GIT

AN INTRODUCTION TO VCS/SCM USING GIT

What we are going to Cover

- Fundamental Concepts
- Installation and Configuration
- Working Locally
- Collaborating in a Team
- Understanding Branches
- Best Practices & Tips and Tricks

Fundamental Concepts

Version Control Systems

Version control (or revision control, or source control) is all about managing multiple versions of documents, programs, web sites, etc.

- Almost all "real" projects use some kind of version control
- Essential for team projects, but also very useful for individual projects

Some well-known version control systems are CVS, Subversion, Mercurial, and Git

- CVS and Subversion use a "central" repository; users "check out" files, work on them, and "check them in"
- Mercurial and Git treat all repositories as equal

Distributed systems like Mercurial and Git are newer and are gradually replacing centralized systems like CVS and Subversion

Why version control?

For working by yourself:

- Gives you a “time machine” for going back to earlier versions
- Gives you great support for different versions (standalone, web app, etc.) of the same basic project

For working with others:

- Greatly simplifies concurrent work, merging changes
- Great way to collaborate in a multi-project multi-Developer environment
- Very effective for due diligence purposes.

Why Git?

Git has many advantages over earlier systems such as CVS and Subversion

- More efficient, better workflow, etc.
- See the literature for an extensive list of reasons
- Of course, there are always those who disagree

Best competitor: Mercurial

- Some like Mercurial better
- Same concepts, slightly simpler to use
- In my (very limited) experience, the Eclipse plugin is easier to install and use
- Much less popular than Git

What is Git?

Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, data Integrity and support for distributed, Non-linear workflows.

Git was initially designed and developed by ***Linus Torvalds*** for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development.

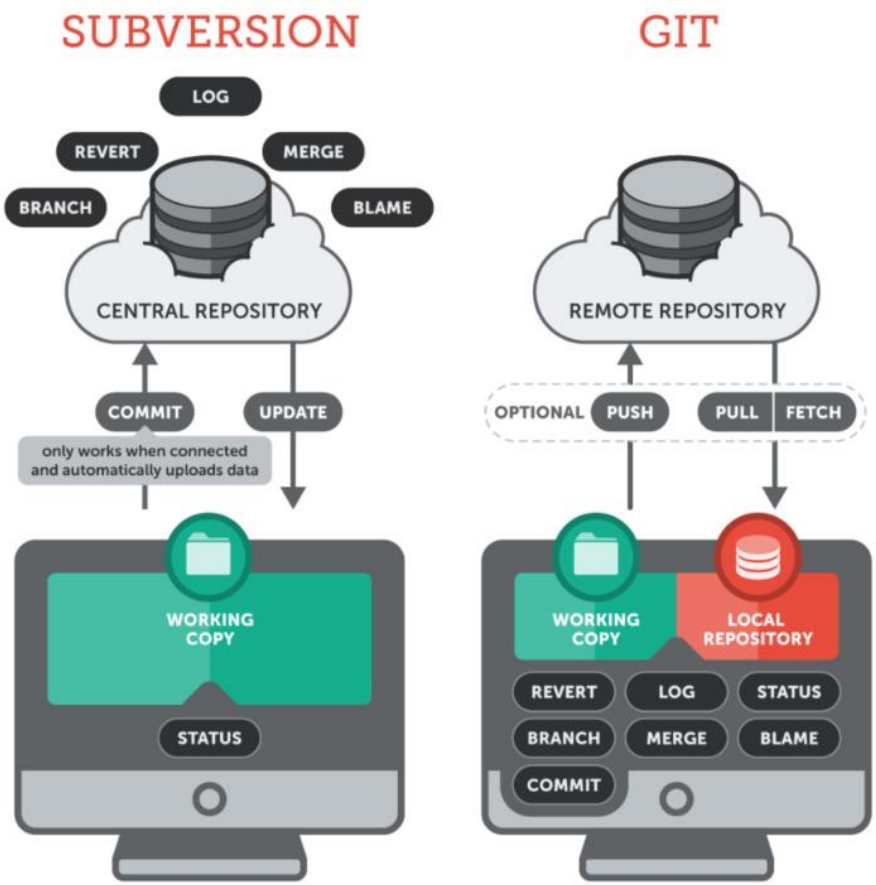
Git Architecture Advantages

- Speed
- Simple design
- Strong support for thousands of parallel branches
- Fully distributed
  - ✓Full local repository
  - ✓Offline commits
  - ✓Full size repository
- Able to handle large projects like Linux kernel effectively
- Ensure Integrity

Distributed Development

- Every Git working directory contains the complete repository, history and full revision tracking capabilities.
- You are not dependent on a Central Server and you don't have to be online.
- Git is extremely fast – much faster than SVN, CVS and other systems.
- Revisions (commits) you make to your local repository are available to you only.
- The next time you connect to the internet, push your changes to a remote repository to share them and back them up.

Centralized vs. Distributed



Installation and Configuration

Download and install Git

Official webpage: http://git-scm.com/downloads

- For GIT on windows or Mac use above link to download the latest version of GIT for respective OS.
- For Linux distributions, standard installation procedure works fine:
  - ✓ *sudo yum install git (for RHEL/CentOS etc.)*
  - ✓ *sudo apt-get install git (for debian/Ubuntu etc.)*

Note: Git is primarily a command-line tool, however there are a number of commercial GUI options available such as GitLab, BitBucket, Tortoise GIT etc.

Introduce yourself to Git

Enter these lines (with appropriate changes):

- *git config --global user.name "firstname lastname"*
- *git config --global user.email emailid@company.com*

Note: You need to do above mentioned modification only once

If you want to use a different name/email address for a particular project, you can change it for just that project

- cd to the project directory
- Use the above commands, but leave out the --global

Choose an editor

When you “commit”, git will require you to type in a commit message

For longer commit messages, you will use an editor

The default editor is probably vim

To change the default editor:

- git config --global core.editor /path/to/editor

You may also want to turn on colors:

- git config --global color.ui auto

Command Summary - Configuration

1. git config --global user.name “firstname lastname"
2. git config --global user.email emailid@company.com
3. git config --global color.ui auto
4. git config --global core.editor /path/to/editor

Working Locally

Create a new repository and add content

1. cd to the project directory you want to use
2. Type in git init
   - This creates the repository (a directory named .git)
   - You seldom (if ever) need to look inside this directory
3. Type in "git add ."
   - The period at the end is part of this command!

     Note: Period means "this directory"
   - This adds all your current files to the repository
4. Type in git commit –m "Initial commit"
   - You can use a different commit message, if you like

Clone an existing repository

- **git clone *URL***
- **git clone *URL mypath***

These make an exact copy of the repository at the given URL

- **git clone git://github.com/rest_of_path/file.git**

Github is the most popular (free) public repository

All repositories are equal but you can treat some particular repository (such as one on Github) as the "master" directory

Typically, each team member works in his/her own repository, and "merges" with other repositories as appropriate

The repository

Your top-level **working directory** contains everything about your project

- The working directory probably contains many subdirectories—source code, binaries, documentation, data files, etc.
- One of these subdirectories, named **.git**, is your **repository**

At any time, you can take a "snapshot" of everything (or selected things) in your project directory, and put it in your repository

- This "snapshot" is called a **commit object**
- The commit object contains (1) a set of files, (2) references to the "parents" of the commit object, and (3) a unique "SHA1" name
- Commit objects do ***not*** require huge amounts of memory

init and the .git repository

When you said **git init** in your project directory, or when you cloned an existing project, you created a repository

- The repository is a subdirectory named **.git** containing various files
- The dot indicates a "hidden" directory
- You do *not* work directly with the contents of that directory; various git commands do that for you
- You ***do*** need a basic understanding of what is in the repository

Making commits

You can work as much as you like in your working directory, but the repository isn't updated until you **commit** something.

If you create new files and/or folders, they are *not tracked* by Git unless you ask it to do so. This is also known as 'staging'.

- git add *newFile1 newFolder1 newFolder2 newFile2*

Committing makes a "snapshot" of everything being tracked into your repository

- A message telling what you have done is required
- git commit –m "My Super Commit Message"
- git commit (This version opens an editor for you to enter the message)

Commit messages

In git, “Commits are cheap.” Do them often.

