EXPERIMENT NO 05

Name: Tejas Gunjal **Class:** D20A

Roll: No: 24 **Batch:** B

Aim: Deploying a Voting/Ballot Smart Contract

Theory:

1. Concept of Decentralized Voting System

A decentralized voting system uses a smart contract to manage the entire voting process in a transparent and tamper-proof manner. The voting logic is written inside the contract, and once deployed, it cannot be altered.

Unlike traditional voting systems, there is no central authority controlling the votes. The smart contract ensures that:

- Each voter can vote only once.
- Votes are counted automatically.
- Results are transparent and verifiable.

Because the voting rules are encoded inside the contract, human intervention is minimized. Every vote is recorded as a transaction on the blockchain, making the system secure and resistant to manipulation.

2. Importance of require Statements in Solidity

In Solidity, the require statement is used to validate specific conditions before a function continues its execution. Since smart contracts operate on a blockchain where transactions are permanent and irreversible, it is essential to ensure that all necessary conditions are satisfied before modifying the contract's state. The require statement helps in enforcing these rules.

If the condition written inside require evaluates to false, the execution of the function stops immediately. Any state changes made during that transaction are reverted, meaning the contract returns to its previous state. This protects the blockchain from storing invalid or unauthorized data. Additionally, developers can include an error message inside require, which helps users understand why the transaction failed. Also, the Unused gas is refunded to the caller.

In general, require is commonly used to:

- Validate input values passed to functions
- Check whether the caller has permission to execute a function
- Ensure the contract is in the correct state before proceeding

Only when the specified condition is satisfied does the function continue executing the remaining logic.

In a decentralized voting system, maintaining fairness and correctness is critical. The require statement ensures that election rules are strictly enforced.

For example, it can be used to:

- Verify that a voter has voting rights.
- Ensure that a voter does not vote more than once.
- Restrict administrative functions to the chairperson.

By implementing these checks, the smart contract maintains security, prevents misuse, and ensures the integrity of the voting process.

3. Understanding mapping, storage, and memory

a) Mapping

A mapping is a key-value data structure used to associate one piece of data with another. It allows quick retrieval of information using a unique key. Its syntax is mapping(keyType => valueType). For example: mapping(address => Voter) public voters;

In a voting contract, mapping is commonly used to connect a voter's blockchain address with their voting details. This ensures that each voter's data can be accessed efficiently without searching through arrays.

Mappings are highly efficient for lookups but do not support iteration or length calculation, which makes them suitable for tracking user-specific records like voting status.

b) Storage

Storage refers to the permanent data area of a smart contract. Any variable declared at the contract level is stored permanently on the blockchain.

Data in storage:

- Remains available throughout the contract's lifetime.
- Persists across multiple transactions.
- Requires higher gas cost for modification.

For example, candidate details and voter records are stored in storage so that the information remains available until the contract is destroyed.

c) Memory

Memory is temporary and exists only during the execution of a function. Once the function finishes execution, memory variables are removed.

Memory is typically used for:

- Function parameters
- Temporary calculations
- Short-term data handling

Since memory does not permanently modify blockchain data, it consumes less gas compared to storage operations. Efficient smart contract design requires proper use of storage and memory to reduce gas consumption.

4. Use of bytes32 instead of string

When designing a Ballot contract, data types for storing proposal names must be chosen carefully.

bytes32 is a fixed-length data type that occupies exactly 32 bytes of space. Because of its fixed size:

- It is easier for the Ethereum Virtual Machine to process.
- Comparisons are faster.
- Gas usage is lower.

However, it limits text length and is less flexible.

On the other hand, string is dynamically sized, meaning it can store variable-length text. While it improves readability and user interaction, it requires more complex internal handling, leading to higher gas consumption.

Therefore:

- bytes32 is preferred when efficiency and performance are the main goals.
- string is preferred when flexibility and user-friendly input are required.

5. Structure of Ballot Smart Contract

Ballot smart contract consists of the following main components:

- **Structs** – Voter and Proposal structures are defined to organize election data. The Voter struct stores information such as voting weight, voting status, delegated address, and selected proposal. The Proposal struct stores the proposal name and total vote count.
- **State Variables** – The contract includes chairperson (contract deployer), voters mapping (address to Voter), proposals dynamic array, and votingDeadline to restrict voting duration. These variables maintain the overall state of the election.
- **Events** – Events such as RightGranted, VoteDelegated, and Voted are declared to record important activities and improve transparency.
- **Constructor** – Initializes the chairperson, assigns initial voting rights, sets the voting deadline, and creates proposal entries during contract deployment.
- **Modifiers** – onlyChairperson ensures administrative control, while beforeDeadline restricts voting and delegation after the deadline.
- **Core Functions** – The contract provides functions to grant voting rights, delegate votes, cast votes, calculate the winning proposal, return the winner's name, and compute vote percentage.

