

Experiment No. 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:

- o internal: accessible within the contract and its child contracts.
- o external: can be called only by external accounts or other contract
- **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 ($1 \text{ Ether} = 10^{18} \text{ Wei}$). 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.

Implementation:

- Tutorial no. 1 – Compile the code



- Tutorial no. 1 – Deploy the contract

The screenshot displays the Remix IDE interface with the following components:

- Left Panel (Deploy & Run Transactions):**
 - ENVIRONMENT:** Remix VM (Osaka)
 - ACCOUNT:** 0x5B3...eddC4 (99.999999999)
 - GAS LIMIT:** Estimated Gas (3000000)
 - VALUE:** 0 Wei
 - CONTRACT:** Counter - remix-project-org/remix-w
 - Deploy Button:** Orange button at the bottom.
- Top Right Panel (Code Editor):**

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     uint public count;
6
7     // Function to get the current count
8     function get() public view returns (uint) {
9         return count;
10    }
11    // Anushka Shahane D20A 56
12

```
- Bottom Panel (Terminal/Logs):**
 - Terminal:** Shows the command prompt and the result of the deployment: `[vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei data: 0x08...f0033 logs: 0 hash: 0xb4a...d6ab3`
 - Logs:**
 - status:** 1 Transaction mined and execution succeed
 - transaction hash:** 0xb4a6d665711c9a9dd7babf7404a527015b9c94da0dc6768f62e17187d8d6ab3
 - block hash:** 0x367acf1c2046c1ddc0cc082ebd86a983d80a85b65324b7504fb67b0f30f04c8
 - block number:** 1
 - contract address:** 0xd9145CCES20386f254917e481e044e9943f39138
 - from:** 0x5B380a6a701c568545dCfc803Fc8875f56beddC4
 - to:** Counter.(constructor)
 - transaction cost:** 155067 gas
 - execution cost:** 94739 gas
 - output:** A long hexadecimal string representing the contract's state.

- Tutorial no. 1 – get

Deployed Contracts 1

COUNTER AT 0XD91...39138

Balance: 0 ETH

dec

inc

count

get

get - call

0: uint256: 0

Low level interactions

CALLDATA

Transact

7 // Function to get the current count

AI copilot

0 Listen on all transactions Filter with transaction hash or ad...

decoded output

logs

raw logs

call to counter.get

CALL [call] from: 0x58380a6a701c568545dCfc803Fc8875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

call to counter.get

