On becoming a Git Novice

By Ralph Vancampenhoudt & Celeste Willems

Version control

Chapter 1

Version control

What is version control?

Simply put, **version control is software that keeps track of every modification made to your code.**

It does this tracking over time, meaning that ***every modification of the code can be compared to previous modifications*** *and can be turned back to that previous state.*

Version control

What is version control?

Version control

Why version control?

When working on code, we - as **developers** - have the following **requirements** and **obligations:**

Version controls fulfills all these requirements!

- ✓ Source **code** is our most valuable asset, we should **treat it with care**
- ✓ We should always have a **backup of our code**
- ✓ We want to make **changes** to source code **over time as a team** in a managed way, with **minimum disruption** to other team members
- ✓ Team members should be able to work on features **individually and concurrently without overwriting** each other's **changes**
- ✓ There should be a **history of changes** that we can trace
- ✓ We should be able to **revert / undo** those **changes**

Why version control?

On the code

With version control, we - as a development team - can inspect and act upon **who** did **what, when**

Furthermore, it provides us with a **versioned backup** of the **entire source code and all files** of our project.

Why version control?

With version control, we - as a development team - can work together **simultaneously** on the same code. When **conflicts arise**, version controls helps to **resolve these conflicts** and allows them to be merged, **without any changes being lost or overwritten**.

Multiple software developers working in a team are continually writing new - and changing existing - source code. When two or more developers make changes in the same file, conflicts can arise.

Version control

Why version control?

Git introduction

Chapter 2

Git introduction

What is Git?

Git is a **version control system**

There are many **categories** of **version control systems**

1. Manually **copying** files to another (time-stamped) directory
2. Local Version Control Systems
3. Centralized Version Control System (**CVCS**)
4. Distributed Version Control System (**DVCS**)

There are many **categories** of **version control systems**

1. Manually **copying** files to another (time-stamped) directory
   - ✓Error-prone
   - ✓Local, no collaboration possible
   - ✓No remote backup (if the disk dies, gone are your code and files)
2. Local Version Control Systems
3. Centralized Version Control System (**CVCS**)
4. Distributed Version Control System (**DVCS**)

There are many **categories** of **version control systems**

1. Manually copying files to another (time-stamped) directory
2. Local Version Control Systems → E.g. RCS
   - ✓ Versioning
   - ✓ Outdated
   - ✓ Local, no collaboration possible
   - ✓ No remote backup (if the disk dies, gone are your code and files)
3. Centralized Version Control System (CVCS)
4. Distributed Version Control System (DVCS)

There are many **categories** of **version control systems**

1. Manually copying files to another (time-stamped) directory
2. Local Version Control Systems
3. Centralized Version Control System (**CVCS**)
   - ✓ Versioning & Collaboration
   - ✓ Remote backup on central repository
   - ✓ One central repository server (single point of failure: offline or corrupt disk)
   - ✓ Clients only *checkout* the latest snapshot (not the complete repository with the entire history)
   - ✓ Is (becoming) outdated (although still often used)
4. Distributed Version Control System (DVCS)

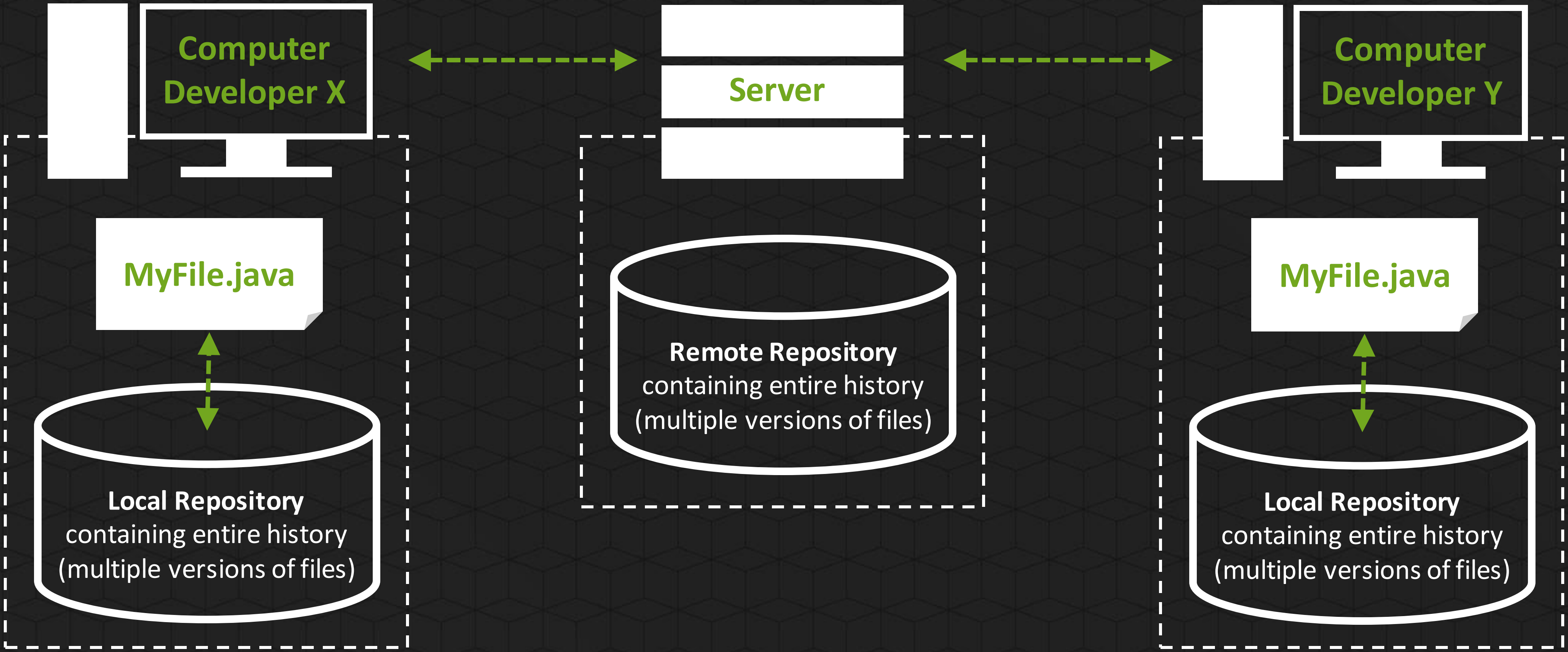
E.g. CVS, Subversion (SVN), Team Foundation Version Control (TFVC)

There are many **categories** of **version control systems**

1. Manually copying files to another (time-stamped) directory
2. Local Version Control Systems
3. Centralized Version Control System (CVCS)
4. Distributed Version Control System (**DVCS**)
   - ✓ Versioning & Collaboration
   - ✓ All clients have a full copy of the complete repository with the entire history
   - ✓ Remote backup on central repository & on all client's hardware!
   - ✓ No single point of failure

E.g. **Git**, Mercurial

The typical high-level **infrastructure** of **DVCSs**


Computer Developer X
MyFile.java
Local Repository containing entire history (multiple versions of files)
Server
Remote Repository containing entire history (multiple versions of files)
Computer Developer Y
MyFile.java
Local Repository containing entire history (multiple versions of files)

Although different VCS tools can vary in use and features, the following **primary benefits** should **always be expected**:

1. **A complete long-term change history** of every file
   Every file has multiple ***versions***
2. **Branching** and **merging**
   Combining and integrating changes
3. **Traceability**
   who did ***what, when***

Git introduction

Why Git?

So, **why Git**? *(1/2)*

- ✓**Fast:** Git does most of the performance-intensive operations on your local machine (as opposed to a remote server in some other systems), and it is built for large repositories from day one (initially built primarily for managing the Linux kernel source code).
- ✓**Free and Open Source Software:** Git is released under an open-source license, meaning that it will always be free for you to use in both your personal and enterprise projects.
- ✓**Large Community:** It is maintained by a large community of contributors, which you'll come to understand, is a really valuable asset of every open source software project.
- ✓**Mature and actively maintained**
- ✓**Secure:** Git ensures cryptographic integrity of your entire codebase.

