# EXPERIMENT NO.4

**Name:**Vaishnal Mali
**Class**:D20A
**Roll No**:32
**Batch**:B

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

**Theory:-**
**1. Primitive Data Types, Variables, Functions - pure, view**

**Primitive (Value) Data Types** in Solidity include:

- bool → true/false
- int / uint (signed/unsigned integers, e.g., uint256 is most common)
- address → 20-byte Ethereum address
- bytes1 to bytes32 → fixed-size byte arrays
- string → dynamic UTF-8 encoded text (reference type, but often grouped here)

**Variables** are declared with a type and can be:

- **State variables** → stored permanently on the blockchain (expensive)
- **Local variables** → exist only during function execution

**Functions** can be marked as:

- pure → does not read or write state (computes only from inputs; cheapest gas)
- view → reads state but does not modify it (e.g., getters; no gas when called externally)

Example

```
function getResult() public view returns (uint product, uint sum) {
   product = num1 * num2;  // reads state
   sum = num1 + num2;
}
```

```
function pureCalc(uint a, uint b) public pure returns (uint) {
    return a + b;  // no state access
}
```

**2. Inputs and Outputs to Functions**

Functions in Solidity can take **parameters** (inputs) and return **values** (outputs).

- **Inputs**: Declared in parentheses; can use data locations like memory or calldata for reference types.
- **Outputs**: Declared after returns keyword; can return multiple values.

Example:

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

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

- Use calldata for external calls (cheaper, read-only).
- Use memory for temporary copies inside functions.

**3. Visibility, Modifiers and Constructors**

**Visibility Specifiers** (for functions and state variables):

- **public** → anyone can call/read (default for state vars creates getter)
- **private** → only inside current contract
- **internal** → current + derived (child) contracts

- **external** → only external calls (cheaper for large data)

**Modifiers** → reusable code blocks that run before/after function body (e.g., access control).

Use _; to insert function body.

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}

function restricted() public onlyOwner { ... }
```

**Constructors** → special function that runs once on deployment (initializes state).

- Syntax: constructor() { ... } (older: same name as contract)

**4. Control Flow: if-else, loops**

Solidity supports standard control structures:

**if-else**:

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

}

**Loops**:

- for (uint i = 0; i < 10; i++) { ... }
- while (condition) { ... }
- do { ... } while (condition);

Avoid unbounded loops (gas limit risk). Use break / continue when needed.

**5. Data Structures: Arrays, Mappings, structs, enums**

- **Arrays**
  - Fixed: uint[5] arr;
  - Dynamic: uint[] arr; (use .push(), .pop(), .length)
- **Mappings** → key-value store (like hash table)

  mapping(address => uint) public balances;

  Keys can be most types; no iteration possible.

**Structs** → custom composite types

```
struct User {
   address wallet;
   uint balance;
   bool active;
}
User public owner;
```

**Enums** → named constants (integers under the hood)

