VERSION CONTROL SYSTEMS - GIT

Rafał Roppel

software
development
academy

PREREQUISITES

HAVING INSTALLED GIT

You can download it from:
https://git-scm.com/downloads

On Linux (RPM-based)

```
$ sudo dnf install git-all
```

On Linux (Debian-based)

```
$ sudo apt-get install git-all
```

On Mac

```
$ brew install git
```

HAVING INSTALLED

or

HAVING AN ACCOUNT ON

or

INTRODUCTION TO VERSION CONTROL

VERSION CONTROL

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

If you are a programmer and want to keep every version of files, a Version Control System (VCS) is a very wise thing to use.

VERSION CONTROL

It allows you to:

- revert selected files back to a previous state
- revert the entire project back to a previous state
- compare changes over time
- see who last modified something that might be causing a problem
- who introduced an issue and when, and more.

LOCAL VERSION CONTROL SYSTEMS

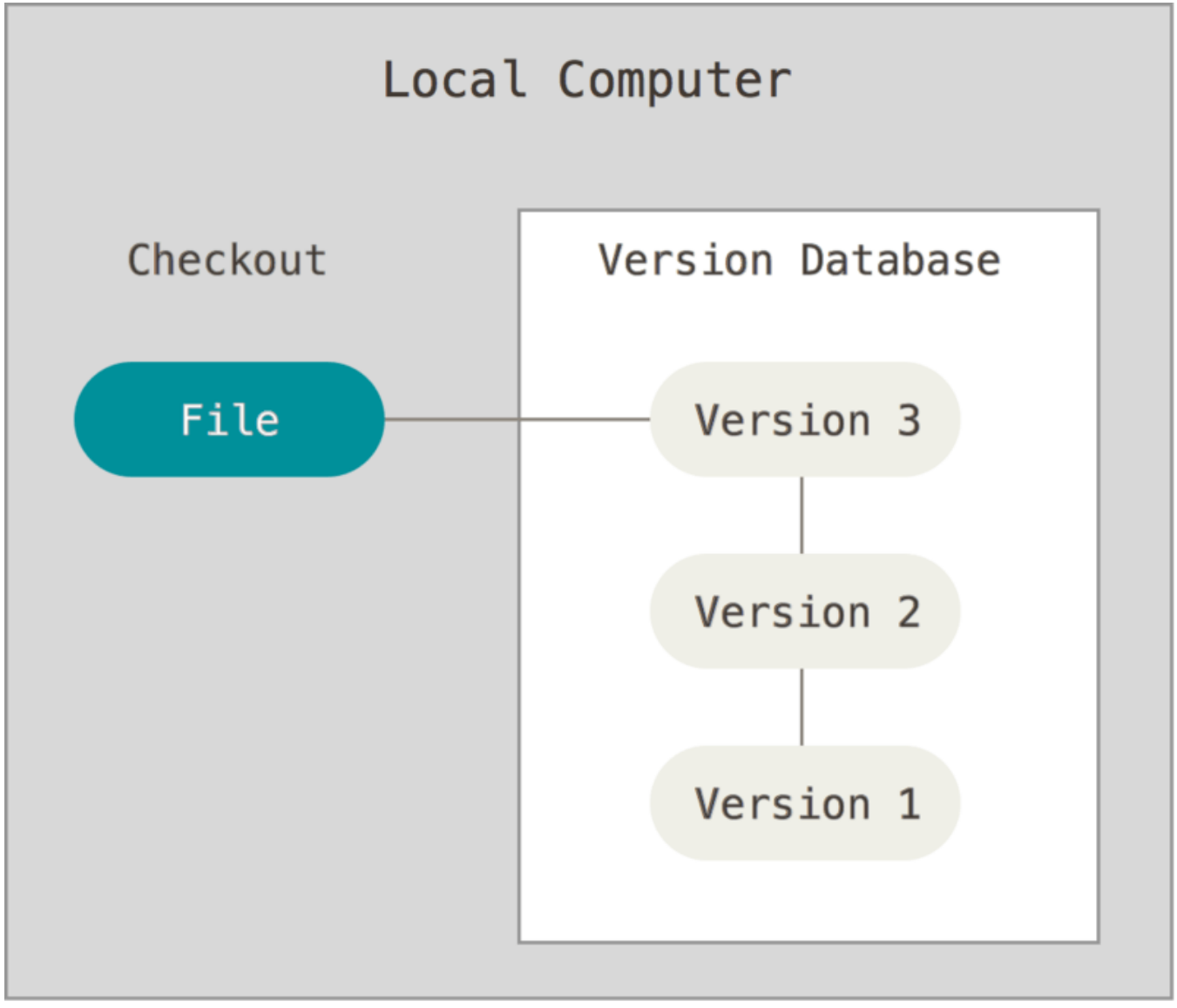
People sometimes copy files into another directory. This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