So, **why Git**? *(2/2)*

- ✓**Widespread use:** Git is one of the most popular version control systems in use right now. Learning Git will give you an advantage in nearly every future software project.
- ✓**Battle tested:** Used internally by large software companies (Google, Facebook, Microsoft, Netflix, etc.) and for big open-source projects (Linux kernel, Android, PostgreSQL, Ruby on Rails,…)
- ✓**Powerful:** Relatively easy to learn thanks to excellent online resources, plus plenty of flexibility/power for advanced users. *Git is challenging to master though…*
- ✓**Distributed:** Git is known as a Distributed Version Control System (DVCS). Essentially, this means that all users own a complete copy of the repository on their local machine, and that you don't even need a server/internet/remote to reap the benefits of version control.

Git basics

Chapter 3

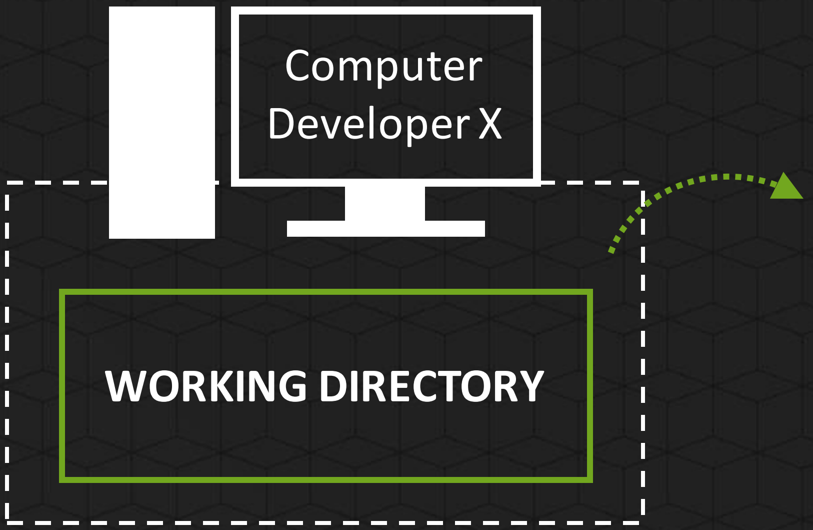
1. Version control
2. Git introduction
3. Git basics local
   - ✓Core components
   - ✓File status lifecycle
   - ✓Commits
   - ✓Basic commands
   - ✓Basic usage
4. Git basics remote

Overview

Git basics

Core components

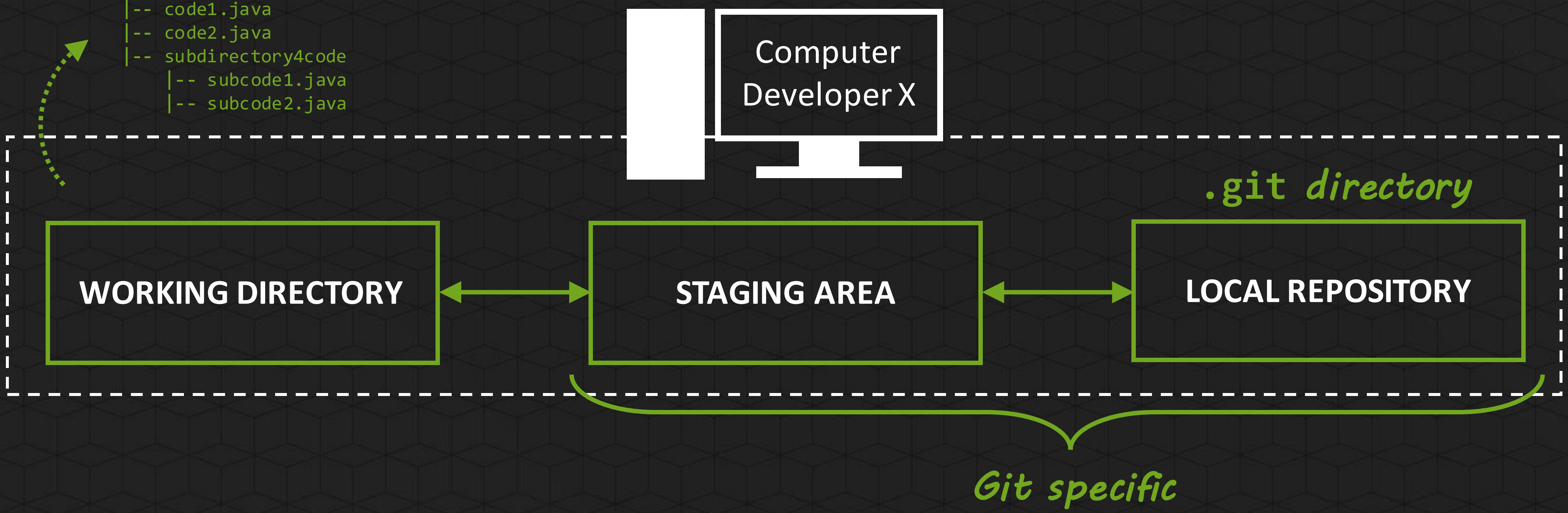
Imagine you have a directory (e.g. `c:\switchfully\project`) that contains a bunch of source code files (*.java / *.cs). The `project` directory is what we call the **working directory, root directory** or **working tree** of your software project.



```
|-- project
    |-- code1.java
    |-- code2.java
    |-- subdirectory4code
        |-- subcode1.java
        |-- subcode2.java
```

To **enable** version control for our project with **Git,** we have to **initialize Git** inside our **working directory**. Once initialized, we will have **3 core components of Git** that work together.

```
|-- project
    |-- code1.java
    |-- code2.java
    |-- subdirectory4code
        |-- subcode1.java
        |-- subcode2.java
```



Core components

- **Working directory (or working tree)**: contains a **single checkout of one version of the project**. These are pulled out of the database located inside the local repository and **placed on disk for you to use**.
- **Staging area**: A concept (part of the index file) contained in your local repository that **contains information** of **what will go into your next commit**. "What files are *staged for commit*"
- **Local repository (or .git directory):** location where Git **stores** the metadata and the ***file-version* database** for your project.

Git basics

Core components

Let's **demonstrate** what we just **covered** *(hands-on in group)*

- ❑ Let's create the following directory structure using CMD

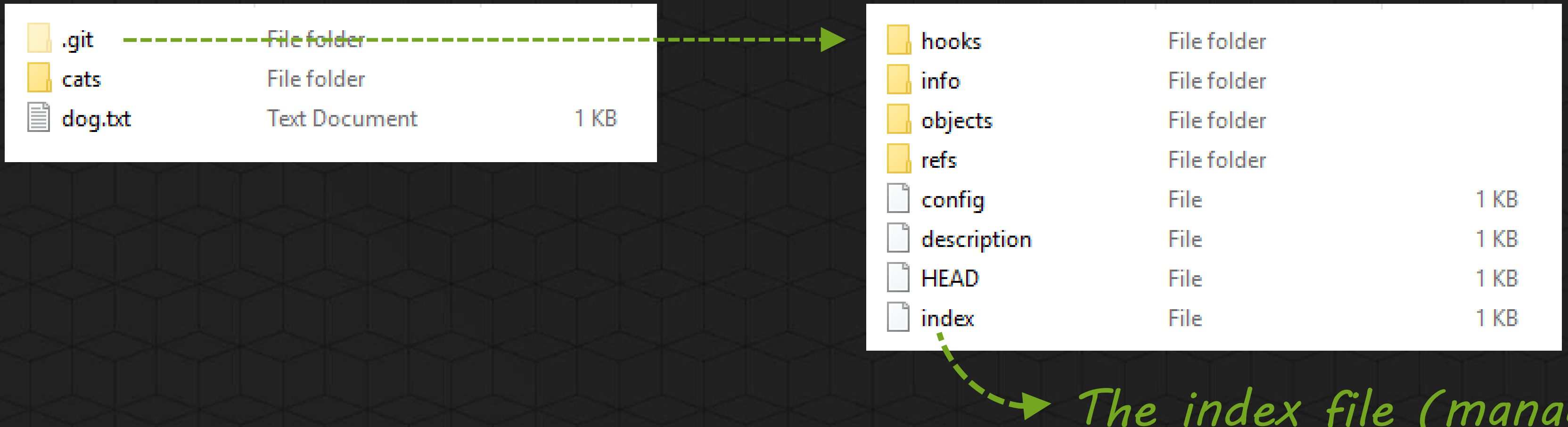
```
|-- catdogproject
    |-- dog.txt
    |-- cats
        |-- cat.txt
```

