Experiment No. 2

Creating the Smart Contract

Aim: Creating and deploying Smart Contract using Solidity and Remix IDE.

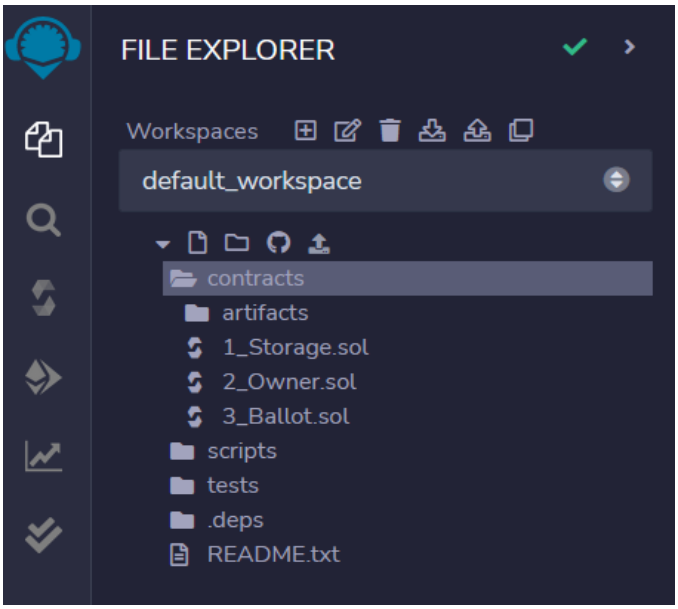
Theory:

- Smart contract is a computer program or a transaction protocol which is intended to automatically execute, control or document legally relevant events and actions according to the terms of a contract or an agreement.

- Remix IDE (Integrated Development Environment) is a web application that can be used to write, debug, and deploy Ethereum Smart Contracts.

- Solidity is a contract-oriented, high-level language for implementing smart contracts.

Step 1: Go to https://remix.ethereum.org/. Under the contracts folder, you will find some default contracts already given to us by Remix.



Step 2: right-click on the contracts and select New File. Name our file as Voting and type the following code. Save file with .sol extension.

```
pragma solidity ^0.6.6;

contract Voting{

    struct Candidate{

        uint id;

        string name;
```

```
        uint voteCount;
    }
    mapping (uint => Candidate) public candidates;
    uint public candidatecount;
    mapping (address => bool) public citizen;
    constructor() public{
        addCandidate("Godlin");
        addCandidate("Hilda");
    }
    function addCandidate(string memory _name) private{
        candidatecount++;
        candidates[candidatecount] = Candidate(candidatecount, _name, 0);
    }
    function vote(uint _candidateid) public{
        require(!citizen[msg.sender]);
        citizen[msg.sender] = true;
        candidates[_candidateid].voteCount ++;
    }
}
```

In above program with line **pragma solidity ^0.6.6;** the version of Solidity is specified as 0.6.6, as it is a stable version. Data type Candidate is defined with struct keyword. Mapping in Solidity acts like a hash table or dictionary in any other language. These are used to store the data in the form of key-value pairs, a key can be any of the built-in data types. In the Candidates mapping, we have the key as type uint. As said already, this will be the ID to identify the candidate. In the Citizen mapping, we are using the key as the address and boolean as the value. So initially, the boolean value for a citizen will be false and once they have cast their vote, it will change to true. By this, we can make sure that each citizen can cast their vote only once.

```
function addCandidate(string memory _name) private{
        candidatecount++;
        candidates[candidatecount] = Candidate(candidatecount,
_name, 0);
}
```

