Shravani Anil Patil
D20A-40

Experiment :04
Blockchain Lab

Aim: Hands on Solidity Programming Assignments for creating Smart Contracts
Theory:

1. Primitive Data Types and Variables in Solidity

Solidity provides several primitive data types that serve as the building blocks for smart contract development:

- **Integer Types (uint, int)**
  Used to store numeric values. uint stores unsigned integers (non-negative), while int stores signed integers. Commonly used sizes include uint256, which is optimized for Ethereum Virtual Machine (EVM).
- **Boolean (bool)**
  Stores logical values true or false, typically used in conditions and decision-making.
- **Address (address)**
  Represents a 20-byte Ethereum account or contract address. It is commonly used for identifying users, transferring Ether, and enforcing ownership.
- **Bytes and String (bytes, string)**
  bytes is used for raw binary data, while string is used for storing textual information.

Variables in Solidity are classified as:

- **State Variables**: Stored permanently on the blockchain.
- **Local Variables**: Declared inside functions and exist temporarily.
- **Global Variables**: Predefined variables like msg.sender, msg.value, and block.timestamp that provide blockchain-related information.

2. Functions and Function Types

Functions define the logic of a smart contract. They can accept inputs, perform computations, and return outputs.

Special function types include:

- **pure functions**: Do not read or modify state variables. They only use input parameters and internal logic.

- **view functions**: Can read state variables but cannot modify them.

Using pure and view functions helps reduce gas costs and improves contract efficiency.

3. Inputs and Outputs to Functions

Solidity functions can take one or more input parameters and return one or more output values. Inputs allow data to be passed into the contract, while outputs return computed results to the caller.

Functions can also use **named return variables**, which enhance code readability and simplify debugging. This feature is especially useful when returning complex data structures.

4. Function Visibility, Modifiers, and Constructors

Function Visibility

Solidity provides access control through visibility specifiers:

- **public**: Can be accessed both internally and externally.
- **private**: Accessible only within the same contract.
- **internal**: Accessible within the contract and derived contracts.
- **external**: Can only be called from outside the contract.

Modifiers

Modifiers are reusable pieces of code used to alter function behavior. They are commonly used to enforce conditions such as ownership or authorization (e.g., onlyOwner).

Constructors

A constructor is a special function executed only once during contract deployment. It is typically used to initialize state variables, such as assigning the deployer as the contract owner.

5. Control Flow Statements

Solidity supports standard control flow mechanisms:

- **Conditional Statements (if-else)**
  Used to execute code based on conditions, such as validating balances before transactions.

- **Loops (for, while, do-while)**
  Used to repeat execution of code blocks. However, excessive looping increases gas consumption, so loops must be used carefully in smart contracts.

6. Data Structures in Solidity

Solidity provides several data structures for efficient data management:

- **Arrays**
  Store ordered collections of elements. Arrays can be fixed-size or dynamic.
- **Mappings**
  Store key-value pairs for fast lookups, such as mapping(address => uint) for account balances. Mappings do not support iteration.
- **Structs**
  Group related variables into a single custom data type, improving code organization.
- **Enums**
  Define a set of named constants, enhancing code readability and reducing logical errors.

7. Data Locations

Solidity uses three data locations to manage variable storage:

- **storage**: Permanent data stored on the blockchain (state variables).
- **memory**: Temporary data used during function execution.
- **calldata**: Read-only and gas-efficient location used for external function parameters.

Choosing the correct data location is crucial for optimizing gas usage and contract performance.

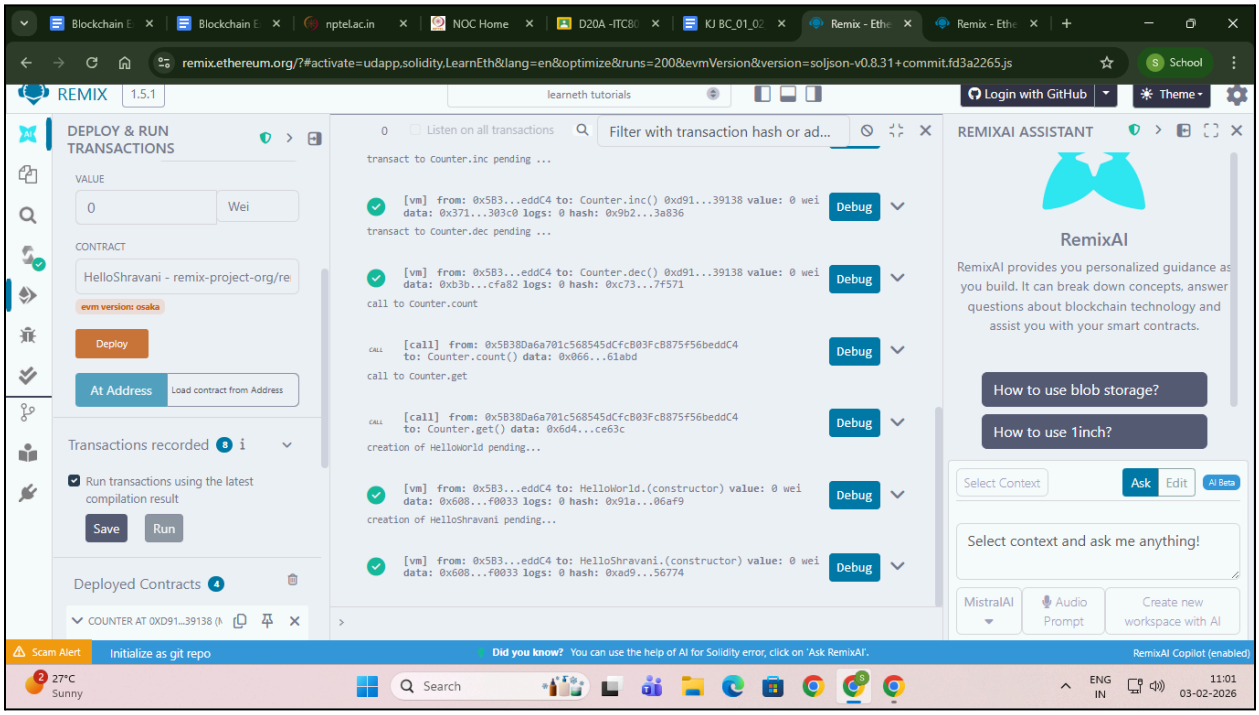
8. Transactions, Ether, and Gas

- **Ether and Wei**
  Ether is the native cryptocurrency of Ethereum. All values are internally handled in Wei, where
  1 Ether = 10^{18} Wei, allowing precise financial calculations.
- **Gas and Gas Price**
  Gas measures the computational effort required to execute transactions or smart contracts. The gas price determines how much Ether is paid per unit of gas.
- **Sending Transactions**
  Transactions are used to transfer Ether or interact with smart contracts. Methods like transfer(), send(), and call() are used, with call() offering greater flexibility and control.

Efficient contract design minimizes gas consumption and ensures secure execution.

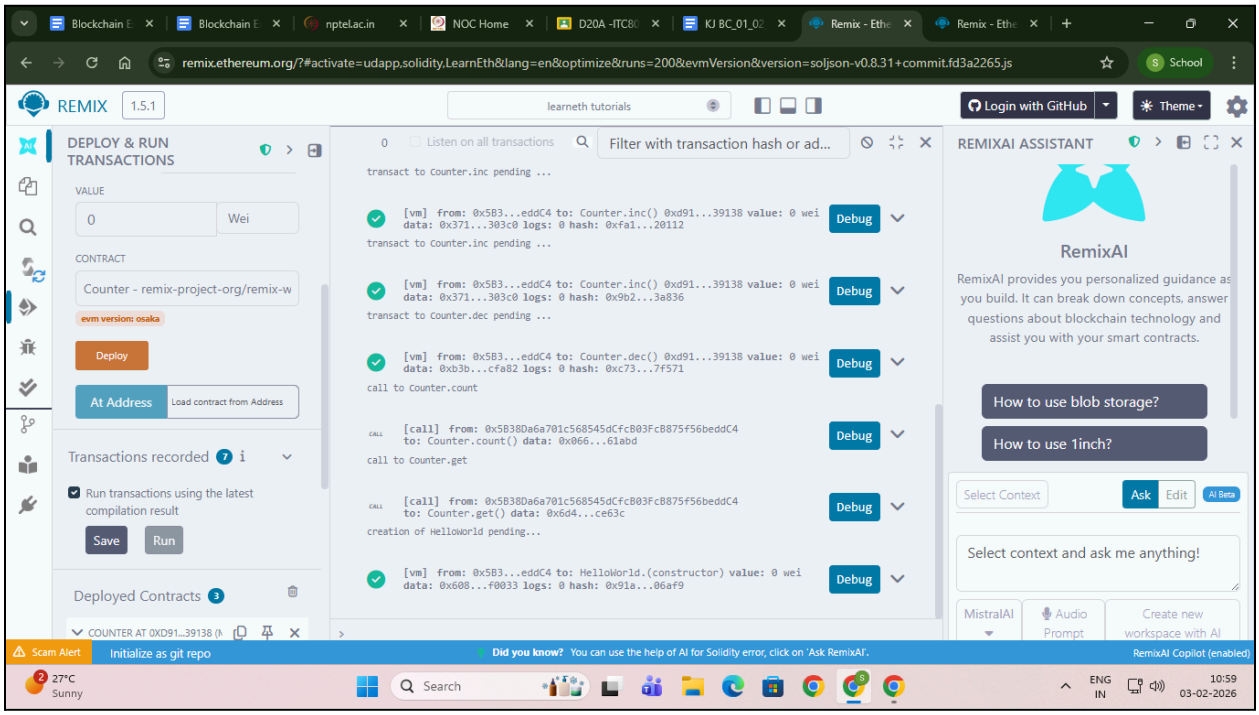
1. basic.sol

```
2   //Shravani Patil - 40
3   // compiler version must be greater than or equal to 0.
4   pragma solidity ^0.8.3;
5
6   contract HelloWorld {
7       string public greet = "Hello World!";
8   }
9
10  contract HelloShravani {
11      string public student ="Shravani Patil-40";
12  }
```