- ❑ Let's navigate to our working directory using CMD
  - ❑ We're now in our working directory
  - ❑ Files on disk, no version control whatsoever
- ❑ Let's initialize Git for our cat-dog project
  - ❑ > `git init`
  - ❑ Git has to be installed in order for the git command to work

Let’s **demonstrate** what we just **covered**

❑ Inspect the working directory

- ❑ .git directory / local repository is created
- ❑ The index file will be created the moment we add our first file to the stage.
- ❑ We can now start tracking and versioning our files



Working directory

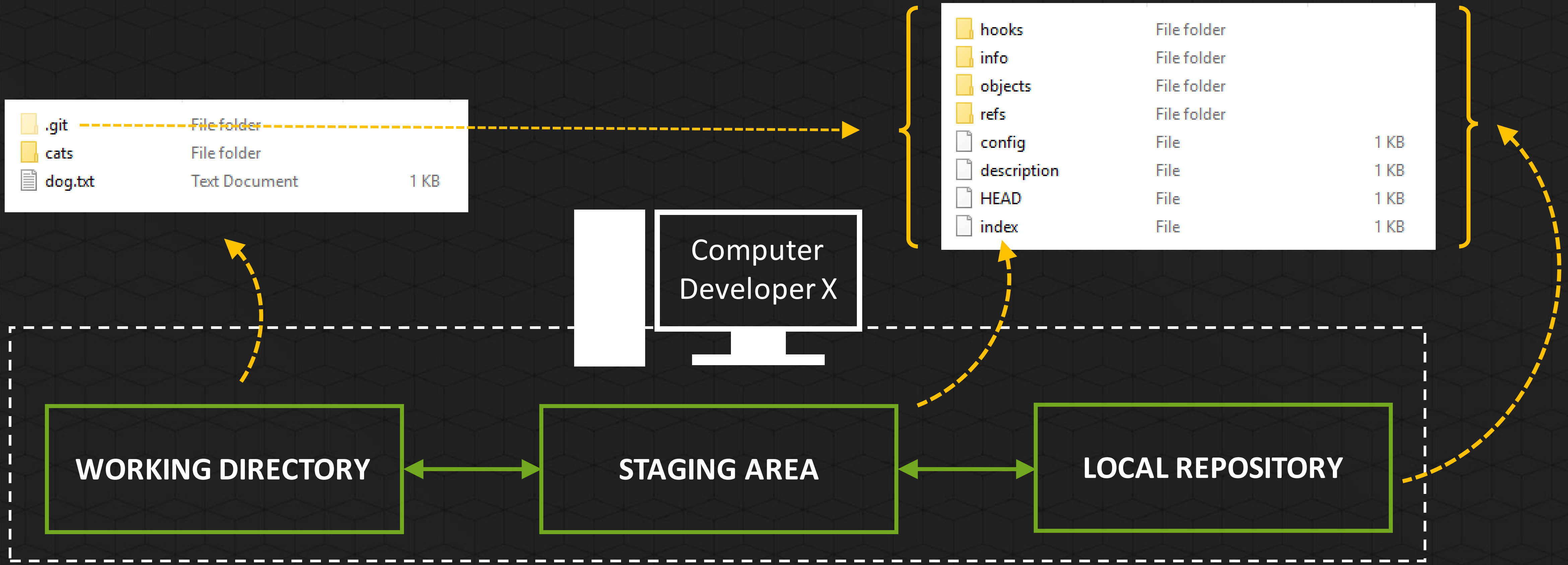
.git	File folder	
cats	File folder	
dog.txt	Text Document	1 KB

.git directory / local repository

hooks	File folder	
info	File folder	
objects	File folder	
refs	File folder	
config	File	1 KB
description	File	1 KB
HEAD	File	1 KB
index	File	1 KB

The index file (managing the staging area)

Let’s **demonstrate** what we just **covered**



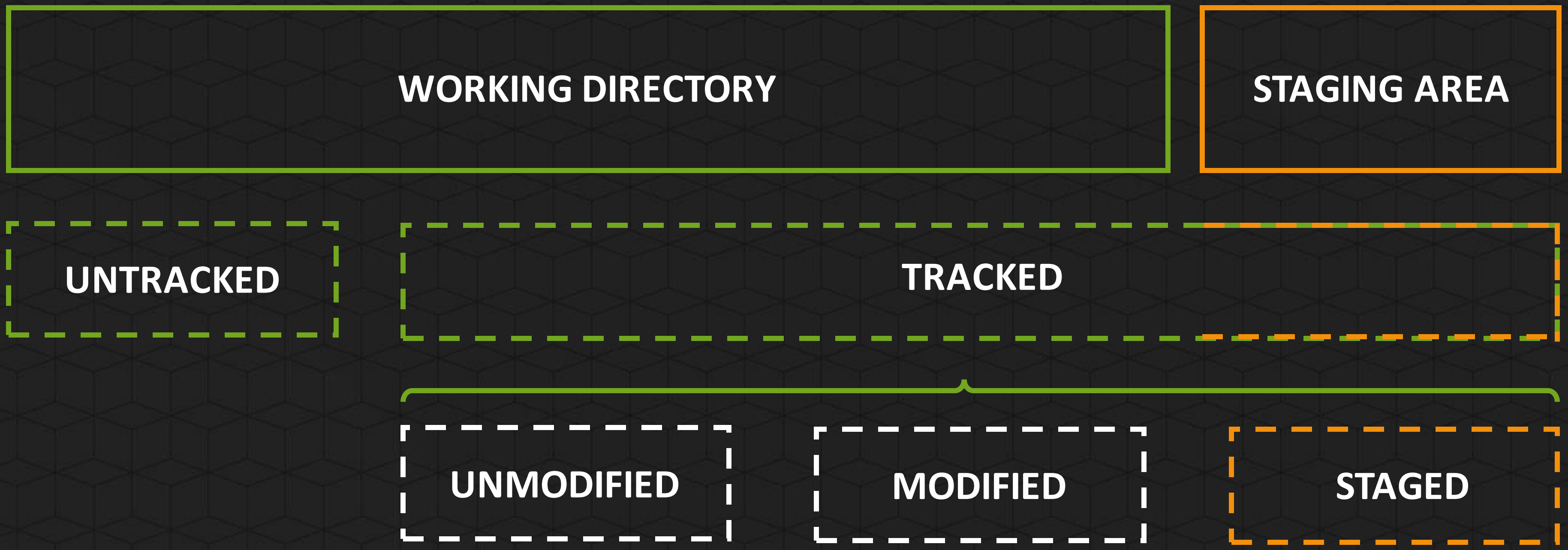
.git	File folder	
cats	File folder	
dog.txt	Text Document	1 KB

hooks	File folder	
info	File folder	
objects	File folder	
refs	File folder	
config	File	1 KB
description	File	1 KB
HEAD	File	1 KB
index	File	1 KB

Git basics

File status lifecycle

Files inside a Git managed **working directory** are either in the **tracked** or **untracked** state. **Tracked files** can be **unmodified**, **modified** or **staged.**



Tracked files: all files that were in your last snapshot (or in your staging area). **Simply put**: all files for which Git performs version control.

Untracked files: all files in your working directory that are not in your staging area and that were not in the latest snapshot. **Simply put**: new (or ignored) files for which Git won't do any versioning (yet).