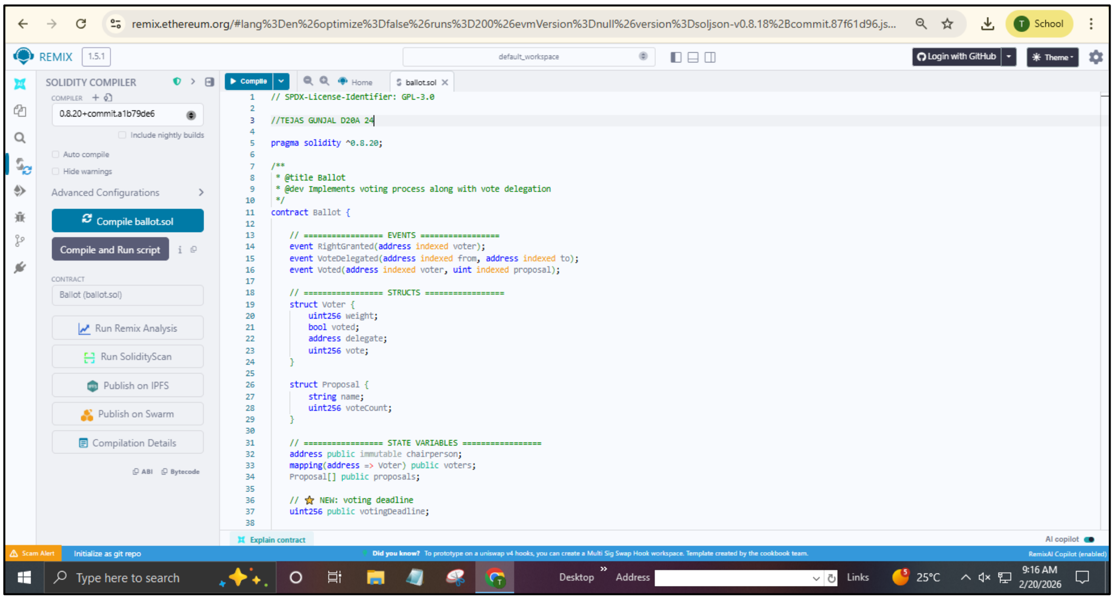
Code:

```
// SPDX-License-Identifier: GPL-3.0

//TEJAS GUNJAL D20A 24

pragma solidity ^0.8.20;

/**
 * @title Ballot
 * @dev Implements voting process along
with vote delegation
 */
contract Ballot {
   // ================= EVENTS
=================
   event RightGranted(address indexed
voter);
   event VoteDelegated(address indexed
from, address indexed to);
   event Voted(address indexed voter, uint
indexed proposal);
   // ================= STRUCTS
=================
   struct Voter {
      uint256 weight;
      bool voted;
      address delegate;
      uint256 vote;
   }

   struct Proposal {
      string name;
      uint256 voteCount;
   }

   // ================= STATE
VARIABLES =================
   address public immutable chairperson;
   mapping(address => Voter) public
voters;
   Proposal[] public proposals;

   // NEW: voting deadline
   uint256 public votingDeadline;

   // =================
CONSTRUCTOR =================

   constructor(string[] memory
proposalNames, uint256
durationInMinutes) {
      require(proposalNames.length > 0,
"No proposals provided");

      chairperson = msg.sender;
      voters[chairperson].weight = 1;

      // set voting deadline
      votingDeadline = block.timestamp +
(durationInMinutes * 1 minutes);

      for (uint256 i = 0; i <
proposalNames.length; i++) {
         proposals.push(Proposal({
            name: proposalNames[i],
            voteCount: 0
         }));
      }
   }
   // ================= MODIFIERS
=================
   modifier onlyChairperson() {
      require(msg.sender == chairperson,
"Only chairperson allowed");
      _;
   }

   // NEW: deadline check
   modifier beforeDeadline() {
      require(block.timestamp <
votingDeadline, "Voting period has
ended");
      _;
   }

   // ================= FUNCTIONS
=================

   function giveRightToVote(address
voter) external onlyChairperson {
      require(voter != address(0), "Invalid
address");
      require(!voters[voter].voted, "Already
voted");
      require(voters[voter].weight == 0,
"Already has voting right");
```