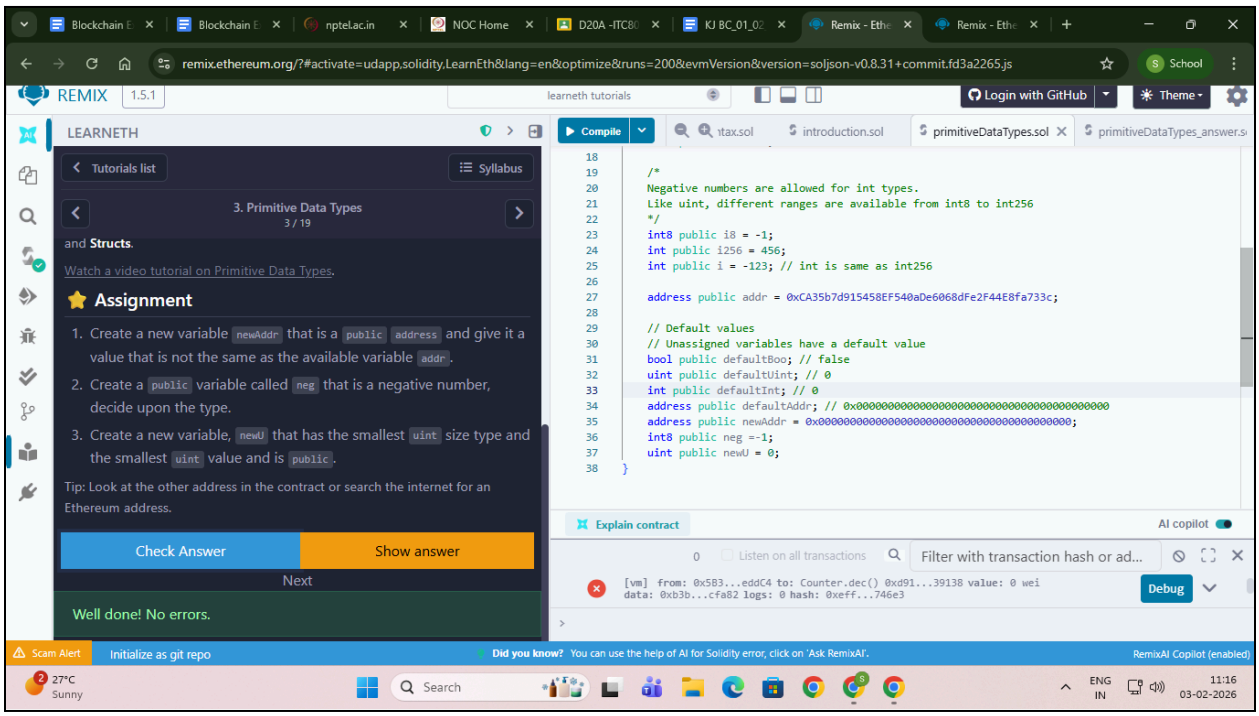


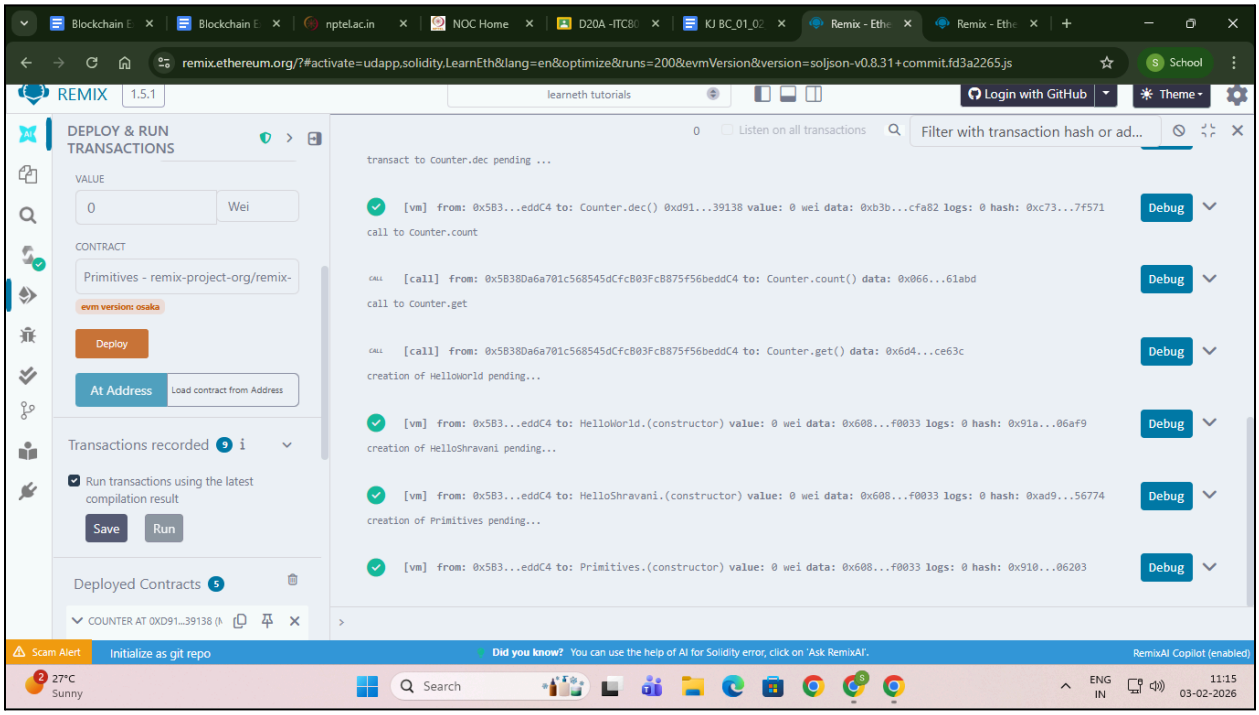
2. intoduction.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
// SPDX-License-Identifier: MIT
//Shravani Patil - 40
contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

