Laborator 2

Elemente generale de programare și dezvoltare folosind Solidity

Note

1. Un tutorial cu elemente de bază - https://www.tutorialspoint.com/solidity/index.htm
2. Un tutorial mediu-avansat - https://solidity-by-example.org/
3. Exemplele de la link-ul din punctul 2 sunt explicate și aici https://www.youtube.com/channel/UCJWh7F3AFyQ_x01VKzr9eyA
4. Tutorialul cu Truffle și Ganache - https://trufflesuite.com/tutorial/index.html
5. Ethereum .NET – www.ethereum.org
6. How To Create An ENTIRE NFT Collection (10,000+) & MINT In Under 1 Hour Without Coding Knowledge - YouTube

Concepte teoretice și definiții elementare

Ce este ETHER?

"Ether is the transactional token that facilitates operations on the Ethereum network. All the programs and services linked with the Ethereum network require computing power (and that computing power is not free). Ether is a form of payment for network participants to execute their requested operations on the network." - https://www.investopedia.com/terms/e/ether-cryptocurrency.asp

Ce este *WEI* ?

"Wei is the smallest denomination of ether—the cryptocurrency coin used on the Ethereum network. One ether = 1,000,000,000,000,000,000 wei (1018). The other way to look at it is one wei is one quintillionth of an ether." - https://www.investopedia.com/terms/w/wei.asp

1 ETHER = 10^{18} WEI

Ce este *GAS*?

Gas este o unitate computațională.

Gaz cheltuit (*gas spent*) – reprezintă cantitatea totală de gaz folosit pentru o tranzacție

Prețul gas-ului (*gas price*) - reprezintă cantitatea de ether pe care ești dispus să o plătești pentru gaz.

Atenție:

- Gazul necheltuit este returnat.
- Există două limite ca și praguri al cantității de gaz pe care îl poți cheltui:

- *gas limit* – cantitatea maximă de gaz pe care dorim să o utilizăm pentru tranzacțiile care le trimitem.
- *block gas limit* – cantitatea maximă de gaz permis întrun bloc, configurate și setat de către rețea

Aplicații practice

1. Utilizarea întregii cantități de gaz pe care îl trimitem cauzează eșecul procesării tranzacției

```
1 pragma solidity ^0.8.10;
2
3 contract Gas {
4     uint public contor = 0;
5
6     //** Utilizarea întregii cantități de gaz pe care
7     //** îl trimitem cauzează eșecul procesării tranzacției.
8     //** Schimbările de stare nu se mai pot
9     //** face după momentul execuției.
10    //** Gazul cheltuit nu este rambursabil
11    function runningInfinite() public {
12        //** mai jos avem o buclă pe care o rulăm până
13        //** când gaz-ul este cheltuit și tranzacția
14        //** și procesarea acesteia eșuează
15        while (true) {
16            contor += 1;
17        }
18    }
19 }
```

2. Utilizarea funcțiilor *mapping*

Sintaxa: `mapping(tipCheie => tipValoare)`

- `tipCheie` – reprezintă tipuri de valori. Exemple: uint, address sau bytes
- `tipValoare` – reprezintă orice tip care să includă alt mapping, vector sau tip de valoare.
- Atenție: Mapping-urile nu sunt iterabile.

```
1 pragma solidity ^0.8.10;
2
3 contract InstructiuniMapping
4 {
5     //** mapping de la adresa la uint
6     mapping(address => uint) public mappingCurent;
7
8     function getAddress(address adresa) public view returns (uint)
9     {
10        //** Un mapping întotdeauna returnează o valoare.
```

```
11          //** Dacă valoarea nu a fost folosită,
12          //** o valoare implicită este returnată
13          return mappingCurent[adresa];
14      }
15
16      function setAddress(address adresa, uint valoare) public
17      {
18          //** actualizează valoarea la adresa menționată
19          mappingCurent[adresa] = valoare;
20      }
21
22      function removeAddress(address adresa) public
23      {
24          //** inițializăm valoarea la valoarea implicită
25          delete mappingCurent[adresa];
26      }
27  }
28
29  contract InstructiuniMappingNested
30  {
31      //** mapping-uri imbricate (sau nested) -
32      //** mapping de la o adresă la alt mapping
33      mapping(address => mapping(uint => bool)) public nImbricat;
34
35      function get(address adresaImbricat, uint val) public view returns
36  (bool)
37      {
38          //** Există posibilitatea de a obține
39          //** valori de la un mapping imbricat
40          //** chiar dacă nu este inițializată
41          return nImbricat[adresaImbricat][val];
42      }
43
44      function set(
45          address adresaImbricat,
46          uint val,
47          bool booleanValue
48      ) public
49      {
50          nImbricat[adresaImbricat][val] = booleanValue;
51      }
52
53      function remove(address adresaImbricat, uint val) public
54      {
55          delete nImbricat[adresaImbricat][val];
56      }
57  }
```

