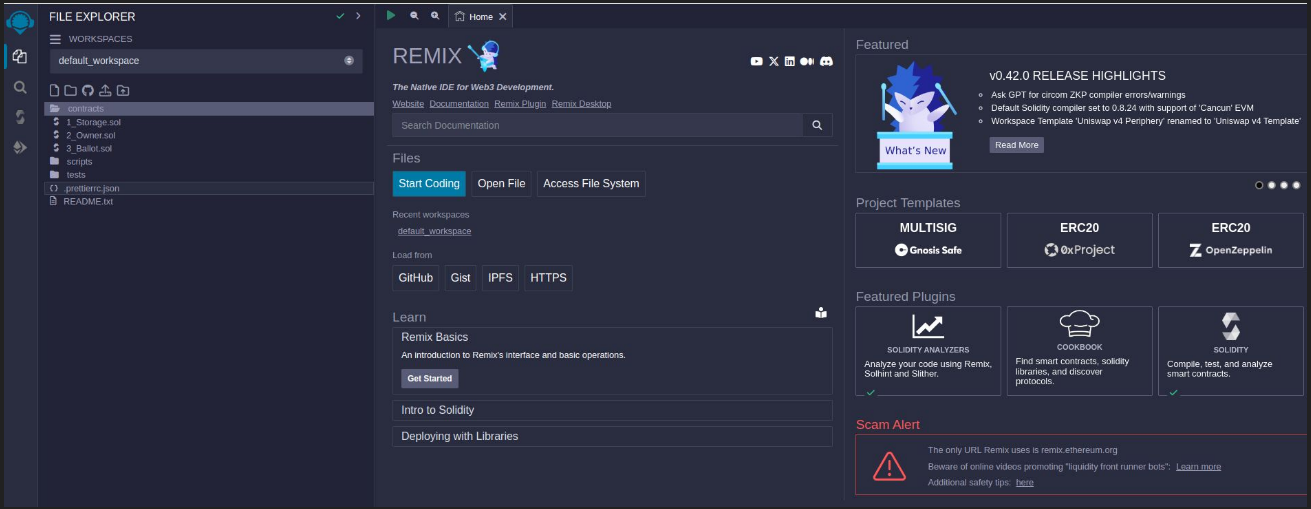
Dev Blockchain - Solidity

Presentation

- Programming Language created by Ethereum team to code smart contracts
- We will first use Remix : a visual web IDE to code SC
- Visit the site : https://remix.ethereum.org/

Remix

First of all just, we are just going to explore Remix



FILE EXPLORER

WORKSPACES

default_workspace

contracts
1_Storage.sol
2_Owner.sol
3_Ballot.sol
scripts
tests
.prettierrc.json
README.txt

Home

REMIX

The Native IDE for Web3 Development.

Website Documentation Remix Plugin Remix Desktop

Search Documentation

Files

Start Coding | Open File | Access File System

Recent workspaces

default_workspace

Load from

GitHub | Gist | IPFS | HTTPS

Learn

Remix Basics

An introduction to Remix's interface and basic operations.

Get Started

Intro to Solidity

Deploying with Libraries

Featured

What's New

v0.42.0 RELEASE HIGHLIGHTS

- Ask GPT for circom ZKP compiler errors/warnings
- Default Solidity compiler set to 0.8.24 with support of 'Cancun' EVM
- Workspace Template 'Uniswap v4 Periphery' renamed to 'Uniswap v4 Template'

Read More

Project Templates

MULTISIG Gnosis Safe | ERC20 0xProject | ERC20 OpenZeppelin

Featured Plugins

SOLIDITY ANALYZERS: Analyze your code using Remix, Solhint and Slither.

COOKBOOK: Find smart contracts, solidity libraries, and discover protocols.

SOLIDITY: Compile, test, and analyze smart contracts.

Scam Alert

The only URL Remix uses is remix.ethereum.org

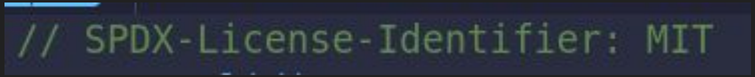
Beware of online videos promoting "liquidity front runner bots": Learn more

Additional safety tips: here

Remix

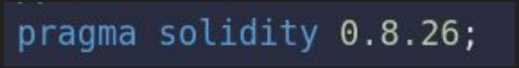
Then create a smart contract called *SimpleStorage.sol*

We need to begin our SC with the SPDX, a.k.a License



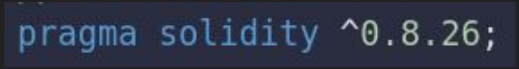
```
// SPDX-License-Identifier: MIT
```

Then we must indicate the pragama, a.k.a the version of Solidity



```
pragma solidity 0.8.26;
```

means we want to use the exact 0.8.26 version



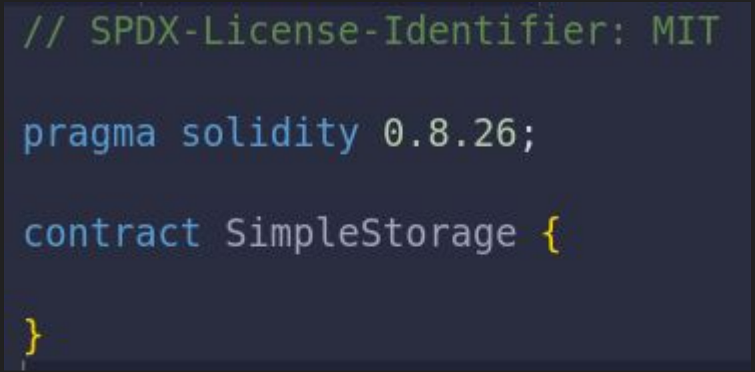
```
pragma solidity ^0.8.26;
```

means we want to use the 0.8.26 version or a superior version.
It’s recommended to indicate the exact version you want

Remix

We are working with the 0.8.19 version.
If we use the “compile” button on Remix, it choose automatically the right compiler for us

Declare a contract by using the *contract* keyword



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

pragma solidity 0.8.26;

contract SimpleStorage {

}
```

Solidity : Basic Types

Reminder : The Solidity doc can be visited here
https://docs.soliditylang.org

- Boolean

```
bool myBool = true;
```

- uint (= unsigned integer) equivalent to uint256 (encoded on 256 bits)

```
uint favoriteNumber = 11;
```

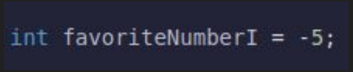
- uint8 (= unsigned integer encoded on 8 bits)

- string

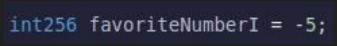
```
string favoriteNumberStr = '11';
```

Solidity : Basic Types

- int (=integer / encoded by default on 256 bits so equivalent to int256)

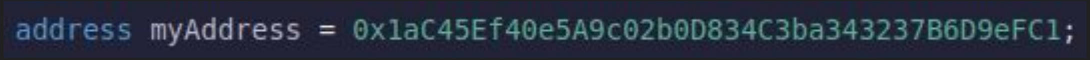


