

Experiment No. 4

Aim: Hands on Solidity programming assignments for creating smart contracts

Theory:

Q1: Data Types, Variables, and Function Modifiers

Solidity is a statically typed language, meaning the type of each variable must be specified at compile time.

- Integers: uint (unsigned) and int (signed) come in steps of 8 bits (e.g., uint8 to uint256). Using the correct size can save gas, though uint256 is the most common as the EVM is optimized for 256-bit words.
- Address: This is a unique type to Solidity. A payable address is a subtype that allows you to send Ether to it using .transfer() or .send().
- State vs. Local: State variables are written into the contract's storage (the blockchain's "hard drive"), making them expensive to change. Local variables exist only in the "stack" during execution and are much cheaper.
- Pure vs. View: View: You are looking at the state (e.g., checking a balance) but not touching it.
Pure: You aren't even looking at the state (e.g., a math utility like $2 + 2$).

Q2: Inputs and Outputs

Functions are the primary way users interact with the blockchain.

- Multiple Returns: Unlike many languages, Solidity supports returning multiple values natively: return (uint a, bool b, string memory c);
- Named Returns: You can define the variable name in the function signature: returns (uint sum). This allows you to simply assign a value to sum inside the function without an explicit return statement at the end.

Q3: Visibility, Modifiers, and Constructors

Visibility is the first line of security in a smart contract.

- External vs. Public: external functions are sometimes more gas-efficient than public when receiving large arrays because the data is read directly from calldata rather than being copied to memory.
- Modifiers: These act as "gatekeepers." A common pattern is the onlyOwner modifier, which checks if msg.sender == owner before allowing the rest of the function code (represented by the _ symbol) to run.
- Constructors: If you don't define one, a default constructor is used. Once the contract is deployed, the constructor code is discarded and can never be called again.

Q4: Control Flow

While Solidity supports standard loops, the Gas Limit introduces a unique constraint.

- The Denial of Service (DoS) Risk: If you create a for loop that iterates through a dynamic array of users, and that array grows too large, the gas required to finish the loop might exceed the block gas limit. This would make the function impossible to execute, effectively "bricking" the contract.
- Recommendation: Favor if-else logic over heavy loops whenever possible.

Q5: Data Structures

- Mappings: Think of these as a Hash Table. They are incredibly gas-efficient for lookups. However, you cannot "list" all keys in a mapping; if you need to know who all the users are, you usually pair a mapping with an address[] array.
- Structs: These allow you to create complex records. For example, a Request struct in a crowdfunding contract might contain a description, a value, and a recipient.
- Enums: These restrict a variable to have one of only a few predefined values, which reduces human error and makes the code self-documenting.

Q6: Data Locations (Storage, Memory, Calldata)

This is often the most confusing part for beginners but is vital for gas optimization.

- Storage: Permanent and expensive. Use this for data that must persist between transactions.
- Memory: Temporary and cheaper. Use this for data that you need to manipulate during a function call.
- Calldata: The cheapest. It is an immutable (read-only) area where function arguments are stored.

