**Blockchain**

DLT is not databse – it’s not a single database.

The backbone of any **DLT** is a network of computers

Each computer is called “**node**”

* Each node has a **ladger** – in the form of a database.
* Main job of “**node**” – to keep its copy synchronized with everyone else. The **ledgers**’ changes are broadcaste to everyone else and everyone includes the new change in their own copy of the ledger. These changes are called **transactions**.

**Blockchain**

For constantly using DLT people invented in order to optimize their work – **block**.

Block – bundle multiple transactions

The order of transactions is important. In order to ensure the correct chronological order of events, the nodes are mathematically linking every block to the previous one – forming a linked list a.k.a chain.

The blockchain is a network of special computers, called nodes. The nodes are accountants that maintain an accounting book, called a **ledger**. In this ledger, the **nodes** append transactions – sending/receiving. The nodes maintain the ledger by storing the transactions in bundle called **blocks**. The **blocks** are always **ordered chronologically**. This order is maintained by linking each new block to the previous one by its unique identifier called **block hash**.

**Secret Recovery Phrase – Very Imporant to be saved.**

Recovery Phrase:

vast oyster prison snow lunch exhaust cinnamon cook track eagle angry tragic

my main wallet address:

0xbf65e959d87aeB6351a3A92CBedD552cA432063c

Just in case wallet address:

0x1F1149eeeb05e118a42FC29759267eE77cbe76C5

**Site for tracking your transactions** <https://ropsten.etherscan.io/>

**Solidity**

Smart contracts in Ethereum is written in language called **Solidity**. Solidity is mixture of JavaScript and Python.

The way that you trigger functionalities of a smart contract is through calls and transactions towards Solidity functions.

**Calls** are used for **reading data** from the blockchain.

**Transactions** are used for **modifying the state** of the blockchain.

**Types** in solidity:

// SPDX-License-Identifier: MIT

contract ValueTypes {

  bool public b = true;

  uint public u = 123; // uint = uint256 0 to 2\*\*256 - 1

                       //        uint8   0 to 2\*\*8 - 1

                       //        uint8   0 to 2\*\*16 - 1

  int public i = -123; // int = int256   -2\*\*127 to 2\*\*127 - 1

  int public minInt = type(int).min; // find minimum value

  int public maxInt = type(int).max; // find maximum

  address public addr = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;

  bytes32 public b32 = 0x935a684a98fa80f5ddb195ad5926756630fcb9b7fdfcbe77d826ace24f979ab0;// hash maybe with 32 byes

}

Functions:

contract FunctionIntro {

    function add(uint x, uint y) external pure returns (uint) {

        return x + y;

    }

    function sub(uint x, uint y) external pure returns (uint) {

        return x - y;

    }

}

3 types of state:

* **Local** – variables declared in function
* **State** – store data into the blockchain. Declared directly inside contract
* **Global -** msg.sender, block.timeStamp, block.number

contract Variables {

    uint public myUint; // state variable means stored in the blockchain

    uint public num;

    uint public count;

    function foo() external {

        uint notStateVariable; // this is local variable, existing only when function executes and then variable will be gone

    }

    function globalVars() external view returns (address, uint, uint) {

        address sender = msg.sender; // global variable that stores the address clled this function.

        uint timestamp = block.timeStamp; // stores unix timestamp when this functions was callsed

        uint blockNum = block.number; // stores current block number

        return (sender, timestamp, blockNum)

    }

    // view function

    function addToNum(uint x) external view returns (uint) {

        return num + x; // view function because read data from the state

    }

    // pure function

    function add(uint x, uint y) external pure returns (uint) {

        return x + y; // because the code inside does not read any data from state, contracts or blockchain

    }

    // neither pure or view because it write something to the state. It's not readonly

    function inc() external {

        counter += 1;

    }

}

**Pure** functions & **View** functions

* **View** functions can read data from the blockchain – **readonly** funcion
* **Pure** functions do not read anything from the blockchain

Default values:

contract DefaultValues {

    // Default values

    bool public b; // false

    uint public u; // 0

    int public i; // 0

    address public a; // 0x000.. sequence of 40 zeroes

    bytes32 public b32; // 0x00.. sequence of 64 zeroes

}

Constants:

They reduce the gas cost when the transaction is made.

**Errors:**

    // require throwns an error

    function testRequire(uint \_i) public pure {

        require(\_i < 10, "i > 10");

    }

    // both are same

    function testRevert(uint \_i) public pure {

        if (\_i > 10) {

            revert("i > 10");

        }

    }

    // use assert to check conditions that should be always equal to true

    function testAssert() public view {

        assert(num == 123);

    }

    function foo() public {

        // accidentally update num

        num += 1;

    }

    // custom error

    error MyError(address caller, uint i);

    function testCustomError(uint \_i) public pure {

        if (\_i > 10) {

            revert MyError(msg.sender, \_i);

        }

    }

With custom error we save gas.