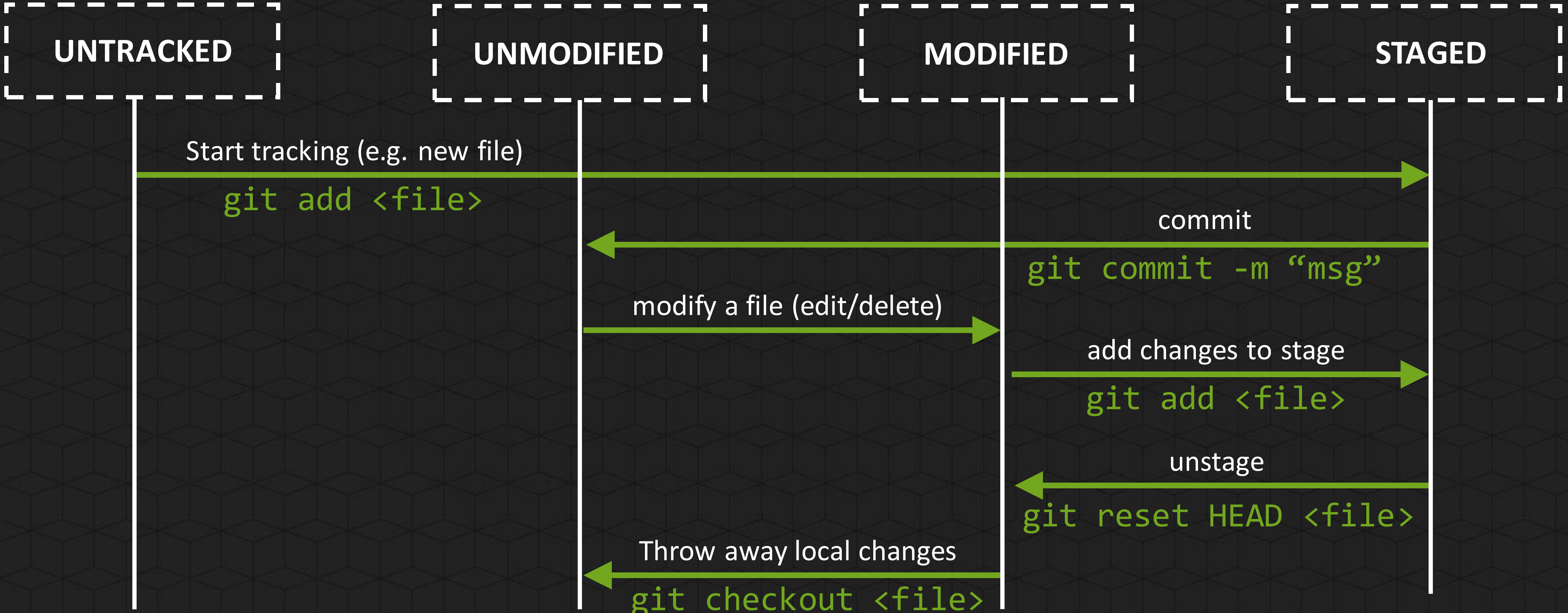
File status lifecycle

Tracked files: all files that were in your last snapshot (or in your staging area). **Simply put**: all files for which Git performs version control.

1. **Unmodified files**: all files that haven't been modified since the last commit. (A reference will be included in the snapshot).
2. **Modified files**: all files that have been modified since the last commit. (The new version of the file will be included in the snapshot if the file is added to the staging area)
3. **Staged files**: all files that are either not present in the last commit (e.g. newly created files that we started tracking) or Modified files. However, in both cases these files have to be manually added to the staging area!

File status lifecycle

File status lifecycle: some possible flows *(from one status to another)*



UNTRACKED

UNMODIFIED

MODIFIED

STAGED

Start tracking (e.g. new file)

`git add <file>`

commit

`git commit -m “msg”`

modify a file (edit/delete)

add changes to stage

`git add <file>`

unstage

`git reset HEAD <file>`

Throw away local changes

`git checkout <file>`

Let’s **demonstrate** this *(hands-on in group)*

❑ Let’s create the following directory structure using CMD

```
|-- songs
    |-- song2.txt
```

❑ Let’s initialize Git inside our **songs** working directory

- ❑ `> git init`

❑ We can check the Git status of our project using the status command

- ❑ `> git status`

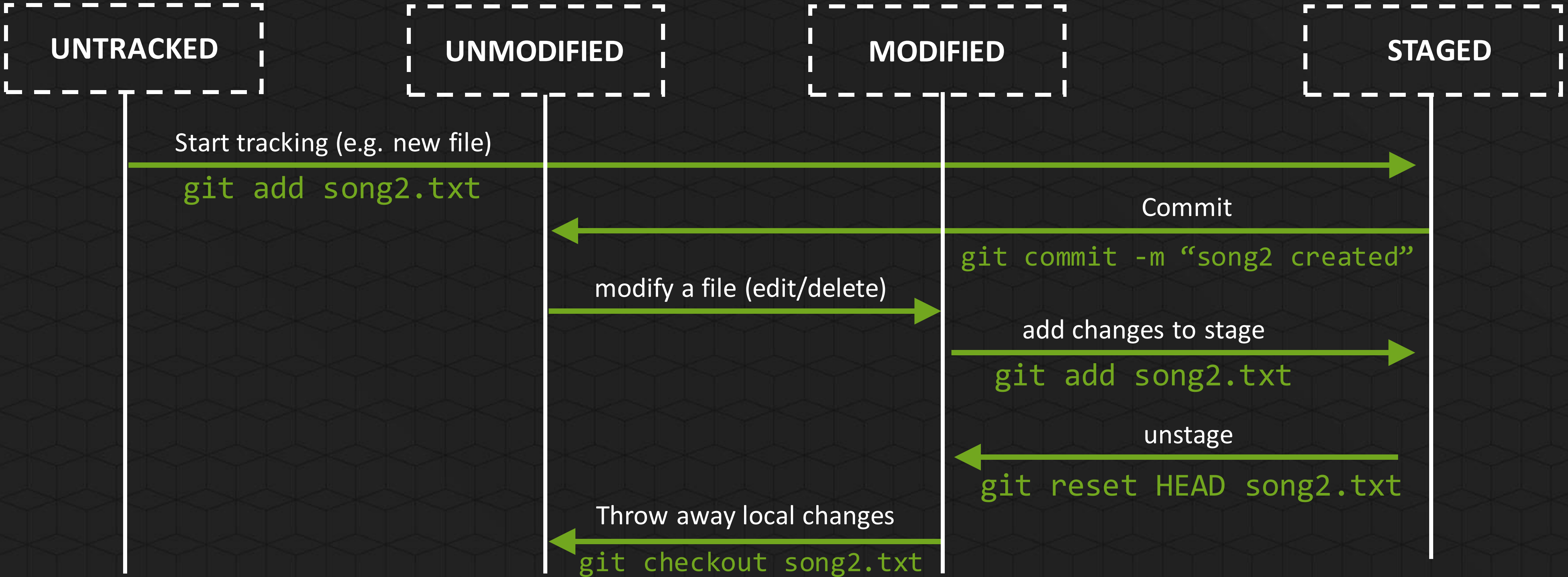
❑ From **untracked** to **staged**

- ❑ `> git add song2.txt`
- ❑ Afterwards, check the status again

Let’s **demonstrate** this *(hands-on in group)*

- □ From **staged** to **unmodified**
  - □
    ```
    > git commit -m “song2 created”
    ```
- □ From **unmodified** to **modified**
  - □
    ```
    > echo lyrics >> song2.txt
    ```
- □ From **modified** to **staged**
  - □
    ```
    > git add song2.txt
    ```
- □ From **staged** to **modified**
  - □
    ```
    > git reset HEAD song2.txt
    ```
- □ From **modifed** to **unmodified**
  - □
    ```
    > git checkout song2.txt
    ```

Let's **demonstrate** this *(hands-on in group)*



```
git add song2.txt
```

```
git commit -m "song2 created"
```

```
git add song2.txt
```

```
git reset HEAD song2.txt
```

```
git checkout song2.txt
```

Git basics

Commits

Git keeps **track of your data** by means of **snapshots**. Simply put, each time you **commit**, a **new snapshot is created**.

A 'picture'. Also called a 'commit'

By executing command
`git commit`

Git basics

Commits

A file that is modified but not added to the stage will have its old state included in the next snapshot (its most recent changes will be excluded from the snapshot)

Each **commit captures** the most **recent state** of **every individual file** in your **entire project**.

Git basics

Commits

As otherwise, the new snapshot would be identical to the previous snapshot.

Once we have **at least 1 file** in the **staging area**, we can **create a new commit (snapshot)**.

Good to know: an empty folder does not exist for Git.