```
int favoriteNumberI = -5;
```



```
int256 favoriteNumberI = -5;
```

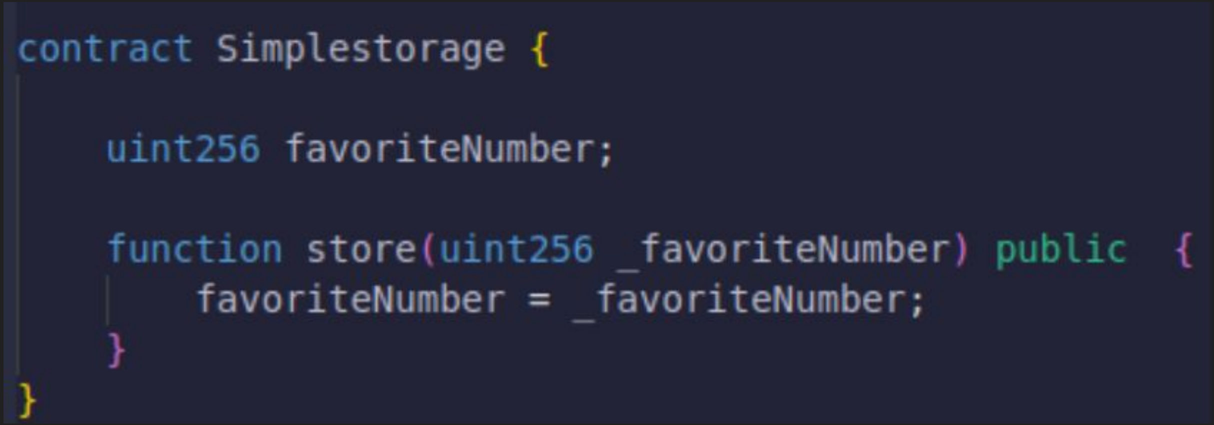
- address (represents and public address like yours in Metamask)



```
address myAddress = 0x1aC45Ef40e5A9c02b0D834C3ba343237B6D9eFC1;
```

Solidity : Functions

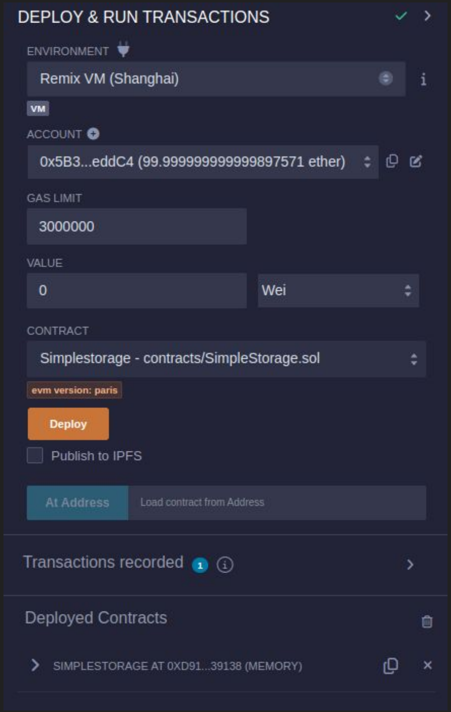
- keyword *function*
- name of the function
- arguments
- visibility of the function



```
contract Simplestorage {

    uint256 favoriteNumber;

    function store(uint256 _favoriteNumber) public  {
        favoriteNumber = _favoriteNumber;
    }
}
```

Solidity : Deploy with Remix

- Choose a fake blockchain environment (like Remix VM (xxx))
- Notice all the fake accounts are provided with some ETH
- Click *Deploy*
- Notice the address of the deployed SC



DEPLOY & RUN TRANSACTIONS

ENVIRONMENT
Remix VM (Shanghai)
VM

ACCOUNT
0x5B3...eddC4 (99.999999999999897571 ether)

GAS LIMIT
3000000

VALUE
0 Wei

CONTRACT
Simplestorage - contracts/SimpleStorage.sol
evm version: paris

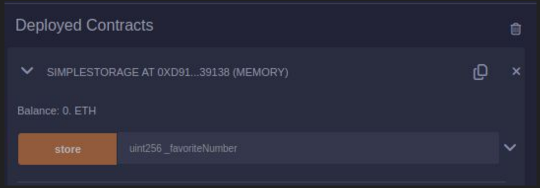
SIMPLESTORAGE AT 0XD91...39138 (MEMORY)

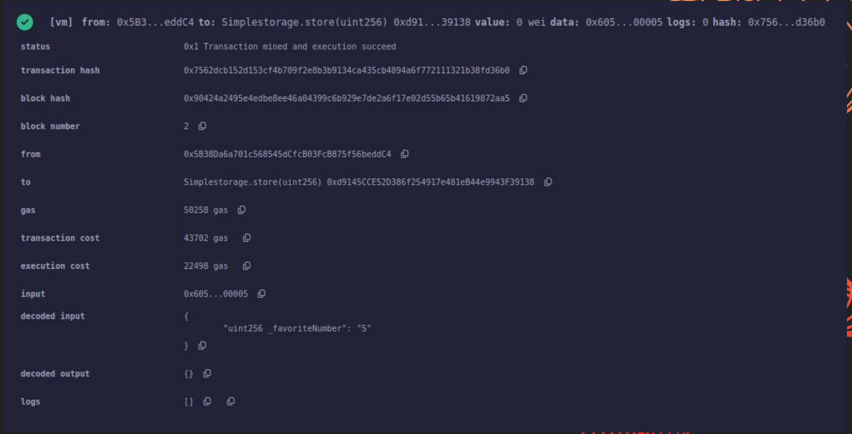
Solidity : Deploy with Remix

Remarks :

- To deploy the SC we must modify the BC
- So we need to pay Tx because modifying the BC has a cost

Then, we can see the function inside the SC in Remix.
We can put a value and click “store”.
We can see that we did a transaction to do that.
Notice that the amount of ETH are less than at the beginning (100 ETH)





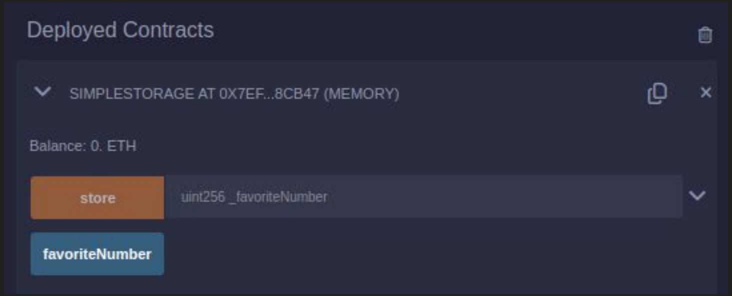
```
[vm] from: 0x5B3...eddC4 to: Simplestorage.store(uint256) 0xd91...39138 value: 0 wei data: 0x605...00005 logs: 0 hash: 0x756...d36b0
status               0x1 Transaction mined and execution succeed
transaction hash     0x7562dcb152d153cf4b709f2e8b3b9134ca435cb4894a6f772111321b38fd36b0
block hash           0x90424a2495e4edbe8ee46a04399c6b929e7de2a6f17e02d55b65b41619872aa5
block number         2
from                 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                   Simplestorage.store(uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                  50258 gas
transaction cost     43702 gas
execution cost       22498 gas
input                0x605...00005
decoded input        {
                             "uint256 _favoriteNumber": "5"
                     }
decoded output       {}
logs                 []
```

Solidity : Deploy with Remix

We modify the value of the state variable but we can not see its value.
How do we see it ?
It's because the state variable *favoriteNumber* has no visibility so it's by default internal visibility.
To see the value of this state variable.
We must change the visibility to *public*.
Go to https://docs.soliditylang.org/en/latest/contracts.html#visibility-and-getters
to see more on visibility

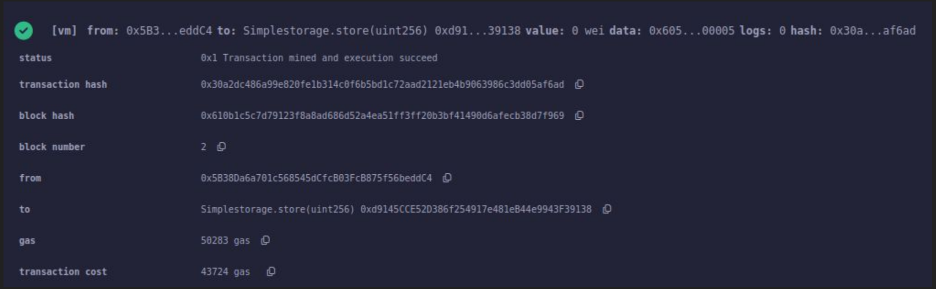
```
uint256 public favoriteNumber;
```

Then we must delete the deployed SC, Re-compile it, and redeploy.
Now we can see the state variable



Solidity : Deploy with Remix

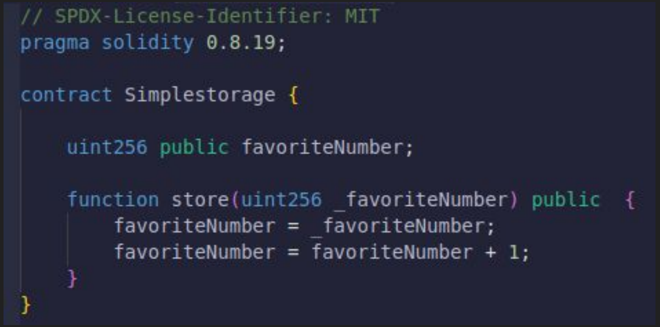
Notice the amount of gas you paid to store the number