```
enum Status { Pending, Active, Cancelled }
Status public state = Status.Pending;
```

**6. Data Locations**

Solidity has three main **data locations** for reference types (arrays, structs, mappings, strings):

- **storage** → permanent blockchain storage (persistent, expensive) Default for state variables.
- **memory** → temporary, function lifetime only (deleted after execution) Cheap; used for local variables & function args/returns.
- **calldata** → read-only, non-modifiable area for function call data Cheapest for external function parameters (immutable copy of tx data).

Rule:

- State vars → always storage
- Function args → prefer calldata (external) or memory
- Local reference vars → must specify location

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

- **Ether units** → smallest is **wei** (1 ETH = $10^{18}$ wei) Other: gwei ($10^{9}$ wei), commonly used for gas prices.
- **Gas** → computational effort unit
    - **Gas Limit** → max gas willing to spend (set by sender)
    - **Gas Price** → price per gas unit (in wei/gwei; set by sender)
    - Total fee = gas used × gas price
- **msg.value** → amount of wei sent with transaction
- **Sending Ether** → use .transfer(), .send(), or .call{value: amount}("") (recommended low-level call)

Example:

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

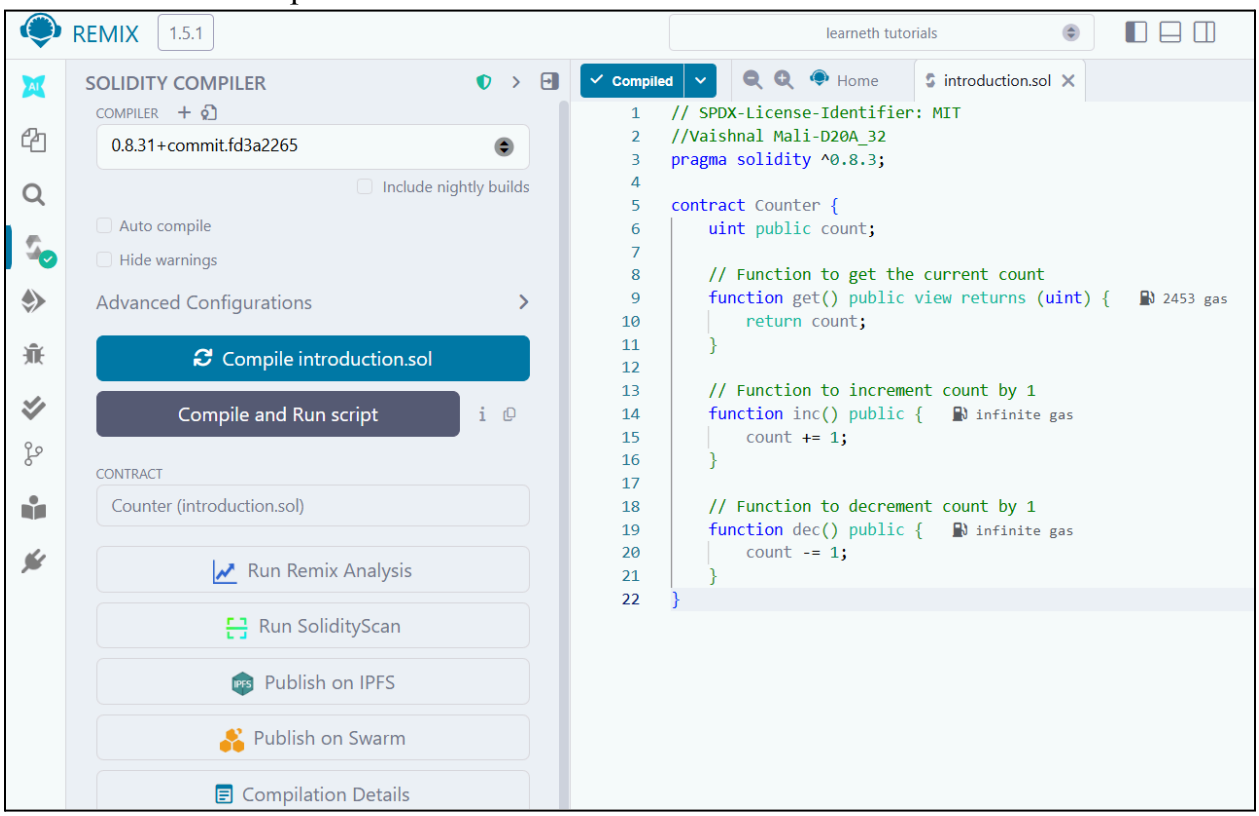
recipient.transfer(msg.value);

}

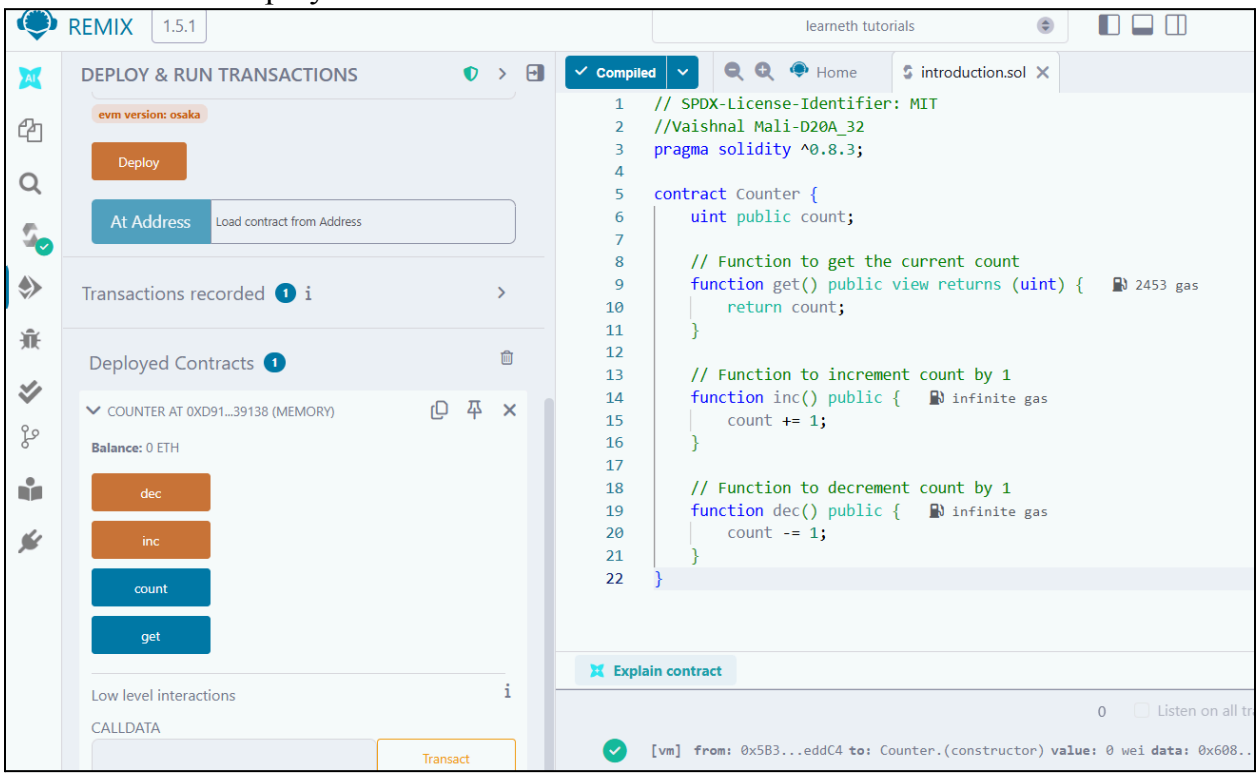
- Use payable for addresses/functions that receive Ether.
- tx.gasprice → current gas price (global var)

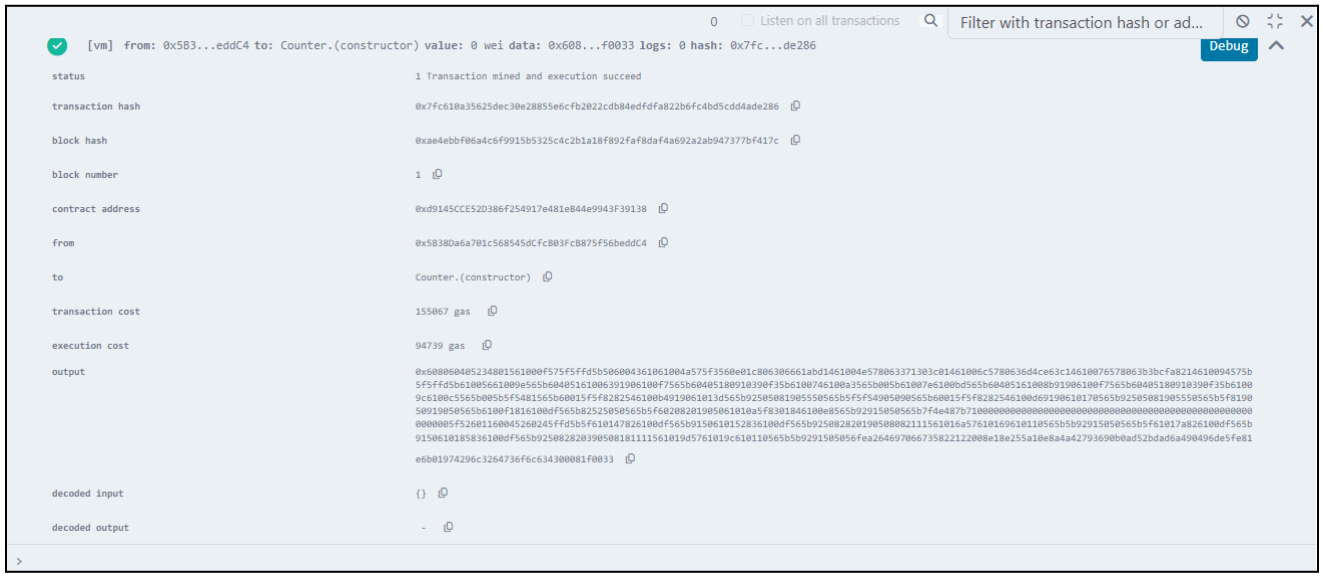
**Implementation:-**

- Tutorial no. 1 – Compile the code

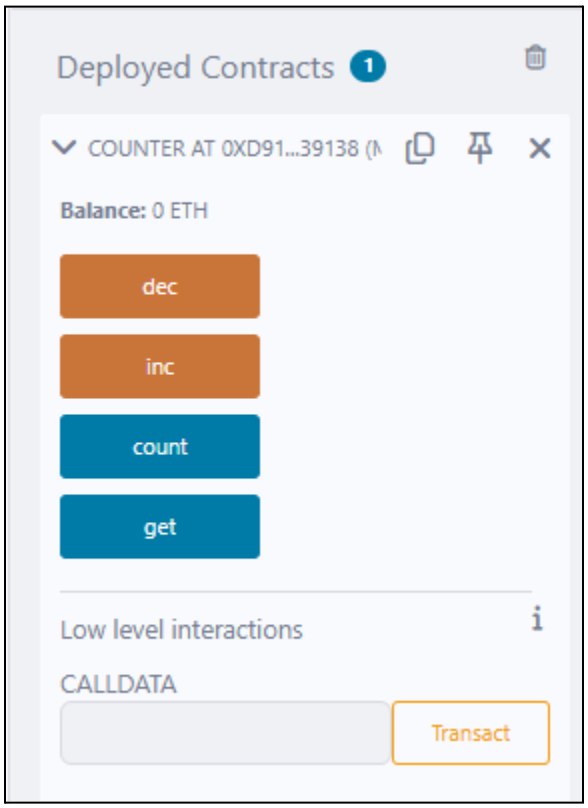


- Tutorial No.1-Deploy the contract