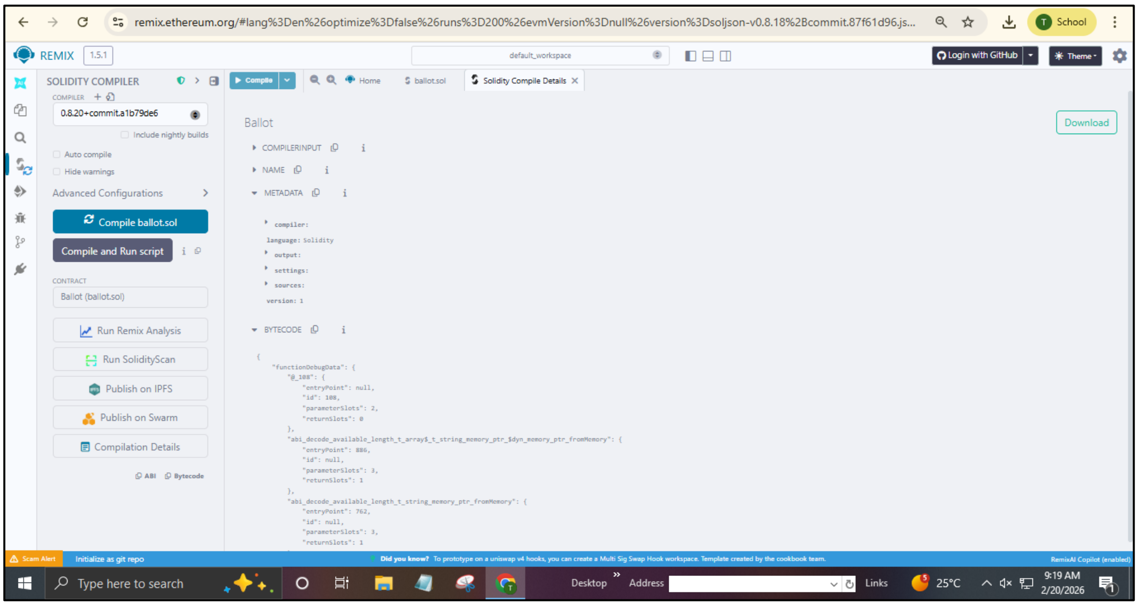
```
        voters[voter].weight = 1;
        emit RightGranted(voter);
    }
    function delegate(address to) external
beforeDeadline {
        Voter storage sender =
voters[msg.sender];

        require(sender.weight != 0, "No right
to vote");
        require(!sender.voted, "Already
voted");
        require(to != msg.sender, "Self-
delegation not allowed");

        while (voters[to].delegate !=
address(0)) {
            to = voters[to].delegate;
            require(to != msg.sender,
"Delegation loop detected");
        }
        Voter storage delegate_ = voters[to];
        require(delegate_.weight >= 1,
"Delegate has no voting right");

        sender.voted = true;
        sender.delegate = to;

        if (delegate_.voted) {
            proposals[delegate_.vote].voteCou
nt += sender.weight;
        } else {
            delegate_.weight += sender.weight;
        }
        emit VoteDelegated(msg.sender, to);
    }
    function vote(uint256 proposal) external
beforeDeadline {
        require(proposal < proposals.length,
"Invalid proposal");

        Voter storage sender =
voters[msg.sender];
        require(sender.weight != 0, "No right
to vote");
        require(!sender.voted, "Already
voted");
        sender.voted = true;
        sender.vote = proposal;
        proposals[proposal].voteCount +=
sender.weight;
        emit Voted(msg.sender, proposal);
    }
    function winningProposal() public view
returns (uint256 winningProposal_) {
        uint256 winningVoteCount = 0;

        for (uint256 p = 0; p <
proposals.length; p++) {
            if (proposals[p].voteCount >
winningVoteCount) {
                winningVoteCount =
proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }
    function winnerName() external view
returns (string memory winnerName_) {
        winnerName_ =
proposals[winningProposal()].name;
    }
    function getProposalsCount() external
view returns (uint256) {
        return proposals.length;
    }
    // NEW FEATURE: vote percentage
    function getVotePercentage(uint256
proposalIndex)
        external
        view
        returns (uint256 percentage)
    {
        require(proposalIndex <
proposals.length, "Invalid proposal");

        uint256 totalVotes = 0;
        for (uint256 i = 0; i <
proposals.length; i++) {
            totalVotes +=
proposals[i].voteCount;
        }
        if (totalVotes == 0) return 0;
        percentage =
            (proposals[proposalIndex].voteCou
nt * 100) /
            totalVotes;
    }
}
```

Output:

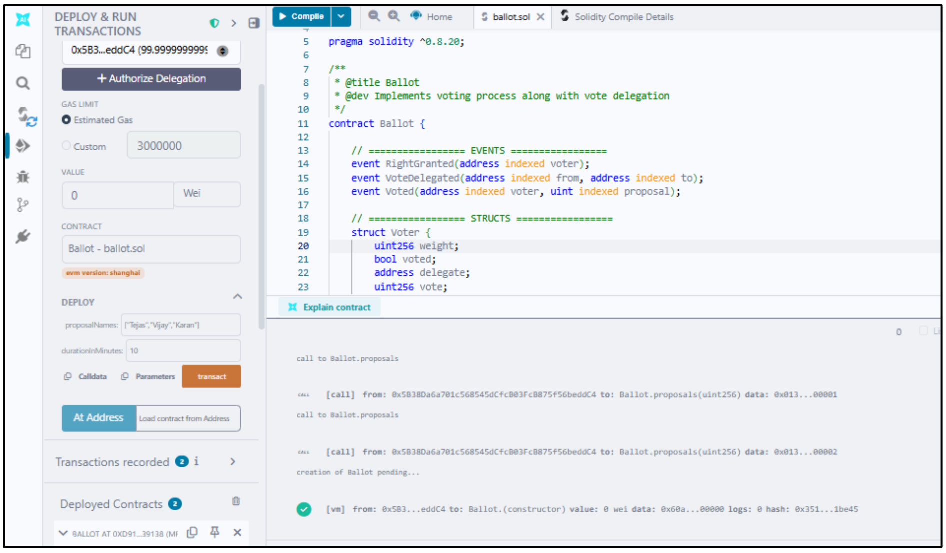
➢ Solidity compiler version 0.8.20 selected in Remix IDE for compiling the Ballot contract