When you commit, you must provide a one-line message stating what you have done

- ✓ Terrible message: “Fixed a bunch of things”
- ✓ Better message: “Corrected the calculation of median scores”

Commit messages can be very helpful, to yourself as well as to your team members

You can’t say much in one line, so commit often.

Commits and graphs

A **commit** is when you tell git that a change (or addition) you have made is ready to be included in the project

When you commit your change to git, it creates a **commit object**

- A commit object represents the complete state of the project, including all the files in the project
- The *very first* commit object has no "parents"
- Usually, you take some commit object, make some changes, and create a new commit object; the original commit object is the parent of the new commit object
  - Hence, most commit objects have a single parent
- You can also **merge** two commit objects to form a new one
  - The new commit object has two parents

Hence, commit objects form a **directed graph**

- Git is all about using and manipulating this graph

Ignoring files in Git

3 places to put file names to be ignored:

- **.gitignore**

  Specific to folder, go with source code to public repo

- **.git/info/exclude**

  Not share with others

- **Git config core.excludefile <path>**

  Can use system, global or default config file.

Working with your repository

A **head** is a reference to a commit object.

The "current head" is called **HEAD** (all caps)

Usually, you will take **HEAD** (the current commit object), make some changes to it, and commit the changes, creating a new current commit object

- This results in a linear graph: A → B → C → …→ HEAD

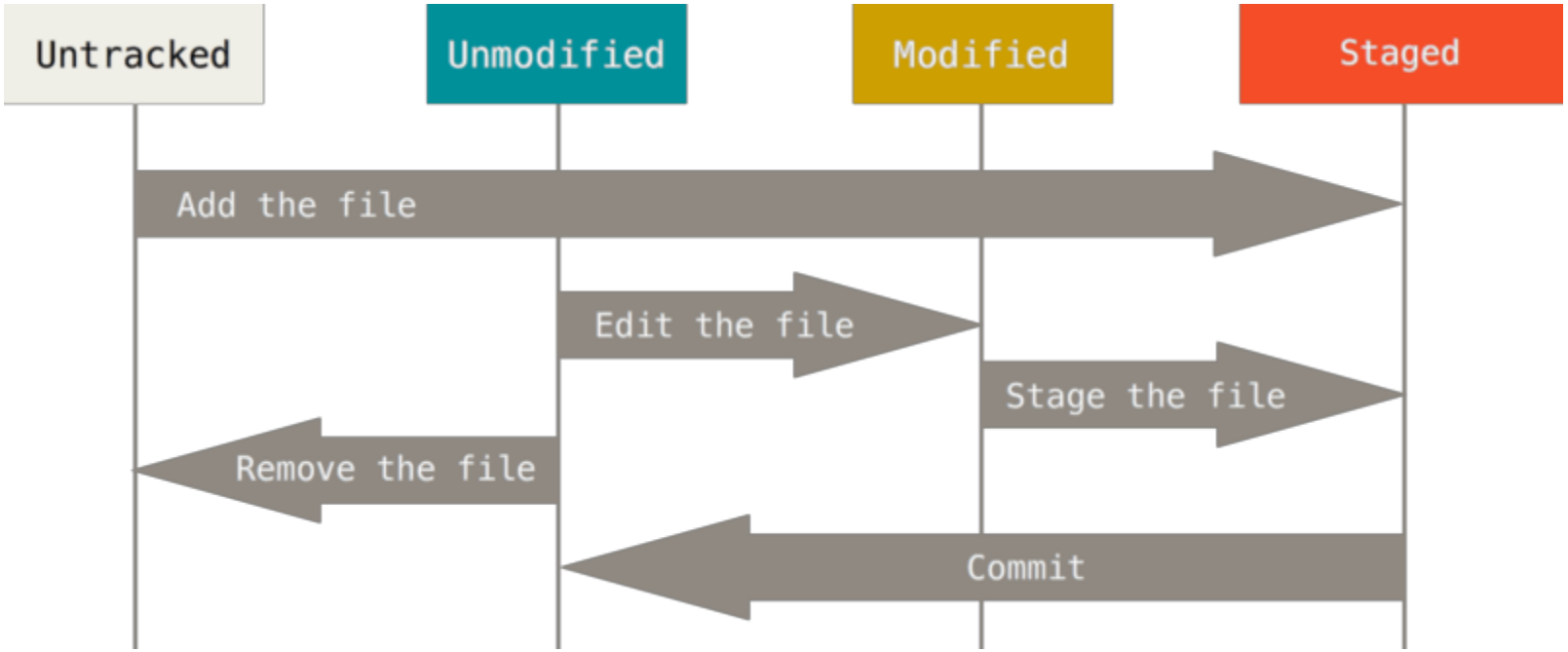
You can also take any previous commit object, make changes to it, and commit those changes

- This creates a **branch** in the graph of commit objects

You can **merge** any previous commit objects

- This joins branches in the commit graph

GIT File Lifecycle



Command Summary – Local Development

1. git init
2. git add .
3. git commit
4. git commit –m “commit message”
5. git clone *URL*
6. *gitignore*

Collaborating in a Team

Collaboration in GIT

SVN uses a single central repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers' working copies and the central repository.

This is different from Git's collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure.

Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

Collaboration via a Central GIT Server

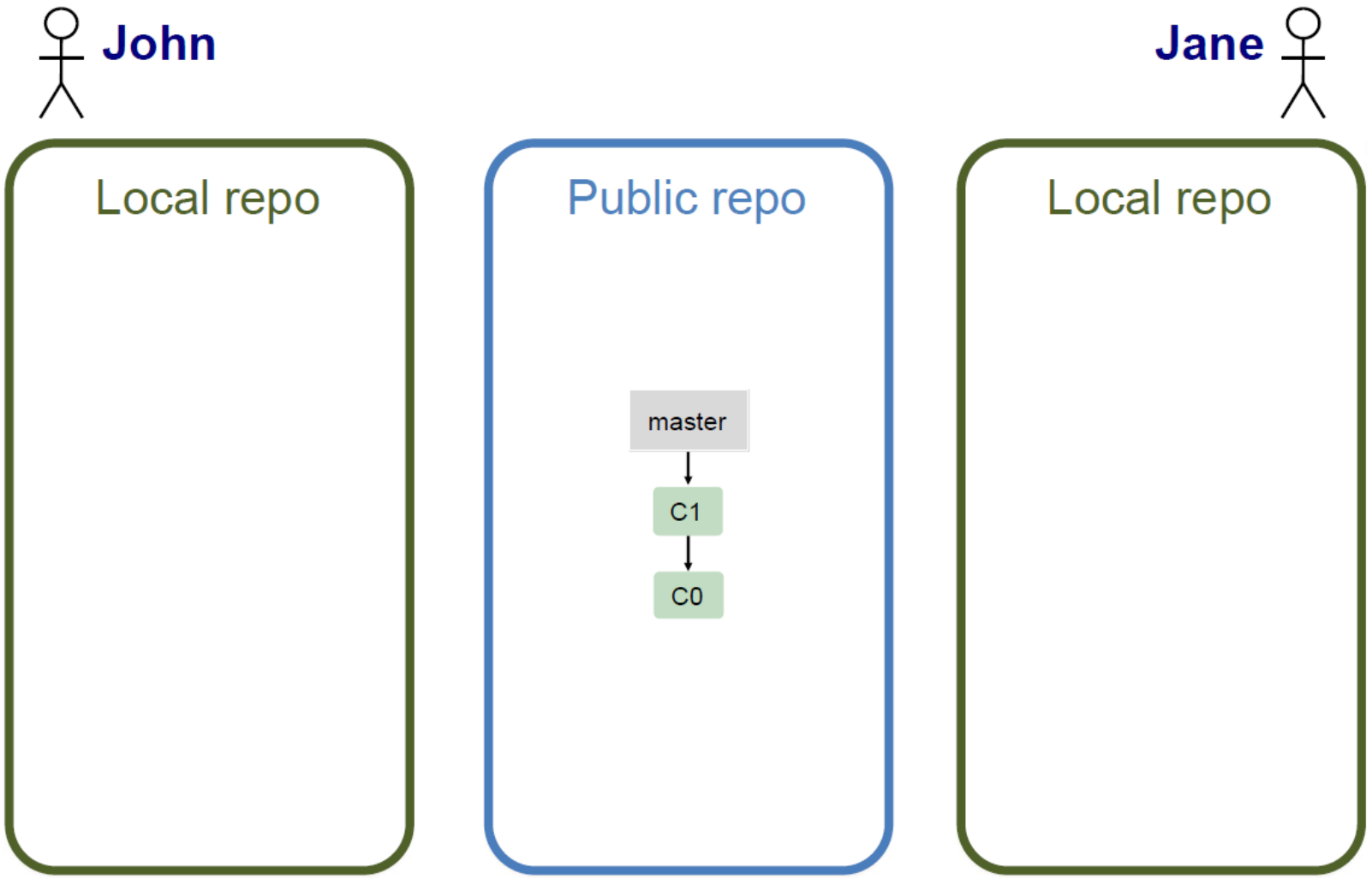
All repositories are equal, but it is convenient to have one central repository in the cloud

