BLOCKCHAIN: SMART CONTRACTS
LECTURE 3 - **SMART CONTRACTS. SOLIDITY**

FLORIN CRACIUN

IMPORTANT

Some of the following slides are the property of

Dr. Emanuel Onica & Dr. Andrei Arusoaie

Faculty of Computer Science,

Alexandru Ioan Cuza University of Iași

and are used with their consent.

CONTENTS

1. Smart contracts
2. Solidity basics
   a. Basic syntax
   b. Basic types (uint, string, array)
   c. Functions
   d. Structs
   e. delete and revert
   f. Remix (compile, deploy, transactions, gas)
3. Our first app in Solidity

SMART CONTRACTS - INTRODUCTION

- **Smart contract** = program deployed and executed within an Ethereum virtual environment
- Typically **used to transfer digital assets** between accounts
- Ethereum **smart contracts** are **similar** with **classes** in object-oriented programming (but there are differences)
- Think of smart contracts as programs consisting of functions which can be called by various addresses

SOLIDITY - INTRO

- **Solidity** = the main programming language for writing smart contracts for Ethereum
- *Contract-oriented language*
- It runs on the Ethereum Virtual Machine (EVM)
- Statically typed
- Inheritance, Libraries, etc.

DOC: https://docs.soliditylang.org/en/v0.8.2/

IDE

- Remix: https://remix.ethereum.org/
- Others:
  - EthFiddle
  - JetBrains IDE
  - Eclipse + YAKINDU
  - Etheratom
  - Visual Studio Code + Solidity Extension
  - …
- Compiler: `solc`

SOLIDITY - A TASTE

- *Contract*-oriented language
  - Different from Object-oriented
  - In a way, smart contracts resemble microservices
  - Contracts are executed by miners
  - Contract execution costs: *gas*
  - The code is visible to the public
  - Anyone in the network can call functions on the smart contract

REMIX

- https://remix.ethereum.org/
- Pragmas
- Demo (empty) contract
- Compilation
- Deployment

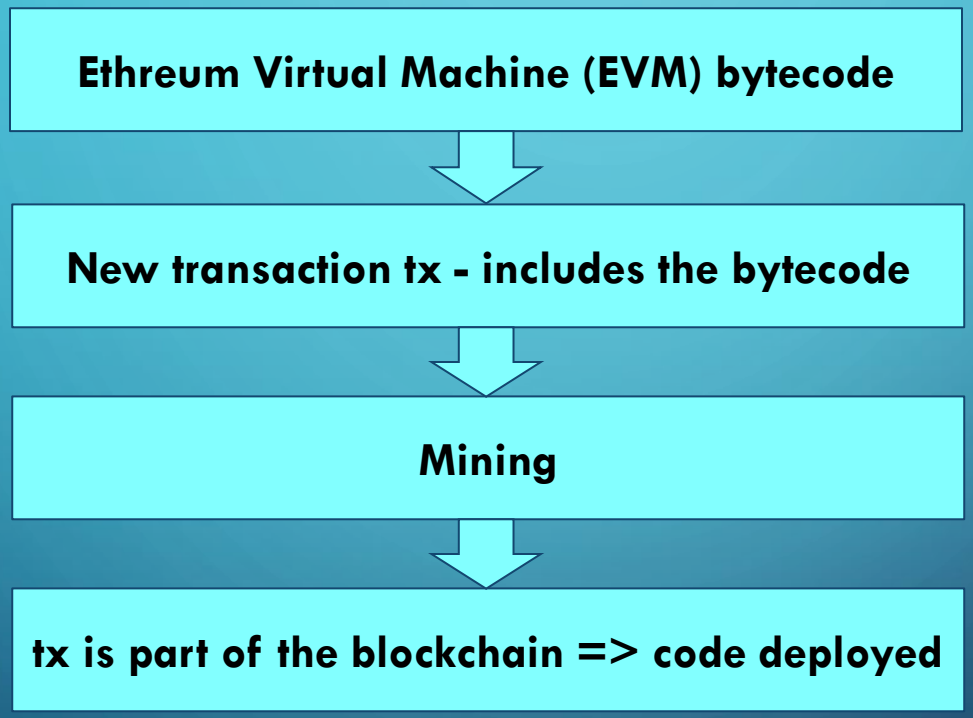
COMPILATION OVERVIEW

```
pragma solidity ^0.5.11;
contract MyContract { }
```

Solidity compiler

Ethreum Virtual Machine (EVM) bytecode

DEPLOYMENT OVERVIEW



A SIMPLE STORAGE CONTRACT - DEMO

MyFirstSmartContract.sol

```
1  pragma solidity ^0.5.11;
2
3  contract MyContract {
4      string public data;
5
6      function set(string memory _data)
7          public
8      {
9          data = _data;
10     }
11
12     function get()
13         public
14         view
15         returns(string memory)
16     {
17         return data;
18     }
19 }
20
```