CALL [call] from: 0x58380a6a701c568545dCfc803Fc8875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

Debug

Debug

Deployed Contracts 1

COUNTER AT 0XD91...39138

Balance: 0 ETH

dec

inc

count

get

0: uint256: 0

Low level interactions

CALLDATA

Transact

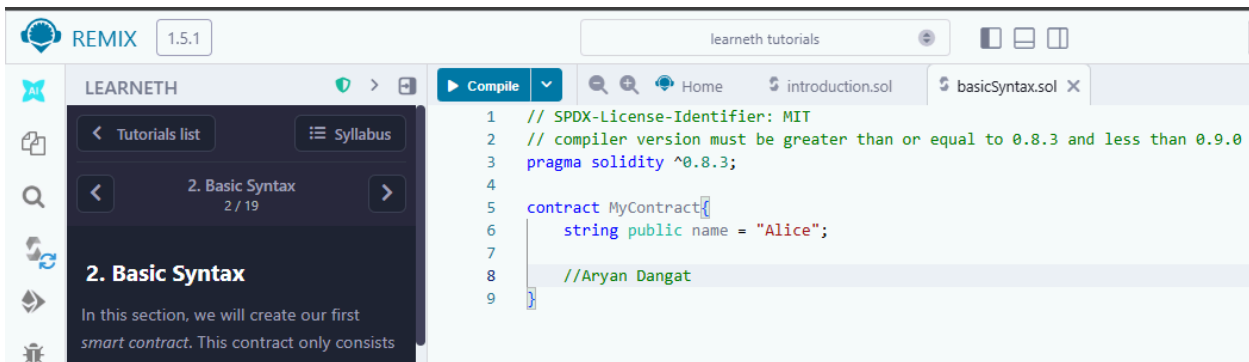
Tutorial no. 1 – Increment

The image shows a web interface for a Counter contract and its transaction details. The top part is a window titled "COUNTER AT 0XD91...39138". It displays "Balance: 0 ETH" and four buttons: "dec", "inc", "count", and "get". Below the buttons, it shows "0: uint256: 0". There is a "Low level interactions" section with a "CALLDATA" input field and a "Transact" button. The bottom part is a transaction details window titled "[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0x3ec...ee5b5". It includes a "Debug" button and a list of transaction details: status (1 Transaction mined and execution succeed), transaction hash (0x3ec3575f0aea03fb5ef9916d2f33b896b1412c58a5696c9acb508458012ee5b5), block hash (0x02b61e069a31beda4a72dbd7b9a1c01eded4c0f050f9151afe476e4a7946758c), block number (2), from (0x5B380Da6a701c568545dCfcB875F56beddC4), to (Counter.inc() 0xd9145CCE520386f254917e481e844e9943f39138), and transaction cost (43517 gas).

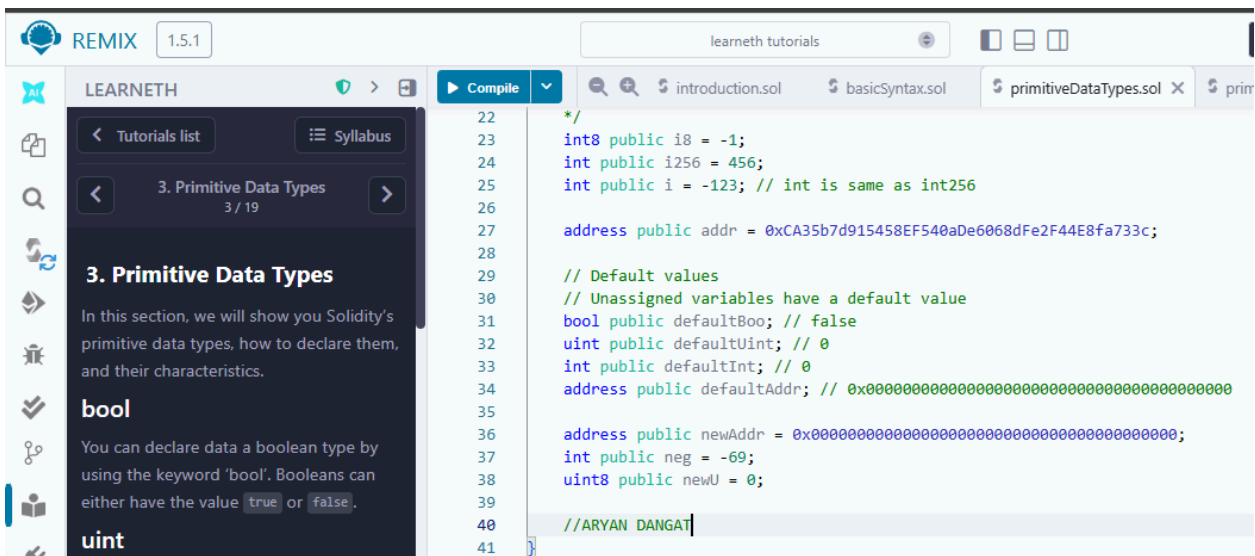
- Tutorial no. 1 – Decrement

The image shows a web interface for a Counter contract. The window is titled "Deployed Contracts 1" and "COUNTER AT 0XSE1...4EFF5 (MEMORY)". It displays "Balance: 0 ETH" and four buttons: "dec", "inc", "count", and "get". Below the buttons, it shows "0: uint256: 1". There is a "Low level interactions" section with a "CALLDATA" input field and a "Transact" button. A "get - call" button is also visible next to the "get" button.

- Tutorial no. 2



- Tutorial no. 3



- Tutorial no. 4

4. Variables
4 / 19

There are three different types of variables in Solidity: *State Variables*, *Local Variables*, and *Global Variables*.

1. State Variables

State Variables are stored in the contract storage and thereby on the blockchain. They are declared inside the contract but outside the function. This contract has two state variables, the string `text` (line 6)

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
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        //Aryan Dangat
20    }
21

