Blockchain
Lab
Experiment 4

Mohit Patil
D20A 04

Aim - Hands on Solidity Programming Assignments for creating Smart Contracts.

Theory -

Primitive Data Types, Variables, and Functions (Pure & View)

In Solidity, primitive data types are the basic building blocks of smart contracts. Commonly used data types include:

- **`uint`** / **`int`** – Unsigned and signed integers of various sizes (e.g., `uint256`, `int128`).
- **`bool`** – Represents logical values (`true` or `false`).
- **`address`** – Stores a 20-byte Ethereum address, typically used for user accounts or contract addresses.
- **`bytes`** / **`string`** – Used to store binary data or textual information.

Types of Variables

Solidity supports different categories of variables:

- **State variables** – Stored permanently on the blockchain.
- **Local variables** – Temporary variables created during function execution.
- **Global variables** – Predefined variables such as `msg.sender`, `msg.value`, and `block.timestamp`.

Functions in Solidity

Functions define the logic of a smart contract. Two important function types are:

- **`pure`** **functions** – Cannot read or modify blockchain state. They rely only on input parameters and internal calculations.
- **`view`** **functions** – Can read state variables but cannot modify them.

Using `pure` and `view` appropriately helps reduce gas costs and ensures function integrity.

Inputs and Outputs of Functions

Solidity functions can accept input parameters and return one or more output values.

- **Inputs** allow users or other contracts to pass data into a function.
- **Outputs** return results after performing computations.

For example, a function may accept an Ether amount and return whether a transaction was successful. Solidity also supports **named return variables**, which enhance code readability and simplify debugging.

Visibility, Modifiers, and Constructors

Function Visibility

Visibility determines who can access a function:

- `public` – Accessible from inside and outside the contract.
- `private` – Accessible only within the same contract.
- `internal` – Accessible within the contract and its derived (child) contracts.
- `external` – Can be called only from outside the contract.

Modifiers

Modifiers are reusable code blocks that alter function behavior. They are commonly used for access control, such as restricting certain functions to the contract owner (e.g., `onlyOwner`).

Constructors

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

Control Flow: if-else and Loops

Solidity's control flow mechanisms are similar to traditional programming languages:

- `if-else` **statements** enable conditional execution, such as verifying sufficient balance before transferring funds.
- **Loops (**`for`**,** `while`**,** `do-while`**)** allow repeated execution of code, such as iterating through an array.

However, loops must be used cautiously because excessive iterations increase gas consumption, making transactions more expensive.

Data Structures: Arrays, Mappings, Structs, and Enums

Solidity provides several data structures:

- **Arrays** – Store ordered lists of elements. They can be fixed-size or dynamic. Example: an array of user addresses.
- **Mappings** – Store key-value pairs for efficient lookups. Example: `mapping(address => uint)` for tracking account balances. Unlike arrays, mappings cannot be iterated directly.
- **Structs** – Custom data types that group related properties. Example: `struct Player { string name; uint score; }`
- **Enums** – Define a set of predefined constants, improving readability. Example: `enum Status { Pending, Active, Closed }`

Data Locations

Solidity uses three main data locations:

- **`storage`** – Permanently stored on the blockchain (used for state variables).
- **`memory`** – Temporary storage available only during function execution.
- **`calldata`** – Non-modifiable, non-persistent storage used for external function parameters. It is more gas-efficient than `memory`.

Understanding data locations is essential because they directly affect gas costs and contract performance.

Transactions: Ether, Wei, Gas, and Sending Transactions

Ether and Wei

Ether is the primary currency of Ethereum. All values are internally measured in **Wei**, the smallest unit:

1 Ether = 10^{18} Wei

Using Wei ensures high precision in financial calculations.

Gas and Gas Price

Every transaction consumes **gas**, which represents the computational effort required to execute it.

- **Gas price** determines how much Ether is paid per unit of gas.
- Higher gas prices encourage miners (validators) to prioritize a transaction.

Sending Transactions

Transactions are used to:

- Transfer Ether
- Interact with smart contracts

Common methods include:

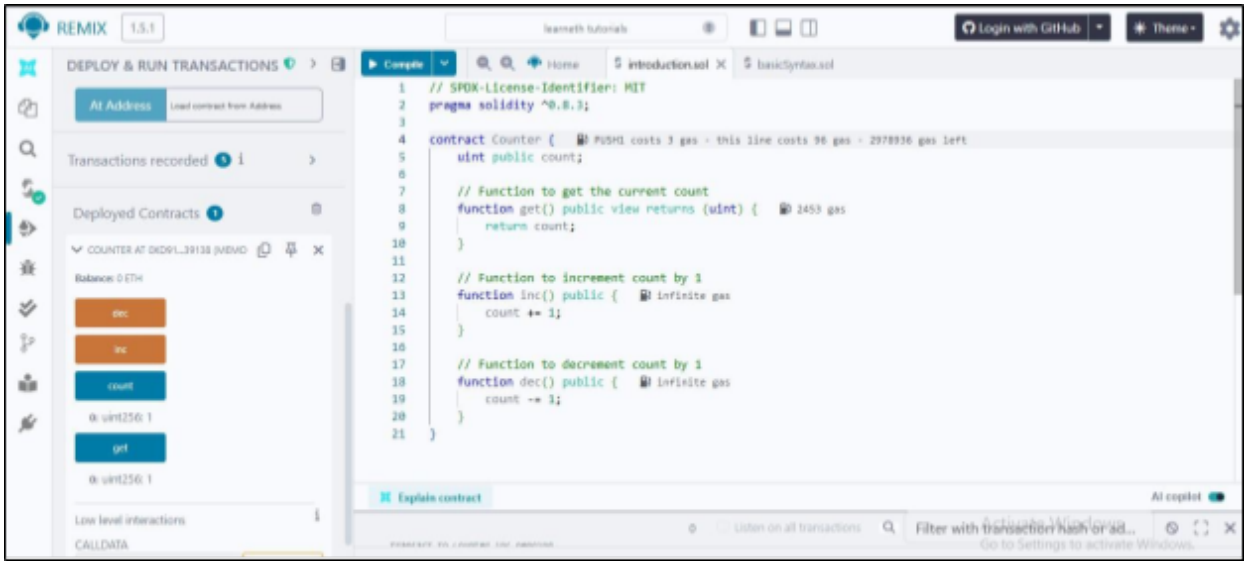
- `transfer()` – Sends Ether with fixed gas.
- `send()` – Similar to transfer but returns a boolean.
- `call()` – More flexible and commonly recommended for sending Ether.

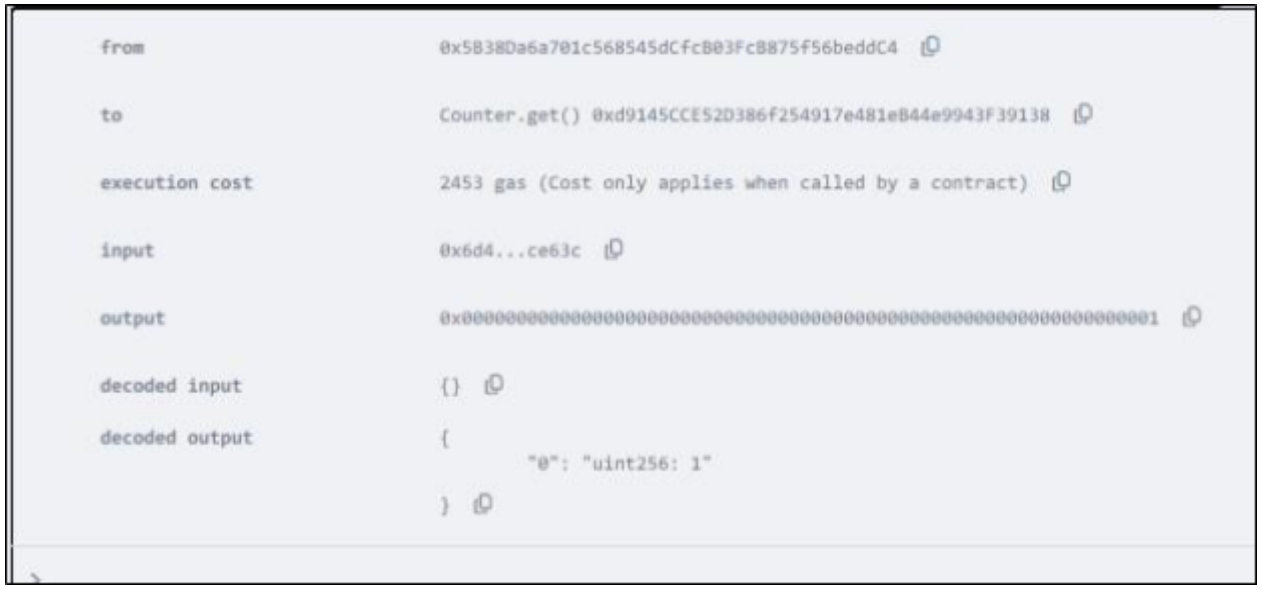
Since every transaction requires gas, writing optimized and efficient smart contracts is crucial.