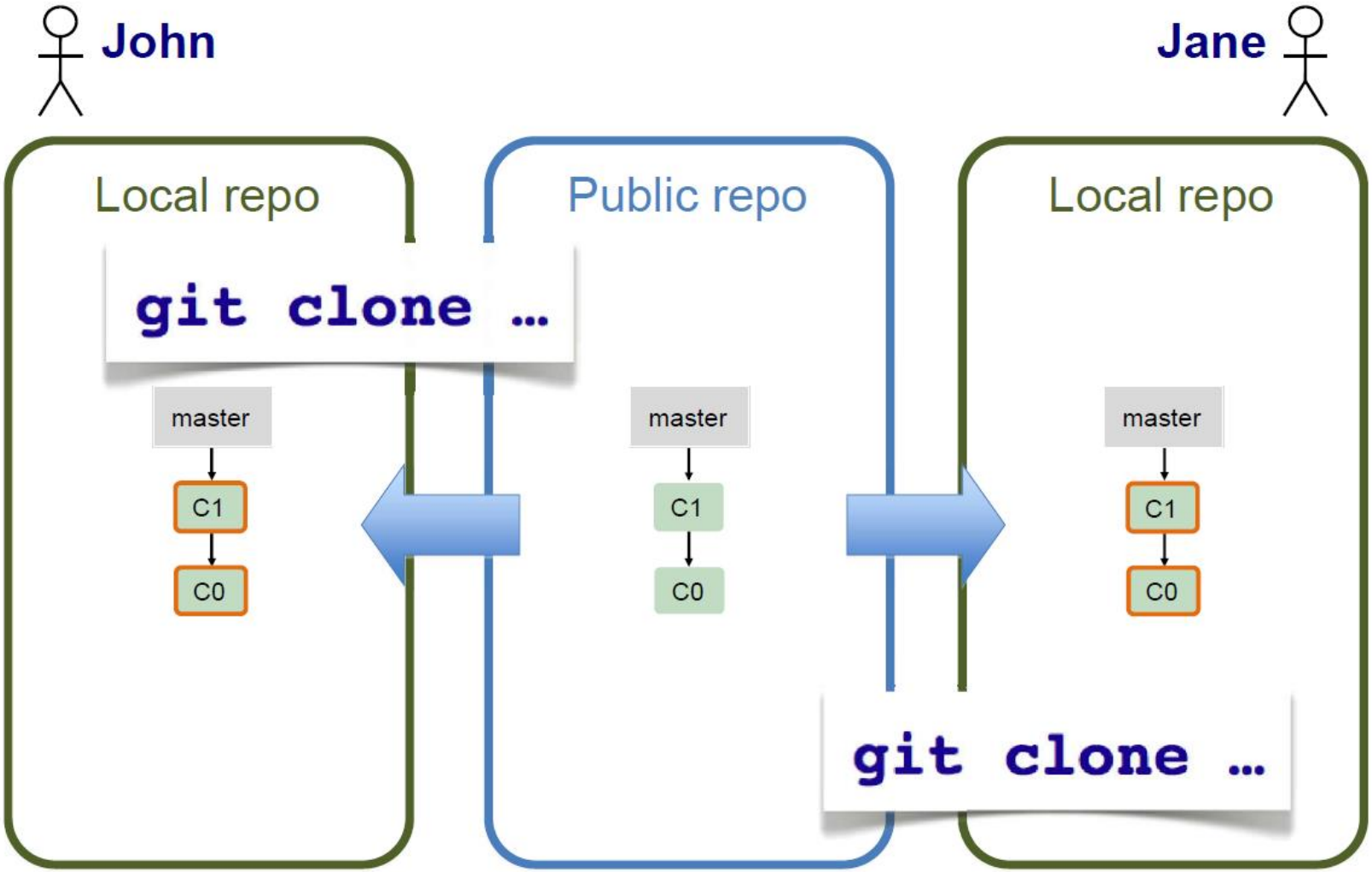
Here's what you normally do:

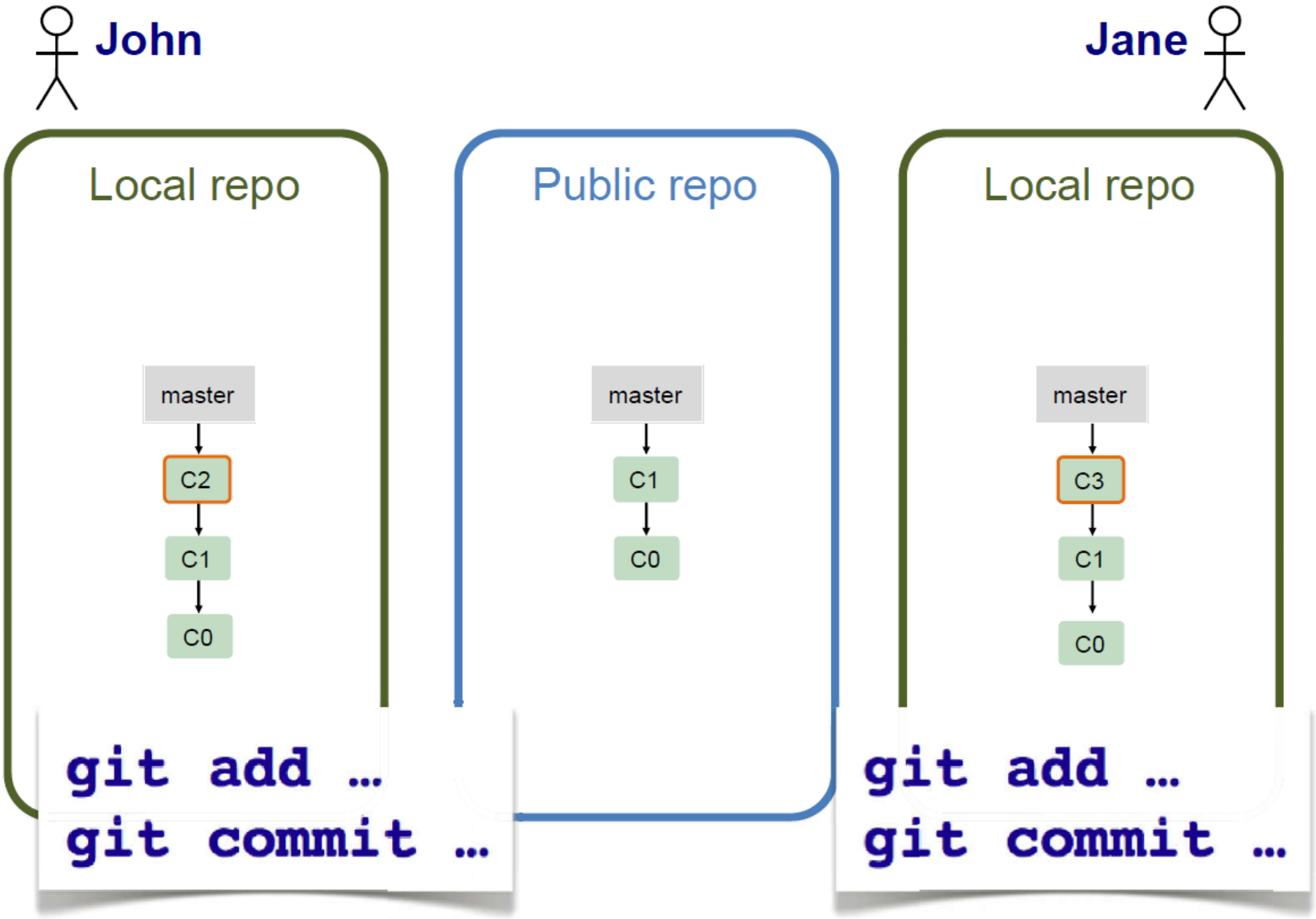
- Download the current HEAD from the central repository
- Make your changes
- Commit your changes to your local repository
- Check to make sure someone else on your team hasn't updated the central repository since you got it
- Upload your changes to the central repository