```

- Tutorial no. 5

5.1 Functions - Reading and Writing to a State Variable
5 / 19

This section will give a short introduction to functions and teach you how to use them to read from and write to a state variable.

As in other languages, we use functions in Solidity to create modular, reusable code. However, Solidity functions have some particularities.

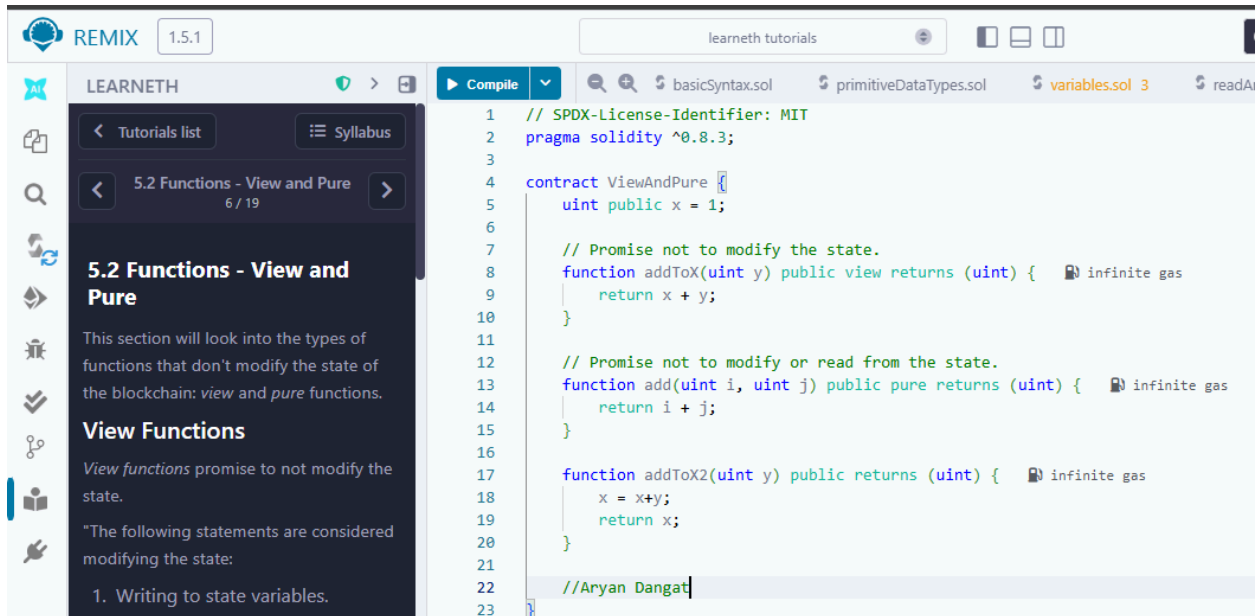
Solidity functions can be split into two types:

```

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    //Aryan Dangat
24

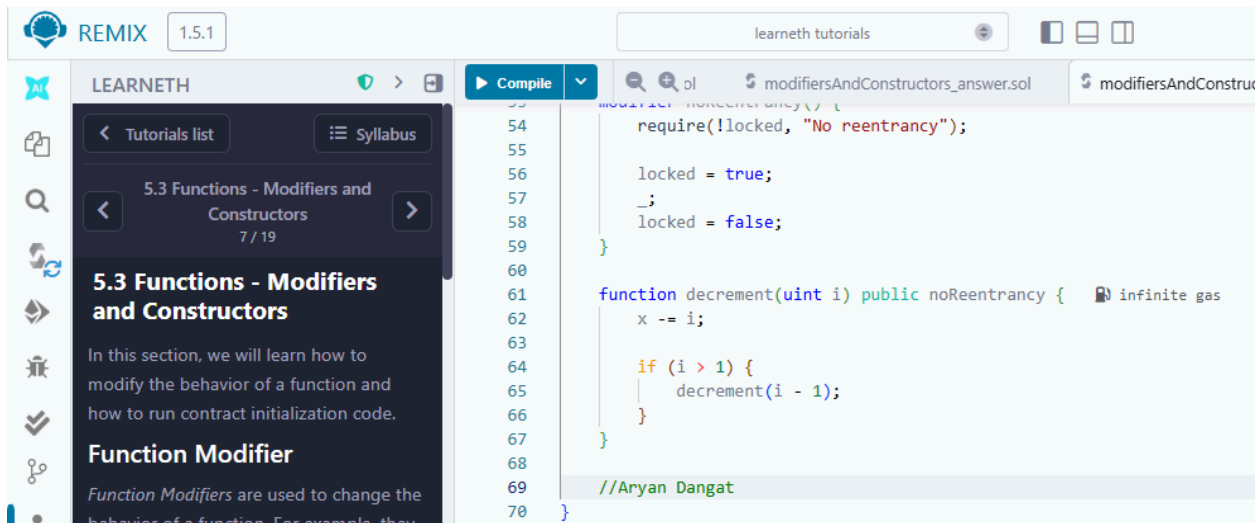
```