Code & Output -

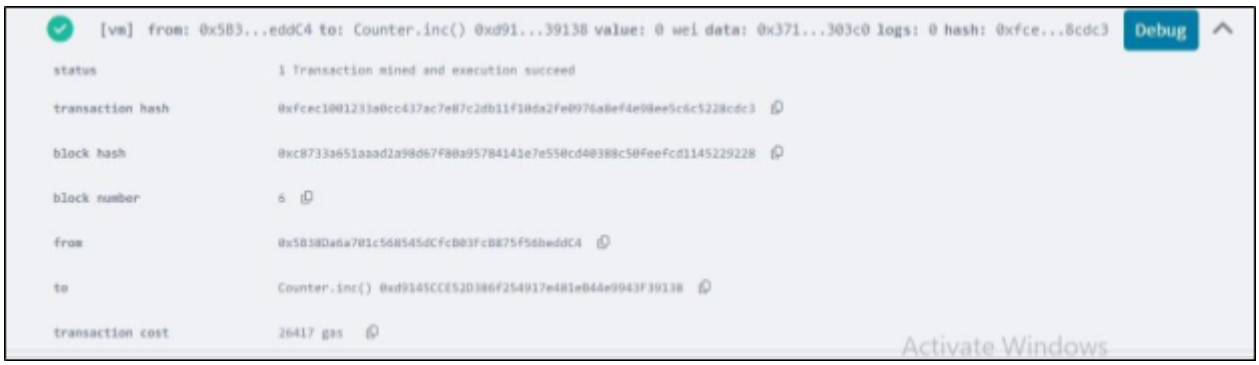
Tutorial 1

1. Get counter value



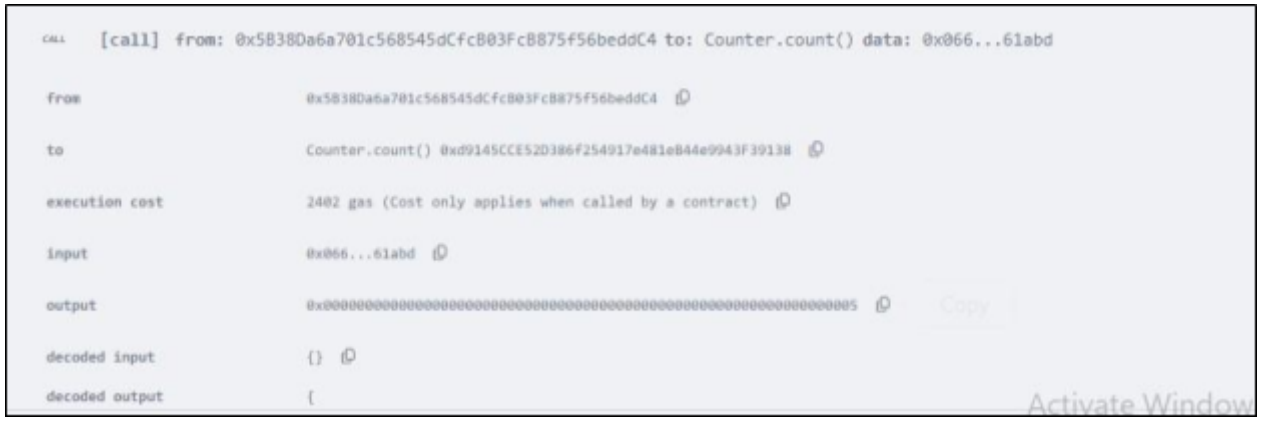


2. Increment counter value

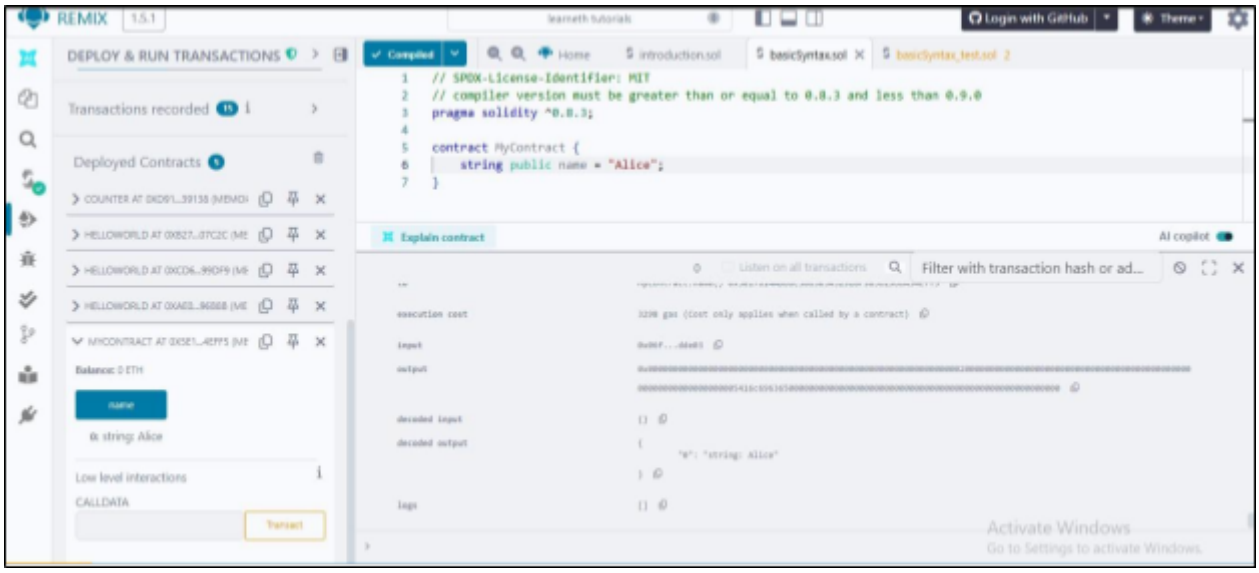


3. Decrement counter value

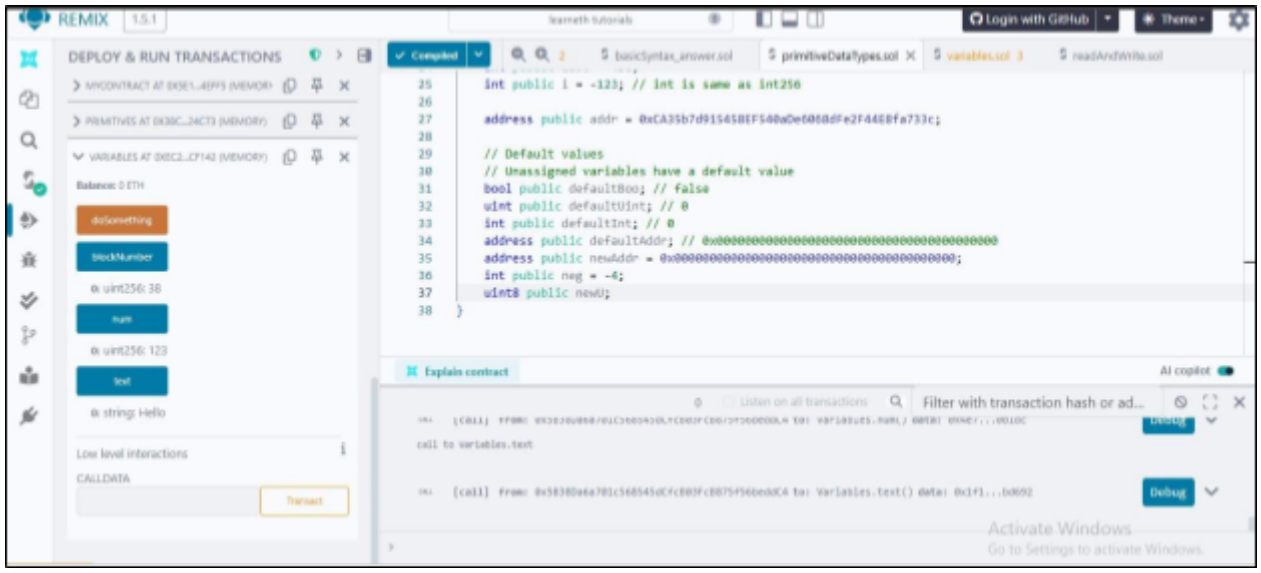