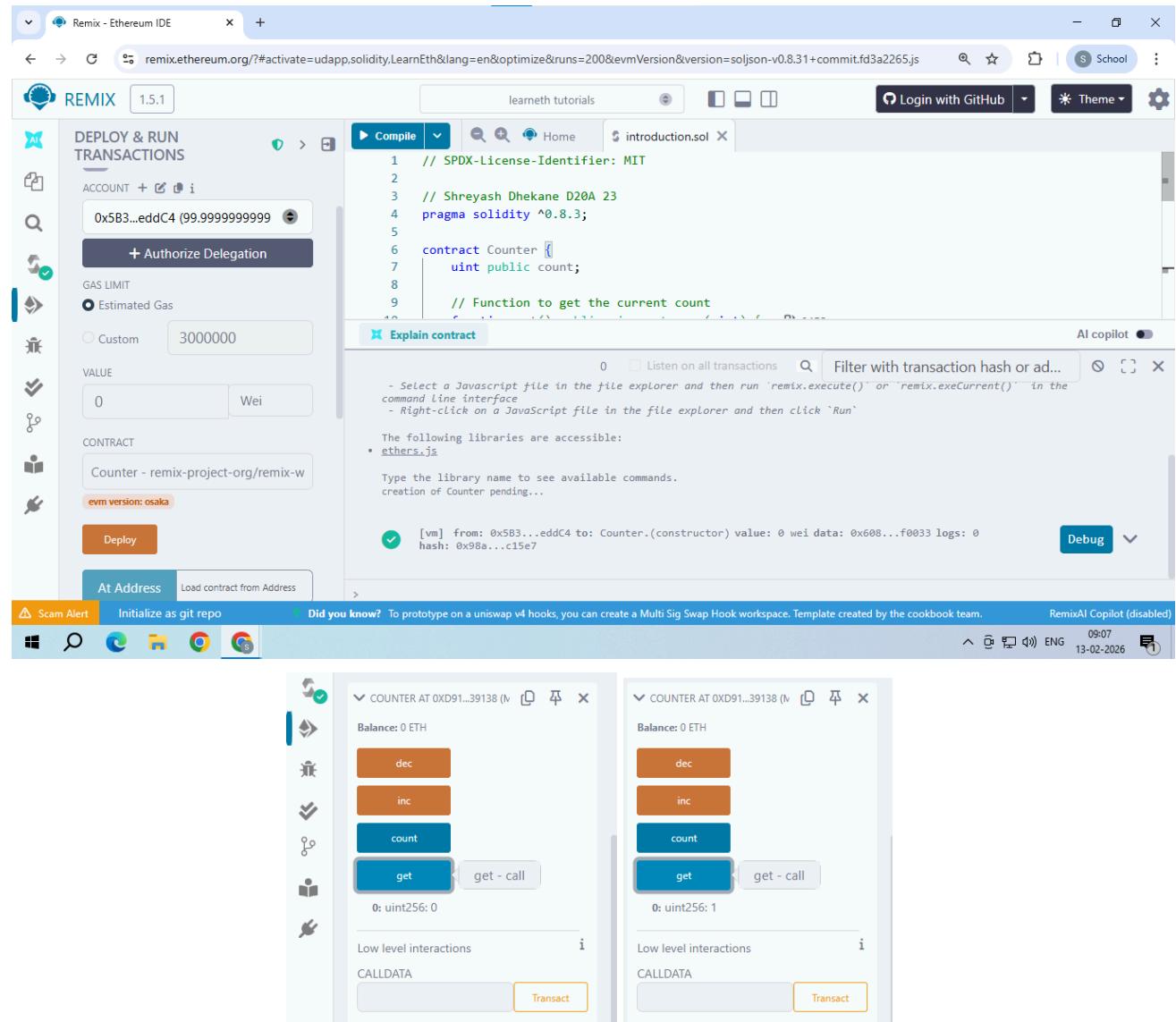
Pro Tip: Changing a variable from storage to memory creates a copy. Changing a pointer to storage affects the actual state variable on the blockchain.

Q7: Transactions and Gas

- Wei: Just as a Dollar has Cents, an Ether has Wei.
 - $1 \{ \text{Ether} \} = 1,000,000,000,000,000,000 \{ \text{Wei} \}$ (10^{18}).
- Gas: Think of Gas as the "fuel" and Gas Price as the "price per gallon."
 - $\{ \text{Total Cost} \} = \{ \text{Gas Used} \} \times \{ \text{Gas Price} \}$
- The call Method: While transfer() was once the standard, the Ethereum community now recommends using .call{value: amount}("") for sending Ether, as it is more flexible and handles gas more gracefully for modern contract interactions.

Tasks Performed:-

Assignment 1:



The screenshot shows the Remix Ethereum IDE interface. On the left, the sidebar includes sections for 'DEPLOY & RUN TRANSACTIONS' (with account set to 0x5B3...eddC4), 'GAS LIMIT' (set to 3000000), and 'CONTRACT' (Counter). The main workspace displays the Solidity code for the Counter contract:

```

1 // SPDX-License-Identifier: MIT
2
3 // Shreyash Dhekane D20A 23
4 pragma solidity ^0.8.3;
5
6 contract Counter {
7     uint public count;
8
9     // Function to get the current count
10    function get() public view returns (uint) {
11        return count;
12    }
13}

```

The 'Explain contract' panel below the code provides information about the contract's methods and variables. At the bottom, the transaction details show a successful deployment with address 0x98a...c15e7 and a log message indicating the creation of the Counter pending...

Below the IDE, two browser tabs are visible, both showing the deployed Counter contract at address 0xD91...39138. The tabs show the contract's balance (0 ETH) and its four functions: dec, inc, count, and get. The 'get' function is highlighted in blue, indicating it is currently selected or being interacted with.