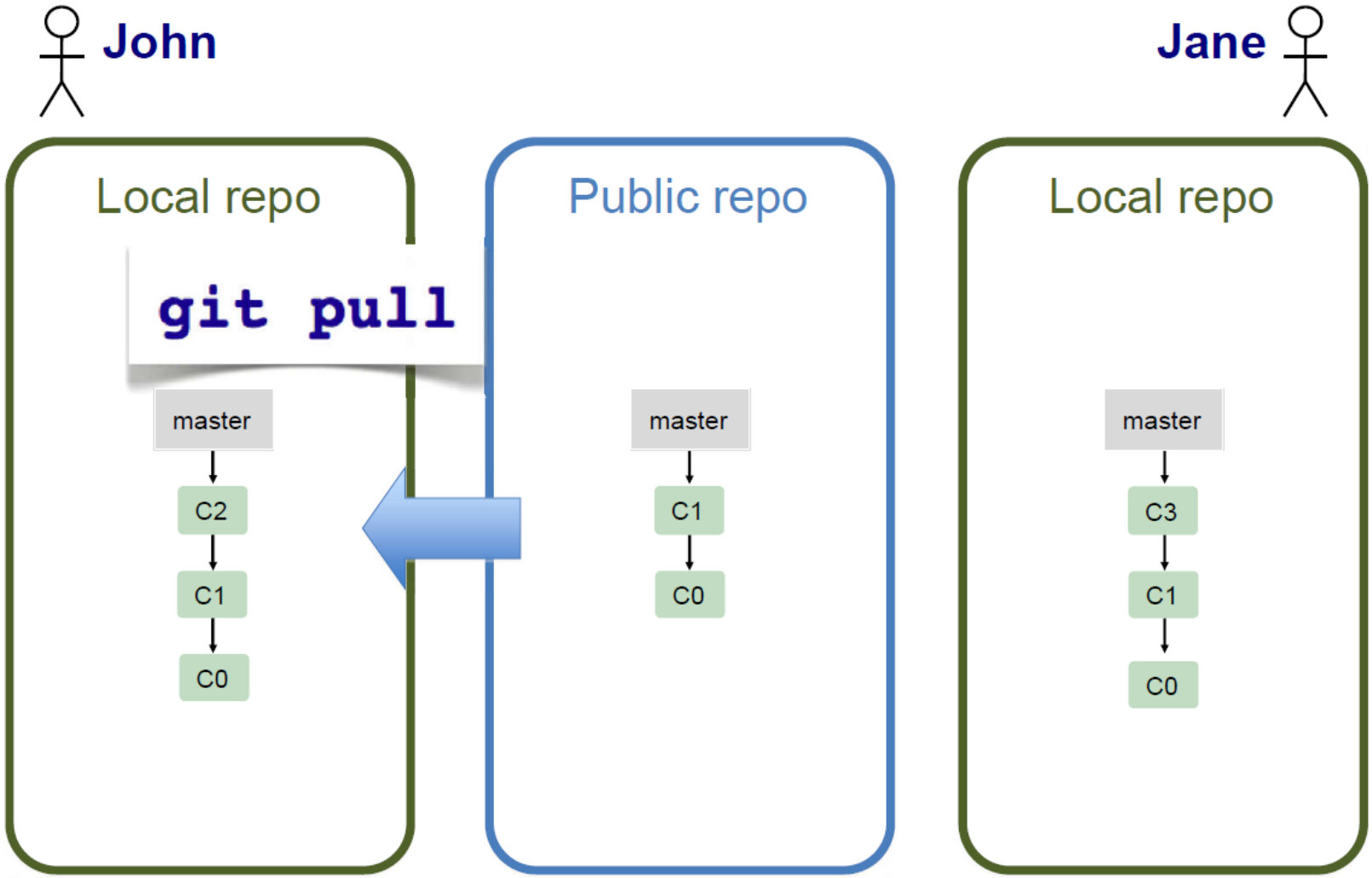
If the central repository *has* changed since you got it:

- It is *your* responsibility to **merge your two versions**
  - This is a strong incentive to commit and upload often!
- Git can often do this for you, if there aren't incompatible changes

