Akruti Dabas D20A 18

Experiment No. 4

- **Aim**: Hands on Solidity Programming Assignments for creating Smart Contracts
- **Lab Objectives**: To explore Blockchain concepts.
- **Lab Outcomes (LO)**: Design Smart Contract using Solidity (LO2)

- **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.

3. **Visibility, Modifiers and Constructors**

- **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).

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

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.
- **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.

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 }.

6. Data Locations

Solidity uses three primary data locations for storing variables:

- **storage**: Data stored permanently on the blockchain. Examples: state variables.
- **memory**: Temporary data storage that exists only while a function is executing. Used for local variables and function inputs.
- **calldata**: A non-modifiable and non-persistent location used for external function parameters. It is gas-efficient compared to memory. Understanding data locations is essential, as they directly impact gas costs and

performance.

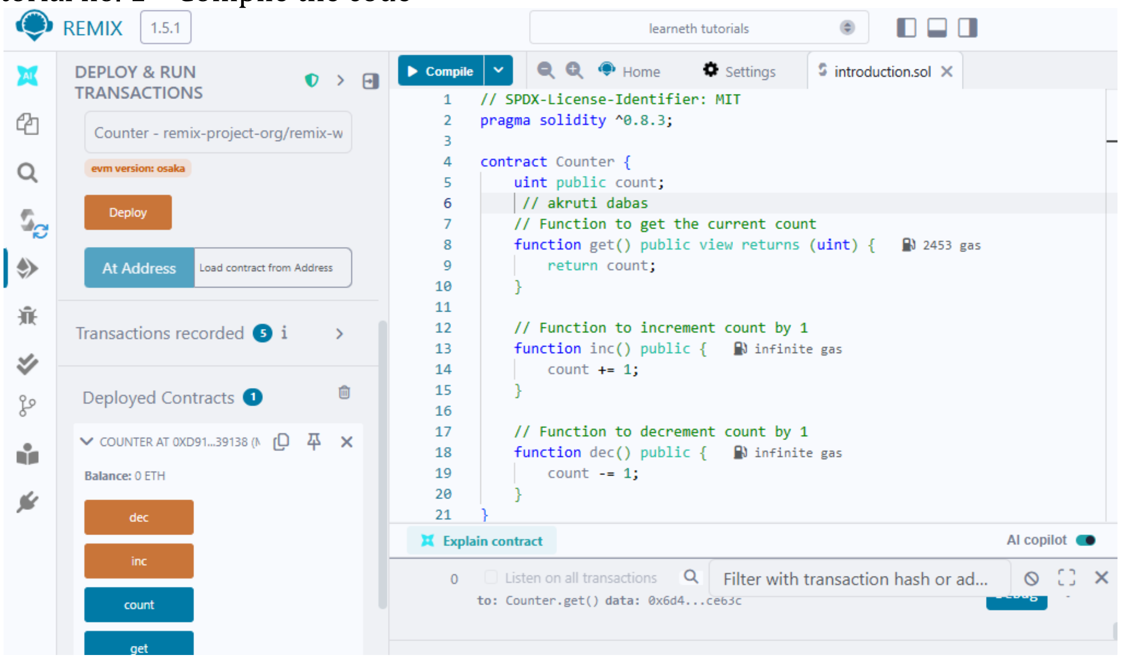
7. **Transactions: Ether and Wei, Gas and Gas Price, Sending Transactions**

- **Ether and Wei**: Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.
- **Gas and Gas Price**: Every transaction consumes gas, which represents computational effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.
- **Sending Transactions**: Transactions are used for transferring Ether or interacting with contracts. Functions like transfer() and send() are commonly used, while call() provides more flexibility. Each transaction requires gas, making efficiency in contract design very important.

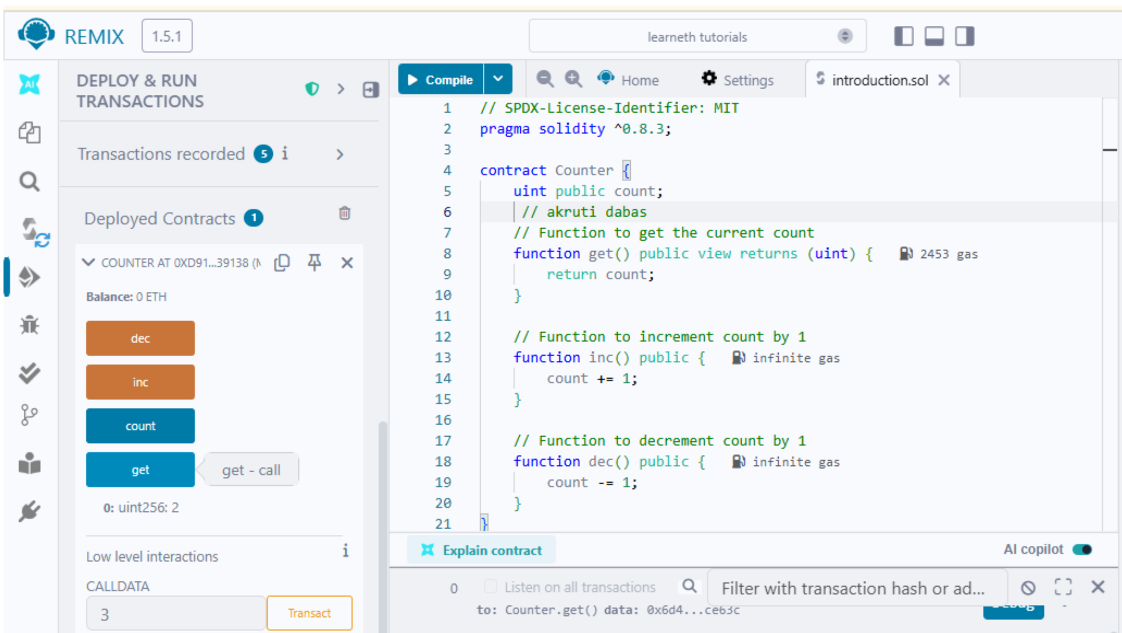
Akruti Dabas D20A 18

➢ **Implementation**:

- Tutorial no. 1 – Compile the code

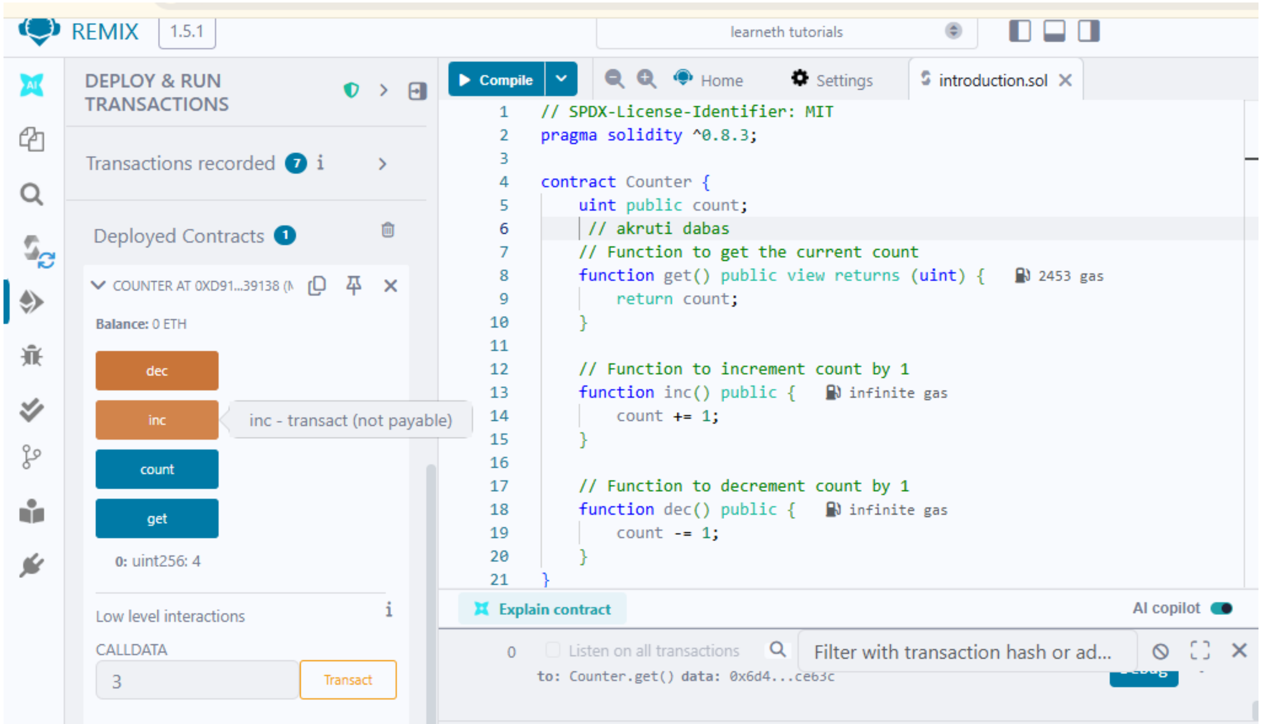


- Tutorial no. 1 – get

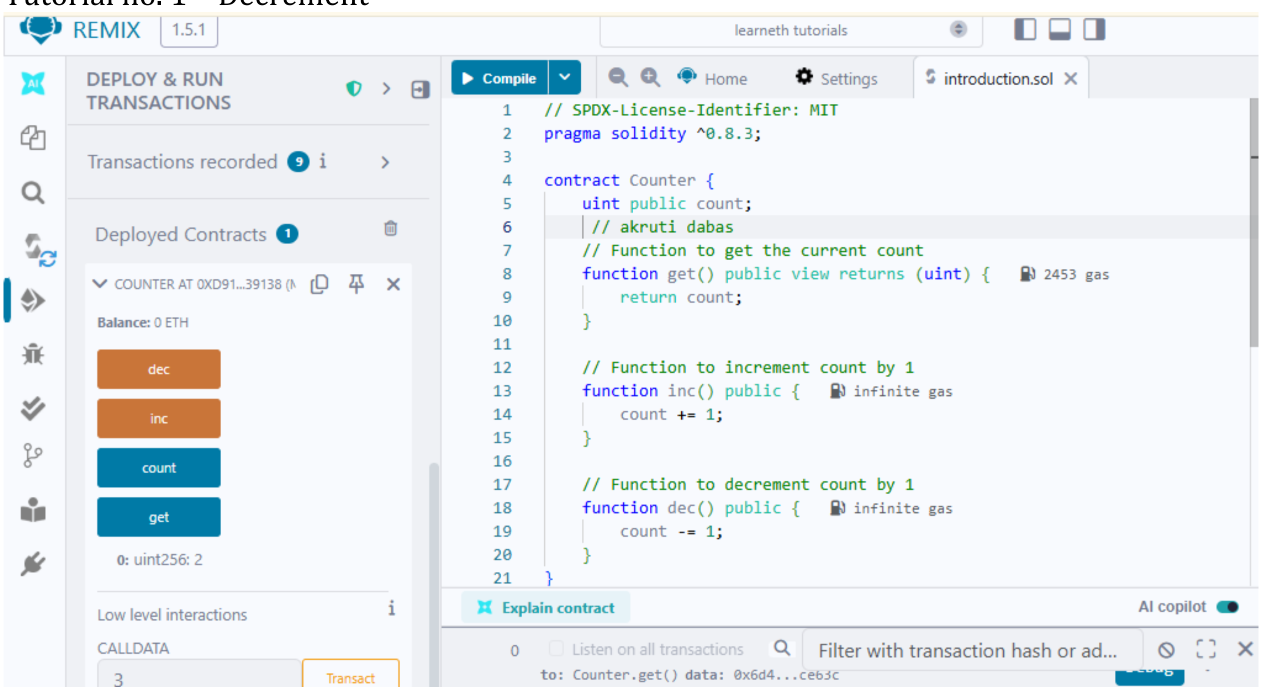


Akruti Dabas D20A 18

- Tutorial no. 1 – Increment

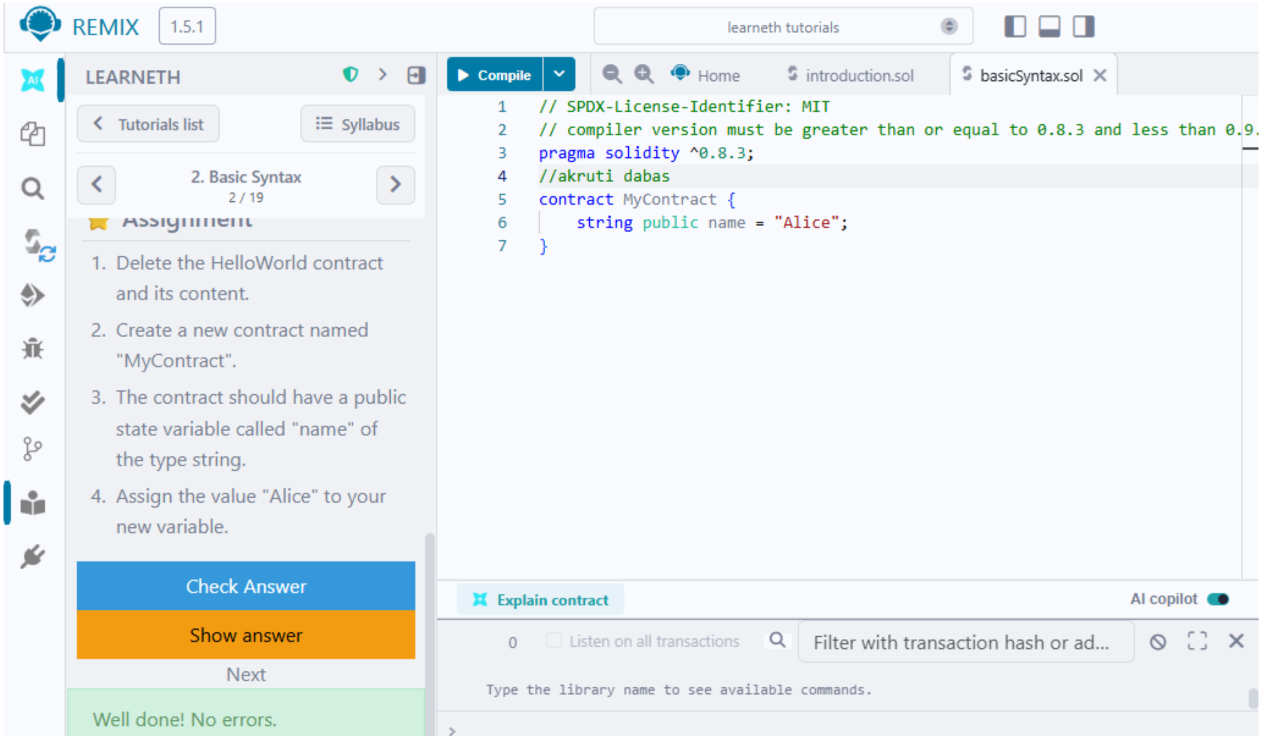