```
[vm] from: 0x5B3...eddC4 to: Simplestorage.store(uint256) 0xd91...39138 value: 0 wei data: 0x605...00005 logs: 0 hash: 0x30a...af6ad
status               0x1 Transaction mined and execution succeed
transaction hash     0x30a2dc486a99e820fe1b314c0f6b5bd1c72aad2121eb4b9063986c3dd05af6ad
block hash           0x610b1c5c7d79123f8a8ad686d52a4ea51ff3ff20b3bf41490d6afecb38d7f969
block number         2
from                 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                   Simplestorage.store(uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                  50283 gas
transaction cost     43724 gas
```

The cost is higher than a simple Tx which is 21 000 gas to do it.

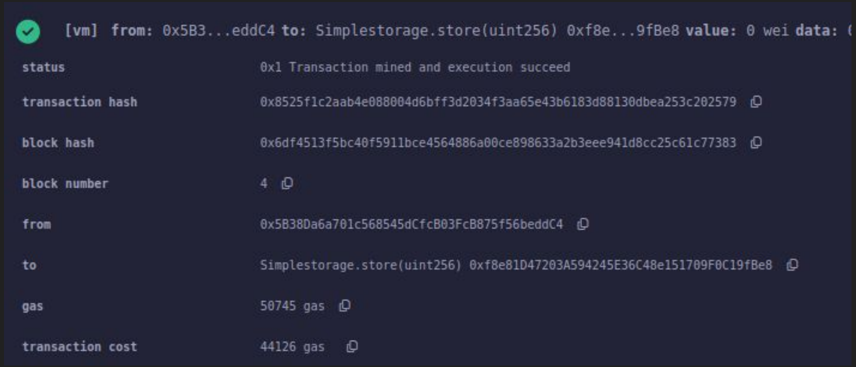
Now, just add some lines in the function



```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

contract Simplestorage {

    uint256 public favoriteNumber;

    function store(uint256 _favoriteNumber) public  {
        favoriteNumber = _favoriteNumber;
        favoriteNumber = favoriteNumber + 1;
    }
}
```

Solidity : Deploy with Remix

Notice the amount of gas now.



```
[vm]  from: 0x5B3...eddC4 to: Simplestorage.store(uint256) 0xf8e...9fBe8 value: 0 wei data: (
status              0x1 Transaction mined and execution succeed
transaction hash    0x8525f1c2aab4e088004d6bff3d2034f3aa65e43b6183d88130dbea253c202579
block hash          0x6df4513f5bc40f5911bce4564886a00ce898633a2b3eee941d8cc25c61c77383
block number        4
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  Simplestorage.store(uint256) 0xf8e81D47203A594245E36C48e151709F0C19fBe8
gas                 50745 gas
transaction cost    44126 gas
```

It's higher again.

So there's a correlation between the code and the transaction you pay

Solidity : Scope Variables

The scope of a variable is only inside the block where the variable is created.

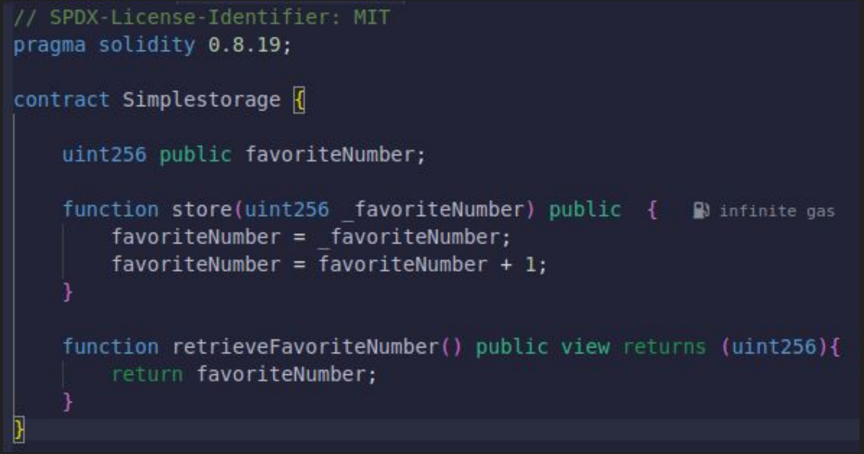
A variable created in a function will be only visible in a function.

A variable created outside a function in the smart contract will be visible everywhere?

It's called a state variable

Solidity : View functions

Create a new function that retrieve the state variable *favoriteNumber*



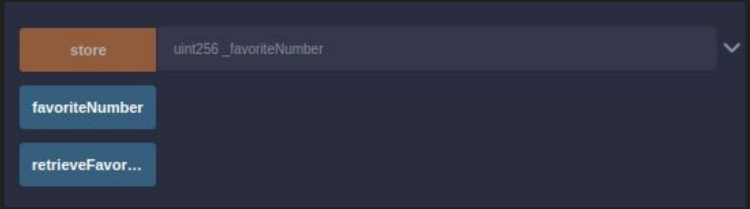
```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