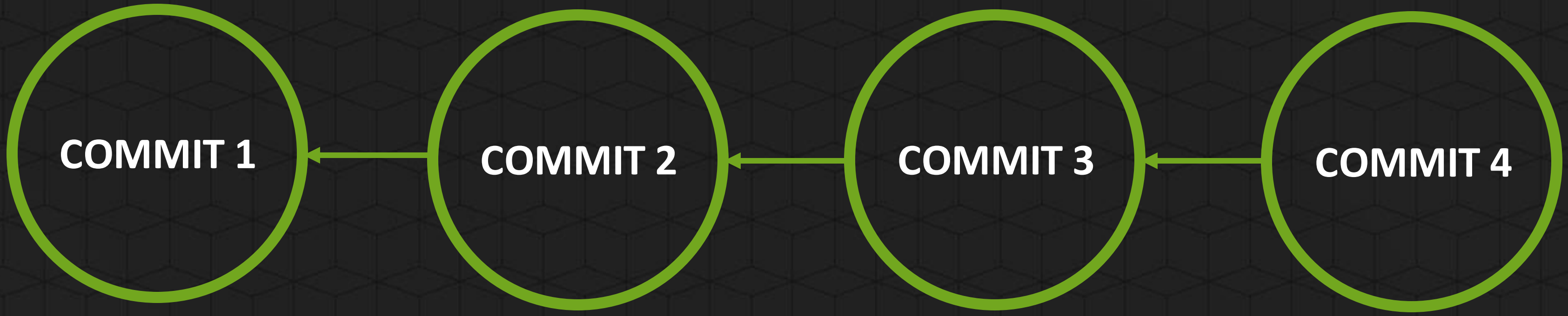
Every **commit (snapshot)** is stored with a **custom message** and receives a **unique reference id**.

A hash-based id. This id can be used to checkout, restore, reset, cherry-pick,… commits

Git basics

Commits

Furthermore, **every commit** knows about its **direct ancestor (parent commit)** (by means of storing its parent’s id)



This shows a stream of snapshots / commits in a more popular and simplified visual representation. It’s a representation you’ll often find in Git resources / documentation.

Git basics

Basic commands

Overview of the **basic Git commands**

`git init <dir>`

Creates an empty Local / Git repository. Without the <dir> argument, the current directory is used.

`git add [<dir> | <file>]`

Adds new (start tracking) or modified files to the staging area. They will be included in the next commit. As arguments we can select an entire directory, one or multiple files or even use wildcards (e.g.: *.txt).

`git commit –m “<message>”`

Create a commit / snapshot. Always provide a message.

Overview of the **basic Git commands**

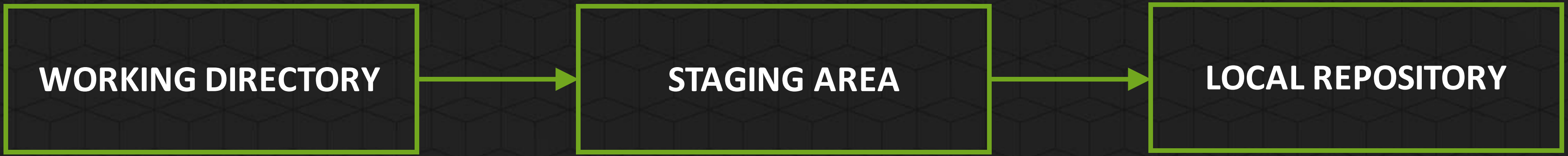
`git status`	Gives an overview of which files are untracked, unstaged or staged.
`git log`	Display the entire commit history (with option **`-n`** you can specify the amount of results (e.g. `-5`)
`git diff`	Shows the changes between the staging area and the working directory

Git basics

Basic usage

The basic **usage of Git** goes something like this:

1. You **modify** some files in your **working directory.**
2. You **check** the **Git status** of your working directory. `(git status)`
3. You **selectively stage** the **changed files**. Only the changed files you select will be added to the **staging area**. `(git add)`
4. You perform a **commit**, which results in **a snapshot** that is stored into your **local repository.** `(git commit)`



Git basics remote

Chapter 4

1. Version control
2. Git introduction
3. Git basics local

4. Git basics remote

✓Introduction

✓Push

✓Pull

✓Remotes

✓Cloning

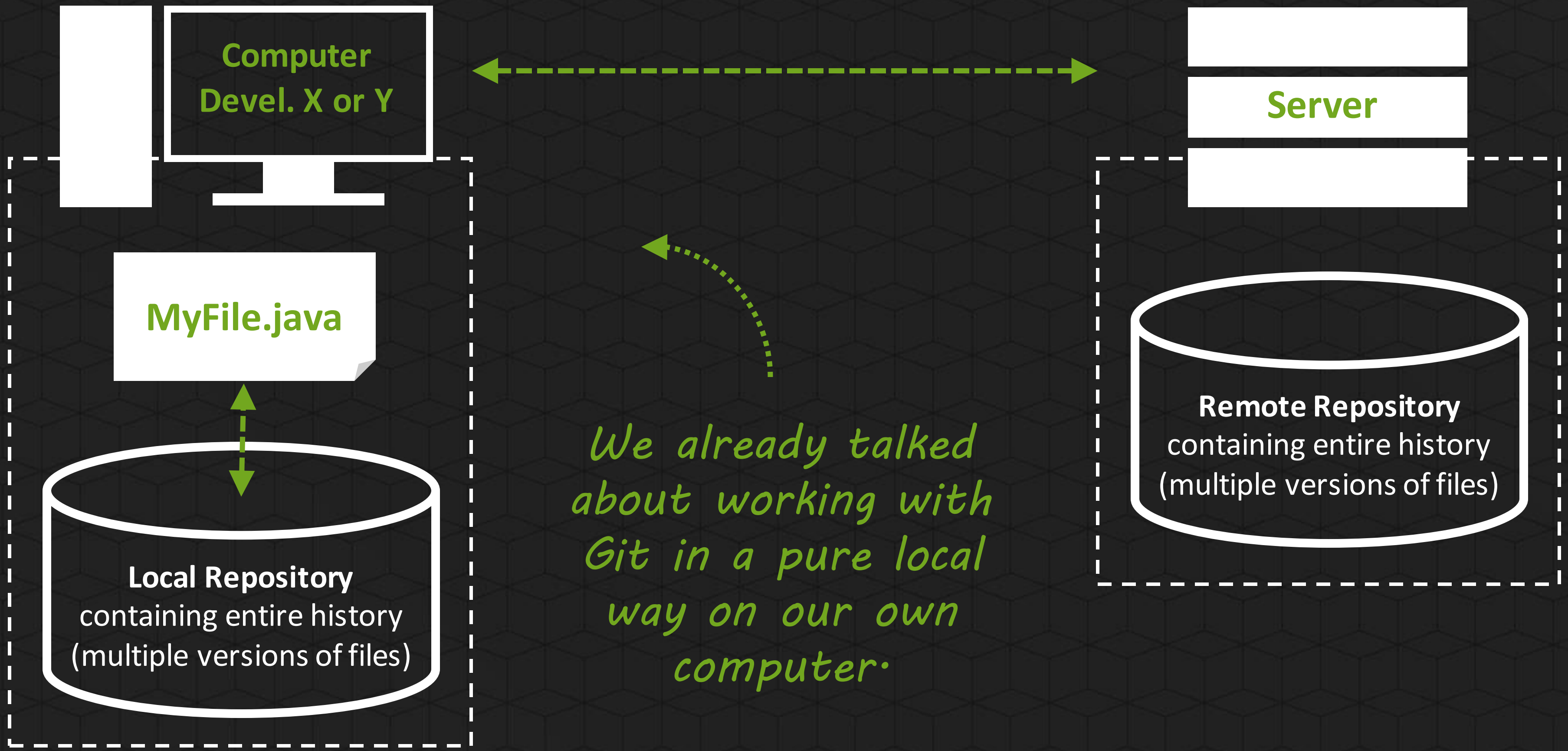
Overview

Git basics remote

Introduction

Up until now, we’ve only talked about the **local** part of **Git**. If our computer would **crash**, all of our files would be **lost**.