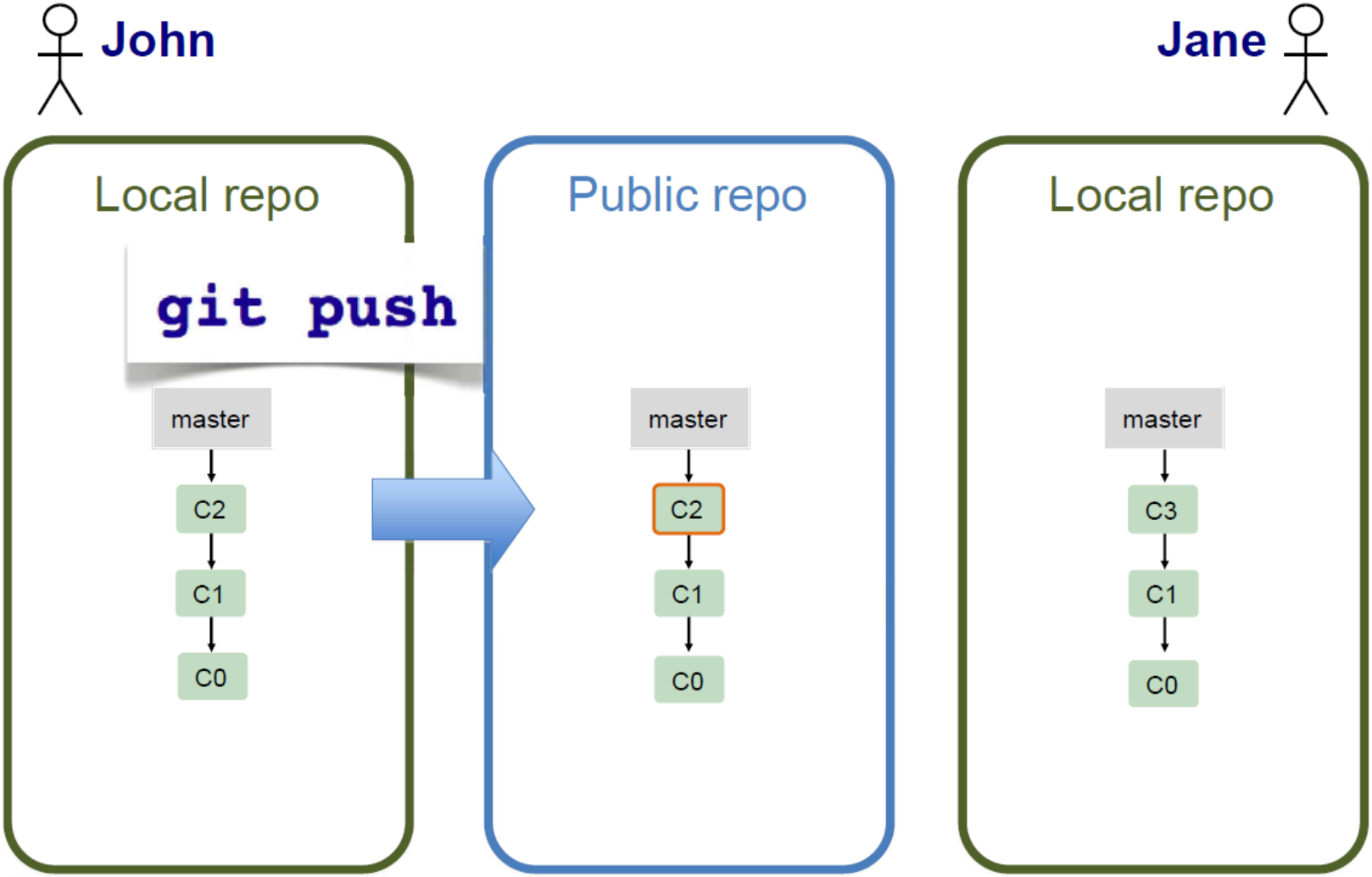

John
Jane
Local repo
Public repo
Local repo
master
C1
C0

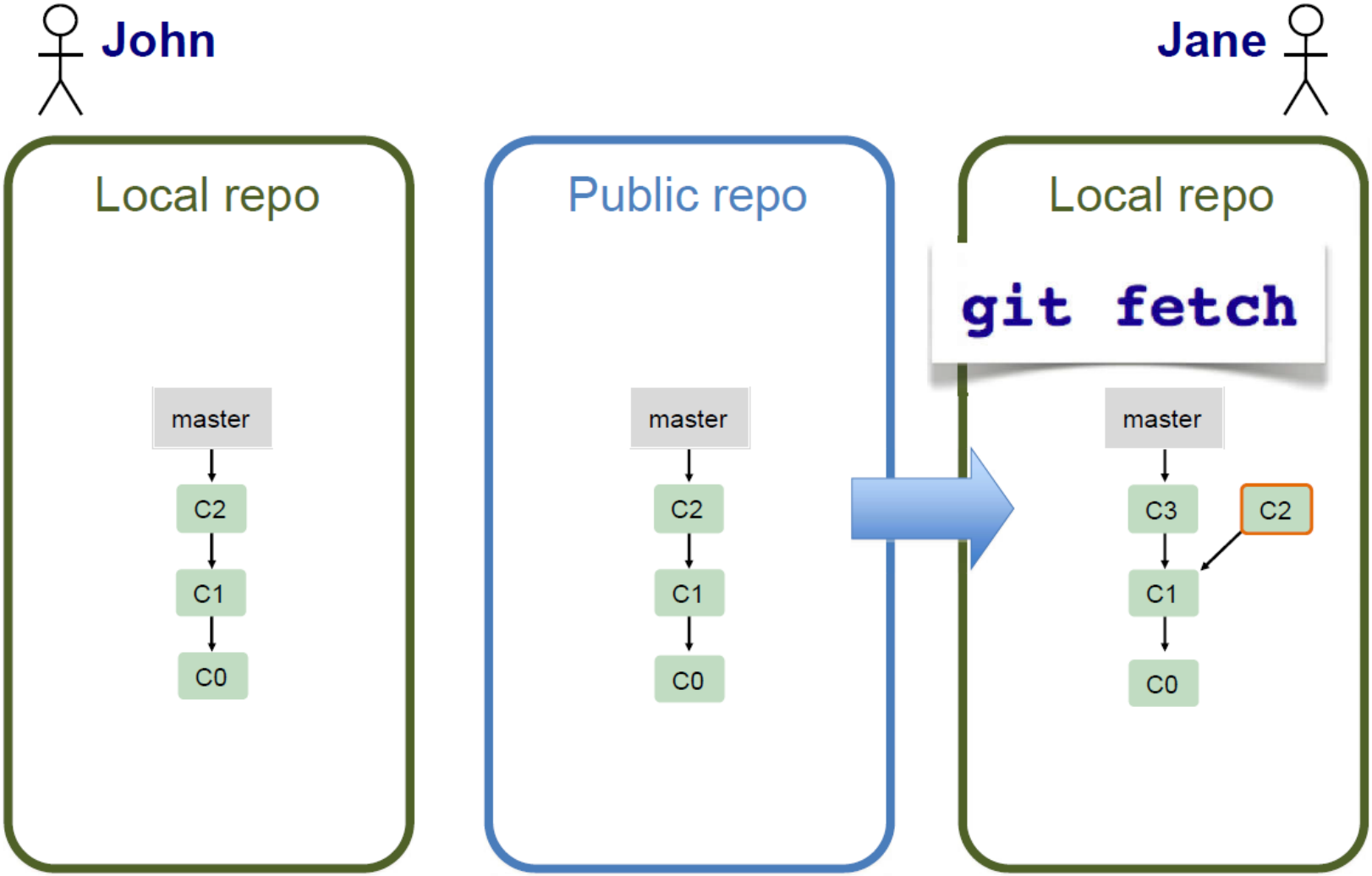

John
Jane
Local repo
Public repo
Local repo
git clone …
master
C1
C0
master
C1
C0
master
C1
C0
git clone …

