DEVELOPING DAPPS IN ETHEREUM: THEORY AND PRACTICE

4TH SCIENTIFIC SCHOOL ON BLOCKCHAIN &
DISTRIBUTED LEDGER TECHNOLOGIES

CAGLIARI 12 SEPTEMBER 2023

ROBERTO TONELLI - ANDREA PINNA

DEVELOPMENT OF SMART CONTRACTS

CONTENTS

- Metamask
- Remix
- Basics of contracts
- Address type
- Contract Calls and delegateCalls
- Structures and mapping
- Events and logs
- Require and modifiers
- Contract creation and factory pattern
- Inheritance
- Proxy pattern
- Token and NFT

METAMASK

Metamask is one of the useful tools for managing your assets (like a wallet) and creating dApps. This is a browser extension. It has more features than a wallet. It is called "Bridge" because it acts as a bridge between our web browser and the blockchain network
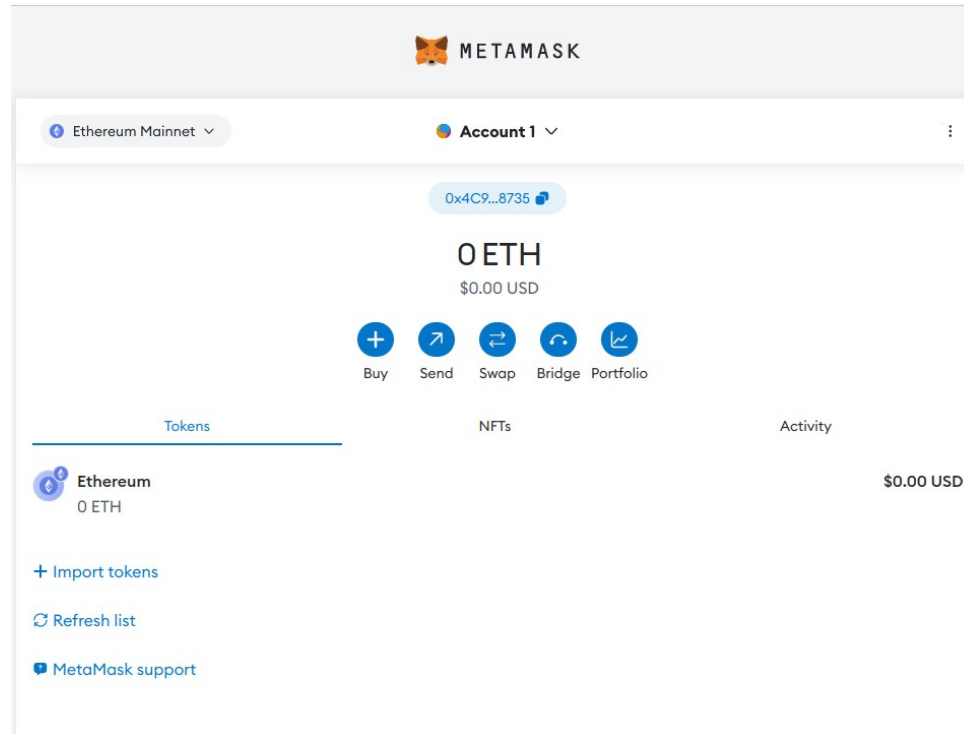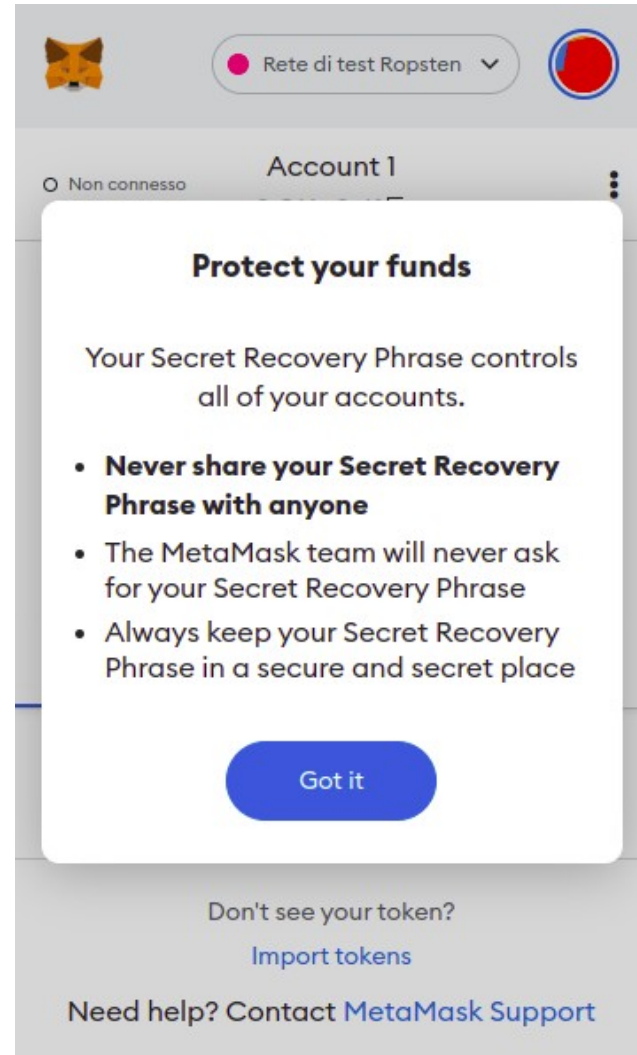
METAMASK

Attention:
The generation of the private key occurs through the generation of a Pass Phrase (to be transcribed)

Metamask saves ours **private keys**.

We are not asked for any information personal information (not even the E-mail).

METAMASK

So, this is a **Phishing** email

Oggetto: IMPORTANT: Verify Your MetaMask Wallet !
Da: MetaMask Wallet <noreply-84565126786500026@informetamaskio.com>
Data: Lun, 18 Aprile 2022 5:56 am
A: "a.pinna"
Priorità: Alta
Opzioni: Visualizza l'intestazione completa | Visualizza versione stampabile | Scarica come file | View

This is an automated email to notify you about a problem related to one or more
wallets registred to this email adress..
Due to recent changes in laws and policies in various countries, all MetaMask
Wallets ar required to be verified. our system showed that at least one of your
wallets did not finish the verification process.
Verification can be done easily on our website by using the link bellow. After
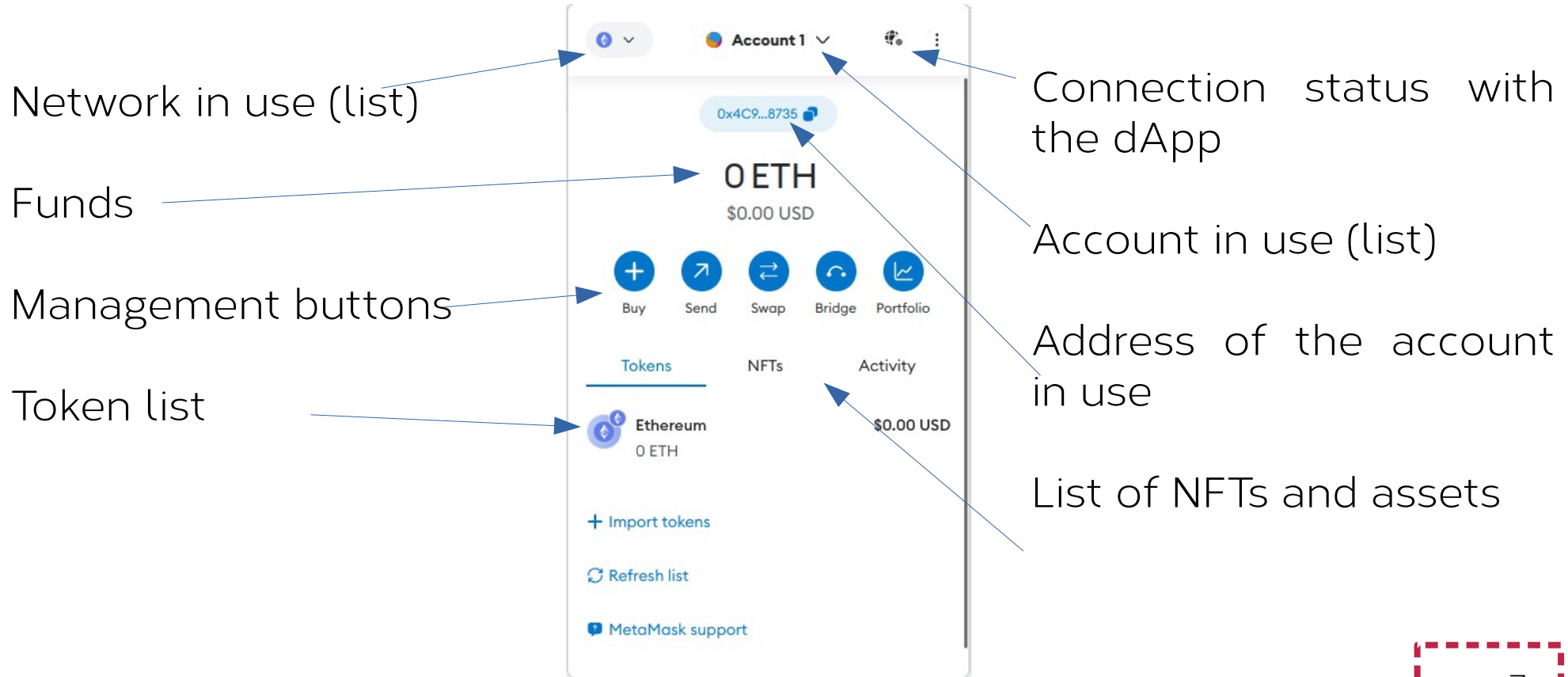connecting, your wallet will automatically be verified.
Validate Your Wallet Now
if you do not want to use our services in the future, you can ignore this email.
When no action is taken, your wallet and personal data will be deleted automatically
by the end of April 2022. We apologise for any inconvenience caused. For more
information on this announcement please visit our support centre.

