In solidity, you can have statements that check preconditions. This is usually done through the "require" clause, which can be used like so:

```
require(variable == expectedValue, "error reason");
```

If the first parameter is not true, the transaction will be reverted, meaning that all of the changes made in that transaction will be rollbacked automatically and that the unconsumed gas is returned.

Now that you have the needed knowledge, in Remix, create a contract that has three functions:

- A "receive" function
- A custom payable function (named deposit)
- A withdraw function that receives an amount, and transfers that amount to the caller(named withdraw)
- A function to check how much an address still has available(the address is received as a parameter)(named balanceOf)

The contract should track how much each address deposited and allow said address to extract just as much as the address deposited(not more than that). The last function should always return the amount deposited by the address minus the amount that was withdrawn.

The contract should be named "**WETH**"

This is a protoversion of a very common Smart Contract usually called WETH, which will Wraps Ethers into another interface (called ERC20, we will learn about it later).

You can see our model implementation of it in **the contracts folder inside** this repo but we strongly recommend that you try to do it by yourself first. Also, keep your contract saved after you finish your task, you will need it later (this is useful tip for any of the code, artifacts that you produce during the course).

Events are a mechanism to communicate what is happening in your Smart Contract with the outside world. Any backend or frontend can read and ask a blockchain node for the events in your contract, but you cannot read (much less act based on) other's contract events.

An example can be seen at:

https://solidity-by-example.org/events/

Modifiers are custom functions that can be used to modify(add behaviour) to functions. Modifiers are great to implement access control(for example, you may use it to protect every function that can only be accessed by an admin of the contract). The modifiers have a special syntax where the modified function has to add the modifier name in the signature.

An example can be seen at:

https://solidity-by-example.org/function-modifier/

There are several mechanisms to design your solution with Smart Contracts. We will review them in the following slides.

Solidity contracts, just like Java classes, can also be modelled using inheritance.

There are some special keywords for it, please look at the examples in **these examples**.

Just as in Java, you can call your parent class like **so**.

Solidity also supports interfaces. Have a look at **this example**.

To have an abstract contract you will need to use the keyword "abstract" before the "contract" keyword. Contracts that do not implement any function that is defined, have to be abstract.

This types of contract, as well as interfaces, can not be deployed.

You can also use libraries in solidity **like so.**

Libraries let you extend your contract easily by letting you use your state with their functions. In this way, contracts that have the same functions can deploy a single library to reuse the functions without sharing state.

It is also a nice way to extend your contract with functions related to a specific data type.

Composing contracts is our recommended way to model your solution. If the Smart Contract is smaller, it is easier to maintain. This, contrary to the sending ether's rule, this is a preference. Sometimes it is better to use inheritance or a library, because it is a cheaper way to do it, but there are other problems with inheritance which will become more clear when learn about contract sizes.

Super important!
Keep in mind that, if a contract allows it, it can be called from any address (contract or EOA) BUT a transaction **ALWAYS** starts from an EOA (i.e. a contract cannot spontaneously execute code). In other terms, every time code is executed it is because an EOA sent a transaction. The transactions can execute a contract that calls another (called an **internal transaction**), and that's what we call a composition of contracts. Please, look and play around with this **example**.