2: Basic Syntax

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

contract MyContract {
    string public name = "Alice";
}
```

3: Primitive Data Types

```
/*
Negative numbers are allowed for int types.
like uint, different ranges are available from int8 to int256
*/
int8 public i8 = -1;
int public i256 = 456;
int public i = -123; // int is same as int256

address public defaultAddr;
// Default
// Unassi
bool publ;
uint public defaultUint; // 0
int public defaultInt; // 0
address public defaultAddress; // 0x0000000000000000000000000000000000000000000000000000000000000000

// New values
address public newAddr = 0x0000000000000000000000000000000000000000;
int public neg = -12;
uint8 public newU = 0;
*/
// Shreyash Dhekane D20A 23
```

4: Variables

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

contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint public num = 123;
    uint public blockNumber;

    function doSomething() public {
        // Local variables are not saved to the blockchain.
        uint _i = 456;

        // Here are some global variables
        uint timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // Address of the caller
        blockNumber = block.number;
    }
}
// Shreyash Dhekane D20A 23
```

5: Functions

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7     bool public b = true;
8
9     // You need to send a transaction to write to a state variable.
10    function set(uint _num) public {
11        num = _num;
12    }
13
14    // You can read from a state variable without sending a transaction.
15    function get() public view returns (uint) {
16        return num;
17    }
18
19    function get_b() public view returns (bool) {
20        return b;
21    }
22}
23
// Shreyash Dhekane D20A 23

```

6: Functions

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract ViewAndPure {
5     uint public x = 3;
6
7     // Promise not to modify the state.
8     function addToX(uint y) public view returns (uint) {
9         return x + y;
10    }
11
12    // Promise not to modify or read from the state.
13    function add(uint i, uint j) public pure returns (uint) {
14        return i + j;
15    }
16
17    function addToX2(uint y) public {
18        x = x + y;
19    }
20}
21
// Shreyash Dhekane D20A 23

```

7: Functions

```

48
49
50    // Modifiers can be called before and / or after a function.
51    // This modifier prevents a function from being called while
52    // it is still executing.
53    modifier noReentrancy() {
54        require(!locked, "No reentrancy");
55
56        locked = true;
57        _;
58        locked = false;
59    }
60
61    function increaseX(uint i) public increased {
62        x += i;
63
64        if (i > 1) {
65            decrease(i - 1);
66        }
67    }
68
69
70 // Shreyash Dhekane D20A 23

```

8: Functions

The screenshot shows the Remix Ethereum IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial section '5.4 Functions - Inputs and Outputs'. The main workspace contains a Solidity code editor with the following code:

```

function arrayOutput() public view returns (uint[] memory) {
    uint[] memory arr;
    return arr;
}

function returnTwo() public pure returns (
    int i,
    bool b
) {
    i = -2;
    b = true;
}

```

Below the code editor, a message says 'Well done! No errors.' and there are 'Check Answer' and 'Show answer' buttons.

9: Visibility

The screenshot shows the Remix Ethereum IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial section '6. Visibility'. The main workspace contains a Solidity code editor with the following code:

```

// State variables cannot be external so this code won't compile.
// string external externalVar = "my external variable";
}

contract Child is Base {
    // Inherited contracts do not have access to private functions
    // and state variables.
    // function testPrivateFunc() public pure returns (string memory) {
    //     return privateFunc();
    // }

    // Internal function call be called inside child contracts.
    function testInternalFunc() public pure override returns (string memory) {
        return internalFunc();
    }

    function testInternalVar() public view returns (string memory, string memory)
    {
        return (internalVar, publicVar);
    }
}

```

Below the code editor, a message says 'Well done! No errors.' and there are 'Check Answer' and 'Show answer' buttons.

10: Control Flow

The screenshot shows the Remix Ethereum IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial section '7.1 Control Flow - If/Else'. The main workspace contains a Solidity code editor with the following code:

```

function ternary(uint256 _x) public pure returns (uint256 internal_ )
{
    if (_x < 10) {
        return 1;
    }
    return 2;
}

function evenCheck(uint y) public pure returns (bool) {
    return y%2 == 0 ? true : false;
}

```

Below the code editor, a message says 'Well done! No errors.' and there are 'Check Answer' and 'Show answer' buttons.