METAMASK

By clicking on the fox icon I get the wallet panel

Network in use (list)

Funds

Management buttons

Token list

Connection status with the dApp

Account in use (list)

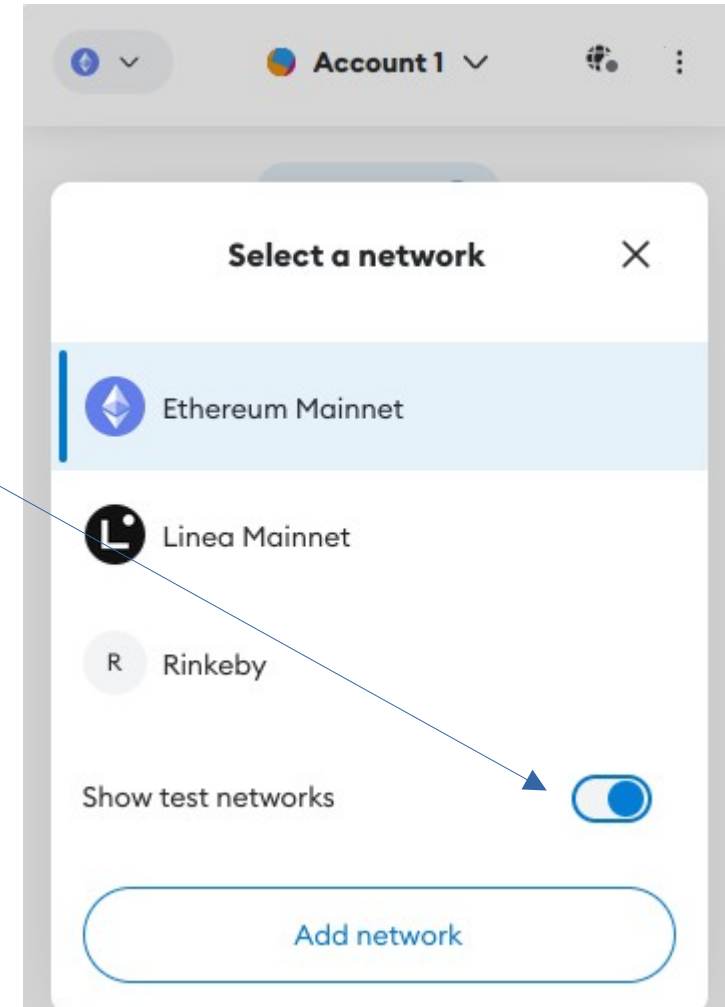Address of the account in use

List of NFTs and assets

7

METAMASK

We will use the sepolia **test network**,
To view the test networks, you need
to **enable them** from the network
list

To obtain test Ether we can use a
faucet service.
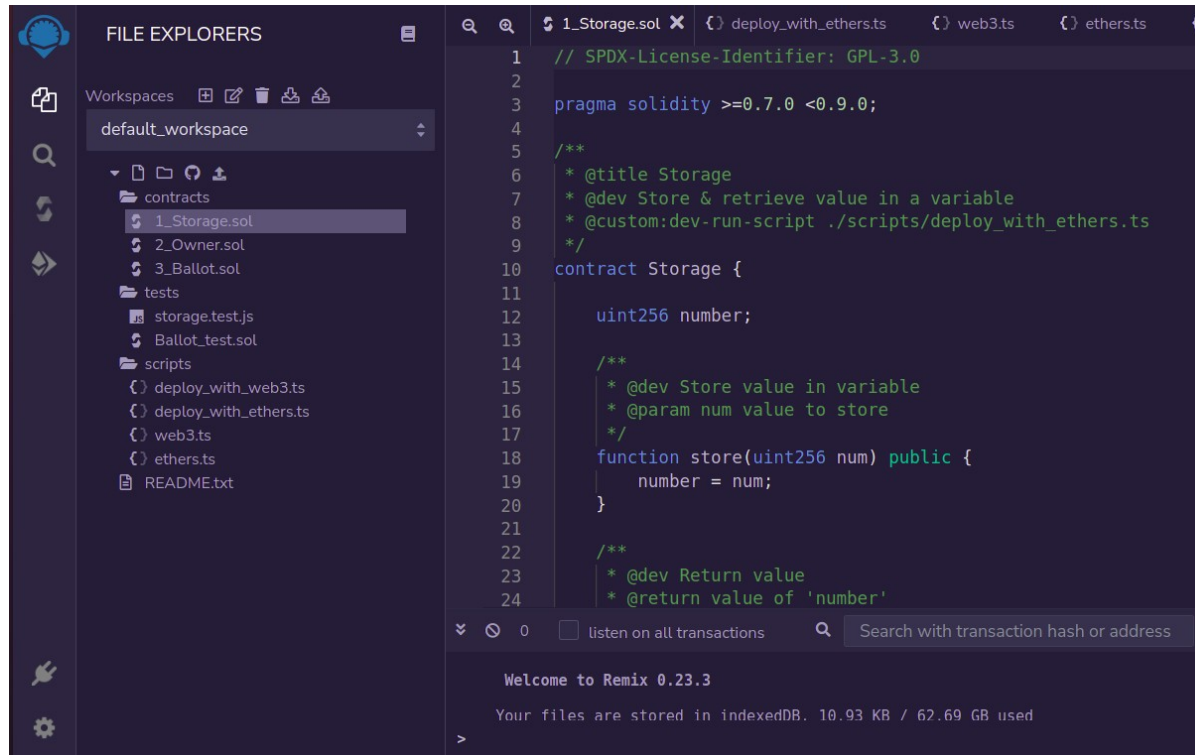
https://faucet.quicknode.com/drip

List:: https://faucetlink.to/sepolia

REMIX

The development of smart contracts can take place through IDEs that support solidity.
The "native" IDE is Remix and **is web-based**. https://remix.ethereum.org/

REMIX - NEW FILE

To create a file you need to click on the sheets icon on the left and then on the single sheet icon in the File Explorers box. Then enter the file name. Solidity sources are plain text files saved with a .sol extension
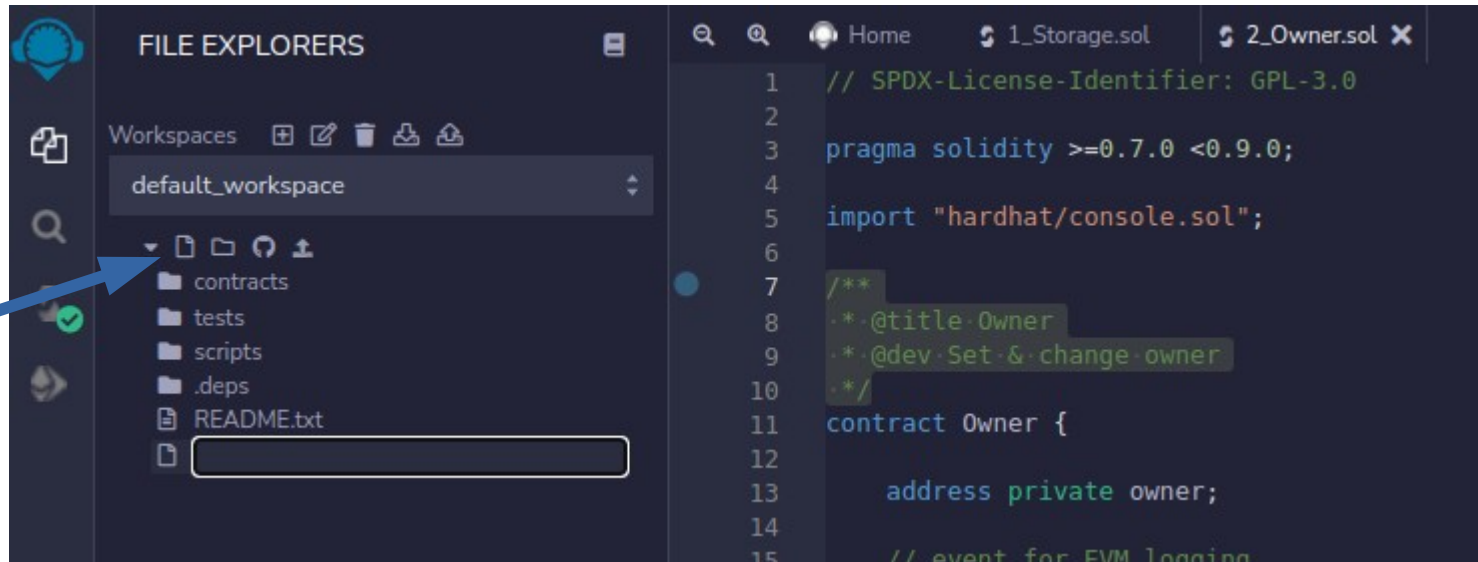
SOLC

The solidity language compiler is **solc** (SOLidity Compiler). The compiler produces machine code for the EVM called bytecode. Remix includes numerous versions of the compiler.

You can also install the compiler locally
https://docs.soliditylang.org/en/latest/installing-solidity.html

```
~$solc --version
solc, the solidity compiler commandline interface
Version: 0.8.17+commit.8df45f5f.Linux.g++
```

SOLIDITY

Solidity is the most used language for programming smart contracts. It was defined for the Ethereum environment and the EVM. Today it is used in numerous "EVM-like" blockchains.
It is a high-level abstraction language, which is inspired by languages such as Java, C++ and Python.

