Sonam Chhabaidiya
D20A 16

Experiment No. 4

Aim: Hands on Solidity Programming Assignments for creating Smart Contracts

Theory:

1. Primitive Data Types, Variables, Functions – pure, view

In Solidity, primitive data types form the foundation of smart contract development. Commonly used types include:

- **uint / int**: unsigned and signed integers of different sizes (e.g., uint256, int128).
- **bool**: represents logical values (true or false).
- **address**: holds a 20-byte Ethereum account address, often used for storing user accounts or contract addresses.
- **bytes / string**: store binary data or textual data.

Variables in Solidity can be **state variables** (stored on the blockchain permanently), **local variables** (temporary, created during function execution), or **global variables** (special predefined variables such as msg.sender, msg.value, and block.timestamp).

Functions allow execution of contract logic. Special types of functions include:

- **pure**: cannot read or modify blockchain state; they work only with inputs and internal computations.
- **view**: can read state variables but cannot alter them. This classification helps optimize gas usage and enforces function integrity.

2. Inputs and Outputs to Functions

Functions in Solidity can accept input arguments and return one or more output values. Inputs enable users or other contracts to pass data into the contract, while outputs make it possible to return results after computation. For example, a function can accept an amount in Ether and return whether the transfer was successful. Solidity also allows named return variables, which improve readability and debugging.

Sonam Chhabaidiya
D20A 16

Example

```
function add(uint a, uint b) public pure returns (uint sum) {
   sum = a + b;
}

function getValues() public view returns (uint, string memory) {
   return (age, name);
}
```

3. Visibility, Modifiers and Constructors

- **Function Visibility** defines who can access a function:
    - public: available both inside and outside the contract.
    - private: only accessible within the same contract.
    - internal: accessible within the contract and its child contracts.
    - external: can be called only by external accounts or other contracts.

- **Modifiers** are reusable code blocks that change the behavior of functions. They are often used for access control, such as restricting sensitive functions to the contract owner (onlyOwner).

```
modifier onlyOwner() {
require(msg.sender == owner, "Not owner");
_;
}
function restricted() public onlyOwner {
// restricted logic
}
```

- **Constructors** are special functions executed only once during contract deployment. They initialize important values, such as setting the deploying account as the owner of the contract.

Sonam Chhabaidiya
D20A 16

4. Control Flow: if-else, loops

Control flow in Solidity is similar to traditional programming languages:

- **if-else** allows conditional decision-making in contract logic, e.g., checking if a balance is sufficient before transferring funds.

```
if (condition) {
   // true block
} else if (anotherCondition) {
   // else-if block
} else {
   // false block
}
```

- **Loops** (for, while, do-while) enable repeated execution of code. For example, iterating through an array of users. However, loops must be used carefully, as excessive iterations increase gas consumption, potentially making the contract expensive to execute.

```
for (uint i = 0; i < 10; i++) {
count++;
}
```

5. Data Structures: Arrays, Mappings, Structs, Enums

- **Arrays**: Can be fixed or dynamic and are used to store ordered lists of elements. Example: an array of addresses for registered users.
- **Mappings**: Key-value pairs that allow quick lookups. Example: mapping(address => uint) for storing balances. Unlike arrays, mappings do not support iteration.
- **Structs**: Allow grouping of related properties into a single data type, such as creating a struct Player {string name; uint score;}.
- **Enums**: Used to define a set of predefined constants, making code more readable. Example: enum Status { Pending, Active, Closed }.

Sonam Chhabaidiya
D20A 16

6. Data Locations

Solidity provides three main data locations for reference types such as arrays, structs, mappings, and strings:

1. Storage

- Permanent blockchain storage
- Persistent and expensive
- Default location for state variables

State variables are always stored in **storage** because they must remain on the blockchain.

2. Memory

- Temporary storage
- Exists only during function execution
- Cheaper than storage

- Used for:
    - Local variables
    - Function arguments
    - Function return values

Memory variables are deleted after the function execution is completed.

3. Calldata

- Read-only
- Non-modifiable
- Cheapest data location
- Used for external function parameters
- Contains immutable transaction data

Calldata is more gas-efficient than memory for external functions.

Important Rules

- State variables → Always storage

- Function parameters → Prefer calldata (for external functions) or memory

- Local reference variables → Must explicitly specify data location

Sonam Chhabaidiya
D20A 16

7. Transactions: Ether and Wei, Gas and Gas Price, Sending Transactions

Ether Units

- Smallest unit of Ether is **wei**
- 1 Ether = 10^{18} wei
- 1 Gwei = 10^{9} wei (commonly used for gas prices)

Gas Concept

Gas represents the computational effort required to execute a transaction.

Gas Components:

- **Gas Limit**
  Maximum gas a sender is willing to spend

- **Gas Price**
  Price per unit of gas (in wei or gwei)

Total Transaction Fee

Total Fee = Gas Used × Gas Price

- Total Fee = Gas Used × Gas Price

Important Global Variables

- msg.value → Amount of wei sent with the transaction
- tx.gasprice → Gas price of the current transaction
- msg.sender → Address of the caller

Sending Ether in Solidity

To send Ether, the function must be marked as payable.

Example:

```
function sendEther(address payable recipient) public payable {

   recipient.transfer(msg.value);

}
```

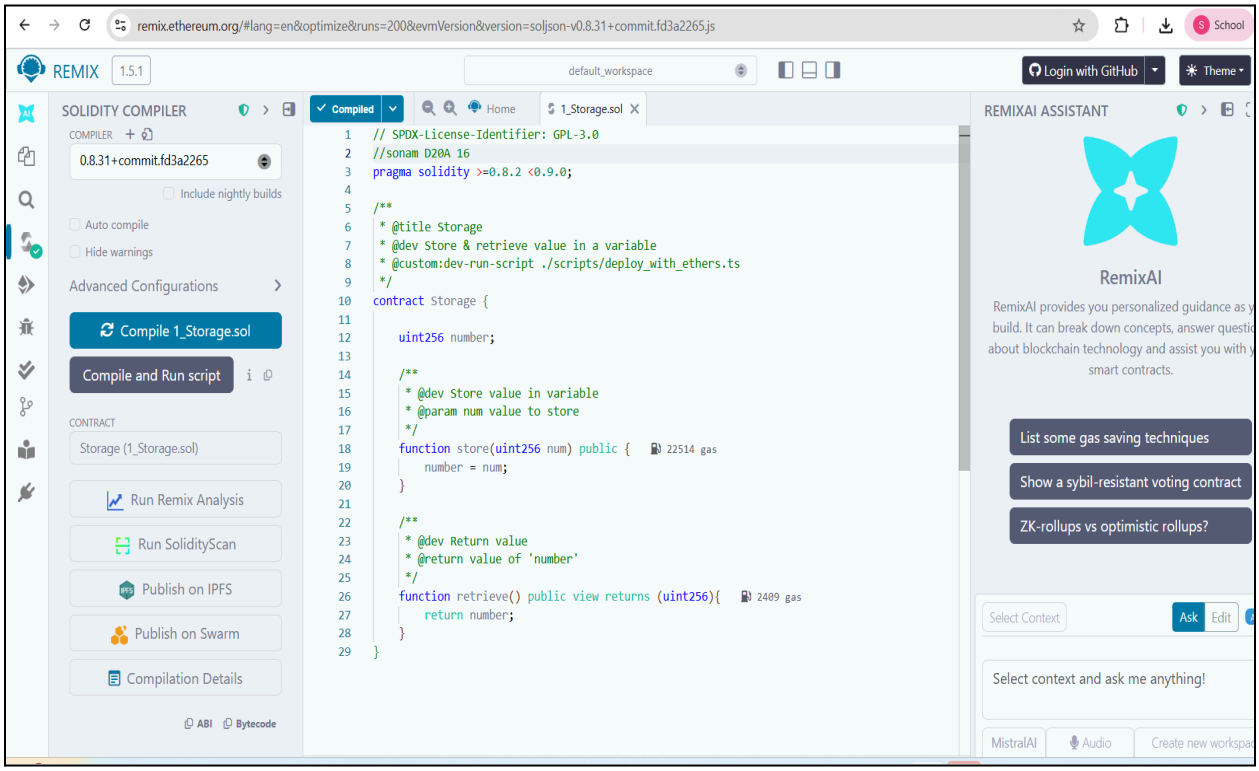
Sonam Chhabaidiya
D20A 16

Explanation:

- payable → Allows function to receive Ether
- recipient.transfer(msg.value) → Transfers Ether to recipient
- msg.value → Amount of Ether sent with transaction

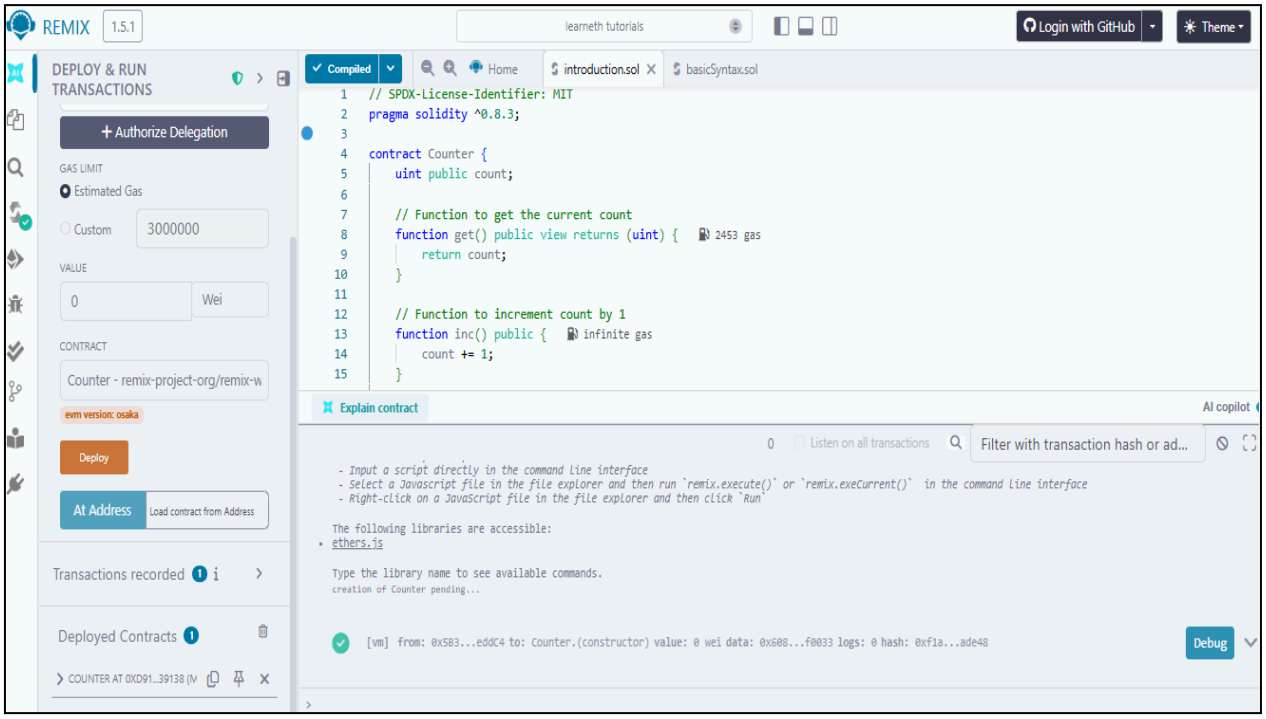
Implementation:

