

BLOCKCHAIN EXP 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.

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).
- **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.

3. 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

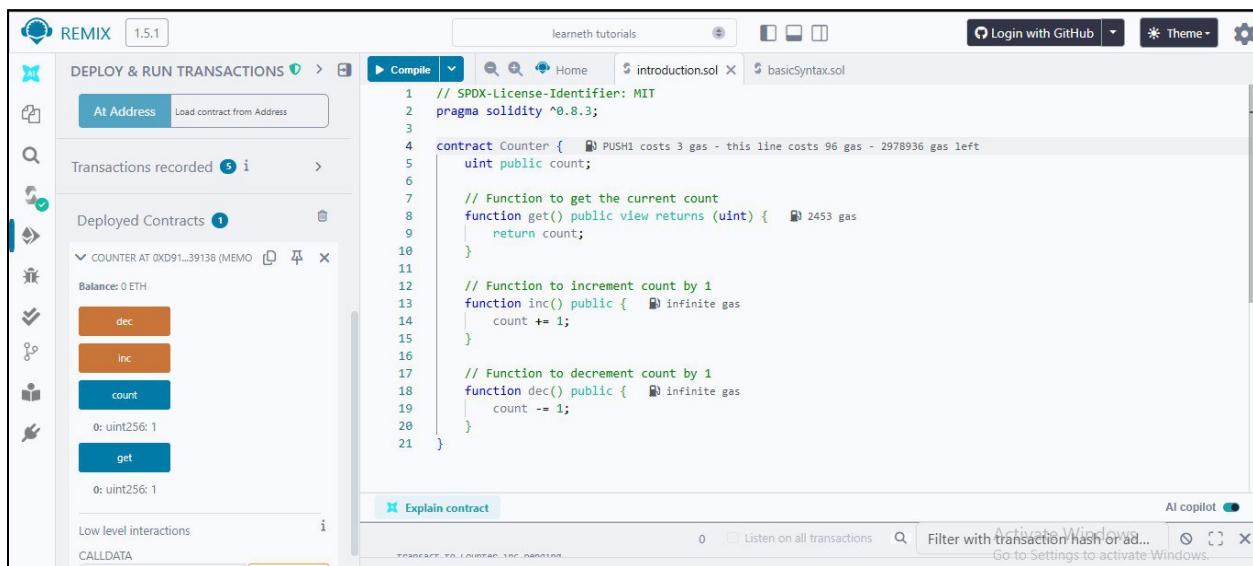
- **Ether and Wei**: Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.

- **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.

Code & Output -

Tutorial 1

1. Get counter value



The screenshot shows the REMIX IDE interface with the following details:

- IDE Version:** 1.5.1
- File:** introduction.sol
- Code:**

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     // PUSH1 costs 3 gas - this line costs 96 gas - 2978936 gas left
6     uint public count;
7
8     // Function to get the current count
9     function get() public view returns (uint) {
10         return count;
11     }
12
13     // Function to increment count by 1
14     function inc() public {
15         count += 1;
16     }
17
18     // Function to decrement count by 1
19     function dec() public {
20         count -= 1;
21     }
}

```
- Deployed Contracts:** COUNTER AT 0xD91...39138 (MEMO)
- Balance:** 0 ETH
- Functions:** dec, inc, count, get
- Low level interactions:** 0: uint256: 1
- CALLDATA:**
- Toolbars:** Deploy & Run Transactions, Compile, Home, introduction.sol, basicSyntax.sol
- Bottom Bar:** Explain contract, Listen on all transactions, Filter with transaction hash or address, Go to Settings to activate Windows.

2. Increment counter value

✓ [vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xfc...8cdc3 Debug ⌂

status	1 Transaction mined and execution succeed
transaction hash	0xfc...8cdc3
block hash	0xc873...5228cdc3
block number	6
from	0x5B38D...beddC4
to	Counter.inc() 0xd9145CCE52D386f25491e481eB44e9943F39138
transaction cost	26417 gas

3. Decrement counter value

[vm]	from: 0x5B3...eddC4	to: Counter.dec()	0xd91...39138	value: 0	wei data: 0xb3b...cfa82	logs: 0	hash: 0x21d...52d6d	Debug
status	1	Transaction mined and execution succeed						
transaction hash	0x21daa184e0a457ef2f35508a0094a8fc0c2eac05b53ab95e6d96e1c2c4452d6d							
block hash	0xca425c3b6aa253d68e49726515232861814af84154eb74afe6c4683415457db8							
block number	7							
from	0x5B38Daa6a701c568545dCfcB03FcB875f56beddC4							
to	Counter.dec()	0xd9145CCE52D386f254917e481eB44e9943F39138						
transaction cost	26461	gas						

4. Get count value

Tutorial 2

Tutorial 3

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, deployment, and running transactions. The main area has tabs for 'Compiled' and 'Solidity'. The 'Compiled' tab is active, displaying the following Solidity code:

