What is GIT?

Intro

Hey guys Lukas here and welcome back to Koding with K! Have you heard of Github? You probably have! But do you know what a Git actually is and how to use it? If not, I got you covered! If you do maybe you can just judge my code or discuss which platform you like best!

So most people know Git-Hub which is an online platform to share your Gits. It was bought by Microsoft some time ago and shortly after became a data kraken for AI. Like most of the internet. There are alternatives to it like Gitlab or the non-profit Codeberg. However, Git can exist completely independently, offline, on your PC!

But what is it??? What is Git?

Well, first of all git is a command you type into the console window, duh!

> git

Jokes aside to understand Git you better first know the problem it solves. So let's code a little piece of software to showcase it. Yes, Git is for code. Mostly. In principle any type of text really. You can even use it writing a novel or a video script! How about a Journal? Anyone can use it!

Let's create a new file with your favorite text editor! Neovim! That's what real hackers use.

> nvim app.rs

For Loop Implementation

For this example I want to write a function in the programming language Rust. This function is meant to calculate the factorial of a number. Very excisting indeed!

The symbol for the factorial in math is the exclamation mark!

```
/*
Factorial:    5! = 1*2*3*4*5 = 120
Note:         0! = 1
Not really defined for negative numbers!
*/
```

Okay.. so that's an actual mathematical expression in case you didn't know it. However, the programming languages I know don't have an operator for that like multiplication. It's usually some function. So let's code one ourselves!

```
fn facto(x: i64) -> i64 {
    let solution = 1;
    // Find solution pls (no AI)
    solution
}
```

Function facto which takes an input value of x as 64 bit integer, and it returns some solution which is of the same type. Let's just keep this simple as a place holder for now. As we've learned the factorial can't be smaller than 1 so that's where I start. Rust like most languages also requires a main function for a program to run. So let's add that too. No inputs, no outputs. Straight and simple.

```
fn main() {
    println!("{}", {facto(5)});
}
```

This prints the solution for 5! out on the screen. In this case the console window. Let's just give it a try. Now in case you are confused by the console. This is just a text version of your regular file explorer. As you can see there is one file in it, our app.rs. In order to list the files in the console I simply use the command "ls". List. I can also use my console file manager ranger. On Windows I recommend yazi.

> ls

Next up let's verify whether the Rust Compiler is installed correctly.

> rustc --version

Alright. Now..

> rustc app.rs

I hand my code file over to the compiler and it turns it into a binary file. It complains a bit because I don't use my input variable x just yet, but that's okay. Let it complain. I'm on the Windows subsystem for Linux btw. so it's not a .exe file. Just in case you wonder. If you do this on Windows it'll be an executable.

I can run my little program like so

> ./app

Works, but let's make it a little more readable.

```
let x = 5;
println!("{}! = {}", x, facto(x));
```

Okay, compile and run it again

> rustc app.rs

Not perfect but much better! Now this shall be the first version of the program. Before I go on I'll create a new folder v0.1 and backup my file into it. This seems efficient now that our program is small but imagine we'd be working on a game hundreds of megabytes in size. This sort of backup and version tracking consumes huge amounts of space. And it is difficult to read. You need a lot of documentation to make sense of it. Where did you change what etc.

Now of course there is a smarter way. And that smarter way is Git! What Git does is to start at an initial state and then only save the changes to it. It's like recording your brush strokes for a painting. You can go back to a previous state by reversing brush strokes at any time. An infinite and persistent Undo so to speak.

But first things first. Let's create a new Git for our project. Like with Python let's first verify Git is installed correctly.

```
git --version
git init
```

For initialization.

That has now created a hidden folder .git. Let's check it out. Okaay, it's full of stuff we don't have to worry about. We will never actually touch this folder. That's why it's hidden. However, it's good to know it's there. If you delete that folder all your tracking is gone for good.