    // Function to increment count by 1
    function inc() public {
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {
        count -= 1;
    }
```

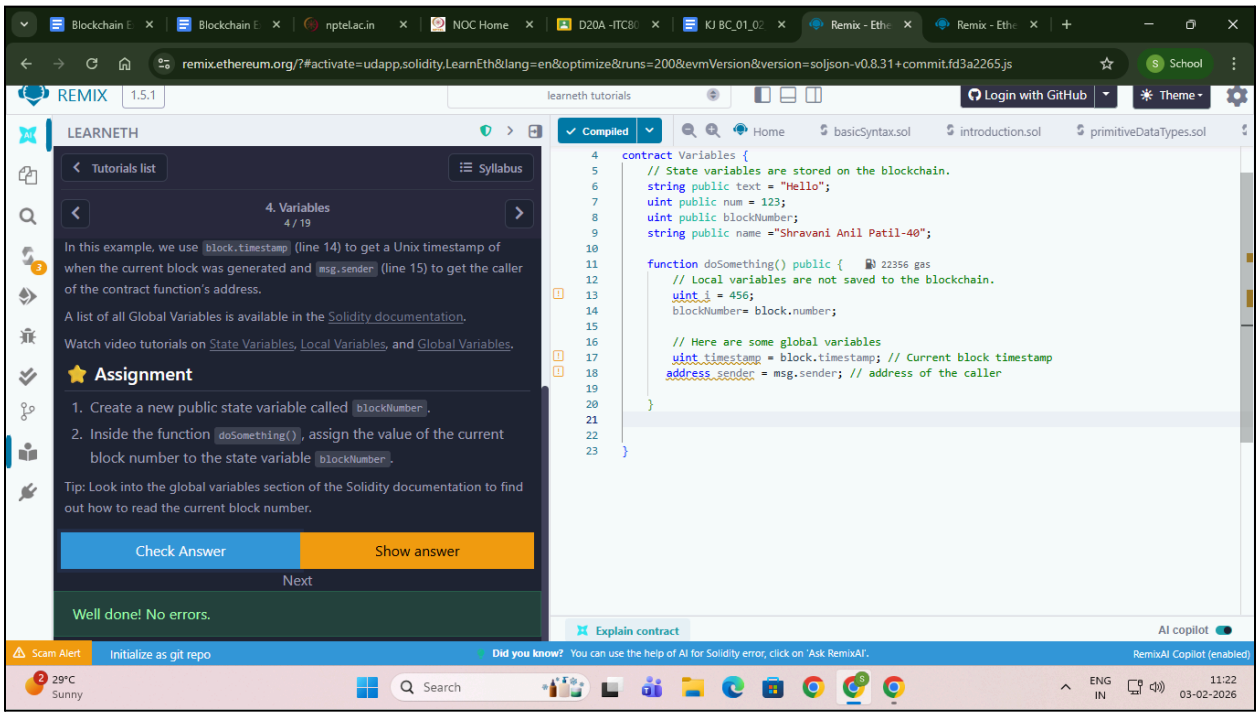


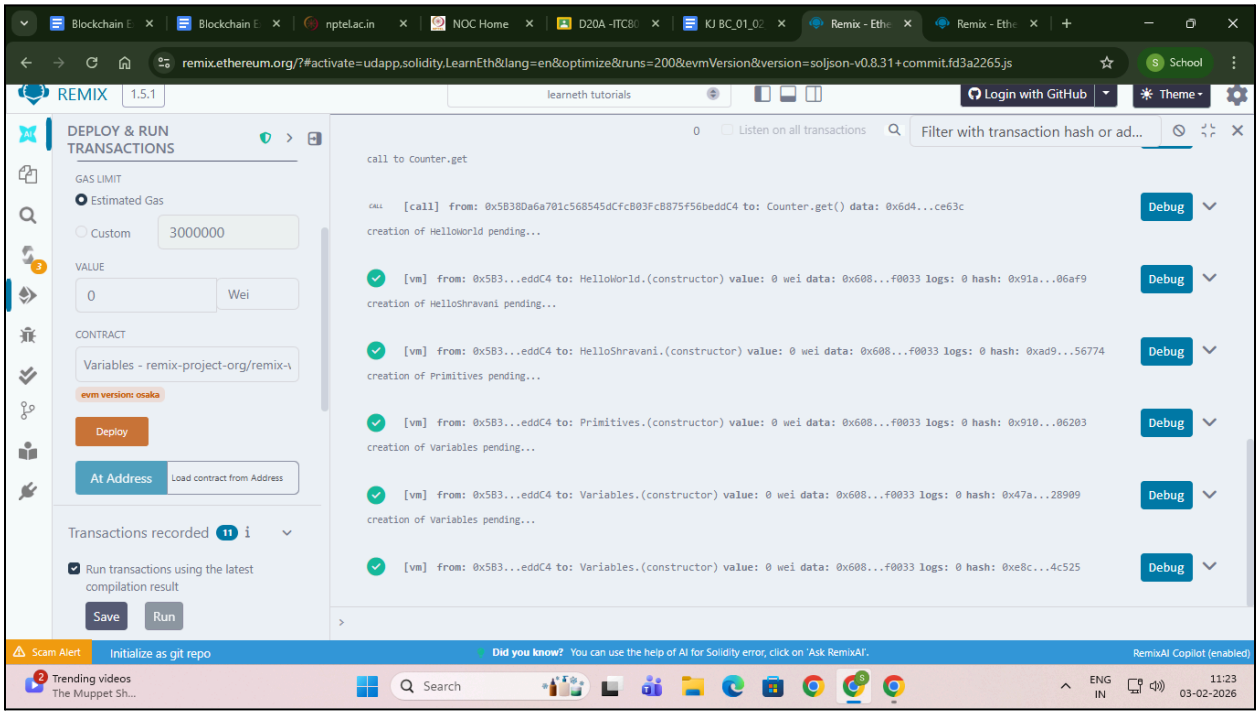
3. PrimitiveDataTypes.col