The language paradigm is "object-oriented" and has unique characteristics that make it suitable for the development of decentralized applications.

Docs: https://docs.soliditylang.org/
Book::
https://github.com/ethereumbook/ethereumbook/blob/develop/07smart-contracts-solidity.asciidoc

12

SOLIDITY

In Solidity we write Smart Contracts (SC), or contracts.

- A SC is similar to an OO class, having data and operations, and being able to inherit from other SCs

- You can also write abstract libraries and contracts

A code file can include other files, and contain the code of multiple SCs.

- However, the unit of code to be "installed" on the blockchain is always one and only one Smart Contract.

- The code, after compilation, becomes bytecode

SOLIDITY

Example of assignment.

```
uint256 aVariable;
uint256 anotherVariable = 10; // with assignment
```

*IMPORTANT: all variables declared **without assignment** in contracts are always initialized to zero or equivalent (empty string etc).

Note: 256 is the number of bits reserved for recording the number. The maximum value, in this case, is **(2^256)-1**

SOLIDITY

License: The first line is used to explain the license in terms of an identifier according to the SPDX standard.

```
// SPDX-License-Identifier: GPL-3.0
```

To know the list of possible licenses:
https://spdx.org/licenses/

Pragma: The first instruction of a solidity source is the declaration of the pragma to clarify in which **version of solidity** we have written our code.

```
pragma solidity ^0.8.1; // versione tra 0.8.1 e 0.9.0
pragma solidity >=0.7.0 <0.9.0; // versione compresa
                                // tra 0.7.0 e 0.9.0
```

SOLIDITY

Contract.

The contract **keyword** allows you to define a block of code similar to a class, in which there will be all the instructions and data necessary to implement the desired functionality.

The contract name must have the first letter capitalized.

Once the source has been compiled, each contract can be uploaded (deployed) to the blockchain independently of the other contracts in the code. **Each contract will be assigned a blockchain address.**

EXAMPLE

Contract: Sintax

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;


contract MyEmptyContract {

    // inside the contract


}
```

EXAMPLE WITH DOCS

Solidity supports **NatSpec** documentation **tags**.

https://docs.soliditylang.org/en/latest/natspec-format.html

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

/// @title Empty Contract
/// @author None
/// @notice This source code is empty!
/// @dev Nothing to describe for the dev.
contract MyEmptyContract {
    // inside the contract
}
```

You can export documentation via a compiler command:

```
solc --userdoc --devdoc example.sol
```

SOLIDITY

Solidity includes Conditional Statements and Loops. The language keywords for conditions, loops, and jumps are:

if
else
while
do
for
break
continue
return

EXAMPLE

Global variable

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;


contract storeValue {
    // inside the contract
    uint256 myVar = 10;
    /* the miavar name will be visible throughout the
        contract*/
}
```

SOLIDITY

Visibility: You can allow a global variable to be visible from outside the contract and **read via a direct request** to the blockchain. To do this, use the public modifier to write in the declaration after the data type.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;


contract Example {


    uint256 public myvar = 10;


}
```

SOLIDITY

Constructor
The constructor is a special function that is executed when the contract is deployed into the blockchain.

Syntax:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ConnractName {
    // ...
    constructor(arguments) {
        // something to be executed
    }
}
```

EXERCISE 1

Exercise: Using Remix, write a "basics.sol" file.

The file must contain code written in a version of solidity compatible with 0.8.1 up to, but excluding, 0.9.0. The file must be released under the GPL-3.0 license

The file contains the code to create a "SaveValue" smart contract that has a public **"value" variable of type uint16**.

This variable is assigned the value 10 during deployment, via the constructor.

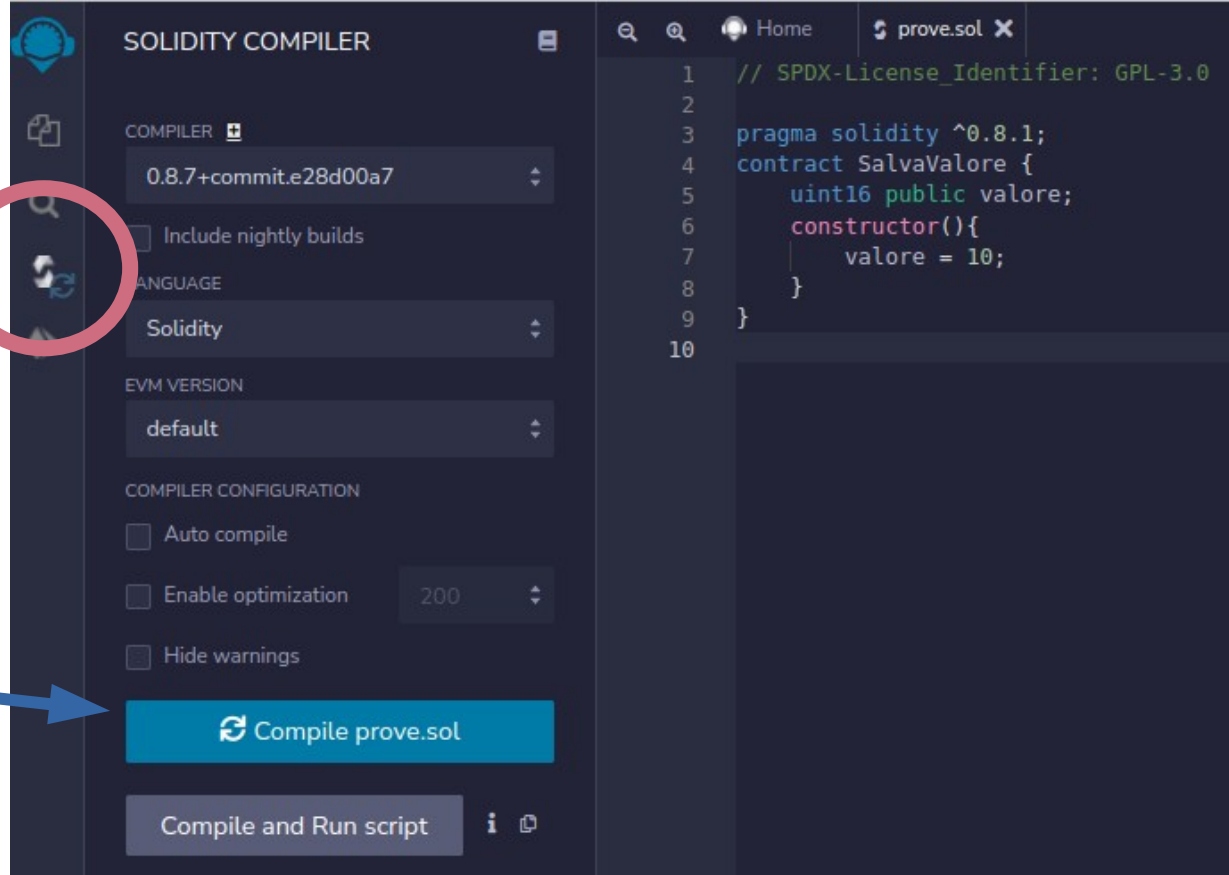# SOLUZIONE ES.1

basics.sol

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.1;

contract SaveValue {
    uint16 public myValue;
    constructor() {
        myValue = 10;
    }
}
```
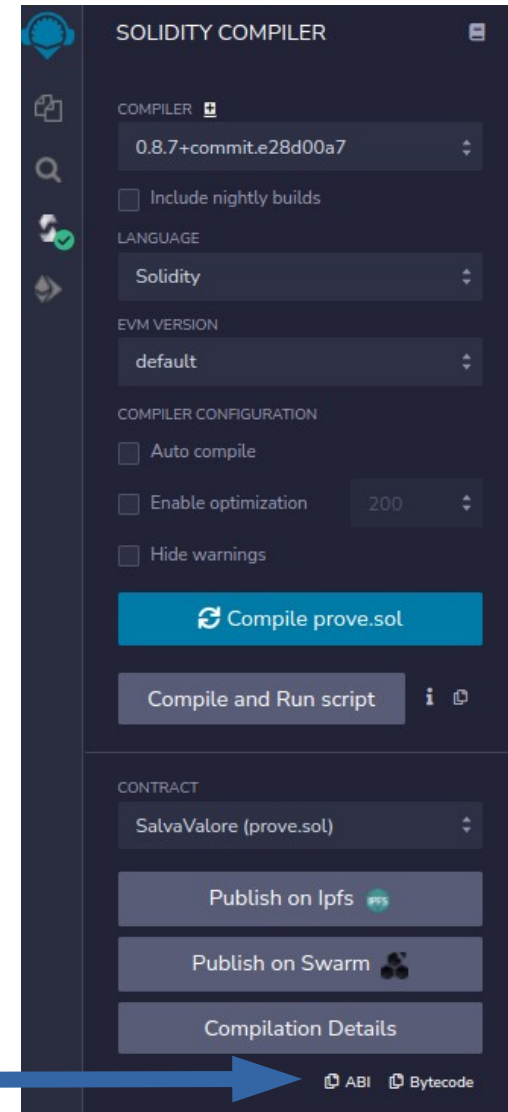
COMPILING

We compile the contract with remix.

COMPILING

After compilation, any errors and warnings will be shown. If everything goes well we can see the result of the compilation (the list of contracts).

Of each of these we can read the details of the compilation and take the **ABI and the Bytecode** (to copy and paste them in a text file or in another file).

DEPLOY

Remix allows you to deploy in different environments. We can choose to use

– virtual environments based on javascript
– an "injected" blockchain network using metamask
– a blockchain network via RPC
– other types of connections

Now let's deploy the contract on the virtual environment that simulates the post-merge version of ethereum

DEPLOY

We deploy the contract on the JavaScript VM virtual environment. We access the "**deploy and run transaction**s" tab, verify the settings and press on Deploy.

