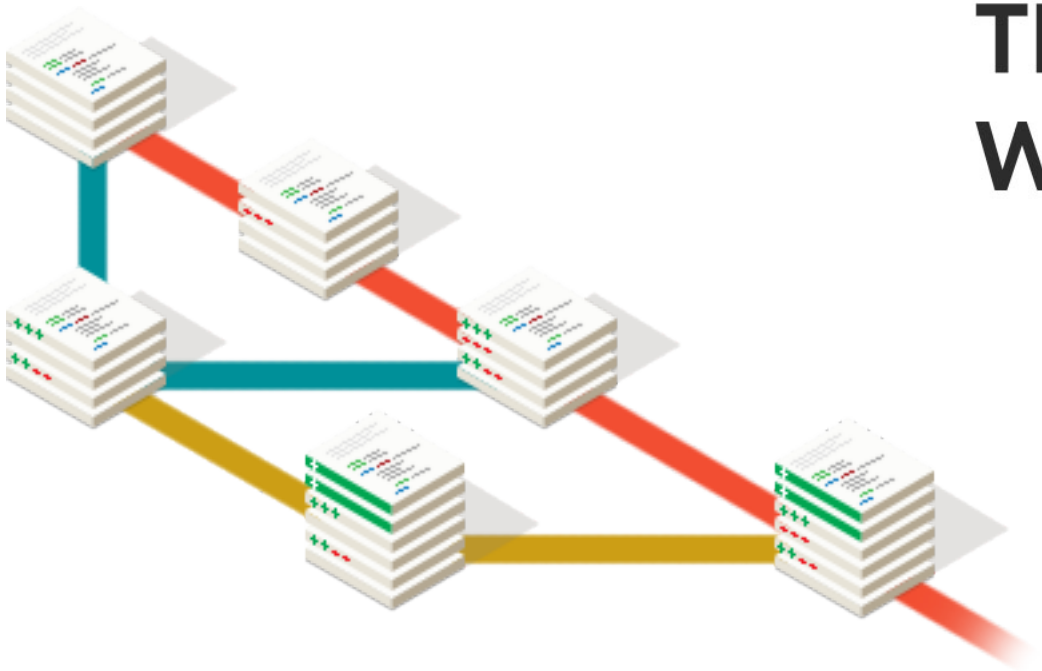
THINKING WITHYOU



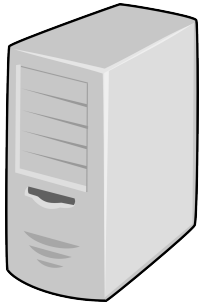
Git from zero to hero

First of all…

Clear your mind

What is a VCS?

Programmer



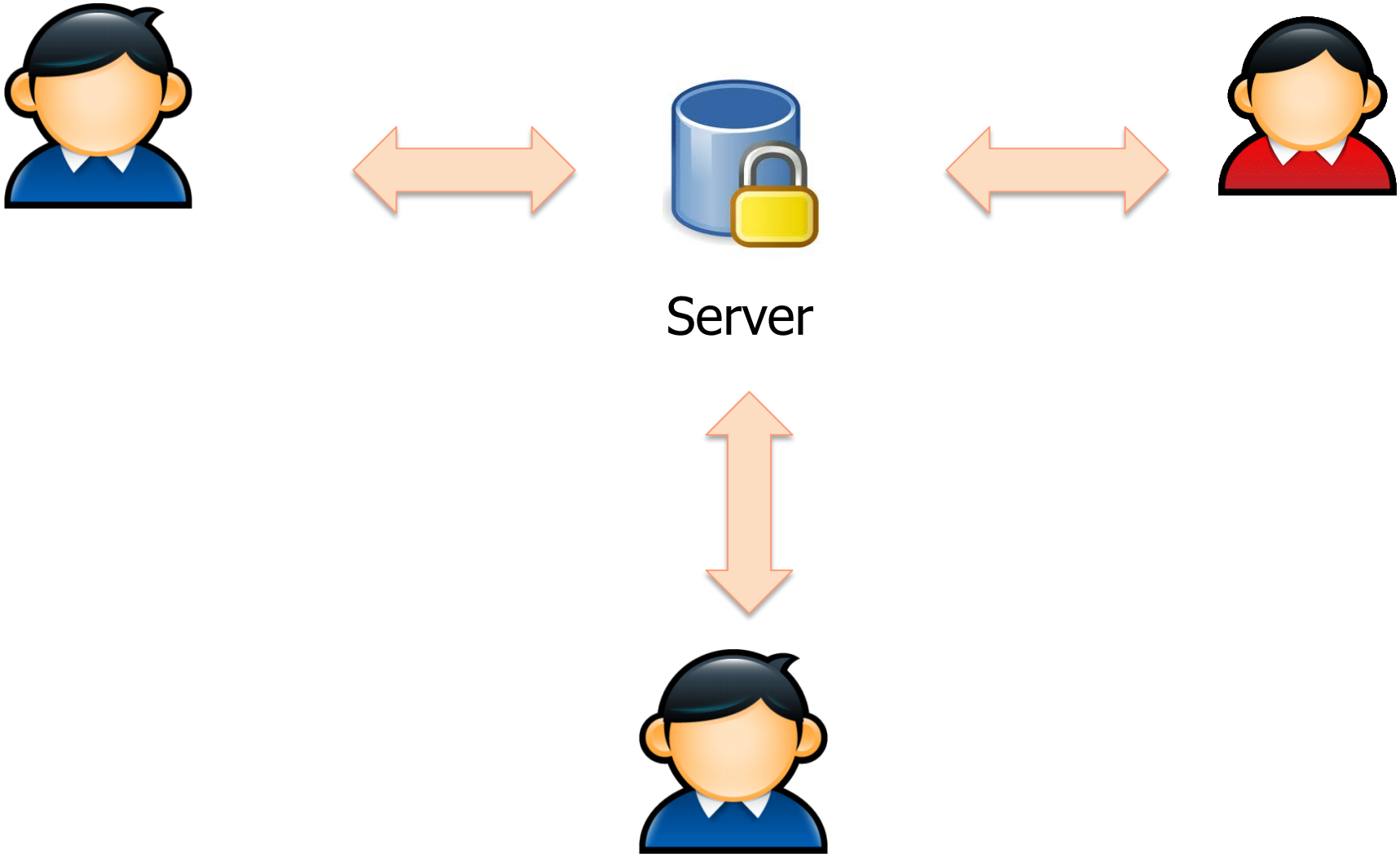
Server: Runs applications

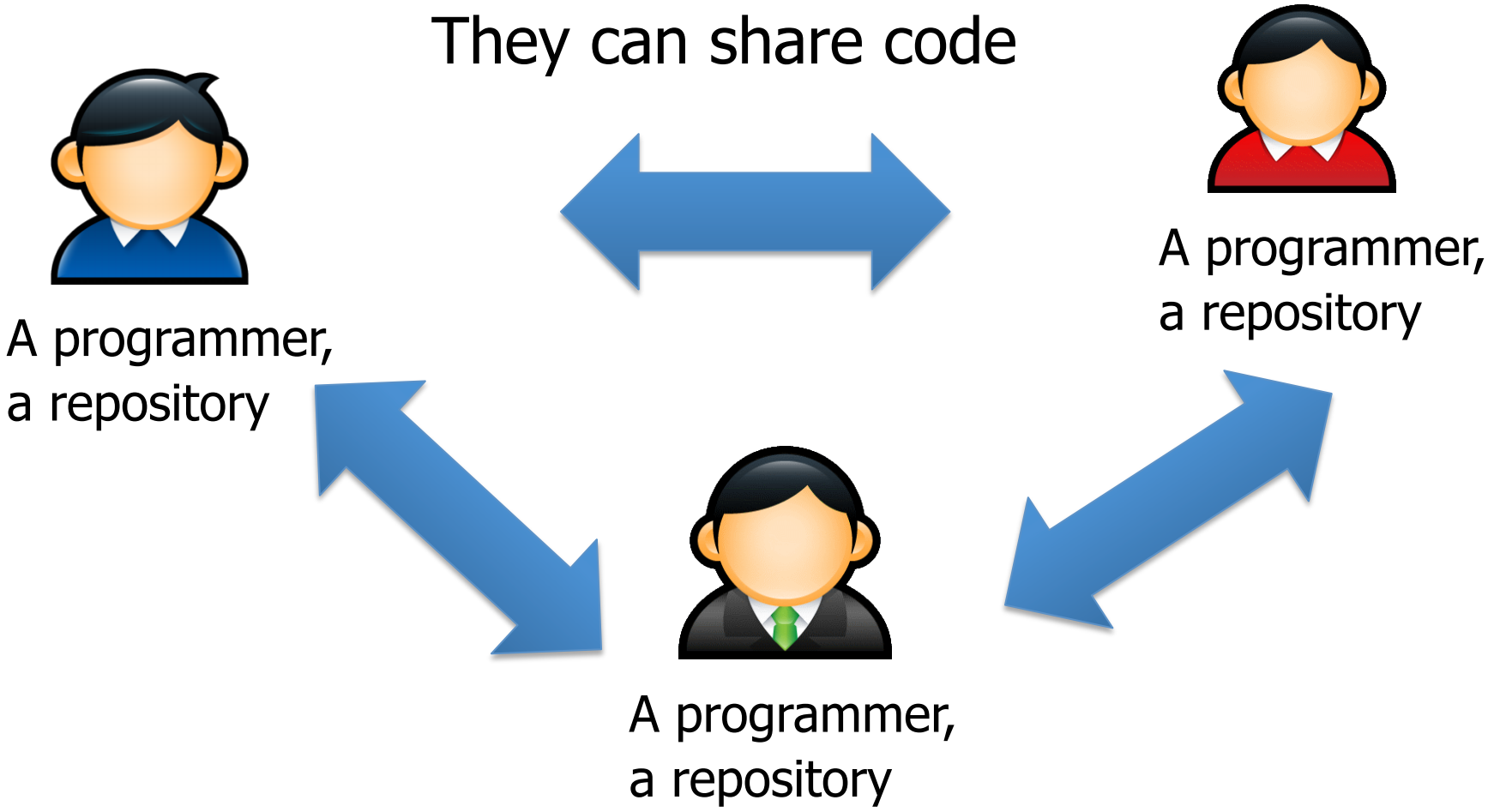
DB: Stores the data

VCS: Stores the code

Automatization of deployment

git (Central)VCS





Everyone has a local repository

"CVS and SVN are remote backups that you use to save your changes.

Git is an editor that you use to write your code's biography."

— isaacs (@izs) May 14, 2010

THINKING
WITHYOU

GIT BASIS

git History

1972: Source Code Control System (SCCS): Closed source, free with Unix

1982: Revision Control System (RCS) : Cross platform. Open Source. More features and faster.

They only could work with one file at the same time

1986-1990: Concurrent Versions System (CVS): Concurrent users working on the same file. Open Source.

2000: Apache Subversion. Snapshot of the directory not just the file. Transactional commits.

2000: BitKeeper SCM. Closed source, proprietary. Distributed version control.

The community version was free. Used for source code of the Linux kernel from 2002-2005

April 2005: The community version not free anymore

Git origin

It was born in 2005, developed by the kernel linux team, leaded by Linus Torvalds.

Tries to improve BitKeeper.