LOCAL VERSION CONTROL SYSTEMS

One of the more popular VCS tools was a system called RCS (Revision Control System).

RCS works by keeping patch sets (the differences between files) in a special format on disk. It can then re-create what any file looked like at any point in time by adding up all the patches.



Source: Getting Started - About Version Control

CENTRALIZED VERSION CONTROL SYSTEMS

The next major issue that people encounter is that they need to collaborate with developers on other systems.

To deal with this problem, CVCSs (Centralized Version Control Systems) were developed.

CENTRALIZED VERSION CONTROL SYSTEMS

These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place.

For many years, this has been the standard for version control.

CENTRALIZED VERSION CONTROL SYSTEMS

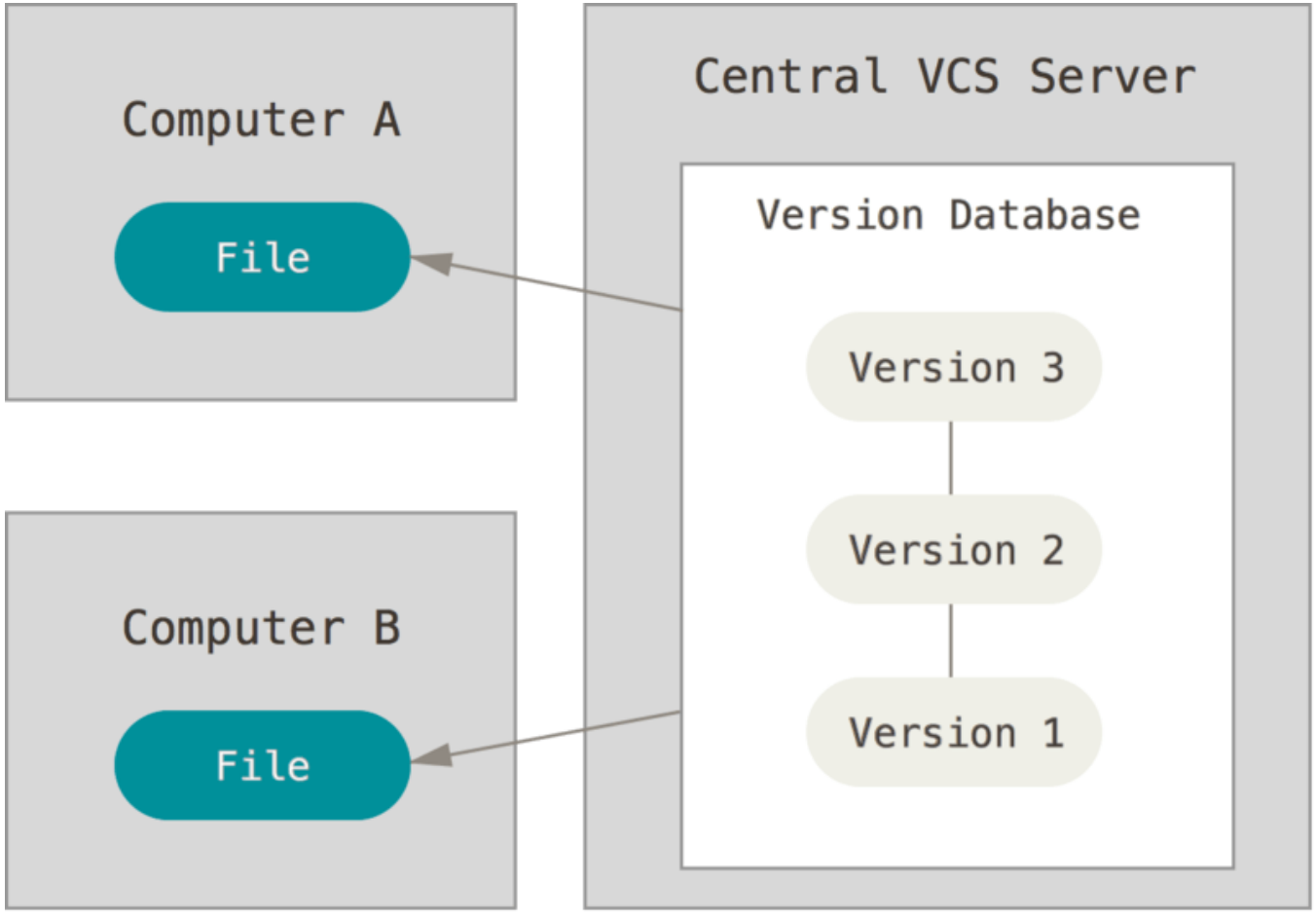
This approach offers many advantages, especially over local VCSs.