- Tutorial no. 1 – Compile the code

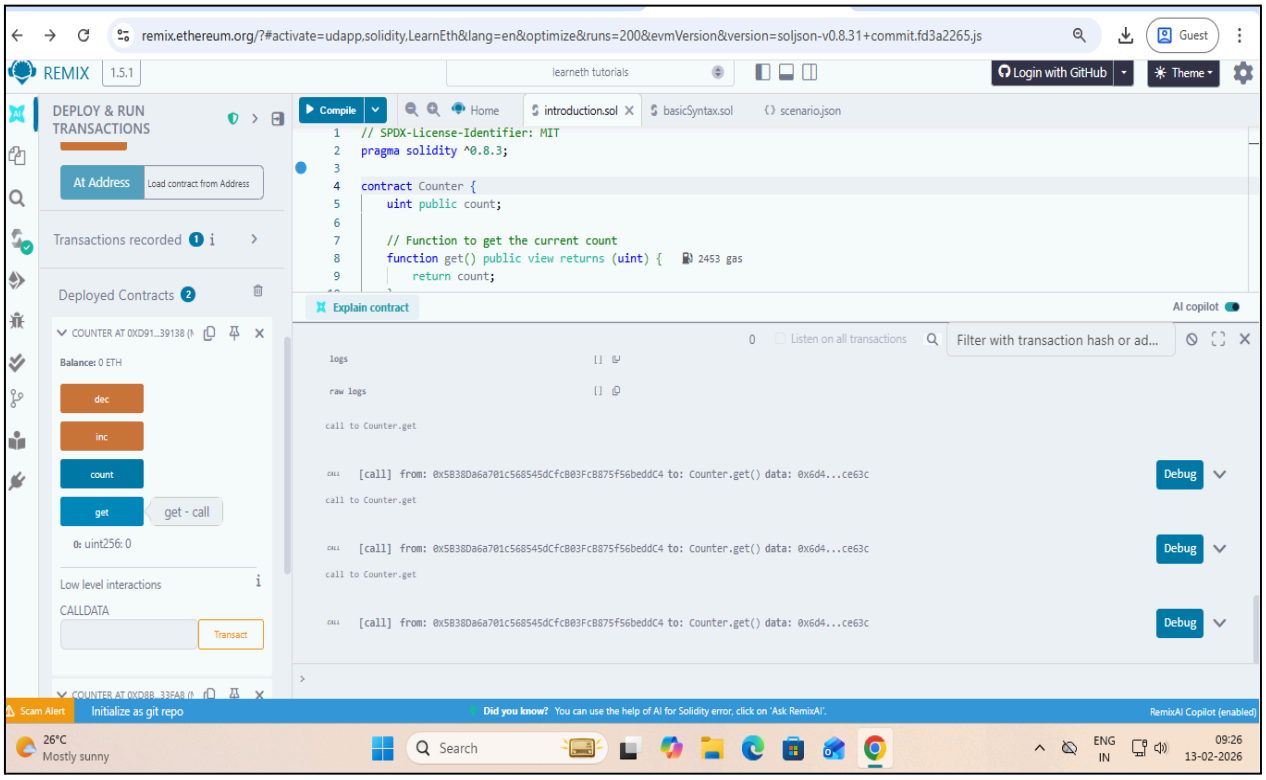


Sonam Chhabaidiya
D20A 16

- Tutorial no. 1 – Deploy the contract

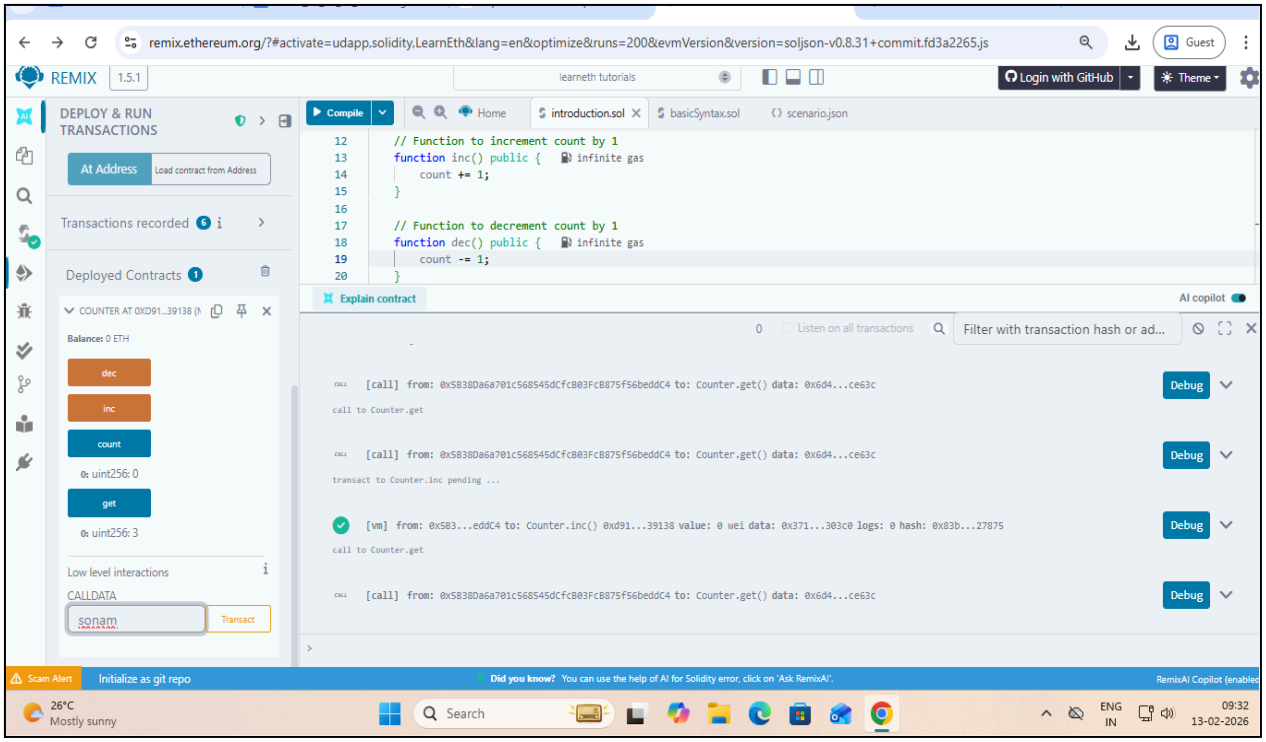


- Tutorial no. 1 – get

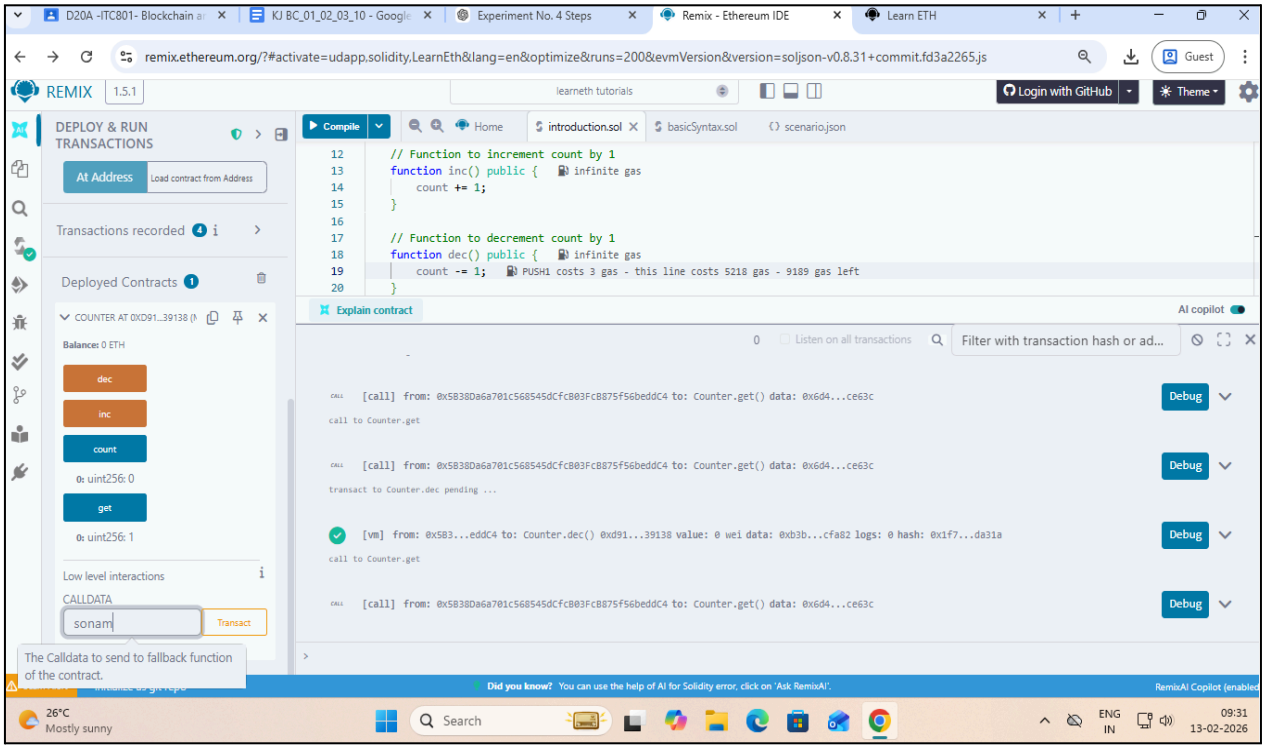


Sonam Chhabaidiya
D20A 16

- Tutorial no. 1 – Increment