Git: Goals

- Fast
- Simplicity
- Multi Branches
- Distributed
- Able to manage big projects

Local operations

Every operation is done in your local repository.

You don’t need connection with some other server.

You can work offline without problems

Faster than centralized repositories

Integrity

Everything is identified by hashes.

It is impossible to change the content of any file or directory without Git knowing about it.

Can’t lose information or get file corruption without Git being able to detect it.

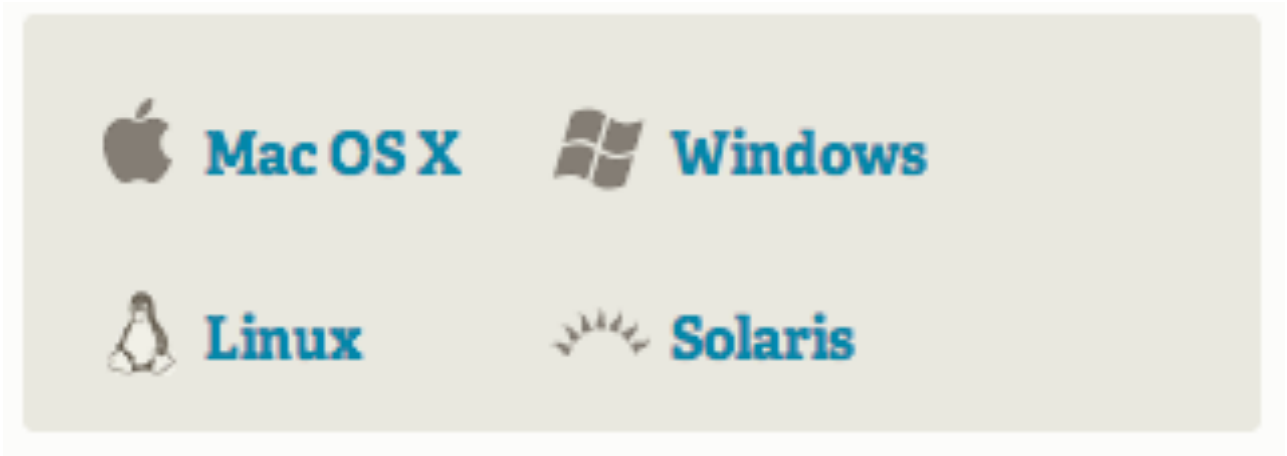
Checksum (sha1):

24b9da6552252987aa493b52f8696cd6d3b00373

Installation

You can install your Git repo here:

http://git-scm.com/downloads



Set Up Git

It’s time to configure your settings:

Username

```
$ git config --global user.name "Your Name Here"
# Sets the default name for git to use when you commit
```

Email

```
$ git config --global user.email "your_email@example.com"
# Sets the default email for git to use when you commit
```

Adding .gitignore

A gitignore file specifies intentionally untracked files that git should ignore.

Files already tracked by git are not affected.

Located in the root folder.

Adding .gitignore

Patterns

#: Comments

!: Negates the pattern

foo/: Directory named foo (not a file foo)

*: Everything

.html , !foo.html, /.js

Create your first repository

1) Create a specific folder where your repository is going to be created.

2) $ git init

```
israelalcazar@Mac-Lamaro-2:~ $ git init
Reinitialized existing Git repository in /Users/israelalcazar/.git/
```

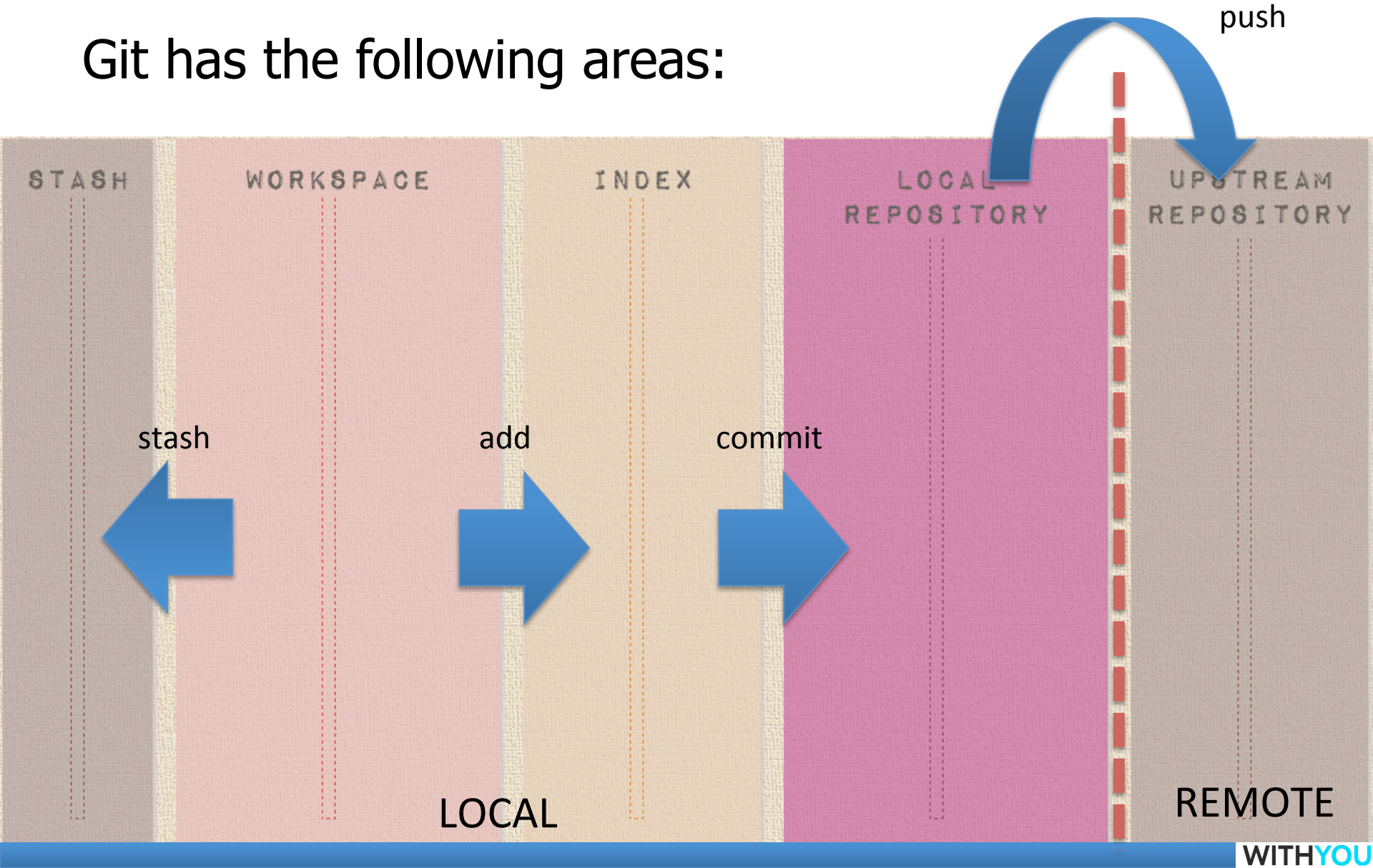
Hands on: Init a repo with .gitignore

THINKING WITHYOU

GIT INTERNALS (I)

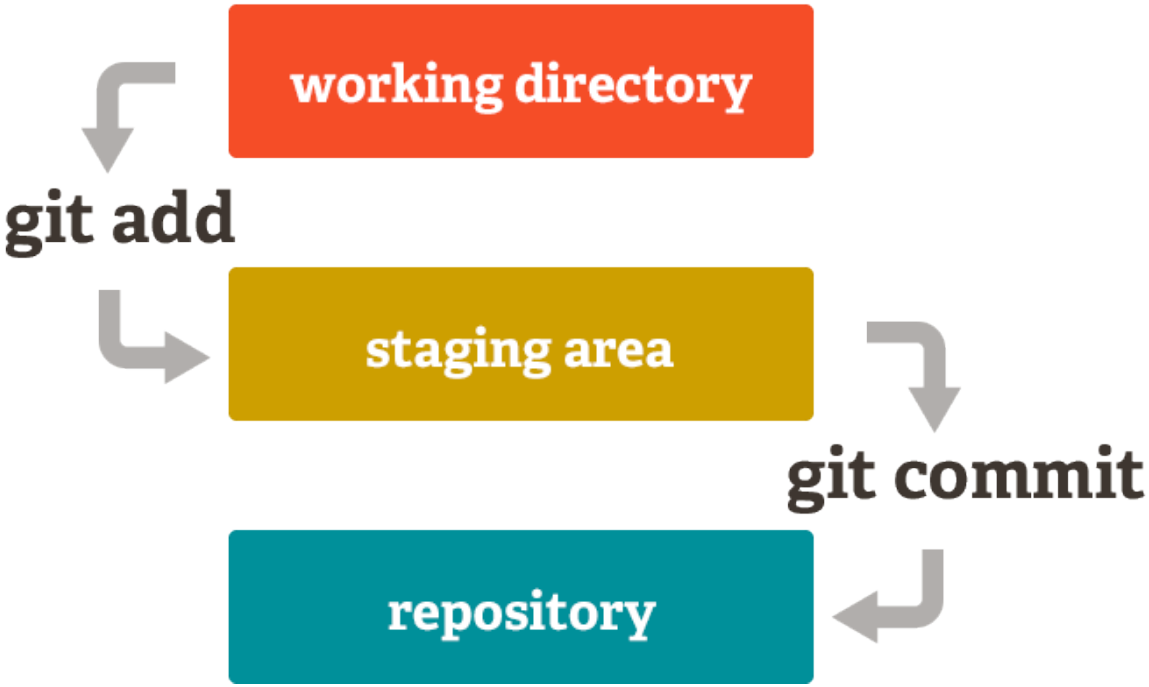
Internal Areas

Git has the following areas:



Staging Area / Index

Intermediate zone where next commit is prepared.