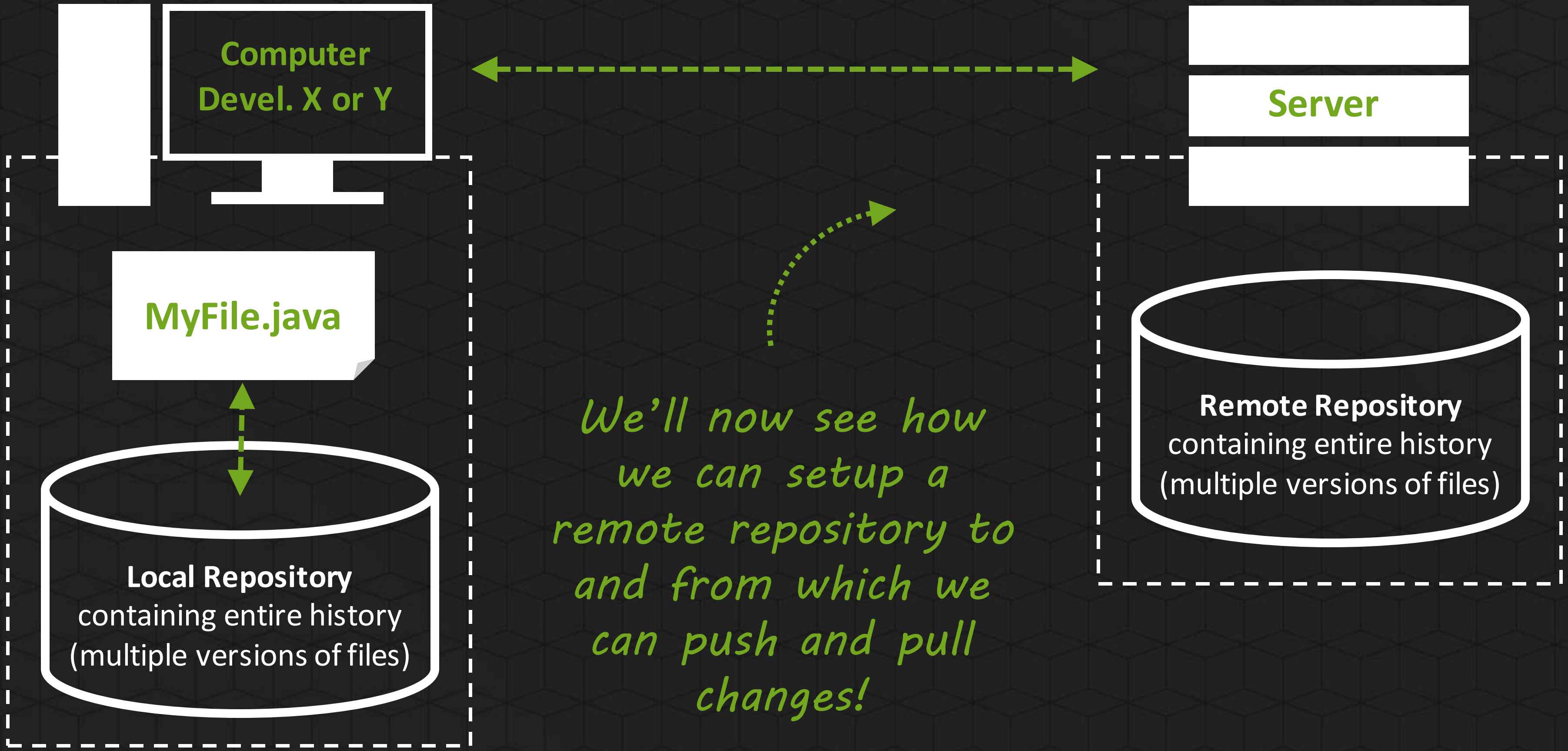
The typical high-level **infrastructure** of **DVCSs**, such as Git



Computer
Devel. X or Y
MyFile.java
Local Repository
containing entire history
(multiple versions of files)
Server
Remote Repository
containing entire history
(multiple versions of files)

We already talked about working with Git in a pure local way on our own computer.

The typical high-level **infrastructure** of **DVCSs**, such as Git

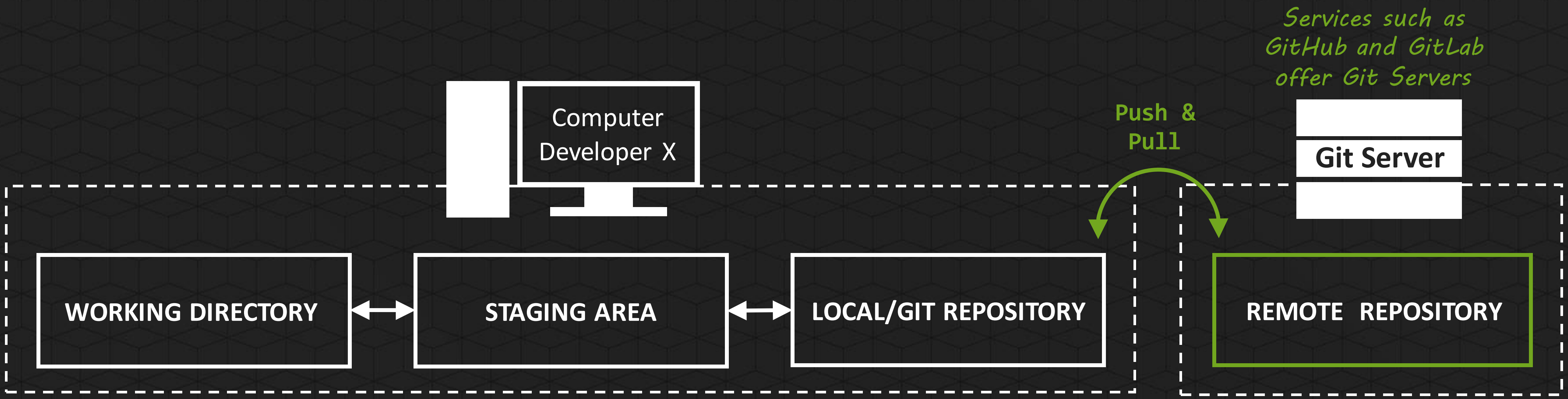


Local Repository
containing entire history
(multiple versions of files)

Remote Repository
containing entire history
(multiple versions of files)

We'll now see how we can setup a remote repository to and from which we can push and pull changes!

A **local git repository** can be set up to communicate with a **remote repository**. This remote repository holds a **centralized version of the entire repository**.



Git basics remote

Push

All commits (the snapshots/commits and all the objects (blob, tree,...) it references) that are not yet present on the remote repository will be sent.

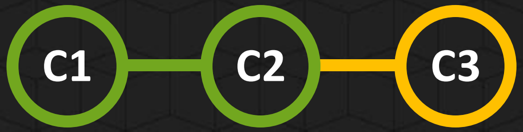
By **pushing**, we can **share / send** all of the **data inside of our local git repository** to a connected **remote repository**.

So, uncommitted changes will not be sent to the remote repository when we issue a push command!

Git basics remote

Push

Local repository

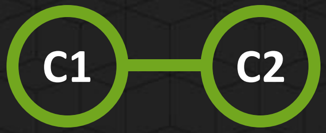


```
echo extra >> existingfile.txt
```

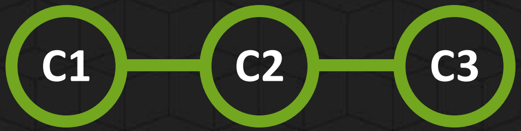
```
git add existingfile.txt
```

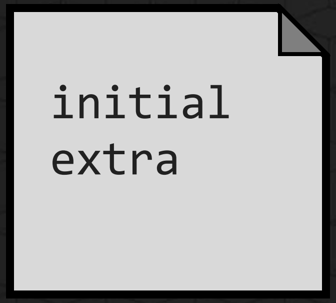
```
git commit –m “third commit”
```

Remote repository



Local repository

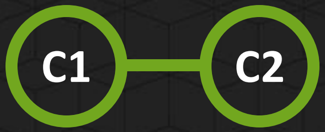


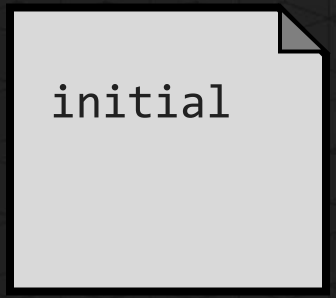


```
initial
extra
```

existingfile.txt

Remote repository



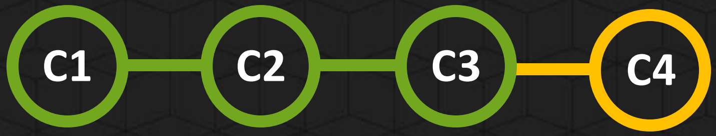


```
initial
```

existingfile.txt

Push

Local repository

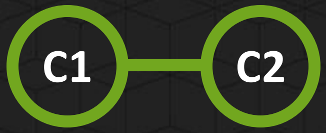


```
echo extraline >> existingfile.txt
```

```
git add existingfile.txt
```

```
git commit -m “fourth commit”
```