- Tutorial no. 6



```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract ViewAndPure {
5     uint public x = 1;
6
7     // Promise not to modify the state.
8     function addToX(uint y) public view returns (uint) { infinite gas
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) { infinite gas
14        return i + j;
15    }
16
17    function addToX2(uint y) public returns (uint) { infinite gas
18        x = x+y;
19        return x;
20    }
21
22    //Aryan Dangat
23 }
```

- Tutorial no. 7



```
54
55
56     locked = true;
57     _;
58     locked = false;
59 }
60
61 function decrement(uint i) public noReentrancy { infinite gas
62     x -= i;
63
64     if (i > 1) {
65         decrement(i - 1);
66     }
67 }
68
69 //Aryan Dangat
70 }
```

- Tutorial no. 8

REMIX 1.5.1 learneth tutorials

LEARNETH

Tutorials list Syllabus

5.4 Functions - Inputs and Outputs 8 / 19

5.4 Functions - Inputs and Outputs

In this section, we will learn more about the inputs and outputs of functions.

Multiple named Outputs

Functions can return multiple values that can be named and assigned to their

```
//  
78  
79  
80  
81 function returnTwo() 472 gas  
82     public  
83     pure  
84     returns (  
85         int i,  
86         bool b  
87     )  
88     {  
89         i = -2;  
90         b = true;  
91     }  
92 //Aryan Dangat  
93 }
```

