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);

    }

}

Mapping:

// Mapping

// How to declare a mapping (simple and nested)

// Set, get, delete

// ["alice", "bob", "charlie"]

// { "alice": true, "bob": true, "charlie": true }

contract Mapping {

    mapping(address => uint) public balances;

    mapping(address => mapping(address => bool)) public isFriend;

    mapping(address => bool) public inserted;

    address[] public keys;

    function set(address \_key, uint \_val) external {

        balances[\_key] = \_val;

        if (!inserted[\_key]) {

            inserted[\_key] = true;

            keys.push(\_key);

        }

    }

    function getSize() external view returns (uint) {

        return keys.length;

    }

    function first() external view returns (uint) {

        return balances[keys[0]];

    }

    function last() external view returns (uint) {

        return balances[keys[keys.length - 1]];

    }

    function get() external view returns (uint) {

        return balances[keys[\_i]];

    }

    function examples() external {

        balances[msg.sender] = 123;

        uint bal = balances[msg.sender];

        uint bal2 = balances[address(1)]; // 0

        balances[msg.sender] += 456; // 123 + 456 = 579

        delete balances[msg.sender]; // 0

        isFriend[msg.sender][address(this)] = true;

    }

}

Struct example:

contract structs {

    struct Car {

        string model;

        uint year;

        address owner;

    }

    Car public car;

    Car[] public cars;

    mapping(address => Car[]) public carsByOwner;

    function examples() external {

        Car memory toyota = Car("Toyota", 1990, msg.sender); // in memory and only exist when this function was called

        Car memory lambo = Car({model: "Lamborghini", year: 1980, owner: msg.sender});

        Car memory tesla; // will be with default values

        tesla.model = "Tesla";

        tesla.year = 2010;

        tesla.owner = msg.sender;

        cars.push(toyota);

        cars.push(lambo);

        cars.push(tesla);

        cars.push(Car("Ferrari", 2020, msg.sender));

        // must change from memory to storage in order to make changes on \_car variable

        Car storage \_car = cars[0];

        \_car.year = 1999;

        delete \_car.owner;

        delete cars[1];

    }

}

Enum example:

contract Enum {

    enum Status {

        None,

        Pending,

        Shipped,

        Completed,

        Rejected,

        Canceled

    }

    Status public status;

    struct Order {

        address buyer;

        Status status;

    }

    Order[] public orders;

    function get() view public returns (Status) {

        return status;

    }

    function set(Status \_status) external {

        status = \_status;

    }

    function ship() external {

        status = Status.Shipped;

    }

    function reset() external {

        delete status;

    }

}

Deploy any contract:

contract Proxy {

    event Deploy(address);

    function deploy(bytes memory \_code) external payable returns (address addr) {

        assembly {

            // create (v, p, n)

            // v = amount of ETH to send

            // p = pointer in memory to start of code

            // n = size of code

            addr := create(callvalue(), add(\_code, 0x20), mload(\_code))

        }

        require(addr != address(0), "deploy failed");

        emit Deploy(addr);

    }

    function execute(address \_target, bytes memory \_data) external payable{

        (bool success, ) = \_target.call{value:msg.value}(\_data);

        require(success, "failed");

    }

}

contract Helper {

    function getByCode1() external pure returns (bytes memory) {

        bytes memory bytecode = type(TestContract1).creationCode;

        return bytecode;

    }

    function getBytecode2(uint \_x, uint \_y) external pure returns (bytes memory) {

        bytes memory bytecode = type(TestContract2).creationCode;

        return abi.encodePacked(bytecode, abi.encode(\_x, \_y));

    }

    function getCalldata(address \_owner) external pure returns (bytes memory) {

        return abi.encodeWithSignature("setOwner(address)", \_owner);

    }

}

Data Locations:

contract DataLocations {

    struct myStruct {

        uint foo;

        string text;

    }

    mapping(address => MyStruct) public myStructs;

    function examples(uint[] memory y, string memory s) external return (uint[] memory) {

        myStructs[msg.sender] = MyStruct({foo: 123, text: "bar"});

        MyStruct memory readonly = myStructs[msg.sender];

        readonly.foo = 456;

        uint[] memory memArr = new uint[](3);

        memArr[0] = 234;

        return memArr;

    }

}

Event:

contract Event {

    event Log(string message, uint val);

    // up to 3 index

    event IndexedLog(address indexed sender, uint val);

    function example() external {

        emit Log("foo", 1234);

        emit IndexedLog(msg.sender, 789);

    }

    event Message(address indexed \_from, address indexed \_to, string message);

    function sendMessage(address \_to, string calldata message) external {

        emit Message(msg.sender, \_to, message);

    }

}

Inheritance

contract A {

    // virual means this function can be inherited and costumized by child contract

    function foo() public pure virtual returns (string memory) {

        return "A";

    }

    function bar() public pure virtual returns (string memory) {

        return "A";

    }

    // more code

    function baz() public pure returns (string memory) {

        return "B";

    }

}

contract B is A {

    function foo() public pure override returns (string memory) {

        return "B";

    }

    function bar() public pure virtual override returns (string memory) {

        return "B";

    }

    // more code here

}

contract C is B {

    function bar() public pure override returns (string memory) {

        return "C";

    }

}

Solidity again:

pragma solidity ^0.8.0;