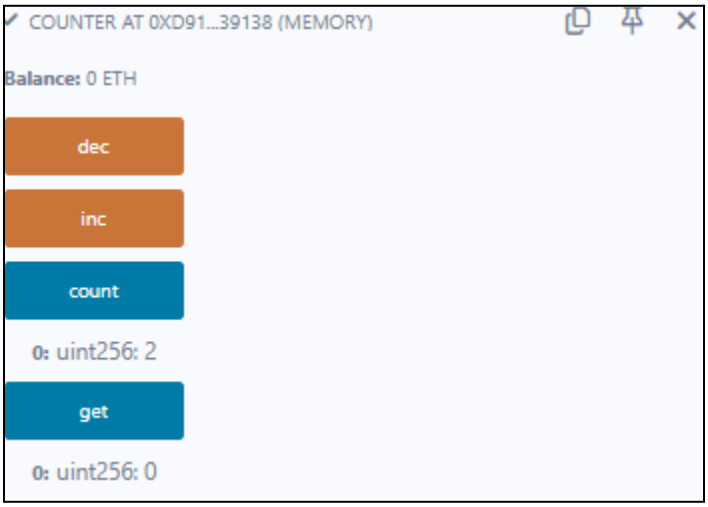


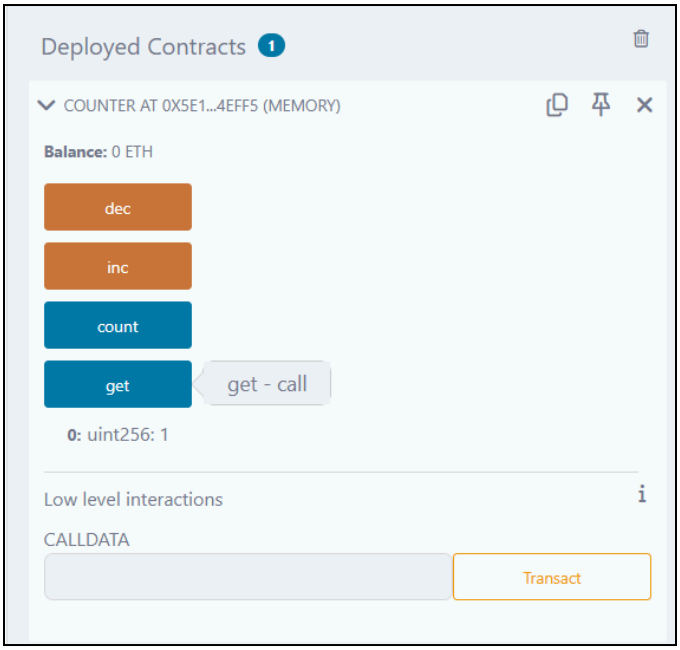
- Tutorial no. 1 – get

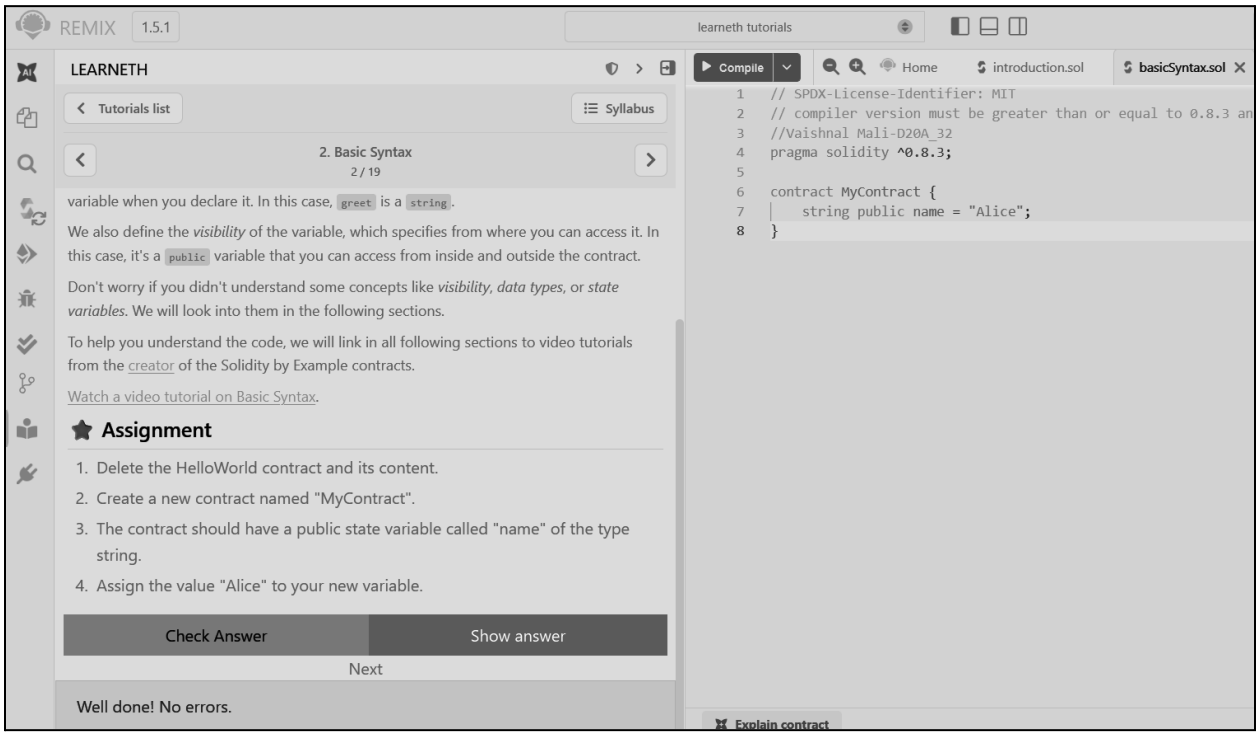
- Tutorial no. 1 – Increment



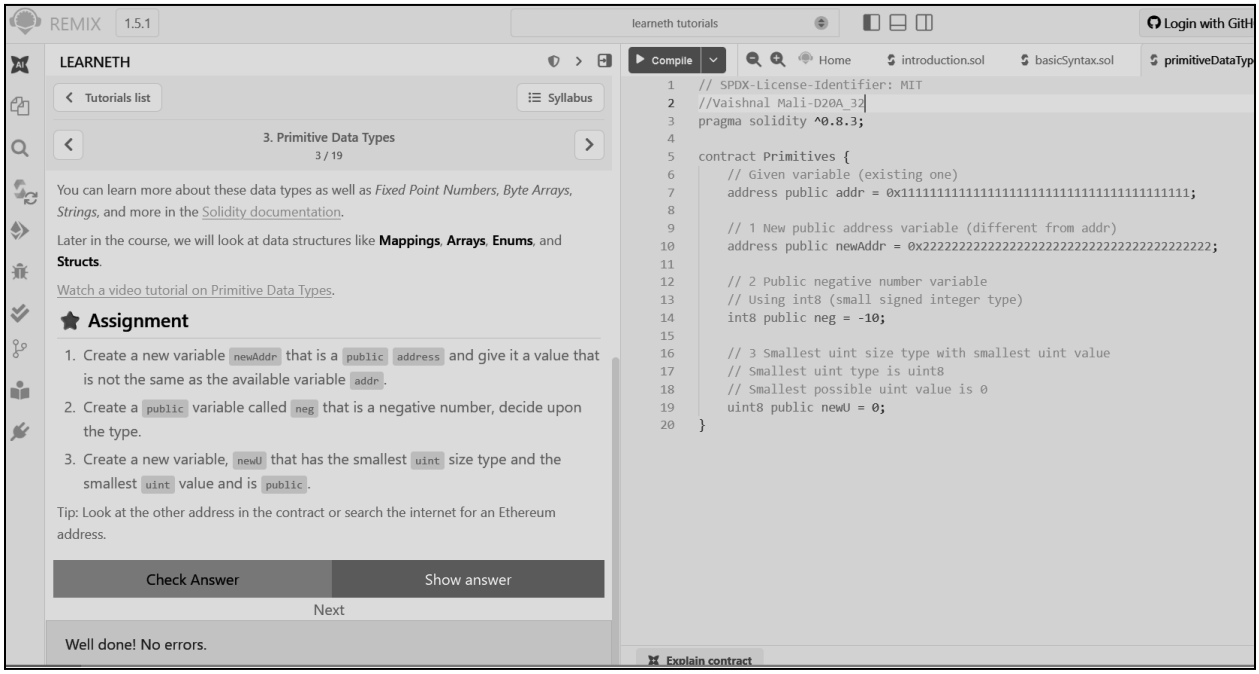

- Tutorial no. 1 – Decrement



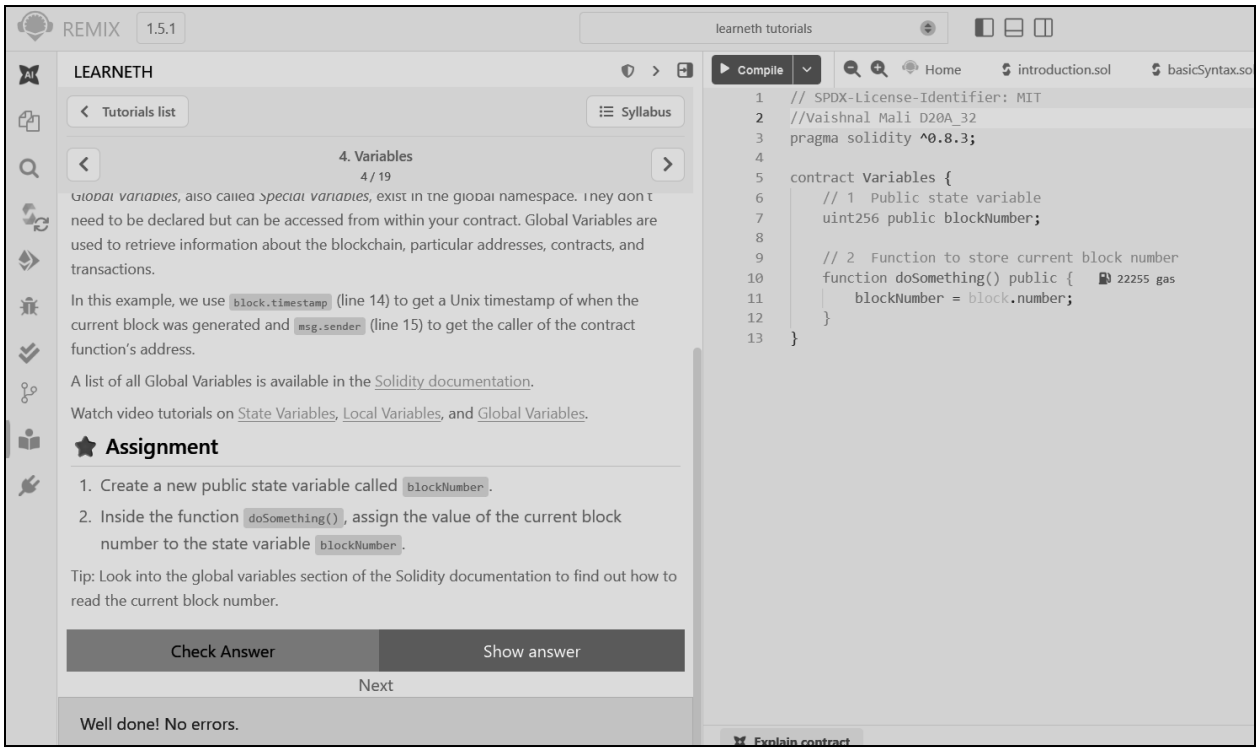
- Tutorial no.2



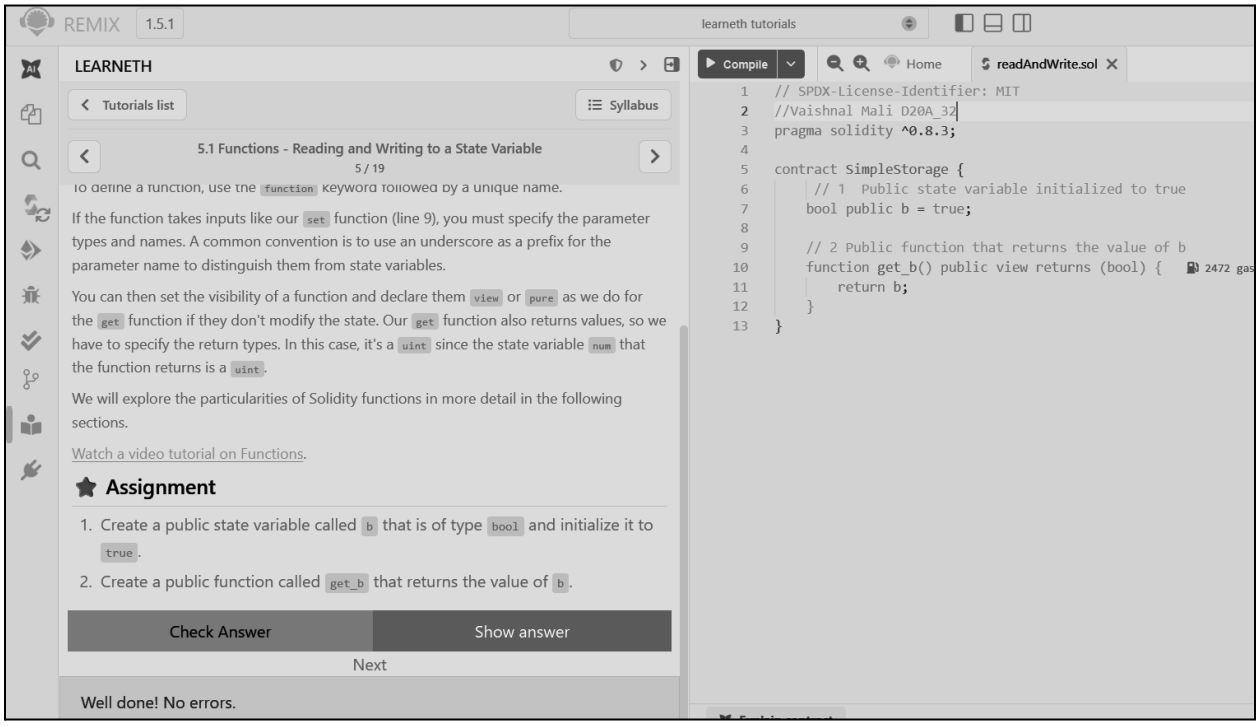

- Tutorial no.3



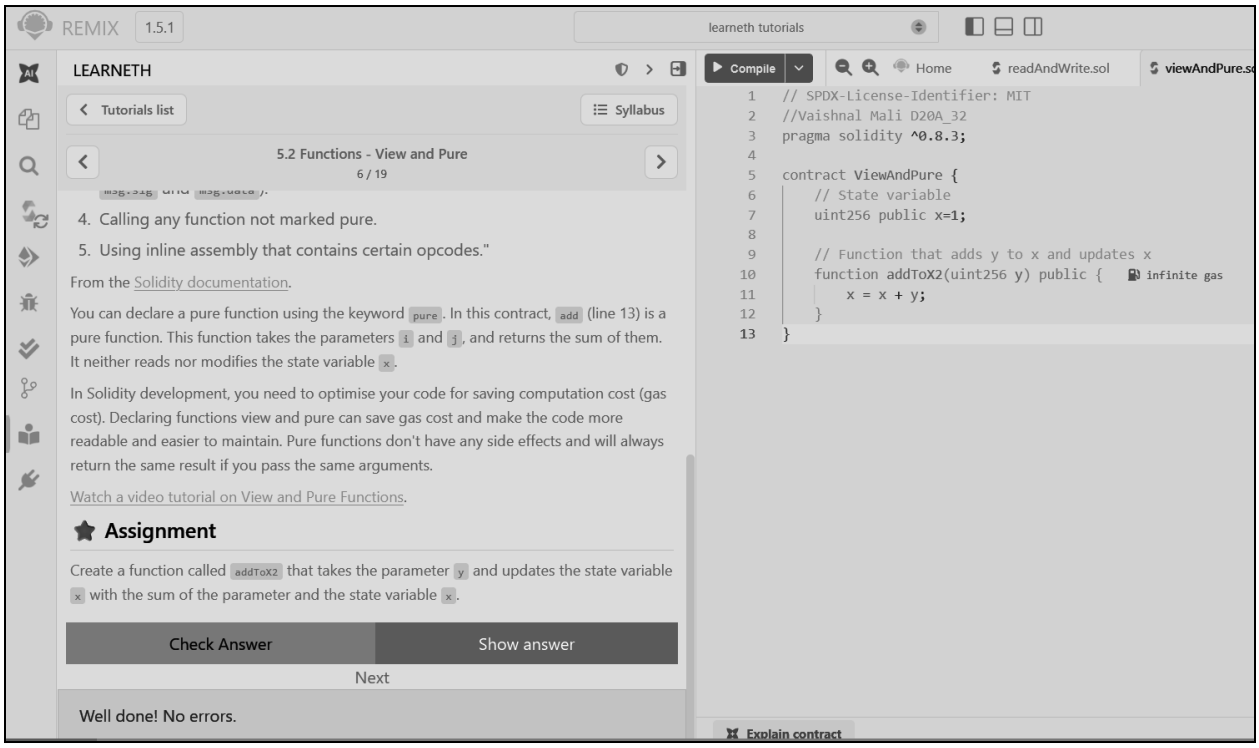