- everyone knows to a certain degree what everyone else on the project is doing
- administrators have fine-grained control over who can do what
- it’s far easier to administer a CVCS than it is to deal with local databases on every client

CENTRALIZED VERSION CONTROL SYSTEMS

This approach also has some serious downsides.

- the centralized server represents the single point of failure
- when that server goes down, then nobody can collaborate at all or save versioned changes to anything they're working on
- whenever you have the entire history of the project in a single place, you risk losing everything



Source: Getting Started - About Version Control

DISTRIBUTED VERSION CONTROL SYSTEMS

In a DVCS (such as Git, Mercurial, Bazaar), clients don't just check out the latest snapshot of the files. They fully mirror the repository, including its full history.

When any server dies, any of the client repositories can be copied back up to the server to restore it.

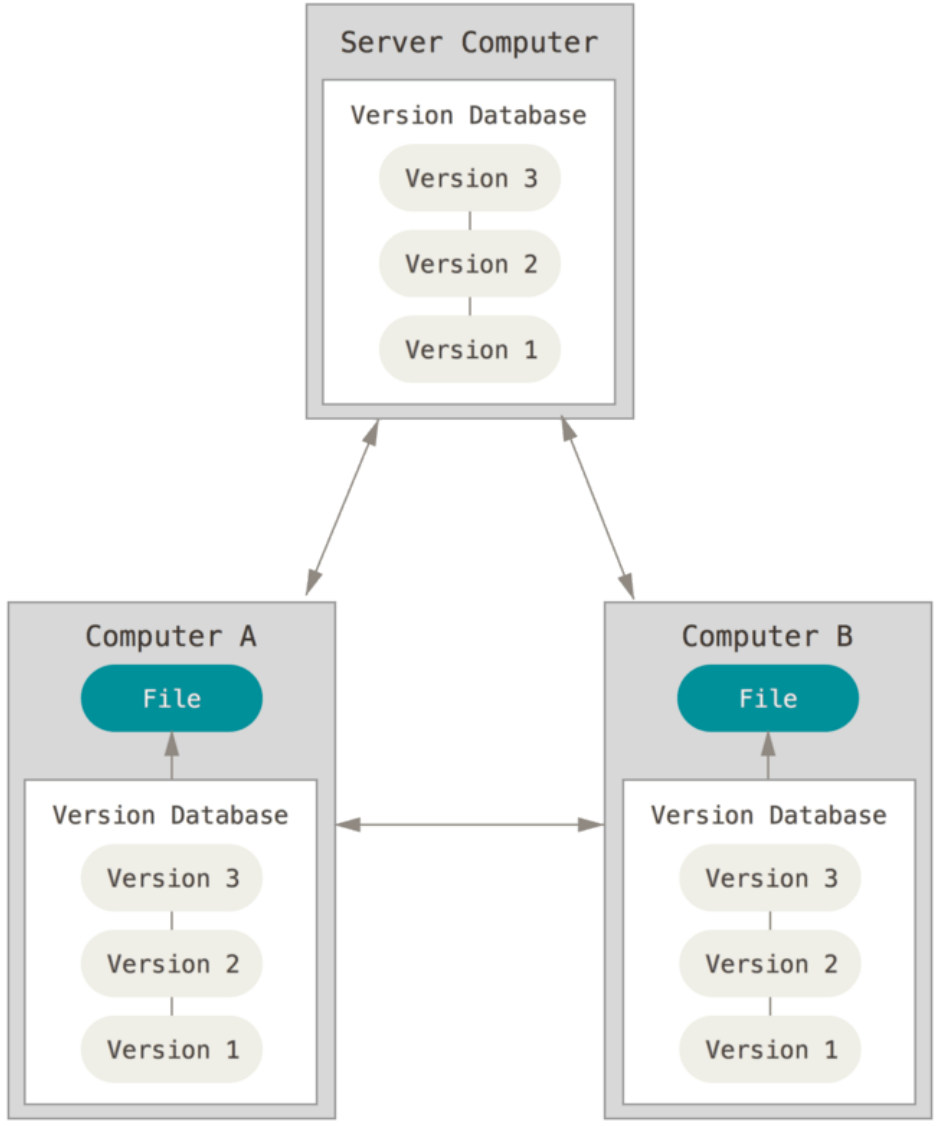
Every clone is really a full backup of all the data.

DISTRIBUTED VERSION CONTROL SYSTEMS

Many of these systems deal pretty well with having several remote repositories they can work with

You can collaborate with different groups of people in different ways simultaneously within the same project.

This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.



Source: Getting Started - About Version Control

A HISTORY OF GIT

The Linux kernel is an open source software project of fairly large scope.

For most of the lifetime of the Linux kernel maintenance, changes to the software were passed around as patches and archived files.

In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

A HISTORY OF GIT

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked.

This prompted the Linux development community (especially Linus Torvalds - the creator of Linux) to develop their own tool.

GIT IN NUTSHELL

- snapshots, not differences
- nearly every operation is local
- Git has integrity
- Git generally only adds data
- Git has three main states