- Tutorial no. 9

REMIX 1.5.1 learneth tutorials Login with GitHub

LEARNETH

Tutorials list Syllabus

6. Visibility 9 / 19

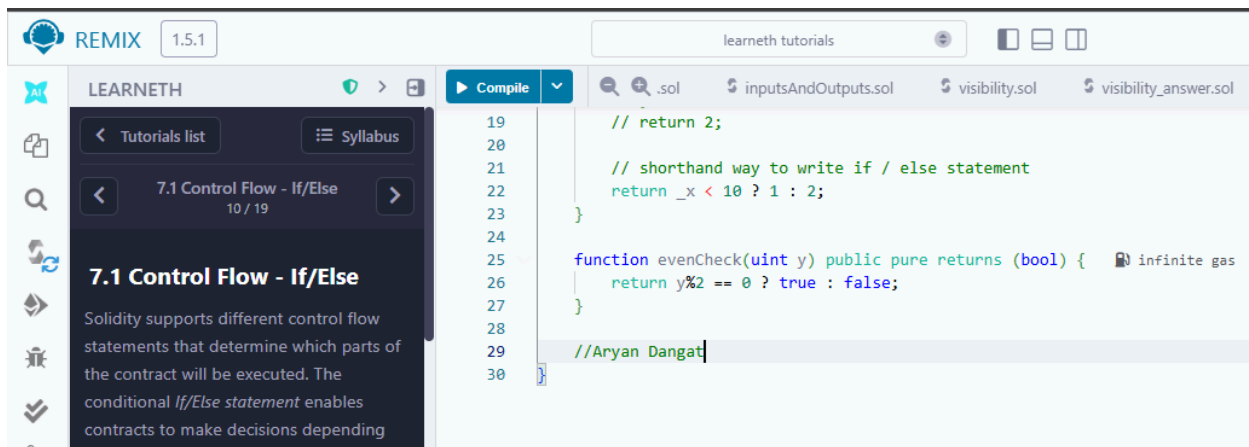
6. Visibility

The `visibility` specifier is used to control who has access to functions and state variables.

There are four types of visibilities: `external`, `public`, `internal`, and `private`.

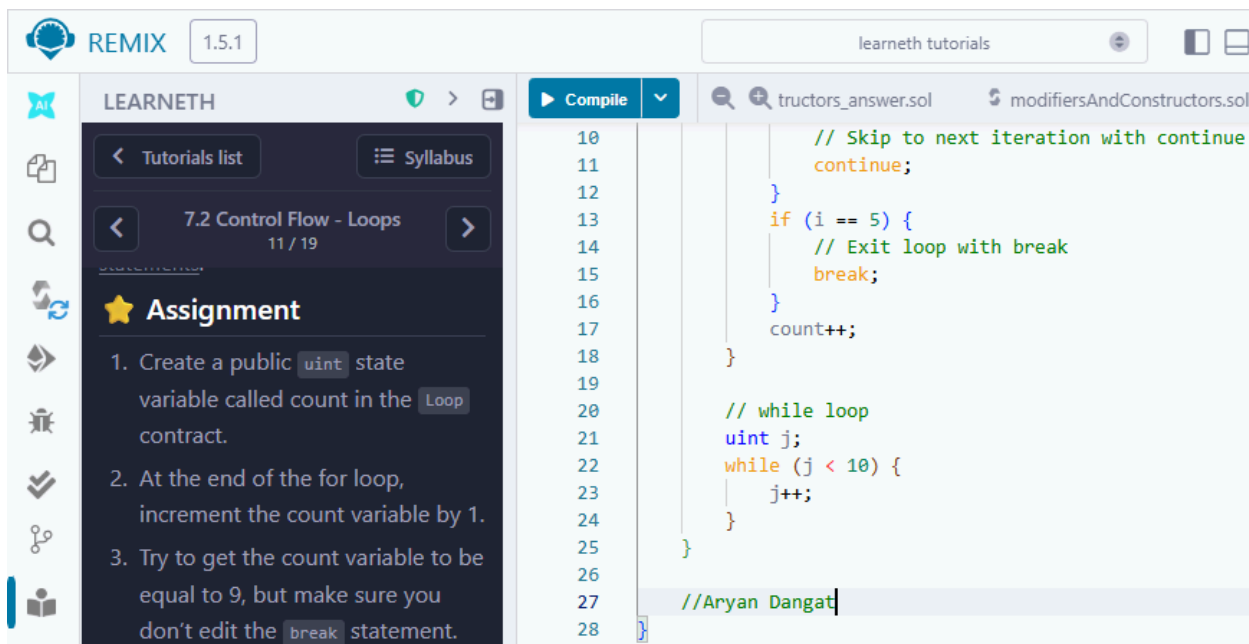
```
58 // function testPrivateFunc() public pure returns (string memory) {  
59     // return privateFunc();  
60 }  
61  
62 // Internal function call be called inside child contracts.  
63 function testInternalFunc() public pure override returns (string memory) { infinite gas  
64     return internalFunc();  
65 }  
66  
67 function testInternalVar() public view returns (string memory, string memory) { infinite gas  
68     return (internalVar, publicVar);  
69 }  
70  
71 //Aryan Dangat  
72 }
```