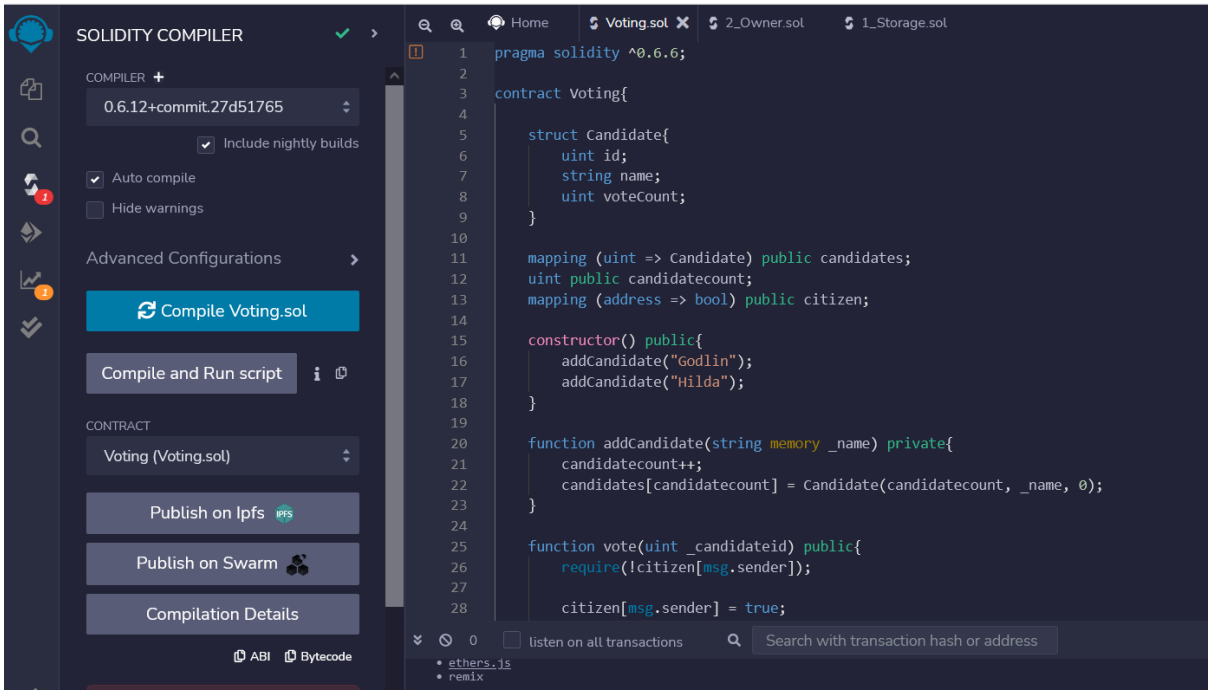
```
function vote(uint _candidateid) public{
        require(!citizen[msg.sender]);
        citizen[msg.sender] = true;
        candidates[_candidateid].voteCount ++;
}
```

This function is used to handle voting. The require(!citizen[msg.sender]) is used to check if the citizen is already voted. Cause in case they have already voted their boolean value will be True, so the condition will fail not allowing the citizen to vote again. If this is their first vote, we change the boolean value of the citizen as True and increment the voteCount of the particular candidate that the city chose to vote with the help of the candidateid