4. Get count value



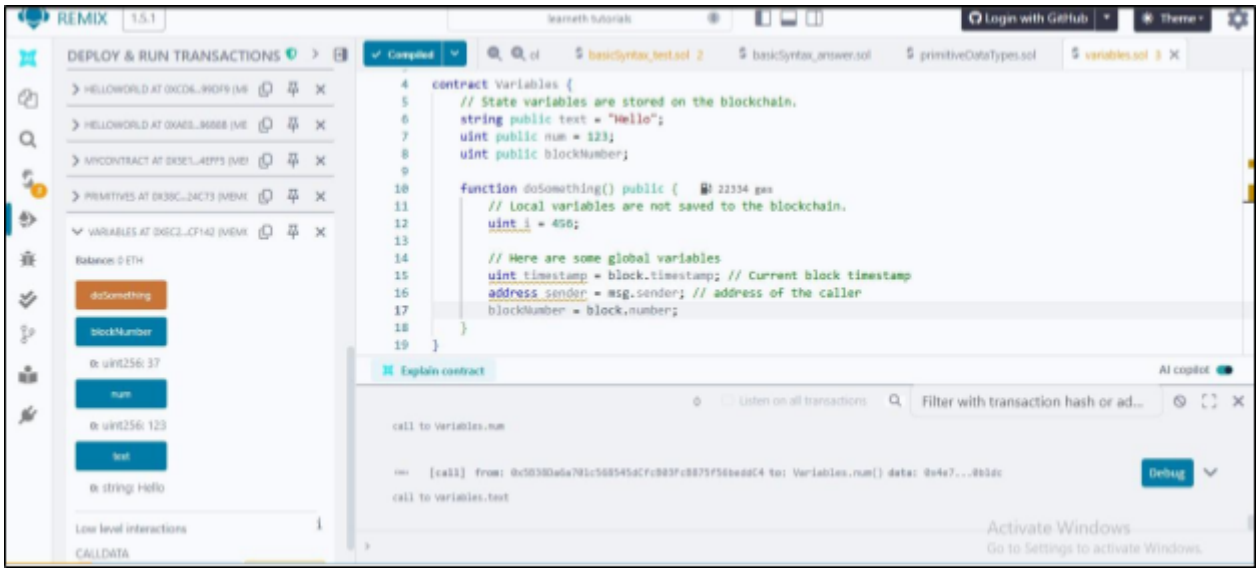
Tutorial 2



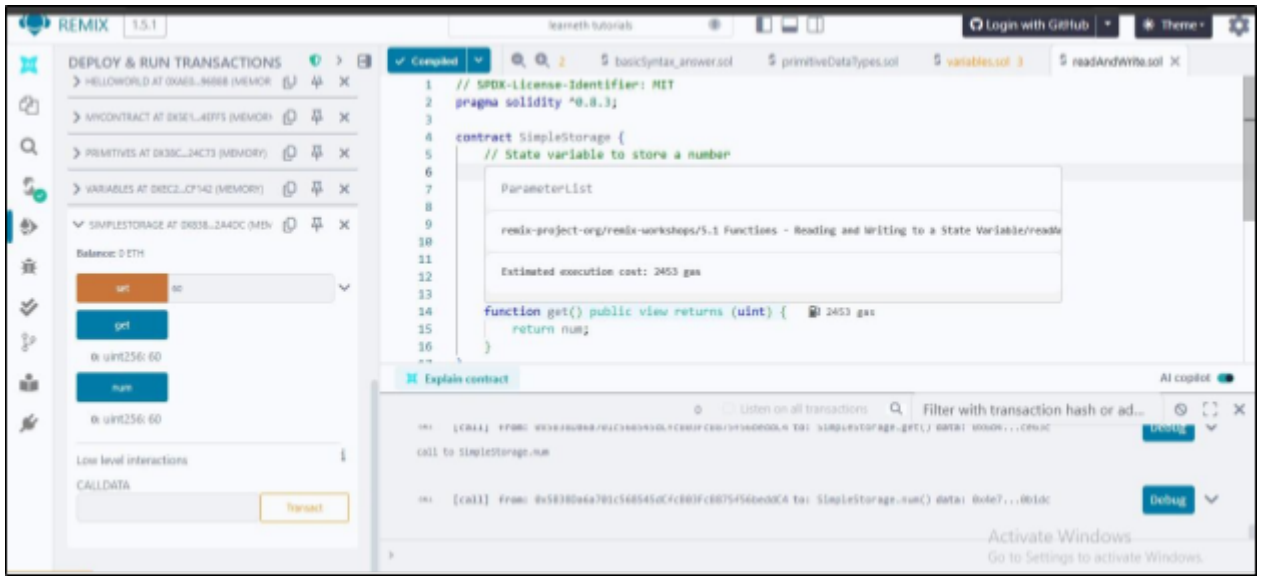
Tutorial 3



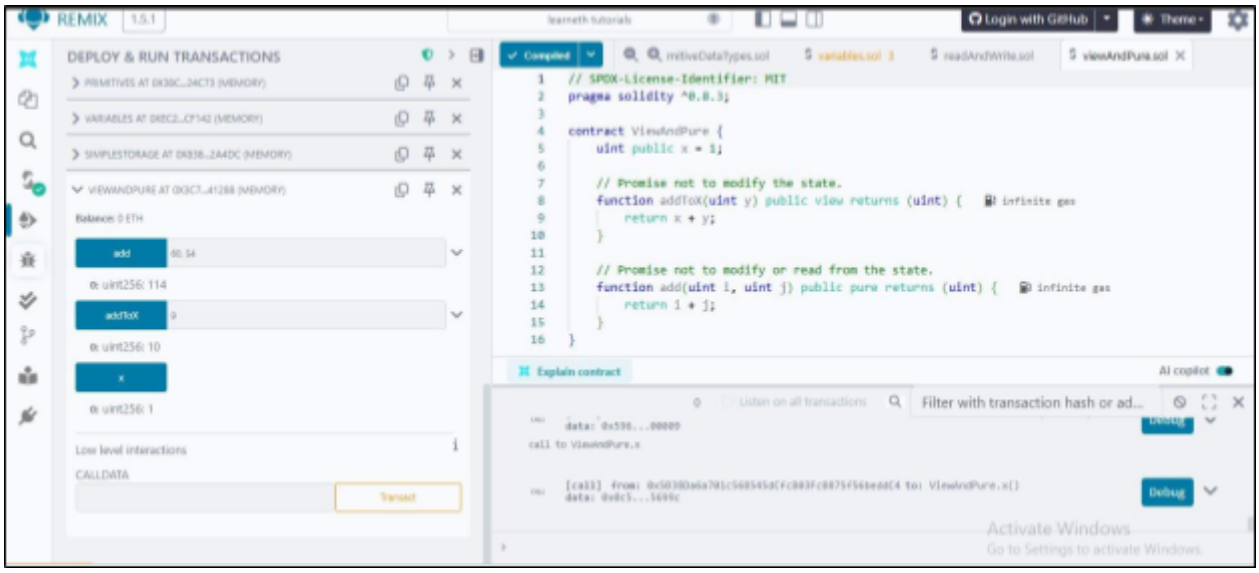
Tutorial 4



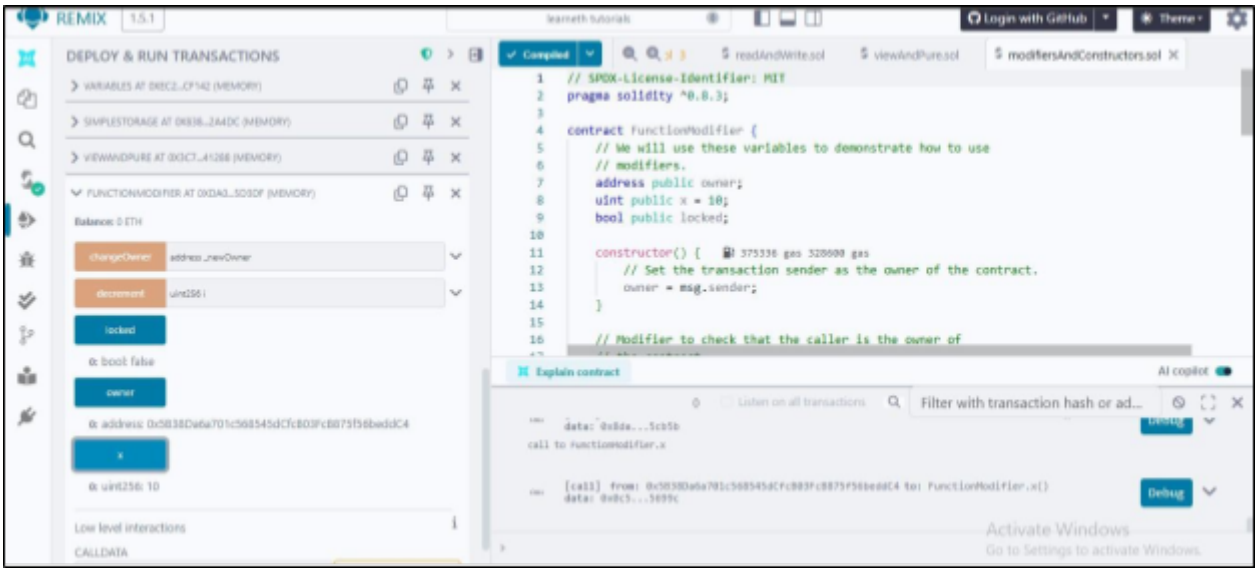
Tutorial 5



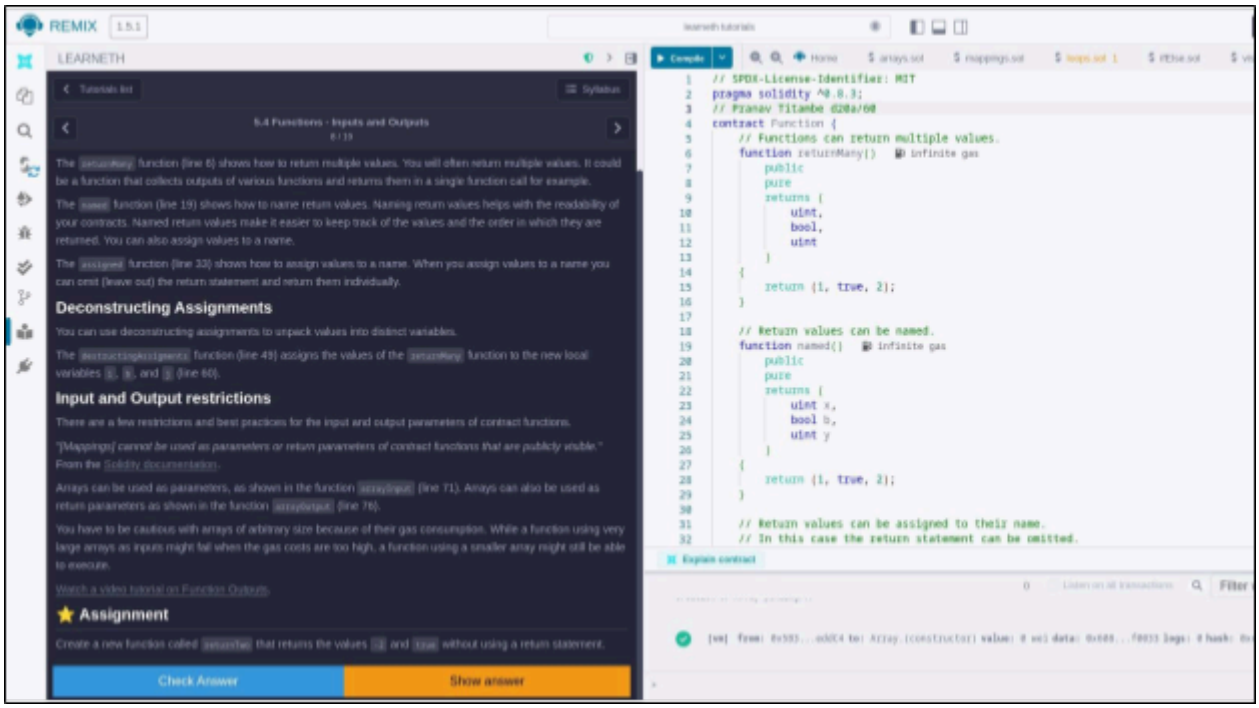