So what now? Git Gud! There are a hand full of Git commands you have to know in order to make proper use of it. All commands can be found using...

```
git --help
git --help command
```

That opens the manual with the documentation. It's a lot but again. You only really need a hand full to get started. So let's start!

```
git status
```

That will always tell us whether there is something to do or not. In this case we have a file Git has found and it wants us to add it. So let's do that.

```
git add app.rs
```

Git will now keep track of changes in that file for us. If we want Git to ignore specific files, we can create a .gitignore file. Let's do that so that git won't complain about the binary file we create, which is not a text file and which we don't want to track.

```
git status
nvim .gitignore git status
```

Okay, now it wants us to commit to the first changes. We have added the file.

```
git commit
```

This looks a little confusing. What Git did was to launch your favorite text editor, in my case neovim, and it wants me to type a short comment for this commit. I'll just type something descriptive. That's all it is.

> Start of project, added first file

These commit comments can be as long as you want. Whatever floats your boat!

Now there is no rule how often you should commit to changes but to keep things tidy you shouldn't commit like every few minutes. Rather commit when you have developed and finished a new function for example. Or written a new scene or chapter of a book. Git just tracks your progress and is not a databse you have to update everytime you turn the PC off. That will get messy real fast. And another unwritten rule is to only commit code that works. Like don't commit stuff that has errors in it. There is no point. Unless you work with somebody else and want them to fix it of course.

Anyways, now that we have our first commit we can check the log.

> git log

There you see our comment and this one here is the commit hash. It's a unique ID for this particular commit. And you see my name and... wait a second, I forgot.

If you're using Git for the first time you will run into an issue because you have no name yet. So in order to change that we can define a name for this project, or like I do globally using the --global option.

> git config --global user.name "Lukas"
> git config --global user.email "@KsNewSpace"

Now the commit and git log should work. Sorry about that!

> git log

I don't know about you but I want to finally implement the function and not just mess with git. Let's do this now. All I do is to multiply al values from 2 to x to by solution using a simple for loop. That's it.

```
fn facto(x: i64) -> i64 {
    let mut solution = 1;
    if x > 1 {
        for i in 2..x {
            solution = solution * i;
```

```
        }
    }
    solution
}

fn main() {
    let x = 5;
    println!("{}! = {}", x, facto(x));
}
```

One thing I forgot to mention is that in Rust the last value inside of a function is just returned by the function. So you often don't need the return command like in other languages.

Okay let's check if that runs on the first try..

> rustc app.rs
>
> ./app

Sort of. We get no error but the solution is wrong. What a surprise! The correct solution is 120. That's just a quirk of how this double dot operator works. It does not include the final value x. That has to do with arrays. An array with 10 items is indexed from 0 to 9. So you'd go from 0 to number of items which would be 10 without going over the boundary. However, Rust has us covered I simply have to add an equals sign to include the x.

```
fn facto(x: i64) -> i64 {
    let mut solution = 1;
    if x > 1 {
        for i in 2..=x {
            solution = solution * i;
        }
    }
    solution
}

fn main() {
    let x = 5;
    println!("{}! = {}", x, facto(x));
}
```

From 2 to equals x.

```
rustc app.rs
./app
```

Okay, so we have made a meaningful change to our project and it runs. Time to commit!

```
git status
```

As you can see it asks us again to add the file to the Git. Now this is because in theory you can change multiple files but only commit to changes you've done to one. So it simply asks you for which changes you want to commit. In our case we commit all so we can simply type..

```
git commit -a
```

That's it. The changes were committed. We can view the log again and have a nice overview about what've done!

```
git log
```

This is what Git is and how you use it. Now an online platform like Github is where you can share your Git with others to collaborate. Or to learn from you. But that's not necessary. Git works 100% offline too.

Bonus Content (Recursion)

You can end the video here but I wanted to show you one more trick using Git. And that trick is branching. Often times in development you will find yourself working on different features at the same time. Or different chapters of a book. Especially when collaborating with others.

```
git branch
```

...will display all the branches. We currently only have one of course. Let's change that! However, let's not branch of from where we are, but where we were. This is where our commit hash comes into play. I want to start a new branch before we implemented our function. Maybe to find a smarter solution.

```
git log
```

Let's grab the hash. Copy and paste.

> git branch smart *hashcode*
> git branch
> git switch smart
> git branch

Now we're in a new branch called smart.