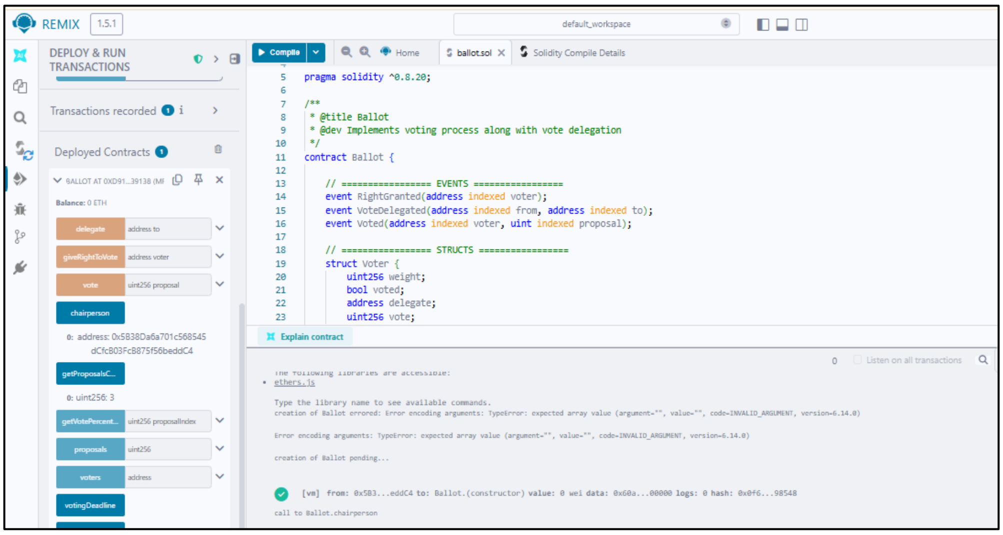
➢ Ballot contract successfully compiled showing compiler details and bytecode information in Remix IDE



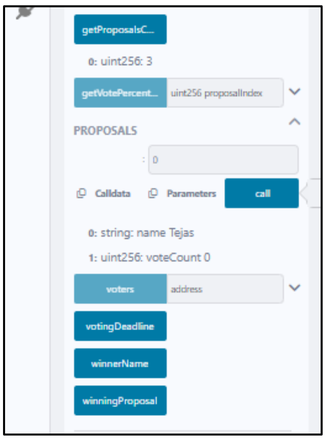
➢ Deploying the Ballot contract with proposal names & duration of 10 minutes in Remix IDE

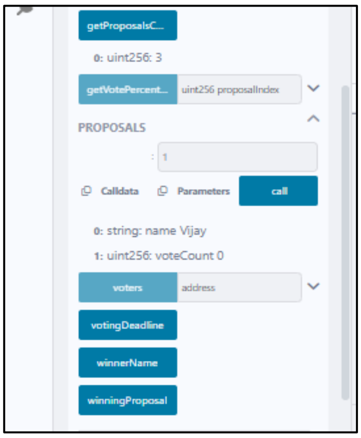


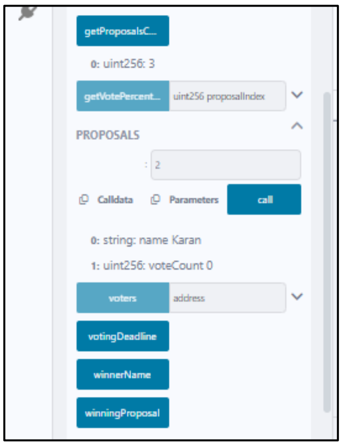
➢ Viewing the chairperson address after successful contract deployment to verify the deployer is correctly set.



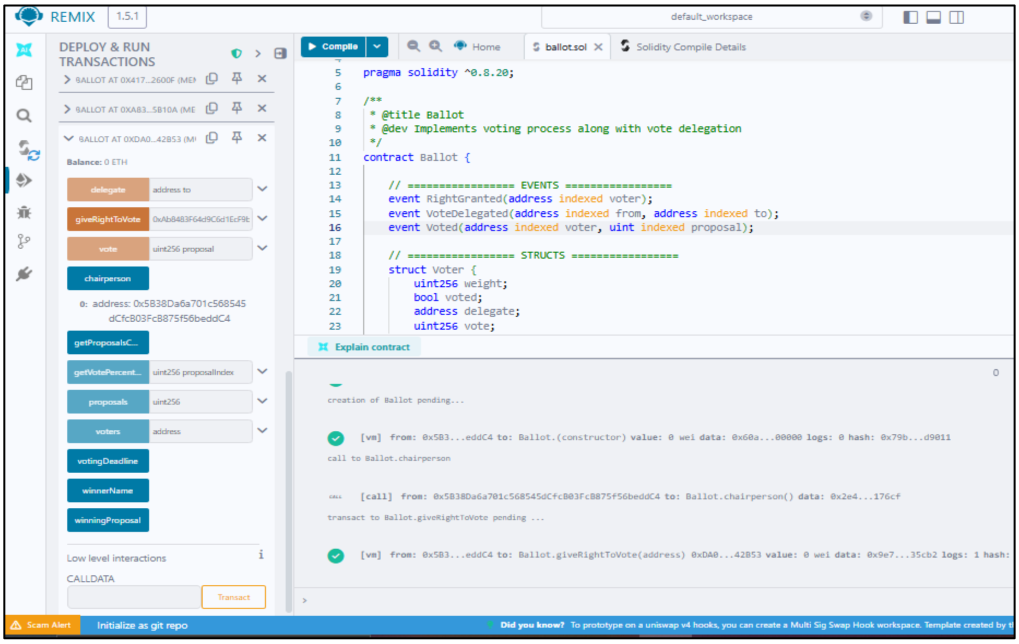
➢ Showing initial vote counts of proposals after deployment