- Tutorial no. 10



```
19 // return 2;
20
21 // shorthand way to write if / else statement
22 return _x < 10 ? 1 : 2;
23 }
24
25 function evenCheck(uint y) public pure returns (bool) {
26     return y%2 == 0 ? true : false;
27 }
28
29 //Aryan Dangat
30 }
```

- Tutorial no. 11



```
10 // Skip to next iteration with continue
11 continue;
12 }
13 if (i == 5) {
14     // Exit loop with break
15     break;
16 }
17 count++;
18 }
19
20 // while loop
21 uint j;
22 while (j < 10) {
23     j++;
24 }
25
26
27 //Aryan Dangat
28 }
```

- Tutorial no. 12

LEARNETH

Tutorials list | Syllabus

8.1 Data Structures - Arrays
12 / 19

important, then we can move the last element of the array to the place of the deleted element (line 46), or use a mapping. A mapping might be a better choice if we plan to remove elements in our data structure.

Array length

Using the length member, we can read the number of elements that are stored in an array (line 35).

[Watch a video tutorial on Arrays.](#)

★ Assignment

1. Initialize a public fixed-sized array called `arr3` with the values 0, 1, 2. Make the size as small as possible.
2. Change the `getArr()` function to return the value of `arr3`.

[Check Answer](#) [Show answer](#)

Next

Well done! No errors.

```

52 // move the last element into the place to delete.
53 function remove(uint index) public {
54     // Move the last element into the place to delete
55     arr[index] = arr[arr.length - 1];
56     // Remove the last element
57     arr.pop();
58 }
59 //Anushka Shahane D20A 56
60 function test() public {
61     arr.push(1);
62     arr.push(2);
63     arr.push(3);
64 }

```

AI copilot

0 Listen on all transactions

Filter with transaction hash or address

logs

raw logs

transact to Counter.dec pending ...

[vm] from: 0x583...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei
data: 0xb3b...cfa82 logs: 0 hash: 0xcc...79d46

Debug

- Tutorial no. 13

REMIX 1.5.1

learneth tutorials

LEARNETH

Tutorials list | Syllabus

8.2 Data Structures - Mappings
13 / 19

8.2 Data Structures - Mappings

In Solidity, *mappings* are a collection of key types and corresponding value type pairs.

```

35
36
37
38
39
40
41
42
43 function set(address _addr1,
44             uint _i,
45             bool _boo
46             ) public {
47     nested[_addr1][_i] = _boo;
48 }

```

25199 gas

```

43 function remove(address _addr1, uint _i) public {
44     delete nested[_addr1][_i];
45 }

```

25045 gas

//Aryan Dangat

- Tutorial no. 14

REMIX 1.5.1 learneth tutorials

LEARNETH

Tutorials list Syllabus

8.3 Data Structures - Structs 14 / 19

8.3 Data Structures - Structs

In Solidity, we can define custom data types in the form of *structs*. Structs are a collection of variables that can consist of

```
44
45     todo storage todo = todos[_index];
46     todo.completed = !todo.completed;
47 }
48
49 function remove(uint _index) public { infinite gas
50     delete todos[_index];
51 }
52 //Aryan Dangat
53 }
```