Solidity compiler version

Contracts

Storage variable

Functions with arguments;
memory do not keep this in the storage

Access modifiers: public and view
view - function does not modify data

REMIX

1. Some syntax, visibility modifiers
2. Compilation
3. Function calls
4. Transaction receipts, addresses, code
5. OPCODES: https://etherscan.io/opcode-tool
6. Gas

PURE VS. VIEW

MyFirstSmartContract.sol

```
1   pragma solidity ^0.5.11;
2
3   contract MyContract {
4       string public data;
5
6       function set(string memory _data)
7           public
8       {
9           data = _data;
10      }
11
12      function get()
13          public
14          pure
15          returns(string memory)
16      {
17          return data;
18      }
19  }
20
```

browser/MyFirstSmartContract.sol:17:16: TypeError: Function declared as pure, but this expression (potentially) reads from the environment or state and thus requires "view".
return data;
^--^

THIS, PRIVATE

```
browser/MyFirstSmartContra
ct.sol:9:9: TypeError:
Member "data" not found or
not visible after
argument-dependent lookup
in contract MyContract.
this.data = _data;
^-------^
```

MyFirstSmartContract.sol

```
1  pragma solidity ^0.5.11;
2
3  contract MyContract {
4      string private data;
5
6      function set(string memory _data)
7          public
8      {
9          this.data = _data;
10     }
11
12     function get()
13         public
14         view
15         returns(string memory)
16     {
17         return data;
18     }
19 }
20
```

Hm… But this works in OOP, right?

THIS, EXTERNAL

this = the current contract, explicitly convertible to Address!

```
    function erase()
        external
    {
        data = "";
    }

    function test_erase()
        public
    {
        this.erase();
    }
```

this works here because erase is declared external

THIS, INTERNAL

internal = resembles protected; not visible from outside, only in the derived contracts

```
browser/MyFirstSmartCo
ntract.sol:29:9:
TypeError: Member
"erase" not found or
not visible after
argument-dependent
lookup in contract
MyContract.
this.erase();
^--------^
```

```
19
20      function erase()
21          internal
22      {
23          data = "";
24      }
25
26      function test_erase()
27          public
28      {
29          this.erase();
30      }
31
```

WRAPPING UP

- Visibility levels:
  - `external` - can be called from another contract via tx or `this`
  - `internal` - can be called only internally (without `this`)
  - `public` - visible anywhere
  - `private` - visible only inside the contract
  - `storage` - variables can be change only via tx
  - `memory` - variables exists only inside the calling function
  - `view` - the function cannot modify the storage
  - `pure` - the function does not even read the storage data
- **https://solidity.readthedocs.io/en/v0.5.11/contracts.html#visibility-and-getters**

ARRAYS

A simple contract where an array is stored on the blockchain with basic operations:

1. append
2. get
3. getAll
4. clear
5. size

Explore gas cost for each operation.

ArrayContract.sol

```
pragma solidity >0.5.10;

contract ArrayContract {
    uint[] numbers;

    function push(uint num)
        external
    {
        numbers.push(num);
    }

    function get(uint index)
        public
        view
        returns(uint)
    {
        return numbers[index];
    }
}
```

DUMMY BANK

DummyBank.sol

```
pragma solidity >=0.5.11;

contract DummyBank {
    struct User {
        uint id;
        string name;
        uint balance;
    }

    User[] private users;
    uint private nextId;

    function create(string memory name, uint amount)
        public
    {
        users.push(User(nextId, name, amount));
        nextId++;
    }

}
```

Struct declaration

array of struct

DUMMY BANK

```
    function read(uint id)
        public
        view
        returns (string memory, uint)
    {
        for (uint i = 0; i < users.length; i++ ) {
            if (users[i].id == id) {
                return (users[i].name, users[i].balance);
            }
        }
    }
```

Loops… Should we be worried about gas consumption?

struct field access

DUMMY BANK

```
32      // deposit
33      function update(uint id, uint value)
34          public
35      {
36          for (uint i = 0; i < users.length; i++) {
37              if (users[i].id == id) {
38                  users[i].balance = value;
39              }
40          }
41      }
```

Code duplication!!!

struct field write

REFACTORING

```
    function find(uint id)  internal view returns(uint) {
        for (uint i = 0; i < users.length; i++) {
            if (users[i].id == id)
                return i;
        }
    }

    function read(uint id) public  view returns (string memory, uint) {
        uint position = find(id);
        return (users[position].name, users[position].balance);
    }

    // deposit
    function update(uint id, uint value) public {
        uint position = find(id);
        users[position].balance = value;
    }

    function deleteById(uint id)  public {
        uint position = find(id);
        delete users[position];
    }
```