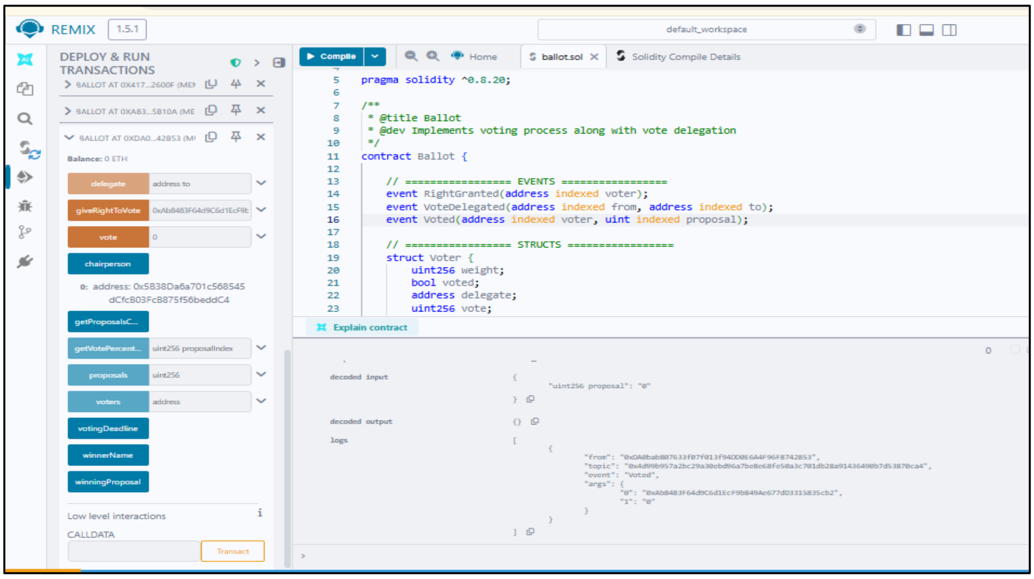




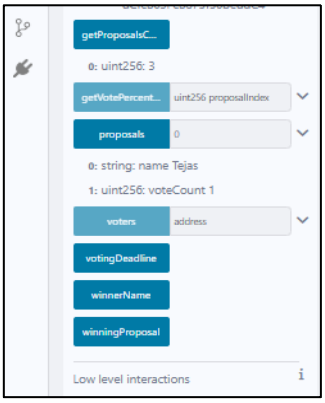
➢ Granting voting right to account 2 using giveRightToVote function by Chairperson.



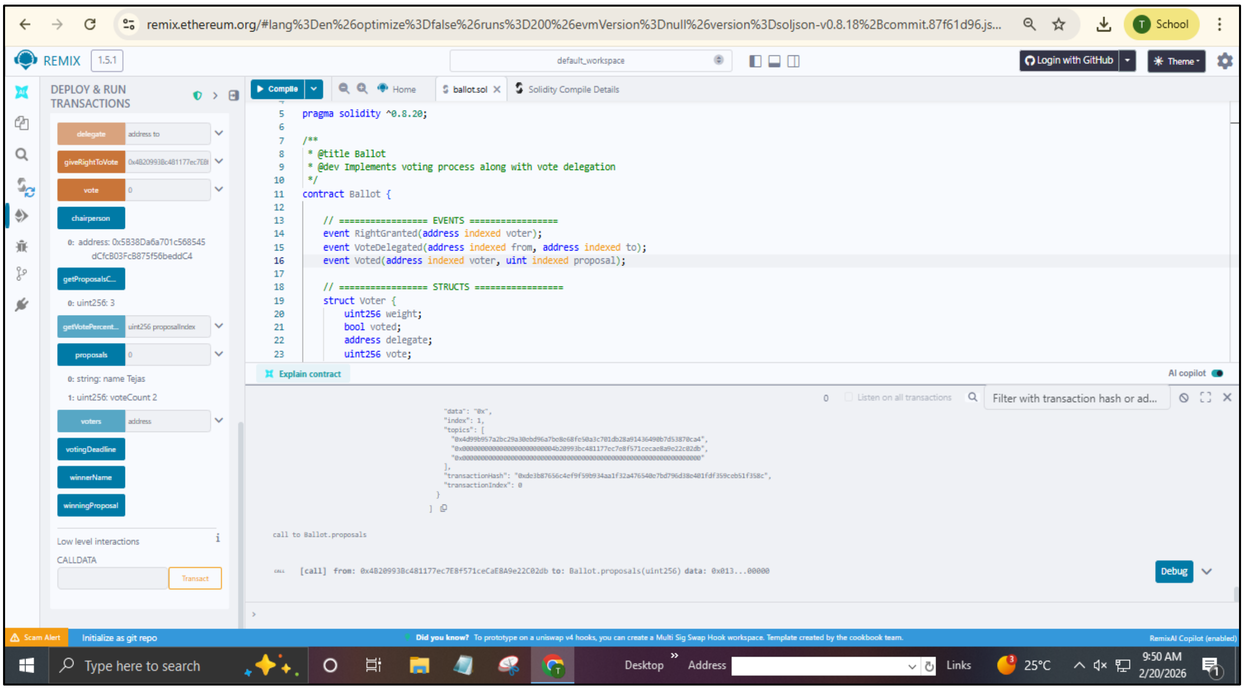
➢ Switching to voter 2 account and Casting vote to proposal 0