```

25 int public i = -123; // int is same as int256
26
27 address public addr = 0xCA35b7d915458EF540aDe6068fFe2F44E8fa733c;
28
29 // Default values
30 // Unassigned variables have a default value
31 bool public defaultBool; // false
32 uint public defaultUint; // 0
33 int public defaultInt; // 0
34 address public defaultAddr; // 0x0000000000000000000000000000000000000000
35 address public newAddr = 0x0000000000000000000000000000000000000000;
36 int public neg = -4;
37 uint8 public newU;
38 }

```

Below the code, there's an 'Explain contract' section with a button to 'Debug' a transaction. The transaction details show a call to 'Variables.text()' from a specific address.

Tutorial 4

The screenshot shows the REMIX IDE interface. The sidebar shows multiple deployed contracts like HELLOWORD and MYCONTRACT. The main area displays a more complex Solidity contract:

```

4 contract Variables {
5     // State variables are stored on the blockchain.
6     string public text = "Hello";
7     uint public num = 123;
8     uint public blockNumber;
9
10    function doSomething() public {
11        // Local variables are not saved to the blockchain.
12        uint i = 456;
13
14        // Here are some global variables
15        uint timestamp = block.timestamp; // Current block timestamp
16        address sender = msg.sender; // address of the caller
17        blockNumber = block.number;
18    }
19 }

```

Similar to Tutorial 3, it includes an 'Explain contract' section with a 'Debug' button for transactions.

Tutorial 5

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract SimpleStorage {
    // State variable to store a number
    uint num;

    function get() public view returns (uint) {
        return num;
    }
}
```

Tutorial 6

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract ViewAndPure {
    uint public x = 1;

    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }

    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }
}
```

Tutorial 7

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract FunctionModifier {
5     // We will use these variables to demonstrate how to use
6     // modifiers.
7     address public owner;
8     uint public x = 10;
9     bool public locked;
10
11 constructor() {
12     // Set the transaction sender as the owner of the contract.
13     owner = msg.sender;
14 }
15
16 // Modifier to check that the caller is the owner of
17 // the contract.

```

Tutorial 8

LEARNETH

5.4 Functions - Inputs and Outputs

The `returnMany` function (line 6) shows how to return multiple values. You will often return multiple values. It could be a function that collects outputs of various functions and returns them in a single function call for example.

The `named` function (line 19) shows how to name return values. Naming return values helps with the readability of your contracts. Named return values make it easier to keep track of the values and the order in which they are returned. You can also assign values to a name.

The `assigned` function (line 33) shows how to assign values to a name. When you assign values to a name you can omit (leave out) the return statement and return them individually.

Deconstructing Assignments

You can use deconstructing assignments to unpack values into distinct variables.

The `destructingAssignments` function (line 49) assigns the values of the `returnMany` function to the new local variables `a`, `b`, and `c` (line 60).

Input and Output restrictions

There are a few restrictions and best practices for the input and output parameters of contract functions.

"[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible." From the [Solidity documentation](#).

Arrays can be used as parameters, as shown in the function `arrayInput` (line 71). Arrays can also be used as return parameters as shown in the function `arrayOutput` (line 76).

You have to be cautious with arrays of arbitrary size because of their gas consumption. While a function using very large arrays as inputs might fail when the gas costs are too high, a function using a smaller array might still be able to execute.

Watch a video tutorial on [Function Outputs](#).

★ Assignment

Create a new function called `returnTwo` that returns the values `-2` and `true` without using a return statement.

