Nadia Fabrizio

Blockchain Laboratory day 2 and 3

Fintech 2025

SOLIDITY: QUICK INTRO

All material of this lession is taken from REMIX OFFICIAL TUTORIAL AND THE FOLLOWING LINKS

BEGINNER COURSE:

https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.25+commit.b61c2a91.js

DOCUMENTATION

https://remix-ide.readthedocs.io/en/latest/

BASIC SYNTAX

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
pragma solidity ^0.8.3;

contract HelloWorld {
    string public greet = "Hello World!";
}
```

Lab Remix

https://remix-ide.readthedocs.io/en/latest/

BASIC INTRO: STATICALLY TYPED LANGUAGE

- Using the **pragma keyword (line 3),** we specify the Solidity version we want the compiler to use. In this case, it should be greater than or equal to 0.8.3 but less than 0.9.0.
- We define a contract with the **keyword contrac**t and give it a name, in this case, HelloWorld (line 5).
- Inside our contract, we define **a state variable** greet that holds the string "Hello World!" (line 6).
- **Solidity is a statically typed language,** which means that you need to specify the type of the variable when you declare it. In this case, greet is a string.

BASIC INTRO: VISIBILITY

The visibility specifier is used to control who has access to functions and state variables.

There are **four types of visibilities:** external, public, internal, and private.

1. **PUBLIC**= you can access from inside and outside the contract, from child contracts, or transactions, and from other contracts
2. **PRIVATE**=Can be called from inside the contract
3. **INTERNAL**=Can be called from inside the contract
4. **EXTERNAL**=Can be called from other contracts or transactions
   - State variables can not be external

DATA TYPES

bool

- Booleans can either have the value true or false.

uint

- keywords **uint and uint8 to uint256 to declare an unsigned integer type** (they don’t have a sign, unlike -12, for example!).
- Uints are integers that are positive or zero and range from 8 bits to 256 bits. The type **uint is the same as uint256.**

int

- keywords **int and int8 to int256 to declare an integer type**.
- **Integers can be positive, negative, or zero** and range from 8 bits to 256 bits. The type int is the same as int256.

DATA STRUCT/custom data types=collection of variables of mixed types

- There are different ways to initialize a struct.
  - **Positional parameters:** We can provide the name of the struct and the values of its members as parameters in parentheses
  - Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 19).
  - Initialize and update a struct: We initialize an empty struct first and then update its member by assigning it a new value (line 23).
- **Accessing structs**
  - To access a member of a struct we can use the dot operator (line 33).
- A particular case is **ENUM I**n Solidity enums are **custom data types** consisting of a limited set of constant values.

DATA LOCATION: STORAGE, MEMORY, and CALLDATA

1. **Stored** permanently on the blockchain ⇒expensive to use.
    - **State variables are always stored in storage.**
2. **Memory**
    - stored temporarily ⇒not on the blockchain.
    - They only exist during the execution of an external function and are discarded afterward. **They are cheaper** to use than values stored in storage.
3. **Calldata**
    - stores function arguments
    - stored temporarily during the execution of an external function
    - values stored in calldata can not be changed. **Calldata is the cheapest data location to use.**

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

pragma solidity ^0.8.3;

 contract Primitives {

    bool public boo = true;

    uint8 public u8 = 1;

    uint public u256 = 456;

    uint public u = 123; // uint is an alias for uint256

     int8 public i8 = -1;

    int public i256 = 456;

    int public i = -123; // int is same as int256
    address public addr =0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

     bool public defaultBoo; // false

    uint public defaultUint; // 0

    int public defaultInt; // 0

    address public defaultAddr; // 0x0000000000000000000000000000000000000000

}
```

VARIABLES: STATE, LOCAL and GLOBAL

- **State Variables**
  - stored in the contract storage ⇒on the blockchain.
  - **declared inside the contract** but outside the function.
- **Local Variables**
  - **stored in the memory**
  - their values are accessible within the function they are defined in but not externally ⇒ not stored on the blockchain.
- **Global Variables (** also said **Special0**
  - exist in the global namespace.
  - don't need to be declared
  - can be accessed from within your contract.
  - used to retrieve information about the blockchain⇒ addresses, contracts, and transactions.