DEPLOY

After clicking on Deploy we get an instance of the contract.

Note that the account has consumed ether
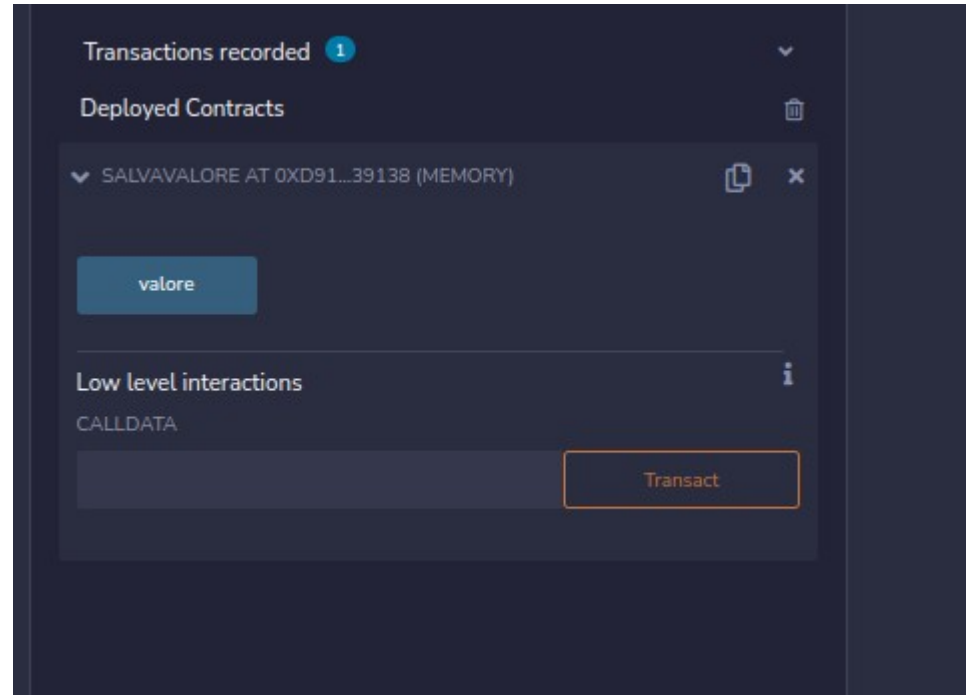
DEPLOY

Note: We use Injected Provider to deploy to the Ethereum blockchain (or testnet) via metamask.

Metamask will ask us to allow the connection between Metamask and Remix.
We have to choose one or more addresses to connect

On metamask, make sure you are on the Sepolia network.
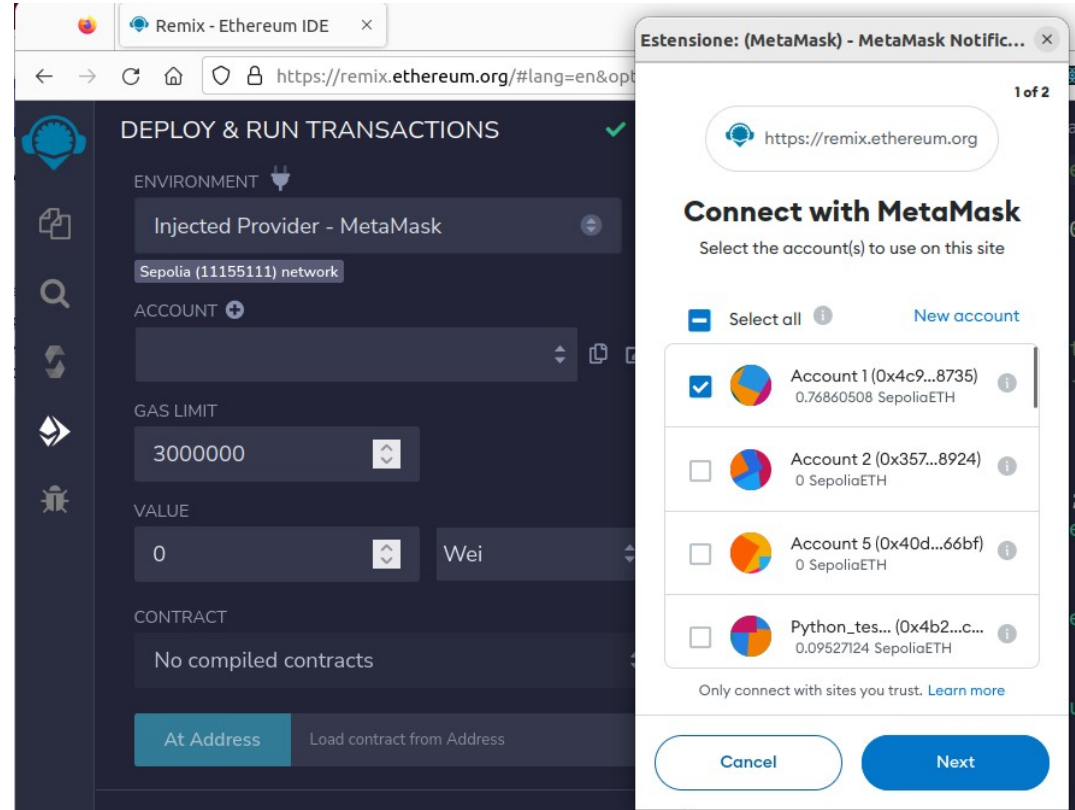
CALLS

Now we have the contract visible under the words "deployed contracts" in the Deploy and run tab.
We can press on the button with the name of the public variable. What we get is the blockchain response with the value.

The value of "myValue" is 10.

The reading operation is performed with a "call".
The data of our request are shown in the box at the bottom right.

OUTPUT DATA

```
[call]
from    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to SaveValue.myValue() 0xd9145CCE52D386f254917e481eB44e9943F39138

execution cost  23510 gas (Cost only applies when called by a
contract)

input  0x445...2015e
decoded input{}
decoded output  {
    "0": "uint16: 10"
}
logs    []
```

Functions.

The implementation of the code occurs through functions that act on global data or parameter values.

A function is defined with the function keyword, has a name, a set of parameters, and a set of modifiers, .

Basic syntax:

```
function nameFunction(parameters) modifiers {
    // body
}
```

Note: The constructor is a function executed in deploy time

PURE AND VIEW

Unless otherwise specified, the execution of the functions involves the consumption of gas. There are some special cases:

- **Pure** function: it is a function that does not modify any data and does not read any data from the blockchain. Useful for creating functions that operate on parameter values.

- **View** functions: it is a function that **reads data from the blockchain** but does not modify any data.

These functions are free only if it is not called by another function that modifies state OR if a contract calls the function via low-level not static call or delegatecall

VISIBILITY

Visibility can also be established for functions.

If **private**, they are "visible" and therefore called only by other functions of the contract itself.

If **public** they can also be called externally.

If **internal** they can be called from the contract and from the contracts that inherit the contract.
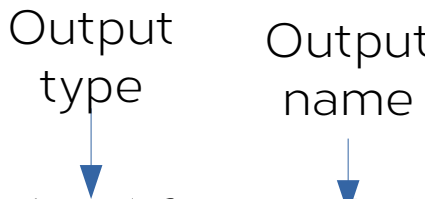
If **external** they can only be called from **outside the contract** but not from its other functions.

SOLIDITY

To return a value you need to declare the data type.
Example: we create the sumTen function which has no arguments and which returns the value of the global variable myValue + 10.

Output type

Output name

```
function sumTen() public view returns(uint16 result){
    return 10+myValue;
}
```

Let's **add this function** to the contract, compile and deploy again.
NOTE: we need to delete old instances.
NOTE2: what changes if the function is defined as internal?

36

SOLIDITY

Our contract now has two blue buttons.

By clicking on sumTen we call the function (execute it) and read the result under the button.

Let's see the output of the call:

```
decoded output    {
        "0": "uint16: result 20"
                          }
```

index   tyoe   name   value
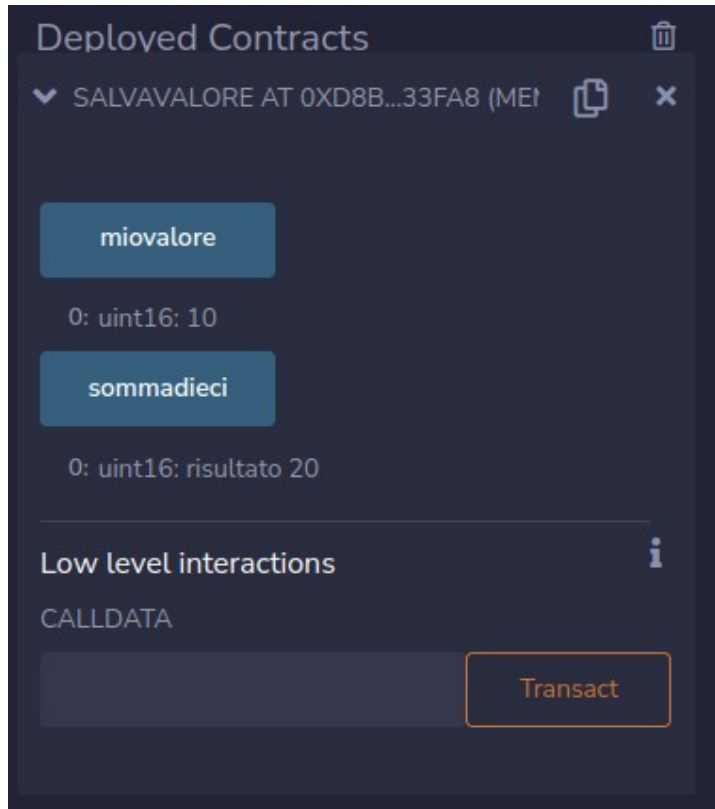
EXAMPLE

Example of a function that takes two arguments and returns a Boolean and performs checks before executing another function.

```
function safeTransfer(address _to, uint256 _value) public returns (bool){

    uint senderBalance = balances[msg.sender];
    if (senderBalance >= _value && _value > 0) {
        senderBalance -= _value;
        balances[msg.sender] = senderBalance;
        balances[_to] += _value;
        transfer(msg.sender, _to, _value);
        return true;
    }
    return false;
}
```

The name is optional

38

EXERCISE 2

In the SaveValue contract inside prove.sol we add a public function "**writeValue**" that takes a 16-bit unsigned integer as an argument. The function returns nothing

The function records the value of the argument in a parameter **named val** and saves the value of val in the global variable **myValue**.