> nvim app.rs

Boom and our file is back to start. Whaaaaaat. Yes, Git has full control over our files now and with great power comes great responsibility. Just so that you know, you can completely ruin your project with the wrong commands. So I would experiment with a project that's not so important first.

Let's now implement our function in a smarter way. Turns out a for loop is not the smartest.

```
fn facto(x: i64) -> i64 {
    x * facto(x-1)
}
```

We solve it recursively. What that means is we call our function inside the function itself. Now that can really hurt in the brain if you have never heard about it, but doing this creates some kind of virtual construct that would normally crash the program. It's an infinte loop into itself. You call the function, it called itself, then again and again.

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

So what we need is a break point in the function that collapses this virtual construct. This sounds way more complicated than it is. All we have to is to add a condition. Only do this when x > 1 otherwise return 1.

```
fn facto(x: i64) -> i64 {
    if x > 1 {
        x * facto(x-1);
    }
    1
}
```

If you haven't noticed yet, in Rust if you want to return a value you just write the value without semicolon. It's a simplification. However, that only works on the last line of a function. Since we want to return a value from inside this if statement we have to add the return like in most other languages.

```
fn facto(x: i64) -> i64 {
    if x > 1 {
        return x * facto(x-1);
    }
    1
}
```

That's it. As long as x is larger than 1 the function will continue calling itself with a smaller value. Once we reach 1 it'll return 1, an actual value and the recursion collapses from the back to the front. You can solve most problems recursively and while it is extremely satiyfying, it's not always the fastest way in terms of computing efficiency. But it's a fun challenge!

Let's try it out!

> rustc app.rs
>
> ./app

Looks exactly the same from the outside but better from the inside. Our program gained character. Anyways, we've yet again done a meaningful change so time for another commit!

> git commit -a

And now that we have multiple branches we can hop between them and observe the code changing. I still remember the first time I learned about it. It was epic!

> git switch master
> git switch smart
> git switch master

Now we can do something you normally should avoid. We can merge smart into master because the master solution is not so great. Normally you'd only merge branches into master that change something about it that

wasn't changed in master afterwards. Because now as you will see there is a conflict. The same file was changed after our branch point. So Git is not sure which changes should be kept.

> git branch
>
> git merge smart

If we now check our file Git added the new code but left the old one. And it added these arrows which make sure it doesn't work anymore. So you can#t leave this untouched by accident. This can be messy but you just have to remove the parts you don't need. But again, this wouldn't have happened had merged new files into master or if we didn't change the files we marge in master. Anyways, we are now left with just the code of the smart branch.

> rustc app.rs

Making sure it compiles without error.

> git status
> git branch

Now that we've done the changes we can commit to them!

> git commit -a

It already filled the merge comment for us so all we got to do is save it.

And we merged! They're both the same now and we can delete the smart branch.

> git branch -d smart

As mentioned using branches is more sensical when you want to develop new features you then merge into master once they're done. Or just backups of your project. And when you have a release version. You create a branch for that.

And now as a short summary of all the Git commands we used. Just by the way, in the info box you can find this video as a pdf document. Feel free to check it out.

> git --help
> git init
> git add filename / foldername
> git log
> git commit (-a)
> git branch git branch branchname
> git switch branchname
> git merge branchname
> git branch -d branchname git config (many options) [git revert commithash] <- dangerous

I put a new one in brackets. It is fairly common to revert changes done by a commit but it is extremely dangerous. This can trash your entire Git. I'm talking from experience. That's why I avoid it. I instead create a new branch from an earlier commit and then merge. That is non destructive. I still keep the old branch. I can delete it or keep it.

Now all these commands have their own options. They have more depth. You can read up on this using...

> git --help commandname
> git --help merge

It opens a browser window with the documentation stored locally. As you can see there is a lot you can read up on if you want. But you don't have to. I didn't. Maybe I should though. You do it, and then tell me if I should :)

Okay, that's it for this video! Machet Jut! Auf Wiedersehen! Au Revoir! Adios! Sayonara! Do Widzenia! See ya! And Sank you for watching.