1. Adding a File to Your Repository

- One file

```
$ git add <filename>

$ git add *.c
$ git add index.html
```

- A directory

```
$ git add <directory>
```

2. Commit One file

```
$ git commit –m “message”
```

If the file is managed by the repo:

```
$ git commit –am “message”
```

3. Remove a file from the staging area

#Before first commit

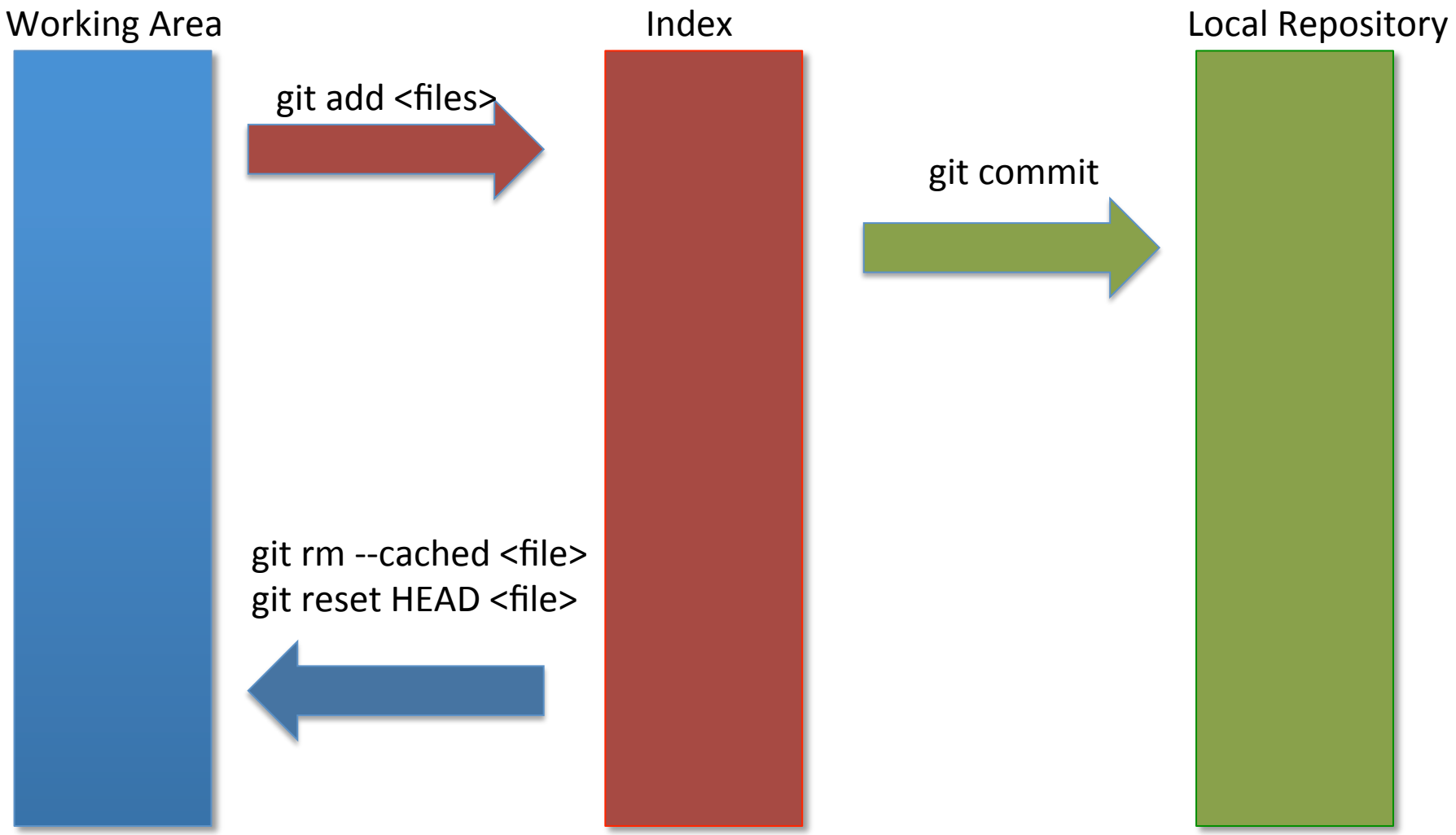
$ git rm --cached <file>

#After first commit

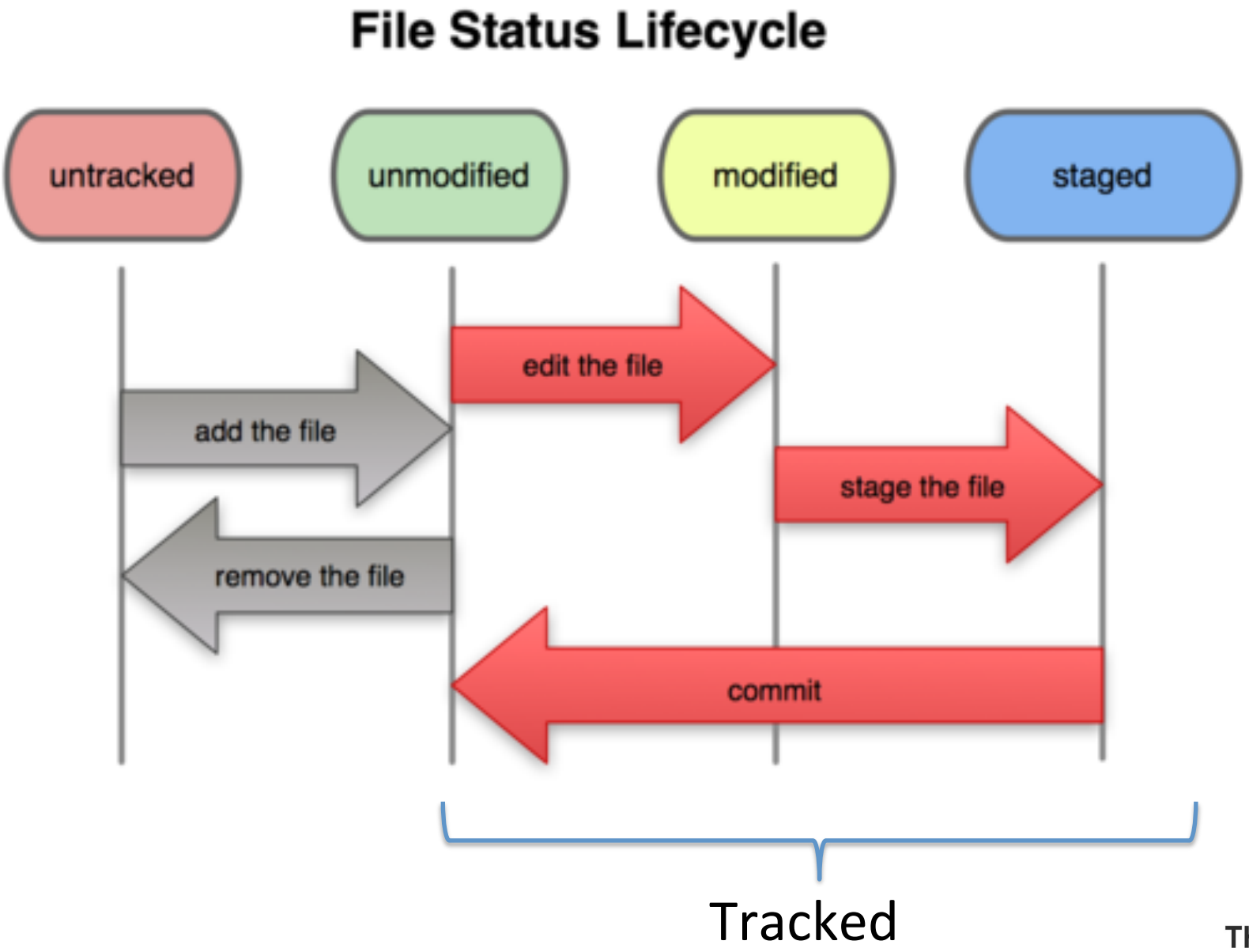
$ git reset HEAD <files>

Hands on: Basic Flow

Basic Flow: First Commit



File Status Lifecycle



File Status Lifecycle

Each file in your working directory can be in one of two states: tracked or untracked.

Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.

Untracked files are everything else.

git Status

You can check your repo status:

```
$ git status
```

THINKING
WITHYOU

STASHING

Description

An intermediate area where you can commit unstable code that you can not deal with.

Use stash when you want to record the current state of the working directory and the index, but want to go back to a clean working directory.

git stashing

A great way to pause what you are currently working on and come back to it later.

$ git stash: Stash your changes away

$ git stash apply: Bring work back

E.g: git stash apply stash@{1}

$ git stash list: List of stashes

$ git stash pop: apply the top stash

$ git stash drop <id>: Manually delete stashes

$ git stash clear: Delete al of the stored stashes.

git Commands

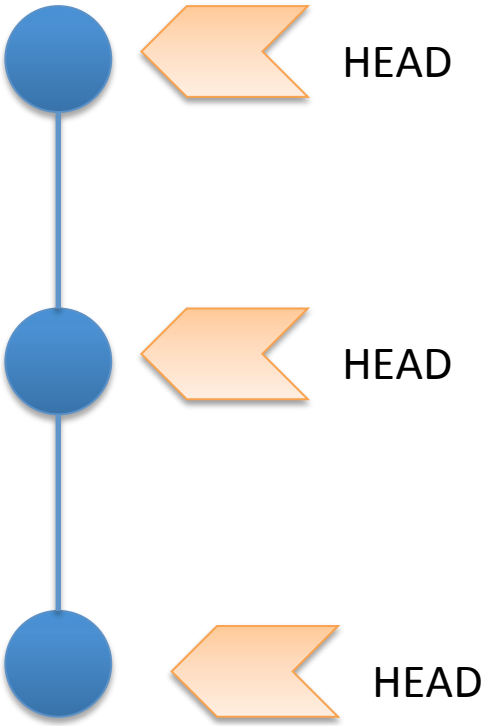
- git stash list
- git stash save "mensaje”
- git stash pop
- git stash show stash@{0}
- git stash apply stash@{0}
- git diff stash@{0}

Hands on: Stashing

THINKING WITHYOU

GIT INTERNALS (II)

HEAD



git HEAD

HEAD is a pointer that points to the current branch.

When you change to a different branch, HEAD changes to point to the new one.

The current HEAD is **local to each repository**, and is therefore **individual for each developer.**

Hands-on

```
$ cd .git
$ cat HEAD
        ref: refs/heads/master

$ git checkout –b newFeature
$ cat HEAD
        ref: refs/heads/newFeature
```

Double Dash --

You can use -- to:

- Separate options from a list of arguments:

```
$ git diff –w master origin -- tools/Makefile
```

- Separate an explicitly identify filenames

```
# Checkout the tag named “main.c”
$ git checkout main.c

#Checkout the file named “main.c”
$ git checkout -- main.c
```

THINKING WITHYOU

REFS AND SYMREFS

git Ref

- A ref is a SHA1 hash ID that refers to an object within the Git object store. Although a ref may refer to any Git object, it usually refers to a commit object.

- A symbolic reference , or symref , is a name that indirectly points to a Git object. It is still just a ref.

- Local topic branch names, remote tracking branch names, and tag names are all refs.

Symrefs

HEAD

ORIG_HEAD

HEAD

Always refers to the most recent commit on the current branch.

When you change branches, HEAD is updated to refer to the new branch's latest commit.

ORIG_HEAD

Certain operations, such as merge and reset, record the previous version of `HEAD` in `ORIG_HEAD` just prior to adjusting it to a new value.

You can use `ORIG_HEAD` to recover or revert to the previous state or to make a comparison.

THINKING
WITHYOU

COMMITS

What is a commit

A commit is used to record changes to a repository.

A commit contains a changeset of files:

What is a commit

A commit is the only method o introducing changes to a repository.

Commits are often introduced explicitly by a developer but Git can introduce commits (as in a merge).

Format

```
$ git commit -m “Message”

$ git commit -am “Message”

$ git commit -m “Message” <file>

$ git commit --amend
```

THINKING WITHYOU

Atomic changesets

Every Git commit represents a single, atomic changeset with respect to the previous state.

Every changes apply in a commit or none (atomicity).

You can be assured that Git has not left your repository in some transitory state between one commit snapshot and the next.

Identifying Commits

Every Git commit is a hex-digit SHA1 ID.