- Tutorial no. 15

REMIX 1.5.1 learneth tutorials

LEARNETH

Tutorials list Syllabus

8.4 Data Structures - Enums 15 / 19

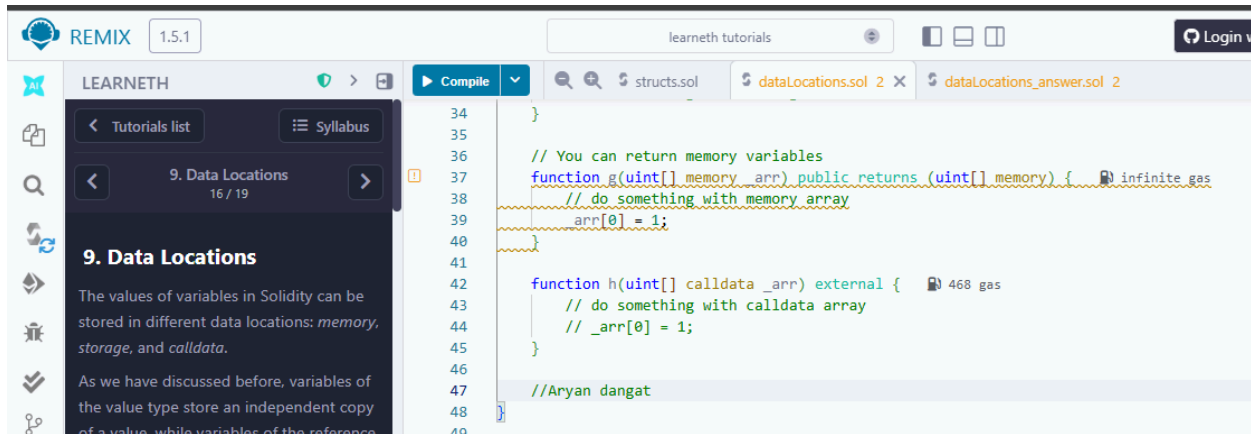
8.4 Data Structures - Enums

In Solidity *enums* are custom data types consisting of a limited set of constant values. We use enums when our variables should only get assigned a value from a predefined set of values.

In this contract, the state variable `status`

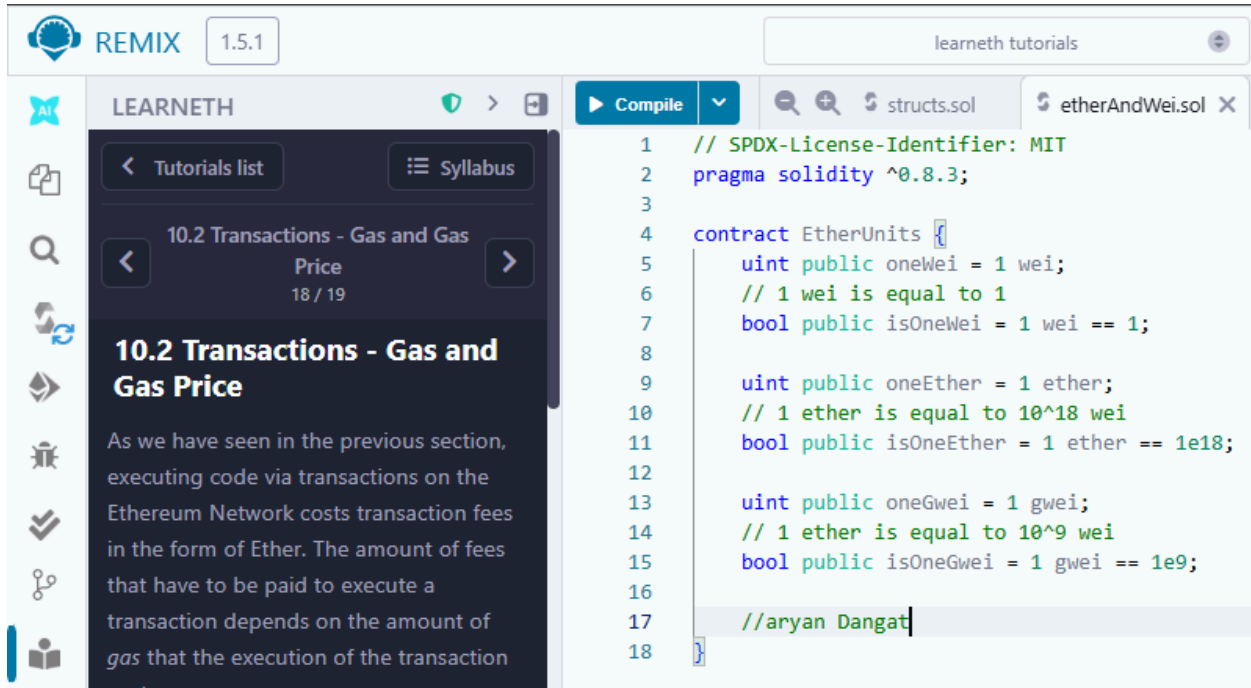
```
35     status = _status;
36 }
37
38 // You can update to a specific enum like this
39 function cancel() public { 24494 gas
40     status = Status.Canceled;
41 }
42
43 // delete resets the enum to its first value, 0
44 function reset() public { 24383 gas
45     delete status;
46 }
47
48 //Aryan Dangat
49 }
```