Check Answer **Show answer**

Tutorial 9

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Base {
7     // Private function can only be called
8     // - inside this contract
9     // Contracts that inherit this contract cannot call this function.
10    function privateFunc() private pure returns (string memory) { return "infinite gas"; }
11    return "private function called";
12 }
13
14 function testPrivateFunc() public pure returns (string memory) { return privateFunc(); }
15
16 // Internal function can be called
17 // - inside this contract
18 // - inside contracts that inherit this contract
19 function internalFunc() internal pure returns (string memory) { return "internal function called"; }
20
21 function testInternalFunc() public pure virtual returns (string memory) { return internalFunc(); }
22
23 // Public functions can be called
24 // - inside this contract
25 // - inside contracts that inherit this contract
26 // - by other contracts and accounts
27
28
29
30
31
32

```

Explain contract

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6

Tutorial 10

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract IfElse {
7     function foo(uint x) public pure returns (uint) {
8         if (x < 10) {
9             return 0;
10        } else if (x < 20) {
11            return 1;
12        } else {
13            return 2;
14        }
15    }
16
17    function ternary(uint _x) public pure returns (uint) {
18        // if (_x < 10) {
19        //     return 1;
20        // }
21        // return 2;
22
23        // shorthand way to write if / else statement
24        return _x < 10 ? 1 : 2;
25    }
26 }

```

Explain contract

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6

Tutorial 11

The screenshot shows the REMIX IDE interface. On the left, the 'Tutorials list' sidebar shows '7.2 Control Flow - Loops' selected. The main content area displays the tutorial text and code. The code editor on the right contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/6
5
6 contract Loop {
7     function loop() public {
8         // for loop
9         for (uint i = 0; i < 10; i++) {
10             if (i == 3) {
11                 // Skip to next iteration with continue
12                 continue;
13             }
14             if (i == 5) {
15                 // Exit loop with break
16                 break;
17             }
18         }
19     }
20
21     // while loop
22     uint j;
23     while (j < 10) {
24         j++;
25     }
26 }

```

The code editor has syntax highlighting for Solidity keywords and comments. Below the code editor is an 'Explain contract' section with a green checkmark icon and a log entry: '[vm] from: 0x5B3...eddc4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6e...7f733'.

Tutorial 12

The screenshot shows the REMIX IDE interface. On the left, the 'Tutorials list' sidebar shows '8.1 Data Structures - Arrays' selected. The main content area displays the tutorial text and code. The code editor on the right contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 // Pranav Titambe D20A/60
4
5 contract Array {
6     // Several ways to initialize an array
7     uint[] public arr;
8     uint[] public arr2 = [1, 2, 3];
9     // Fixed sized array, all elements initialize to 0
10    uint[10] public myFixedSizeArr;
11
12    function get(uint i) public view returns (uint) {
13        return arr[i];
14    }
15
16    // Solidity can return the entire array.
17    // But this function should be avoided for
18    // arrays that can grow indefinitely in length.
19    function getArr() public view returns (uint[] memory) {
20        return arr;
21    }
22
23    function push(uint i) public {
24        // Append to array
25        // This will increase the array length by 1.
26        arr.push(i);
27    }
28
29    function pop() public {
30        // Remove last element from array
31        // This will decrease the array length by 1
32        arr.pop();
33    }
}

```

The code editor has syntax highlighting for Solidity keywords and comments. Below the code editor is an 'Explain contract' section with a green checkmark icon and a log entry: '[vm] from: 0x5B3...eddc4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6e...7f733'.

Tutorial 13

The screenshot shows the REMIX IDE interface. The top navigation bar includes tabs for 'REMX' (selected), '1.5.1', 'learneth tutorials', and several open files like 'arrays.sol', 'mappings.sol', 'loops.sol', and 'ifElse.sol'. The left sidebar has sections for 'Tutorials list' and 'Syllabus'. The main content area displays a tutorial on 'Data Structures - Mappings'. It includes a heading '8.2 Data Structures - Mappings', a sub-section 'Creating mappings' with code syntax highlighting for `mapping`, and another section 'Accessing values' with code syntax highlighting for `get` and `set`. On the right, there's a code editor with Solidity code for a 'Mapping' contract and a 'NestedMapping' contract. Below the code editor are buttons for 'Compile', 'Run', and 'Deploy'. A status bar at the bottom shows transaction details: '(vm) from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: ...'. A footer bar at the bottom right contains icons for 'Listen on all transactions' and 'Filter'.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