contract Simplestorage {

    uint256 public favoriteNumber;

    function store(uint256 _favoriteNumber) public  {    infinite gas
        favoriteNumber = _favoriteNumber;
        favoriteNumber = favoriteNumber + 1;
    }

    function retrieveFavoriteNumber() public view returns (uint256){
        return favoriteNumber;
    }
}
```

A function marked as *view* indicates that it can retrieve/read state variables.

We indicate as well the returned type by keyword *returns*

Solidity : color functions in Remix

If you deploy the previous SC, you can notice that functions have different colors

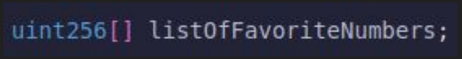


Functions marked with orange color can update the state of the Blockchain.

Functions marked with blue color can not update the state of the Blockchain.

Solidity : Array & Struct

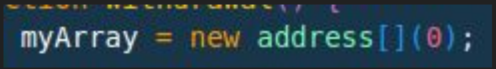
ARRAY :
We are going to create an array of *favoriteNumbers* by using an array



```
uint256[] listOfFavoriteNumbers;
```

Arrays in Solidity is the same than others languages. It's zero indexed (= the first index is 0).

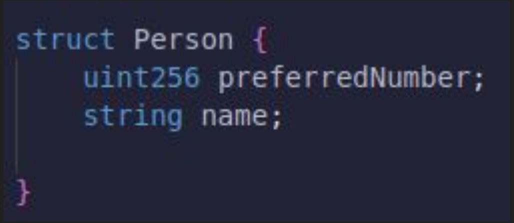
RESET ARRAYS :
Syntax : new <type>[](0)



```
myArray = new address[](0);
```

STRUCT :

It's like a *type* in *TypeScript*.
Ex :



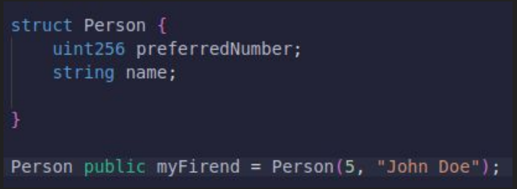
```
struct Person {
    uint256 preferredNumber;
    string name;

}
```

Solidity : Array & Struct

STRUCT :

Now we can use this struct/type and assign this type to a variable. This represents the metaData of a JSON object

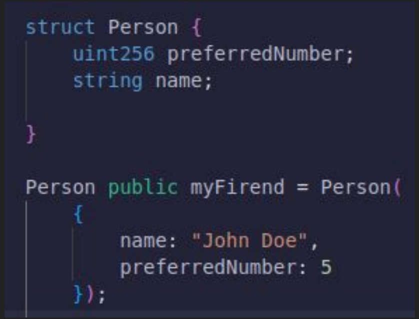
Ex :



```
struct Person {
    uint256 preferredNumber;
    string name;

}

Person public myFirend = Person(5, "John Doe");
```

With that writing we must respect the order struct variables appear

But we can write differently and not respect the natural order of struct variables



```
struct Person {
    uint256 preferredNumber;
    string name;

}

Person public myFirend = Person(
    {
        name: "John Doe",
        preferredNumber: 5
    });
```

Solidity : Array & Struct

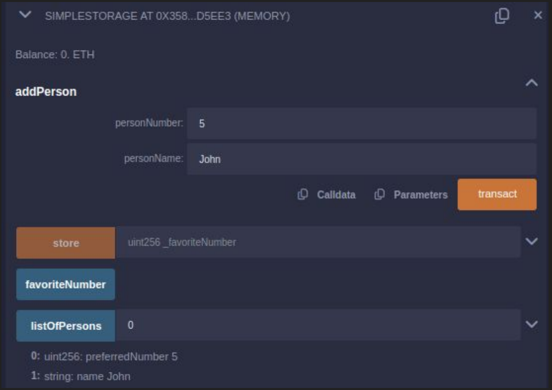
Struct & Arrays :

Now let's create a function that push a Person in the array of Persons

```
function addPerson(uint256 personNumber, string memory personName) public {
    listOfPersons.push(Person(personNumber, personName));
}
```

Deploy this new smart contract and play with *listOfPersons* variable by retrieving the index 0 before adding a Person, then add a Person and try to retrieve the index 0

etc


SIMPLESTORAGE AT 0X358...D5EE3 (MEMORY)

Balance: 0. ETH

addPerson

personNumber: 5

personName: John

listOfPersons: 0

```
0: uint256: preferredNumber 5
1: string: name John
```

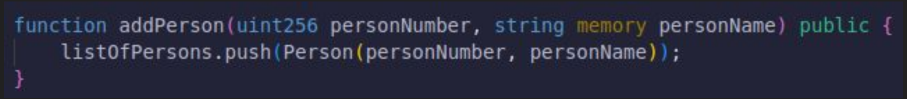
Solidity : Memory & storage

There are 6 places we can store data in Solidty :

- Storage
- calldata
- memory
- code
- logs
- stack

We'll only focus on *calldata*, *memory* & *storage*.

Notice the keyword memory we use to pass the string argument previously



```
function addPerson(uint256 personNumber, string memory personName) public {
    listOfPersons.push(Person(personNumber, personName));
}
```

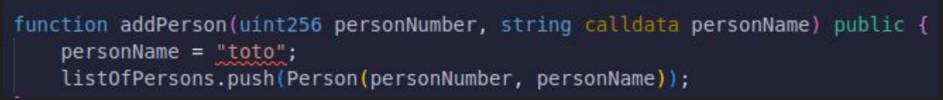
memory and *calldata* are keyword to indicate that the variable is only accessible for the duration of the function call. The life of the variable is only the function call.

Solidity : Memory & storage

Difference between *calldata* and *memory* :

Calldata :

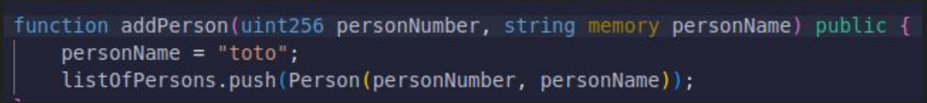
We can not modify the variable. So the variable is *immutable*



```
function addPerson(uint256 personNumber, string calldata personName) public {
    personName = "toto";
    listOfPersons.push(Person(personNumber, personName));
```

Memory :

We can modify the variable. So the variable is *mutable*