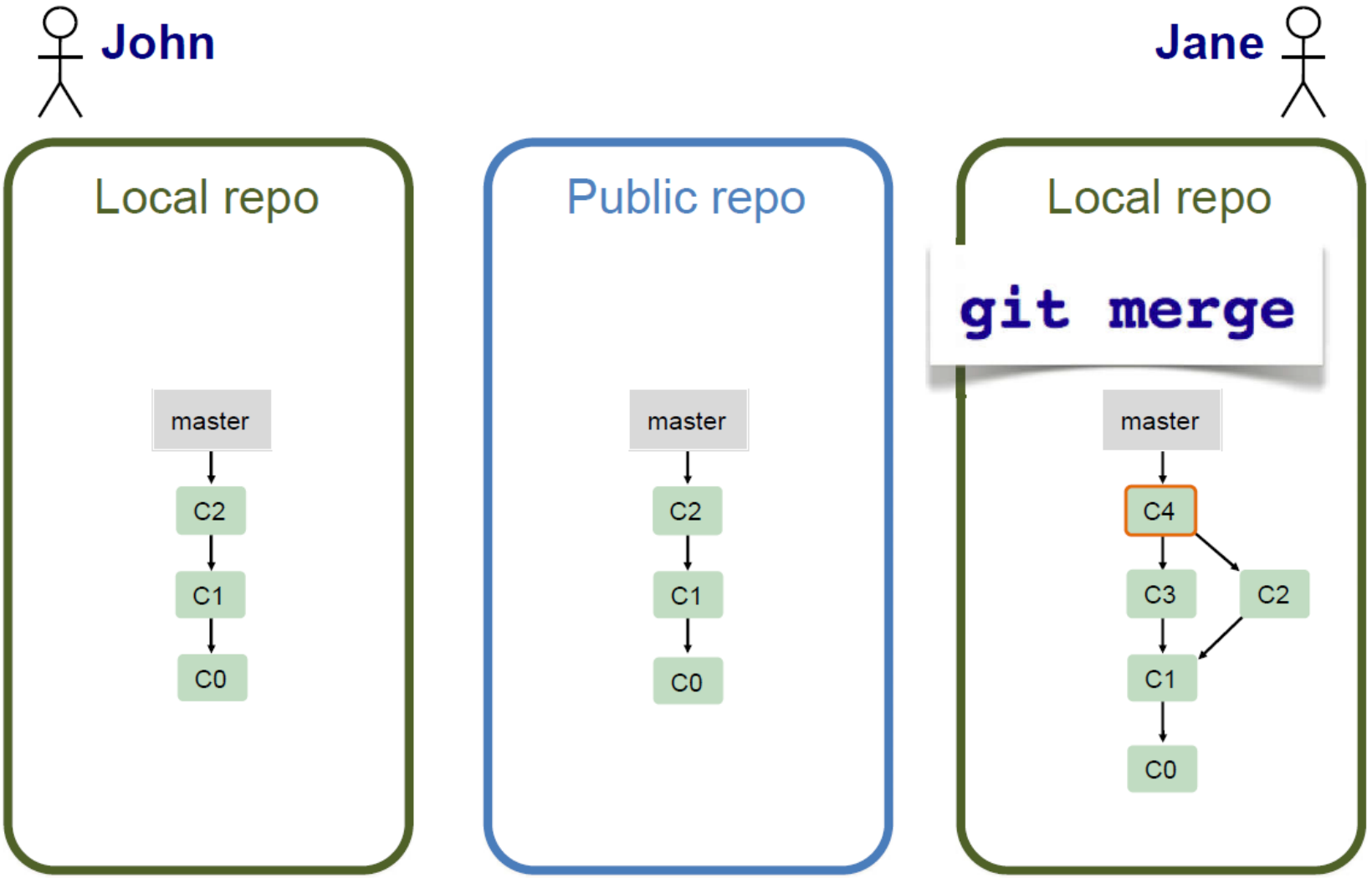

John
Jane
Local repo
Public repo
Local repo
master
C2
C1
C0
master
C1
C0
master
C3
C1
C0
git add …
git commit …
git add …
git commit …



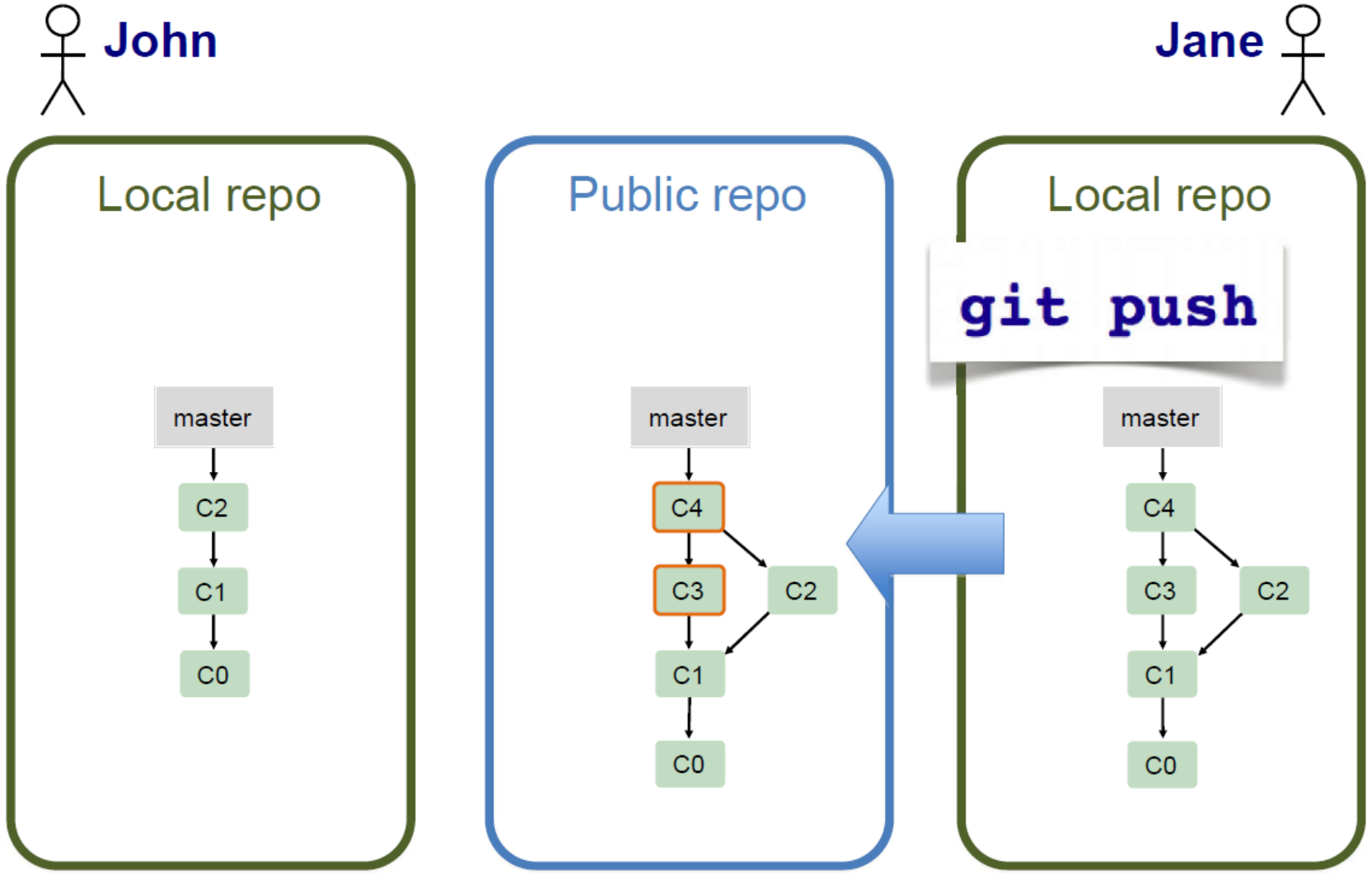
(nothing new to pull)

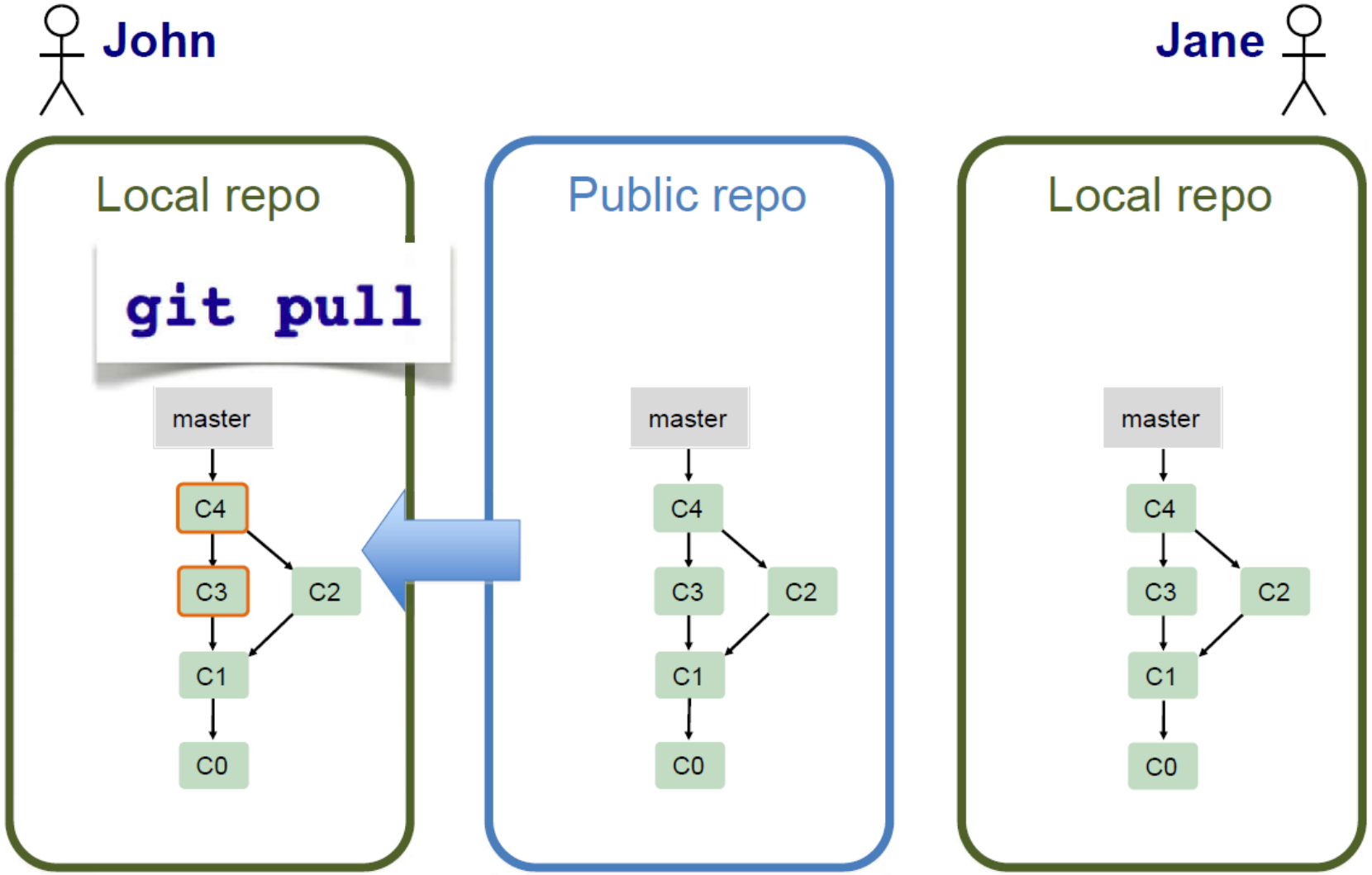

John
Jane
Local repo
Public repo
Local repo
git push
master
C2
C1
C0
master
C2
C1
C0
master
C3
C1
C0


