Solidity is an imperative language, and as such it provides the usual Flow control mechanisms like **ifs**, **fors**, **whiles**, and **try/catch**.

If you are anxious, you can have a look at other examples in here https://solidity-by-example.org/

Examples beyond "Library", will be explored in later parts of this course, because, for the moment being, we will go straight to our first project in a real setup! So, say goodbye to Remix.

To complete the following tasks, you will have to clone this **repository**.

In the template repository you will have a setup very similar to what we use in productive projects (we just removed some options and libraries so that this is smoother for you, but do not worry, we will add them later). In it you will be working mainly on three different directories, named "contracts", "test", and "deploy".

As said, we will use **Hardhat** which is a great, open-source framework to develop Smart Contracts.

Its greatness comes from its flexibility, and that flexibility comes from the fact that you are able to add scripts, tasks and plugins made by the community.

You are encouraged to have a look at **its doc** after this challenge, but for the moment being you will need to know only about what the template offers you.

Regarding the installation of the template, please read the **README.md** file in it.

The template comes with a series of useful scripts:

- **deploy-local:** Deploys the contract in a local network (we will explain how networks are defined in the following slide)
- **test:** Runs automated tests that you will create
- **lint:** Executes a linter over your files
- **format:** Executes a prettier over your files
- **console-local:** Runs an interactive console pointing to the local network
- **node:** Runs a local mockchain (explained later)

All of these can be run using **npm run <script-name>.**

Hardhat executes and reads a file named **hardhat.config.ts** or **hardhat.config.js** when you run a task. In it you will find, an object exported, named "config" which defines the networks you have available. In it, you will find "hardhat", "localhost", and "kovan".

Hardhat is a local network that is run specifically when you are using a hardhat script/task. It is a mockchain, so it has some special features like being faster to process transactions or the ability to manipulate the time in the blockchain

Localhost is the same service but it has to be run manually and as a standalone process.

Finally, Kovan is an ethereum testnet network, and we won't be using it at the moment.

Each of this networks is defined by an object, which have three different properties:

- **"accounts":** Data needed to have the private keys that are going to be used in that network.
- **"chainId":** An identifier of the chain, you will understand why this is needed in a later part of this course
- **"url":** This is the url of a node of that particular blockchain. As we will need a node to interact (particularly, to send transactions and to read info) with the blockchain.

In the template, you won't need to modify any of these configurations.

This exercise will be about extending the **WETH** contract you already coded.

To start with the exercise, please create a new file in the contracts directory called "**WETH.sol**" with the contract you have created in the previous exercise (with the pragma statement included).

Please have a look at the file inside the "deploy" directory. This file will already be complete, but it will let you deploy your contract (i.e. it will create an instance of that "class"). This file will be called at least once when you run your tests.

Now you will create automated tests, to check that your contract works as expected. Note that, when dealing with Smart Contracts, it is extremely cheap to test (because it almost doesn't have external dependencies) and that a bug that reaches production is **EXTREMELY** expensive, so you should change your mindset to one that prioritizes tests over code (TDD is recommended for those willing to try).

For this particular exercise, please test (at least) that your contract returns the right value with balanceOf once you have deposited and that you cannot withdraw more than you have deposited.

Please look into the next slide to have some guidance.

To understand how to test, we have provided you two test samples. Both of them are in the **test-samples.test.ts.**
The first one deploys the contract and checks that the contract has been deployed. To do this we used a special function that is defined in the **common-fixtures.test.ts.** We will explain what a fixture is in later parts of this course, for the moment being keep in mind that this executes the code in the file located in the **deploy** directory.

The second one checks that deposit is able to receive ethers. This one also uses the previously mentioned function, but in top of that it interacts with the contract through sending a transaction! Keep in mind that any interaction with a Smart Contract is asynchronous so to get the result of it you should **await** it.

To run your tests, you can just execute **npm run test** in the command line, being in the root directory of your project.

To help you on this task, you may want to have a look at all the possible assertions that waffle provides for you at its official docs.

Once you have tested what you already coded, you will make your contract an ERC20. But first, lets understand what an ERC is.

ERC stands for Ethereum Request for Comments and are standards/proposed standards to be used in the Ethereum Network. They usually emerge after being discussed on an EIP(Ethereum Improvement Proposal). Some common examples (which will be treated in later parts of this course) are ERC20, ERC721, and ERC712.

An ERC20 is a contract that implements the interface defined in the **EIP20**.
An ERC20 is a token that is **fungible**, just like the Ethers are.
It basically has a way to transfer tokens from one address to another, a function to query the balance of an address, a set of functions to allow transfers in the name of another address (approve, transferFrom, and allowance) and a Transfer event to signal transfers.

The word token can be used interchangeably for the contract that manages the balances or the balance itself.

For example, you could say that the DAI token has the address **0x6b175474e89094c44da98b954eedeac495271d0f** and that you have 15 DAI tokens.

The first time you would be talking about the contract and the second one you would be talking about your balance.

To help us with this task, we will be using the OpenZeppelin's contract library. This library defines a lot of base contracts that will help you (just as much as they help us in our daily work) to extend your Smart Contract.

To use it in your code (you already have it installed thanks to the template) you just have to import it pasting the following line after your pragma statement.

Note that we are importing the base implementation of an ERC20 (OZ library also has a base implementation of ERC721, ERC1155 and much more things, we will be reading more about them in the following parts of this course).

Now you will need to make your contract extend (through inheritance) the ERC20 of OZ. After this you will need to stop tracking the balance of the users since this will be the responsibility of the ERC20 implementation, but you will have to let the implementation know that the user deposited. To do this you will need to call the **_mint** function provided by the library in your **deposit** function. The equivalent has to be done with **_burn** and **withdraw.**

Mint and burn are the verbs usually used when you create and destroy tokens, respectively.

On top of that, you will have to set some informational values through the constructor, so you will need to create a constructor that calls the ERC20 constructor. Have a look at the ERC20 constructor at **github** and how to call the constructor of the parent class at **the official docs**.

Now you should test that your integration works correctly!

The previous tests should keep working, but you can also test that **deposit** emits a Transfer event from the zero address (pro tip, there is a constant in ethersjs that defines this address) and that **withdraw** emits a Transfer event to the zero address, and many other things. Note that it is not necessary to test that the library works correctly but feel free to do so if you feel that gives you insight.

Finally, we can format the contract by running

npm run format

and check that the linter does not detect any issue with your code through

npm run lint

To deploy this contract locally, you will have to have a mockchain node setted up which can be done executing

npm run node

After that, in another terminal run

npm run deploy-local

Once you have done that, you can interact it manually in a REPL environment. To do this execute npm run console-local. Once inside the console, you can get an object to interact with your contract by executing

```
const weth = await
hre.deployments.getOrNull('WETH')

const Weth = await
ethers.getContractFactory('WETH');

const WETH = Weth.attach(weth.address);
```

After this you can interact with it by calling its functions.
For example:

```
const [myAddress] = await hre.getUnnamedAccounts()

console.log((await
WETH.balanceOf(myAddress)).toString());
```

Will print your balance.

Thank you ;)

Globant University

ATIX
A GLOBANT DIVISION