```
function addPerson(uint256 personNumber, string memory personName) public {
    personName = "toto";
    listOfPersons.push(Person(personNumber, personName));
```

We must specify the keyword for *string* types or *arrays* not primitive types like *uint*.

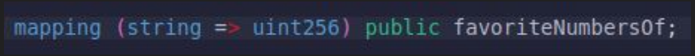
string type is considered like an array because it's an array of bytes

Solidity : Mappings

Arrays is a data structure not adapted if we want for instance retrieve the favorite number of a Person.

To implement this we need a more advanced data structure like a *mapping* which is a set of *keys & values*

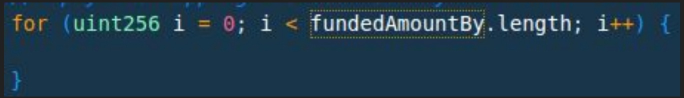
Declaration :



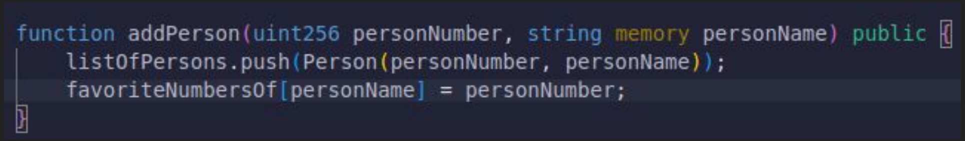
```
mapping (string => uint256) public favoriteNumbersOf;
```

Call :
The call is similar to an array call.
We can't loop through the mapping like an array with a for ... loop

Ex :



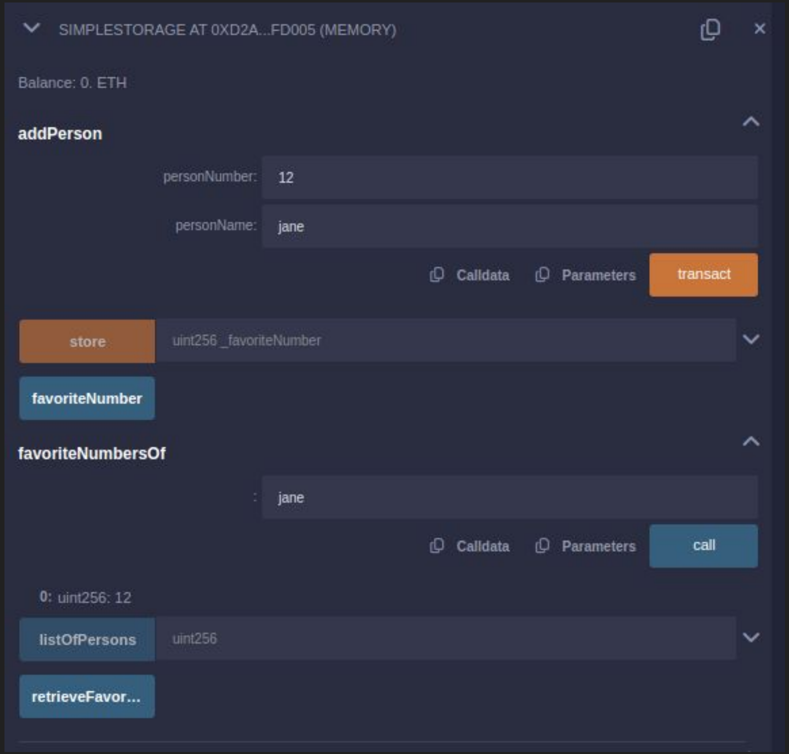
```
for (uint256 i = 0; i < fundedAmountBy.length; i++) {

}
```



```
function addPerson(uint256 personNumber, string memory personName) public {
    listOfPersons.push(Person(personNumber, personName));
    favoriteNumbersOf[personName] = personNumber;
}
```

Solidity : Mappings

Let’s deploy the new smart contract and play with it on Remix



SIMPLESTORAGE AT 0XD2A...FD005 (MEMORY)

Balance: 0. ETH

addPerson

personNumber: 12

personName: jane

favoriteNumbersOf

: jane

0: uint256: 12

Solidity : Deploy on a real testnet

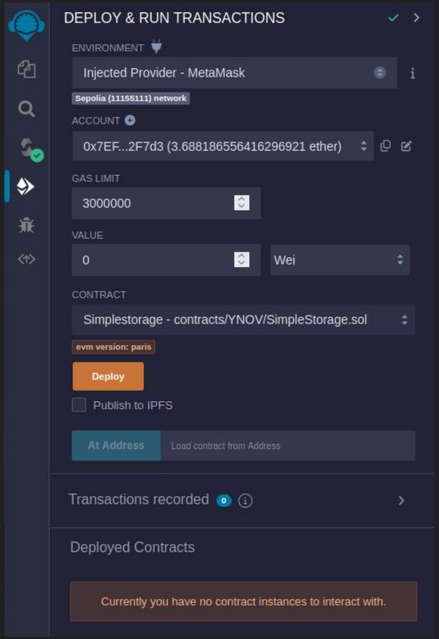
To do this, you will need a Metamask account with sepoliaETH.

Use a faucet to get some sepoliaETH.

So, prerequisites are :

- Metamask installed
- Wallet created
- Sepolia Network setup
- some seopliaETH present in the wallet

Then, in Remix you must choose the right environment in the *Deploy* Tab



DEPLOY & RUN TRANSACTIONS

ENVIRONMENT
Injected Provider - MetaMask
Sepolia (11155111) network

ACCOUNT
0x7EF...2F7d3 (3.688186556416296921 ether)

GAS LIMIT
3000000

VALUE
0 Wei

CONTRACT
Simplestorage - contracts/YNOV/SimpleStorage.sol
evm version: paris

Deploy
Publish to IPFS

At Address
Load contract from Address

Transactions recorded 0

Deployed Contracts

Currently you have no contract instances to interact with.

Solidity : Deploy on a real testnet

To deploy, the Metamask plugin pops up to initiate a transaction.

Click confirm.

And see the result in the block explorer for Sepolia Network.

Notice the big amount of *Input Data* that was sent with this transaction

Now the smart contract has an address on the blockchain.

Copy/paste the address on the blockchain explorer.

You can see :

- the transaction hash
- the *To* field that is marked as *Contract creation*

Solidity : Deploy on a real testnet

To learn this interaction between your smart contract and a real testnet,
let’s play with the functions of our smart contract deployed on Sepolia Network

Solidity : require

One of the most used *keyword* in the *solidity* language.

It's a flow control statement that must. It can be placed at the beginning of the function or elsewhere.

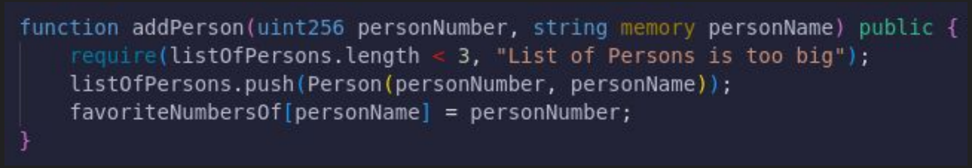
It contains :

- a condition
- a revert message (= the tx reverts with an error message)

If the required condition is not filled, the tx revert with the message

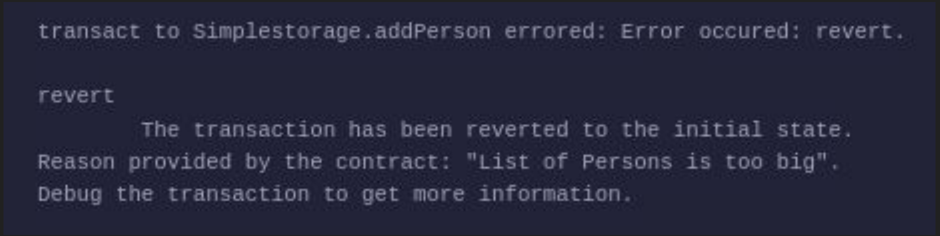
Solidity : require

Example :

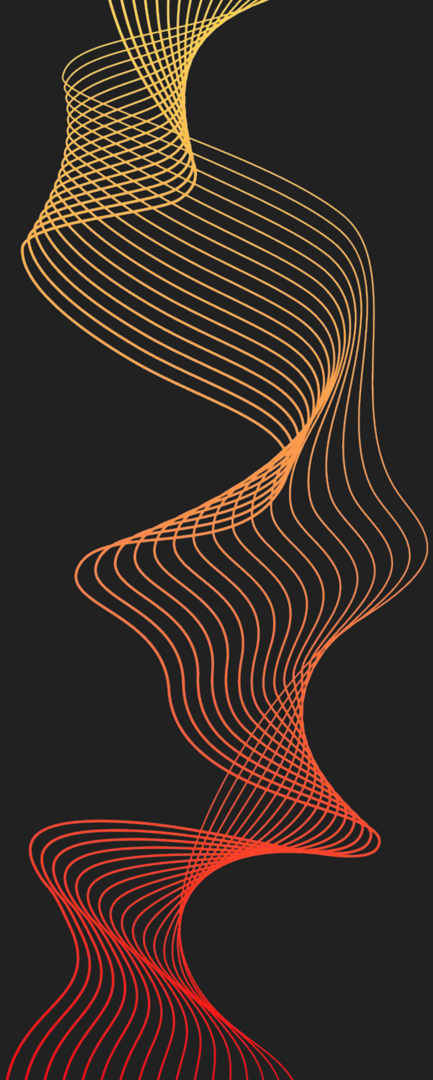