HMM.. ANY PROBLEMS WITH THIS CODE?

```
21      function find(uint id)  internal view returns(uint) {
22          for (uint i = 0; i < users.length; i++) {
23              if (users[i].id == id)
24                  return i;
25          }
26      }
27
28      function read(uint id) public  view returns (string memory, uint) {
29          uint position = find(id);
30          return (users[position].name, users[position].balance);
31      }
32
33      // deposit
34      function update(uint id, uint value) public {
35          uint position = find(id);
36          users[position].balance = value;
37      }
38
39      function deleteById(uint id)  public {
40          uint position = find(id);
41          delete users[position];
42      }
```

What if this is never the case?

What is the default return value?

FIX AND TEST

Avoid identifiers = 0

```
11      uint private nextId = 1;
12
13      function find(uint id)  internal view returns(uint) {
14          for (uint i = 0; i < users.length; i++) {
15              if (users[i].id == id)
16                  return i;
17          }
18          revert('user does not exists');
19      }
```

Revert everything if user not found!

SO FAR WE COVERED THE BASICS

1. Basic syntax
2. Basic types (uint, string, array)
3. Functions
4. Structs
5. delete and revert
6. Remix (compile, deploy, transactions, gas)

IMPROVED DUMMY BANK

- Issues in our DummyBank contract: arrays
  - Search time is linear => works bad when the bank has millions of clients
  - A loop increases the gas cost

SOLUTION: MAPS

Declaration

Update

Read

Delete

```
contract DummyBank {
    struct User {
        string name;
        uint balance;
    }

    mapping(uint => User) private users;
    uint private nextId = 1;

    function create(string memory name, uint amount)
        public
    {
        users[nextId] = User(name, amount);
        nextId++;
    }

    function read(uint id) public  view returns (string memory, uint) {
        return (users[id].name, users[id].balance);
    }

    // deposit
    function update(uint id, uint value) public {
        users[id].balance = value;
    }

    function deleteById(uint id)  public {
        delete(users[id]);
    }
```

IMPROVED DUMMY BANK

Update with arrays

transaction cost	25664 gas
execution cost	3944 gas

Update with mappings

transaction cost	22253 gas
execution cost	533 gas

ADDING AN ADMIN TO OUR BANK

```
13      address owner;
14
15      modifier onlyOwner() {
16          require(msg.sender == owner);
17          _;
18      }
19
20      function create(string memory name, uint amount)
21          public
22          onlyOwner
23      {
24          users[nextId] = User(name, amount);
25          nextId++;
26      }
```

In Solidity you can define your own modifier!

`require` throws an error if condition is false

A placeholder for a function call (e.g., `create`)

modifier usage

`msg` - a special object holding metadata (see details on the next slide)

MSG

- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address payable`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message

Docs: https://solidity.readthedocs.io/en/develop/units-and-global-variables.html#block-and-transaction-properties

BACK TO DUMMY BANK WITH ADMIN

Set the owner to the address of the one who deploys the contract

```
13      address owner;
14
15      constructor() public {
16          owner = msg.sender;
17      }
```

Run as admin

```
transact to DummyBank.create pending ...

[vm] from:0x147...c160c to:DummyBank.create(string,uint256) 0x0fd...bcfb7 value:0 wei
data:0x46f...00000 logs:0 hash:0x00b...85d7c
```

RUN AS NON-ADMIN

Only an admin has the rights to append new clients in our bank system

```
[vm] from:0xca3...a733c to:DummyBank.create(string,uint256) 0x0fd...bcfb7
value:0 wei data:0x46f...00000 logs:0 hash:0x100...57ae8
Debug

transact to DummyBank.create errored: VM error: revert.
revert  The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should
be less than your current balance.      Debug the transaction to get more information.
```

A PEN STORE

- Minimal requirements (this list will grow):
  - Users can buy a pen in exchange for ether
  - We need a seller
  - We learn how to transfer coins in Ethereum
  - We learn a new modifier: `payable`

A TRIVIAL PEN STORE

The payable enables the address to receive ether

The function is payable, i.e, it can receive ether

Transfer coins to the seller in exchange for 1 pen

```
pragma solidity >= 0.5.11;

contract PenStore {
    mapping(address=>uint) public penBalance;
    address payable seller;

    constructor () public {
        seller = msg.sender;
    }

    function buyPen()
        public
        payable
    {
        // buy 1 pen
        penBalance[msg.sender] += 1;
        // transfer ether to seller address
        seller.transfer(msg.value);
    }
}
```