Each commit ID is globally unique—not just for one repository, but for any and all repositories

Relative Commit Names

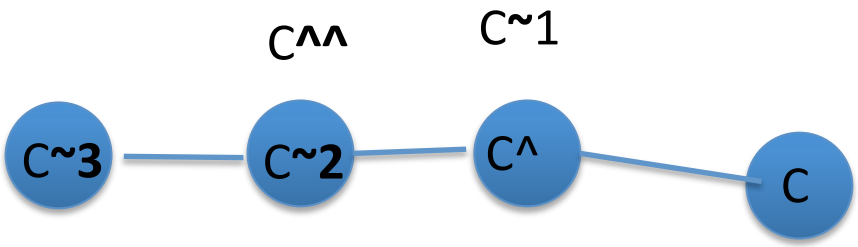
Git provides mechanism for identifying a commit relative to another reference:

^: One commit before (master^, master^^, HEAD^,etc)

~: N commits before (master~2, HEAD~4)

Relative Commit Names

C is a commit



git show-branch --more=35

Commit history

You can see the history of commits:

$ git log (same as git log HEAD)

$ git log <commit-name>: The log starts at the named commit.

E.g: $ git log master

http://www.kernel.org/pub/software/scm/git/docs/v1.5.0.2/git-log.html

Specify a commit range (since..until)

$ git log master~12..master~10

Show only commits for a path

$ git log -- <path>

E.g. git log -- README3.txt

Show commits with a regular expression

$ git log --grep=‘reg-exp’

--no-merges: Removes merge commits

Show changes during a time

$ git log --since={2010-04-18}
$ git log --before={2010-04-18}
$ git log --after={2010-04-18}

Show the graph

--decorate --graph --oneline

```
*   17e4c5f (HEAD, develop) Merge branch 'hotfix-1.0.1' into develop
|\
| * f43bb33 (hotfix-1.0.1) Hotfix2
| * 50a102d Hotifx1
| * d2fe7d6 Version 1.0.1
| *   d534111 (tag: 1.0) Merge branch 'release-1.0'
| |\
* | \   77e7eac Merge branch 'release-1.0' into develop
|\ \ \
| | |/
| |/|
| * | 9261fad (release-1.0) Release10Fix2
| * | 2562cb3 Release10Fix1
| * | c51b802 Version 1.0
* | | fbdd638 Work6
* | | 7353285 Work5
|/ /
* | 538bb9a Work4
* |   cdef254 Merge branch 'feature1' into develop
```

THINKING
WITHYOU

MANAGING BRANCHES

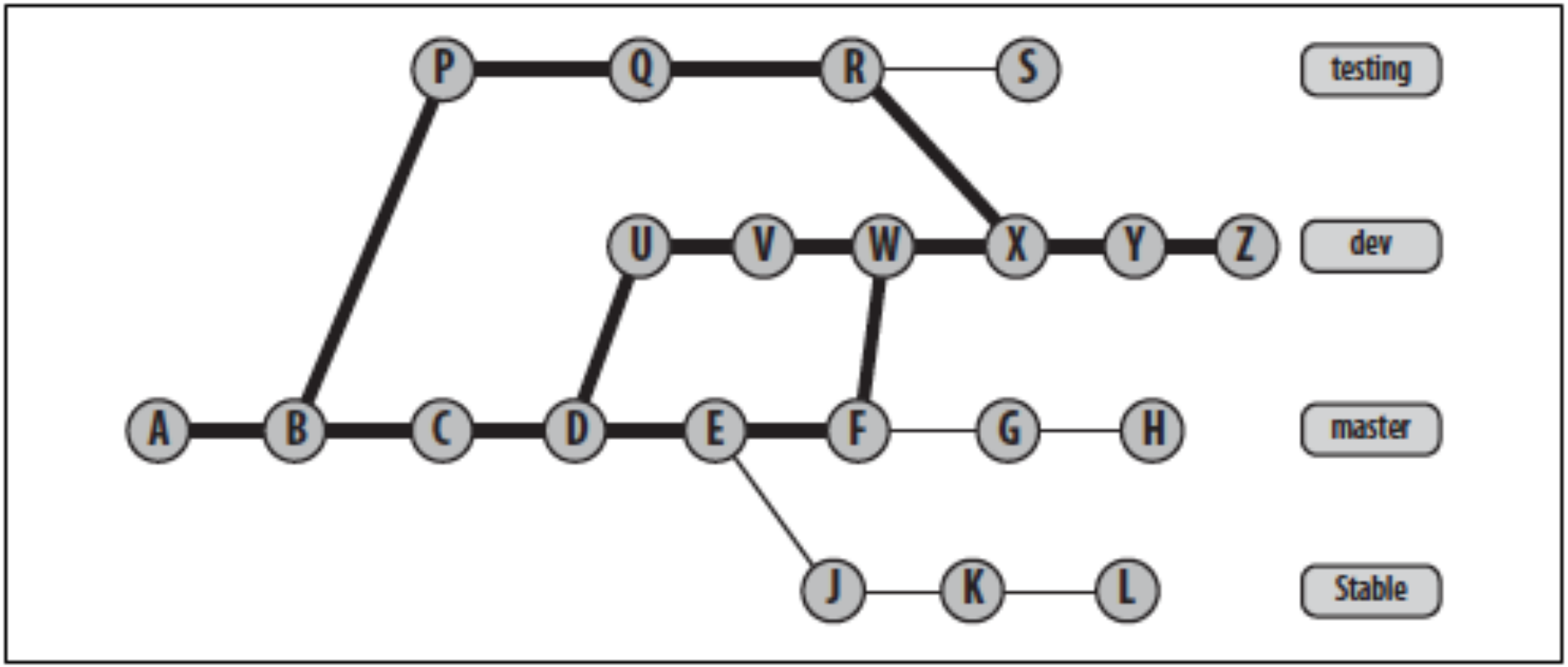
What is a branch

A separated line of development.

A split from a kind of unified, primal state, allowing development to continue in multiple directions **simultaneously** and, potentially, to produce different versions of the project.

Often, a branch is reconciled and merged with other branches to reunite disparate efforts.

What is a branch



Branch names

- Default branch in a repo is “master”

- Use an arbitrary name
  - / is allowed (not at the end)
  - No slash-separated component can begin with a dot (**.**). **feature/.new** is invalid
  - Cannot contain two consecutive dots (..)
  - Cannot contain whitespace, **~** , **^**, **:** ,**?**, ***, [**

Using branches

There may be many different branches within a repository at any given time.

There is at most one “active” branch that determines what files are checked out in the working directory.

By default, master is te active branch.

Creating branches

From the active branch:

$ git branch <branch-name>: Only creates a branch

$ git checkout -b <branch-name>: Creates and switches to the new branch

Listing branches

List branches names found in the repository:

$ git branch

```
  bug/pr-1
  dev
* master
```

$ git branch –a

Lists all branches (also tracked branches)

Viewing branches

```
$ git show-branch <branch1> <branch2>
```

```
$ git show-branch bug/*
```

Limits the result to the specific branches names.

Switching branches

Your working directory can reflect only one branch at a time.

To start working on a different branch:

$ git checkout <branch-name>

Delete local branches

You cannot delete the active branch.

$ git branch –d <branch>: Deletes if you have done merge.

$ git branch –D <branch> : Force the action even though you haven’t done merge.

2.- Push changes

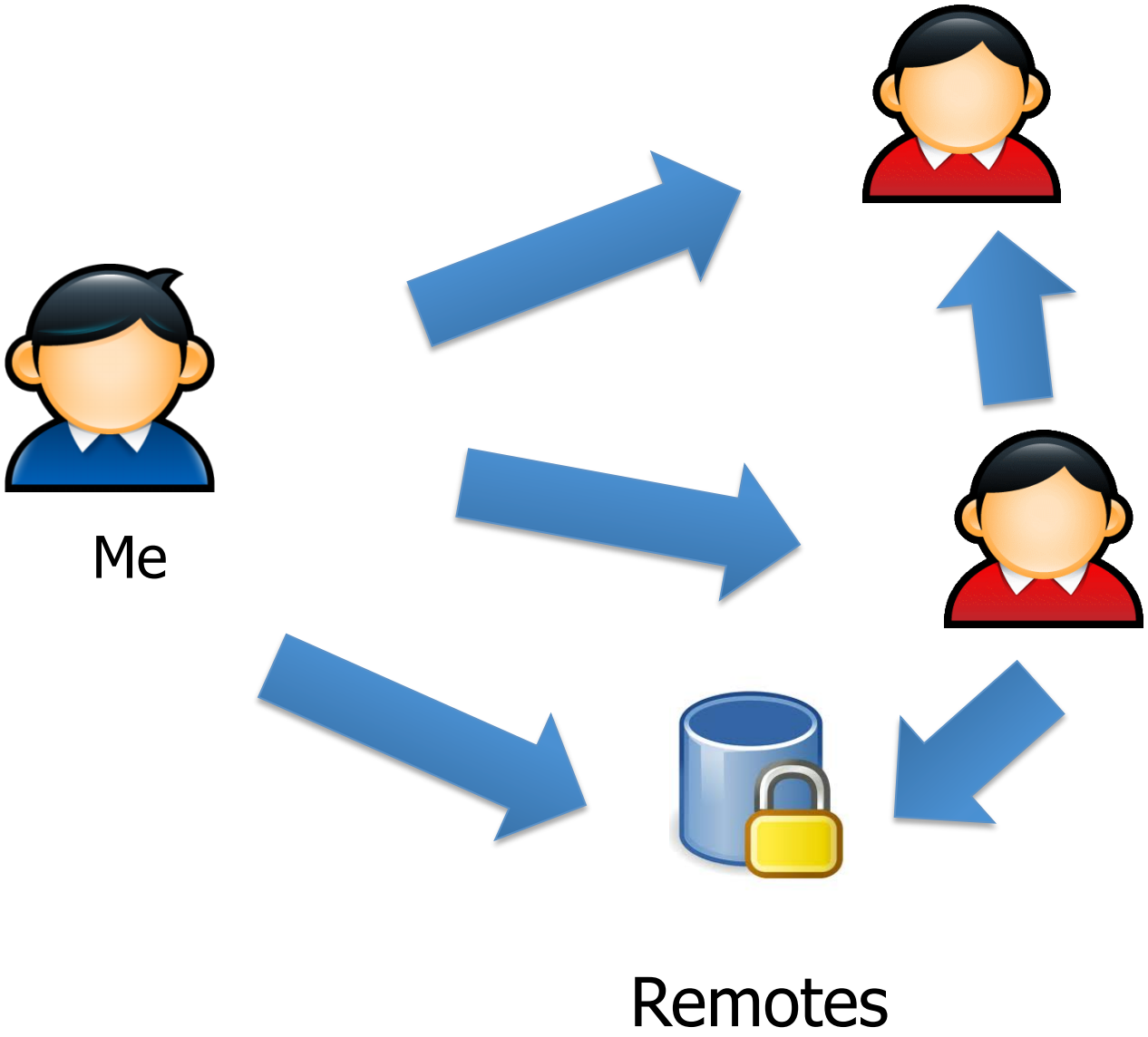
$ git push <remote> :<branch>

Hands on: Creating and deleting branches

THINKING WITHYOU

REMOTE REPOSITORIES

What is a remote



Nice to meet you!