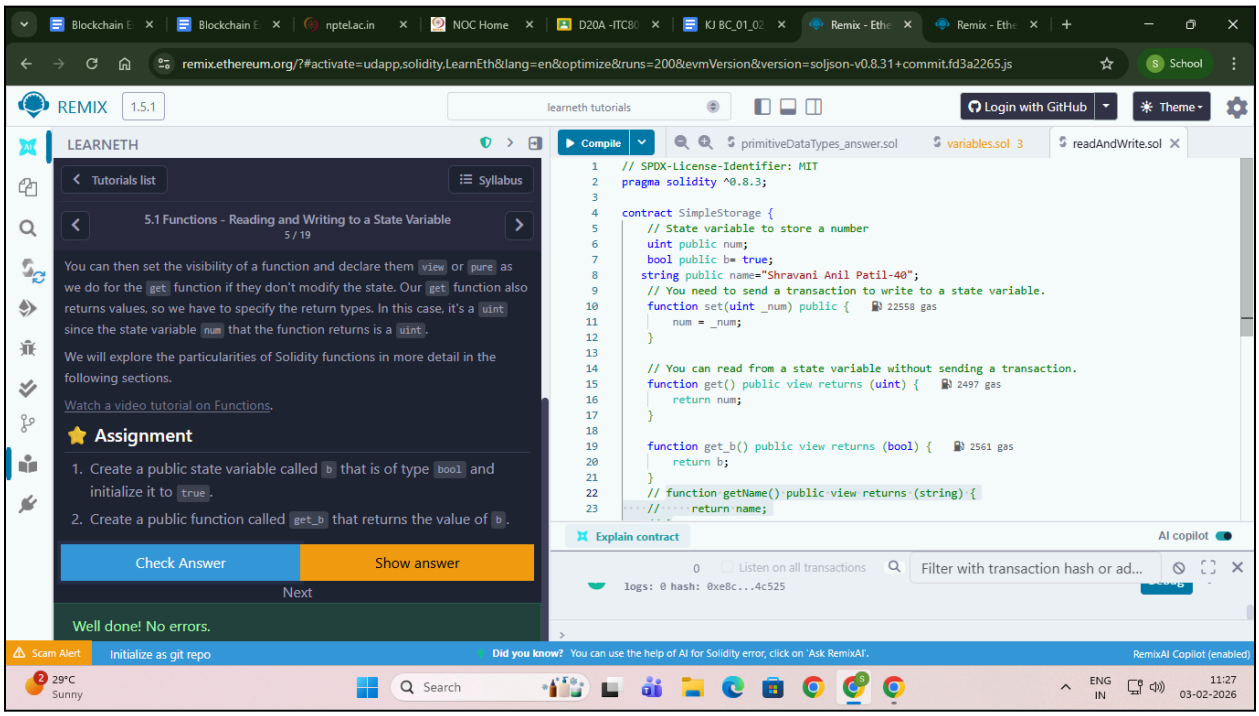


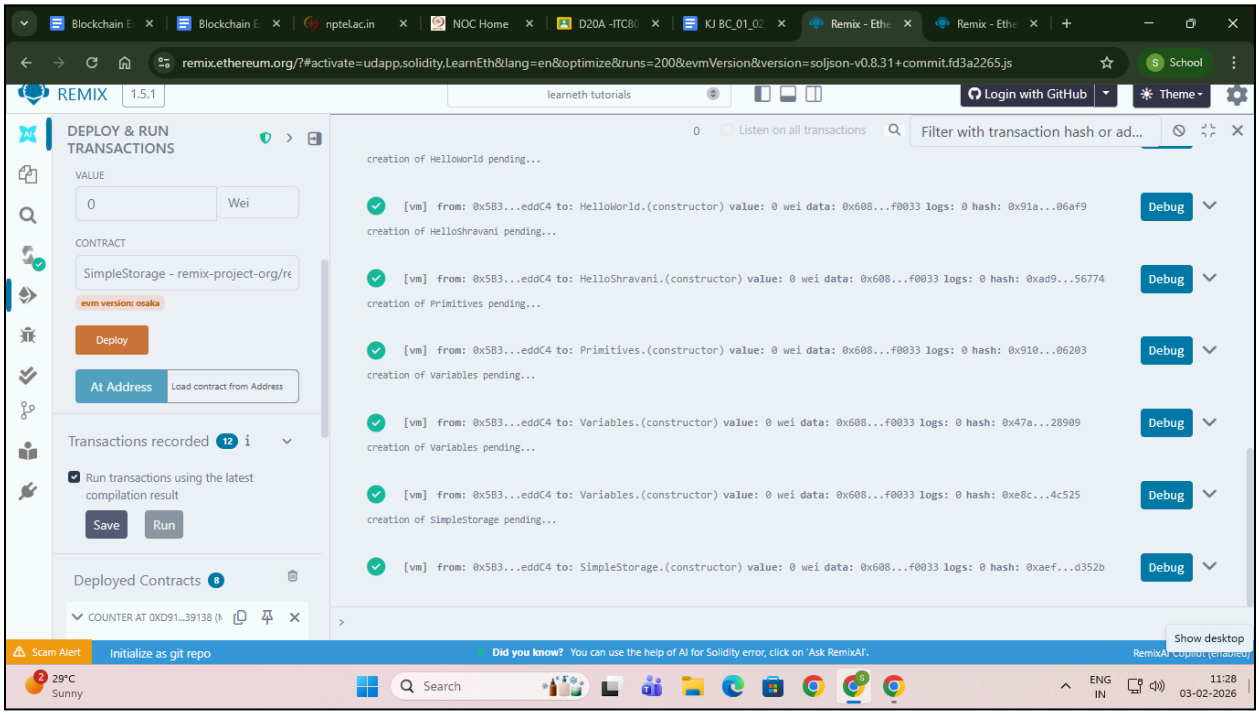
4. Variables.sol



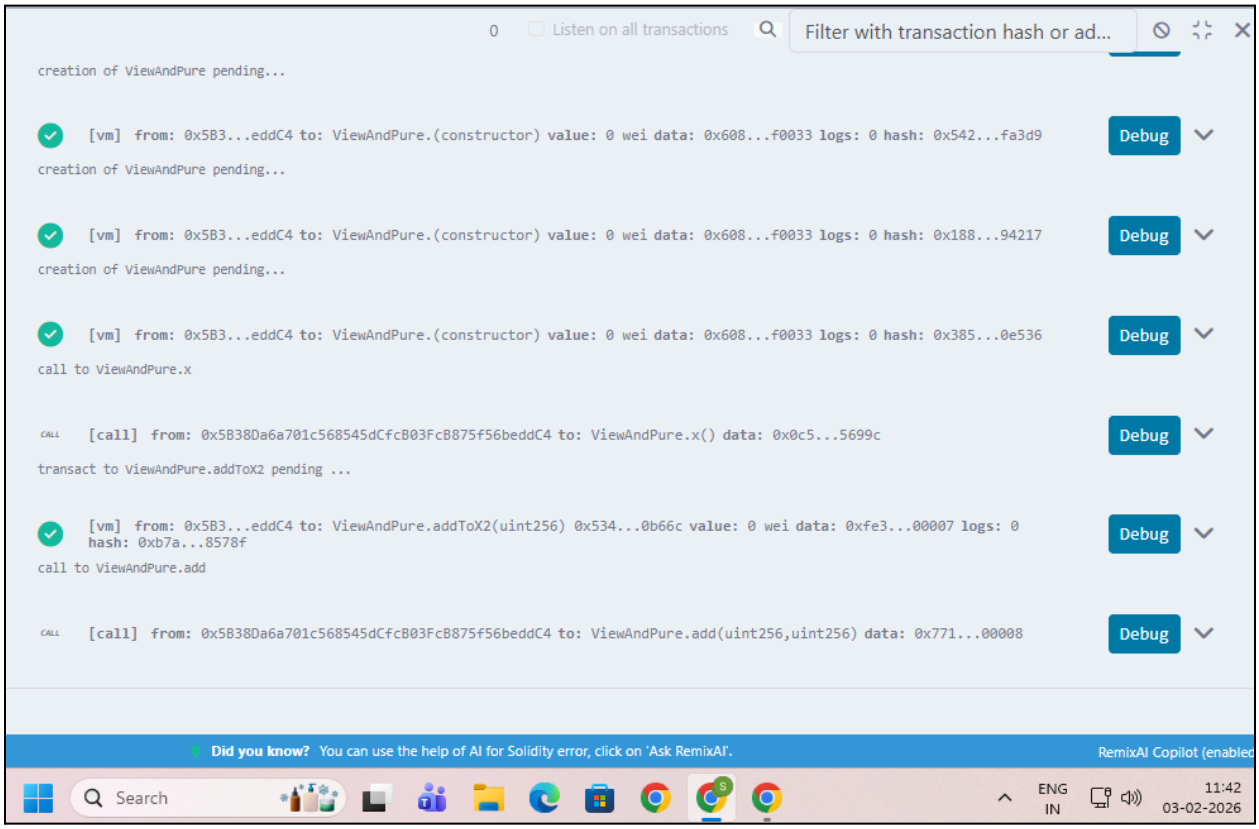


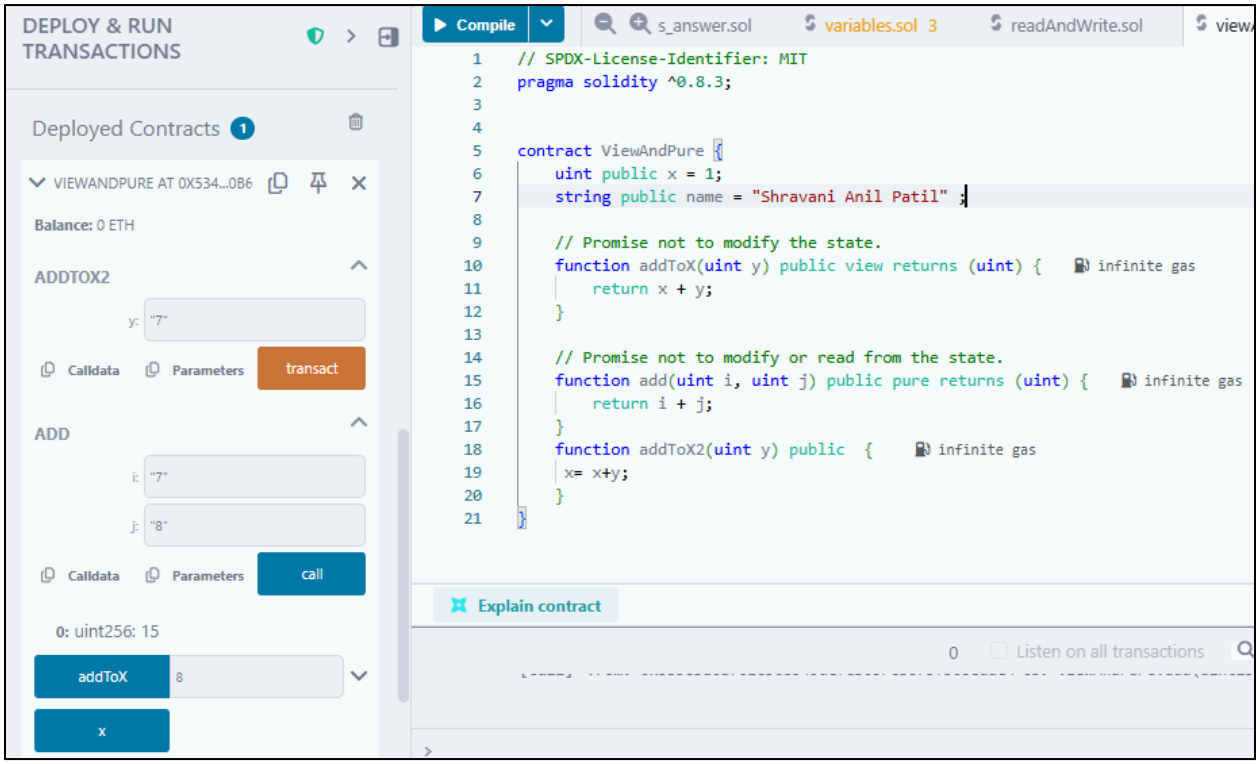
5. readandWrite.sol



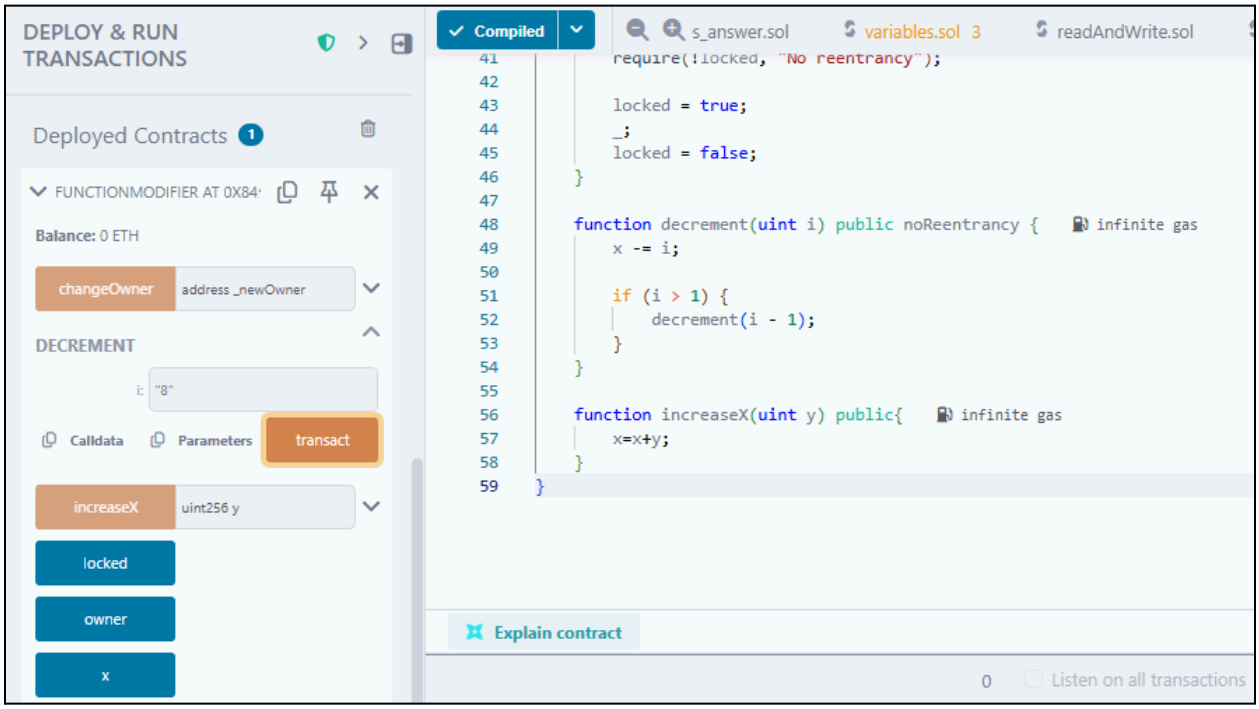


6. **ViewAndPure.sol**