- Tutorial no. 1 – Decrement

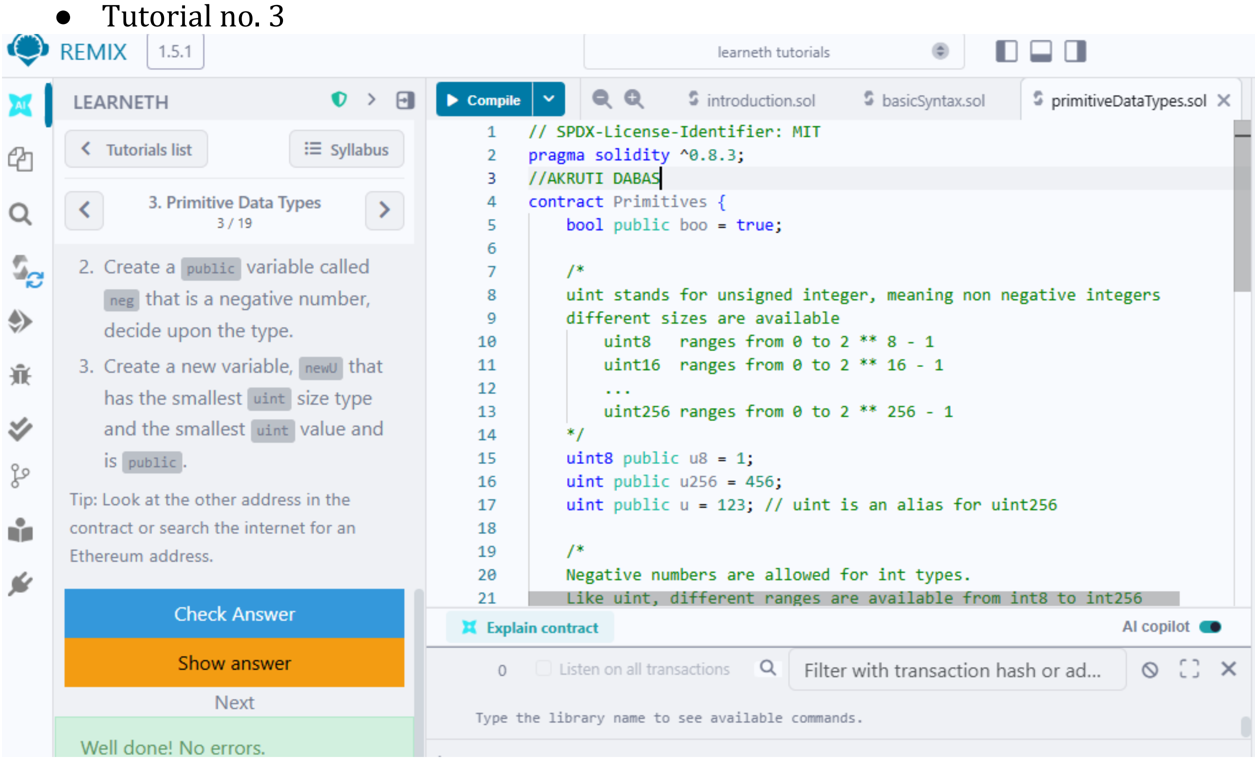


Akruti Dabas D20A 18

- Tutorial no. 2

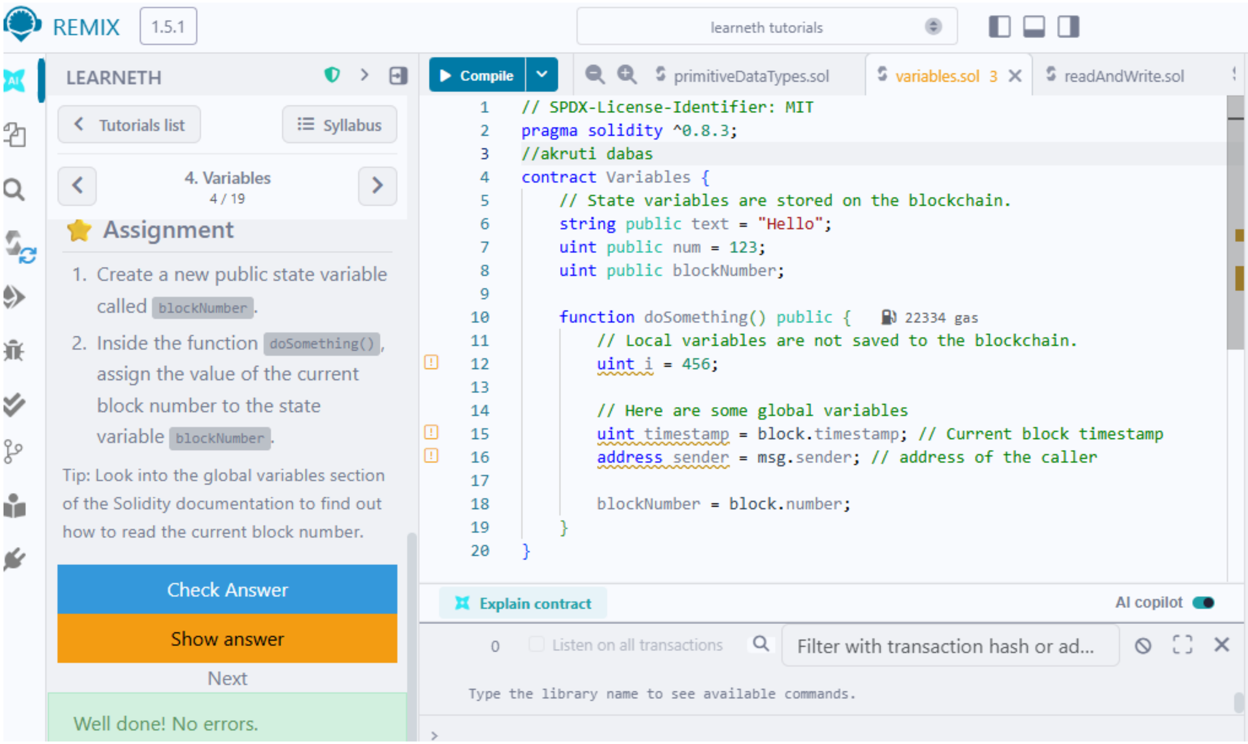


- Tutorial no. 3

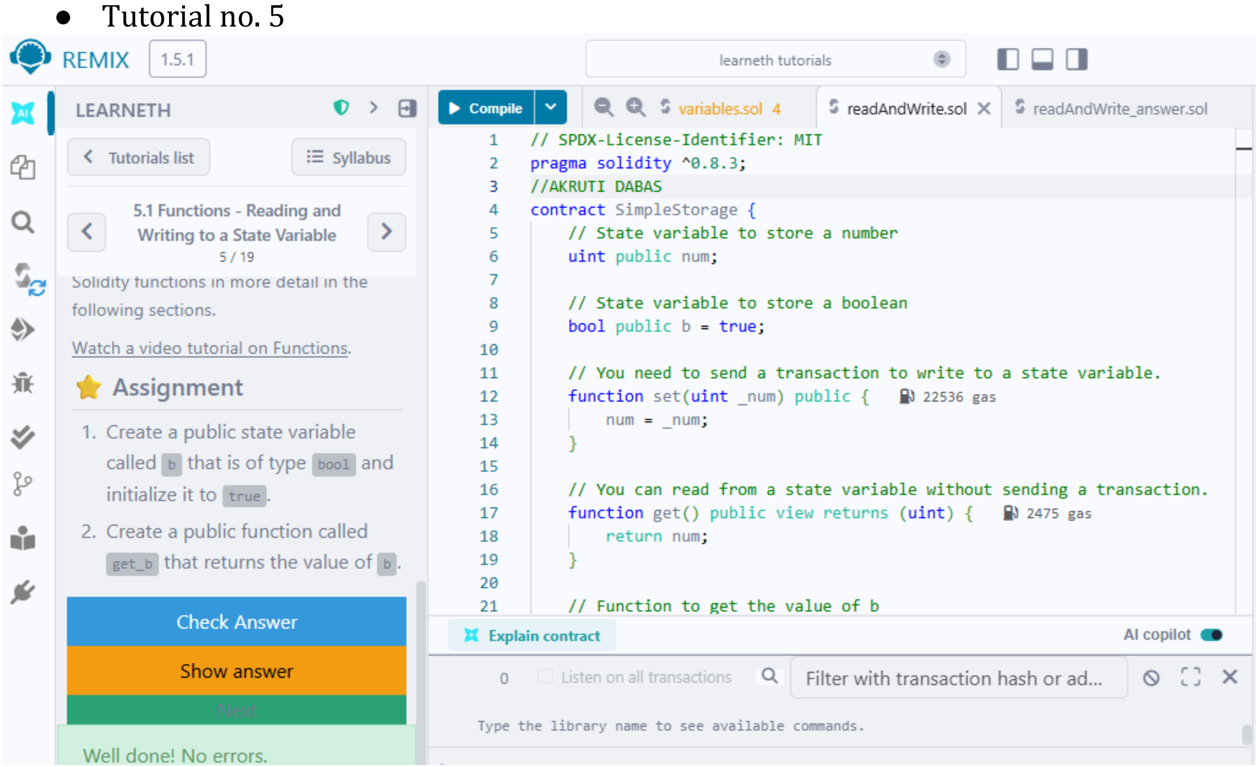