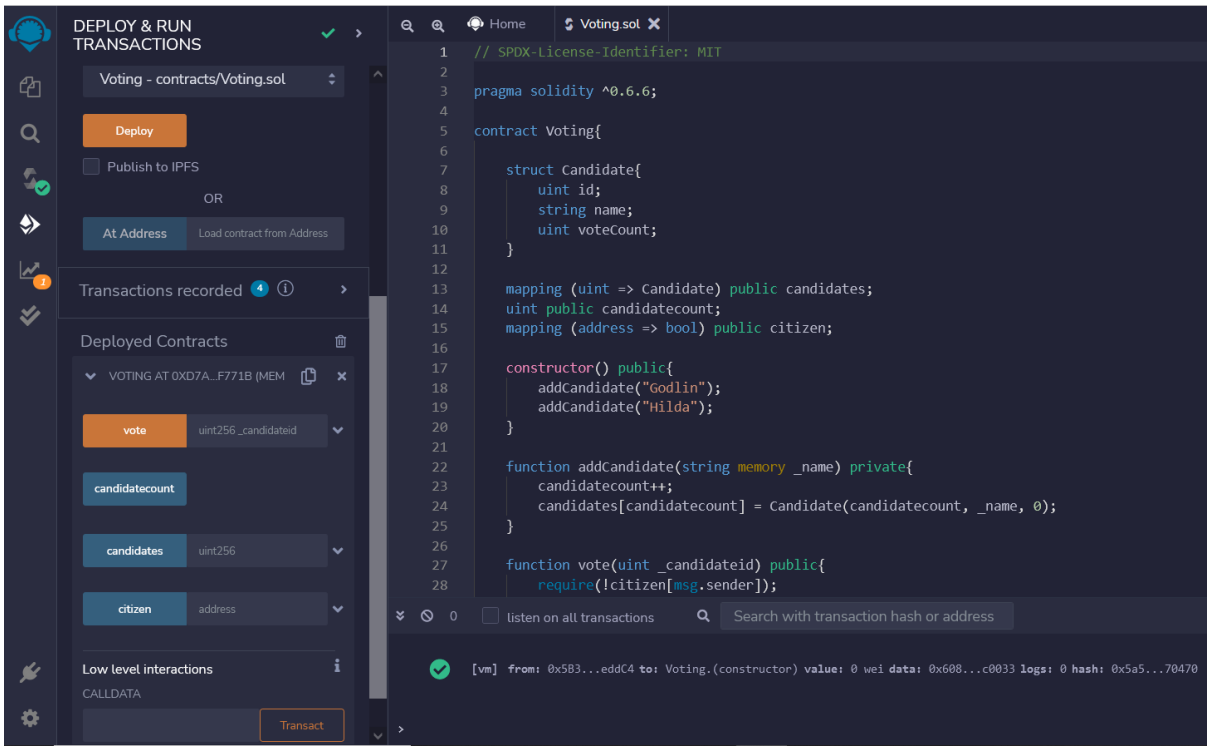
```
constructor() public{
        addCandidate("Godlin");
        addCandidate("Hilda");
    }
```

As most of you are already aware, a constructor is called at the beginning of a program. In our case, the constructor is called when we deploy our contract. So when we are deploying, we are adding two candidates named — Godlin and Hilda.

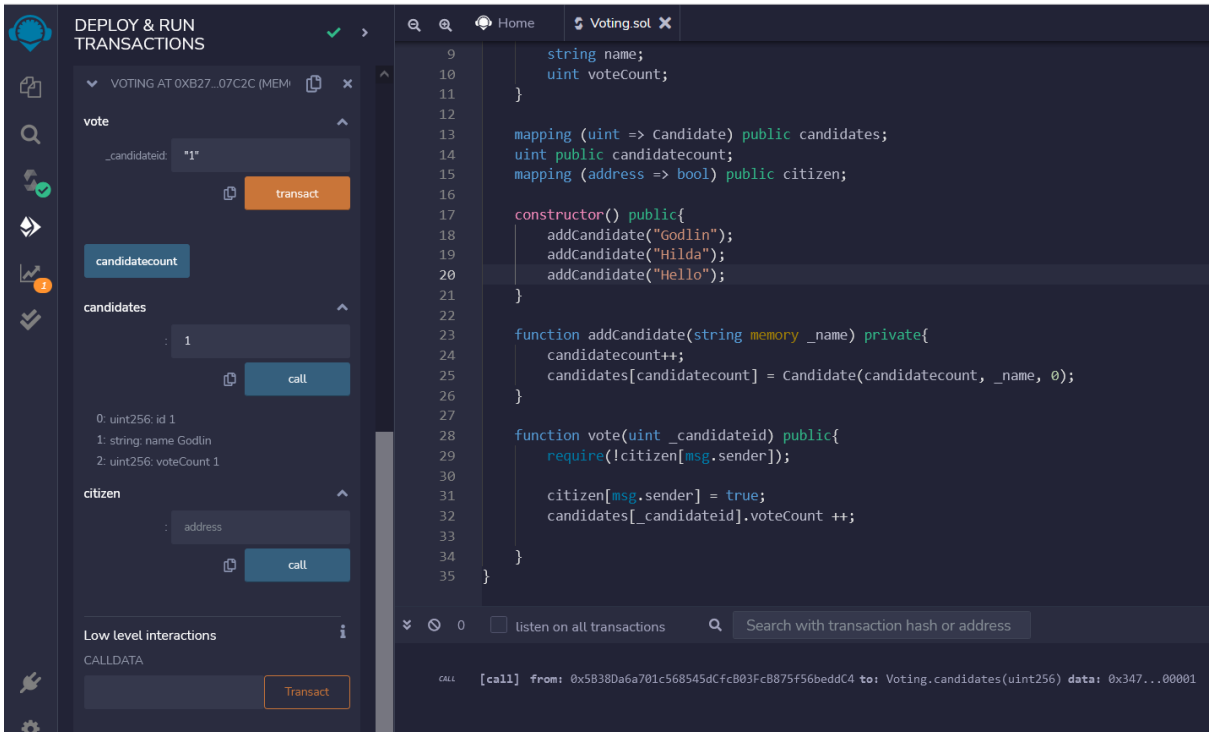
Step 3 : Click on the *solidity compiler* present on the left. You can select *Auto-compile* so our contract automatically compiles when we do some changes.