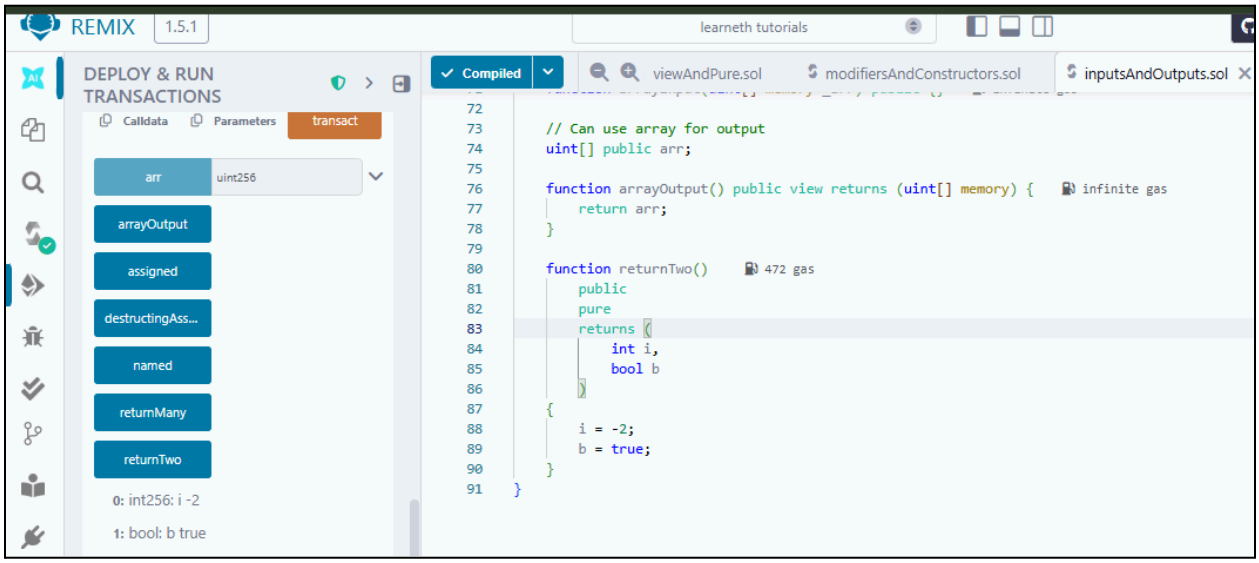


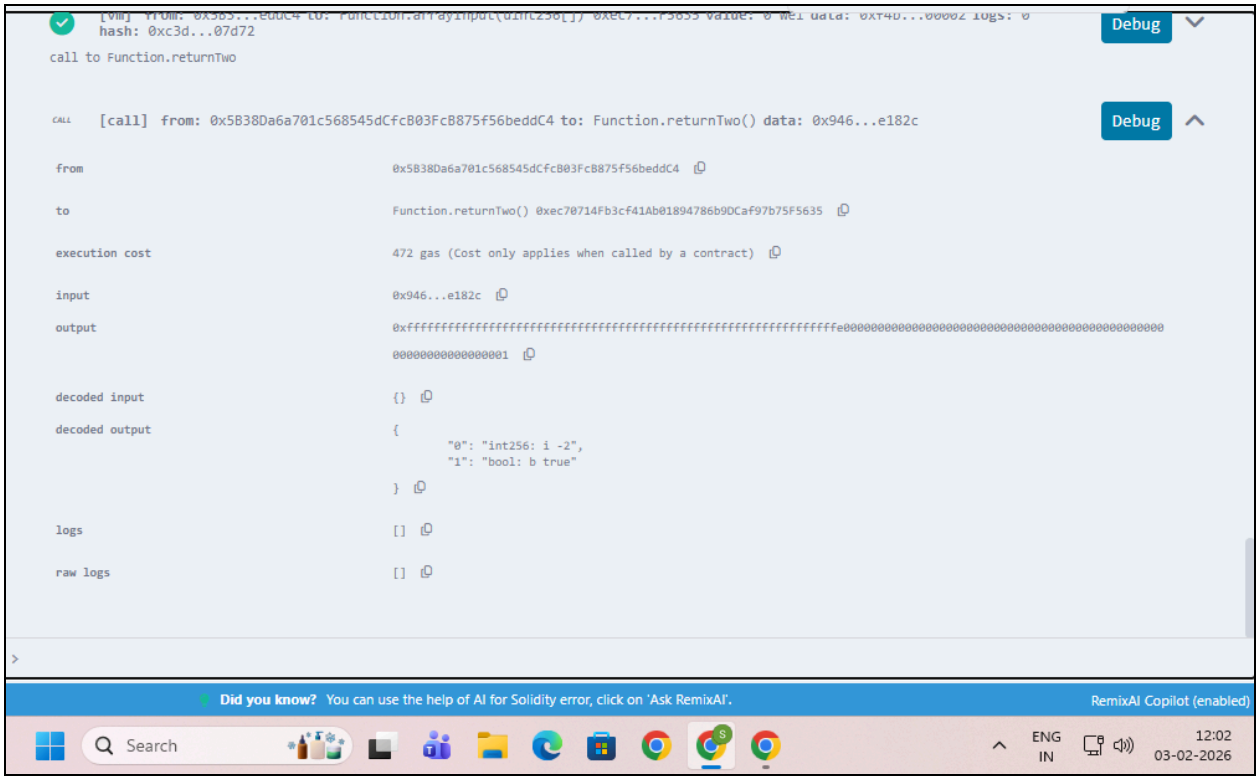


7. ModifiersandConstructors.sol

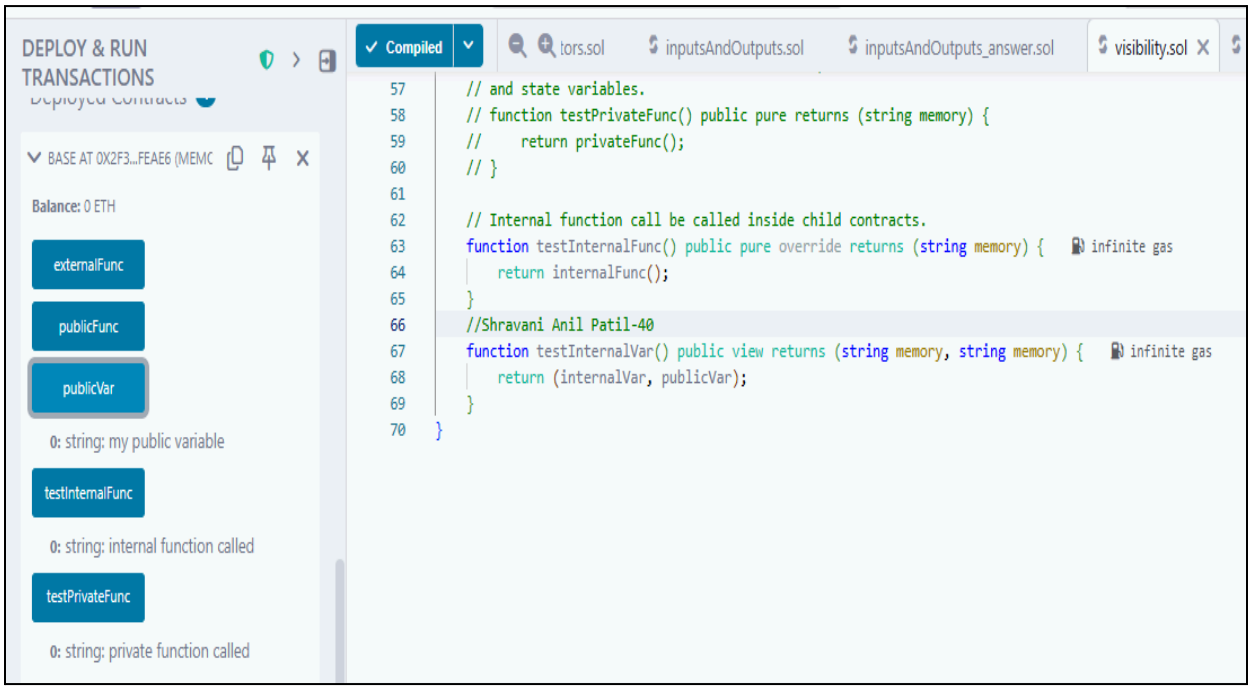


8. inputsandOutputs.sol

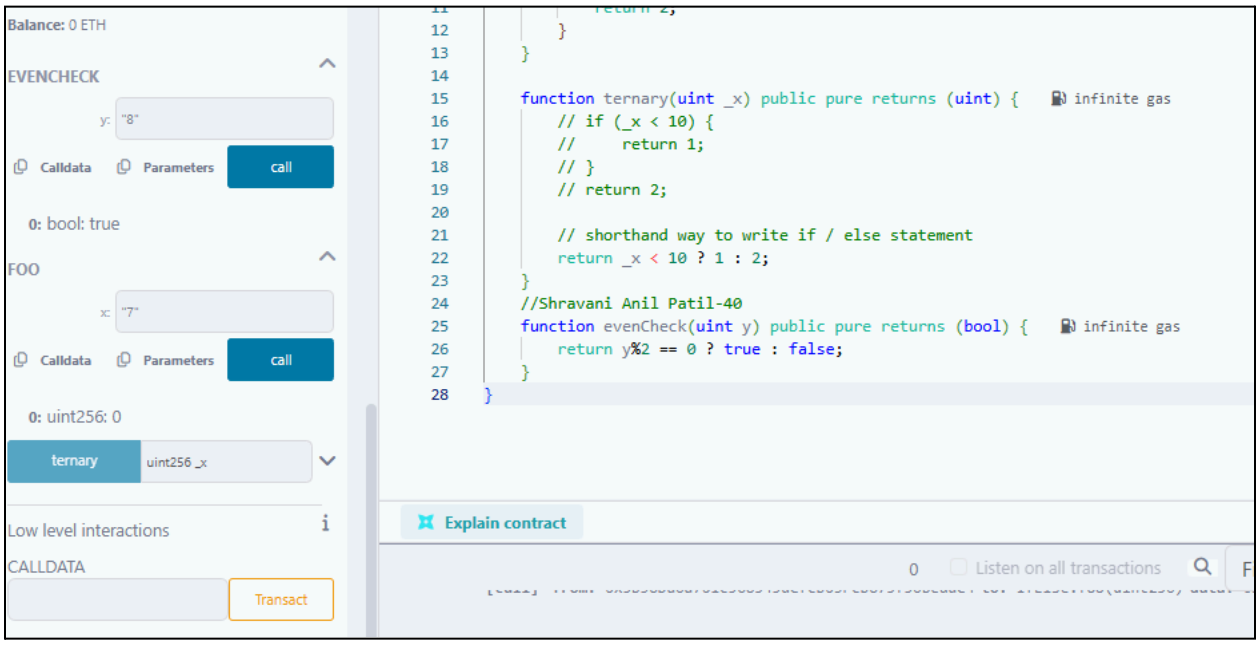




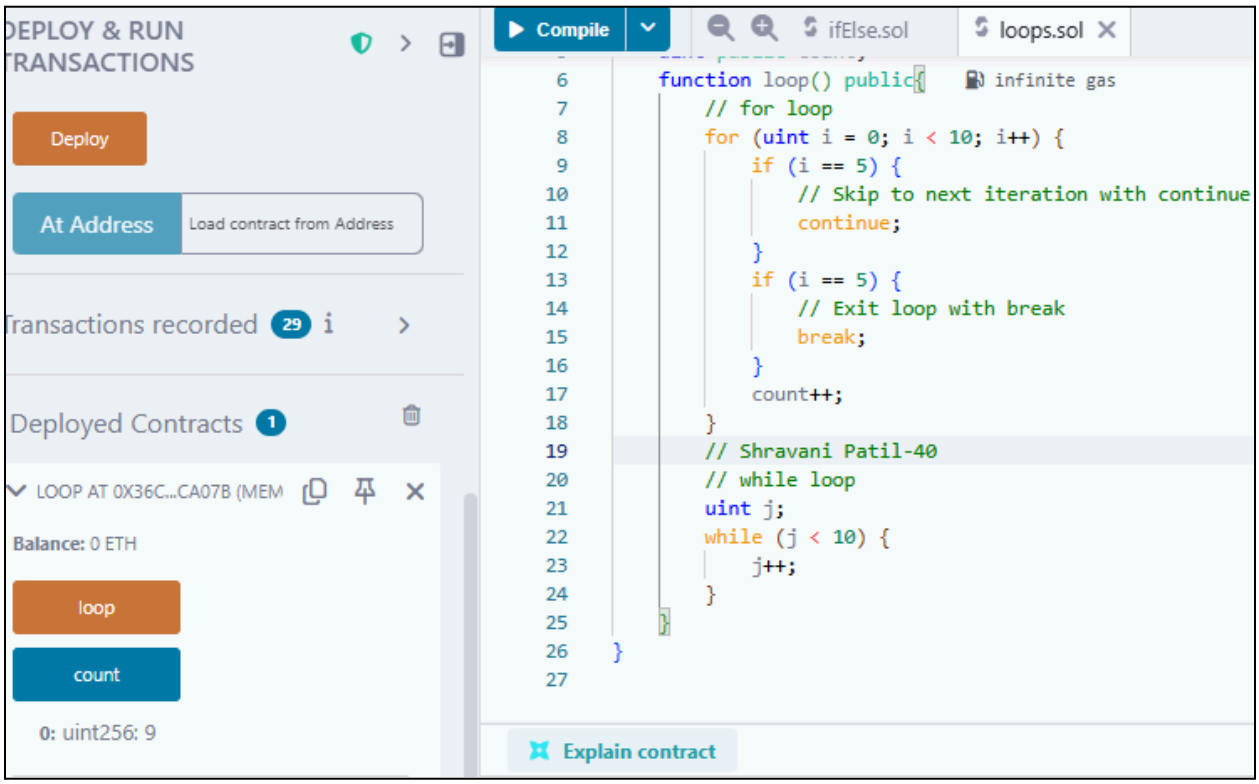
9. Visibility.sol