- Tutorial no.4



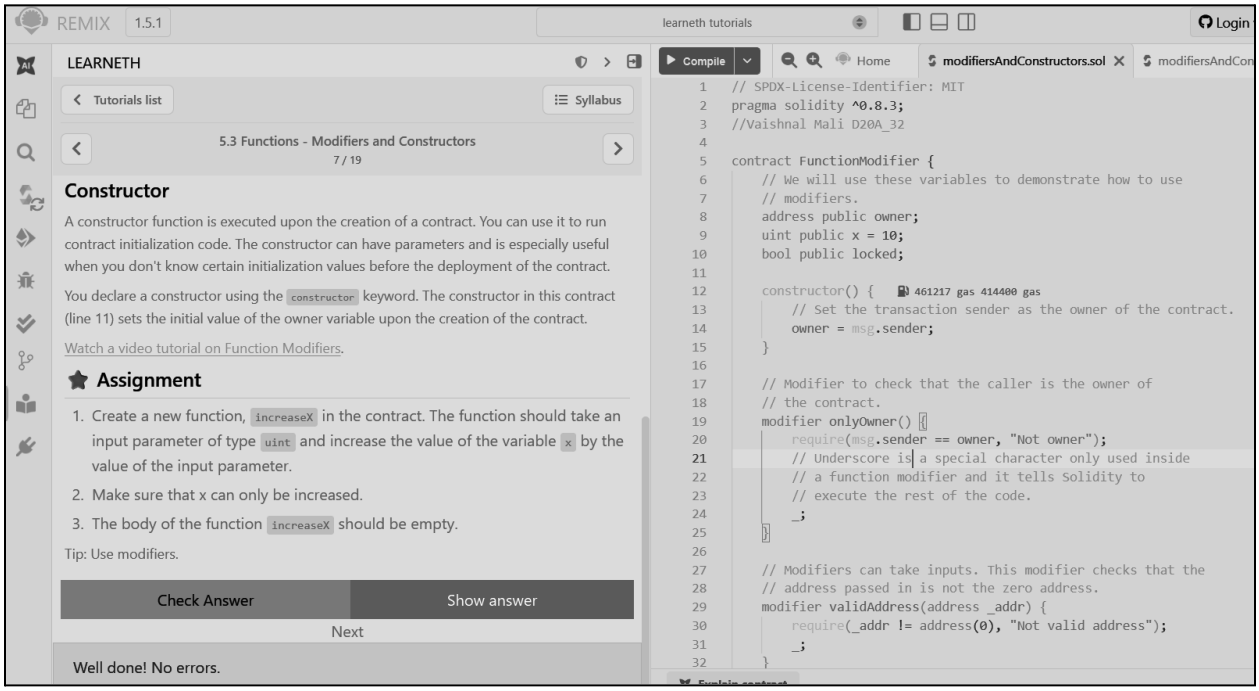

- Tutorial no.5



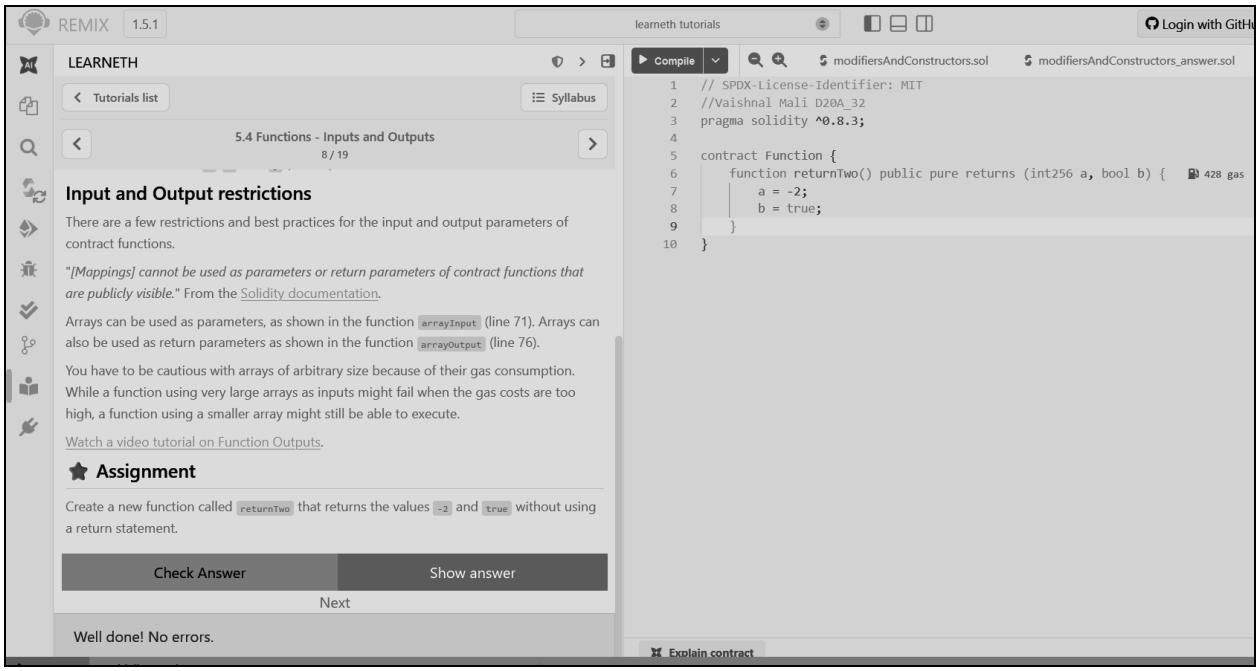
- Tutorial no.6



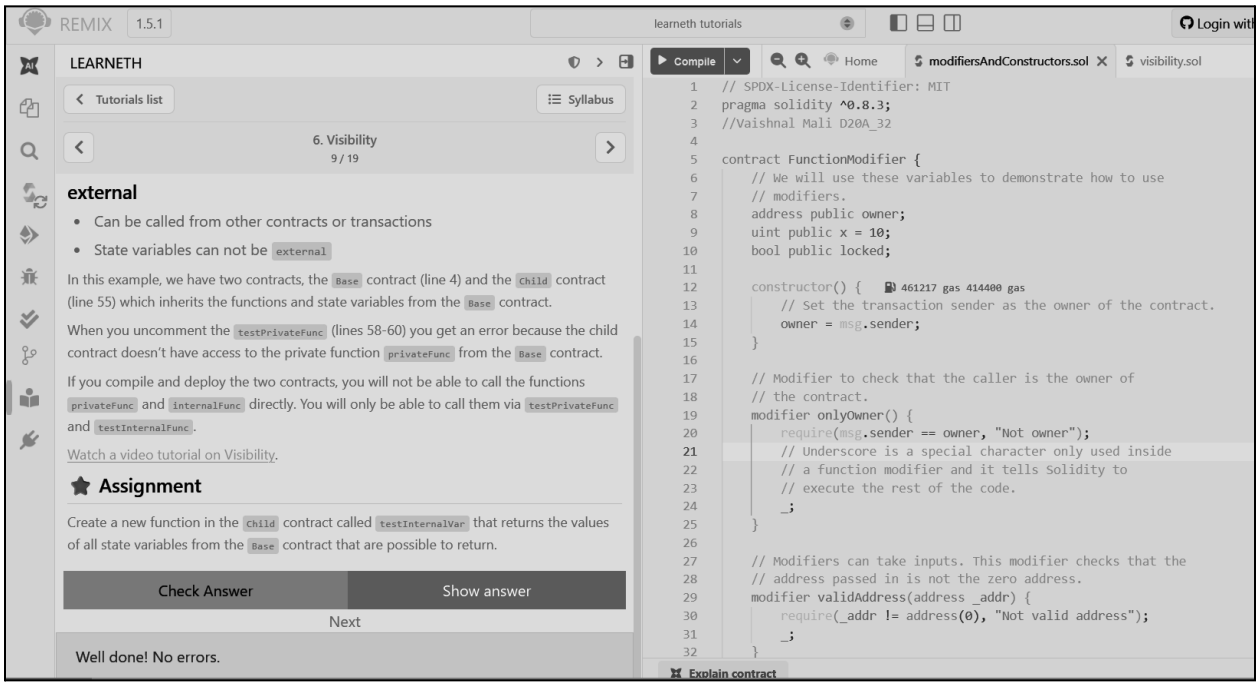

- Tutorial no.7



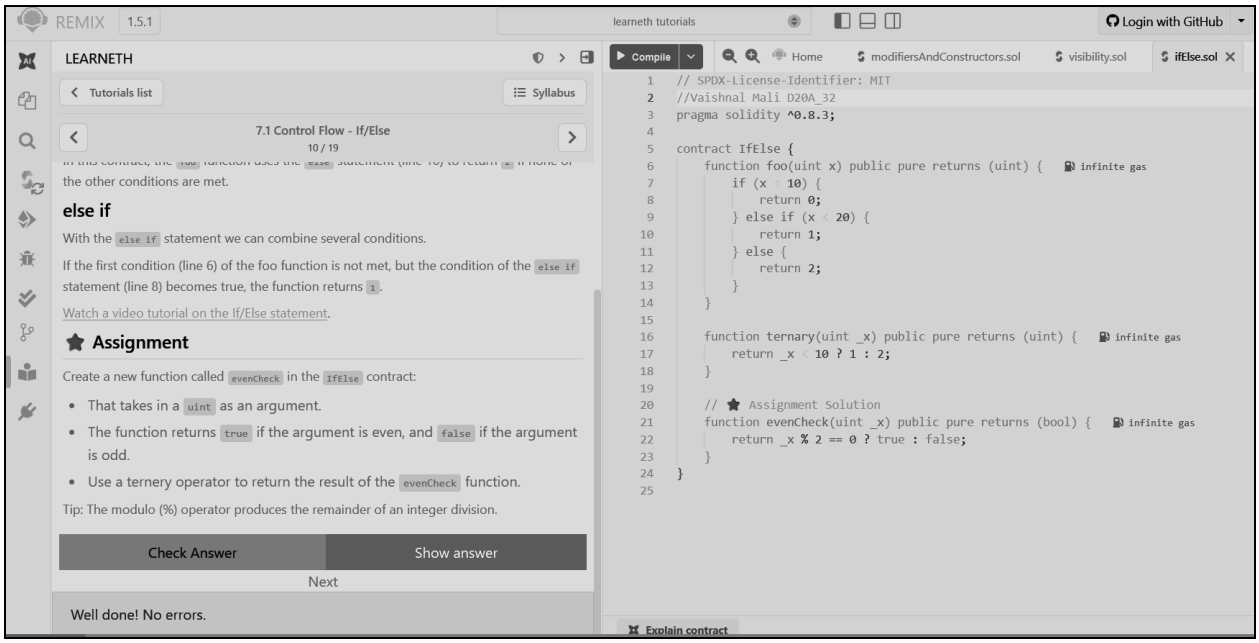
- Tutorial no.8



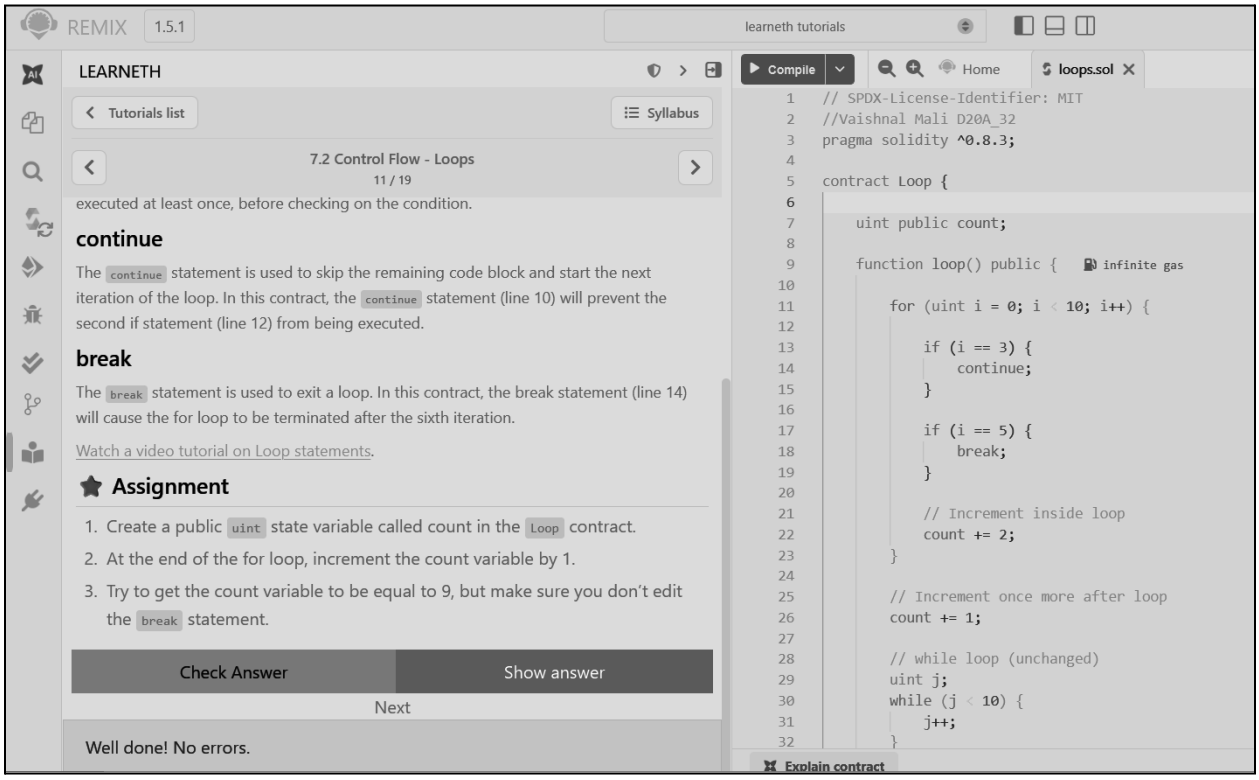

- Tutorial no.9



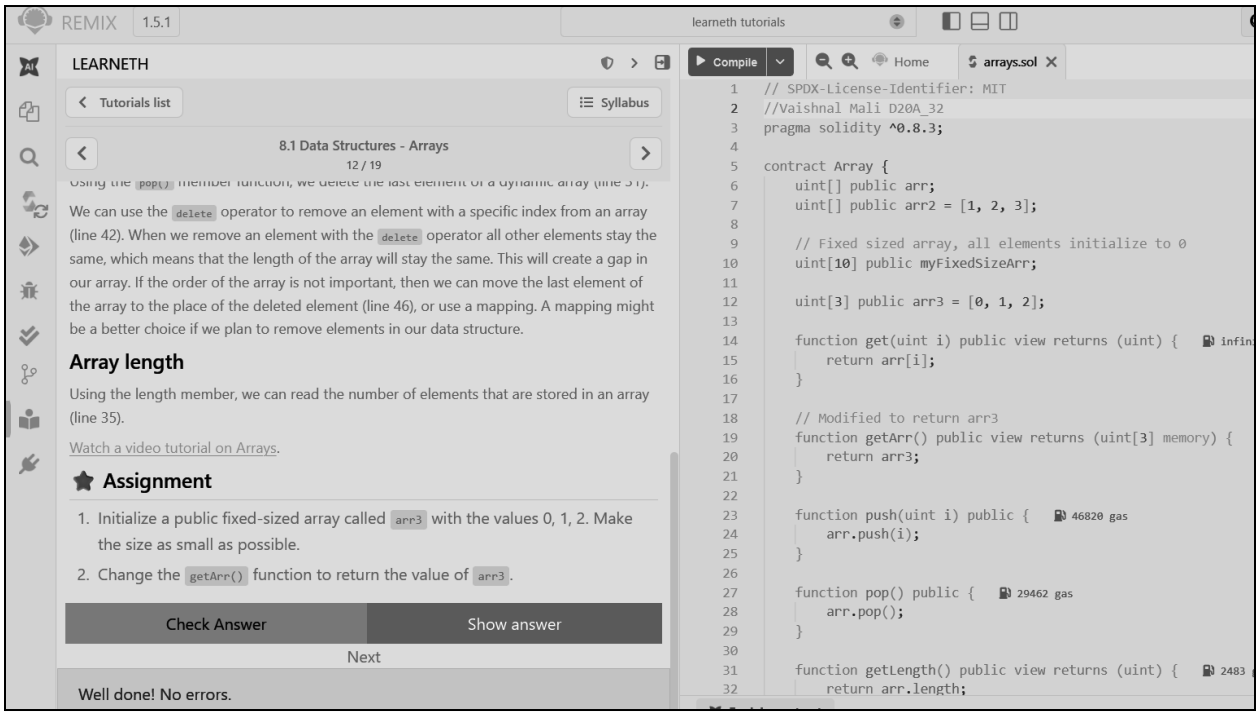
- Tutorial no.10



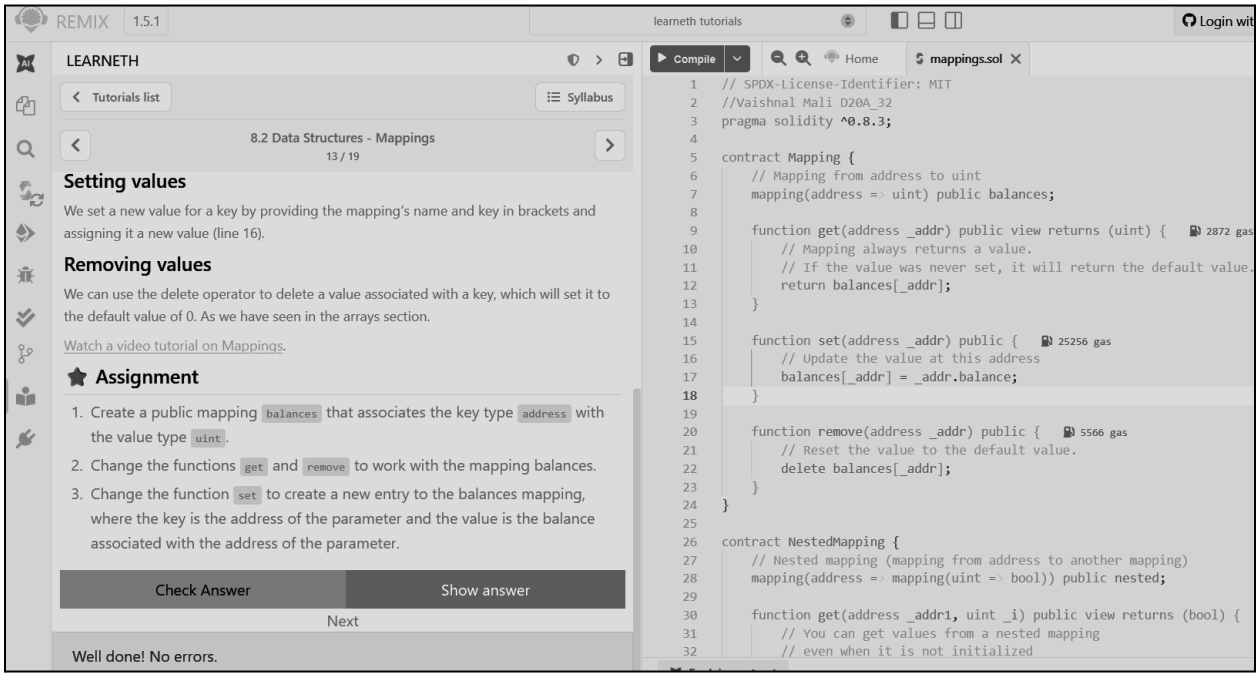

- Tutorial no.11