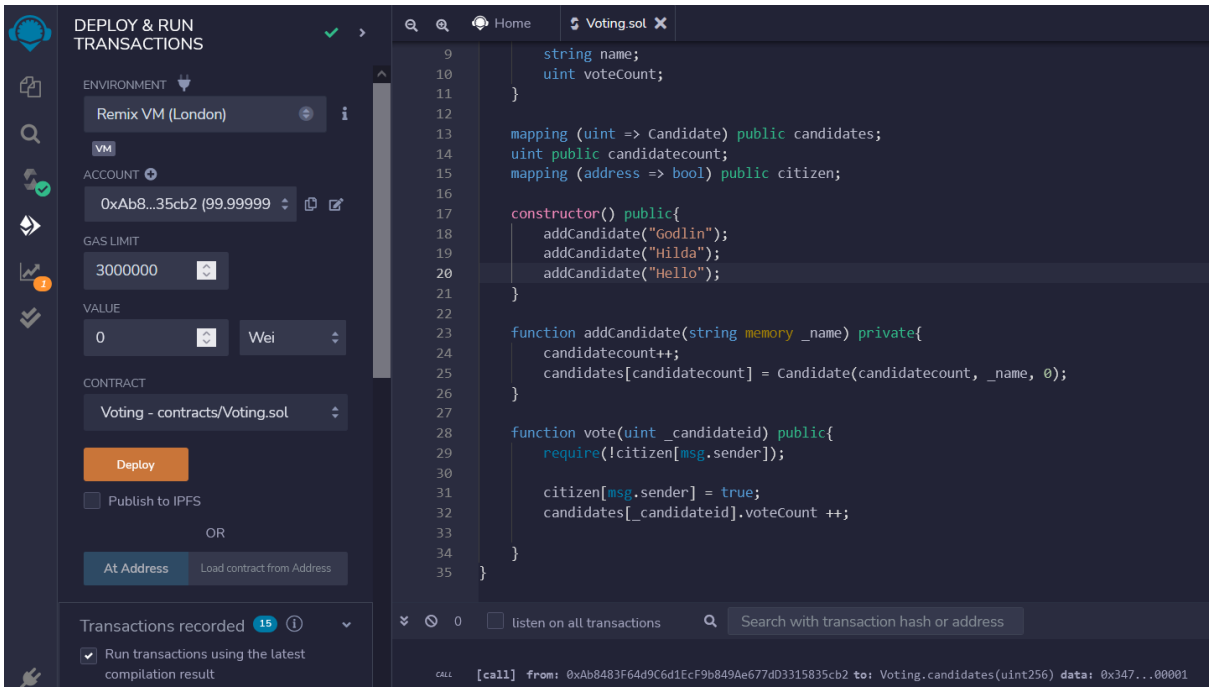


Step 4: After compiling our contract, now is the time to deploy our simple contract. For that click on run and deploy transactions on the left. Then click on Deploy. After your contract is successfully deployed, you will able to see your contract under the deployed contract.