**Function modifiers:**

    // function modifier

    bool public paused;

    uint public count;

    function setPause(bool \_paused) external {

        paused = \_paused;

    }

    modifier whenNotPaused() {

        require(!paused, "paused");

        \_; // this line of code tells to run the functions as modifier

    }

    function int() external whenNotPaused {

        counter += 1;

    }

    function dec() external whenNotPaused {

        count -= 1;

    }

    // sandwich

    modifier cap(uint \_x) {

        require(\_x < 100, "x >= 100");

        \_;

        // more code will be executed after funcion in this case incBy() is done.

        count \*= 2;

    }

    function incBy(uint, \_x) external whenNotPaused cap(\_x) {

        count += 1;

    }

**Constructor:**

    // constructor

    address public owner;

    uint public x;

    constructor(uint \_x) {

        owner = msg.sender;

        x = \_x;

    }

**Example of owner with constructor:**

contract Ownable {

    address public owner;

     constructor () {

         owner = msg.sender;

     }

     modifier onlyOwner() {

         require(msg.sender == owner, "not owner");

         \_;

     }

     function onlyOwnerCalCallThisFunc() external onlyOwner {

         // code

     }

     function anyOneCanCall() external {

         // code

     }

     function setOwner(address \_newOwner)  external onlyOwner {

         require(\_newOwner != address(0), "invalid address");

         owner = \_newOwner;

     }

}

**Basic example of return many variables:**

    function returnMany() public pure returns (uint, bool) { // public because in another example we'll be using that function inside of object

        return (1, true);

    }

    // this uses more gas!

    function named() public pure returns (uint x, bool b) {

        return (1, true);

    }

    // this saves a little bit of gas

    function assigned() public pure returns (uint x, bool b) {

        x = 1;

        b = true;

    }

    // Destructuring Assigment

    function destructionAssigments() public pure {

        (uint x,  bool b) = returnMany(); // this way we catch the data coming from returnMany

        (,  bool \_b) = returnMany(); // only catch the second output

    }

**Arrays:**

contract Example {

    uint[] public nums = [1, 2, 3]; // initialize array

    uint[3] public numsFixed; // fixed size of array

    function examples() external {

        nums.push(4); // [1, 2, 3, 4]

        uint x = nums[1]; // get first element

        nums[2] = 777; // [1, 2, 777, 4]

        delete nums[1]; // [1, 0, 777, 4] delete sets to the default

        nums.pop(); // [1, 0, 777]

        uint len = nums.length;

        // create array in memory

        uint[] memory a = new uint[](5); // array in memory with size of 5

        a[1] = 123; // working with array in memory

        // a.pop(); not working with array in memory

        // a.push(); not working with array in memory

    }

    // return array which is in memory but it's not recomended because uses a lot of gas and the function is unusable

    function returnArray() external view returns (uint[] memory) {

        returns nums;

    }

}

**Remove element of array by shifting:**

contract Example {

    uint[] public arr;

    function example() public {

        arr = [1, 2, 3];

        delete arr[1]; // [1, 0, 3]

    }

    // [1, 2, 3] -- remove(1) --> [1, 3, 3] --> [1, 3]

    // [1, 2, 3, 4, 5, 6] -- remove(2) --> [1, 2, 4, 5, 6, 6] --> [1, 2, 4, 5, 6]

    // [1] -- remove(0) --> [1] --> []

    function remove(uint \_index) public {

        require(\_index < arr.length, "index out of bound");

        for (uint i = \_index; i < arr.length - 1; i++) {

            arr[i] = arr[i + 1];

        }

        arr.pop();

    }

    function test() external {

        arr = [1, 2, 3, 4, 5];

        remove(2);

        // [1, 2, 4, 5]

        assert(arr[0] == 1);

        assert(arr[1] == 2);

        assert(arr[2] == 4);

        assert(arr[3] == 5);

        assert(arr.length == 4);

        arr = [1];

        remove(0);

        // []

        assert(arr.length == 0);

    }

}

**Remove element by shifting to last element and .pop() it:**

// remove array element by shifting elements to left

// [1, 2, 3, 4, 5, 6] -- remove(2) --> [1, 2, 4, 5, 6, 6] --> [1, 2, 4, 5, 6]

contract ArrayReplaceLast {

    uint[] public arr;

    // [1, 2, 3, 4]  -- remove(1) --> [1, 4, 3]

    // [1, 4, 3]     -- remove(2) --> [1, 4]

    function remove(uint \_index) public {

        arr[\_index] = arr[arr.length - 1];

        arr.pop();

    }

    function test() external {

        arr = [1, 2, 3, 4];

        remove(1);

        // [1, 4, 3]

        assert(arr.length == 3);

        assert(arr[0] == 1);

        assert(arr[1] == 4);

        assert(arr[2] == 3);

        remove(2);

        // [1, 4]

        assert(arr.length == 2);

        assert(arr[0] == 1);

        assert(arr[1] == 4);

    }

}

Solidity again:

pragma solidity ^0.8.0;