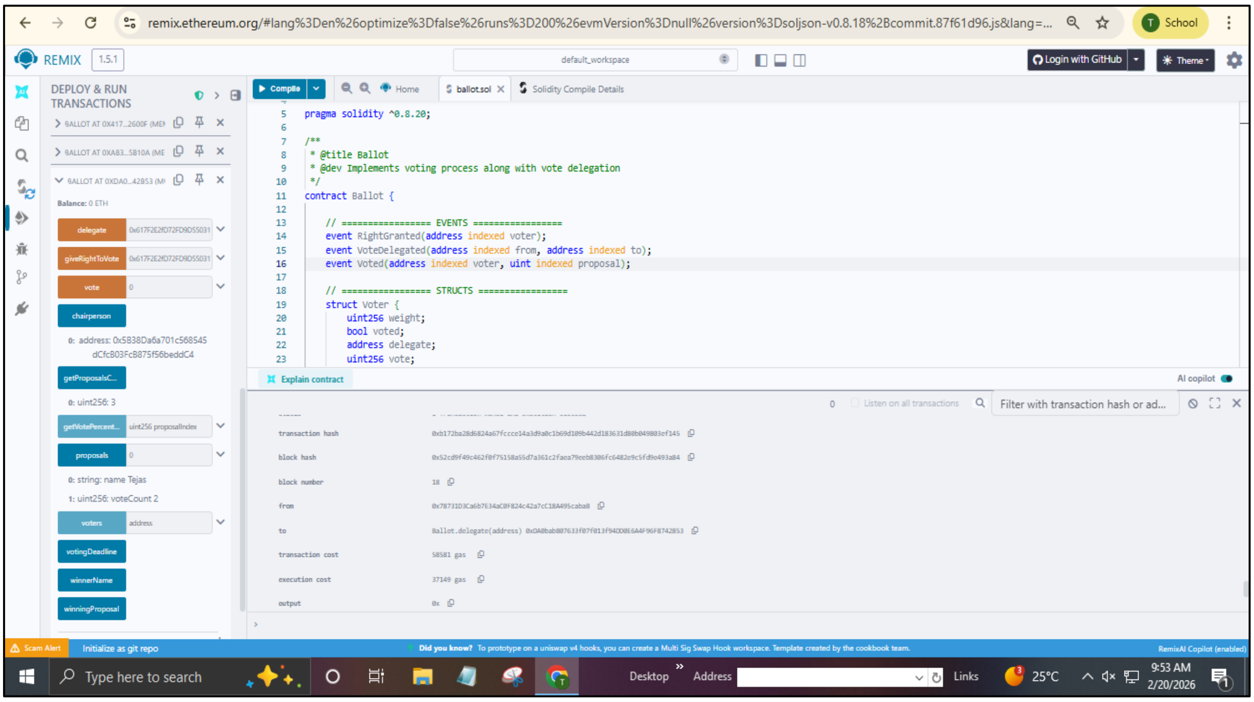
➢ Displaying updated vote count for proposal 0 after successful voting