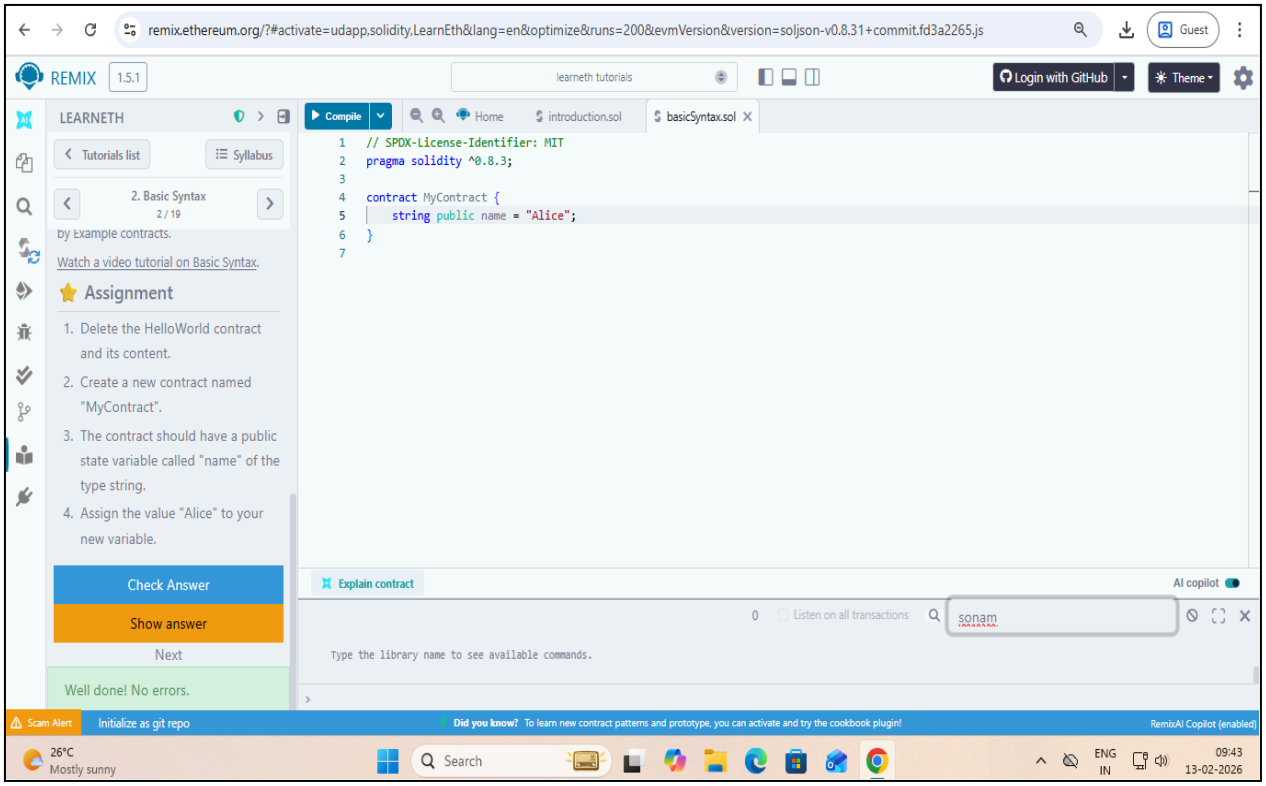


- Tutorial no. 1 – Decrement

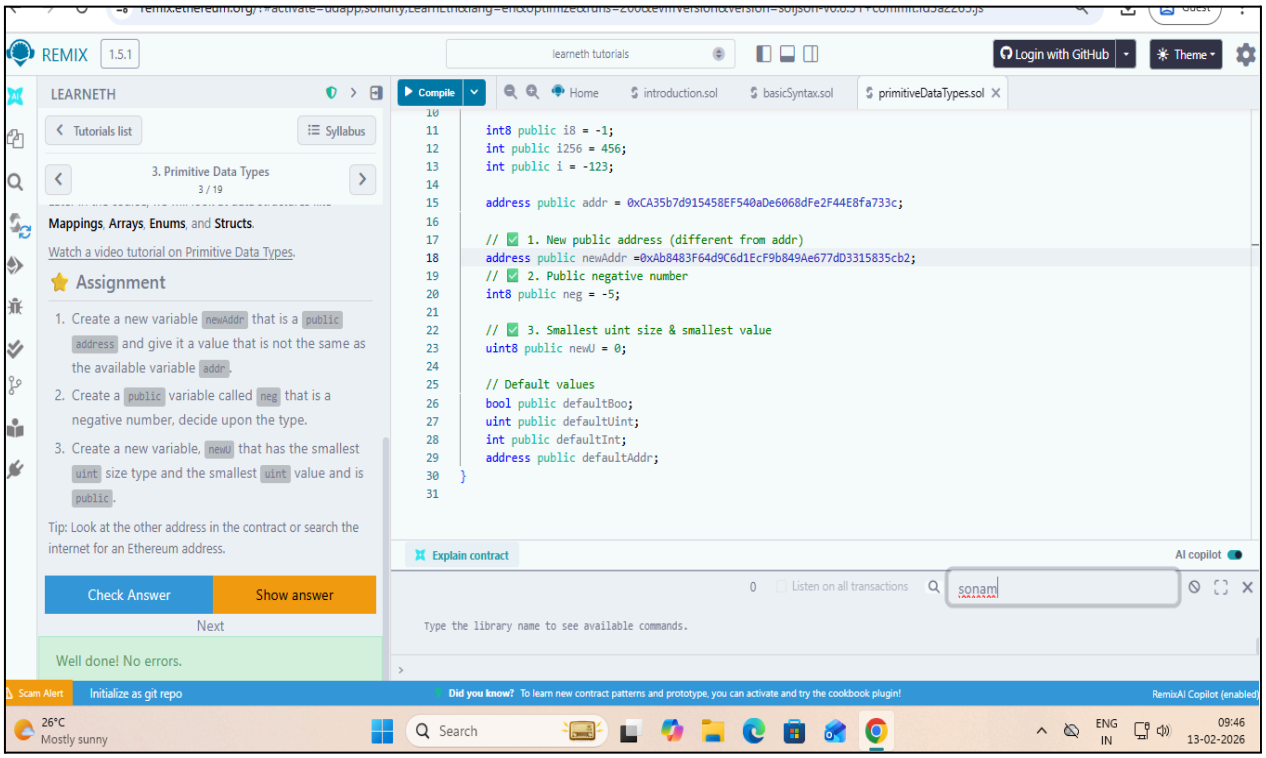


Sonam Chhabaidiya
D20A 16

- Tutorial no. 2

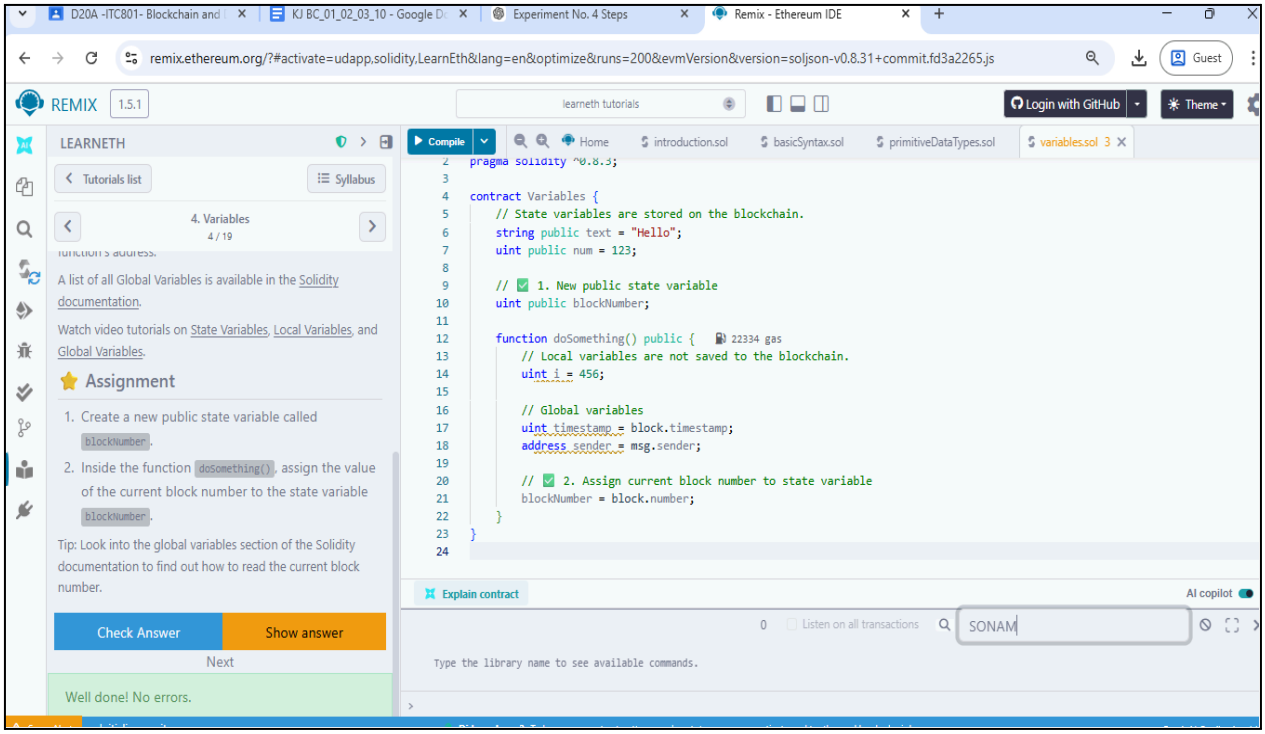


- Tutorial no. 3

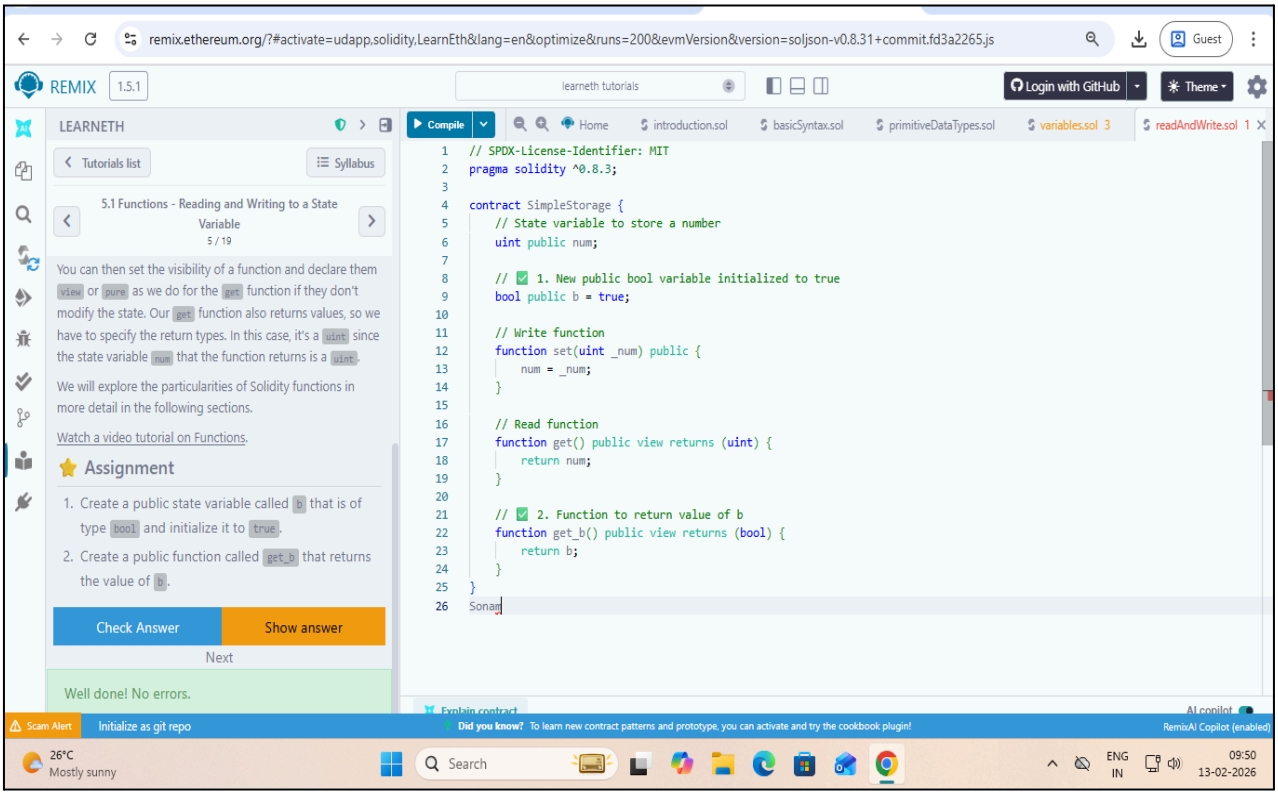


Sonam Chhabaidiya
D20A 16

- Tutorial no. 4

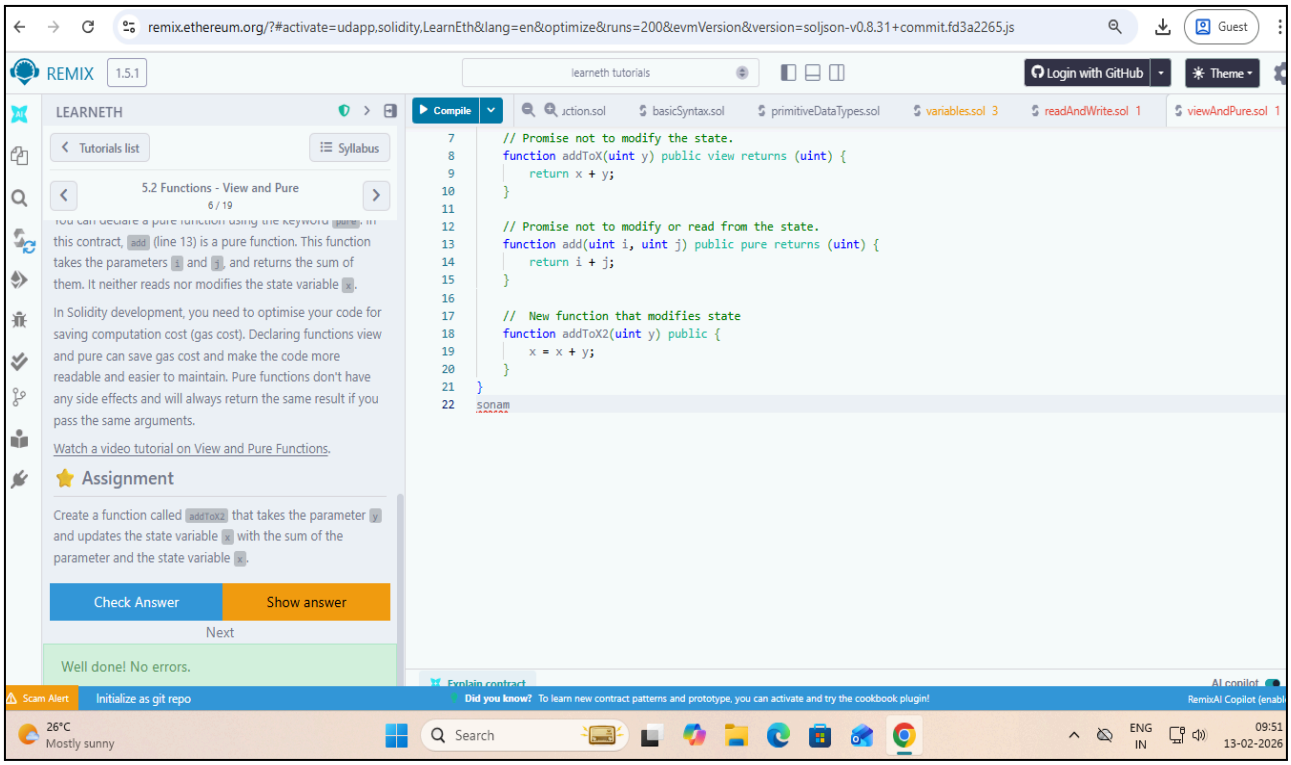