Create a repository

You can create or clone a Git repository:

- Create

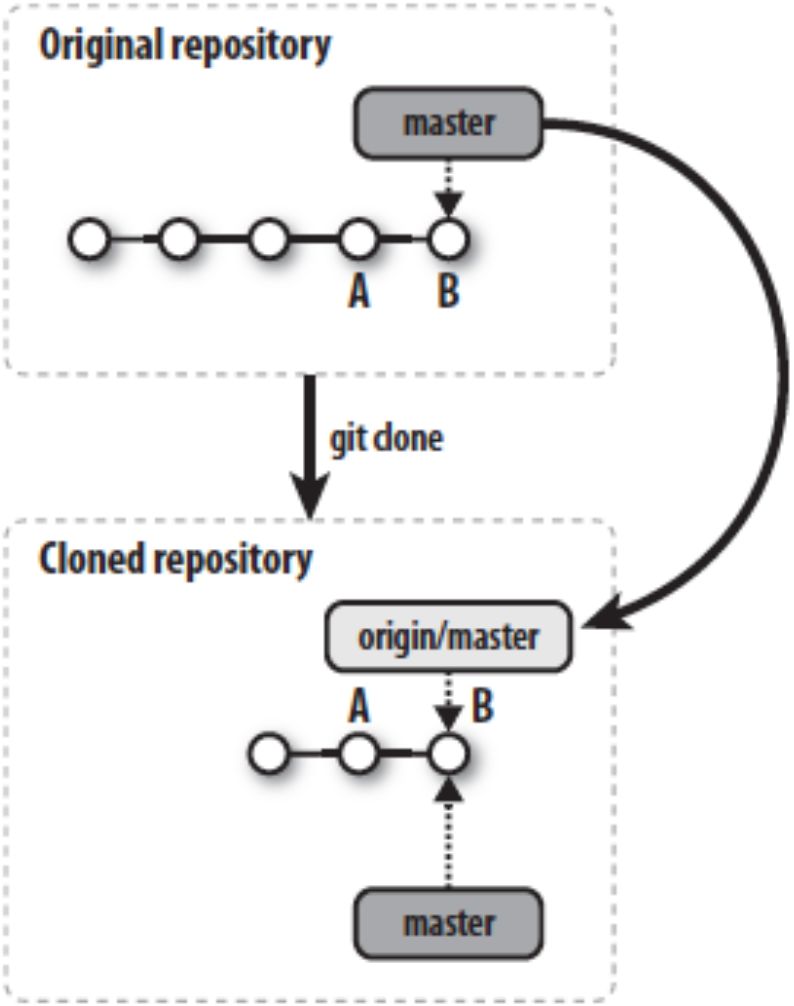
```
$ git init
```

- Clone

```
$ git clone <remote-url>
```

Create a repository

Clone a repo:



A remote repository is a reference to another repository.

A remote is a shorthand name for a Git URL.

You can define any number of remotes in a repository

Remotes

Git uses remote and tracking branch to reference and facilitate the “connection” to another repository.

$ git remote: to manipulate remotes

Remotes

$ git remote

add: adds a remote

rm: deletes a remote

rename: rename a remote repo

-v: list all remotes

Remotes

In addition to git clone , other common Git commands that refer to remote repositories are:

- git fetch: Retrieves objects and their related metadata from a remote repository
- git pull: Fetch + Merge
- git push: Transfers objects and their related metadata to a remote repository

Tracking branches

Used exclusively to follow the changes from another repository.

Not merge or make commits onto a tracking branch

Tracking branches

```
$ git fetch <remote-repo>
```

```
$ git checkout --track –b <local-branch> <remote-repo> / <remote-branch>
```

Push to a remote branches

Push

$ git push <repo> <branch>

Force Push

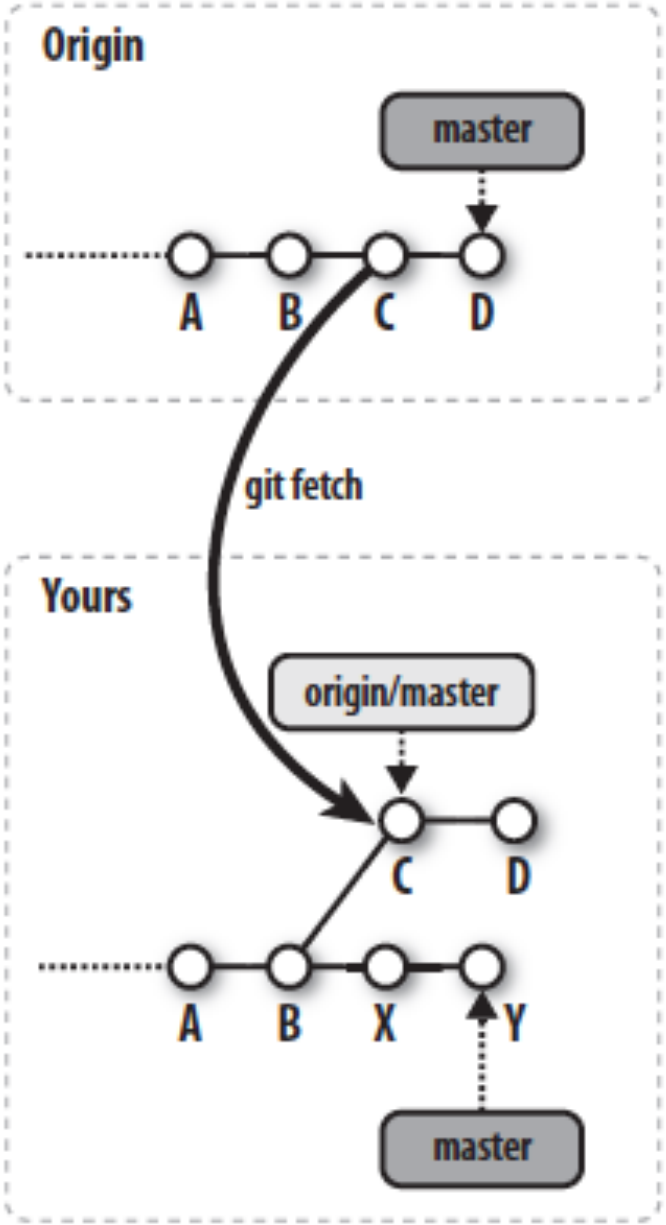
$ git push –f <repo> <branch>

¡Be careful forcing the push!

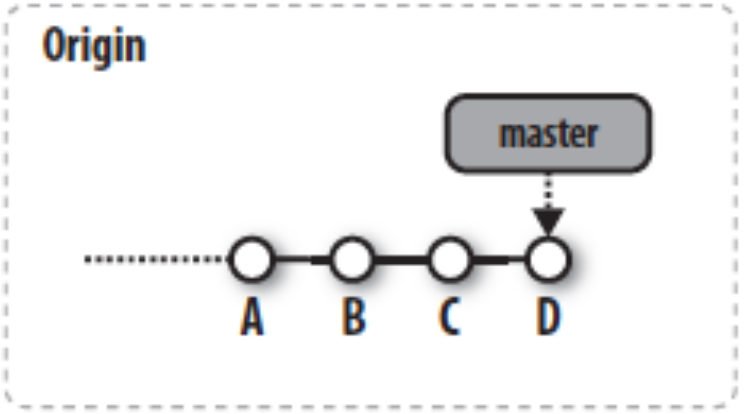
THINKING WITHYOU

git Fetch

$ git fetch <repo> <branch>

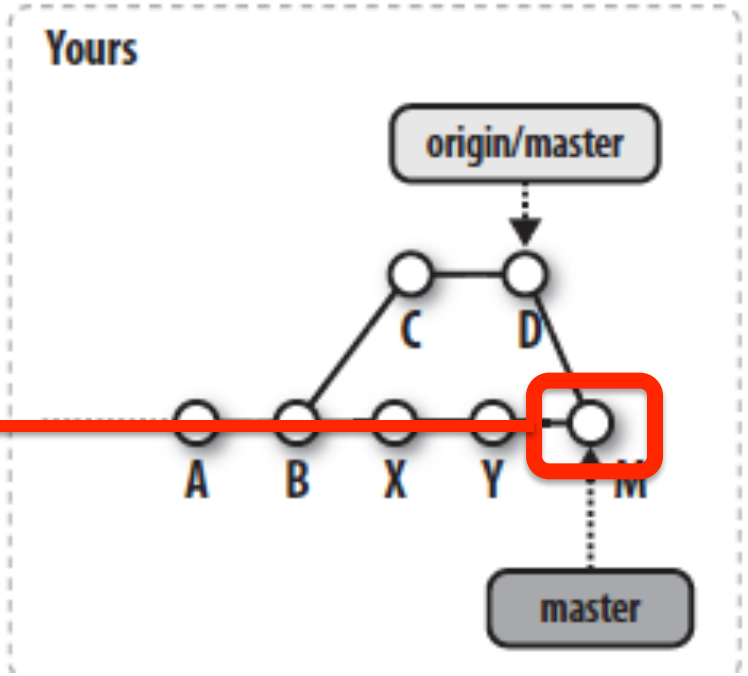


Merge



$ git merge <branch>

New commit is created



Pull

$ git pull <repo> <branch>

Git Fetch + Git Merge → Exactly the same as doing separated

Pull remote branches

When you clone a repository you can only clone the master branch.

$ git branch –a : View every branch you have got.

$ git checkout -b <local-branch> <repo>/<remote-branch>: Pull a remote branch into a local branch and switch it.

$ git branch <local-branch> <repo>/<remote-branch>: Pull a remote branch into a local branch.

THINKING
WITHYOU

MERGE

What is a merge

Unifies two or more commit history branches.

Most often, a merge unites just two branches.

Git supports a merge of three, four or many branches at the same time

All the branches to be merged must be present in the same repository.

What is a merge

When modifications in one branch do not conflict in another branch → new commit

When branches conflict (alter the same line) Git does not resolve the dispute.

Merge examples

You have to switch to the target branch and execute the merge:

```
$ git checkout destinyBranch
```

```
$ git merge anotherBranch
```

Preparing for a Merge

First of begin a merge → tidy up working directory

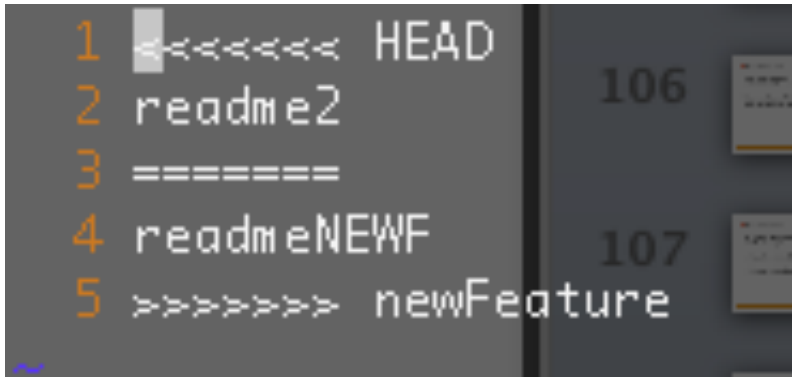
If you start a merge in a dirty state, Git may be unable to combine the changes from all the branches and those in your working directory or index in one pass.

Merge Conflicts

Git warns you about the conflict:

```
israelalcazar@Mac-Lamaro:~/dev/git/examples2 (master) $ git merge newFeature
Auto-merging README3.TXT
CONFLICT (content): Merge conflict in README3.TXT
Automatic merge failed; fix conflicts and then commit the result.
```

And marks the file:



THINKING WITHYOU

Locating conflicted Files

Which files have got conflicts?