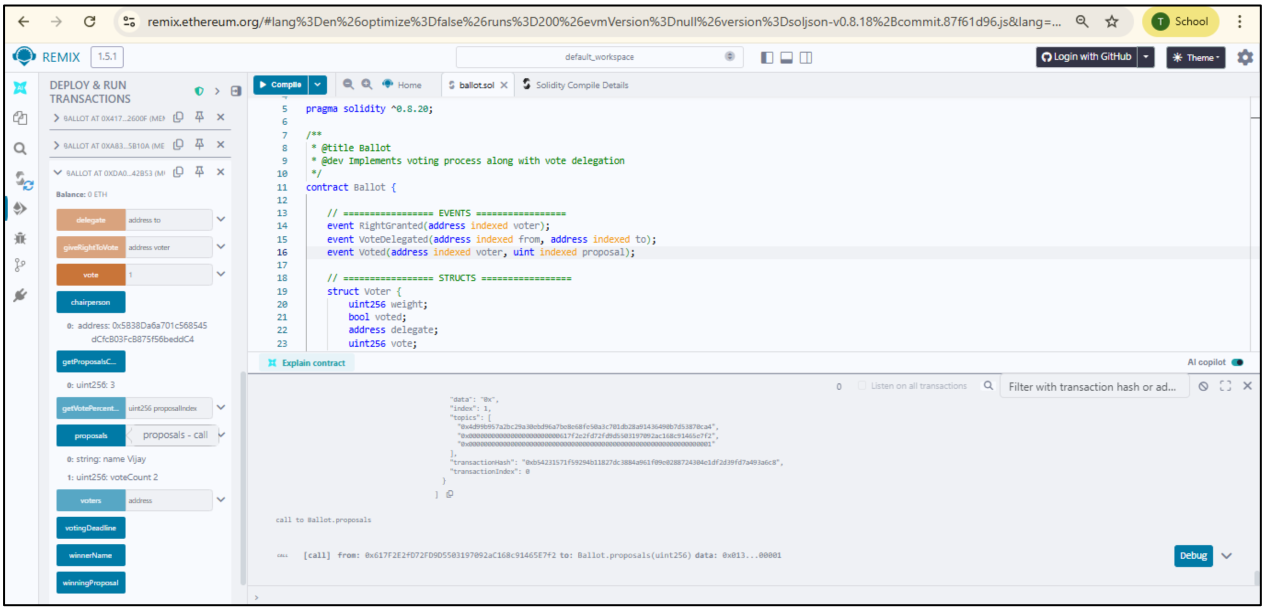
➢ Account 3 receives voting right from chairperson and casts vote for proposal 0, increasing its vote count to 2



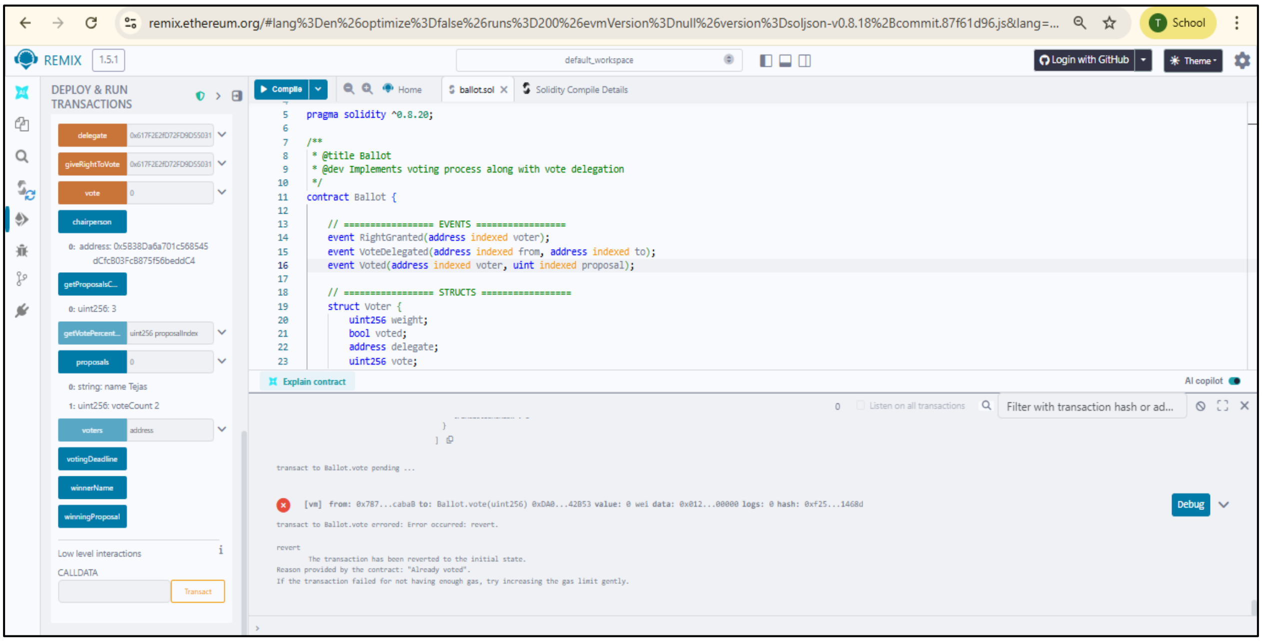
➢ Account 4 delegating voting power to account 5 using the delegate function



➢ Account 5 voting on behalf of account 4 for proposal 1, increasing its vote count to 2