```
function addPerson(uint256 personNumber, string memory personName) public {
    require(listOfPersons.length < 3, "List of Persons is too big");
    listOfPersons.push(Person(personNumber, personName));
    favoriteNumbersOf[personName] = personNumber;
}
```

If I try to add more than 3 persons, my transaction fails (= reverts) and launch a message



```
transact to Simplestorage.addPerson errored: Error occured: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "List of Persons is too big".
Debug the transaction to get more information.
```



Solidity : error

Errors are the new way to revert Tx.

You must declare the error

```
error AddPersonError();
```

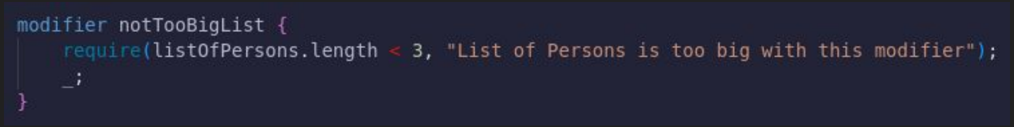
Then you must revert it in the function

```
function addPerson_withError(uint256 _number, string memory _name) public {
    if (listOfPersons.length >= 3) revert AddPersonError();
    listOfPersons.push(Person(_number, _name));
    favoriteNumberOf[_name] = _number;
}
```

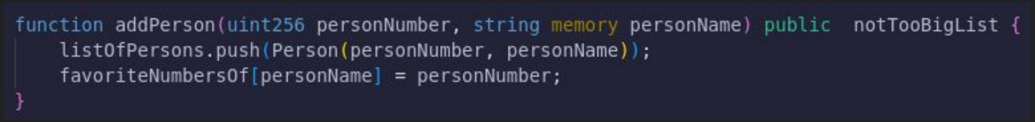
Solidity recommends now to use “errors” instead of “require”

Solidity : modifiers

A function can have multiple *require*. In order to make it more readable, we can factorize the function by embedding the require in a *modifier.*
Example :



```
modifier notTooBigList {
    require(listOfPersons.length < 3, "List of Persons is too big with this modifier");
    _;
}
```



```
function addPerson(uint256 personNumber, string memory personName) public  notTooBigList {
    listOfPersons.push(Person(personNumber, personName));
    favoriteNumbersOf[personName] = personNumber;
}
```

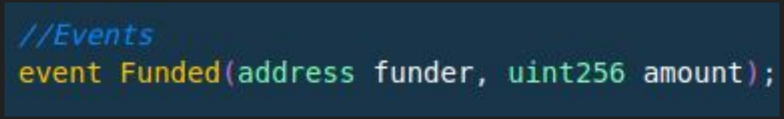
This code produces the same result than previously with the *require* inside the function

The "_;" indicates that if the require is OK, the flox of the function can continue.
So it indicates that it's like to place the require at the beginning of the function.
The "_;" can be placed at the beginning of the modifier that indicates that the require behaves like it would be placed at the end of the function but <u>it is not a good practice</u>

Solidity : events

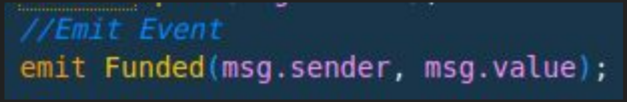
Events in Solidity are used to facilitate communication between smart contracts and their user interfaces.
They are emitted by smart contracts when certain actions are performed, allowing applications to react to these actions

Declaration :
With keyword *event*



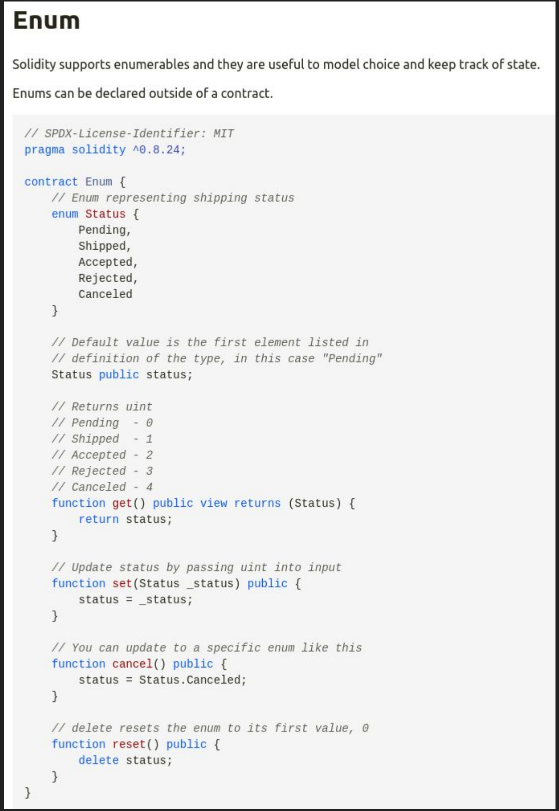
```
//Events
event Funded(address funder, uint256 amount);
```

Call :
With keyword *emit*