Akruti Dabas D20A 18

- Tutorial no. 4

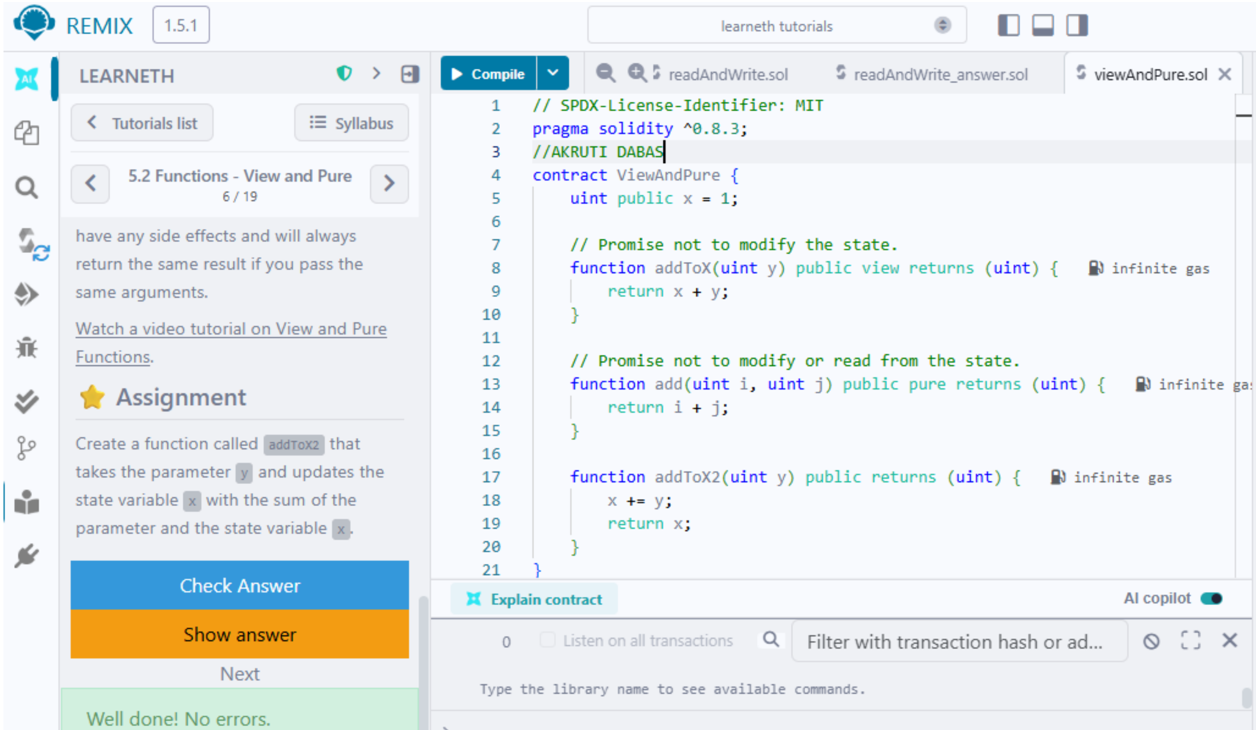


- Tutorial no. 5

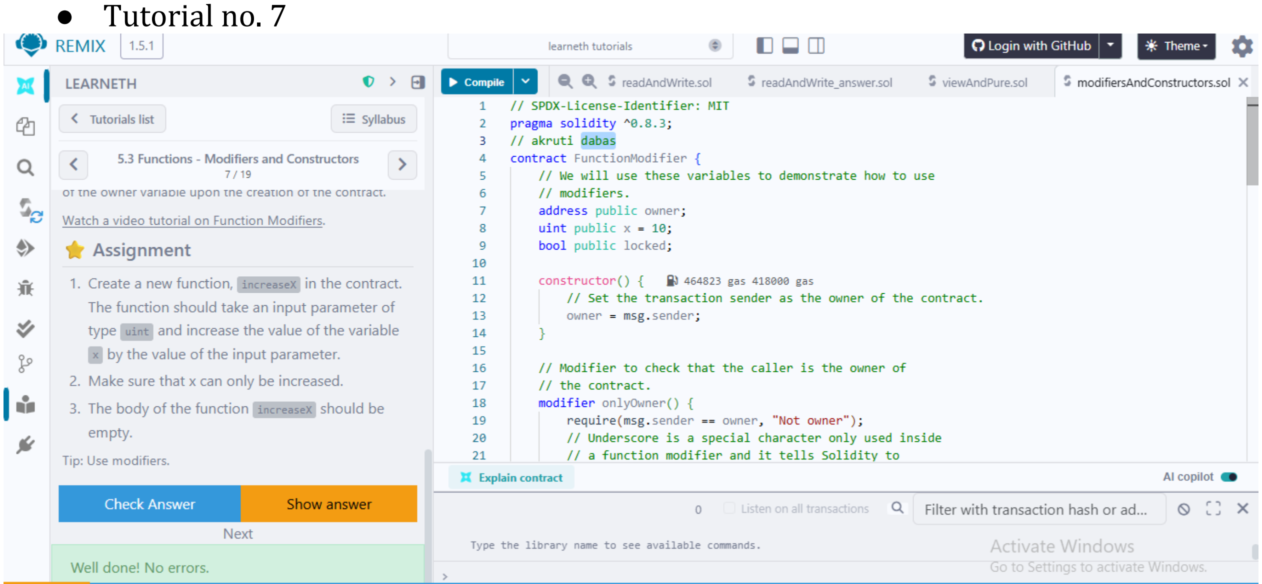


Akruti Dabas D20A 18

- Tutorial no. 6

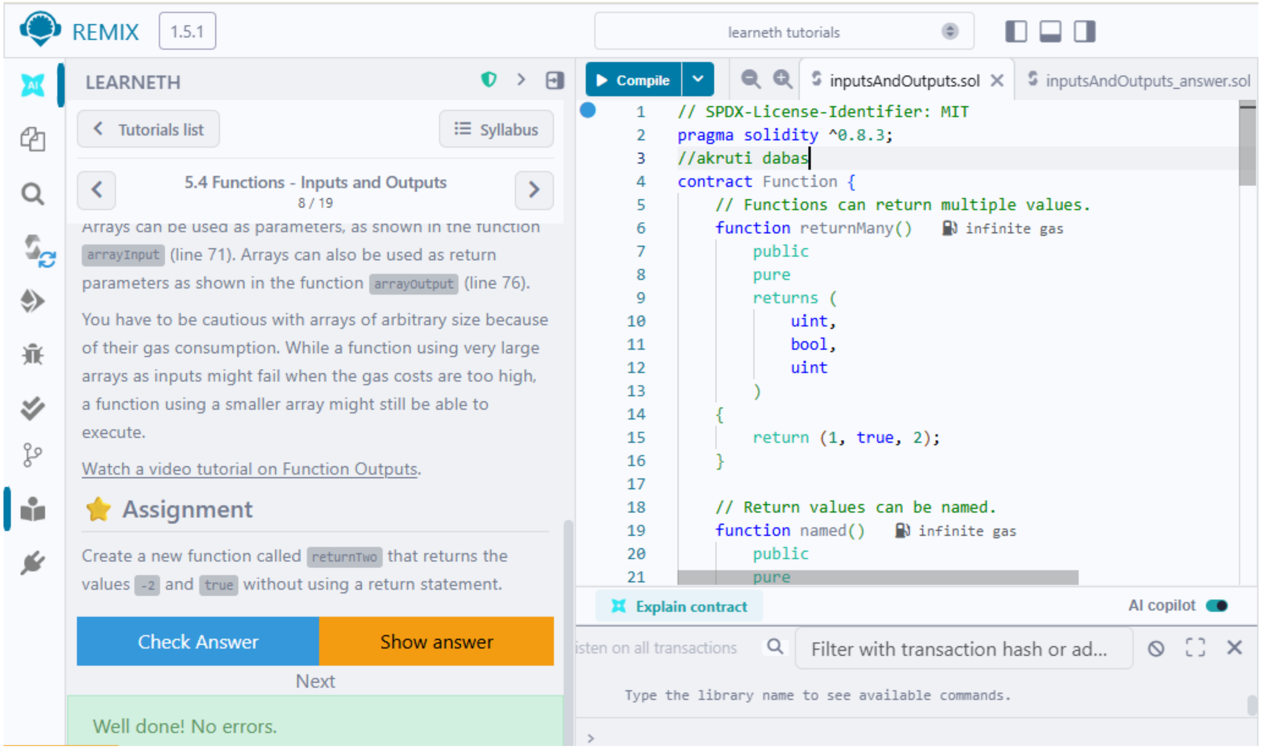