Git keeps track of problematic files by marking each one in the index as conflicted or unmerged

You can use one of these commands:

$ git status

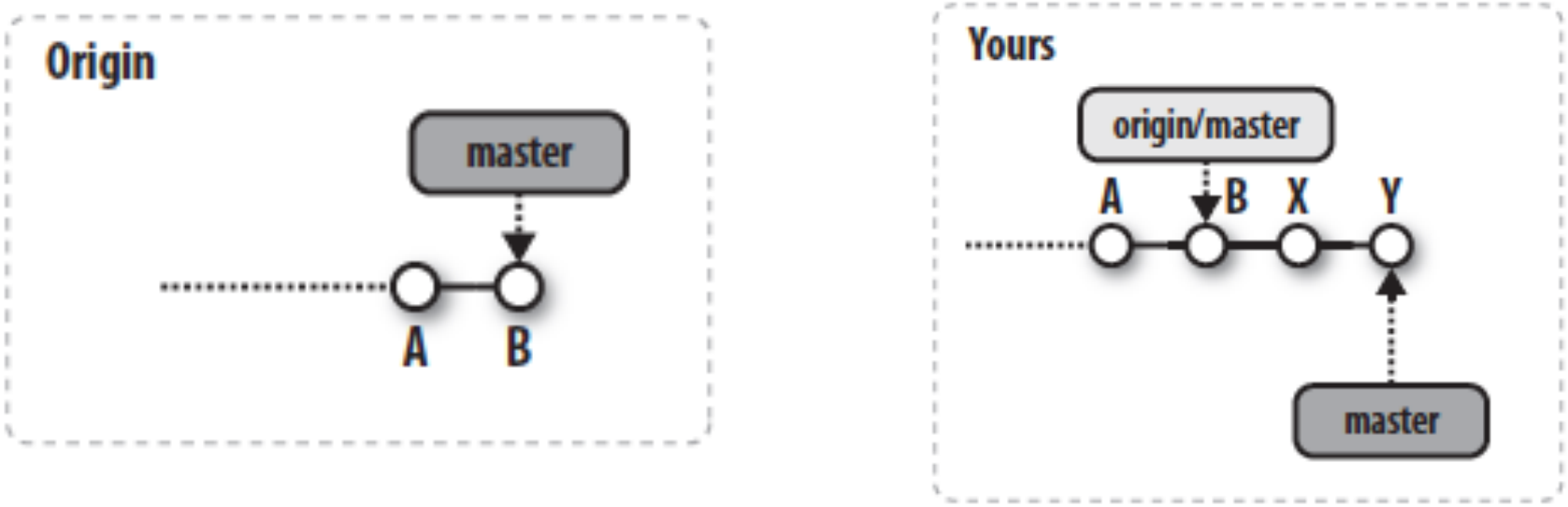
$ git diff

Abort a merge

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

Fast Forward

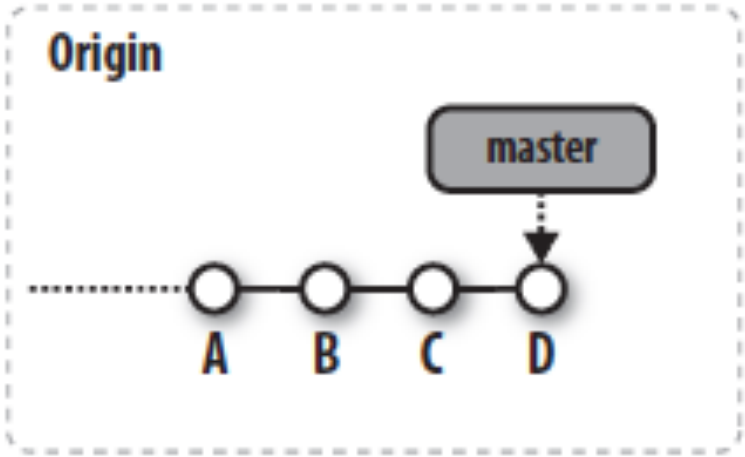
A simple linear history advancement operation

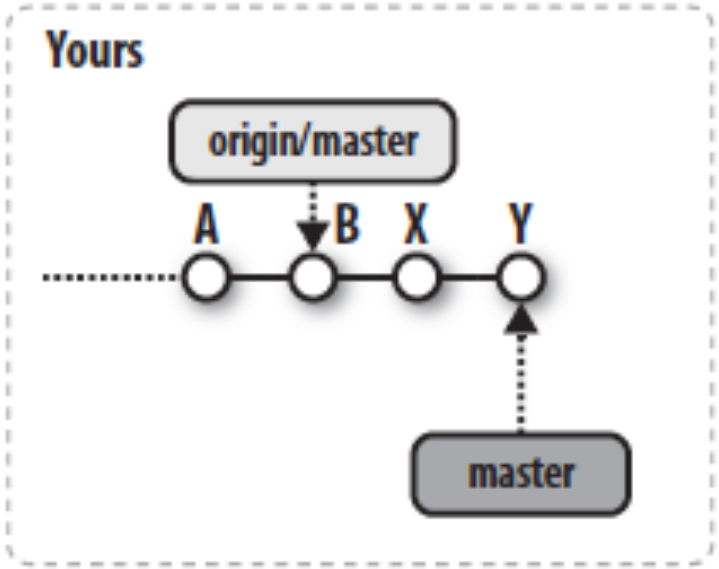


Fast forward has happened in origin from B to X

Fast Forward

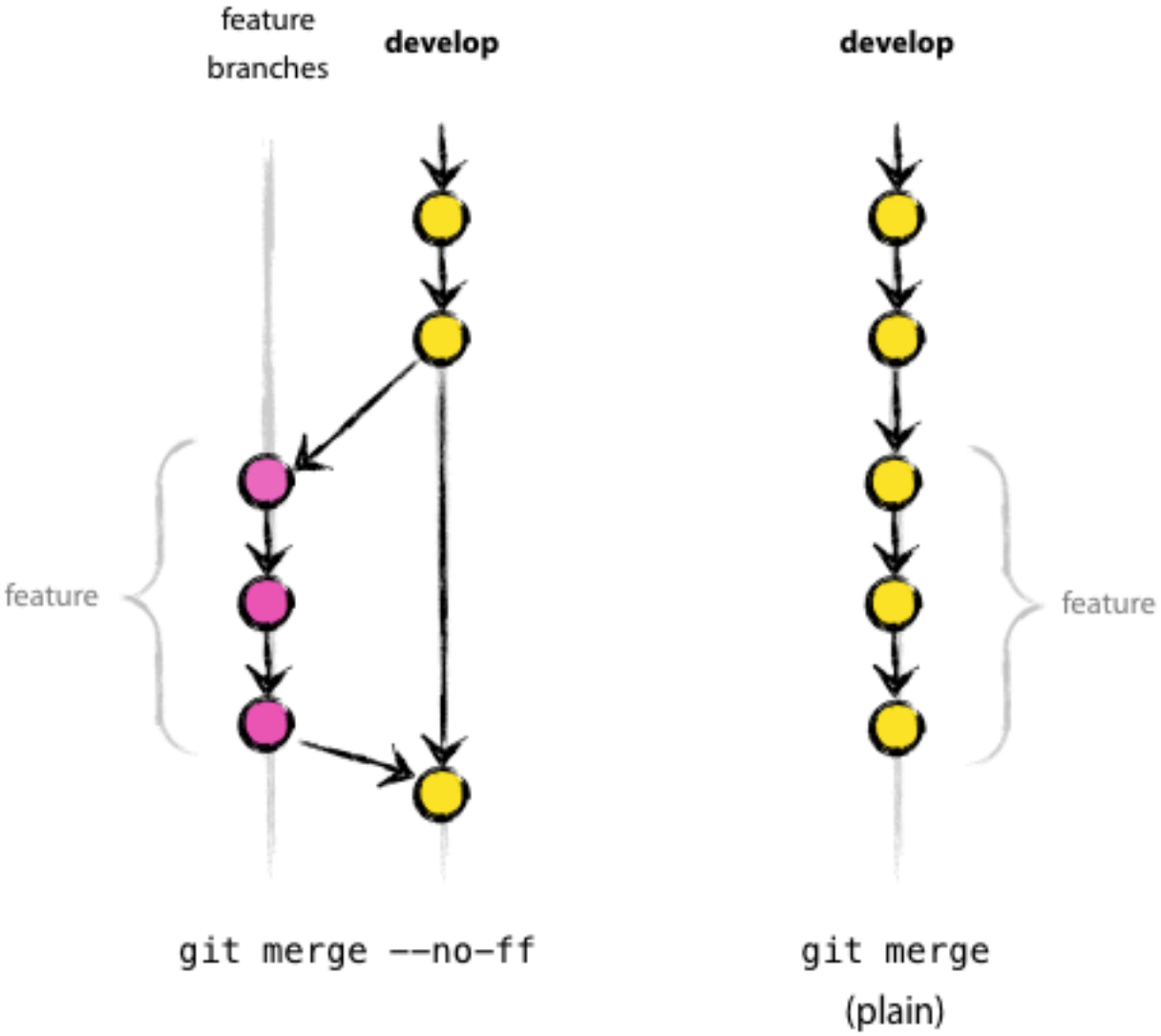
If another developer has pushed some changes (C, D) you can't push with fast forward.





You should merge your changes first.