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

pragma solidity ^0.8.3;

contract Variables {

    // State variables are stored on the blockchain.

    string public text = "Hello";

    uint public num = 123;

    function doSomething() public {

        // Local variables are not saved to the blockchain.

        uint i = 456;

        // Here are some global variables

        uint timestamp = block.timestamp; // Current block timestamp

        address sender = msg.sender; // address of the caller

    }

}
```

Functions - Reading and Writing to a State Variable

Solidity functions types are two:

A) Functions that **modify the state of the blockchain,** like writing to a state variable. I

B) **Functions that don't modify the state of the blockchain**.

  a) marked =view or pure.

  b) View= do not modify the state (see next slide)

If the function takes inputs , you must specify the parameter types and names.

Note:get function also returns values, so we have to specify the return types.

A common convention is to use an underscore as a prefix for the parameter name to distinguish them from state variables.

Functions that modify the state

- Writing to state variables.
- Emitting events.
- Creating other contracts.
- Using selfdestruct.
- Sending Ether via calls.
- Calling any function not marked view or pure.
- Using low-level calls.
- Using inline assembly that contains certain opcodes."

can be a question in exam

examples of functions that modify the state

Functions MODIFIER & CONSTRUCTOR

MODIFIER

- **used to change the behavior of a function**.
  - often check for a condition prior to executing a function to restrict access or validate inputs.
- The function **changeOwner** can change this ownership⇒It takes an input parameter of the type address and assigns its value to the state variable owner.

CONSTRUCTOR

- executed upon the creation of a contract. The constructor can have parameters and is useful when you don't know certain initialization values before the deployment of the contract.

ASSIGNMENT n 1

1. Create a public state variable called b that is of type bool and initialize it to true.
2. Create a public function called get_b that returns the value of b.

LOOPS (not necessary to understand everything, just to give an idea)

Three types of **loops**: **for, while, and do while loops**.

for

-you should specify tn of iterations to avoid running out of gas

while

to break the loop based on a condition

Loops are seldom used in Solidity since transactions might run out of gas

do while

The do while loop is a special kind of while loop where you can ensure the code is executed at least once, before checking on the condition.

continue

The continue statement is used to skip the remaining code block

ERC20 functions

- **totalSupply() -** Returns the total units of this token that currently exist
- **balanceOf(**address) - Returns the token balance of an address - transfer(address, amount)
- **Transfers** amount of tokens to address, from the balance of the address that executed the transaction
- transferFrom(sender, recipient, amount) - Transfers token from sender to recipient - Used in combination with approve
- **approve**(recipient, amount) - Authorizes recipient to execute several transfers up to amount, from the address that executed the transaction
- **allowance(**owner, spender) - Returns the remaining amount that the spender is approved to withdraw from the owner
- **Transfer event** - Triggered upon successful transfer (call to transfer or transferFrom), even for 0 value transfers
- **Approval event** - Logged upon successful call to approve

ERC20 functions

- **name()** - Returns a human-readable name for the token (i.e. “Ether”)
- **symbol()** - Returns a human-readable symbol for the token (i.e. “ETH”)
- **decimals()** - Returns the number of decimals used to divide token amounts - i.e. if decimals == 2, then the token is divided by 100 to get its user representation

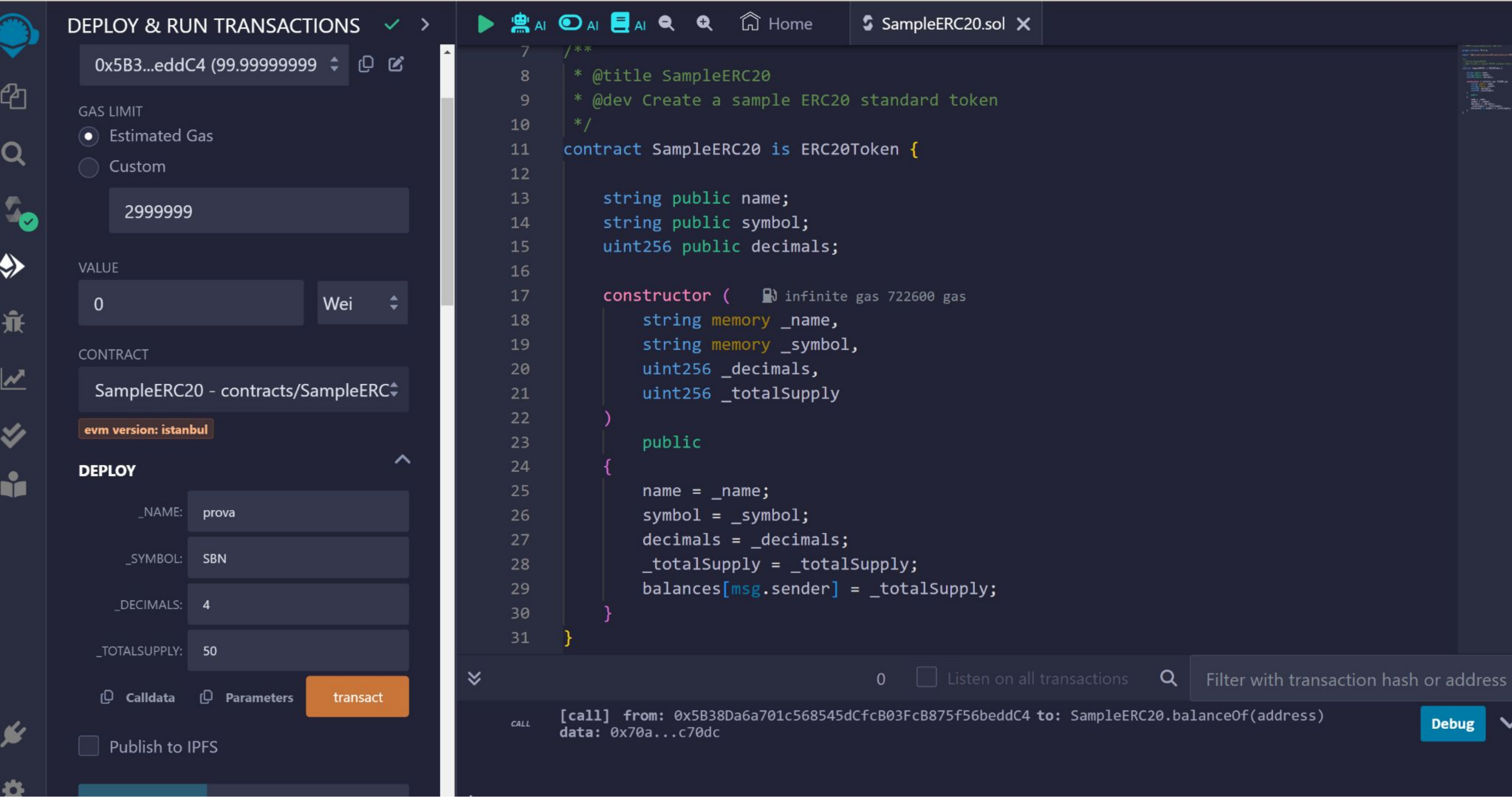
ERC20 structures

All ERC20 contracts contain 2 data structures:

- **balances (**owner_address => balance_amount) - Allows the token contract to keep track of who owns the tokens - Each transfer is a deduction from one balance and an addition to another balance

- **allowances** (owner_address => (spender_address => amount_allowed)) - In ERC20 tokens, an owner can delegate authority to a spender to spend a specific amount from their balance

Let’s create and run our first ERC20



0x5B3...eddC4 (99.99999999

GAS LIMIT

● Estimated Gas

○ Custom

2999999

VALUE

0 Wei

CONTRACT

SampleERC20 - contracts/SampleERC

evm version: istanbul

DEPLOY

_NAME: prova

_SYMBOL: SBN

_DECIMALS: 4

_TOTALSUPPLY: 50

□ Publish to IPFS