John
Jane
Local repo
master
C2
C1
C0
Public repo
master
C2
C1
C0
git fetch
Local repo
master
C3
C2
C1
C0



NB: **git pull** = fetch + merge

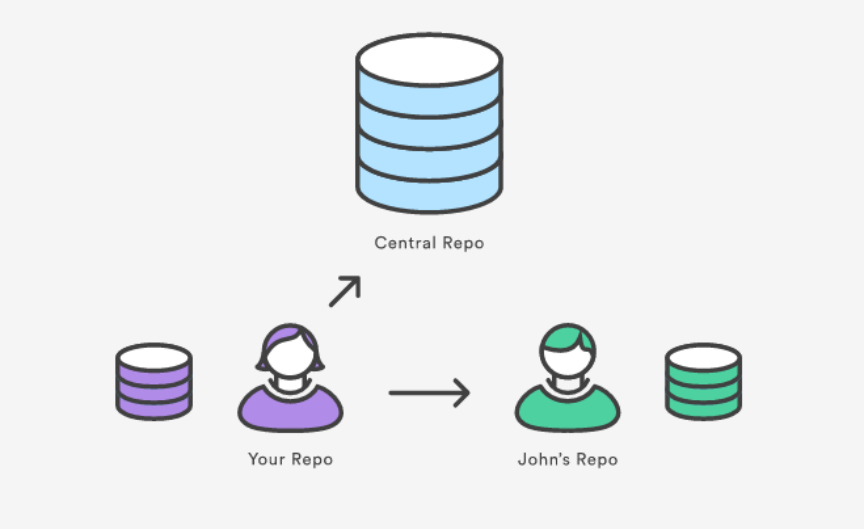

John
Jane
Local repo
Public repo
Local repo
git push
master
C2
C1
C0
master
C4
C3
C2
C1
C0
master
C4
C3
C2
C1
C0


John
Jane
Local repo
git pull
master
C4
C3
C2
C1
C0
Public repo
master
C4
C3
C2
C1
C0
Local repo
master
C4
C3
C2
C1
C0

GIT Remote

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer's repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



GIT Remote - usage

$ git remote

List the remote connections you have to other repositories.

$ git remote -v

Same as the above command, but include the URL of each connection.

$ git remote add <name> <url>

Create a new connection to a remote repository. After adding a remote, you'll be able to use <name> as a convenient shortcut for <url> in other Git commands.

$ git remote rm <name>

Remove the connection to the remote repository called <name>.

$ git remote rename <old-name> <new-name>

Rename a remote connection from <old-name> to <new-name>.

GIT fetch

$ git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

$ git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

To approve the changes and merge them into your local master branch with the following commands:

$ git checkout master

$ git log origin/master

Then we can use git merge origin/master

$ git merge origin/master

The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.

GIT Pull

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with git fetch followed by git merge, but git pull rolls this into a single command.

$ git pull <remote>

Above command will fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as:

$ git fetch <remote>

followed by

$ git merge origin/<current-branch>

*Note: You can think of **"git pull"** as Git's version of **"svn update"***

GIT Push

Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.

$ git push <remote> <branch>

Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository.

$ git push <remote> --force

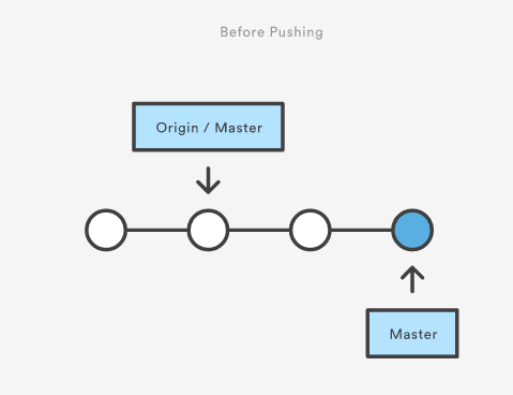
Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you're absolutely sure you know what you're doing.

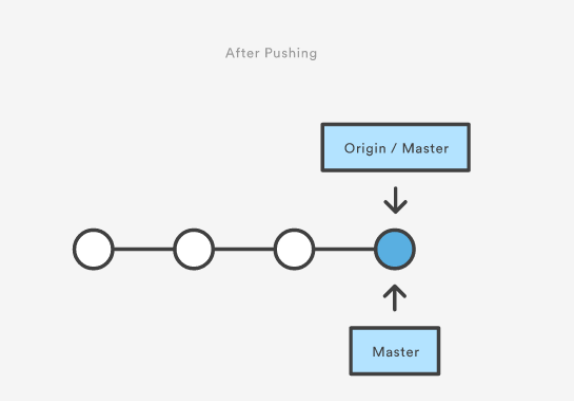
$ git push <remote> --all

Push all of your local branches to the specified remote.

GIT Push

The most common use case for git push is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.





The above diagram shows what happens when your local master has progressed past the central repository's master and you publish changes by running git push origin master. Notice how git push is essentially the same as running git merge master from inside the remote repository.

Command Summary – Collaboration

1. git remote
2. git fetch
3. git merge
4. git pull
5. git push

Understanding Branching in GIT

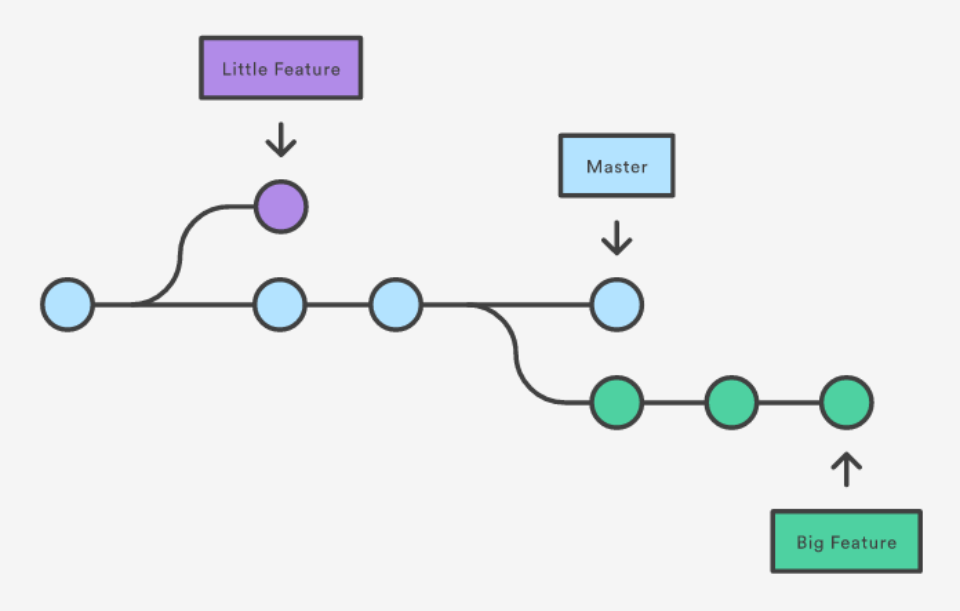
GIT Branch

A branch represents an independent line of development. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The git branch command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

GIT Branch

In Git, branches are a part of your everyday development process. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes sure that unstable code is never committed to the main code base, and it gives you the chance to clean up your feature's history before merging it into the main branch.