EXERCISE 2 SOL.

```
function writeValue(uint16 val) public{
    myValue = val;
}
```

We add this function to the contract and compile and deploy it again (possibly remove the old contract by clicking on the X on the deployment section)

What happens if we try to declare it pure or view?

SOLIDITY

Now our contract has a different colored 'writeValue' button because executing this function **incurs a cost**.

We insert the value 50 in the box corresponding to our val parameter and press the writeValue button.

Executing the function involves the **consumption of gas** and therefore ether.
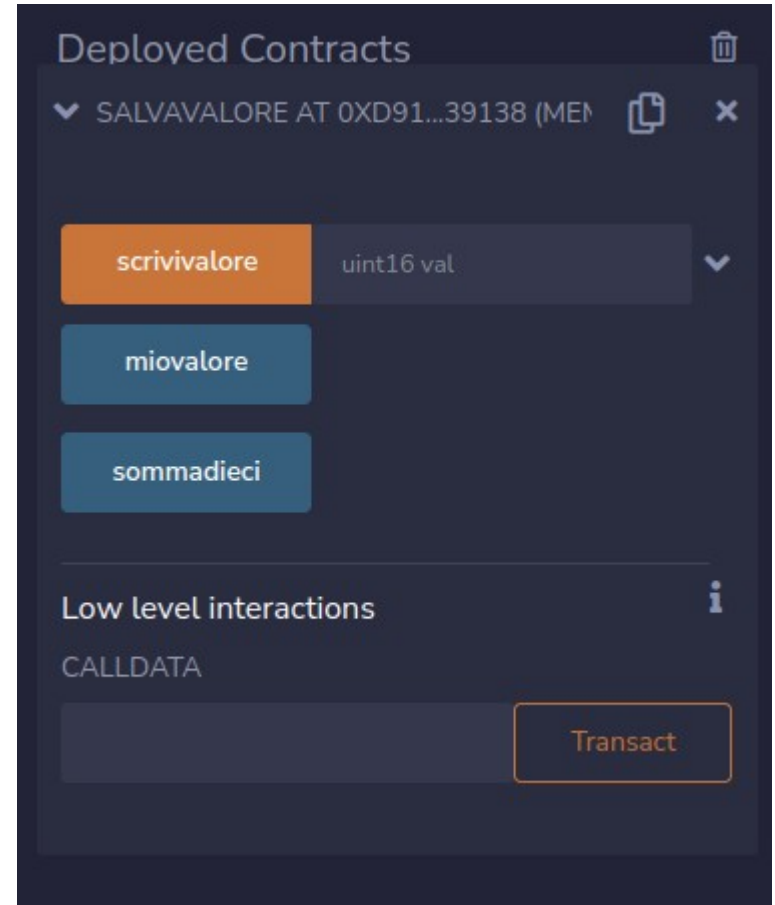
EXERCISE 3

We want to add to the contract the ability to record the number of writings that have been made.

– Add a public variable **numWriting** of type uint16 to the SaveValue contract.

– Add a private **incrementNumWriting** function with no arguments to the contract that adds 1 to the value of numWriting.

– Edit the **writeValue** function so that it calls the incrementNumWriting function when called.

EXERCISE 3 - SOL.

```
uint16 public numWriting;
function incrementNumWriting() private{
    numWriting +=1;
}


function writeValue(uint16 _valore) public{
    incrementNumWriting();
    myValue = _valore;
}
```

EXERCISE 4 - UINT

Now let's experiment with the characteristics of **uint types**.

Let's create a new **Test_uint8 contract** that contains:

– a public variable of type uint8 (8 bit) named '**little**'.
– a **constructor** that assigns 250 to the variable 'little'.
– an '**increment**' function without arguments and publish that adds 1 to the variable 'small'
– a function '**assign5**' without arguments and public that assigns 5 to the variable 'small'
– a '**decrement**' function without arguments and publish that subtracts 1 from the variable 'small'

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.1;
contract Test_uint8 {
    uint8 public little;
    constructor() {
        little = 250;
    }
    function increment() public{
        little += 1;
    }
    function assign5() public{
        little = 5;
    }

    function decrement() public{
        little -= 1;
    }
}
```

45

EXERCISE 4 - INFO

After deploying we can try to execute the increment function and then read the value of little.

We find that if the value of the "little" variable is 255, the next execution of the increment function will cause an error.

This is because uint8 can hold unsigned numbers that are representable with **8 bits (0 to 255).**

What is the maximum for uint32 and uint256?

Now let's try to use the function **assign5** and then decrease. We will see that we cannot subtract 1 from zero.

EXERCISE 4 – INFO

In old versions of solidity (for example 0.4) the execution of the function would not have led to an error.

Let's try changing the pragma to ^0.4.1 and compile and deploy the code.
We find that now, when the small variable is worth 255, at the next increment **its value returns to zero.**

This behavior could generate unexpected events (bugs) in the execution of the code. To avoid these problems, a library (another contract) called "**safemath**" was imported

SOLIDITY: TYPES

Among the simple data types of solidity we have:
- **address** which contains a blockchian address
- **bytes32** (or other lengths) containing raw data
- **string** : which contains text strings
- **uint256** (or other dimensions) containing unsigned integers
- **bool** which contains true or false

Structured data.
on solidity we can use the **arrays** and **structs**.

There is also a key-value data type called **mapping**.

ADDRESS TYPE

The **address type** contains an address compatible with the Ethereum blockchain. At any time we can recall two system variables of type address.

msg.sender: is the address of the person who interacts directly with the contract.

tx.origin : is the address of performing the original interaction towards the blockchain. It is used in case of chain calls.

global names: in addition to msg and tx there are other global variables.

https://docs.soliditylang.org/en/latest/units-and-global-variables.html

ADDRESS TYPE

Contract address,
We can query the blockchain to find out the address of a contract. Contract to address type conversion is used. This means that the instance of a contract also contains its address.

Syntax:
address(Contract contract)

I can get the address of the contract being worked on using the keyword this.

address(this)

BALANCE

Balance of an address.
The balance is the quantity of the main cryptocurrency expressed through the smallest unit (for example wei on ethereum) associated with an address.

The address balance method allows reading the balance of any address.

```solidity
address owner;
uint256 balance1 = owner.balance;


address(this).balance; // balance of contract
msg.sender.balance; // balance of the caller.
```

VALUE

Value of a message.

The message always has an associated value in ether which by default is zero. The value of the message **is transferred to the smart contract.**

Within the code, you can retrieve the contract value via the msg.value variable
.

msg.value; // value of the message.

Note: during the deploy, the value of the message is transferred to the new contract account.

PAYABLE

Address and payable functions.
Within the contract we can distinguish between "normal" addresses and **payable addresses**. Solidity allows payable addresses **to send cryptocurrency** to the contract and **receive cryptocurrency** from the contract
.

address payable user;

To create a function that when called allows cryptocurrency to be sent to the contract I must specify **the payable modifier.**


```
function deposit() public payable {
        totalAmount += msg.value;
    }
```

TRANSFER

Sending cryptocurrency from the contract to a payable address. The simplest but **not recommended way** is to use the address transfer method. Suppose we have a payable dest address:

```
dest.transfer(amount);//the contract sends the "amount" to dest.
```

The **recommended pattern** for sending cryptocurrency from the contract is based on the address call method which we will delve into later.

```
(bool success, ) = dest.call{value: amount}("");
require(success, "Transfer failed.");
```

EXERCISE 5

Create a Moneybox contract that contains a variable uint256 deposits the number of deposits made and an address payable owner variable. The constructor stores the address of the creator of the contract in owner.
The contract has three public functions:

- A payable "**deposit**" function that adds one to the number of deposits if the message value is greater than zero.
- A "**withdrawal**" function that sends the collected funds to the creator of the contract via transfer.
- A "**getDeposits**" view function that returns the value of deposits.

ESERCIZIO 5 SOL.

```solidity
// // SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.1;

contract Moneybox{
    uint256 deposits;
    address payable owner;
    constructor(){
        owner = payable(msg.sender);
    }
    function deposit() public payable {
        if (msg.value > 0) {
            deposits += 1;
        }
    }
    function withdrawal() public {
        owner.transfer(address(this).balance);
    }
    function getDeposits() public view returns(uint256 num_deposits){
        return deposits;
    }
}
```

EXERCISE 5 - INFO

What happens if the transfer function is called when its argument is greater than the balance of the contract?

Let's try to modify the withdrawal function to pass the amount you want to withdraw.

In case of failure, **the transfer** function **raises an exception (revert)**, adds a fixed commission of 2300 gas.

SEND

The send address method is similar to the transfer method but behaves differently.

```
anAddress.send(uint256 importo) returns (bool)
```

Sends the amount passed as an argument in Wei to the address, and in case of failure **it does not revert but returns false** and adds a fixed commission of 2300 gas

CALLS TO CONTRACTS

CALL

Executes a low-level call. with a bytes payload.