- Tutorial no. 7

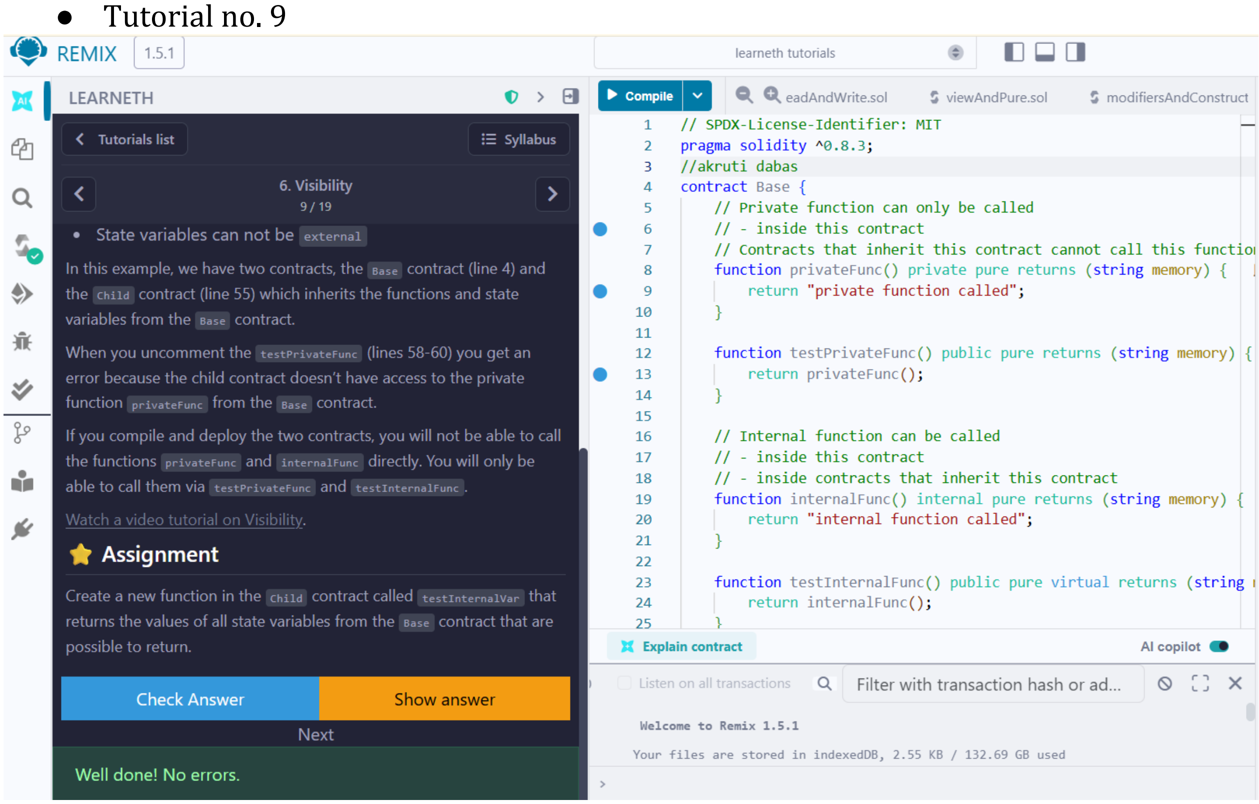


Akruti Dabas D20A 18

- Tutorial no. 8

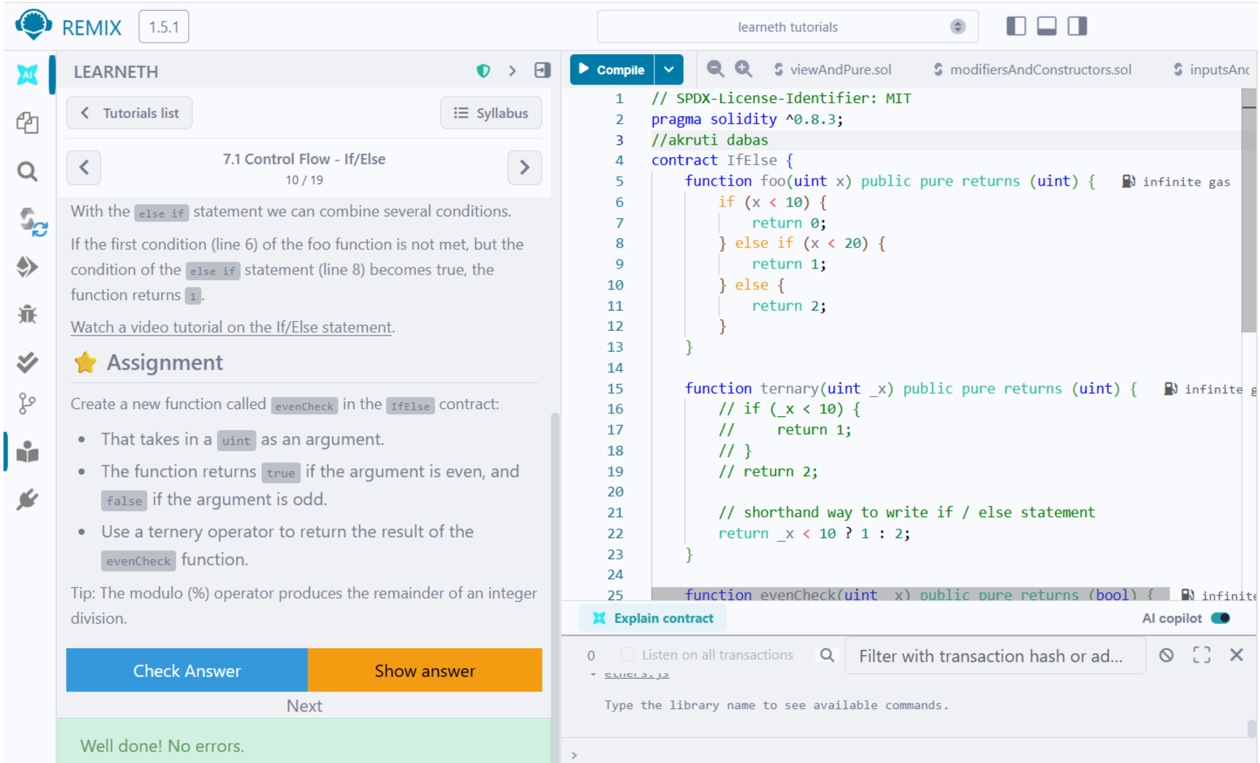


- Tutorial no. 9

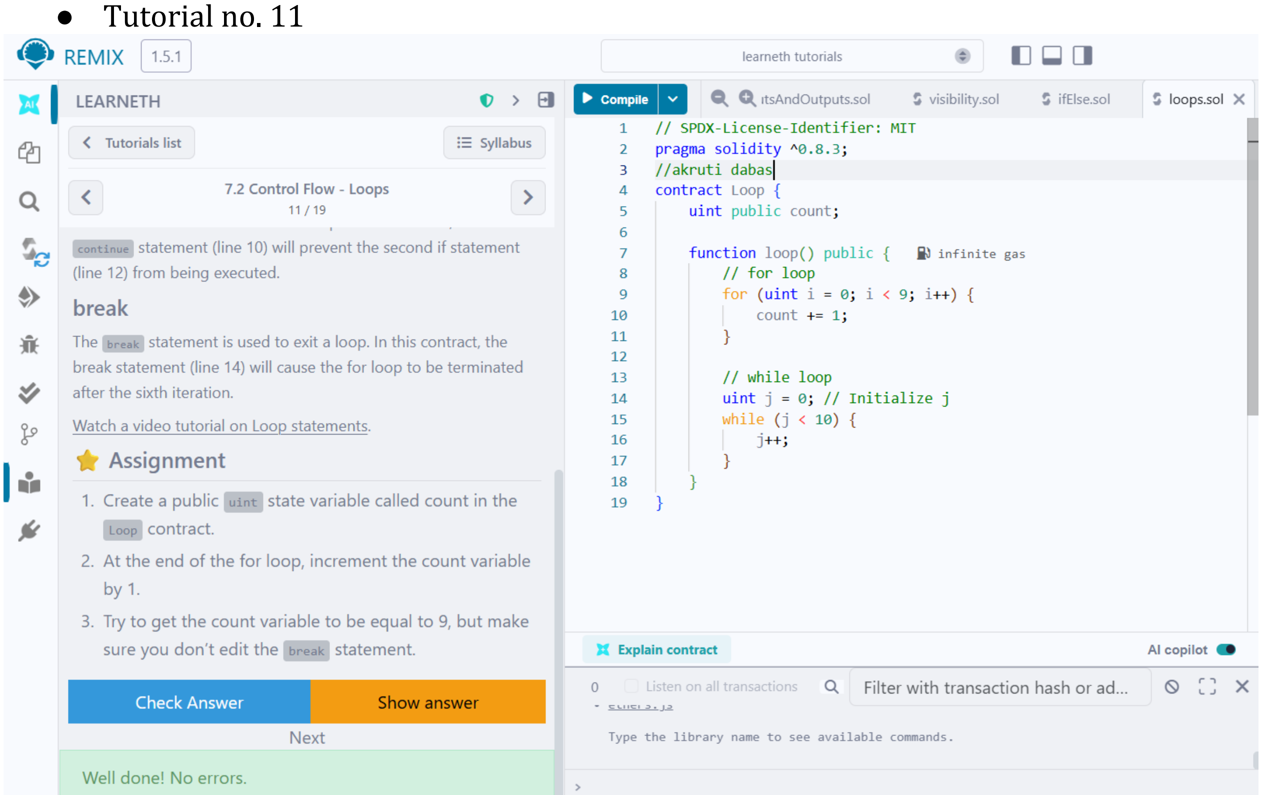


Akruti Dabas D20A 18