Tutorial 6



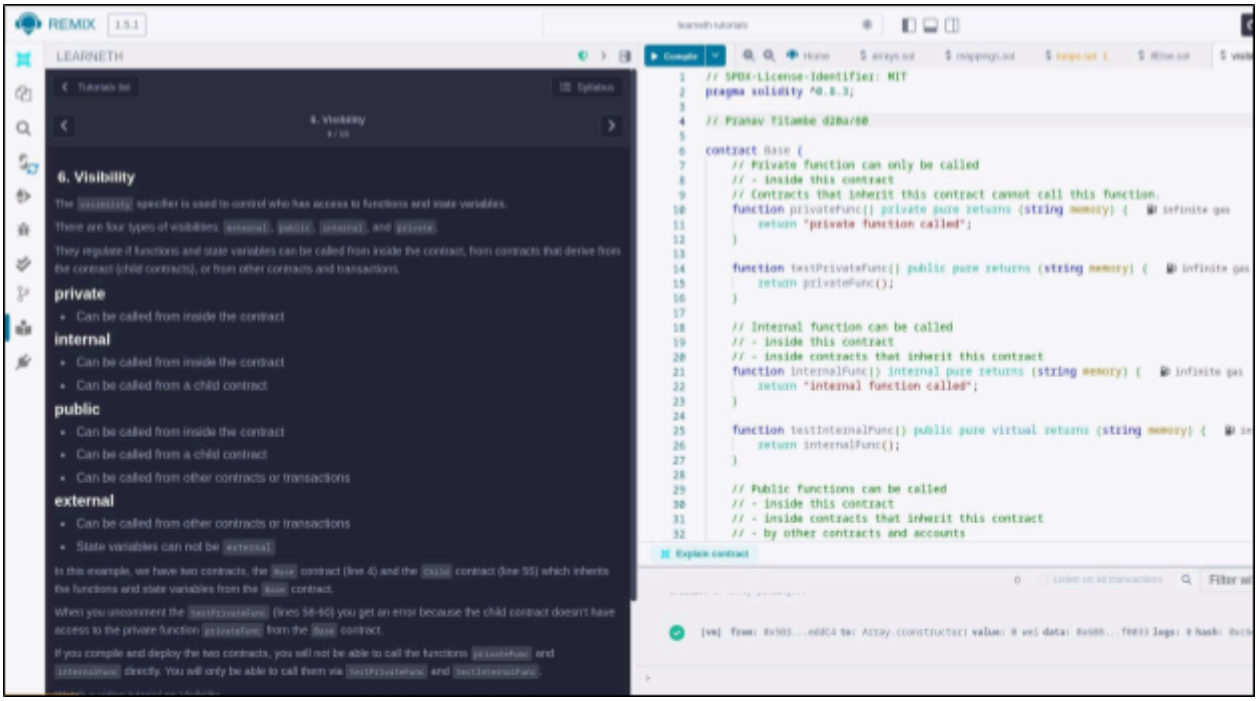
Tutorial 7



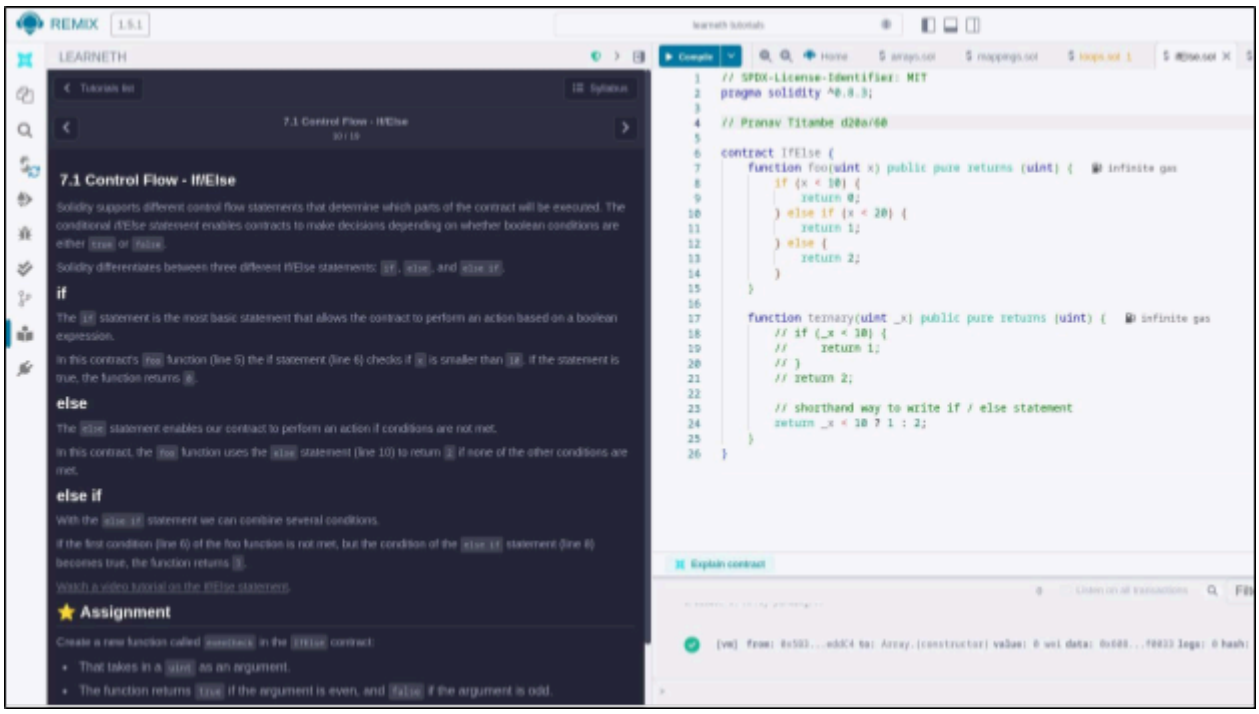
Tutorial 8



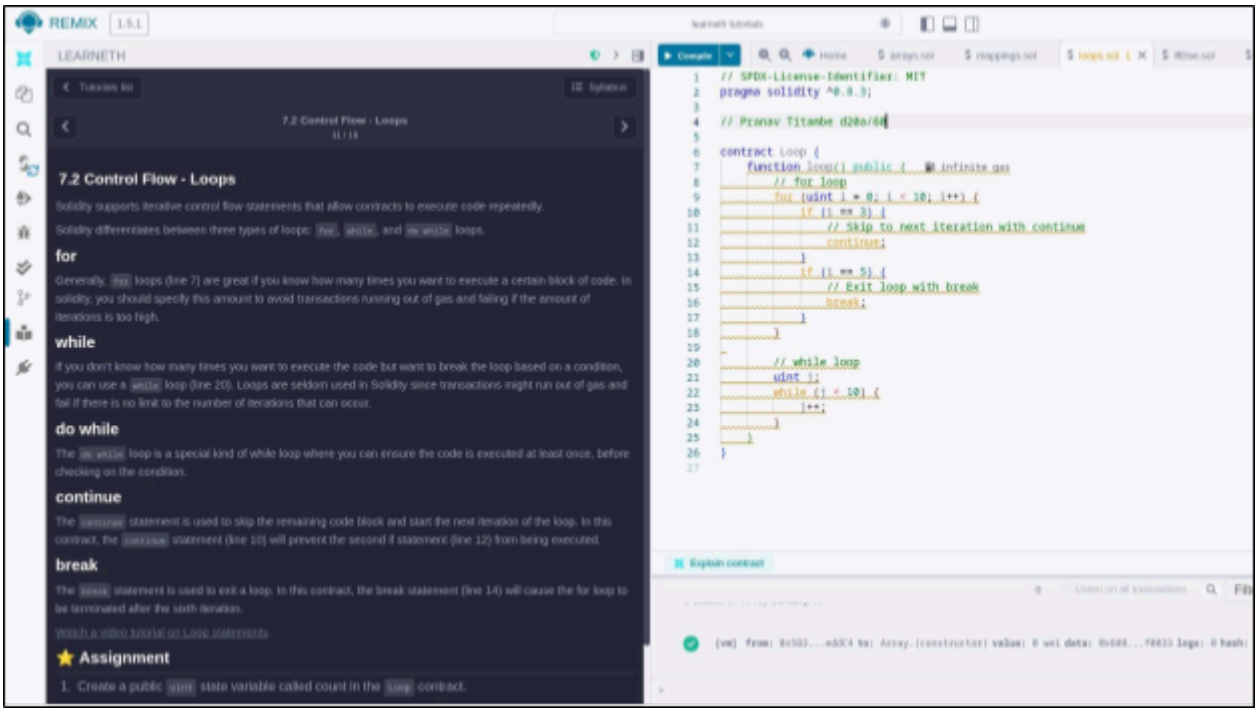
Tutorial 9



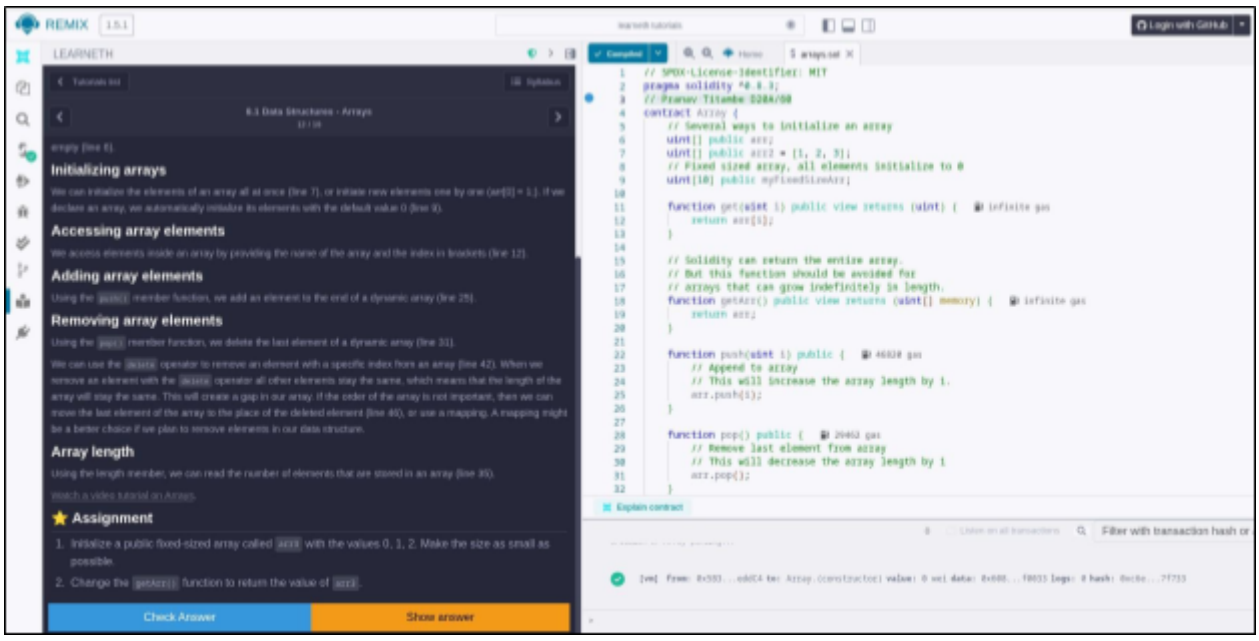
Tutorial 10



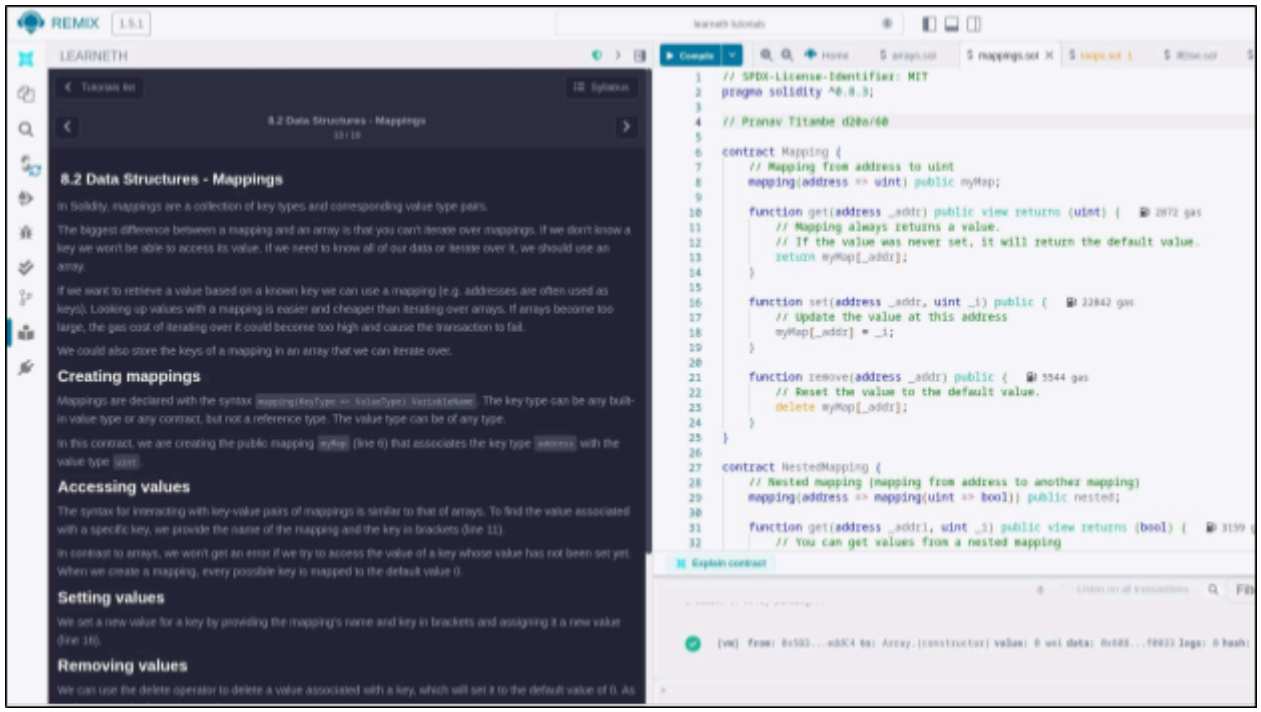