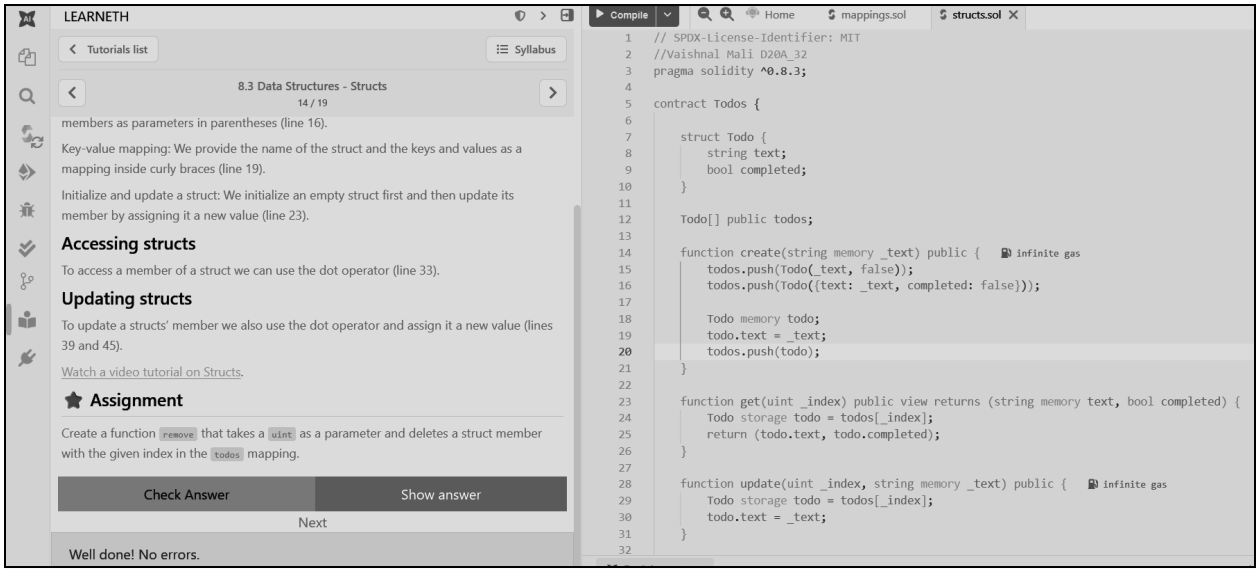
- Tutorial no.12



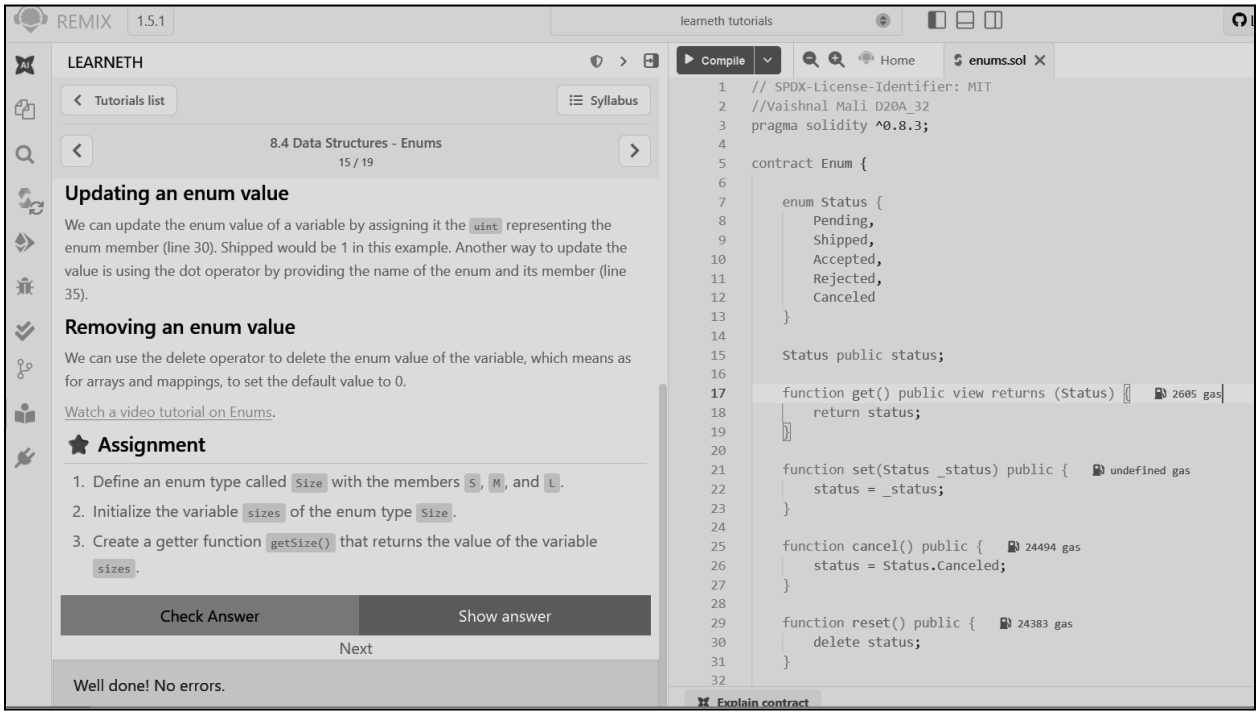

- Tutorial no.13



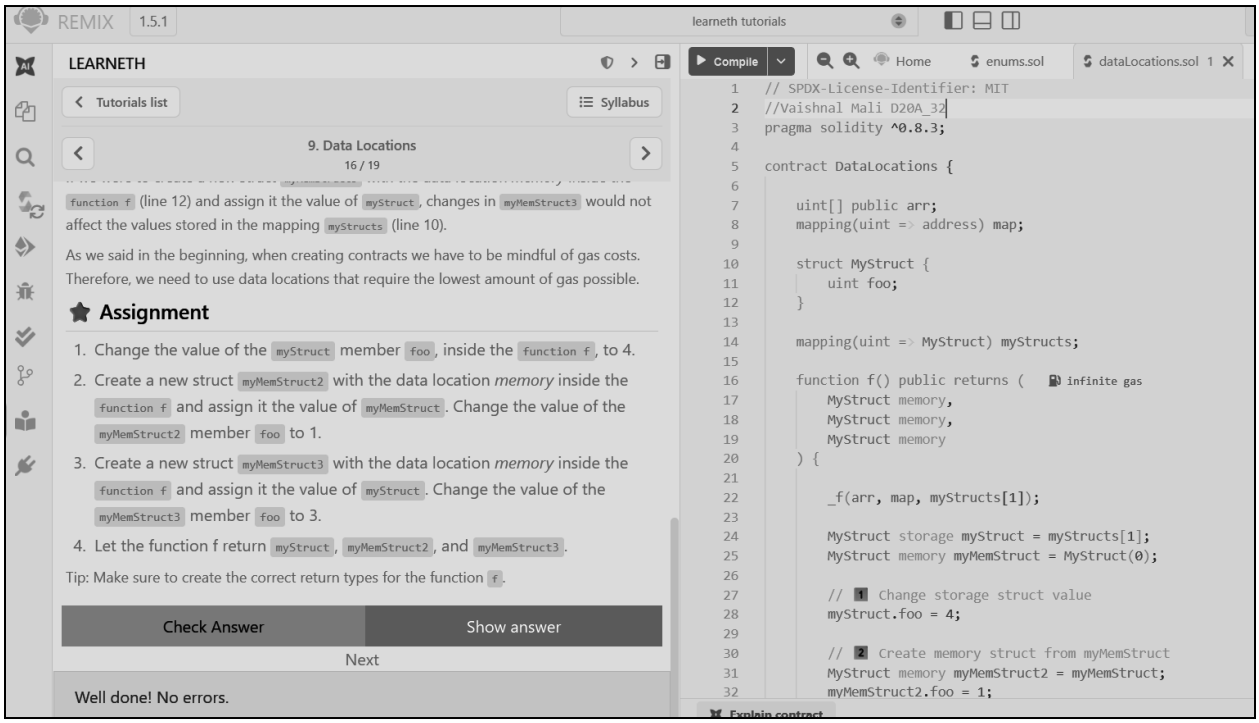
- Tutorial no.14



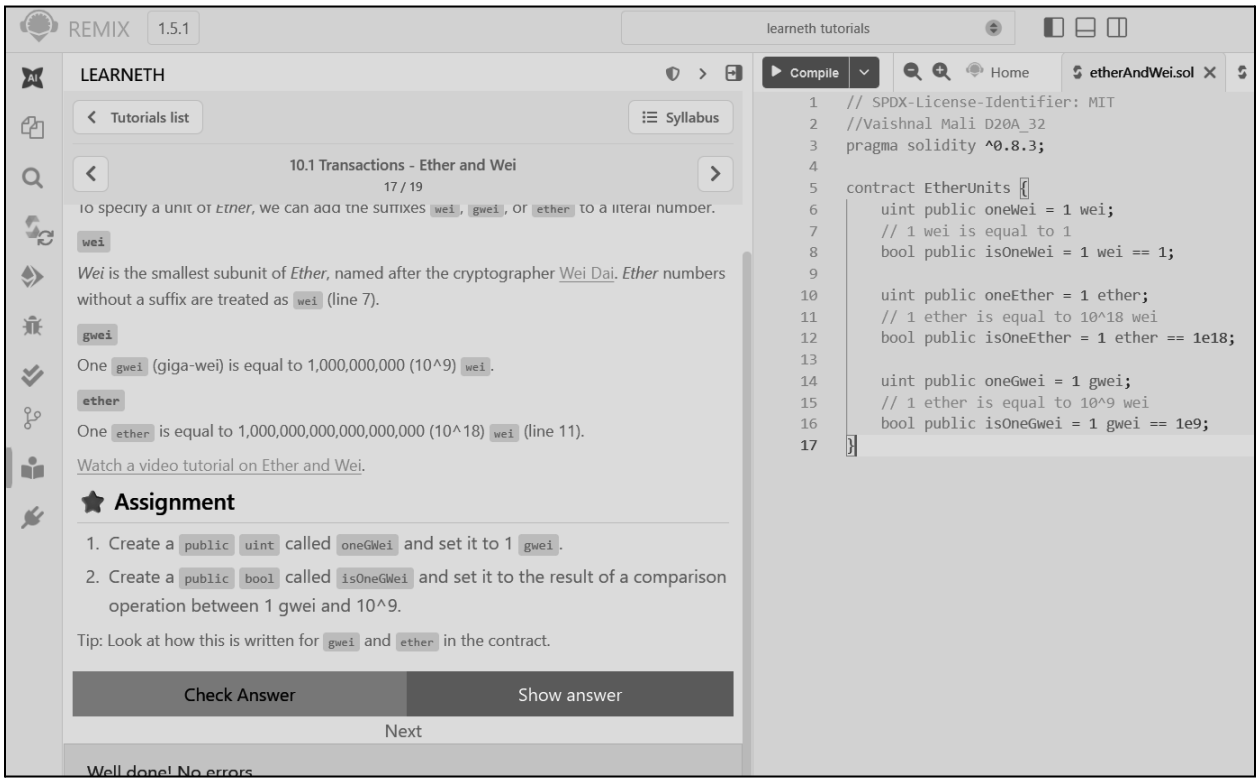

- Tutorial no.15



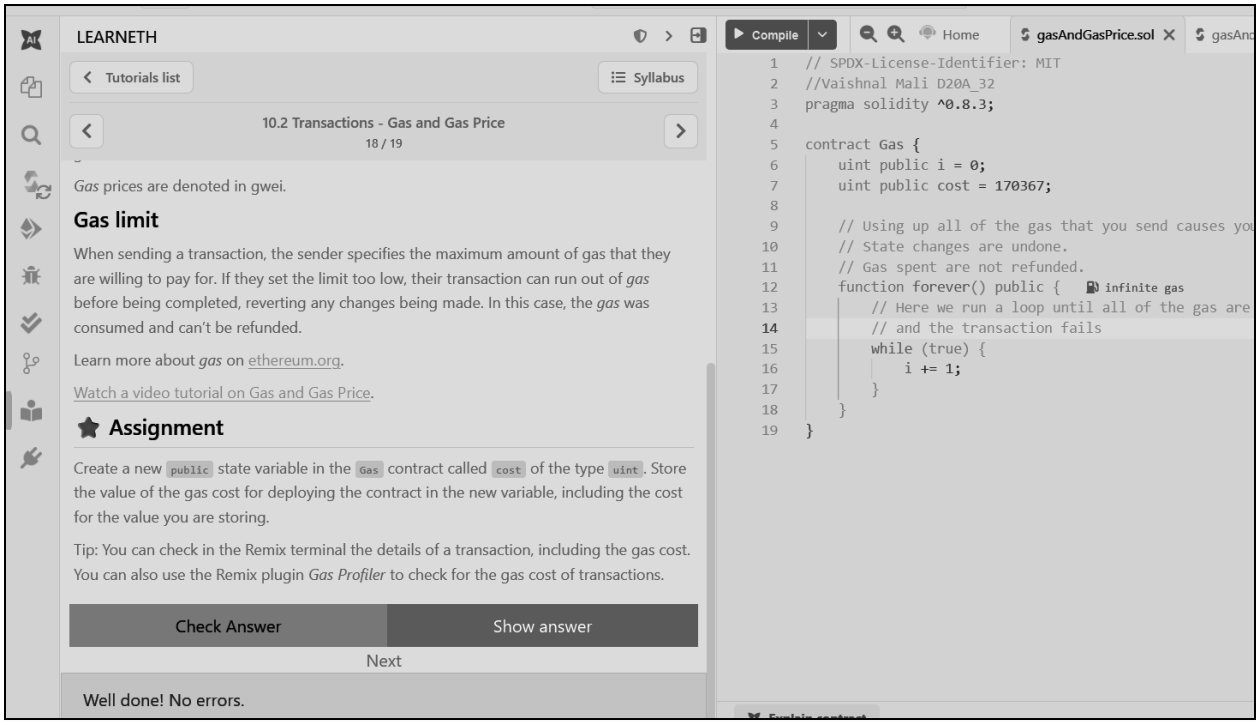
- Tutorial no.16



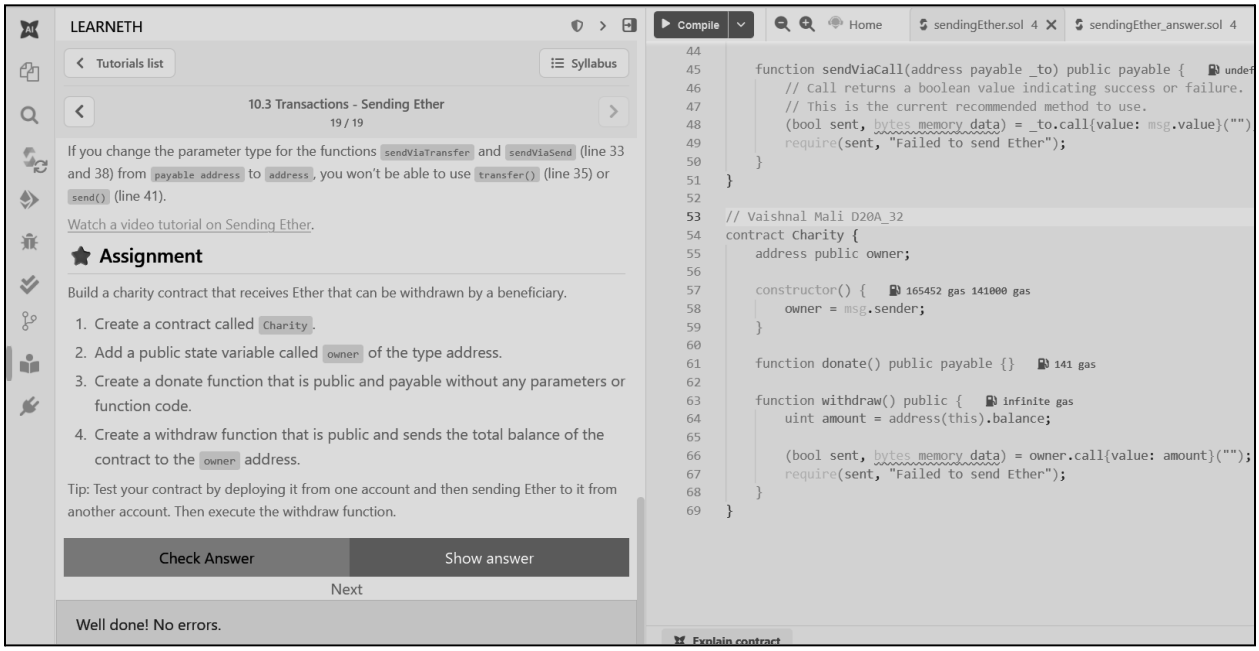

- Tutorial no.17



- 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.