- Tutorial no. 5

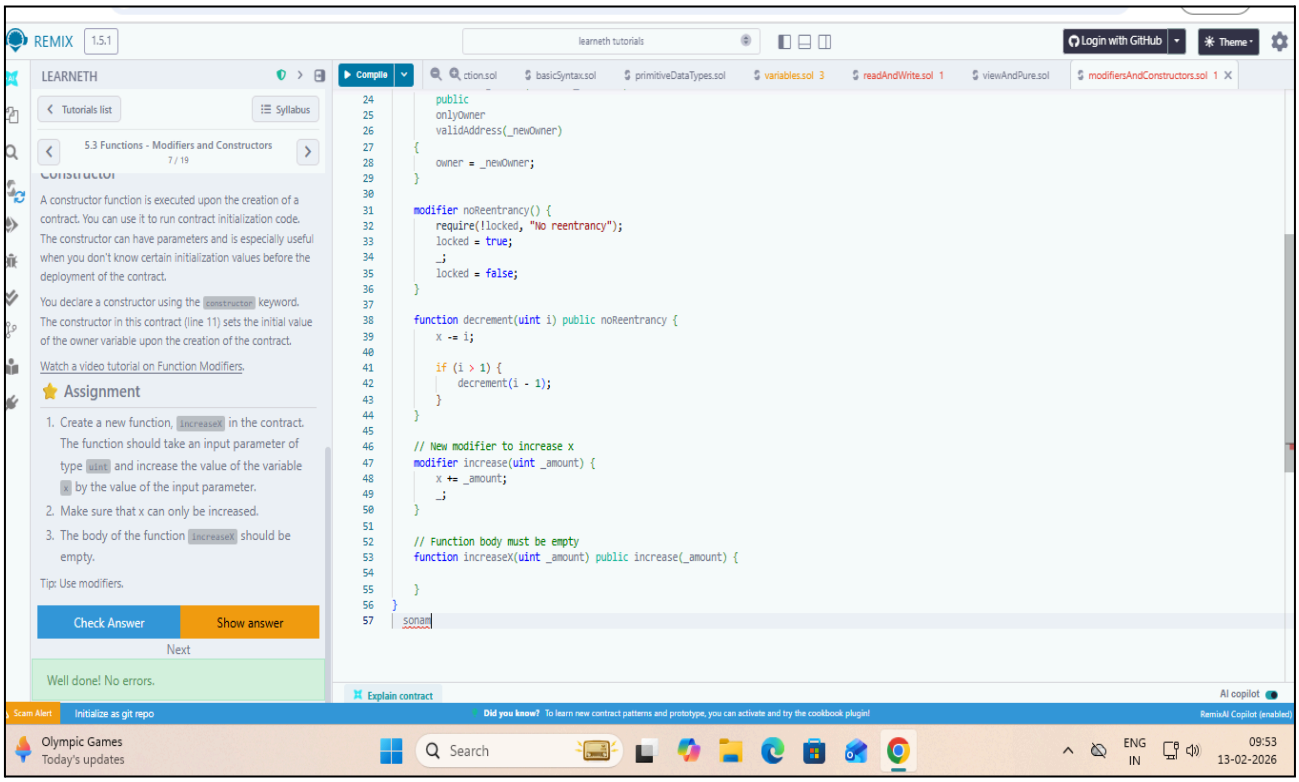


Sonam Chhabaidiya
D20A 16

- Tutorial no. 6

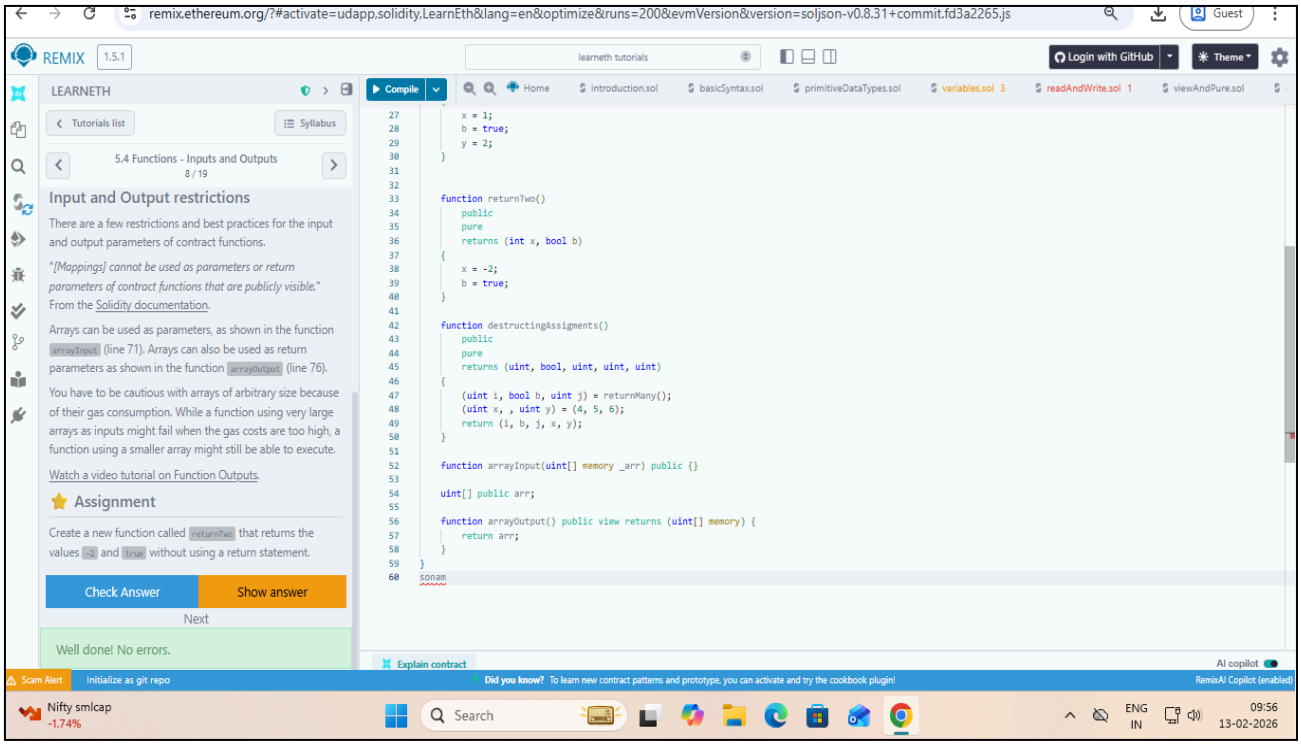


- Tutorial no. 7

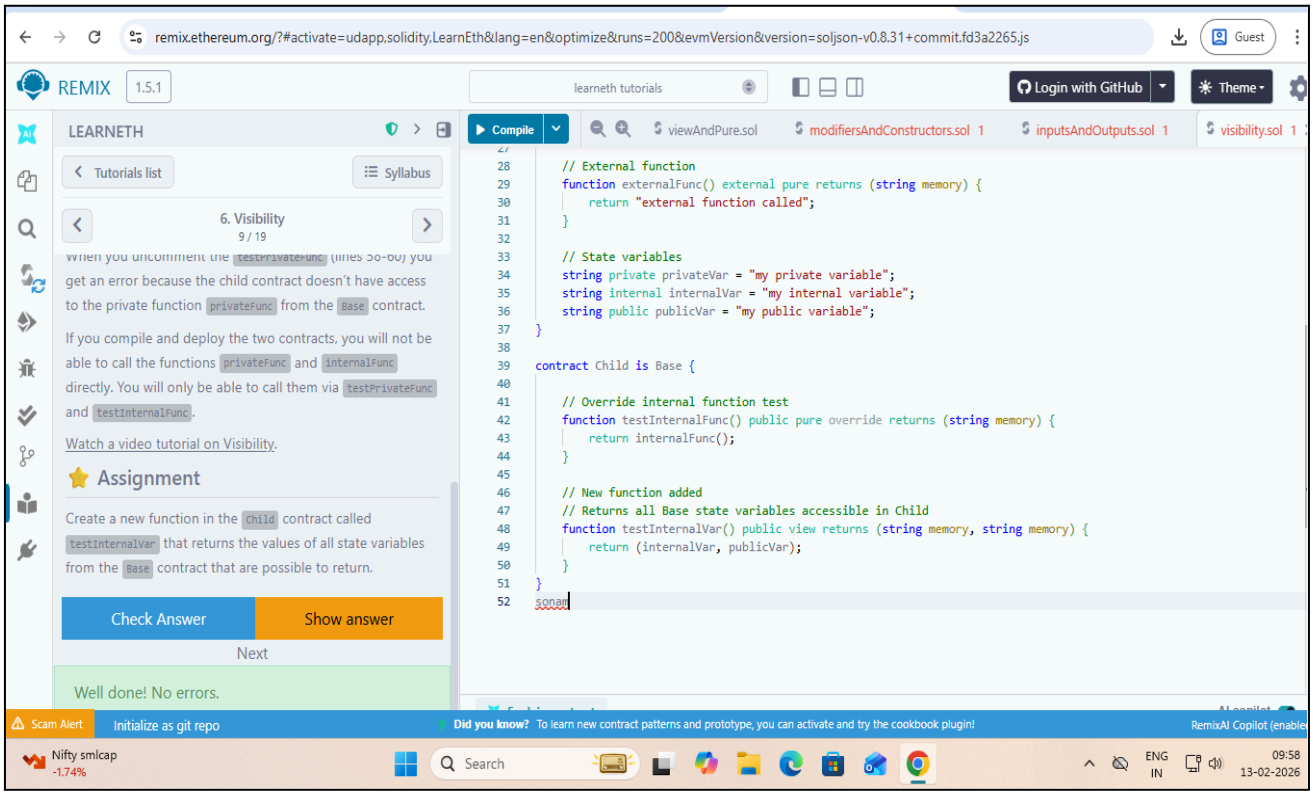


Sonam Chhabaidiya
D20A 16

- Tutorial no. 8

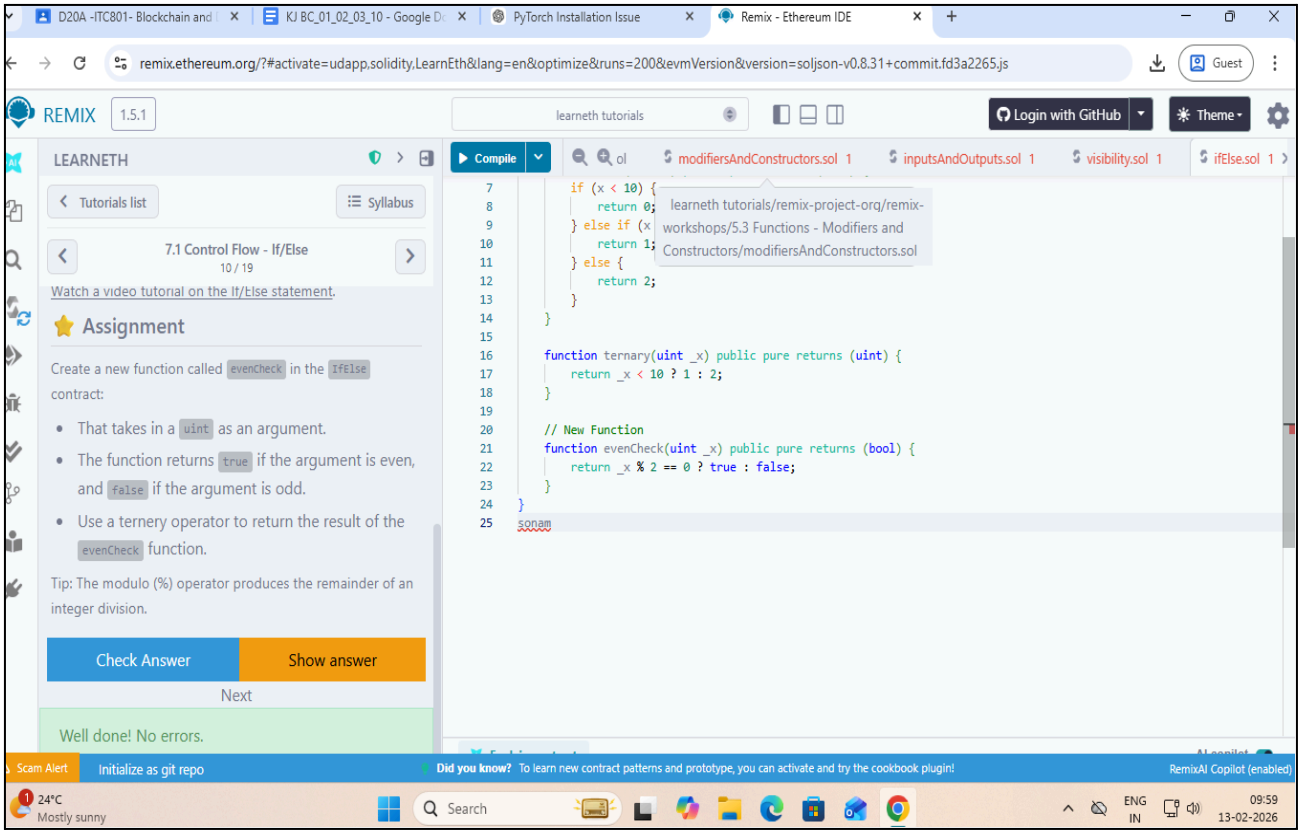


- Tutorial no. 9