// Pranav Titambe d20a/60

contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public myMap;

    function get(address _addr) public view returns (uint) {
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return myMap[_addr];
    }

    function set(address _addr, uint _i) public {
        // Update the value at this address
        myMap[_addr] = _i;
    }

    function remove(address _addr) public {
        // Reset the value to the default value.
        delete myMap[_addr];
    }
}

contract NestedMapping {
    // Nested mapping (mapping from address to another mapping)
    mapping(address => mapping(uint => bool)) public nested;

    function get(address _addr1, uint _i) public view returns (bool) {
        // You can get values from a nested mapping
    }
}
```

Tutorial 14

The screenshot shows the REMIX IDE interface. The top navigation bar includes tabs for 'LEARNETH' and 'Tutorial'. On the left, there's a sidebar with various icons and a 'Tutorials list' button. The main content area has a title '8.3 Data Structures - Structs' and a subtitle '14 / 19'. The code editor displays the following Solidity code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
// Pranav Titambe d20a/60

contract Todos {
    struct Todo {
        string text;
        bool completed;
    }

    // An array of 'Todo' structs
    Todo[] public todos;

    function create(string memory _text) public {
        // 3 ways to initialize a struct
        // - calling it like a function
        todos.push(Todo(_text, false));

        // key value mapping
        todos.push(Todo({text: _text, completed: false}));

        // initialize an empty struct and then update it
        Todo memory todo;
        todo.text = _text;
        // todo.completed initialized to false
        todos.push(todo);
    }

    // Solidity automatically created a getter for 'todos' so
    // you don't actually need this function.
}
```

Below the code editor, there's an 'Explain contract' button. At the bottom, there are buttons for 'Check Answer' and 'Show answer'.

Tutorial 15

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Enum {
7     // Enum representing shipping status
8     enum Status {
9         Pending,
10        Shipped,
11        Accepted,
12        Rejected,
13        Canceled
14    }
15
16    // Default value is the first element listed in
17    // definition of the type, in this case "Pending"
18    Status public status;
19
20    // Returns uint
21    // Pending - 0
22    // Shipped - 1
23    // Accepted - 2
24    // Rejected - 3
25    // Canceled - 4
26    function get() public view returns (Status) {
27        return status;
28    }
29
30    // Update status by passing uint into input
31    function set(Status _status) public {
32        status = _status;
33    }

```

Explain contract

[VM] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

Tutorial 16

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract DataLocations {
7     uint[] public arr;
8     mapping(uint => address) map;
9     struct MyStruct {
10        uint foo;
11    }
12    mapping(uint => MyStruct) myStructs;
13
14    function f() public {
15        // call _f with state variables
16        _f(arr, map, myStructs[1]);
17
18        // get a struct from a mapping
19        MyStruct storage myStruct = myStructs[1];
20        // create a struct in memory
21        MyStruct storage myMemStruct = MyStruct(0);
22    }
23
24    function _f() {
25        uint[] storage _arr,
26        mapping(uint => address) storage _map,
27        MyStruct storage _myStruct
28    } internal {
29        // do something with storage variables
30    }
31
32    // You can return memory variables

```

Explain contract

[VM] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

Tutorial 17

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' tutorial sidebar for '10.1 Transactions - Ether and Wei' is visible. The main content area displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract EtherUnits {
7     uint public oneWei = 1 wei;
8     // 1 wei is equal to 1
9     bool public isOneWei = 1 wei == 1;
10
11    uint public oneEther = 1 ether;
12    // 1 ether is equal to 10^18 wei
13    bool public isOneEther = 1 ether == 1e18;
14 }

```

On the right, the 'Explain contract' panel shows a transaction log:

- [VM] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

Tutorial 18

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' tutorial sidebar for '10.2 Transactions - Gas and Gas Price' is visible. The main content area displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Gas {
7     uint public i = 0;
8
9     // Using up all of the gas that you send causes your transaction
10    // State changes are undone.
11    // Gas spent are not refunded.
12    function forever() public {
13        infiniteGas
14        // Here we run a loop until all of the gas are spent
15        // and the transaction fails
16        while (true) {
17            i += 1;
18        }
19    }

```

On the right, the 'Explain contract' panel shows a transaction log:

- [VM] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

Tutorial 19

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract ReceiveEther {
7     /*
8      * Which function is called
9      */
10    send Ether
11    | msg.data is empty?
12    |   / \
13    |   yes no
14    |
15    receive() exists? fallback()
16    |   /
17    |   yes no
18    |   /
19    |   receive() fallback()
20    |
21    */
22
23    // Function to receive Ether. msg.data must be empty
24    receive() external payable { undefined gas
25
26    // Fallback function is called when msg.data is not empty
27    fallback() external payable { undefined gas
28
29    function getBalance() public view returns (uint) {
30        return address(this).balance;
31    }
32 }

```

Explain contract

[VM] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

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.