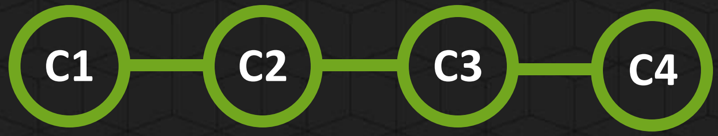
Remote repository

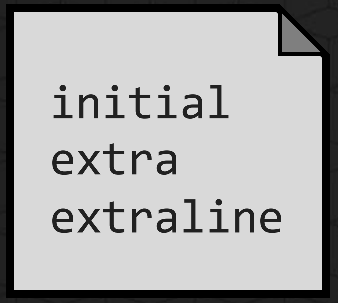


Git basics remote

Push

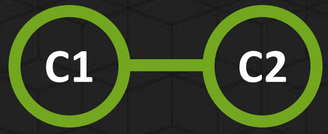
Local repository

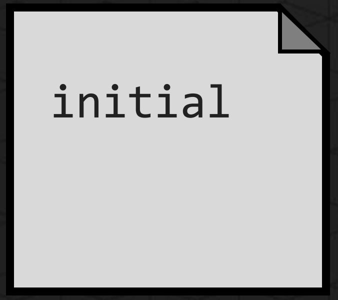



initial
extra
extraline

existingfile.txt

Remote repository




initial

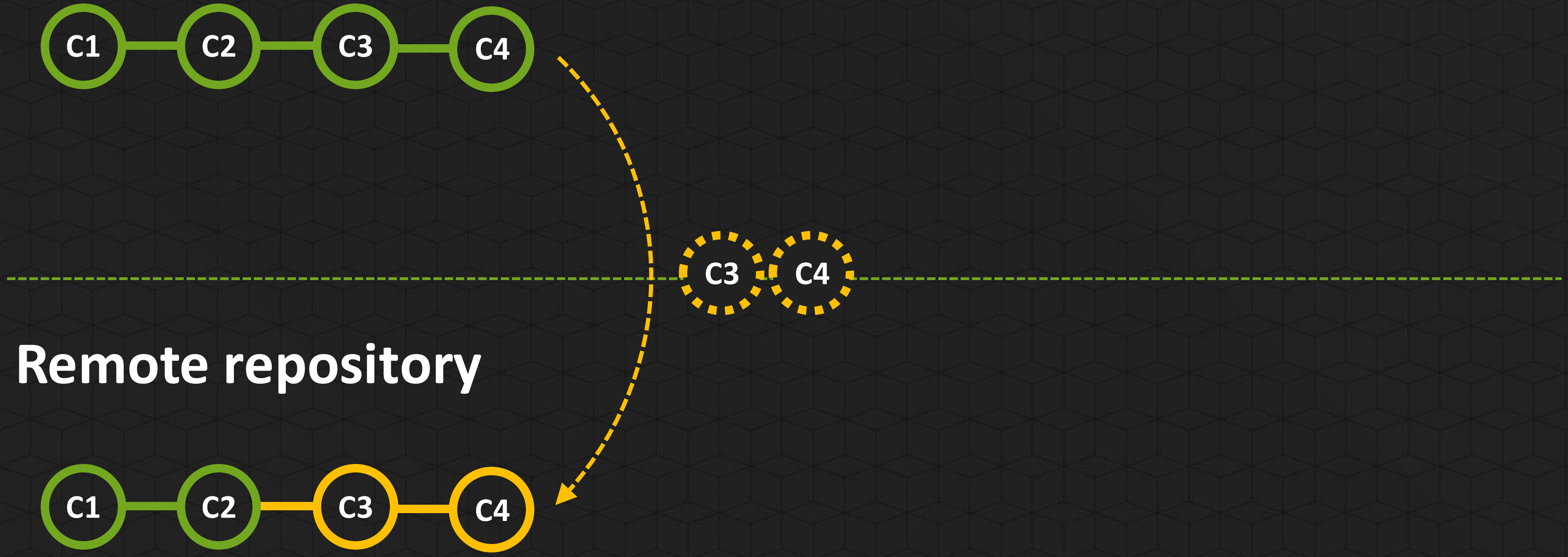
existingfile.txt

Git basics remote

Push

Local repository

```
git push origin master
```

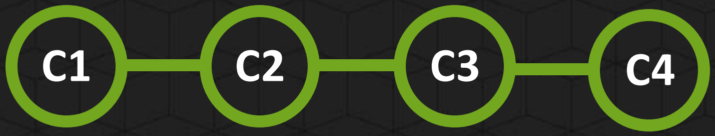


Remote repository

Git basics remote

Push

Local repository

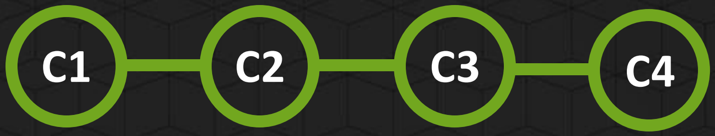


C1 C2 C3 C4

```
initial
extra
extraline
```

existingfile.txt

Remote repository



C1 C2 C3 C4

```
initial
extra
extraline
```

existingfile.txt

Push

The **Push command**

```
git push [<remote> [<src>][:<dst>]]
```

Pushes the specified branch along with the commits (+ the newly created blob-objects) to the specified remote repository.

- Argument `<remote>` is the name of the remote repository we want to push to.
- Argument `<src>` specifies from which local **branch** we want to push
- Argument `:<dst>` specifies to which remote **branch** we want to push
  - Both are part of the **`<refspec>`** option. Leaving out `:<dst>` will push from the local branch to the remote branch with the same name.

Git basics remote

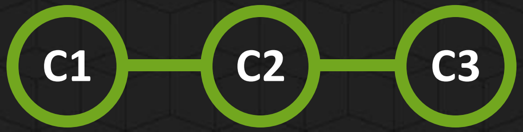
Pull

Commits / Snapshots pushed by other developers working on the same repository

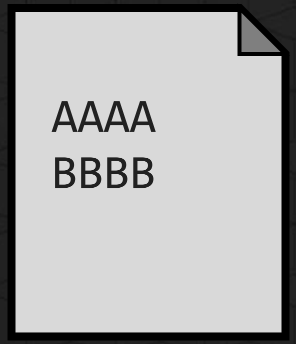
By **pulling**, we can **fetch updates** from the **remote repository** and **incorporate them into our own** local repository.

Pull

Local repository (of developer X)



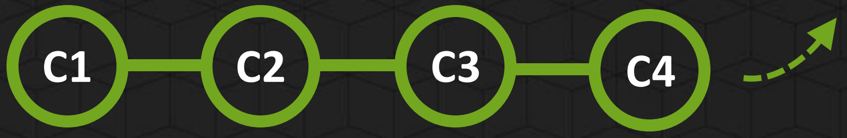
There exists a file called **`fileA.txt`**



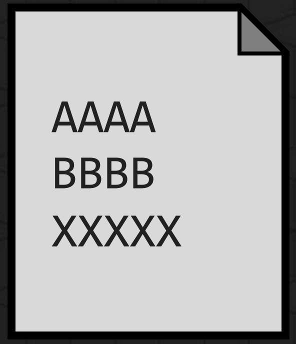
AAAA
BBBB

fileA.txt

Remote repository



***Commit C4** was **pushed** by **developer Y**, it contains (among other changes) a change in the existing file* **`fileA.txt`**