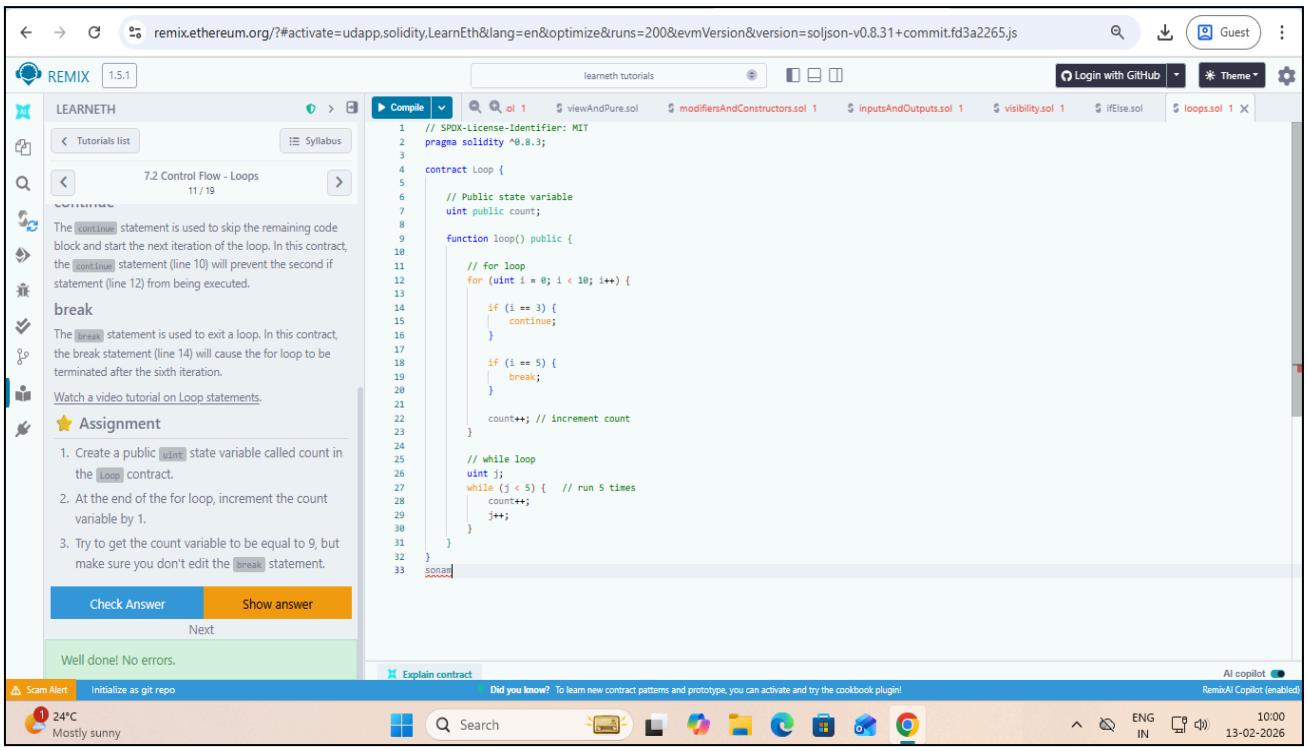


Sonam Chhabaidiya
D20A 16

- Tutorial no. 10

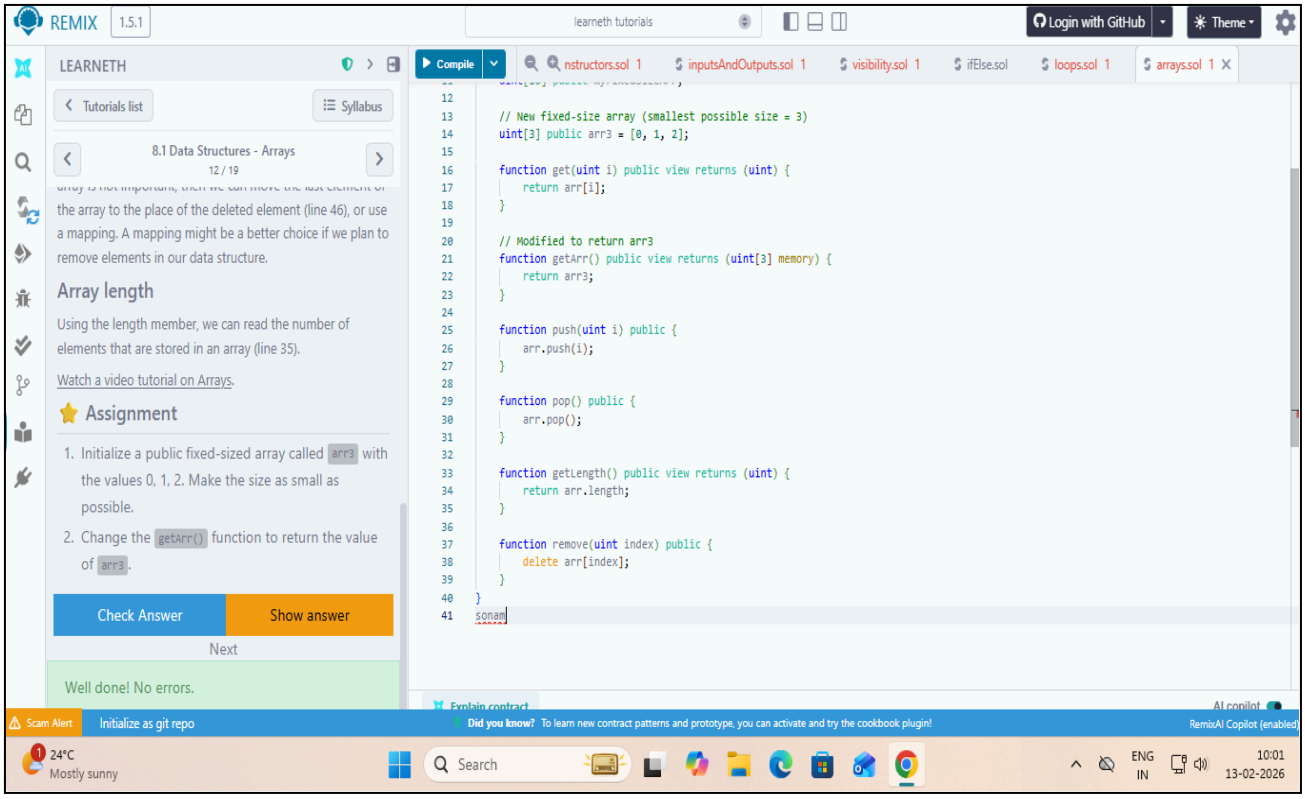


- Tutorial no. 11

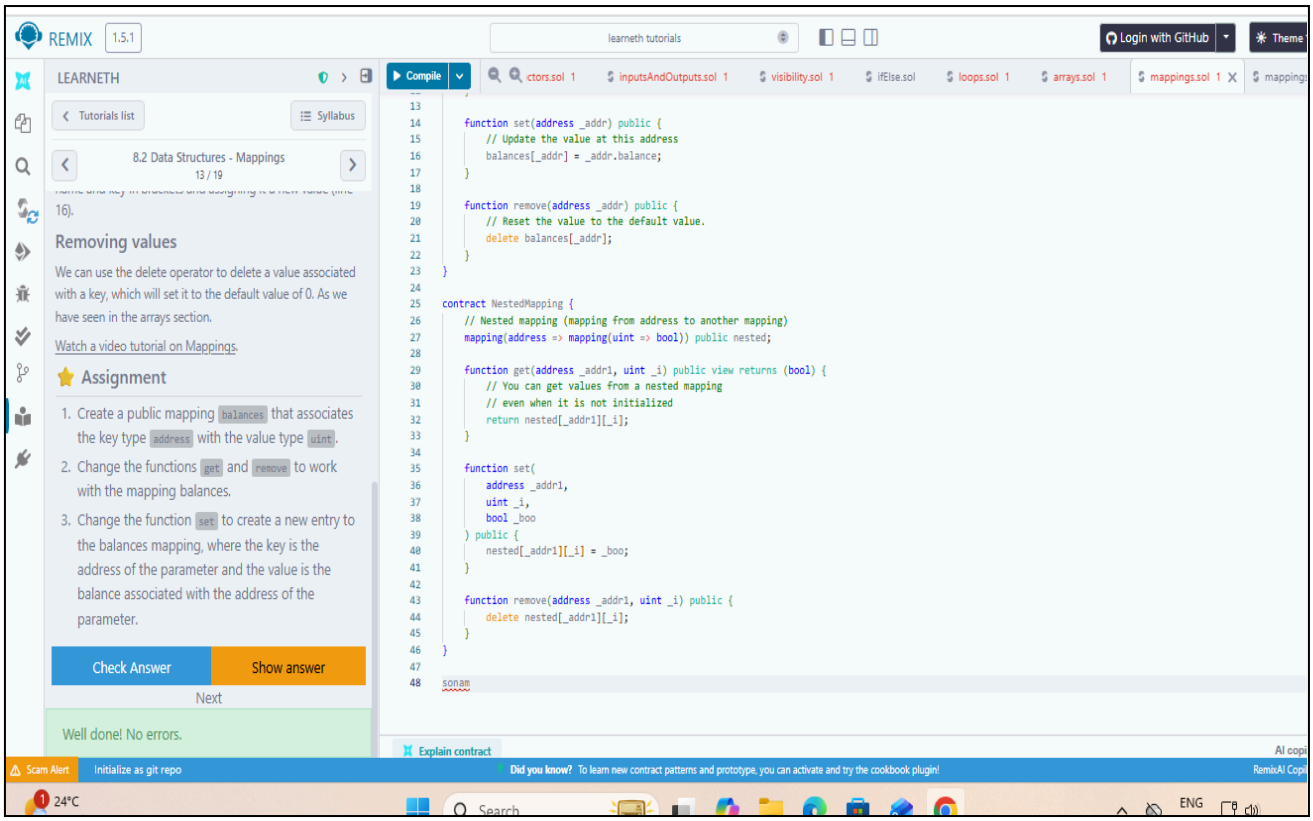


Sonam Chhabaidiya
D20A 16

- Tutorial no. 12

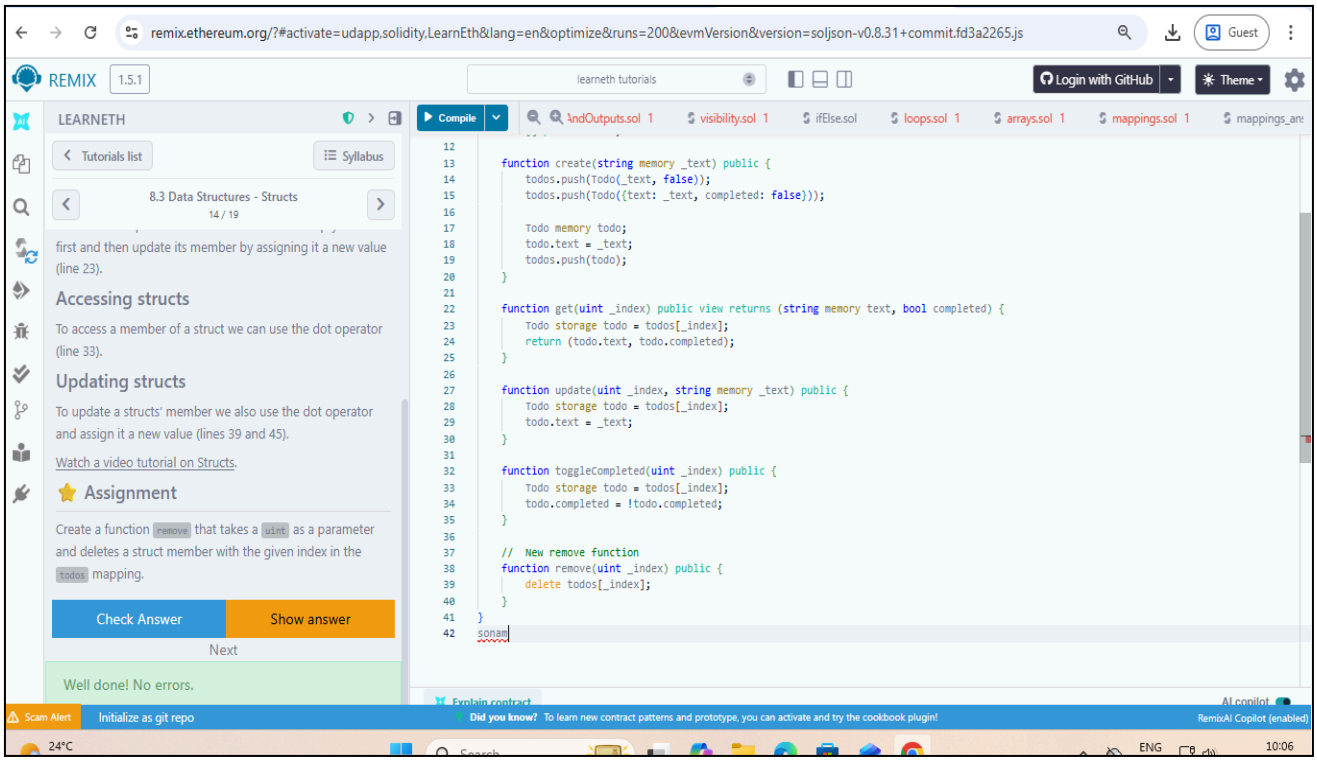


- Tutorial no. 13

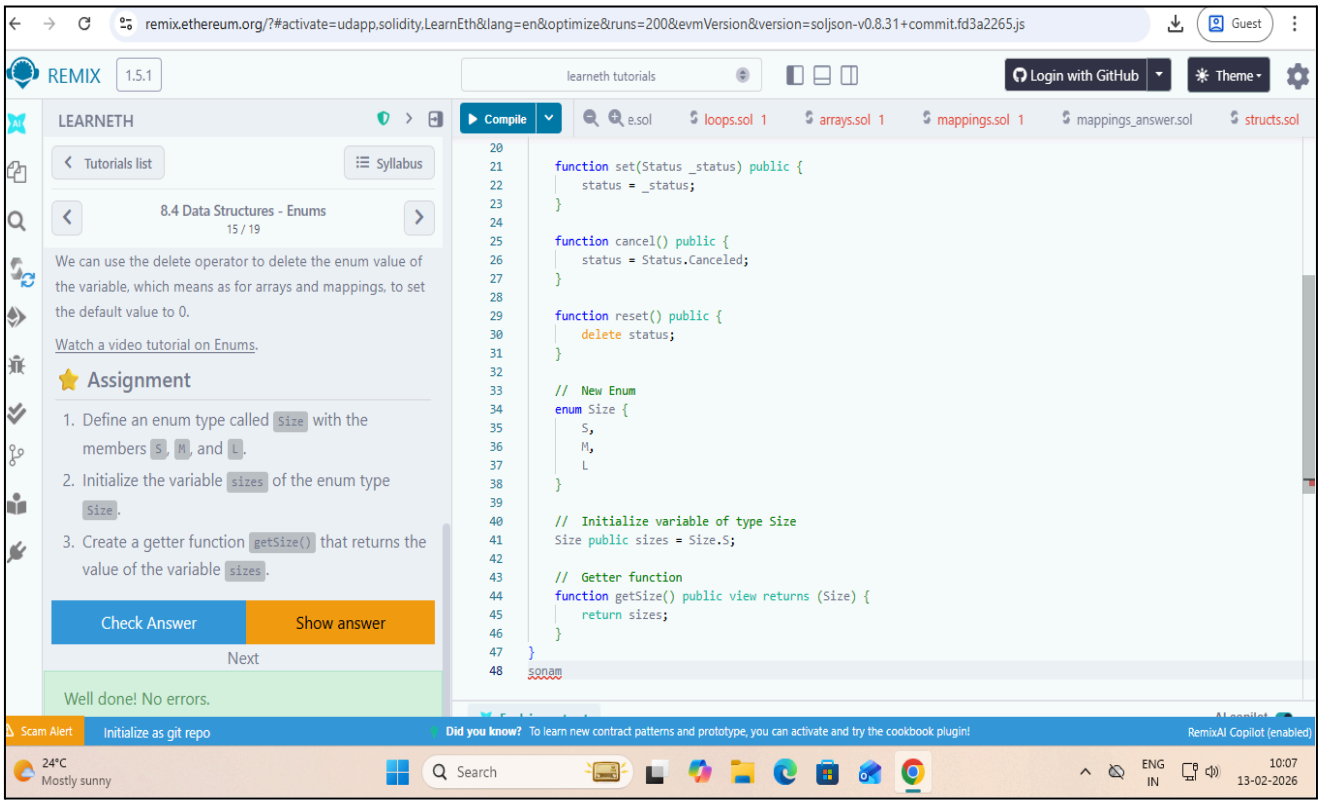