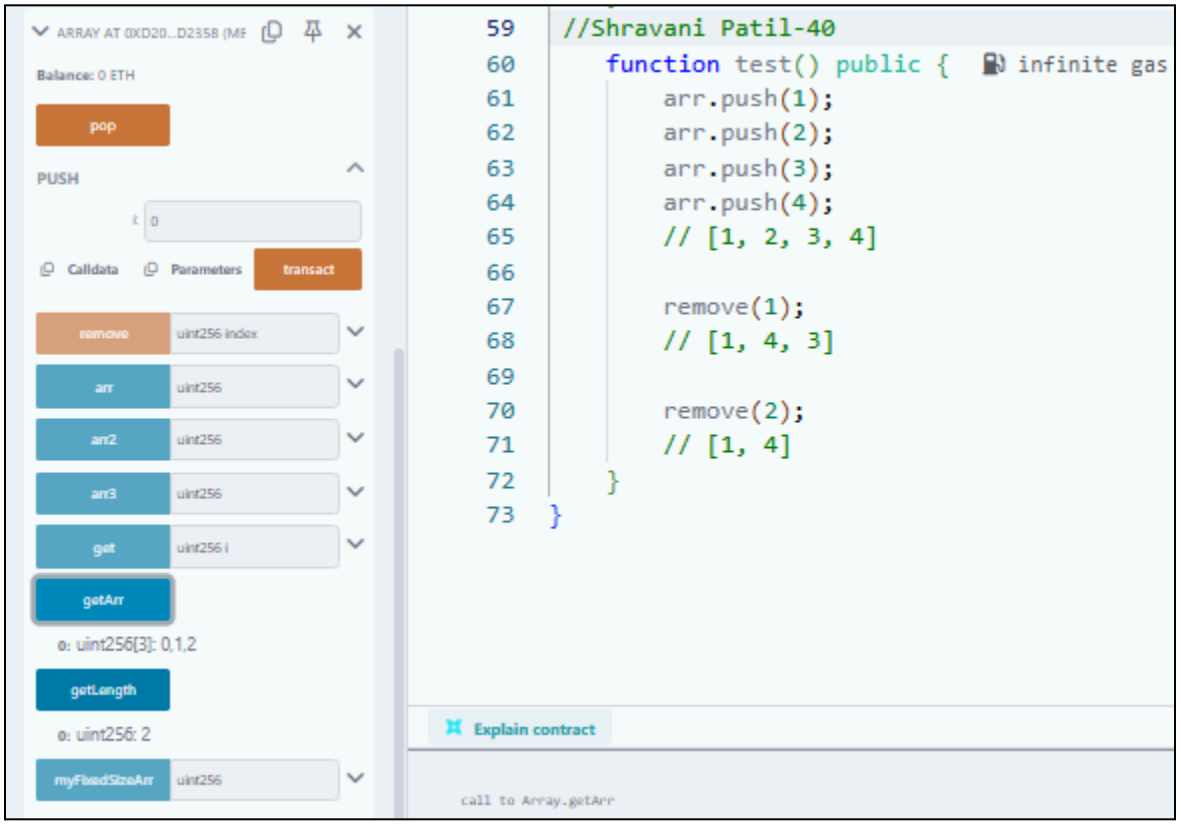
10.IfElse.sol



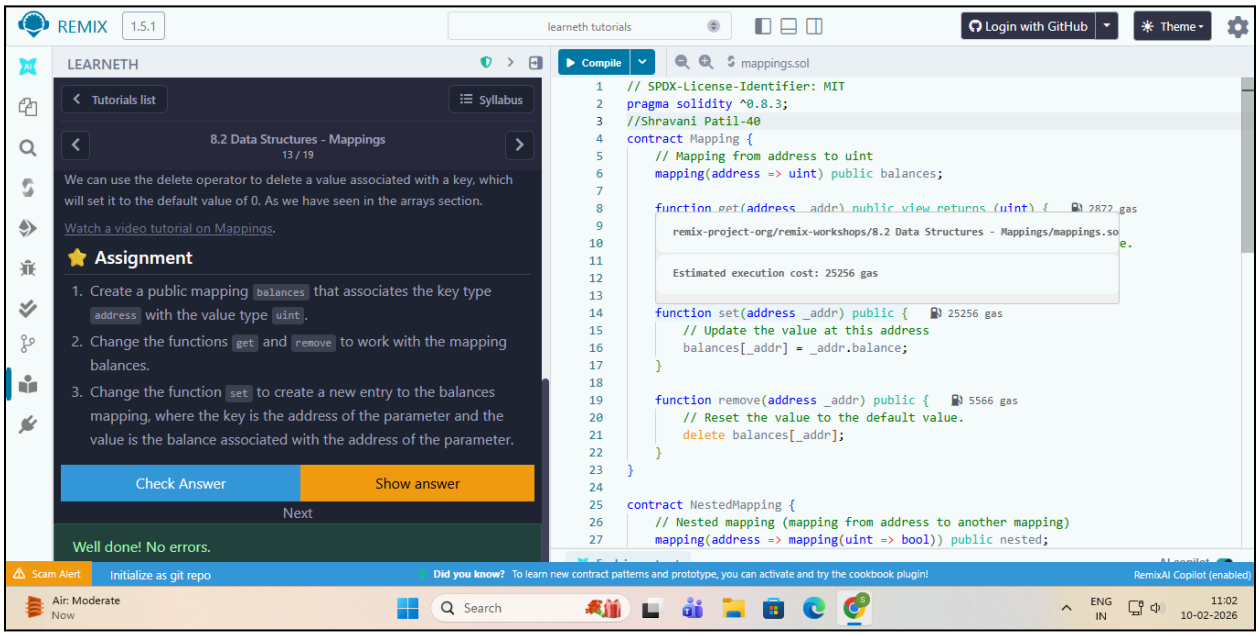
11. Loops.sol



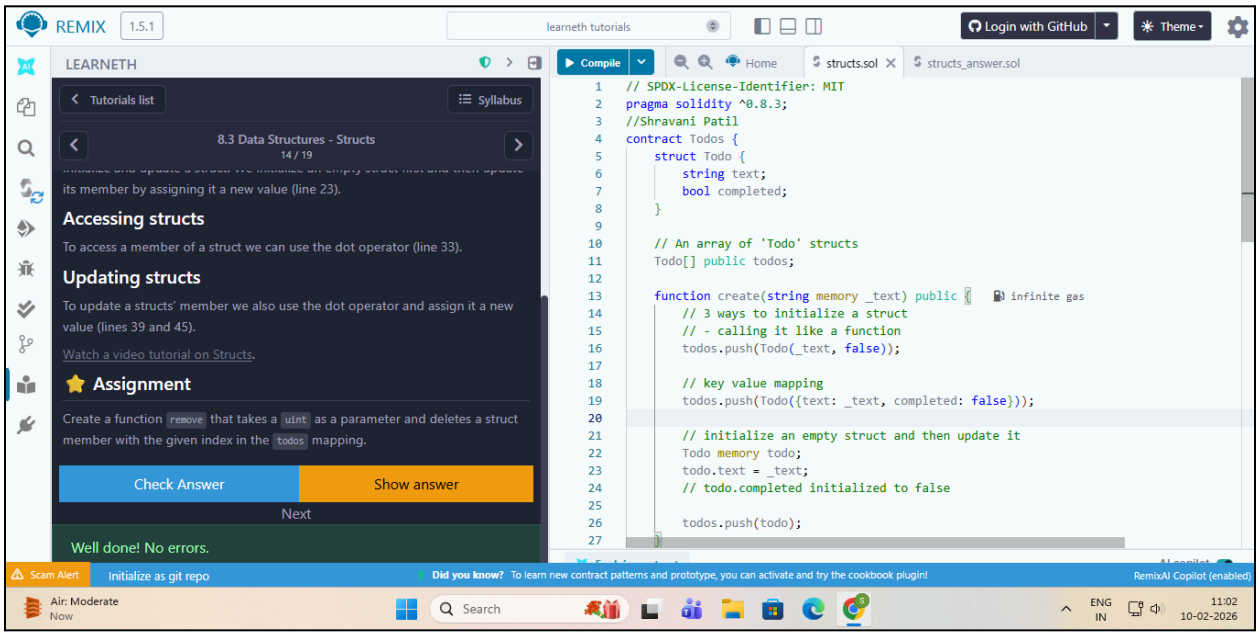
12. Arrays.sol



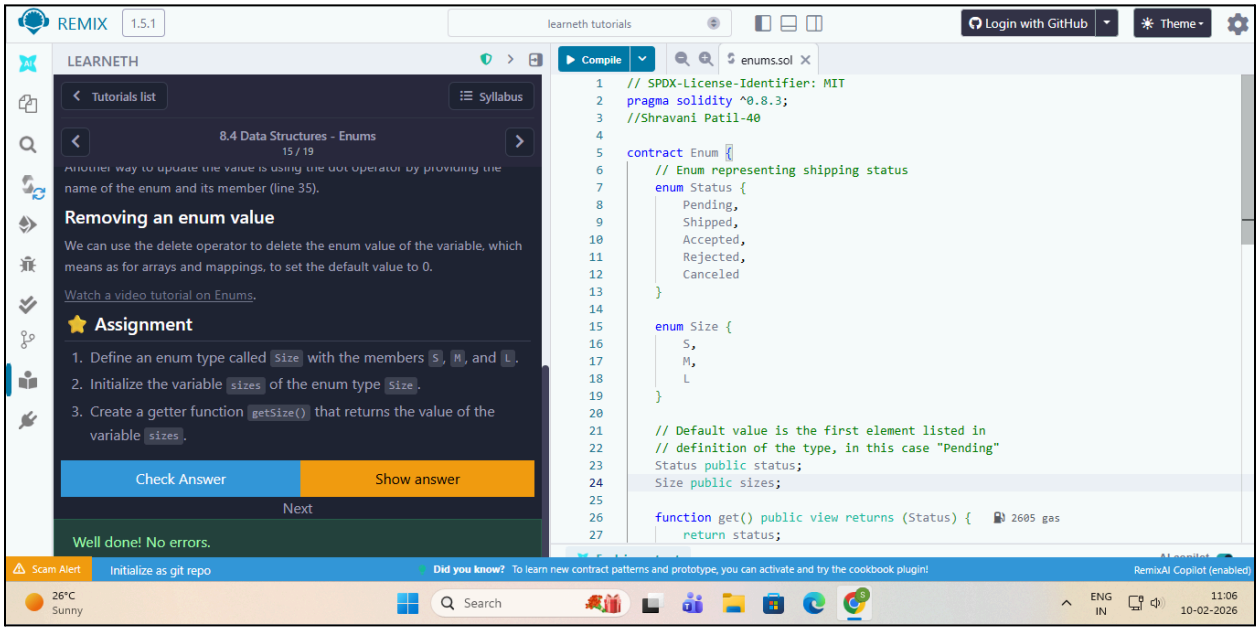
13. Mappings.sol



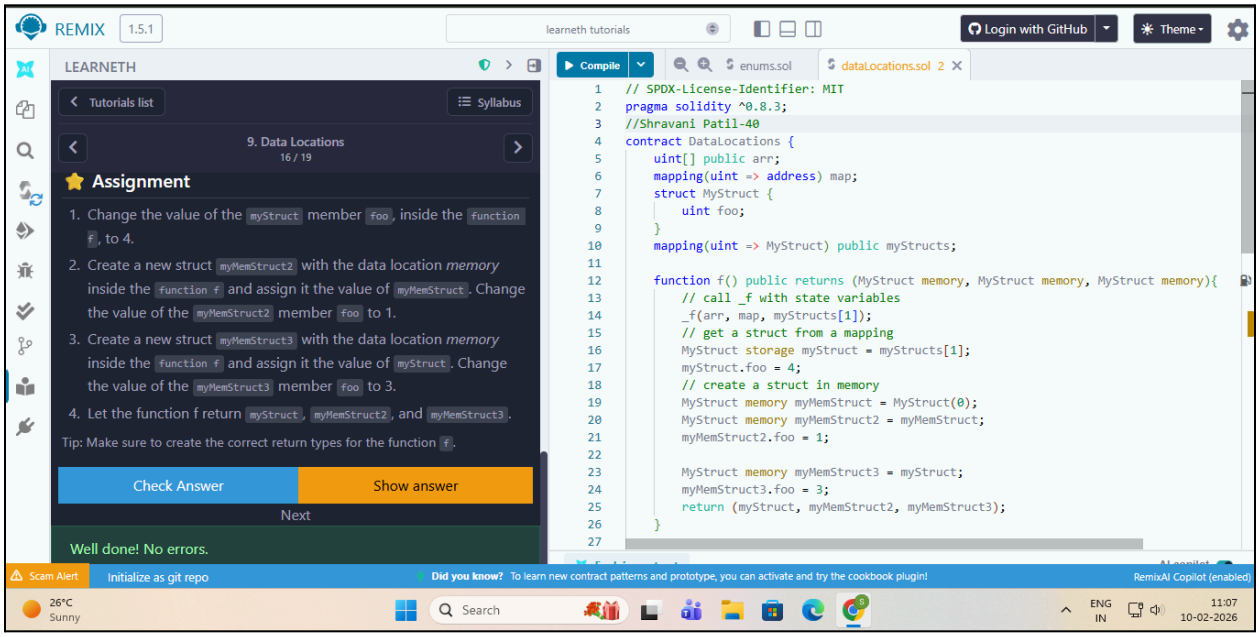
14.Struct.sol