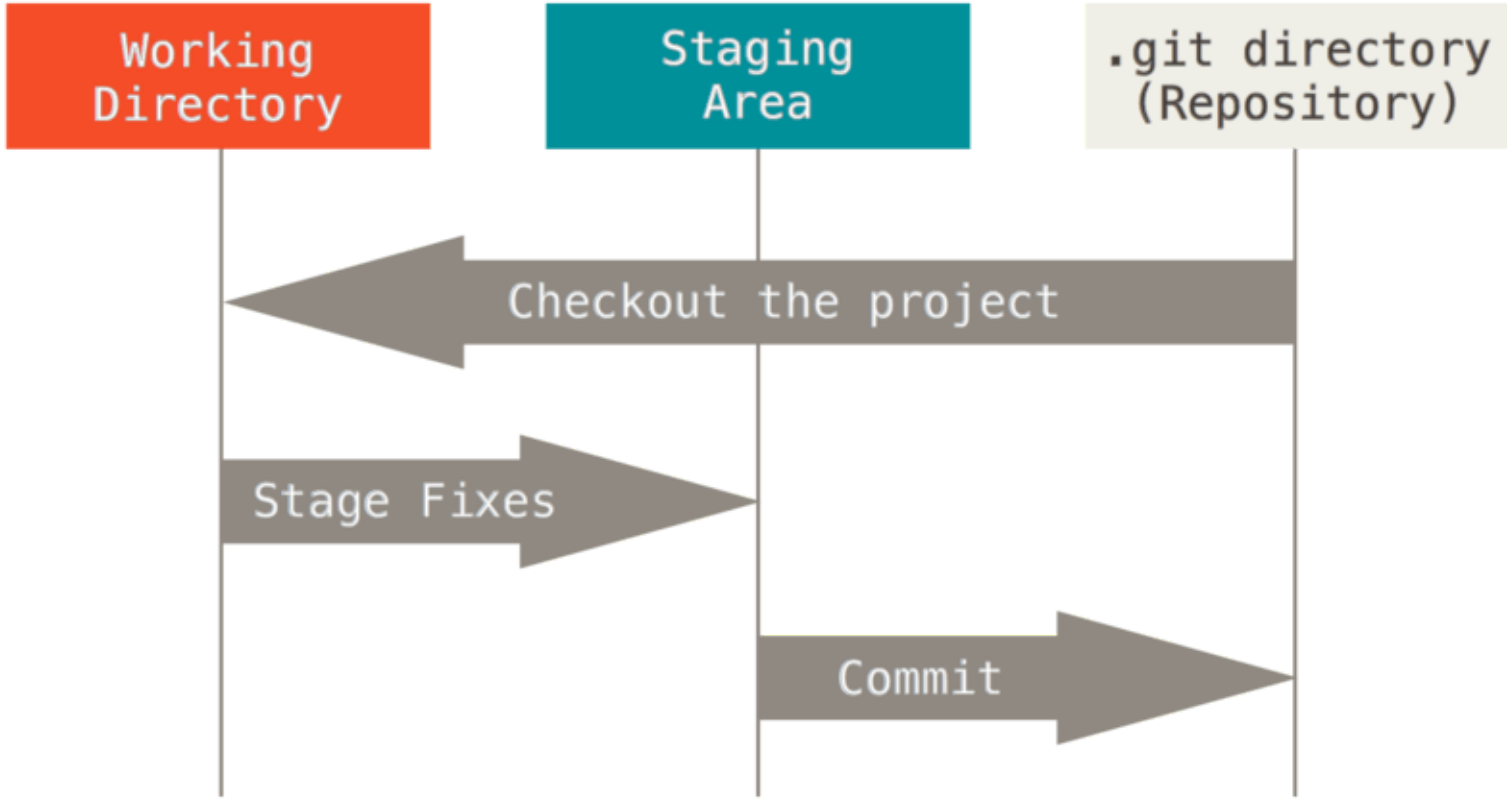
INTRODUCTION TO VERSION CONTROL

WHAT HAVE WE LEARNED ABOUT VERSION CONTROL

- You should have a basic understanding of what Git is
- How it’s different from any centralized version control systems
- You learned a short history of Git

GIT BASICS

WORKING WITH GIT



Source: Getting Started - Git Basics

GETTING A GIT REPOSITORY

You can obtain a Git repository in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository
2. You can *clone* an existing Git repository from elsewhere

INITIALIZING A NEW REPOSITORY

for Windows
for Linux
for Mac

```
$ cd /c/<user>/my_project
```

```
$ cd /home/<user>/my_project
```

```
$ cd /Users/<user>/my_project
```