- Tutorial no. 10

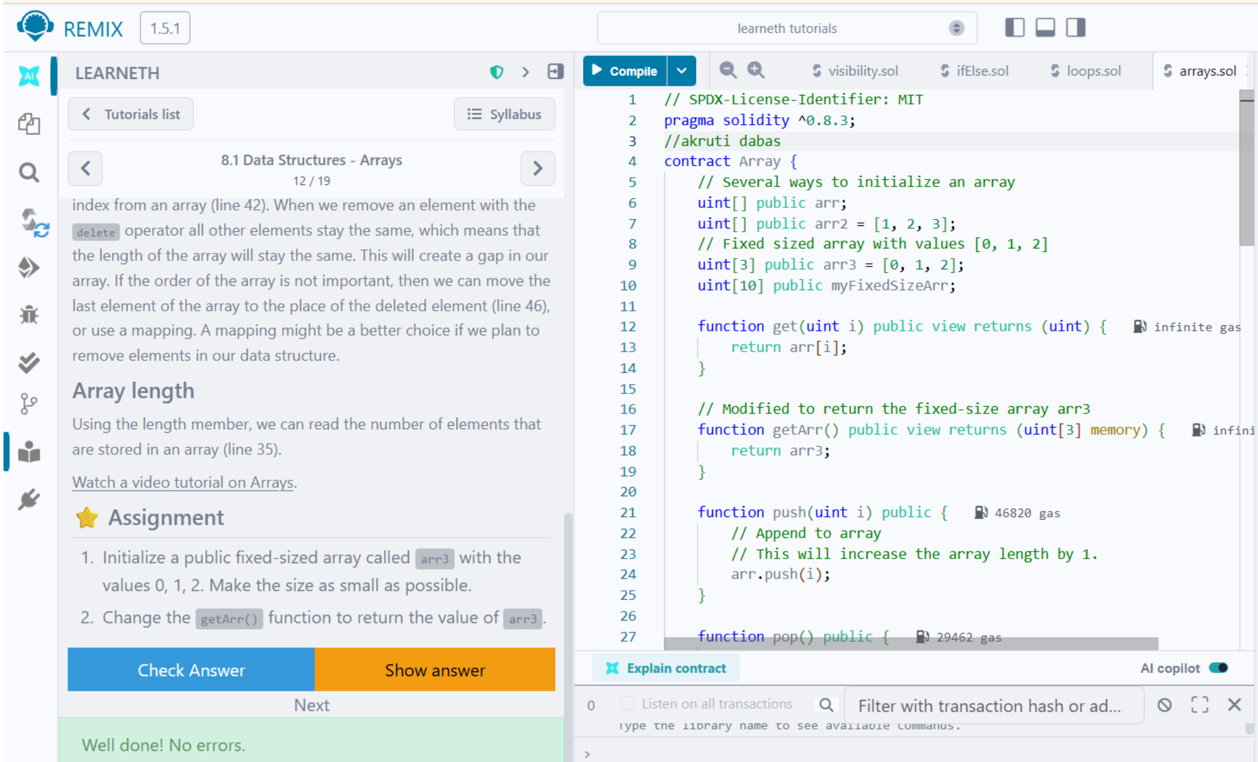


- Tutorial no. 11

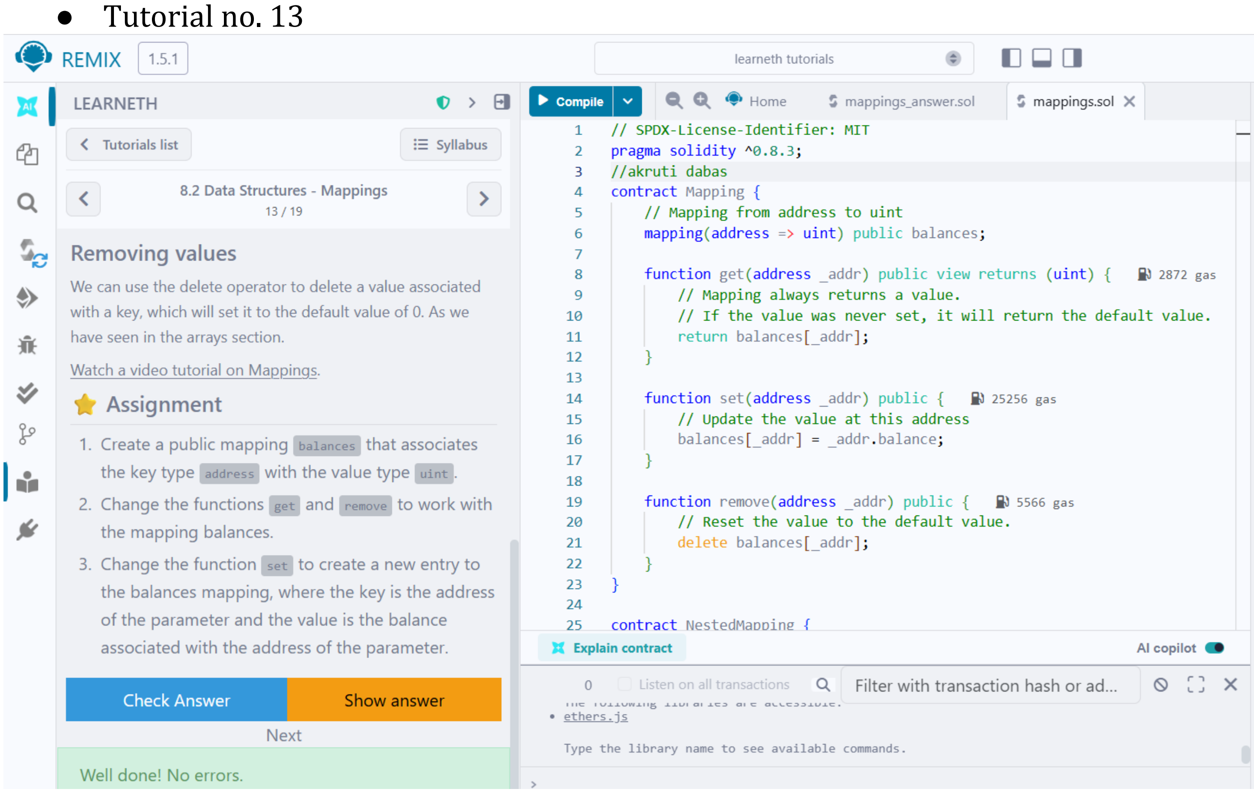


Akruti Dabas D20A 18

- Tutorial no. 12

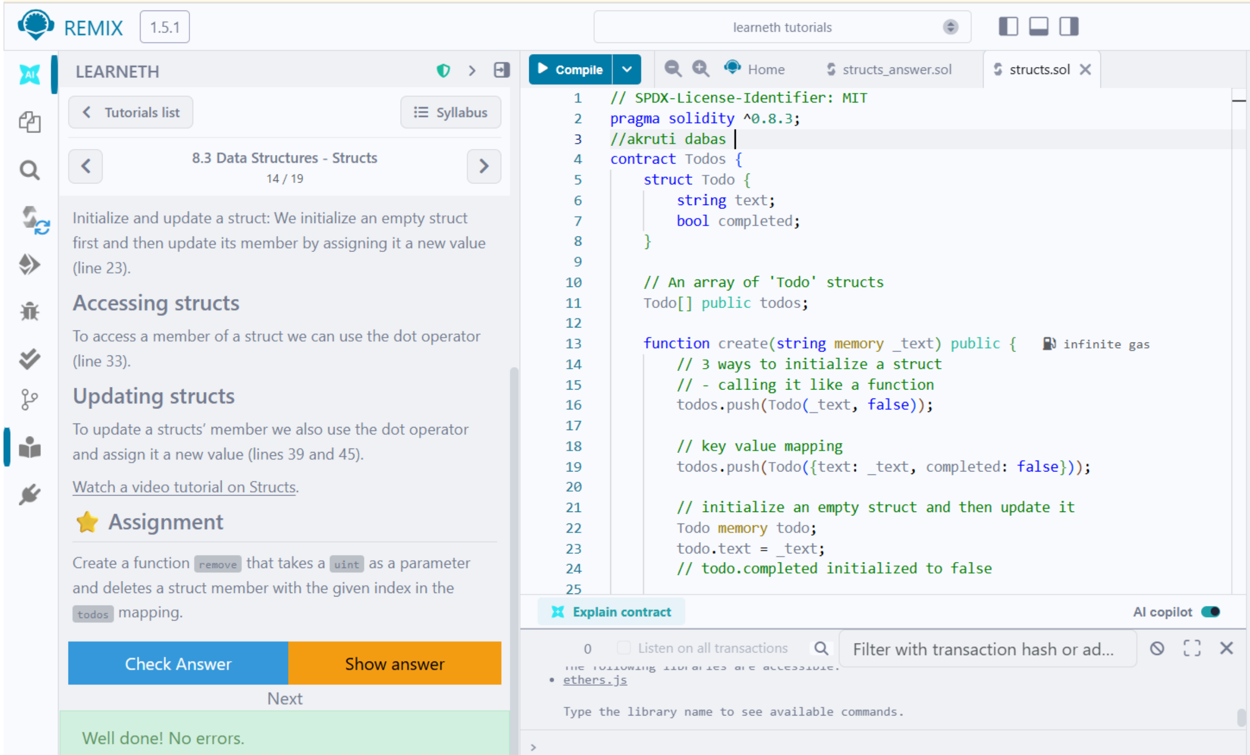


- Tutorial no. 13