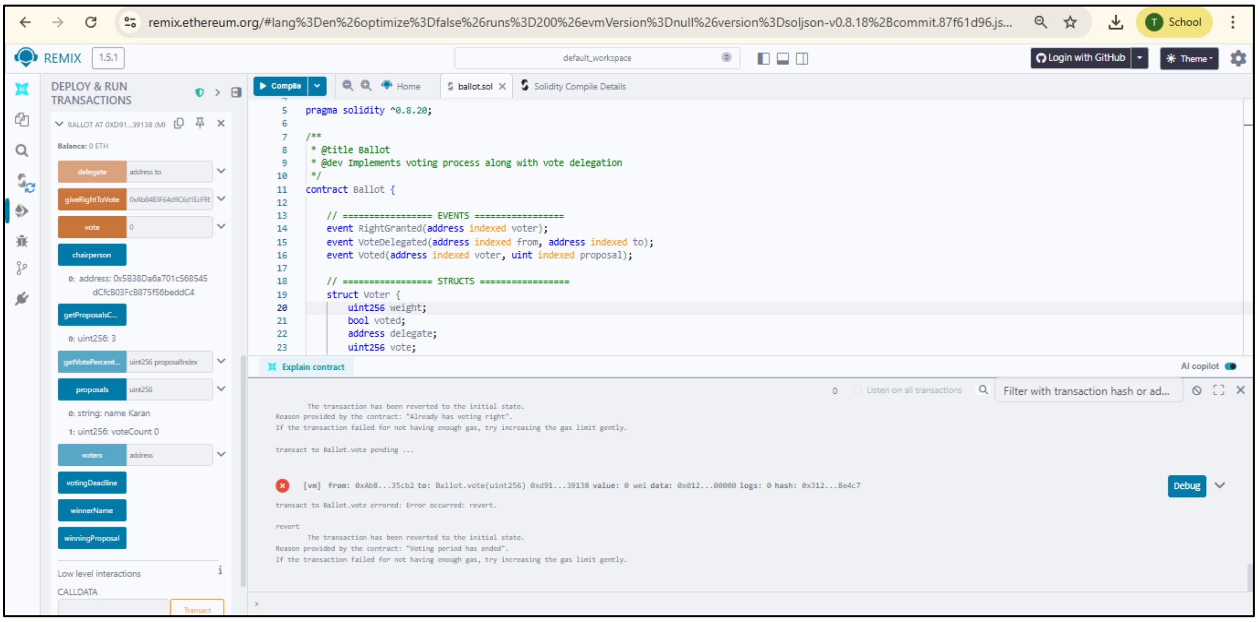


➢ Account 5 attempting to vote again but transaction fails with "Already voted" error message

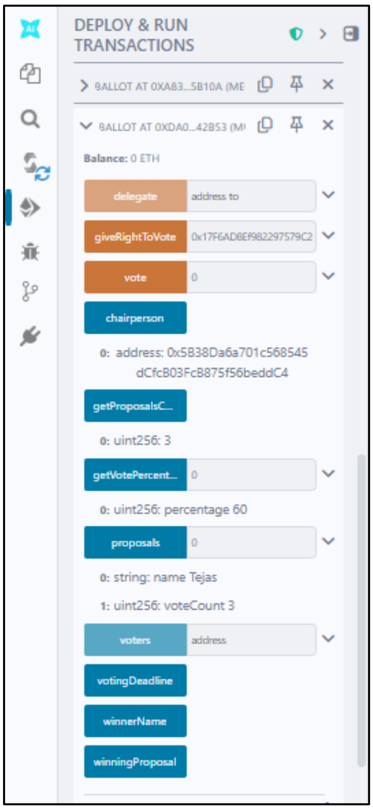


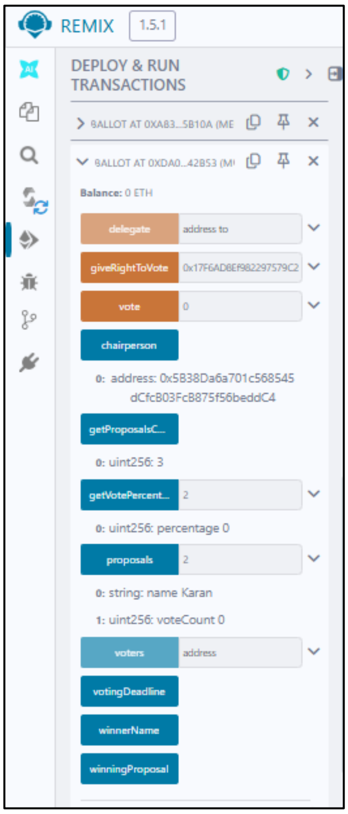
➢ Similarly Account 6 voted for proposal 0 updating its count to 3

➢ Account attempting to vote after deadline but transaction fails with "Voting period has ended" error message



➢ Displaying final vote counts and vote percentage for each proposal





➢ Final results showing winning proposal

Conclusion :

This practical successfully demonstrates the deployment of a decentralized Ballot smart contract using Solidity. The contract ensures secure and transparent voting with features like vote delegation, deadline restriction, and automatic winner calculation. The use of require statements enforces strict validation rules to maintain fairness and prevent misuse. Concepts such as mapping, storage, and memory were effectively applied for efficient data handling. Events were used to improve transparency of voting activities.

Overall, the system showcases how blockchain technology enables a secure, tamper-proof, and reliable digital voting mechanism.