Tutorial 11



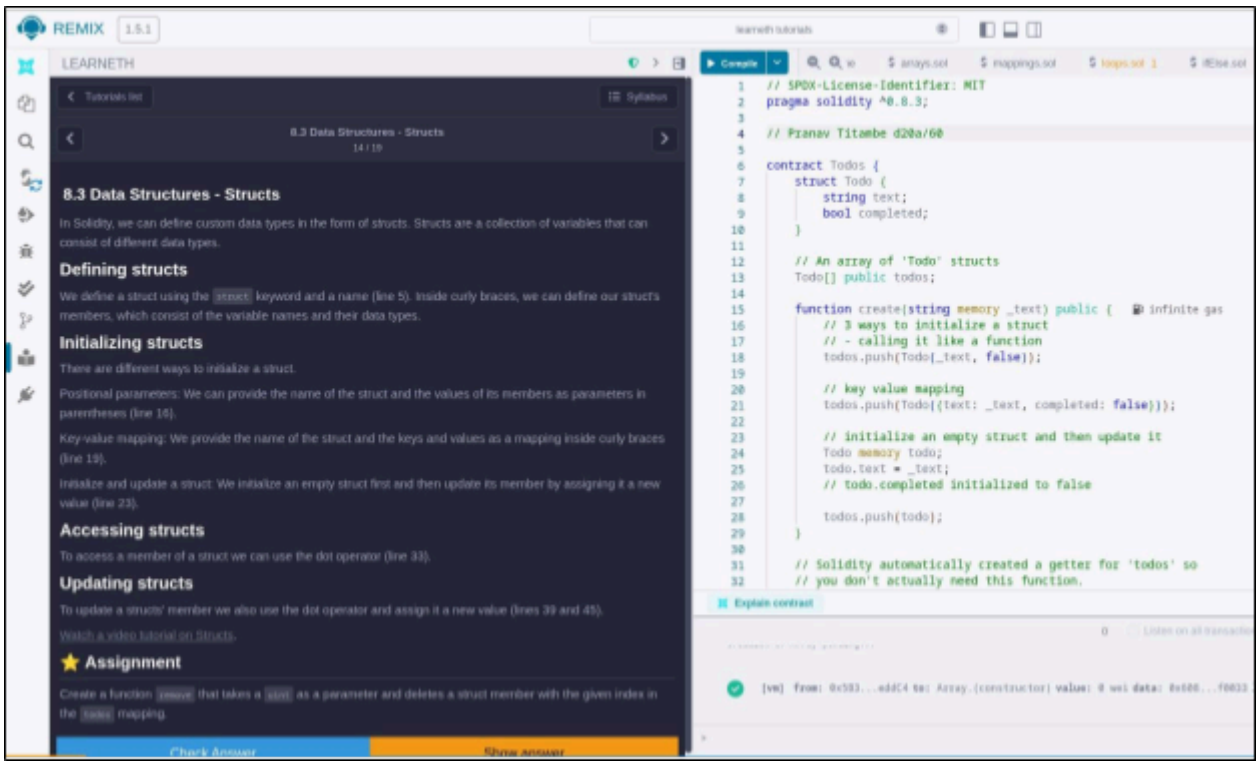
Tutorial 12



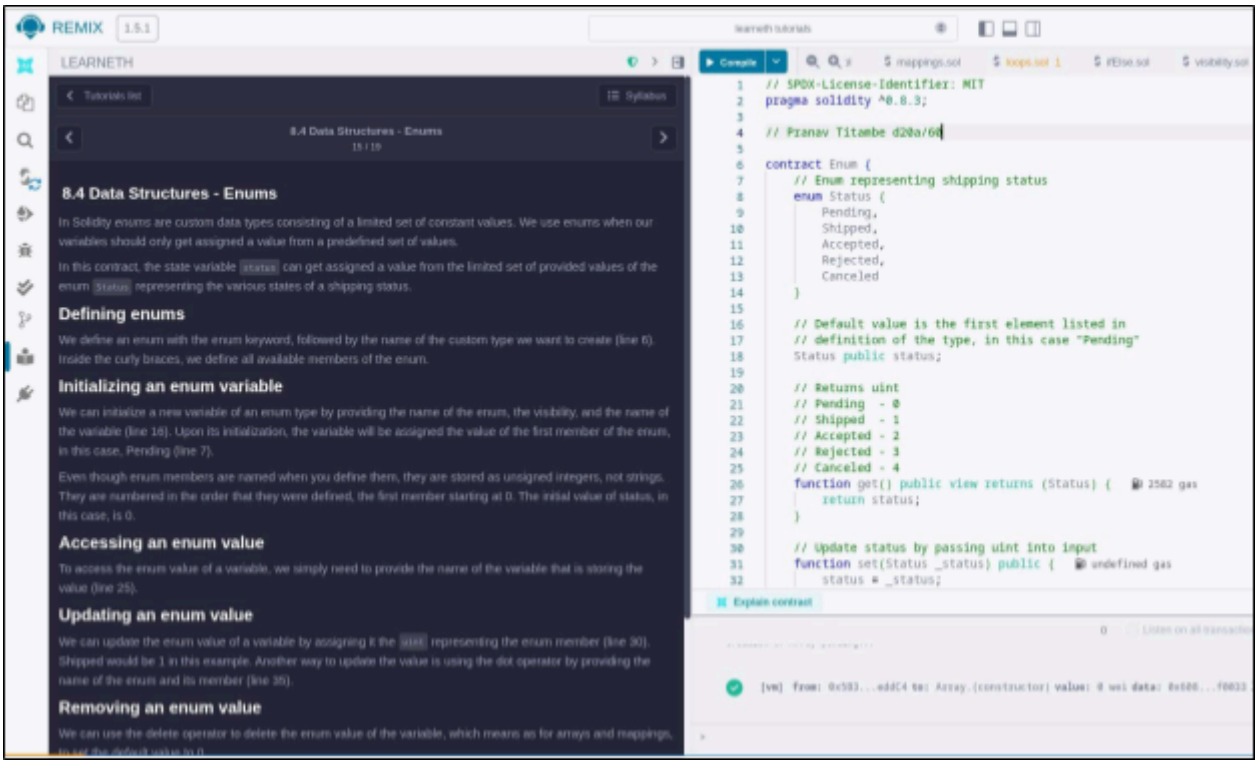
Tutorial 13



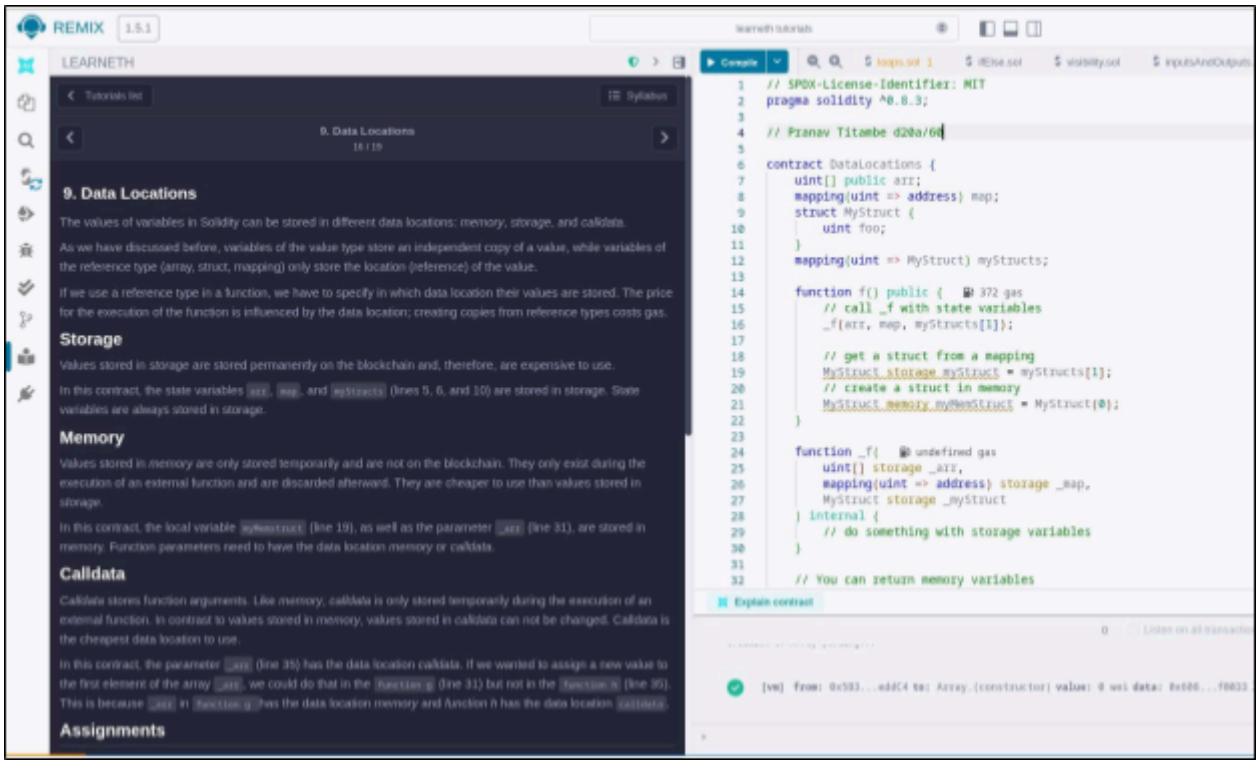
Tutorial 14



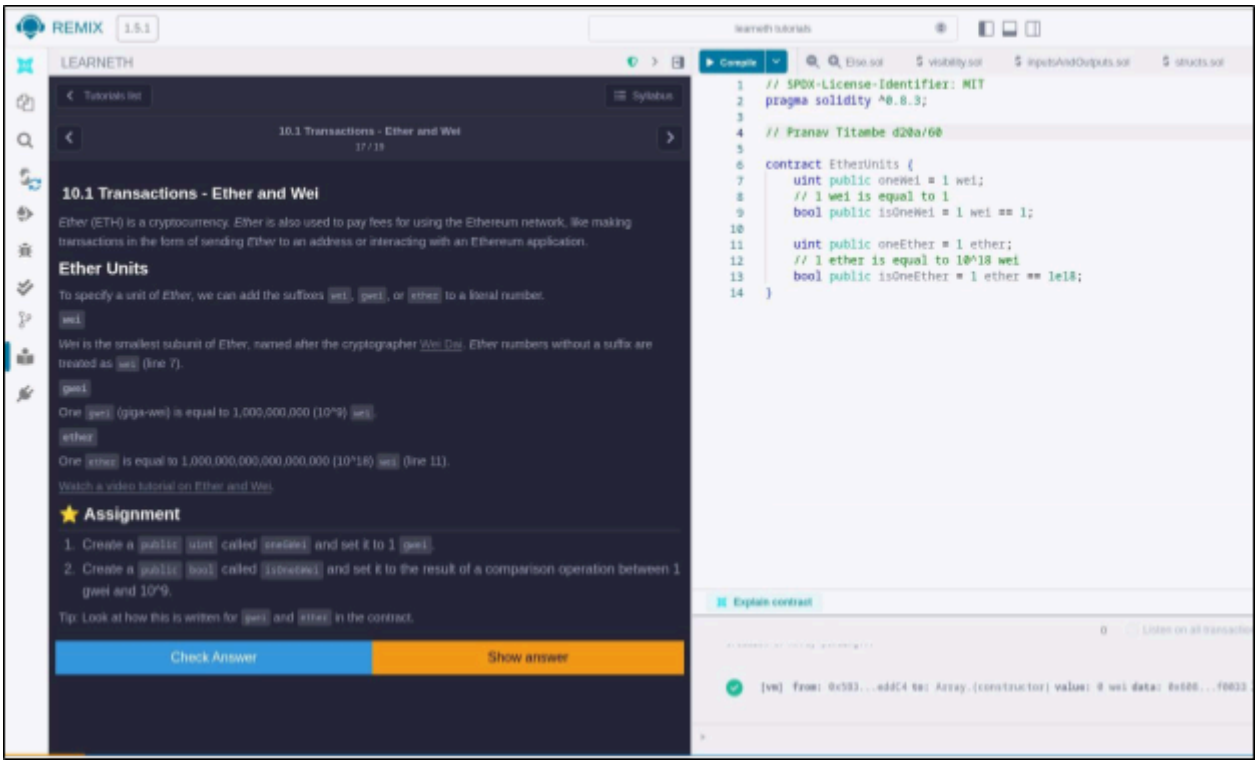
Tutorial 15



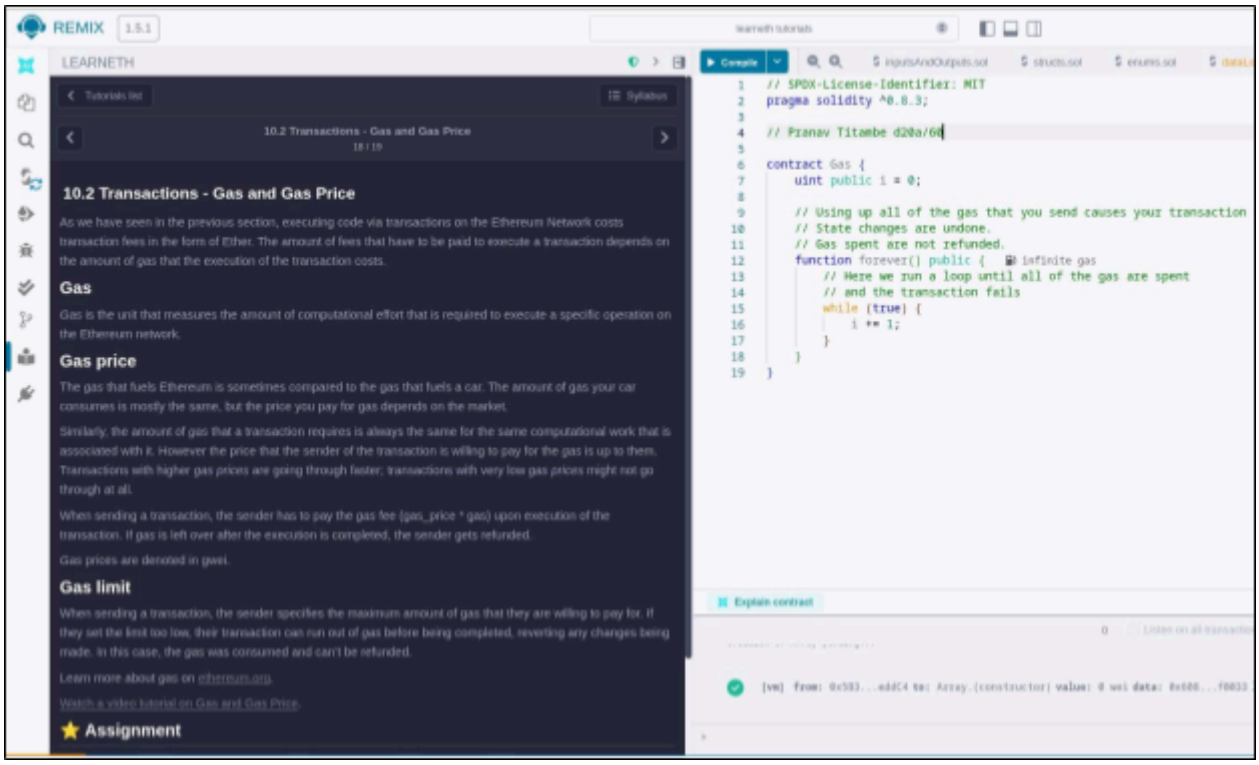