GIT Branch - usage

$ git branch

List all of the branches in your repository.

$ git branch <branch>

Create a new branch called <branch>. This does not check out the new branch.

$ git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

$ git branch -D <branch>

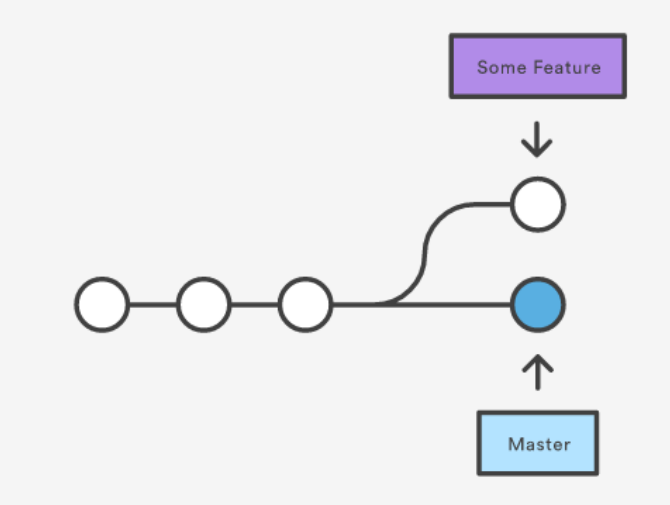
Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

$ git branch -m <branch>

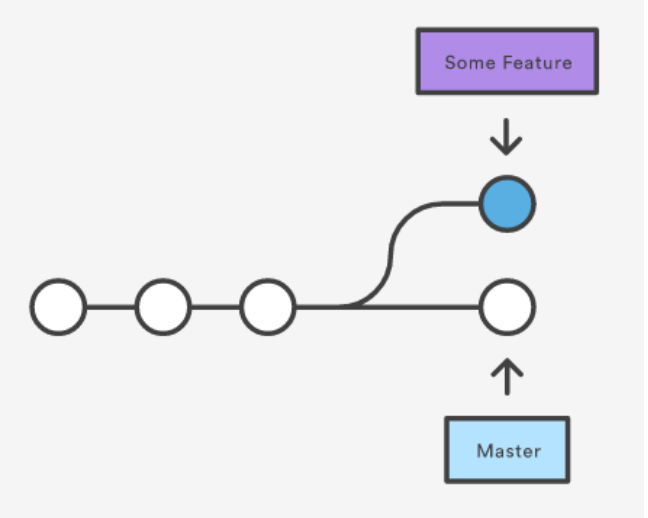
Rename the current branch to <branch>.

GIT Checkout

git checkout works hand-in-hand with git branch. When you want to start a new feature, you create a branch with git branch, then check it out with git checkout. You can work on multiple features in a single repository by switching between them with git checkout.



$ git checkout master



$ git checkout some feature

GIT Checkout - usage

$ git checkout <existing-branch>

Check out the specified branch, which should have already been created with git branch. This makes <existing-branch> the current branch, and updates the working directory to match.

$ git checkout -b <new-branch>

Create and check out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new-branch>

$ git checkout -b <new-branch> <existing-branch>

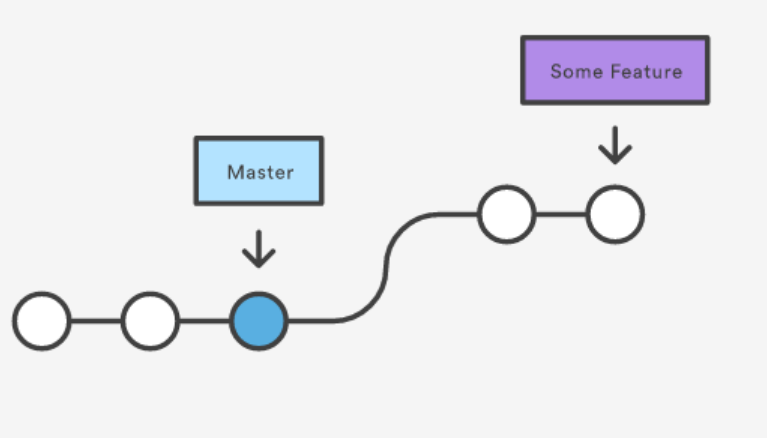
Same as the above invocation, but base the new branch off of <existing-branch> instead of the current branch.

GIT Merge

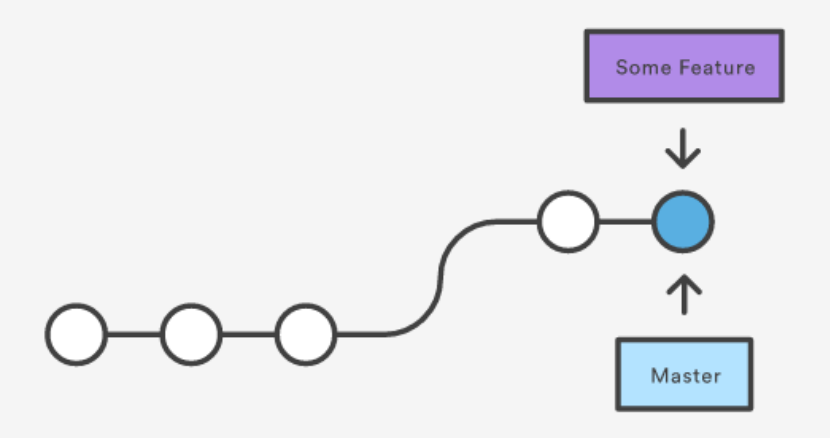
Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

$ git merge <branch>

Merge the specified branch into the current branch.



Before merging



After merging

Best Practices
And
Tips and Tricks

Commit/Push Best practices

Commit as often as you build/compile

- ✓ *a good commit diff, like a good method, should fit on your screen*
- ✓ *don't worry if it's not good enough (no one can see it until you push)*
- ✓ *you might want to get back here (the next thing you do might break)*
- ✓ *you can always squash later*
- ✓ *don't forget to push when you want other people to see your work*

Branch for every complete functional set/bug fix/feature

- ✓ *keeping functionality separate makes life easier later*
- ✓ *feel free to branch retroactively (but not what svn means by that)*

Keeping it simple

If you:

- Make sure you are current with the central repository
- Make some improvements to your code
- Update the central repository before anyone else does

Then you don't have to worry about resolving conflicts or working with multiple branches. All the complexity in git comes from dealing with above mentioned.

Therefore:

- Make sure you are up-to-date before starting to work
- Commit and update the central repository frequently

If you need help: https://help.github.com/

Other useful commands

$ git status

Shows changes in you working directory and staging area.

$ git log

Shows you the history of where you are and other useful information about the repository.

$ git checkout

Changes the checked out branch to the specified one.

You can use the '-b' flag to checkout to make a new local branch and switch your working directory to it at the same time

GIT Aliases

With the help of aliases, you can shorten long and commonly used commands just like in the example below:

Original GIT command:

```
$ git log --pretty=oneline –graph
```

Create an alias:

```
$ git config --global alias.lg "log --pretty=oneline --graph"
```

Now the new alias can be used instead of original git command:

```
$ git lg
```

Configuration Files

3 Config files:

/etc/gitconfig → all users, repositories (--system)
~/.gitconfig → one user, all repo (--global)
[repo]/.git/config → specific to repository (default)

List all config values:

```
$ git config --list
```

When I say I hate CVS with a passion, I have to also say that if there are any SVN [Subversion] users in the audience, you might want to leave. Because my hatred of CVS has meant that I see Subversion as being the most pointless project ever started. The slogan of Subversion for a while was "CVS done right", or something like that, and if you start with that kind of slogan, there's nowhere you can go. There is no way to do CVS right.

*--**Linus Torvalds**, as quoted in Wikipedia*