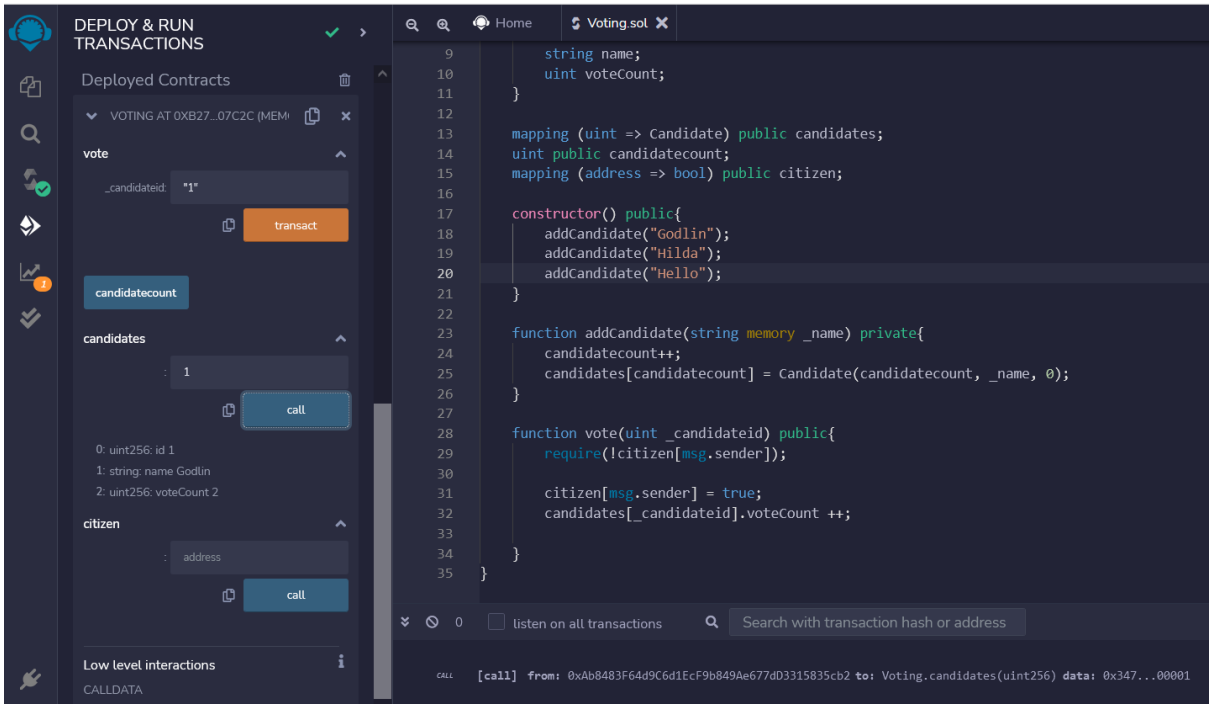


Step 5: Now we can access the information of the Candidate using the id. You will see the candidate mapping will require `uint256` as input. If you give the input as 1, you will be to see the details of our first candidate. In our case, the first candidate is Godlin.





Step 6: You will notice that the first account will have less ether when compared to the rest, that is because by default Remix will the first account to deploy our contract. If we want to deploy a contract we have to pay some ether as gas. Now select the second account and let's cast our vote. Go to our deployed contract present in the bottom and vote for your candidate.



Conclusion: We have successfully created and deployed the smart contract.

Review Questions

Q. 1 What is Remix IDE?

Q. 2 How smart contract is created using Remix IDE?

Q. 3 How the smart contract is deployed using Remix IDE?

Q. 4 What is Gas?

Q. 5 What is account in Remix IDE?