Tutorial 16



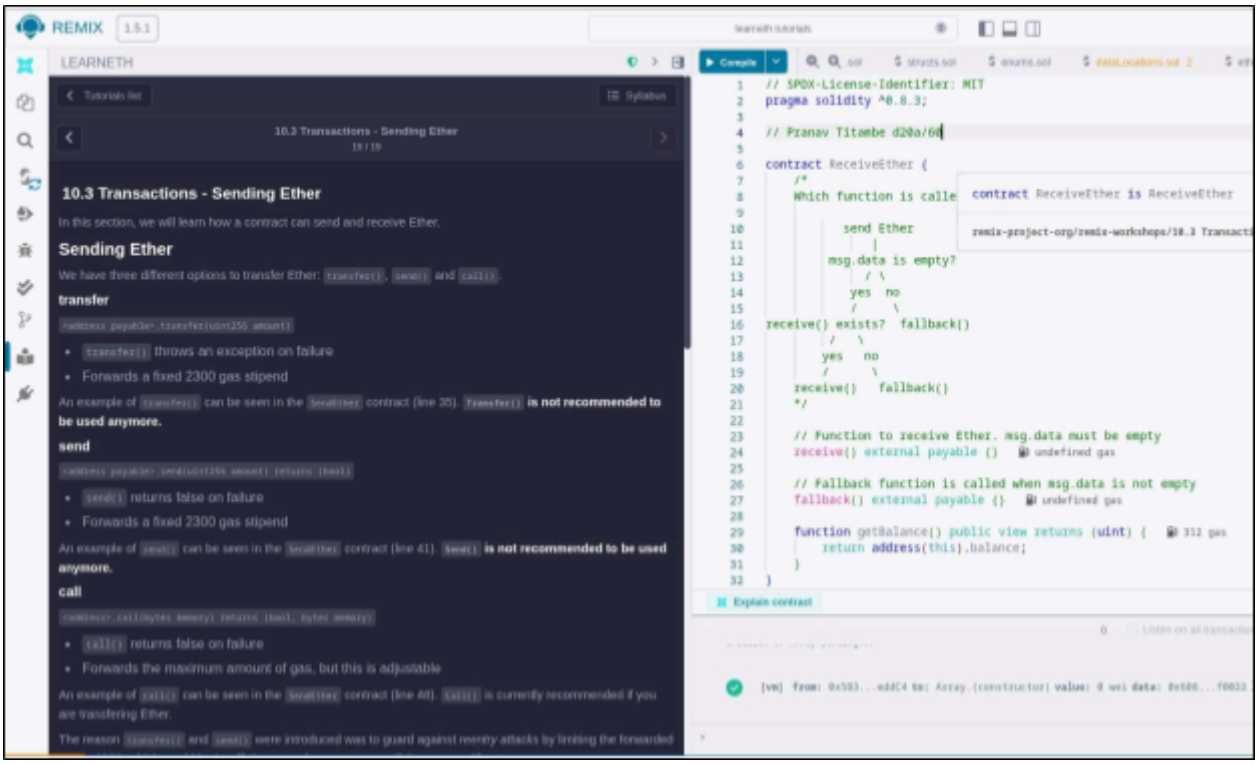
Tutorial 17



Tutorial 18



Tutorial 19



Conclusion:

This experiment provided an in-depth exploration of Solidity programming through structured practical implementation using the Remix IDE. Fundamental concepts—including data types, variable classifications, function definitions, visibility specifiers, modifiers, constructors, control flow mechanisms, data structures, and transaction management—were systematically implemented and evaluated.

The process of designing, compiling, and deploying smart contracts on the Remix Virtual Machine (VM) enabled a comprehensive understanding of smart contract architecture and blockchain execution mechanisms. The practical exposure strengthened conceptual clarity and technical proficiency in Ethereum-based development.