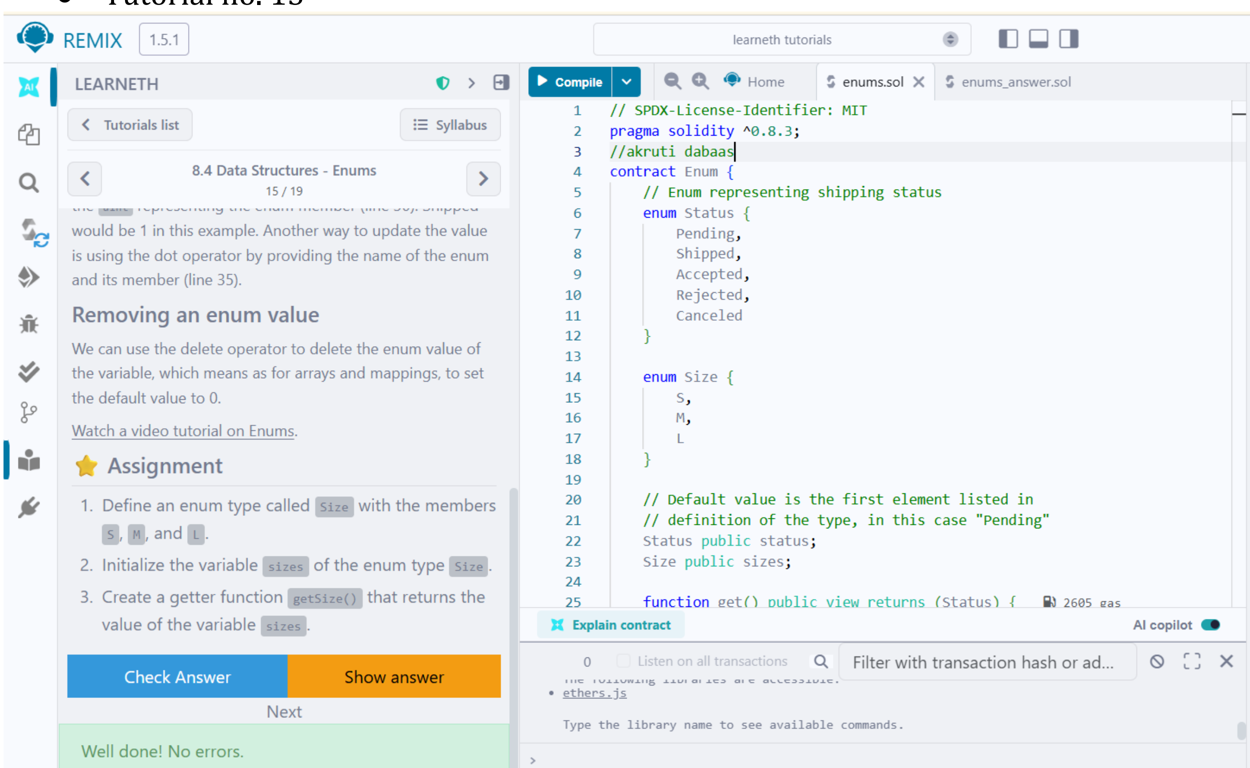


Akruti Dabas D20A 18

- Tutorial no. 14

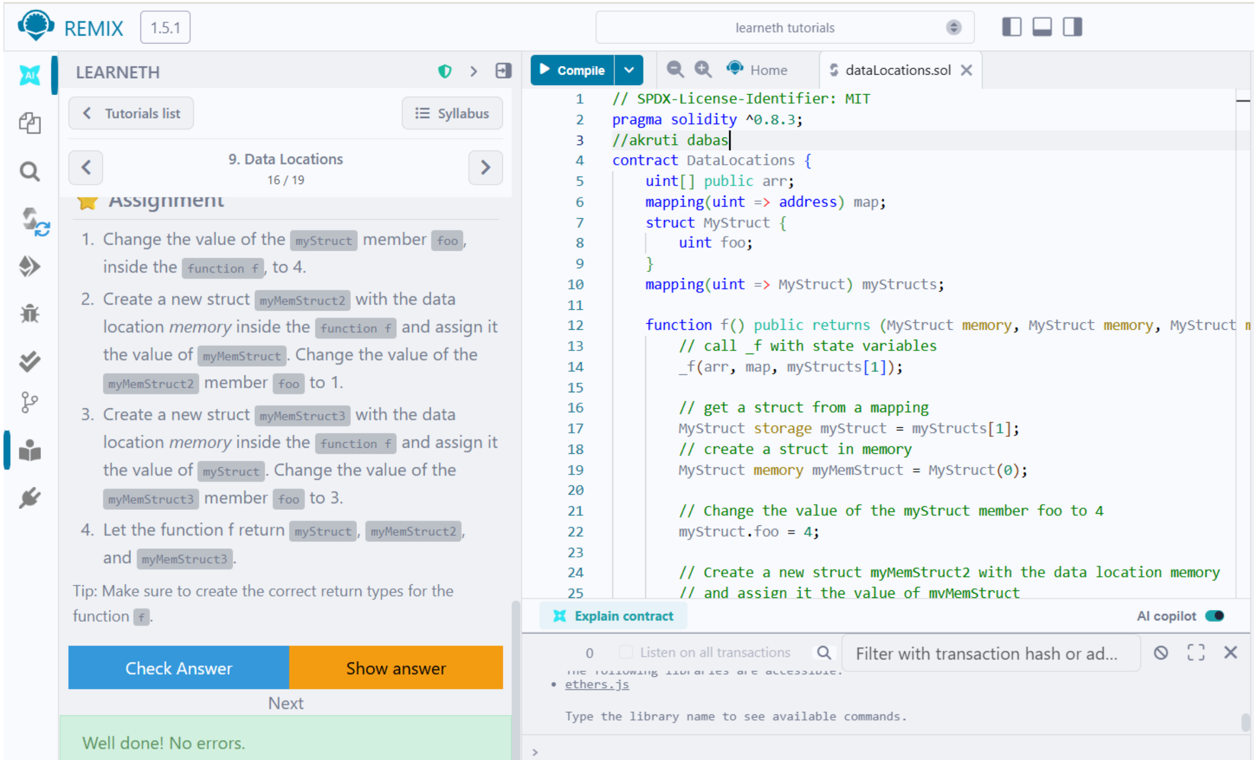


- Tutorial no. 15

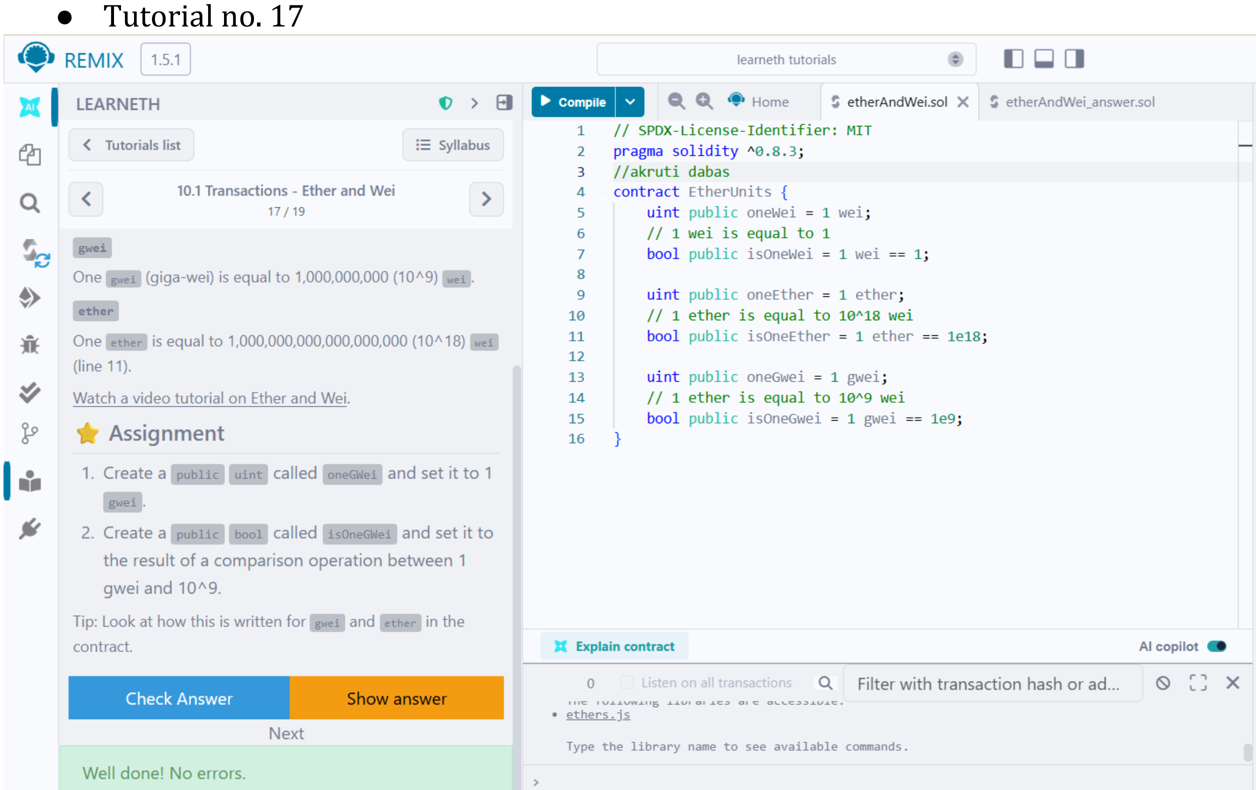


Akruti Dabas D20A 18

- Tutorial no. 16