11: Control Flow

```

function loop() public{ infinite gas
    // for loop
    for (uint i = 0; i < 10; i++) {
        if (i == 5) {
            // Skip to next iteration with continue;
            continue;
        }
        if (i == 5) {
            // Exit loop with break;
            break;
        }
        count++;
    }

    // while loop
    uint j;
    while (j < 10) {
        j++;
    }
}

```

// Shreyash Dhekane D20A 23

12: Data Structure - Arrays

```

// Move the last element into the place to delete
arr[index] = arr[arr.length - 1];
// Remove the last element
arr.pop();
}

function test() public {
    infinite gas
    arr.push(1);
    arr.push(2);
    arr.push(3);
    arr.push(4);
    // [1, 2, 3, 4]
    remove(1);
    // [1, 4, 3]
    remove(2);
    // [1, 4]
}

```

// Shreyash Dhekane D20A 23

13: Data Structures - Mapping

```

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
        // even when it is not initialized
        return nested[_addr1][_i];
    }

    function set(address _addr1, uint _i, bool _boo) public {
        nested[_addr1][_i] = _boo;
    }

    function remove(address _addr1, uint _i) public {
        delete nested[_addr1][_i];
    }
}

// Shreyash Dhekane D20A 23

```

14: Data Structures - Structs

```

1 // SPDX-License-Identifier: MIT
2
3 pragma solidity ^0.8.1;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53

```

15: Data structures - Enums

```

1 // SPDX-License-Identifier: MIT
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

```

16: Data Locations

```

1 // SPDX-License-Identifier: MIT
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

```

17: Transactions

The screenshot shows the Remix Ethereum IDE interface. The left sidebar displays the 'LEARNETH' tutorial, specifically section 10.1 Transactions - Ether and Wei. The main code editor contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract EtherUnits {
5     uint public oneWei = 1 wei;
6     // 1 wei is equal to 1
7     bool public isOneWei = 1 wei == 1;
8
9     uint public oneEther = 1 ether;
10    // 1 ether is equal to 10^18 wei
11    bool public isOneEther = 1 ether == 1e18;
12
13    uint public oneGwei = 1 gwei;
14    // 1 ether is equal to 10^9 wei
15    bool public isOneGwei = 1 gwei == 1e9;
16 }
17
18 // Shreyash Dhekane D20A 23

```

Below the code, there are 'Check Answer' and 'Show answer' buttons, and a message saying 'Well done! No errors.'

18: Transactions

The screenshot shows the Remix Ethereum IDE interface. The left sidebar displays the 'LEARNETH' tutorial, specifically section 10.2 Transactions - Gas and Gas Price. The main code editor contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Gas {
5     uint public i = 0;
6     uint public cost = 170367;
7
8     // Using up all of the gas that you send causes your transaction to fail.
9     // State changes are undone.
10    // Gas spent are not refunded.
11    function forever() public {
12        // Here we run a loop until all of the gas are spent
13        // and the transaction fails
14        while (true) {
15            i++;
16        }
17    }
18
19 // Shreyash Dhekane D20A 23

```

Below the code, there are 'Check Answer' and 'Show answer' buttons, and a message saying 'Well done! No errors.'

19: Transactions

The screenshot shows the Remix Ethereum IDE interface. The left sidebar displays the 'LEARNETH' tutorial, specifically section 10.3 Transactions - Sending Ether. The main code editor contains the following Solidity code:

```

51 }
52
53 contract Charity {
54     address public owner;
55
56     constructor() {
57         owner = msg.sender;
58     }
59
60     function donate() public payable {}
61
62     function withdraw() public {
63         uint amount = address(this).balance;
64
65         (bool sent, bytes memory data) = owner.call{value: amount}("");
66         require(sent, "Failed to send Ether");
67     }
68
69 // Shreyash Dhekane D20A 23

```

Below the code, there are 'Check Answer' and 'Show answer' buttons, and a message saying 'Well done! No errors.'

Conclusion:

This experiment provided a comprehensive immersion into the Solidity programming language, bridging the gap between theoretical blockchain architecture and practical software engineering. By utilizing the Remix IDE, a sophisticated development environment, a wide array of technical constructs were explored and implemented, including state variables, complex data structures like mappings and structs, and the nuances of visibility specifiers and custom modifiers. This hands-on approach allowed for a granular examination of the smart contract lifecycle—from the initial state defined by constructors to the execution of complex logic via control flow statements—ensuring a robust understanding of how code behaves within a decentralized context.