```
<address>.call(bytes memory)
                    returns (bool, bytes memory):
```

Returns true on success, false on failure, and return data;
It forwards all available gas to the call.

The call **executes a blockchain transaction** of which we can edit every parameter. The transaction involves the **consumption of gas.**

CALL

Example: call is used to **transfer ether as a secure pattern**, instead of transfer.

To do this, we can specify the value to send along with the transaction by adding some tags.

For example, to send 100 wei and limit the gas to 5000 we use this syntax:

```
(bool success,) = anAddress.call{value: 100, gas: 5000}
```

CALL - PAYLOAD

The payload in bytes passed as an argument must correspond to a function of the called contract.

The first 4 bytes of the payload are the first 4 bytes of the SHA3 hash of the **function "signature"** (only the complete name and types of the parameters, without spaces), as reported in the ABI.

Example of calculation of the first part of the payload:

```
bytes4(keccak256("setX(uint256)"))
```

The parameters follow, in blocks of 32 Bytes (256 bits)

CALL - PAYLOAD

f we know the abi of the called contract we can use the system function **"abi.encodeWithSignature"** which converts a string with the prototype of the function (signature) followed by the value of the parameters in the payload into bytes.

```
bytes memory payload = abi.encodeWithSignature(
        "functionName(type1,type2,type3)", param1, param2, param3);


(bool success, bytes memory result) = targetAddress.call(payload);
```

Docs:

https://docs.soliditylang.org/en/v0.8.19/units-and-global-variables.html#abi-encoding-and-decoding-functions

STATICCALL

Staticcall is designed to make calls **with the certainty that they will not change the state of the blockchain**.

```
<address>.staticcall(bytes memory)
        returns (bool, bytes memory):
```

Allows you to call view functions of other contracts.

It works like the call but does not allow you to specify the value.

Raise an exception and end with a "revert" if the call alters the state of the contract.

CALL AND STATIC CALL
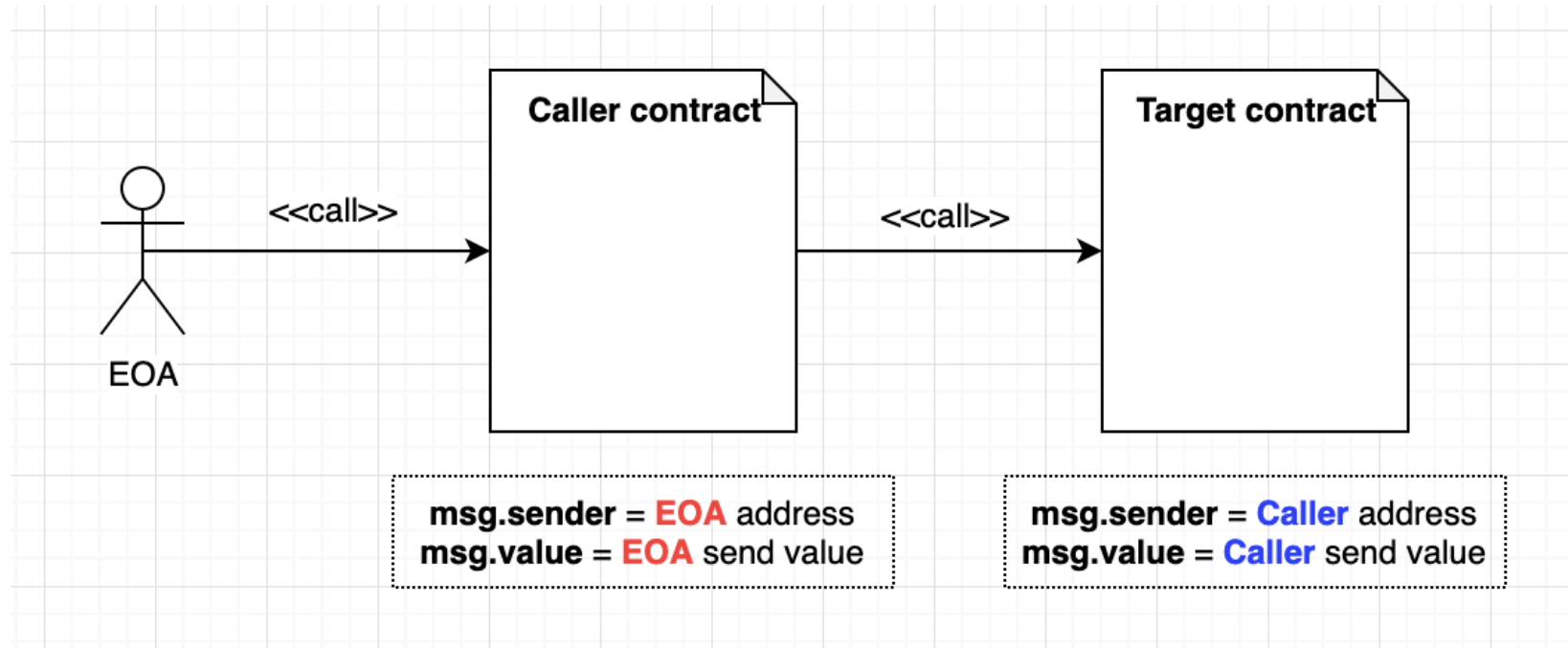
DELEGATECALL

The delegatecall **uses only the code of the called contract,** but the caller's storage and Ether.

It can be used to call already deployed contracts containing libraries of useful functions.

```
<address>.delegatecall(bytes memory)
        returns (bool, bytes memory):
```

Executes a low-level call with the bytes memory type payload.

Returns true on success, false on failure, and returns data; forwards all available gas to the call.

DELEGATE CALL
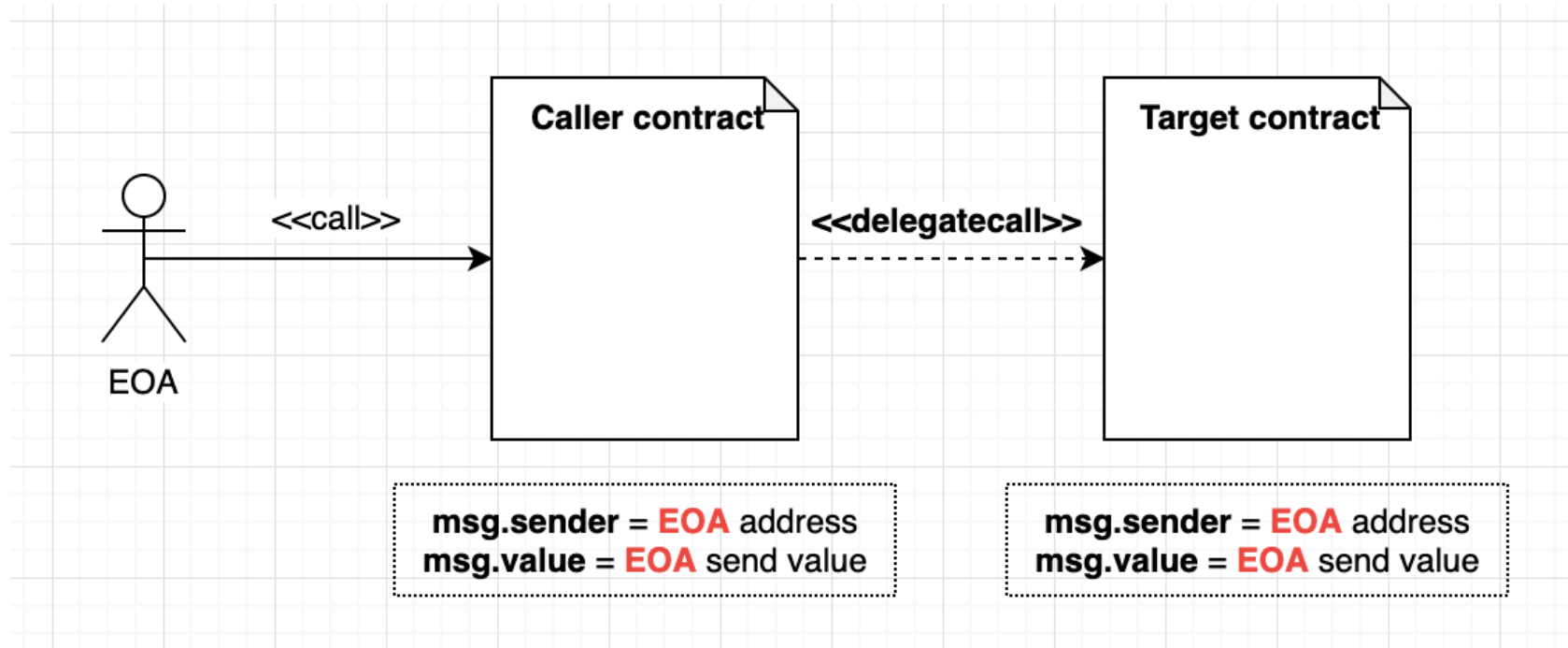
EXERCISE 6

Write two contracts in the same **targetAndCaller.sol** file:

-A **Target** contract that has a public `checkCaller` function with no arguments that returns a boolean based on the comparison between the msg.sender and the tx.origin.

-A **Caller** contract with one function:
`callTarget(address _target)`
which calls the _target's staticcall method to execute the `checkCaller` function.

EXERCISE 6

The callTarget function of the second contract returns the **bytes memory value returned by the staticcall.**

```
bytes memory payload = abi.encodeWithSignature(
        "functionName(type1,type2,type3)", param1, param2,
param3);
```

Deploy the first contract and run `checkCaller`

Deploy the second contract and execute `callTarget` passing the address of the first contract.

EXERCISE 6 - SOL

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract Target{
    function checkCaller() public view returns(bool check) {
        return msg.sender == tx.origin;
    }
}


contract Caller{
    function callTarget(address _targetAddress) public view returns(bytes memory){
        // create the function payload (no argumnents)
        bytes memory payload = abi.encodeWithSignature("checkCaller()");
        // execute the call
        (bool success, bytes memory result) = _targetAddress.staticcall(payload);
        return result;
    }
}
```

EXERCISE 6 – INFO

If we call the checkCaller function we see that **it returns True.**
But if we call callTarget and examine the outputs we can see that the result is a **hexadecimal number corresponding to 0, i.e. false,**

The **returned value** is of type bytes. It is a variable length data type.
In this case, the data fits in a **single word** (which is always 32 bytes).