- Tutorial no. 16



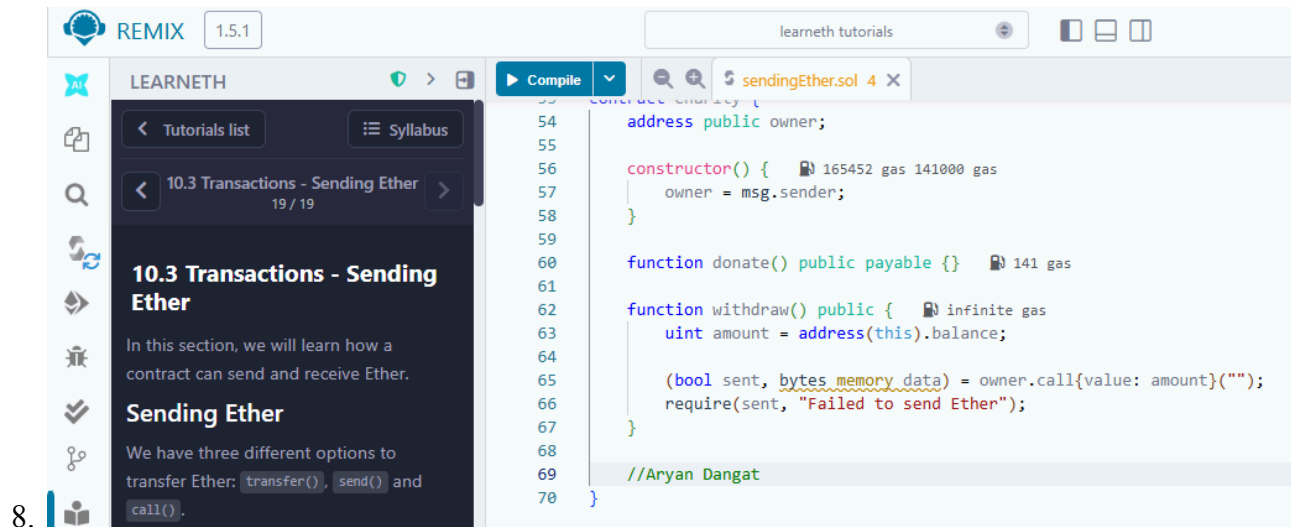
```
34 }
35
36 // You can return memory variables
37 function g(uint[] memory arr) public returns (uint[] memory) { infinite gas
38     // do something with memory array
39     arr[0] = 1;
40 }
41
42 function h(uint[] calldata _arr) external { 468 gas
43     // do something with calldata array
44     _arr[0] = 1;
45 }
46
47 //Aryan dangat
48
49 }
```

- Tutorial no. 18



```
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    //aryan Dangat
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.

Aryan Dangat D20A-19