and type:

```
$ git init
```

CLONING AN EXISTING REPOSITORY

```
$ git clone <url>
```

For example:

```
$ git clone https://github.com/google/guava
```

or:

```
$ git clone https://github.com/google/guava myguava
```

EXERCISES

- `git init`
- `git clone`

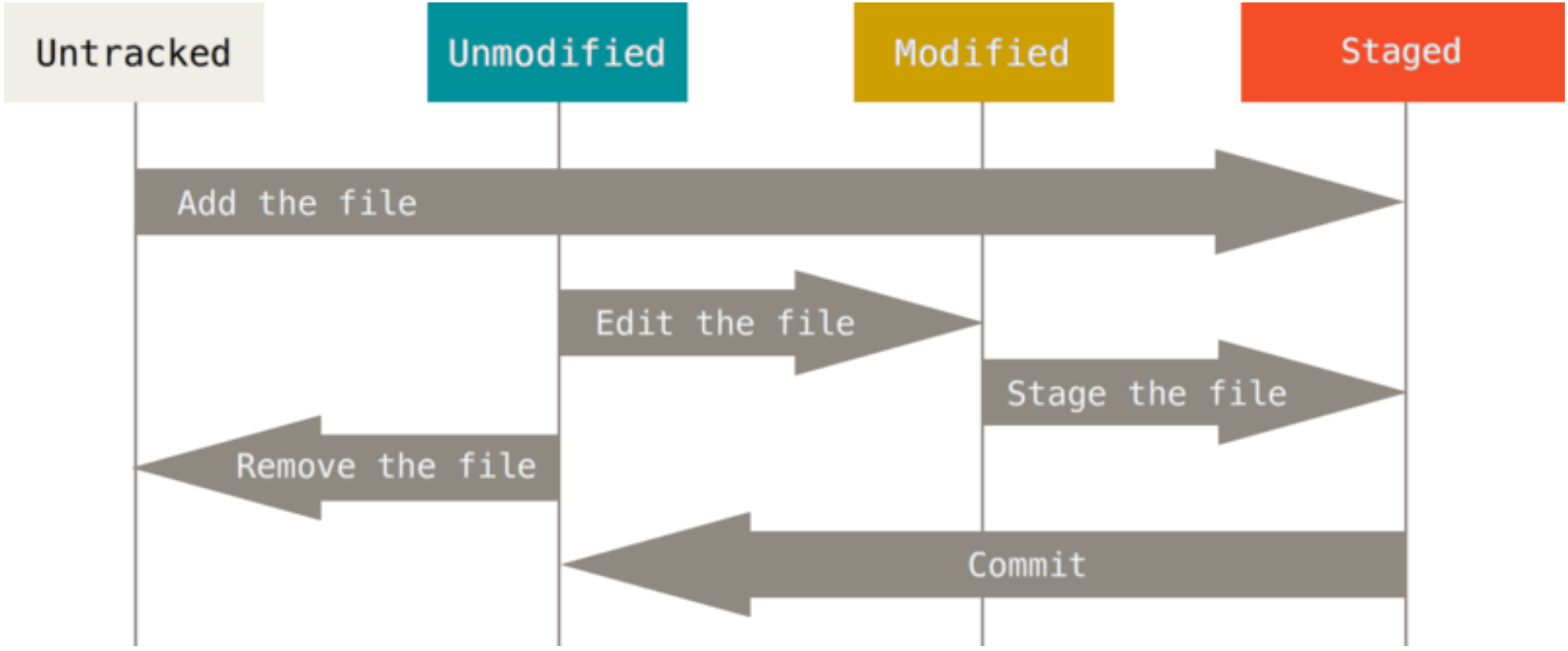
ADDING CHANGES TO THE REPOSITORY

Remember that 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, files that Git knows about.

Untracked files are any files in your working directory that were not in your last snapshot and are not in your staging area.

THE LIFECYCLE OF THE STATUS OF YOUR FILES



Source: Git Basics - Recording Changes to the Repository

CHECKING THE STATUS OF FILES

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

CHECKING THE STATUS OF FILES

```
$ echo "Hello, World" > README
```

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README

nothing added to commit but untracked files present (use "git add" to track)
```

TRACKING NEW FILES

```
$ git add README
```

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README
```

COMMITTING CHANGES

Now you can commit your changes. The simplest way to commit is to type:

```
$ git commit
```

To put the commit message you can use `-m flag`:

```
$ git commit -m "<message>"
```

for example:

```
$ git commit -m "I added Hello World text"
[master (root-commit) cffafa4] I added Hello World text
 1 file changed, 1 insertion(+)
 create mode 100644 README
```

MOVING FILES

If you want to rename a file in Git, you can type:

```
$ git mv <file_name> <new_file_name>
```