The staticcall does not change the state of the blockchain and the caller function is a view.

Let's try to change the Caller contract, replacing the word staticcall with delegatecall. **What happen?**

How can you transfer some value from the Caller contract to the Target?

SOLIDITY: COMPLEX DATA

Arrays: similar to Java arrays. They use the notation with "[ ]" fixed size in compilation:

```
bool[3] boolVet;
boolvet[1] = true;
int[5][3] v2; // array of 3 arrays of 5 int each.
v2[0]=[1,-72,3,13,4];
delete v2; // sets all elements to zero
v2.length;  // number of elements
```

There are also dynamic arrays.

RECORD - STRUTTURE

Solidity allows the declaration of structured data of type struct.

Since Solidity 0.6.* they can be declared at file level (before contracts)

Fields are accessed with "dot" notation. **They are new types**.

 Example:

```
struct Person{
    string name;
    uint birthdate;
    enum gender;
}
Person aPerson;
aPerson.name = "Andrea";
```

They are passed by reference.

MAPPING

Mapping.

Mapping is a way to record a collection of data on solidity.

Allows you to create a **"key-value"** association between two pieces of data. The first is a reference (equivalent to an index of an array) which we call a key, the second is the value (equivalent to the contents of a cell of an array).

In solidity, the mapping is defined by specifying what data type the reference is and what data type it refers to.

MAPPING

Syntax of Mapping

mapping(**referenceType** => **referredType**) **public** nameMapping;

To access the values recorded by the mapping we can use the name of the mapping followed by square brackets, writing a value of the reference type as the key.

Example: we associate a uint16 data with a uint8 data.

```
mapping(uint8 => uint16) public mymapping;
mymapping[10] = 500;        // creates a new value for the key 10
readValue = mymapping[10]; // assing the value of the key 10
```

MAPPING

Struct and Mapping: example from 3_ballot.sol (remix).

```
struct Voter {
    uint weight; // weight is accumulated by delegation
    bool voted;  // if true, that person already voted
    address delegate; // person delegated to
    uint vote;   // index of the voted proposal
}
mapping(address => Voter) public voters;
```

We can see that the mapping creates an association between a data of a certain type, to another data of another type (even structured).
The behavior is similar to the python dictionary

MAPPING

Voters

0x018123

0x121b2a

0xa1d92c

....

mapping

Voter

weight = 1

Voted = false

Delegate = 0x1234

Vote = 1

Voter

weight = 3

Voted = true

Delegate = 0x1334

Vote = 0

Voter

weight = 1

Voted = true

Delegate = 0xa334

Vote = 1

EXERCISE 7

We create a "Certificates" contract that records some information for each certificate generated for the attendees of a course. The contract has a "price" variable with the value of 1/1000 of ether. The contract implements a struct "certificate" and a mapping as following:

```
struct certificate{
        uint8 grade;
        address professor;
        uint timestamp;
        bytes signature;
    }
mapping (address => certificate) certificates;
```

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.1;

contract Certificates {
    uint price = 1 ether / 1000;
    struct certificate{
        uint8 grade;
        address professor;
        uint timestamp;
        bytes signature;
      }
     mapping (address => certificate) certificates;;
}
```

EXERCISE 8

In the Certificates contract code we add an "**attendance**" mapping between address and boolean,

we implement a public function without arguments:

addMe ( )

The function sets the value of the "attendance" mapping for the caller's address (msg.sender) to true.

The function also accepts payment in wei and will therefore be **payable**.

```
mapping (address => bool) attendance;

function addMe() public payable{
        attendance[msg.sender]=true;
    }
```

Let's deploy and execute the addMe function.

EVENTS AND LOGS

Events are specific messages that the blockchain emits upon request of smart contracts. They are useful for creating **"log" documentation** in the blockchain and for **capturing events** to report off-chain and to the front end. We use the keyword **event** to declare an event and **emit** to invoke it.

Declaration syntax (to be written in the main code of the contract)

```
event eventName(declaration of arguments to emit);
```

Call syntax (to be written inside a function)

```
emit eventName(values to emit);
```

LOGS

During code execution, events can be generated, with associated data.

The type and data relating to events are stored within the blockchain recording of the transactions that generate them.

There is an index (**Bloom filter**) stored on the blockchain to find events.

This data structure (**logs**) can be read from outside the blockchain, **but not from within the SC functions**.

Logs can be used to record information no longer useful for the execution of SC, but **of interest to external observers**.

LOGS

Example: we declare the "**valueUpdated**" event which emits a uint8 named "newValue".

```
event valueUpdated(uint8 newValue);
```

Example: we emit the `valueUpdated` event which takes the `newVal` variable as input.

```
emit valueUpdated(newVal);
```

Note: You can declare up to 3 event arguments as **indexed**. These will be used in the calculation of the event signature (you can filter the search for events using those parameters). Those not indexed will be included in the event bytes

EXERCISE 9

In the **Certificates** contract we declare the "**newAttendance**" event which emits an address named "**attendee**" and a uint named "**payment**".

In the addMe function we **emit** the newAttendance event which takes as input the caller's address and the msg.value.

EXERCISE 9 SOL

```
contract Certificates {
    event newAttendance(address attendee, uint payment);
    uint price = 1 ether / 1000;
    struct certificate{
        uint8 grade;
        address professor;
        uint timestamp;
        bytes signature;
    }
    mapping (address => certificate) certificates;
    mapping (address => bool) attendance;
    function addMe() public payable{
        attendance[msg.sender]=true;
        emit newAttendance(msg.sender, msg.value);
    }
}
```

ERROR MANAGEMENT

Solidity includes some error generation functions that help determine the behavior of the smart contract.

assert`(bool condition)` : causes a *Panic* error and therefore the reversal of the change of state if the Boolean condition is not satisfied (if it is false). To be used for internal errors.

require`(bool condition, string memory message)`: performs recovery if the Boolean condition is not satisfied. Use for errors in inputs or external components. The text string is optional and allows you to show the error.

revert`(string memory reason)` : stops execution and rolls back state changes. The string is optional and allows you to show the reasons for the error

REQUIRE

Require: con questa parola chiave è possibile esplicitare alcune condizioni da rispettare durante l'esecuzione del codice. Si utilizza dentro le funzioni.

Sintassi:

require`(boolean condition, "output string")`

If the condition is false, execution is aborted and the message is produced as the second argument.

Example:

require`(balance > 0, "insufficient balance")`

EXERCISE 10

In the contract `Certificates`, add two require statements in the function addMe to:

- prevent a user to add himself two or more times
- check that the value of the message is equal to the price.

EXERCISE 10 SOL.

```
function addMe() public payable{
    require(!attendance[msg.sender],  "you  are  already
registered");
    require(msg.value == price, "invalid amount");
    attendance[msg.sender]=true;
    }
```

SOLIDITY MODIFIER

On solidity it is possible to create new function modifiers.

On solidity, modifier definitions are portions of code with syntax similar to that of functions and which are executed before the body of each modified function is executed.

I can reuse the same modifier in any function, writing its name in the declaration, after the other modifiers (e.g. public, pure, etc)

The aim is **to shorten the functions** and create more correct, tidy and **readable code.**

MODIFIER

Modifier. syntax

```
modifier nameModificatore (argument declaration){
    // modifier body
    _; // represents the jump to the function body
}
```

MODIFIER - EXAMPLE

Suppose we have a contract with many functions that can only be executed from the address saved in the owner variable.
For each function I should write a line containing the require.

```
require(msg.sender == owner, "access denied");
```

Instead of writing the same require in all functions, I can implement a modifier.

```
modifier onlyowner(){
    require(msg.sender == owner, "accesso negato");
    _;
}
```

MODIFIER - EXAMPLE

Once the onlyowner modifier has been defined, I can use it in the function declaration.

```
function modifyValue(uint32 newValue) public onlyowner{
      // function body
}
```

I insert the modifier here

EXERCISE 11

In the Certificates contract implement a modifier onlyowner which requires, with a require, that the msg.sender is the same one registered on owner.

NOTE: We will need an "owner" global variable of type address and a constructor that assigns the msg.sender value to owner.

Create a new mapping "**professors**" (address to bool) and a public function "**addProfessor**" that uses the modifier onlyowner that, given an address, sets to True the mapping professors for the given address.

EXERCISE 11: SOL.

```solidity
address owner;
mapping (address => bool) professors;

constructor(){
    owner = msg.sender;
}

modifier onlyOwner(){
    require(msg.sender == owner, "only the owner");
    _;
}

function addProfesor(address professor) public onlyOwner{
    professors[professor] = true;

}
```

CONTRACTS CREATION

CONTRACTS CREATION

A smart contract can create instances of other contracts.

To create an instance we use the new keyword

syntax:

```
ContractName cInstance = new ContractName(arguments);
```

CONTRACTS

Example: Suppose we have this definition:

```
Contract Deposit {
    address owner;
    constructor(){
        owner = msg.sender;
    }
}
```

In another contract we can write the following code to create a new smart contract on the blockchain (deploy)

```
Deposit d = new Deposit();
```

The Depoisit argument will be the constructor parameters.
The msg.sender will be the calling contract.

FACTORY PATTERN

To create a new contract you can use the factory pattern.
A contract factory is designed to deploy contract instances on demand.