3. Vectori – Exemplu cu operații fundamentale, dimensiune fixă sau dimensiune dinamică

```
pragma solidity ^0.8.10;

contract Array
{
    //** moduri de inițializare a vectorilor
    uint[] public vector;
    uint[] public vector2 = [3, 12, 32];

    //** vectori cu dimensiune fixă, toate elementele
    //** sunt inițializate cu 0
    uint[10] public myFixedSizeArr;

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

    //** Solidity returnează întregul vector.
    //** Această metodă ar trebui evitată pentru vectori
    //** care pot crește nedeterminat în lungime.
    function getVector() public view returns (uint[] memory)
    {
        return vector;
    }

    function push(uint index) public
    {
        //** adăugarea de elemente în vector
        //** lungimea vectorului va crește în lungime cu 1.
        vector.push(index);
    }

    function pop() public
    {
        //** stergerea ultimului element din vector.
        //** această funcție reduce dimensiunea vectorului cu 1
        vector.pop();
    }

    function getLungime() public view returns (uint)
    {
        return vector.length;
    }
```

```
44
45      function sterge(uint index) public
46      {
47          //** ștergerea nu schimbă lungimea vectorului.
48          //** resetează valoarea de la indexul menționat la valoarea
49  implicită,
50          //** în cazul de față la 0
51          delete vector[index];
52      }
53
54      function exemplu() external {
55          //** declarea unui vector în memorie, cu dimensiune fixă
56          uint[] memory a = new uint[](5);
57      }
58  }
```

4. Vectori – Ștergerea de elemente dintrun vector prin rotația elementelor de la dreapta la stânga. Studiu de caz despre cum sa nu implementezi, rulați exemplul, studiați comportamentul.

```
 1  pragma solidity ^0.8.10;
 2
 3  contract StergeElementVectorPrinShiftare
 4  {
 5      // [1, 2, 3] -- remove(1) --> [1, 3, 3] --> [1, 3]
 6      // [1, 2, 3, 4, 5, 6] -- remove(2) --> [1, 2, 4, 5, 6, 6] --> [1, 2,
 7  4, 5, 6]
 8      // [1, 2, 3, 4, 5, 6] -- remove(0) --> [2, 3, 4, 5, 6, 6] --> [2, 3,
 9  4, 5, 6]
10      // [1] -- remove(0) --> [1] --> []
11
12      uint[] public vector;
13
14      function sterge(uint indexV) public
15      {
16          require(indexV < vector.length, "indexul se afla in afara
17  limitelor");
18
19          for (uint poz = indexV; poz < vector.length - 1; poz++)
20          {
21              vector[poz] = vector[poz + 1];
22          }
23          //** stergem
24          vector.pop();
25      }
26
27      function testare() external
```

```
28     {
29         vector = [1, 2, 3, 4, 5];
30         sterge(2);
31 
32         assert(vector[0] == 1);
33         assert(vector[1] == 2);
34         assert(vector[2] == 4);
35         assert(vector[3] == 5);
36         assert(vector.length == 4);
37 
38         vector = [1];
39         sterge(0);
40 
41         assert(vector.length == 0);
42     }
43 }
```

5. Definirea propriilor tipuri prin crearea structurilor.

```
1  pragma solidity ^0.8.10;
 2 
 3 contract Taskuri
 4 {
 5     struct Task
 6     {
 7         string text;
 8         bool realizat;
 9     }
10 
11     // vector de structuri
12     Task[] public taskuri;
13 
14     function creareTask(string memory numeTask) public
15     {
16         // există trei moduri de inițializare a vectorilor
17         // - printrun apel al funcției
18         taskuri.push(Task(numeTask, false));
19 
20         // - mapping prin valoare cheie
21         taskuri.push(Task({text: numeTask, realizat: false}));
22 
23         // - inițializarea unei structuri implicite și apoi prin
24 actualizarea acesteia
25         Task memory task;
26         task.text = numeTask;
27 
28         // adăugăm
29         taskuri.push(task);
```

```
30      }
31
32      function get(uint indexTask) public view returns (string memory text,
33  bool completed)
34      {
35          Task storage task = taskuri[indexTask];
36          return (task.text, task.realizat);
37      }
38
39      // actualizăm taskul
40      function actualizare(uint indexTask, string memory _text) public
41      {
42          Task storage todo = taskuri[indexTask];
43          todo.text = _text;
44      }
45
46      // actualizăm progresul
47      function actualizareProgres(uint indexTask) public
48      {
49          Task storage task = taskuri[indexTask];
50          task.realizat = !task.realizat;
51      }
52  }
```