```
//Emit Event
emit Funded(msg.sender, msg.value);
```

Solidity : enum



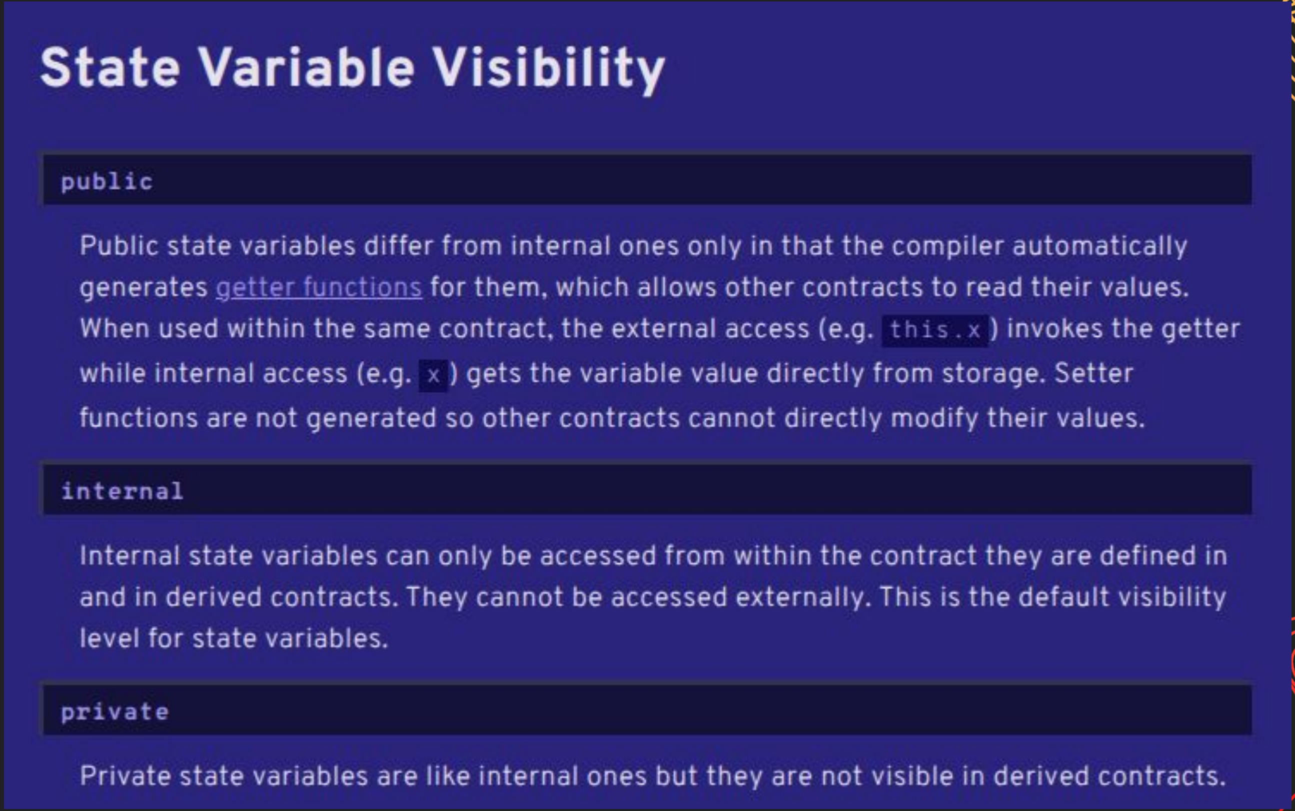
Enum

Solidity supports enumerables and they are useful to model choice and keep track of state.

Enums can be declared outside of a contract.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract Enum {
    // Enum representing shipping status
    enum Status {
        Pending,
        Shipped,
        Accepted,
        Rejected,
        Canceled
    }

    // Default value is the first element listed in
    // definition of the type, in this case "Pending"
    Status public status;

    // Returns uint
    // Pending  - 0
    // Shipped  - 1
    // Accepted - 2
    // Rejected - 3
    // Canceled - 4
    function get() public view returns (Status) {
        return status;
    }

    // Update status by passing uint into input
    function set(Status _status) public {
        status = _status;
    }

    // You can update to a specific enum like this
    function cancel() public {
        status = Status.Canceled;
    }

    // delete resets the enum to its first value, 0
    function reset() public {
        delete status;
    }
}
```

Solidity : Visibility



State Variable Visibility

`public`

Public state variables differ from internal ones only in that the compiler automatically generates getter functions for them, which allows other contracts to read their values. When used within the same contract, the external access (e.g. `this.x`) invokes the getter while internal access (e.g. `x`) gets the variable value directly from storage. Setter functions are not generated so other contracts cannot directly modify their values.

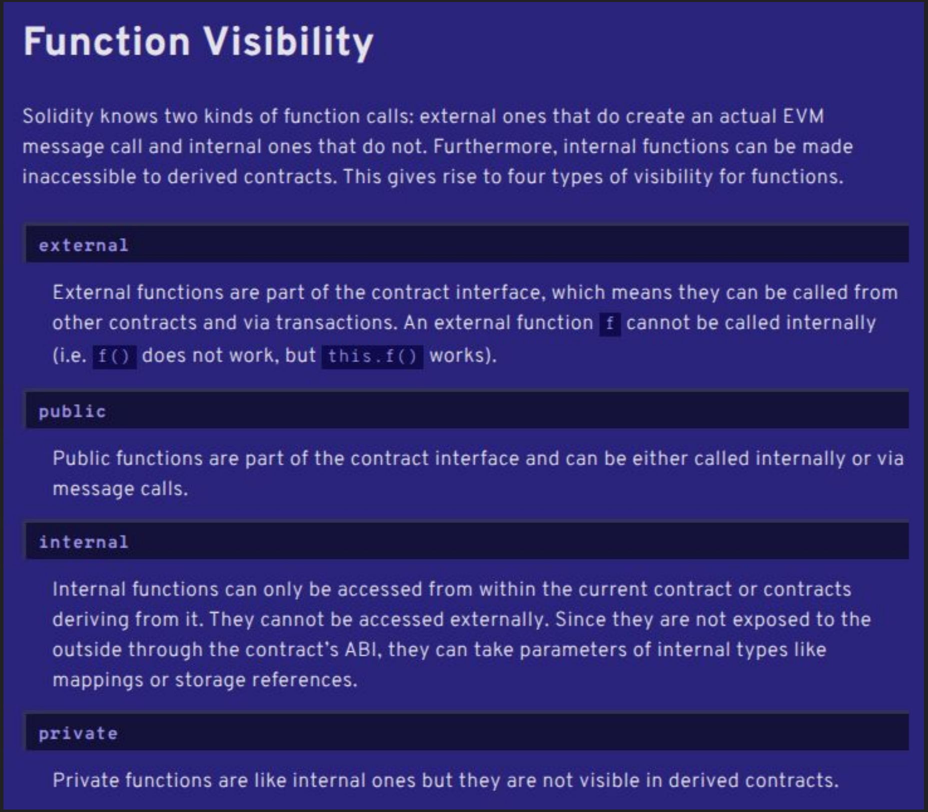
`internal`

Internal state variables can only be accessed from within the contract they are defined in and in derived contracts. They cannot be accessed externally. This is the default visibility level for state variables.