For example:

```
$ git mv README README.md
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README -> README.md
```

REMOVING FILES

To remove a file from Git, you have to:

- remove it from your tracked files
- commit

The `git rm` command does that, and also removes the file from your working directory

REMOVING FILES

```
$ git rm README
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    README
```

EXERCISES

- `git status`
- `git add`
- `git commit`
- `git mv`
- `git rm`
- `git diff`

VIEWING THE COMMIT HISTORY

After you have created several commits, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

VIEWING THE COMMIT HISTORY

```
$ git log
commit 594e3b73fe99b66110920c5fd7f2a6ddb95f2e6d
Author: cpovirk <cpovirk@google.com>
Date:   Tue Feb 6 08:44:22 2018 -0800

    Hide class used from Google-internal test.

    RELNOTES=n/a

    -------------
    Created by MOE: https://github.com/google/moe
    MOE_MIGRATED_REVID=184686532

commit 198c384ccdb14af182fea9b484963ba6c249fb7d
```

UNDOING THINGS

The common undos takes place when you commit too early and forget to add some files, or you want to change commit message.

If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the `--amend` option.

UNDOING THINGS

For example:

```
$ git add forgotten_file.txt
```

```
$ git commit --amend
[master 8c42249] I added Hello World text
 Date: Mon Feb 5 21:51:55 2018 +0100
 2 files changed, 2 insertions(+)
 create mode 100644 README
 create mode 100644 forgotten_file.txt
```

UNSTAGING A STAGED FILE

You've changed one file and create another, and you want to commit them as two separate changes, but you accidentally type git `add *` and stage them both.

```
$ git add *
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README
        new file:   info.txt
```

UNSTAGING A STAGED FILE

You can use `git reset HEAD <file>...` to unstage.

```
$ git reset HEAD README
Unstaged changes after reset:
M       README
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   info.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README
```

UNMODIFYING A MODIFIED FILE

If you don't want to keep your changes to the README file, you can easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned) using `git checkout` command.

UNMODIFYING A MODIFIED FILE

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README

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

```
$ git checkout -- README
```

```
$ git status
On branch master
nothing to commit, working directory clean
```

EXERCISES

- `git log|blame|show`
- `git commit --amend`
- `git reset`
- `git revert`
- `git clean`
- `git checkout`

WORKING WITH REMOTES

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

SHOWING REMOTE REPOSITORIES

To see which remote servers you have configured, you can run the `git remote` command.

```
$ git clone https://github.com/google/guava
Cloning into 'guava'...
remote: Counting objects: 213028, done.
remote: Total 213028 (delta 0), reused 0 (delta 0), pack-reused 213027
Receiving objects: 100% (213028/213028), 199.46 MiB | 4.84 MiB/s, done.
Resolving deltas: 100% (153880/153880), done.
Checking connectivity... done.
```

```
$ cd guava
```

```
$ git remote
origin
```

SHOWING REMOTE REPOSITORIES

You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote.

```
$ git remote -v
origin  https://github.com/google/guava (fetch)
origin  https://github.com/google/guava (push)
```

ADDING REMOTE REPOSITORY

To add a new remote Git repository as a shortname you can reference easily, run `git remote add <shortname> <url>`

```
$ git remote add myguava https://github.com/rafos/guava
```

```
$ git remote -v
myguava https://github.com/rafos/guava (fetch)
myguava https://github.com/rafos/guava (push)
origin  https://github.com/google/guava (fetch)
origin  https://github.com/google/guava (push)
```

PUSHING TO REMOTE REPOSITORY

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push <remote> <branch>`.

```
$ git push myguava master
Counting objects: 91561, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (16213/16213), done.
Writing objects: 100% (91561/91561), 66.10 MiB | 2.08 MiB/s, done.
Total 91561 (delta 47768), reused 91560 (delta 47767)
remote: Resolving deltas: 100% (47768/47768), done.
To https://github.com/rafos/guava
 * [new branch]      master -> master
```

INSPECTING A REMOTE REPOSITORY

If you want to see more information about a particular remote, you can use the `git remote show <remote>` command.

```
$ git remote show myguava
* remote myguava
  Fetch URL: https://github.com/rafos/guava
  Push  URL: https://github.com/rafos/guava
  HEAD branch: master
  Remote branch:
    master tracked
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

REMOVING REMOTE REPOSITORY

If you want to remove a remote repository you can either use `git remote remove` or `git remote rm`.