```solidity
contract Clone{
    address public owner;
    constructor(){
        owner = address(tx.origin); //msg.sender;
    }
}


contract CloneFactory{
    uint256 public number_Of_clones;
    mapping(uint => address) clones;

    function createClone() public returns(address aClone) {
        Clone c = new Clone();
        number_Of_clones += 1;
        clones[number_Of_clones] = address(c);
        return address(c);
    }
}
```

IMPORT

The import statement allows you to insert the contents of another file.

```
import "fileName";
import * as symbol from "fileName";
import "fileName" as symbol;
```

In the 2nd and 3rd cases, the global entities contained in the file are imported as: `symbol.EntityName`

it is also possible to import a single entity from the file.

```
import {symbol1 as alias} from "filename";
```

And from a URL (with REMIX)

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/access/Ownable.sol"
```

IMPORT

I can import multiple solidity files into my contract. In this way it is as if I put all the code already implemented in the imported sources in my file.

Example

```
import "./aSourceFile.sol";
```

I can also import from github

```
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol";
```

SOLIDITY: INHERITANCE

Each contract can **inherit** all the functionality or declarations of other contracts (and libraries) after importing the code. Solidity allows **multiple inheritance**.

We use the keyword is.

```
import "./aSourceFile.sol"
contract MyContract is AnotherContract, AThirdContratct {

    // body


}
```

INHERITANCE

All the source codes of the inherited contracts **must be available** to the compiler (it is not inherited from the instances in the blockchain but it is inherited at the code level).

If contract A inherits from B, B's **code is copied into contract A**. A will be created and inserted into the blockchain.

With **"is"** you inherit all the state variables and the **internal and public functions**.

```solidity
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

contract mySupercontract{
    uint internal i;
    uint private priv;
    uint public pub;
}
contract mySubcontract is mySupercontract{
    function set() public {
        i = 10;
        //priv = 10; It is not visible
        pub = 10;
    }
}
```

INHERITANCE EXAMPLE 2

```solidity
contract Ownable {

    address private owner;       // variabile di stato
    constructor() {              // costruttore
        owner = msg.sender;      // address di creazione
    }
    modifier onlyOwner {   // funzione modifier
        require(msg.sender == owner);
        _;
    }
}
contract Courses is Ownable {
    // The Ownable code becomes part of my contract
    // The code here can use "onlyOwner".
}
```

INHER. AND CONSTRUCTOR

When the contract from which we inherit has a constructor we are obliged to call it (with the appropriate parameters) in the derived contract.

There are two main ways of calling the constructor.

1. during the contract declaration phase, passing the values to the contract from which it is inherited to set the parameters in a fixed manner.

```
contract DerivedConstructor is BaseConstructor(values)
```

2. via the constructor of the derivative contract, leaving the parameter setting to the creator of the derivative contract.

```
contract DerivedConstructor2 is BaseConstructor{

    constructor(uint256 _number) BaseConstructor(_number){}
```

EXAMPLE

```
contract BaseConstructor{

    uint256 public value;

    constructor(uint256 _value){

        value = _value;

    }

}

contract DerivedConstructor is BaseConstructor(10){

    constructor() {}

}

contract DerivedConstructor2 is BaseConstructor{

    constructor(uint256 _number) BaseConstructor(_number) {}

}
```

PROXY PATTERN

In Ethereum, creating a contract and recreating a data structure containing data have significant costs.

Suppose we have a SC A that is used by other SCs and has a significant amount of data in its storage.

Since A cannot be changed in any way, in case bugs or defects are found, to remedy this, a new contract A' must be **installed**, copying all data from A into it.

Such an operation would be **very expensive in terms of gas**.

Furthermore, **all SCs using A would have to change its address to that of A'**, and thus the change could cascade.

109

PROXY PATTERN

The Proxy pattern is a solution to this problem.
A proxy P is a contract that keeps in storage:
- The address of the delegated SC A, but then of A', A", etc.
- The owner's address, authorized to change the mentioned address

External contracts send messages to P, which redirects them first to A, then possibly A', A", etc.

In this way, replacing A with A', just change the address within P, and all external contracts will continue to work.

PROXY PATTERN

This **can be achieved** using:

The **fallback function**, which allows you to send a call to P that P will forward to the delegate without the function signature belonging to P's ABI

The **delegatecall**, which allows P to pass the message so that the delegate sees the original one as the caller and not P

PROXY PATTERN

Proxy scheme:

Since replacing A with A' involves recreating all of A's storage, a very expensive operation:

you can put the storage in P, leaving the control logic (to be replaced) in A, A'.

PROXY PATTERN

To manage storage without having to entrust it to the proxy, the **Eternal Storage** pattern is used,
This pattern involves using a third CS that contains the data from contract A (and its updates)

Stores the storage structure and is inherited by proxy and logic contract

PROXY SOLUZIONE

The messages include the address of the receiving SC, gas data, any ETH sent, the payload

Proxy P limits itself to registering the address of the SC and the address of the owner, the only one authorized to change the address of the SC to forward the msg to.

- P's fallback, using EVM low-level assembly code:
- recovers the payload of the failed call (which includes the first 4 bytes of the function signature hash)
- it sends it back as a msg to the address of the SC for which it is proxy, via a delegatecall
- retrieves the return payload, and returns it to the address of the initial call

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/Proxy.sol

PROXY UPGRADABLE

The address of the "implementation" contract is recorded on a specific memory slot (ERC 1967 standard) obtained as:

bytes32(**uint256**(**keccak256**('eip1967.proxy.implementation')) - 1)

Basic implementation uses low-level code:

```
bytes32 constant _IMPLEMENTATION_SLOT =
        0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
struct AddressSlot {
        address value;
    }
function getAddressSlot(bytes32 slot) internal pure returns (AddressSlot storage r) {
        assembly {
            r.slot := slot
        }}
```

Openzeppelin complete implementation:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/ERC1967/ERC1967Upgrade.sol

TOKEN

ERC20 fungible tokens were proposed by Fabian Vogelsteller and Vitalik Buterin in 2015.

https://eips.ethereum.org/EIPS/eip-20

Fungible Token ERC20 provided under MIT license by Openzeppelin.
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

Note: There is no mint public function.

CUSTOM ERC20

Example. We import the ERC20 contract from openzeppelin and add the constructor.

```
import     "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC20/ERC20.sol";


contract MyToken is ERC20 {


    // custom body
    constructor()  ERC20(){}
}
```

We can add our own mint function or attribute all Token to the contract creator.

EXERCISE 12

Import the ERC20 implementation into your code and create a contract called SchoolToken that calls the ERC20 constructor with the name "BCSCHOOL" and the symbol "BST". The contract implements a function called mint which takes as input an address and a uint256 and calls the _mint function of the basic contract (ERC20).
Also implement a modifier to allow only the contract creator to mint new tokens.

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC20/
ERC20.sol"
contract SchoolToken is ERC20 {

    // custom body

}
```

EXERCISE 12 SOL.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.7;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol";

contract SchoolToken is ERC20 {
    address owner;
    error notTheOwner(address _address);
    constructor() ERC20("BCSCHOOL", "BST")  {
        owner = msg.sender;
    }

    modifier onlyOwner(){
        if (msg.sender != owner) revert notTheOwner(msg.sender);
        _;
    }

    function mint(address recipient, uint256 quantity) public onlyOwner{
        _mint(recipient,quantity);
    }
}
```

STANDARD ERC-721

The ERC721 standard allows the management of non-fungible tokens. Submitted by William Entriken, Dieter Shirley, Jacob Evans and Nastassia Sachs in January 2018.

https://eips.ethereum.org/EIPS/eip-721

Implementations of NFT ERC721:

Openzeppelin:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol

Nibbstack (simpler)
https://github.com/nibbstack/erc721/blob/master/src/contracts/tokens/erc721.sol

STANDARD ERC-721

Function	Description	Interface
balanceOf	Number of tokens for an owner	ERC-721
ownerOf	Owner of a given NFT	ERC-721
safeTransferFrom	Transfer token from one owner to another	ERC-721
transferFrom	Non-safe implementation of transfer	ERC-721
approve	Approve asset transfer for third-parties	ERC-721
setApprovalForAll	Approve asset transfer of all third-parties	ERC-721
getApproved	Get approved address for a token	ERC-721
isApprovedForAll	Query if addess is approved for transferring from another address	ERC-721
supportsInterface	Check if the contract supports a function	ERC-165
onERC721Received	Implement if the contract is able to receive erc721 tokens	ERC721TokenReceiver
name	Name of the token	ERC721Metadata
symbol	Symbol of the token	ERC721Metadata
tokenURI	URI for the metadata json	ERC721Metadata

STANDARD ERC-721:

The standard establishes 3 **events**:

- Transfer(address _from, address _to, uint256 _tok)
- Approval(address _owner, address _approved, uint256 _tok)
- ApprovalForAll(address _own, address _app, bool _ok)

Tokens are uniquely identified by an integer uint256 and a second token cannot be created with the same number.

Each token has a proprietary (transferable) address and an owner can own several tokens.

Furthermore, the **owner can approve (and revoke) the transfer** of a single token, or of all his tokens to a given address.

NOTE: **How** new tokens (mint) are created depends on the policy adopted by the implementation.

CUSTOM NFT

Example: Similarly, we want to implement an ERC721 contract, adding the constructor.

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol";
contract MyToken is ERC721 {
    // custom body
    constructor() ERC20("theToken", "TTK") {}
     function  mint(address  recipient,  uint256  quantity) public{
        _mint(recipient,quantity);
    }
}
```