`private`

Private state variables are like internal ones but they are not visible in derived contracts.

Solidity : Visibility



Function Visibility

Solidity knows two kinds of function calls: external ones that do create an actual EVM message call and internal ones that do not. Furthermore, internal functions can be made inaccessible to derived contracts. This gives rise to four types of visibility for functions.

`external`

External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works).

`public`

Public functions are part of the contract interface and can be either called internally or via message calls.

`internal`

Internal functions can only be accessed from within the current contract or contracts deriving from it. They cannot be accessed externally. Since they are not exposed to the outside through the contract's ABI, they can take parameters of internal types like mappings or storage references.

`private`

Private functions are like internal ones but they are not visible in derived contracts.

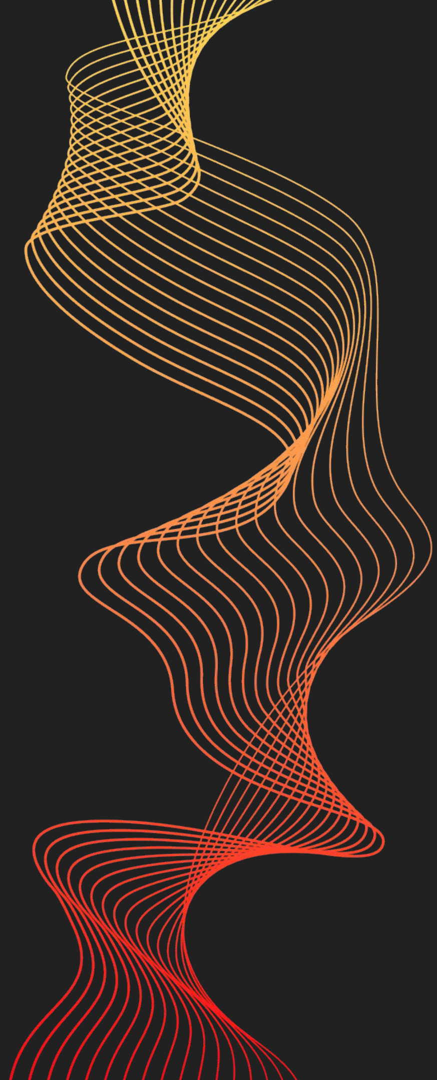
Solidity : global variables

There are special variables and functions which always exists in the global namespace and are mainly used to provide information about the blockchain :

- *block.number* : current block number
- *block.timestamp* : current block timestamp as seconds since unix epoch
- *msg.sender* : address of the sender of the message (in the tx)
- *msg.value* : amount of wei sent with the message
- *tx.origin* : address of the sender of the transaction (full call chain)

You can see others variables on the link

https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#special-variables-functions



Solidity : msg.sender vs tx.origin

tx.origin has always the same value in a flow of smart contracts.

It's not the case in msg.sender : the value changes depending of the caller

Example :

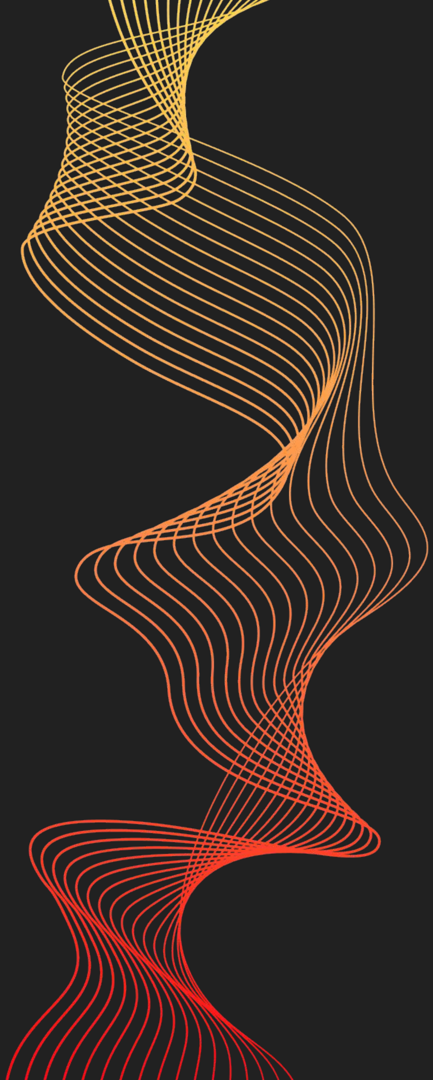
```
EOA -> Contract A -> Contract B -> Contract C
```

- For Contract A : tx.origin = address of EOA | msg.sender = address of EOA
- For Contract B : tx.origin = address of EOA | msg.sender = address of contract A
- For Contract C : tx.origin = address of EOA | msg.sender = address of contract B

Solidity : global variables

msg.sender is the “sender” of the Tx as it’s said

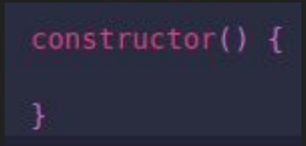
When you “deploy” a smart contract on the blockchain, you must sign a transaction.
So the public key that deploys the SC on the BC is the deployer
and msg.sender will take the value of the public key of the deployer



Solidity : Constructor

Like many others POO Language, a smart contract can contain a *CONSTRUCTOR*.

This constructor is the first function that is called when a SC is deployed



```
constructor() {

}
```

Solidity : TP #1

You must implement a voting system.

This system must :

- allow the registration of candidates only for the deployer/administrator of the SC
- enable accounts to vote for these candidates
- keep track of the votes received for each candidate
- contain at least 2 events

Requirements :

The smart contract must contain :

- A *struct* called *Candidate* with properties :
    - id (identifier of the candidate of type uint)
    - name (name of the candidate of type string)
    - voteCount (count of votes the candidate received)
- State variables
    - a *mapping* to store candidates with key *id* of the candidate and value the struct *Candidate*
    - an array of candidate ids to facilitate iteration
    - a mapping of voters with key = address of the voter and value = a boolean that indicates if a voter has voted
    - the owner of the smart contract (= that deploy the contract, so it must be assigned in the constructor of the SC)