```
$ git remote
myguava
origin
```

```
$ git remote remove myguava
```

```
$ git remote
origin
```

EXERCISES

- `git clone`
- `git remote`
- `git push`
- `git fetch`
- `git pull`

GIT BASICS

WHAT HAVE WE LEARNED ABOUT GIT BASICS

- creating or cloning a repository
- making changes
- staging and committing changes
- viewing the history of the changes

GIT BRANCHING

BRANCHES IN A NUTSHELL

Almost every VCS has some form of branching support.

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast.

CREATING A NEW BRANCH

To create a new branch, you should use `git branch <branch name>` command.

```
$ git branch testing
```

```
$ git status
On branch master
nothing to commit, working directory clean
```

Why?

CREATING A NEW BRANCH

Because you are still on the *master* branch

```
$ git log --oneline --decorate
8c42249 (HEAD -> master, testing) I added Hello World text
```

You can see the "master" and "testing" branches that are right there next to the `8c42249` commit.

SWITCHING BRANCHES

To switch to an existing branch, you run the `git checkout <branch name>` command.

```
$ git checkout testing
Switched to branch 'testing'
```

And then

```
$ git log --oneline --decorate
8c42249 (HEAD -> testing, master) I added Hello World text
```

`HEAD` to points to the *testing* branch.

CREATING AND SWITCHING AT THE SAME TIME

This:

```
$ git checkout -b testing
Switched to a new branch 'testing'
```

Is shorthand for:

```
$ git branch testing
```

```
$ git checkout testing
Switched to branch 'testing'
```

RENAMING BRANCHES

To rename branch you can use `git branch -m <branch name> <new branch name>` command.

```
$ git branch -m testing another
```

MERGING BRANCHES

If you want to merge your *another* branch into *master*, you cant do it using `git merge` command.

First, let's see all the files from *master* branch.

```
$ git checkout master
Switched to branch 'master'
```

```
$ ls
README    forgotten_file.txt
```

MERGING BRANCHES

And all files from *another* branch.

```
$ git checkout another
Switched to branch 'another'
```

```
$ ls
Phone.java    README    User.java    forgotten_file.txt
```

Then, switch back to *master* branch.

```
$ git checkout master
Switched to branch 'master'
```

MERGING BRANCHES

Merging *another* branch to *master* (current) branch.

```
$ git merge another
Updating 8c42249..f47ea95
Fast-forward
 Phone.java | 0
 User.java  | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Phone.java
 create mode 100644 User.java
```

MERGING BRANCHES

Situation after merge.

```
$ git status
On branch master
nothing to commit, working directory clean
```

```
$ git ls
Phone.java    README    User.java    forgotten_file.txt
```

DELETING BRANCHES

Now you can remove *another* branch.

```
$ git branch -d another
Deleted branch another (was f47ea95).
```

EXERCISES

- `git branch`
- `git log`
- `git merge`
- `git checkout`

GIT BRANCHING

WHAT HAVE WE LEARNED ABOUT GIT BRANCHING

- You should know how to create and switch to new branches
- Switch between branches
- Merge local branches together
- Share your branches by pushing them to a shared server

ADDITIONAL TOPICS

EXERCISES

- `git config`
- `git stash`
- `git tag`
- `.gitignore`
- `.gitkeep`

GRAPHICAL INTERFACES

- Git in GitHub Desktop/BitBucket
- Git in IntelliJ IDEA

FURTHER READING

- Git cheat sheet by GitHub
- Git Immersion
- Become a git guru by Atlassian
- Learn Git Branching
- git-flow cheatsheet by Daniel Kummer