Sonam Chhabaidiya
D20A 16

- Tutorial no. 14

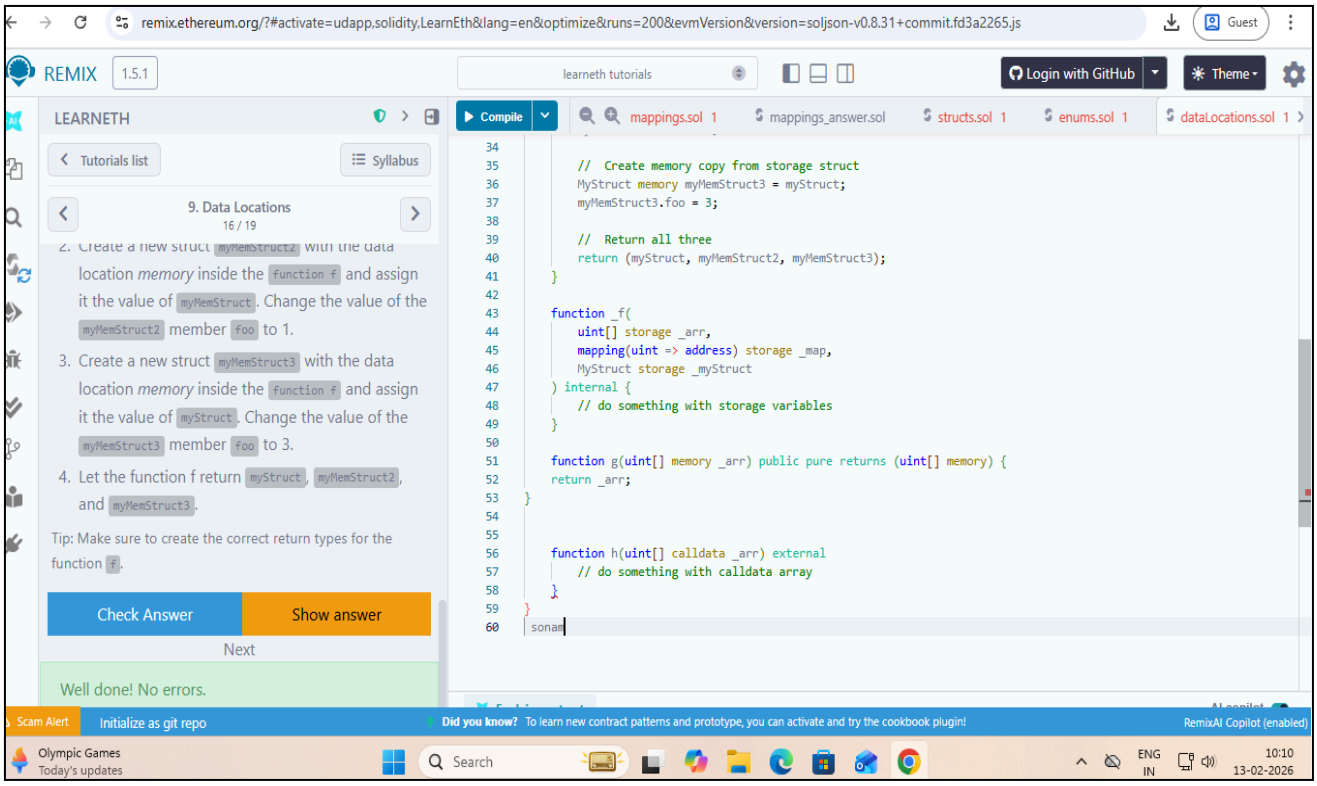


- Tutorial no. 15

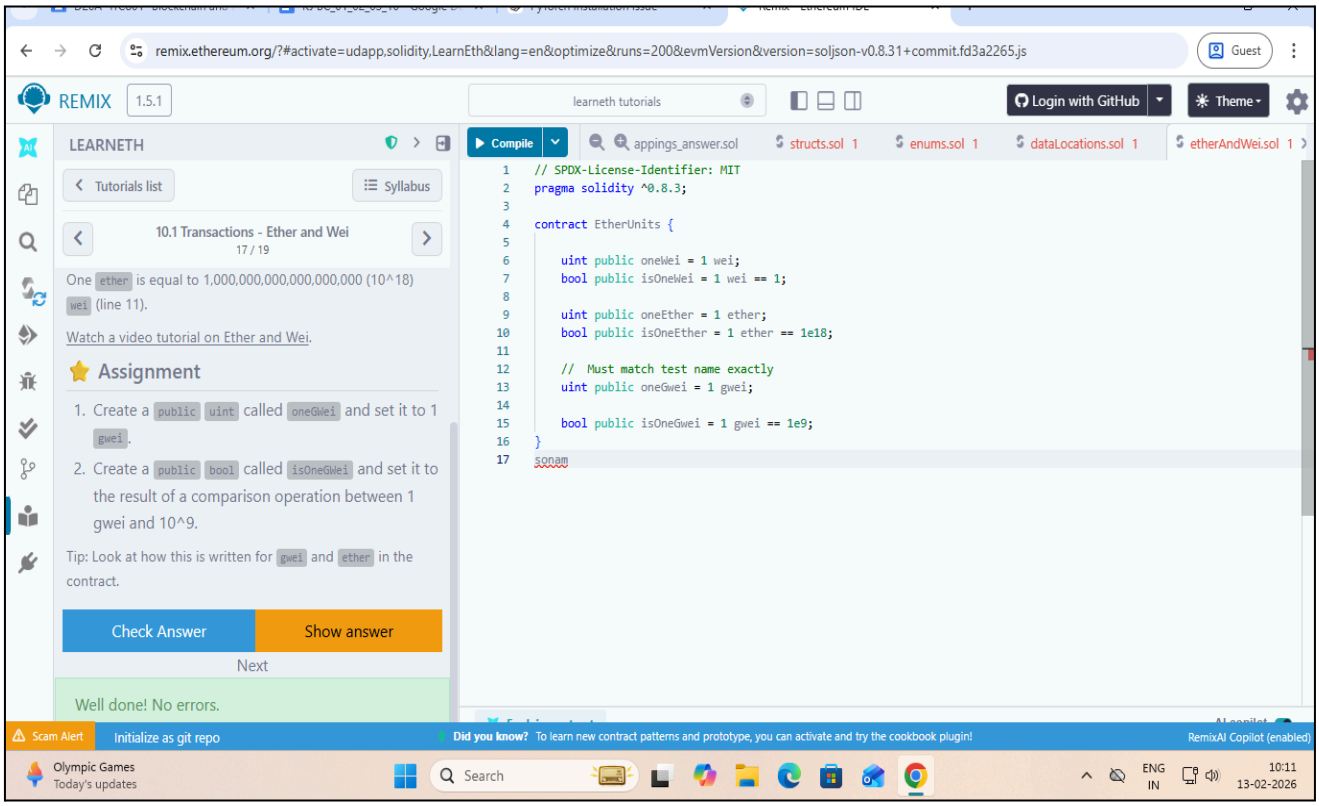


Sonam Chhabaidiya
D20A 16

- Tutorial no. 16

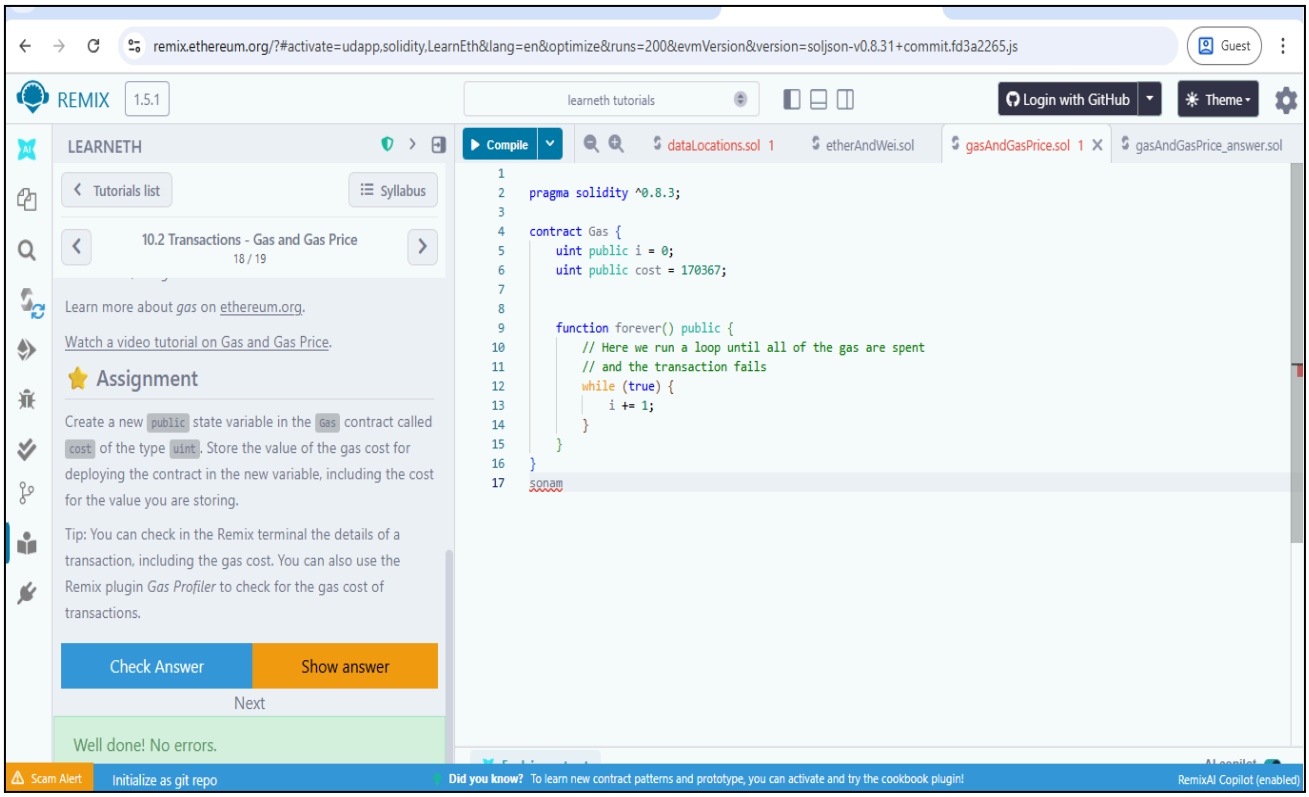


- Tutorial no. 17

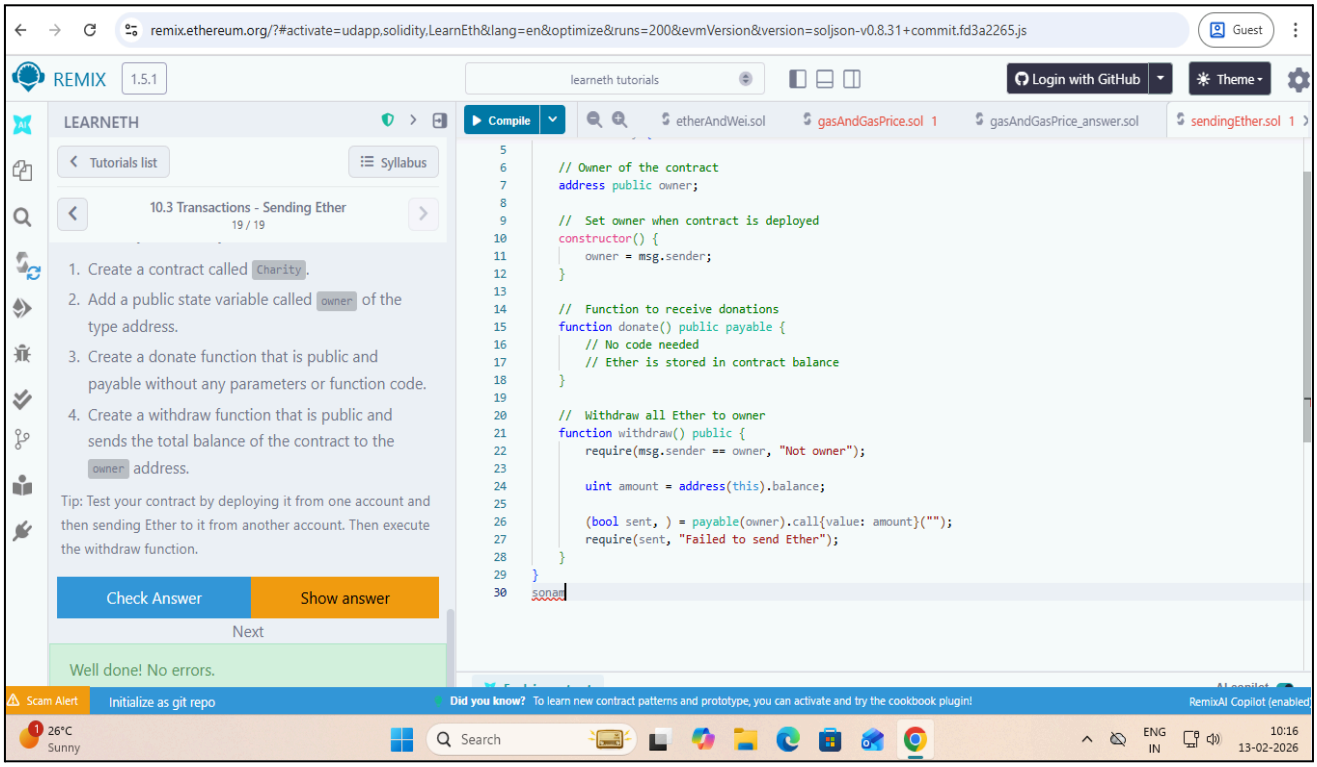


Sonam Chhabaidiya
D20A 16

- Tutorial no. 18



- Tutorial no. 19



Sonam Chhabaidiya
D20A 16

Conclusion:

Through this experiment, the fundamentals of Solidity programming were explored by completing practical assignments in the Remix IDE. Concepts such as data types, variables, functions, visibility, modifiers, constructors, control flow, data structures, and transactions were implemented and understood. The hands-on practice helped in designing, compiling, and deploying smart contracts on the Remix VM, thereby strengthening the understanding of blockchain concepts. This experiment provided a strong foundation for developing and managing smart contracts efficiently.