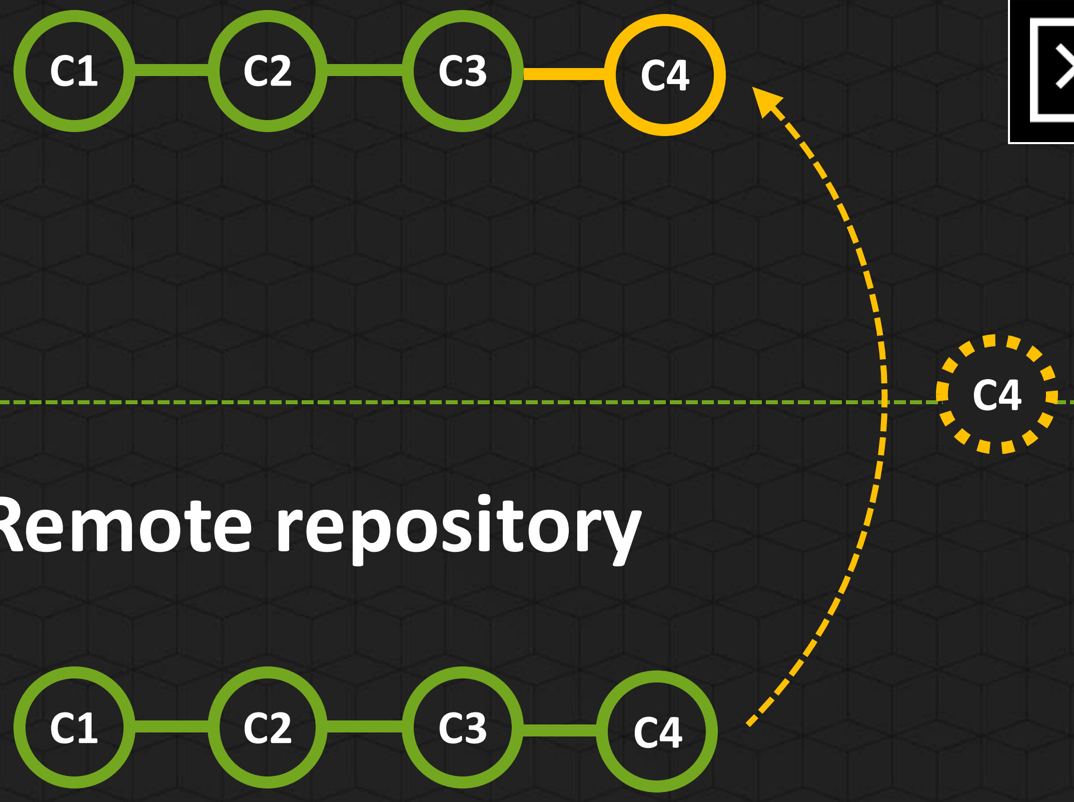


AAAA
BBBB
XXXXX

fileA.txt

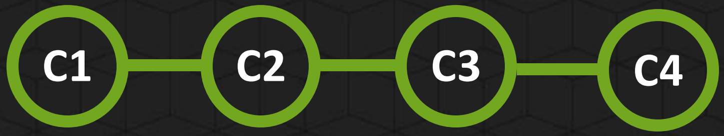
Local repository (of developer X)

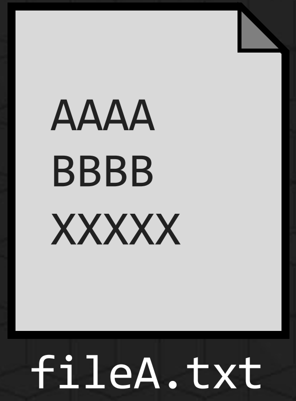
```
git pull origin master
```



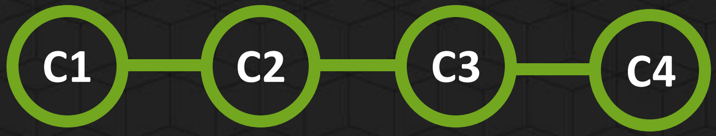
Remote repository

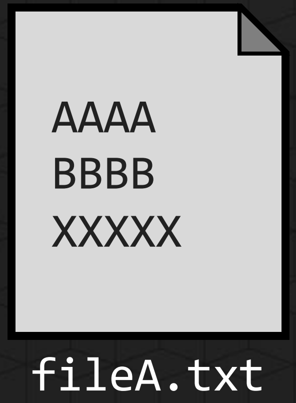
Local repository (of developer X)





Remote repository





The **Pull command**

```
git pull [<remote> [<refspec>]]
```

Fetches, from the specified remote branch on the specified remote repository, the commits (+ the other newly created objects) **and** automatically **merges** the changes into the local branch.

- Argument `<remote>` is the name (or URL) of the remote repository (to fetch from)
- Argument `<refspec>` is normally used to specify the remote branch

In its default mode, **`git pull`**, is the combination of **`git fetch`** and **`git merge FETCH_HEAD`**

Git basics remote

Remotes

Remotes are a set of **tracked repositories**.

Most often, they are repositories on a (remote) Git server.

How to configure a **remote?**

```
git remote add <remote_name> <remote_url>
```

With this command, we add a new remote repository. We can choose the name (which we’ll have to use as a reference when pushing) and the url (of the remote repository). The name “origin” is often used as the default name of your main remote repository.

```
git remote -v
```

Shows the configured / added remote repositories.

A **practical scenario**

Assumption: We already initialized a repository and created some files which we added to the stage and committed.

Goal: We want to push our local repository to a remote one.

1. Firstly, we create a new repository on GitHub (or create or own Git server).
   - E.g.: `https://github.com/user/repo-name.git`
2. Secondly, we add the repository as a new remote which we'll name "origin".

   -
     ```
     git remote add origin https://github.com/user/repo-name.git
     ```
3. Thirdly, we push our live/main (master) branch to the remote called "origin"

   -
     ```
     git push -u origin master
     ```
   - When it's the first time we push a branch, we add the `-u` option. It will set the association between the local branch and the branch at the remote (origin). The next time we push the master branch to that remote (origin), we can leave out the `-u` option.

only the project maintainer or contributors have the rights to push!

4. A complete copy of our repository is now available on GitHub

Git basics remote

Cloning

We already covered how to **initialize our own local git repository** and how to **connect it** with an **empty remote repository** (by firstly **adding a remote**). We'll now talk about **cloning**.

Git basics remote

Cloning

When we **clone**, we retrieve a **full copy of an existing (remote) Git repository** and place it into a **directory** on our local computer **that is not under version control**.

Git basics remote

Cloning

Git **clone command**

```
git clone <repo_url>
```

Clones a repository located at **`<repo_url>`** onto your local machine, in the directory in which the command was executed.

A **practical scenario**

Assumption: On GitHub we have an existing repository containing a bunch of code.

Goal: We want to be able to access, change, commit and push changes to this code

1. Firstly, we create a working directory on our computer
   - E.g.: `c:\switchfully\work\newproject`
2. Secondly, we clone the existing repository into this `newproject` directory
   - `git clone https://github.com/user/the-existing-repo.git`
3. We now have a full copy of the remote repository, containing all the files from live/main branch.
   - By cloning, git has set the "origin" remote.
   - We can now make changes, add them to the stage, commit and push using `git push origin master`

So, we now have **two ways to obtain a Git repository**:

1. **By creating our own**: You **turn a directory** that is not under version control **into a Git repository** using **`git init`** (which you then connect with a remote repository if you want to be able to push).

2. **By cloning** an **already existing repository** (containing source code) **into a** non-version-controlled **directory**.

So, we now have **two ways to obtain a Git repository**:

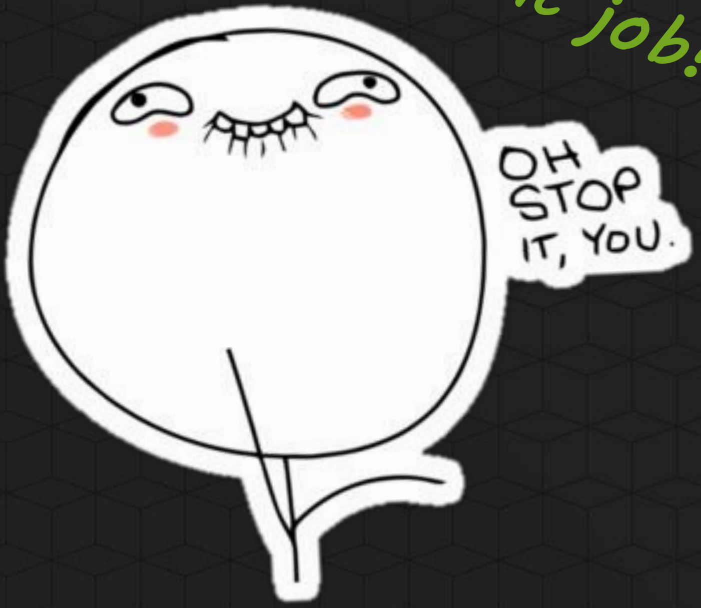
1. **By creating our own**: You **turn a directory** th ler version control **into a Git reposito** it (which you then connect with a re), if you want to be able to push).

It's one or the other, not both!

2. **By** ing an **already existing repository** (containing source code) **into a** non-version-controlled **directory**.

1. √ Version control
2. √ Git introduction
3. √ Git basics local
4. √ Git basics remote

You did a fantastic job!



Overview