Fast Forward Merging



THINKING WITHYOU

FIXING COMMITS

Fixing commits Flow

1. Discard changes in a file in the index to the copy in working directory

```
$ git checkout -- <files>
```

2. Reverting a file some commits before

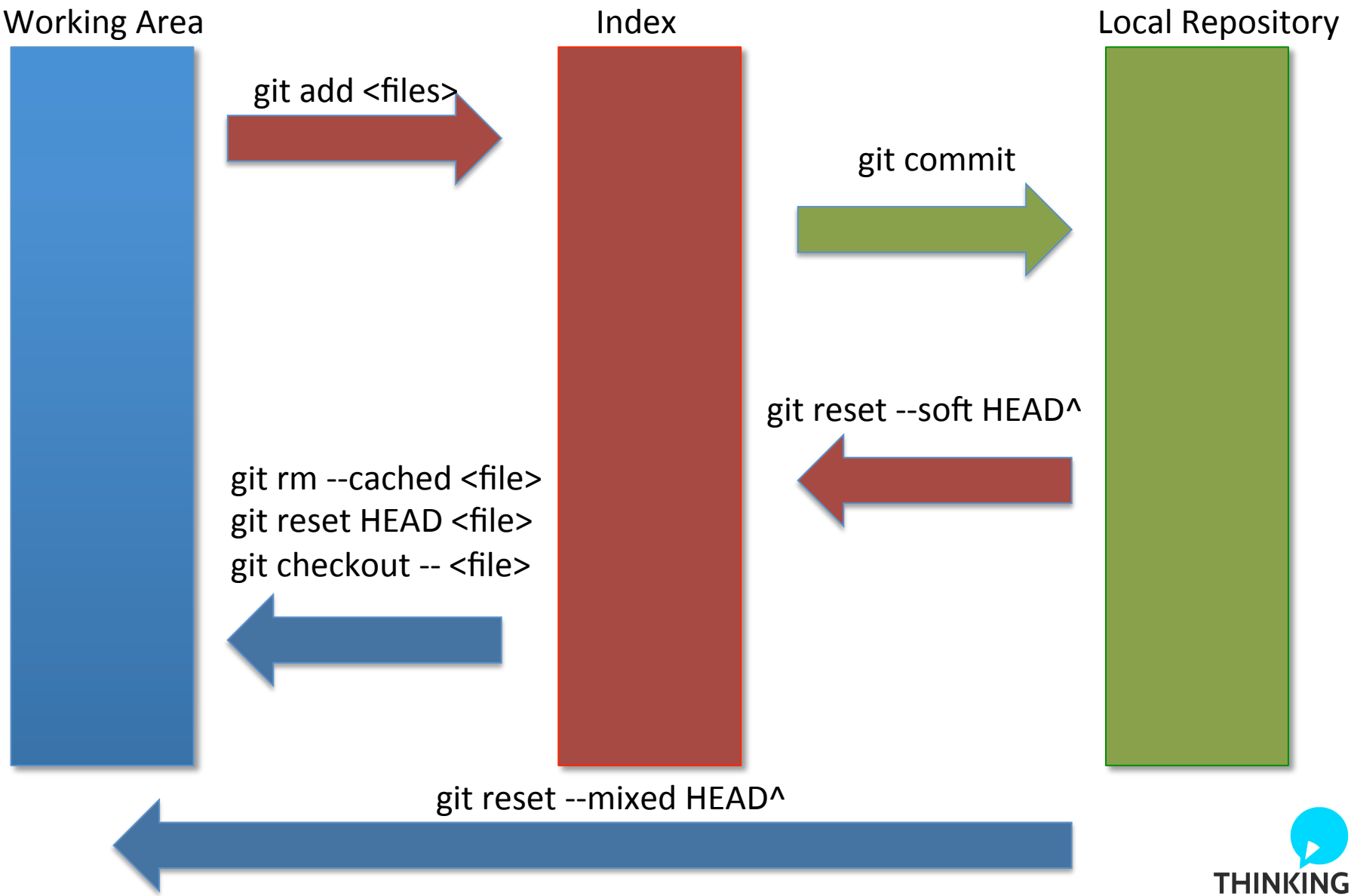
```
$ git checkout <commit-id> file
```

Basic Flow: Reset

You can reset some status of your repository:

```
$ git reset

$ git reset HEAD <file>

$ git reset --soft <commit-id>

$ git reset --hard <commit-id>

$ git reset --mixed <commit-id>
```

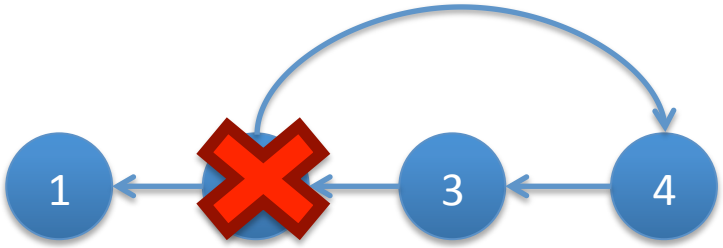
Basic Flow: Reset



Revert

Undoes a committed snapshot but instead of removing the commit appends a new commit undoing changes introduced by the commit.

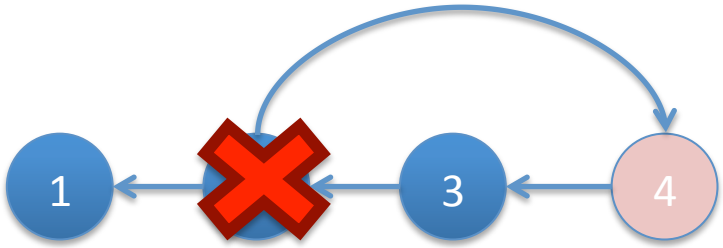
This prevents losing history from Git.



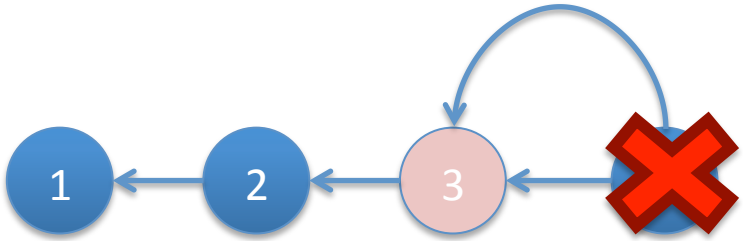
git revert <commit>

Revert vs Reset

Revert undoes a single commit, it does not "revert" back to the previous state of a project.



git revert



git reset

THINKING
WITHYOU

TAGS

Tags vs Branches

A tag is meant to be a static name that does not change or move over time. Once applied, you should leave it alone.

A branch is dynamic and moves with each commit you make. The branch name is designed to follow your continuing development.

You can give a branch and a tag the same name.

If you do, you will have to use their full ref names to distinguish them.

For example, you could use refs/tags/v1.0 and refs/heads/v1.0

Commands

$ git tag : List tags

$ git tag –a <name> -m “text to …”: Create a tag

$ git show <name>

$ git push <repo> --tags: Sends all tags
$ git push <repo> <tag-name>: Sends the tag

- **Lightweight**

Like a branch that doesn't change. A pointer to a specific commit

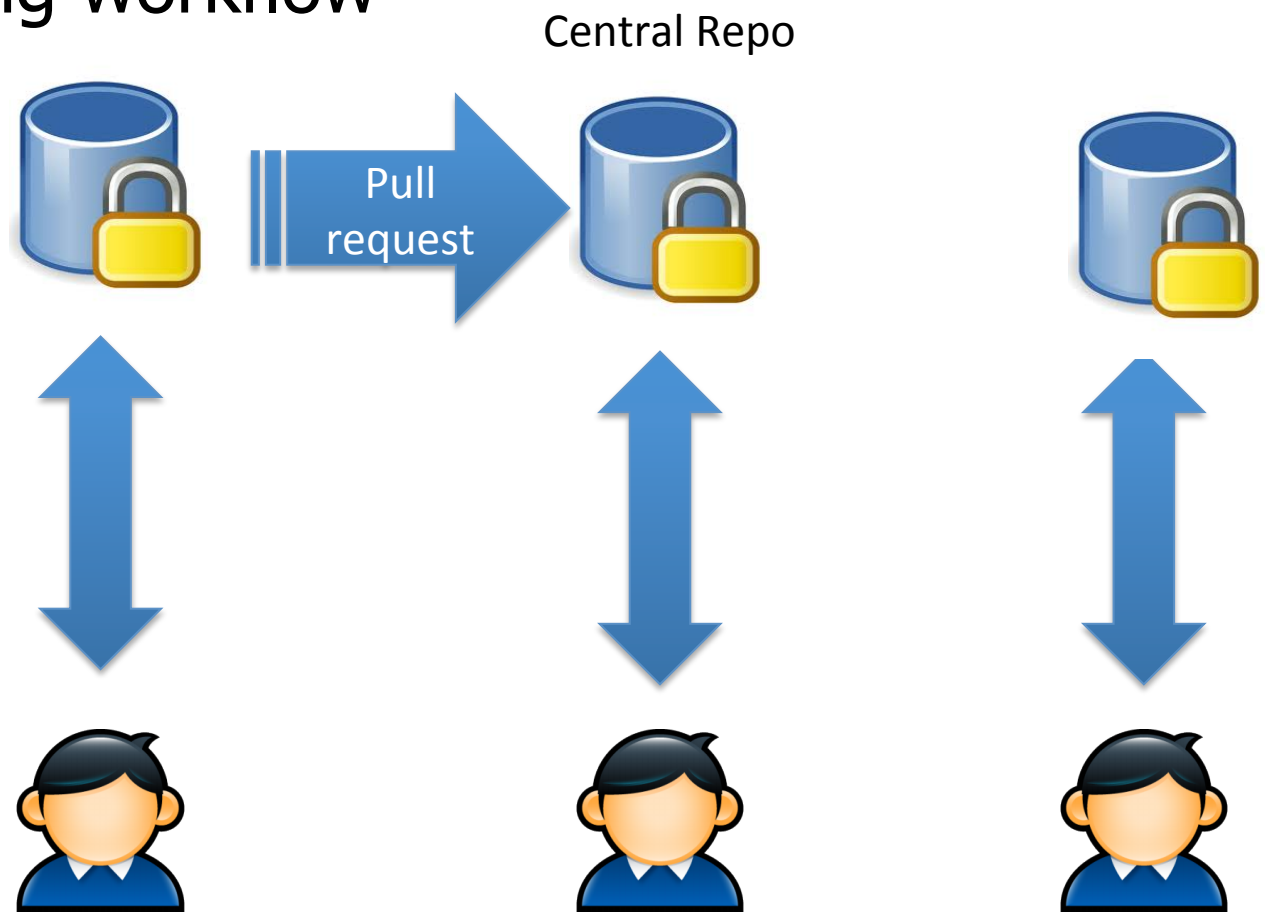
- **Annotated**

Are stored as full objects in the Git database. They are checksummed; contain the tagger name, email and date. Can be signed and verified.