```
/**
 * @title SampleERC20
 * @dev Create a sample ERC20 standard token
 */
contract SampleERC20 is ERC20Token {

    string public name;
    string public symbol;
    uint256 public decimals;

    constructor (    infinite gas 722600 gas
        string memory _name,
        string memory _symbol,
        uint256 _decimals,
        uint256 _totalSupply
    )
        public
    {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        _totalSupply = _totalSupply;
        balances[msg.sender] = _totalSupply;
    }
}
```

0 □ Listen on all transactions

Filter with transaction hash or address

```
CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SampleERC20.balanceOf(address)
      data: 0x70a...c70dc
```

Let’s do make the Hello Coin!

```
pragma solidity ^0.4.18;
contract HelloCoin {
string public name = 'HelloCoin';
//currency name. Please feel free to change it
string public symbol = 'coin_nadia';
//choose a currency symbol. Please feel free to change it
mapping (address => uint) balances;
//a key-value pair to store addresses and their account balances
event Transfer(address _from, address _to, uint256 _value);
```

Hello Coin cont.

```
// declaration of an event. Event will not do anything but add a
record to the log
constructor() public {
//when the contract is created, the constructor will be called
automatically
balances[msg.sender] = 10000;
//set the balances of creator account to be 10000. Please feel free to
change it to any number you want.
}
function sendCoin(address _receiver, uint _amount) public
returns(bool sufficient) {
if (balances[msg.sender] < _amount) return false;
```

Hello coin cont.

```
// validate transfer
balances[msg.sender] -= _amount;
balances[_receiver] += _amount;
emit Transfer(msg.sender, _receiver, _amount);
// complete coin transfer an
return true;
}
function getBalance(address _addr) public view returns(uint) {
//balance check
return balances[_addr];
}
```

Hello Coin 2nd version

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping(address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }
```

Hello Coin 2nd version cont.

```
    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```