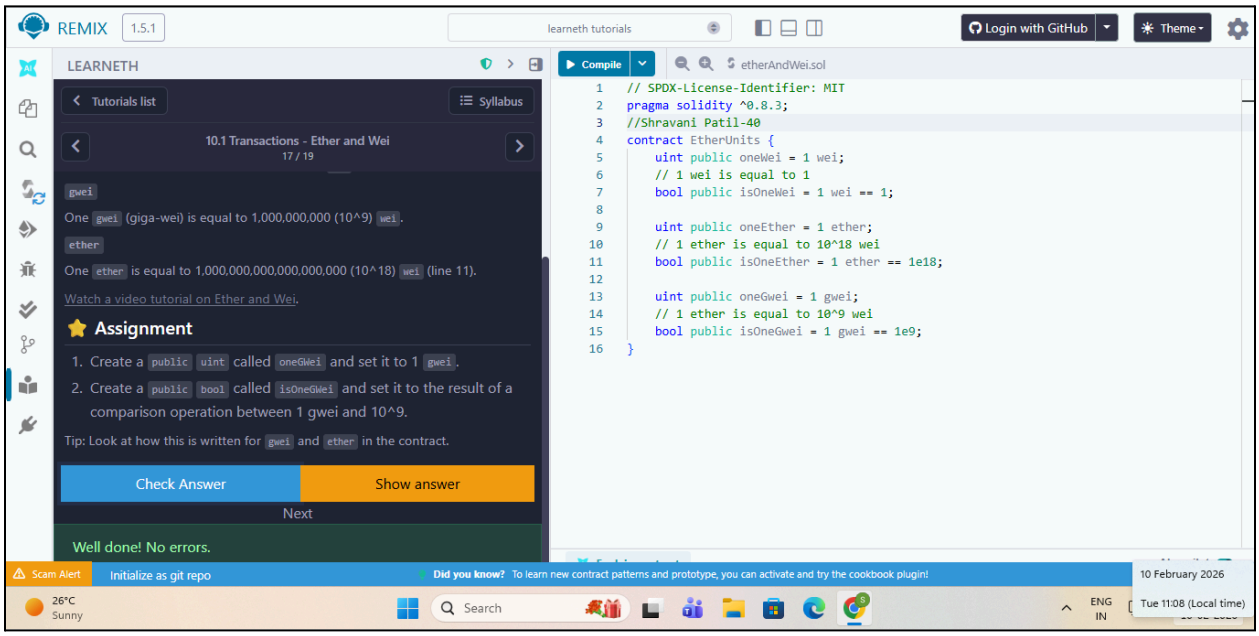
15.enums.sol



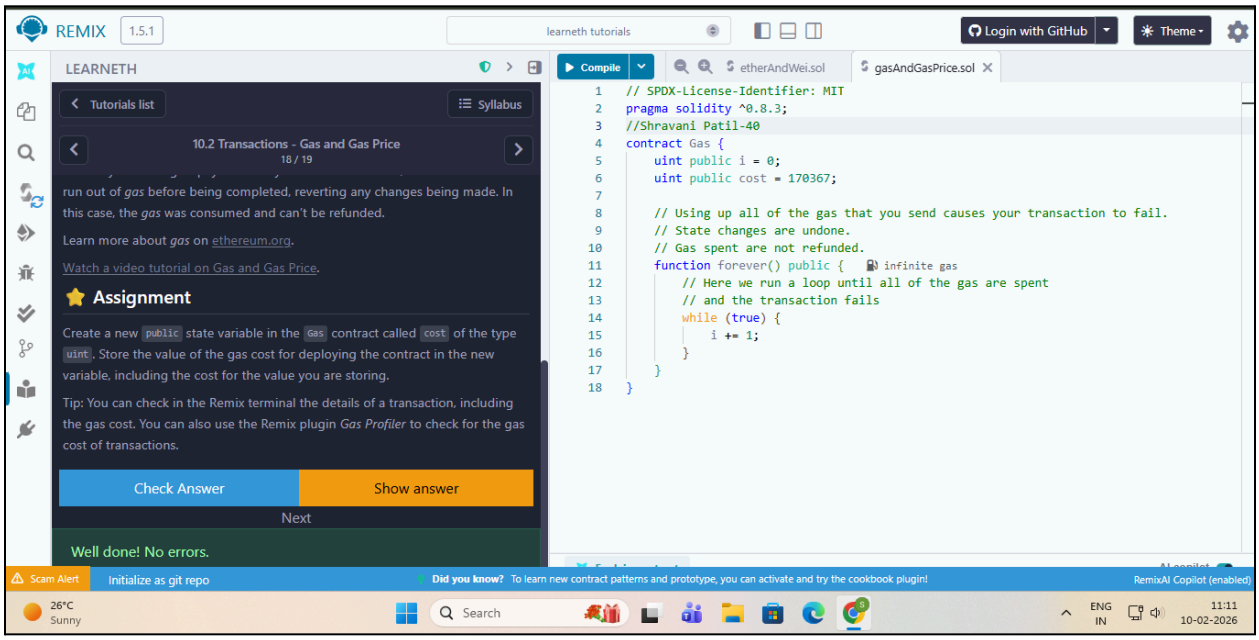
16. dataLocations.sol



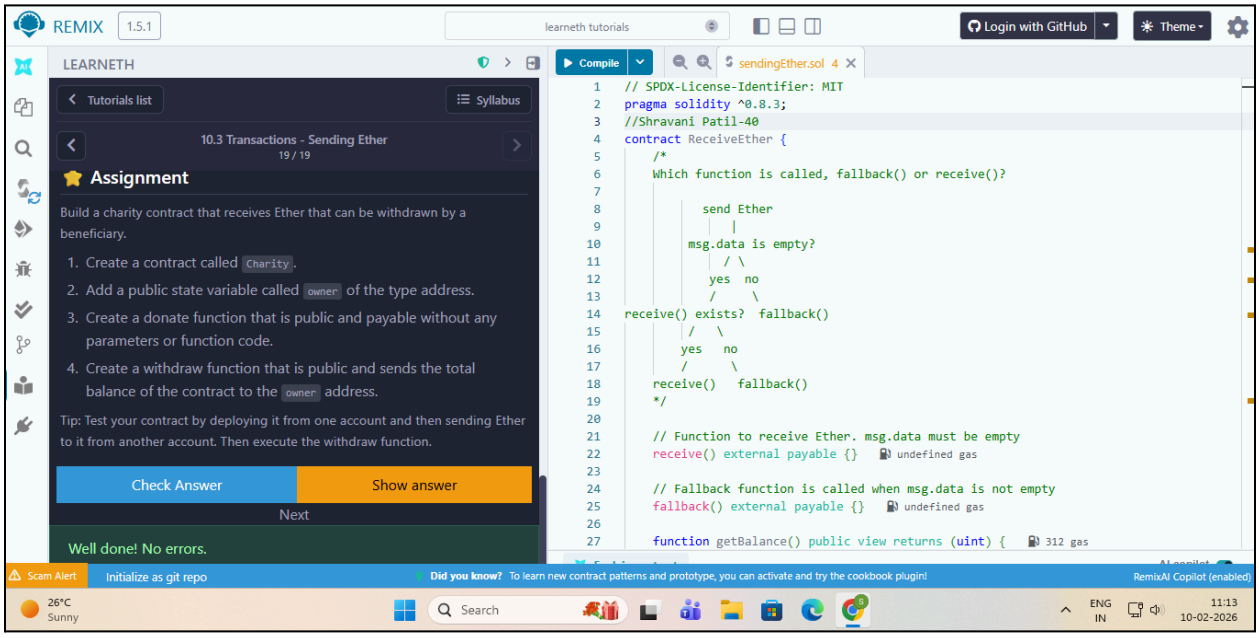
17.etherandWei.sol



18.gasandGasPrice.sol



19. SendingEther.sol



Conclusion:

This experiment provides hands-on experience in Solidity programming by exploring core blockchain concepts, data types, control structures, and transaction handling. Understanding these fundamentals enables the design and deployment of secure, efficient, and reliable smart contracts on the Ethereum blockchain.