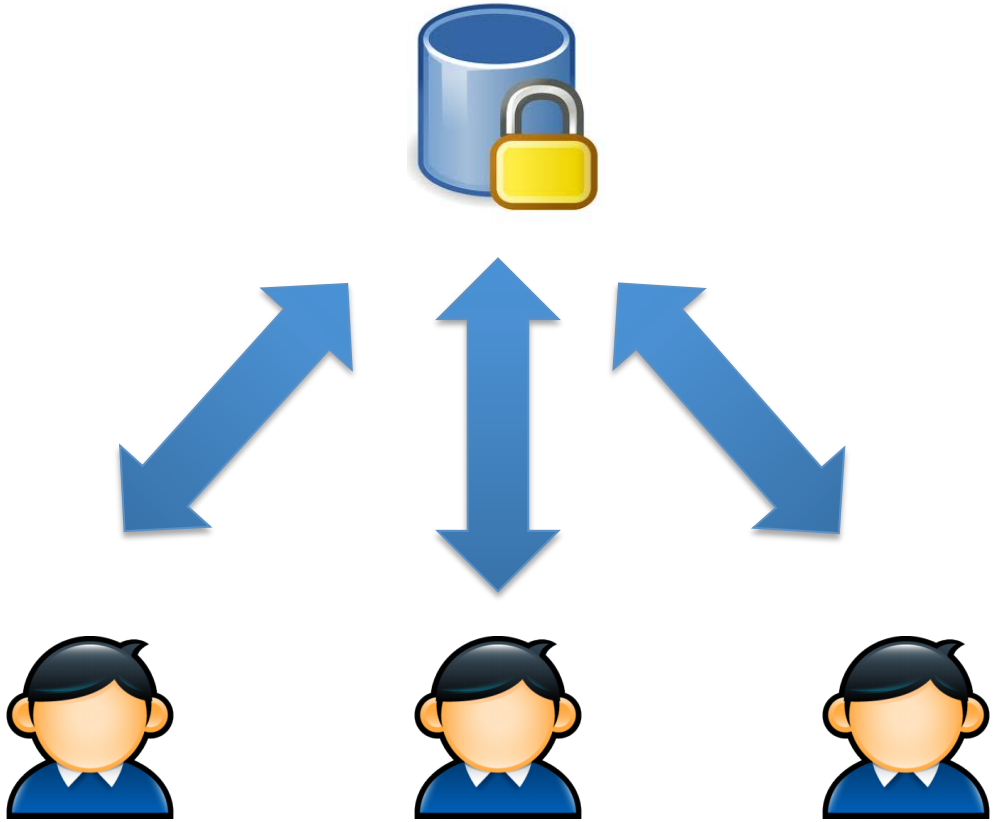
THINKING WITHYOU

WORKFLOWS

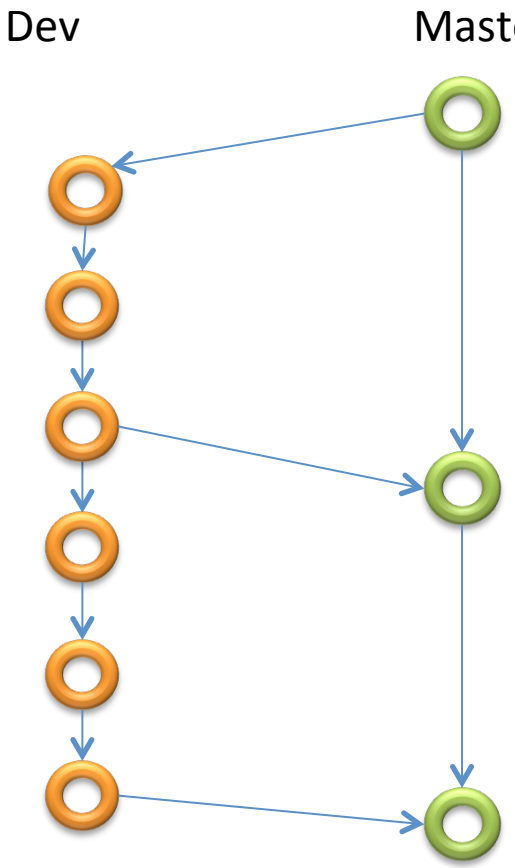
Forking workflow



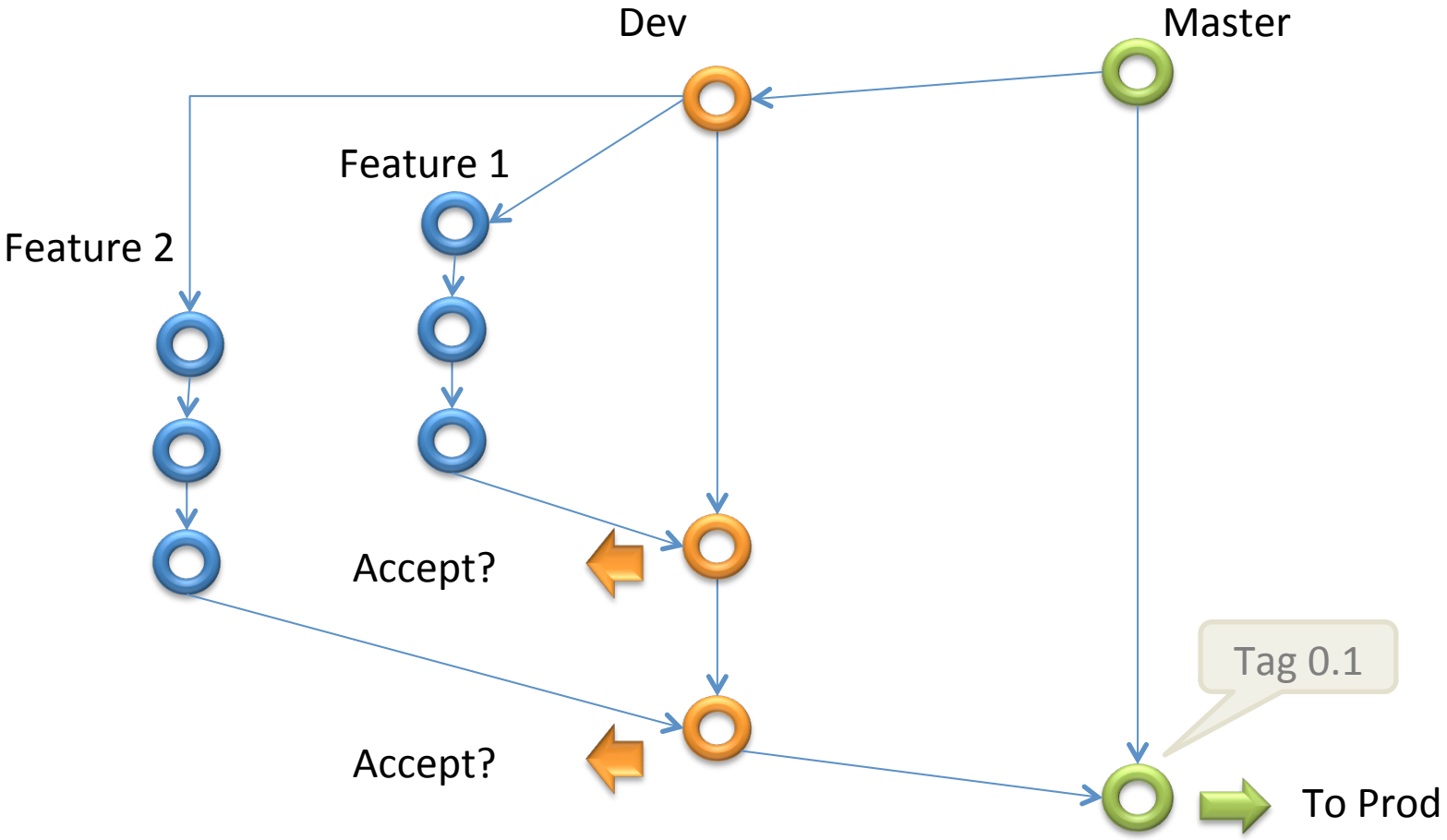
Centralized workflow



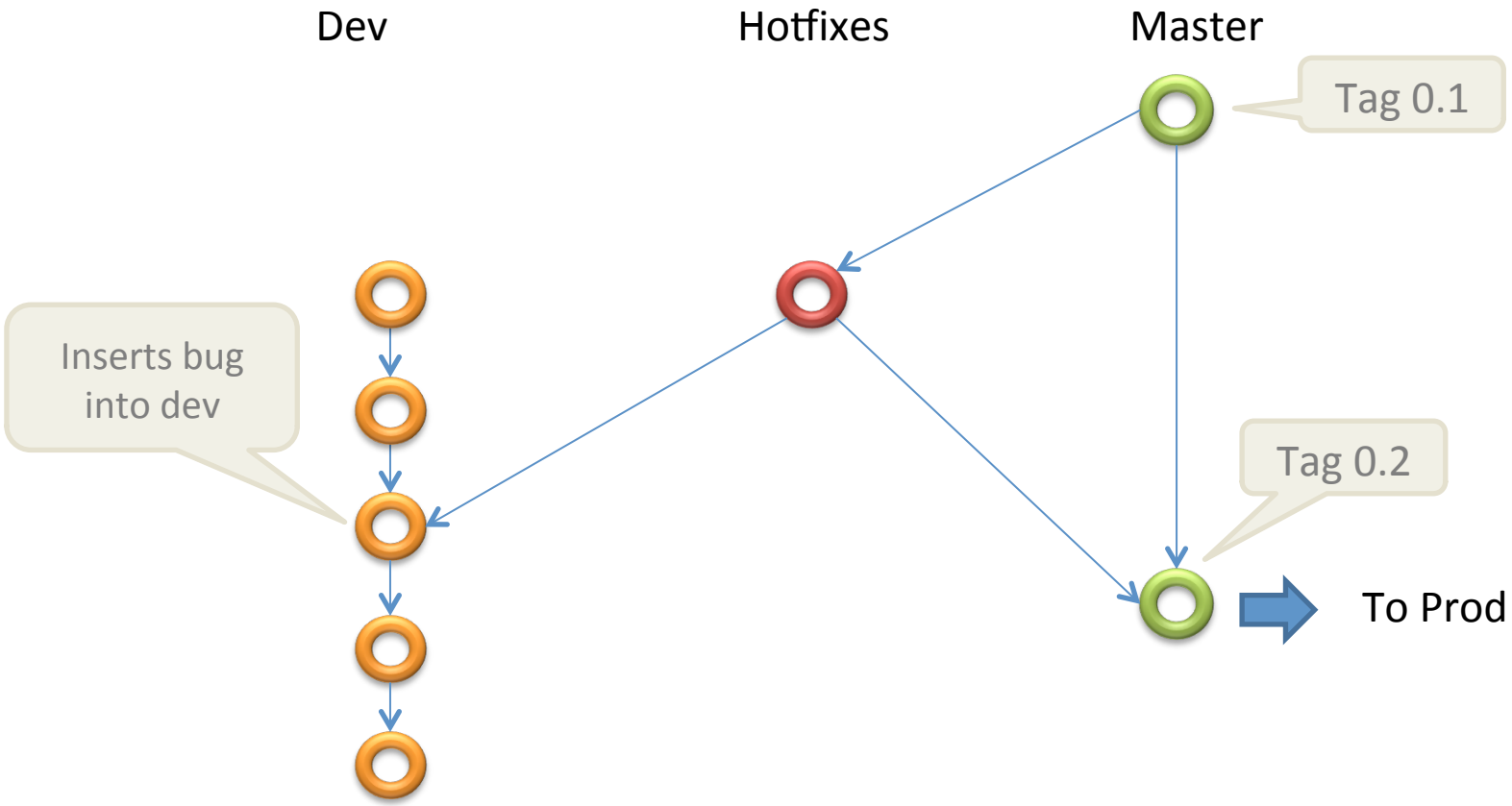
1) Two main branches: Dev & Master



2) One feature / one branch (local)



3) Every bug is fixed in an isolated branch



THINKING WITHYOU

ADVANCED CONFIGURATION

Set a default tool

```
$ git config --global merge.tool diffmerge

$ git config --global mergetool.diffmerge.cmd
"diffmerge –merge --result=\$MERGED \$LOCAL \
$BASE \$REMOTE"

$ git config –global
mergetool.diffmerge.trustExitCode true
```

git Alias

Aliases can avoid typing the same command over and over again.

Aliases were added in Git 1.4.0

git Alias

Configuration:

- Edit .git/config

```
[alias]
        ci = commit
```

- $HOME/.gitconfig: Available everywhere

- $ git config –global alias.ci commit

Git Blame

Tells you who last modified each line of a file and which commit made the change:

$ git blame <file>

Repository Types

There are two different types of Git repos:

- **Bare:** Has no working directory. Not be used for normal development. No direct commits.

- **Development:** Typical repository. Maintains current branch, provides checked-out copy of the current branch in a **working directory**

Repository Types

Bare repo is crucial for collaborative development.

Other developers clone and fetch from the bare repository and push updates to it

Bare repo creation

$ git init <repository> --bare

Best Practice:

A published repository should be bare

THINKING WITHYOU

BONUS:WORKING WITH DROPBOX

git Commands

- Create a folder into Dropbox folder
- $ git init --bare project.git
- Go to your project folder
  - $ git init
- Add the remote:
  - $ git remote add repositoryName ~/Dropbox/project.git
- $ git commit in project folder

- $ git push <repo> <branch>

Another developer clones the proyect:

- $ git clone ~/Dropbox/project
- Add remote
- $ git pull <repo> <branch>

Preguntas

Israel Alcázar
israel@thinkingwithyou.com
@ialcazar

THINKING WITHYOU