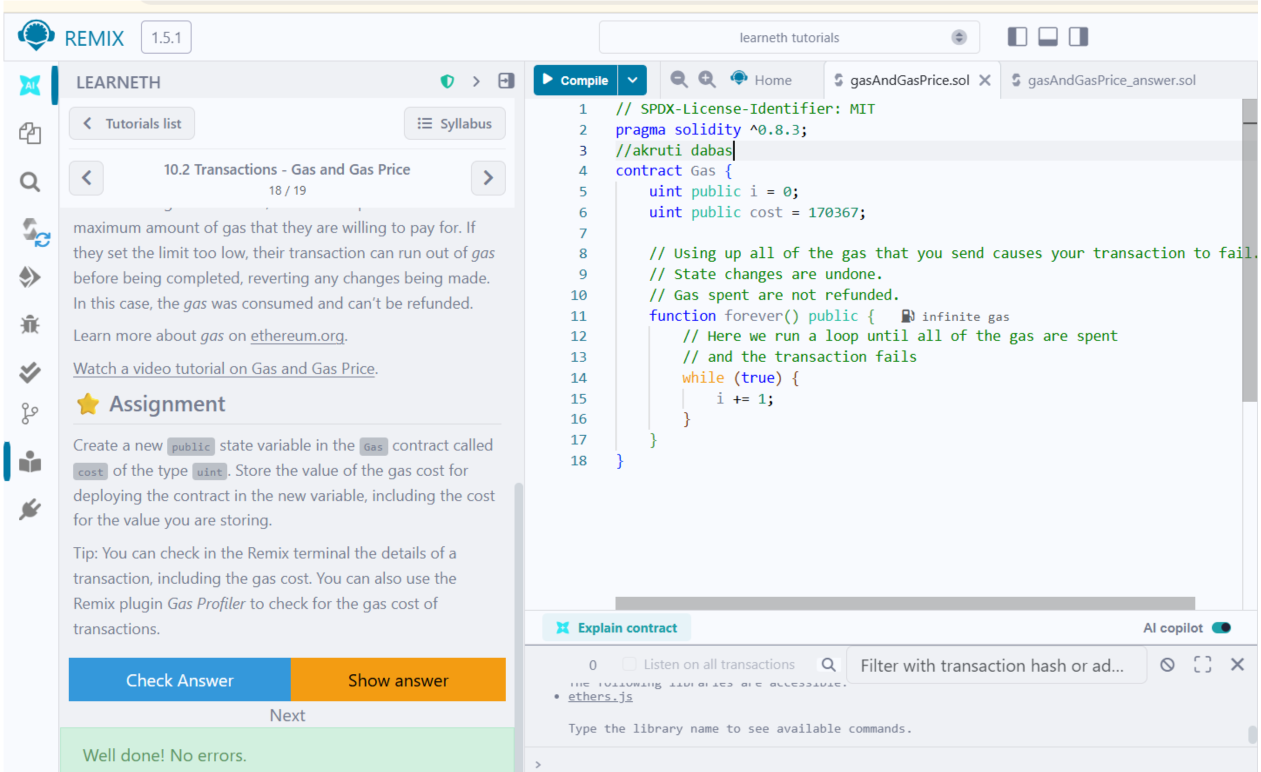


- Tutorial no. 17

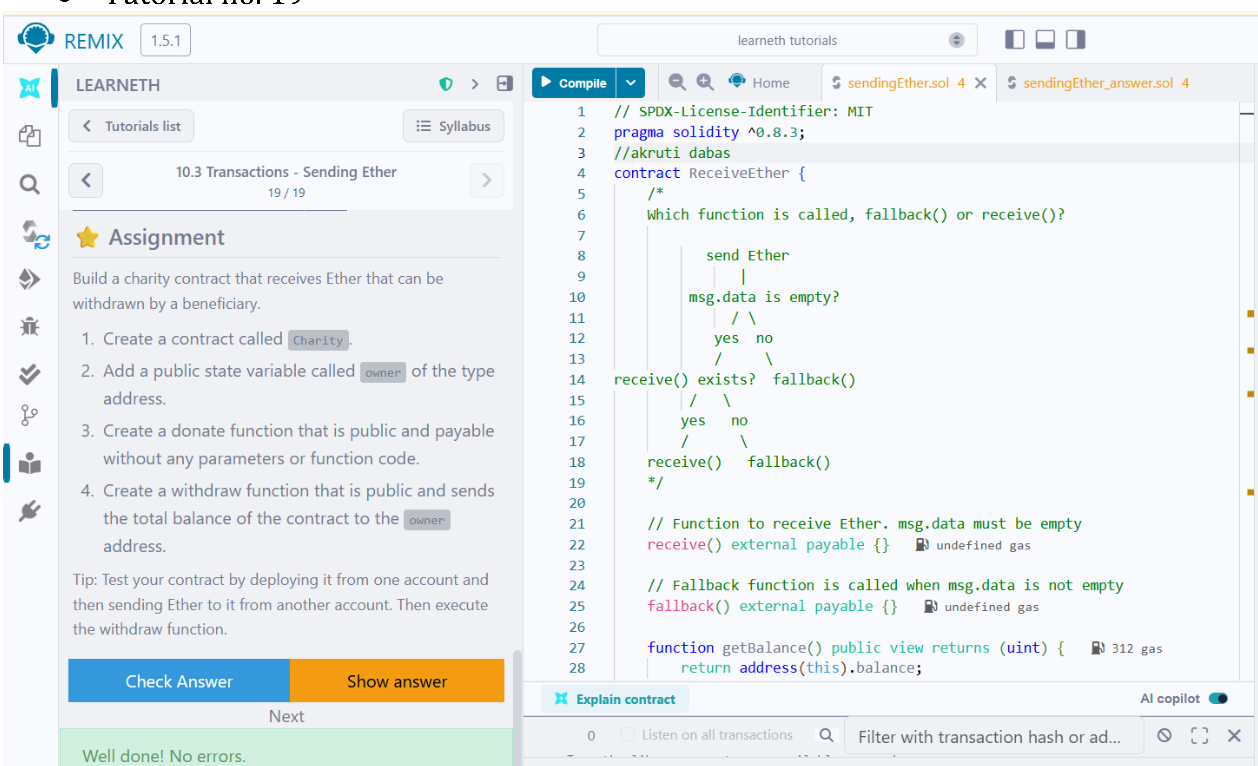


Akruti Dabas D20A 18

- Tutorial no. 18



- Tutorial no. 19



- **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.