

## 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 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.

## Implementation:

- Tutorial no. 1 – Compile the code

The screenshot shows the Remix IDE interface. At the top, there is a navigation bar with tabs for 'Compiled' (which is selected), 'Home', and a file tab labeled '\$ introduction.sol X'. Below the navigation bar is a code editor containing the following Solidity code:

```
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) { 2453 gas
9         return count;
10    // Anushka Shahane D20A 56
11 }
12 }
```

Below the code editor, there is a button labeled 'Explain contract'. To the right of the code editor, there are several status indicators: '0' (tx count), a checkbox for 'Listen on all transactions', a search icon, a filter input field with placeholder 'Filter with transaction hash or address...', and a 'Close' button.

The main workspace below the status bar displays a welcome message: "Welcome to Remix 1.5.1". It also shows information about indexedDB storage: "Your files are stored in indexedDB, 84.21 KB / 142.59 GB used". A terminal-like interface follows, with instructions for using the terminal and a list of accessible libraries. The accessible libraries include "ethers.js". A command-line input field at the bottom is populated with the placeholder "Type the library name to see available commands.".

- Tutorial no. 1 – Deploy the contract

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons for file operations like Deploy, Run, and Save. The main area has tabs for 'DEPLOY & RUN TRANSACTIONS' and 'ENVIRONMENT'. Under 'ENVIRONMENT', 'Remix VM (Osaka)' is selected. Below it, there's a button for 'Authorize Delegation'. A 'GAS LIMIT' section shows 'Estimated Gas' set to 3000000 and 'Custom' set to 0 Wei. The 'CONTRACT' section shows 'Counter - remix-project-org/remix-w' and 'evm version: osaka'. At the bottom, a 'Deploy' button is visible.

The right side displays the Solidity code for a 'Counter' contract:

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

contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }
}

// Anushka Shahane D20A 56
```

Below the code, there's an 'Explain contract' button. The terminal window at the bottom shows a successful transaction for deploying the contract, with details like status, transaction hash, block hash, block number, contract address, from, to, transaction cost, execution cost, and output.

- Tutorial no. 1 – get

# Aryan Dangat D20A-19

The screenshot shows the Truffle UI interface for a deployed Ethereum contract. The main window displays the contract's methods: `dec`, `inc`, `count`, and `get`. The `get` method is currently selected, showing its return value as `0: uint256: 0`. Below the methods, there are sections for `Low level interactions` and `CALldata`, each with a `Transact` button.

**Deployed Contracts** (1)

**COUNTER AT 0xD91...39138**

Balance: 0 ETH

methods:

- dec
- inc
- count
- get

0: uint256: 0

Low level interactions

CALldata

Transact

**Explain contract**

// Function to get the current count

decoded output

logs

raw logs

call to Counter.get

CALL [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

call to Counter.get

CALL [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

Debug

AI copilot

## Tutorial no. 1 – Increment

The screenshot shows the Truffle UI interface for a deployed contract named "COUNTER AT 0xD91...39138".

**Contract Interface:**

- Balance:** 0 ETH
- Buttons:** dec, inc, count, get
- Output:** 0: uint256: 0

**Low level interactions:**

- CALldata:** [empty]
- Transact:** [button]

**Transaction Details:**

- [vm] from:** 0x5B3...eddC4 **to:** Counter.inc() **0xd91...39138** **value:** 0 **wei data:** 0x371...303c0 **logs:** 0 **hash:** 0x3ec...ee5b5
- status:** 1 Transaction mined and execution succeed
- transaction hash:** 0x3ec3575f0aea03fb5ef9916d2f33b896b1412c58a5696c9acb508458012ee5b5
- block hash:** 0x92b61e869a31beda4a72dbd7b9a1c01edad4c0f050f9151afe476e4a7946758c
- block number:** 2
- from:** 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
- to:** Counter.inc() 0xd9145CCE52D386F254917e481e844e9943F39138
- transaction cost:** 43517 gas

- Tutorial no. 1 – Decrement

The screenshot shows the Truffle UI interface for a deployed contract named "COUNTER AT 0X5E1...4EFF5 (MEMORY)".

**Contract Interface:**

- Balance:** 0 ETH
- Buttons:** dec, inc, count, get
- Output:** 0: uint256: 1

**Low level interactions:**

- CALldata:** [empty]
- Transact:** [button]

- Tutorial no. 2

The screenshot shows the REMIX IDE interface. The top bar displays "REMX 1.5.1" and "learneth tutorials". The left sidebar has icons for AI, Tutorials list, Syllabus, and a search bar. The main workspace shows a Solidity code editor with the following content:

```
1 // SPDX-License-Identifier: MIT
2 // compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
3 pragma solidity ^0.8.3;
4
5 contract MyContract{
6     string public name = "Alice";
7
8     //Aryan Dangat
9 }
```

- Tutorial no. 3

The screenshot shows the REMIX IDE interface. The top bar displays "REMX 1.5.1" and "learneth tutorials". The left sidebar has icons for file operations like New, Open, Save, etc., and sections for Tutorials list, Syllabus, and the current topic, 3. Primitive Data Types. The main workspace shows a Solidity code editor with the following content:

```
/*
int8 public i8 = -1;
int public i256 = 456;
int public i = -123; // int is same as int256

address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

// Default values
// Unassigned variables have a default value
bool public defaultBool; // false
uint public defaultUint; // 0
int public defaultInt; // 0
address public defaultAddr; // 0x0000000000000000000000000000000000000000000000000000000000000000

address public newAddr = 0x0000000000000000000000000000000000000000000000000000000000000000;
int public neg = -69;
uint8 public newU = 0;

//ARYAN DANGAT|
```

- Tutorial no. 4

```

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 { 22334 gas
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
20 } //Aryan Dangat
21

```

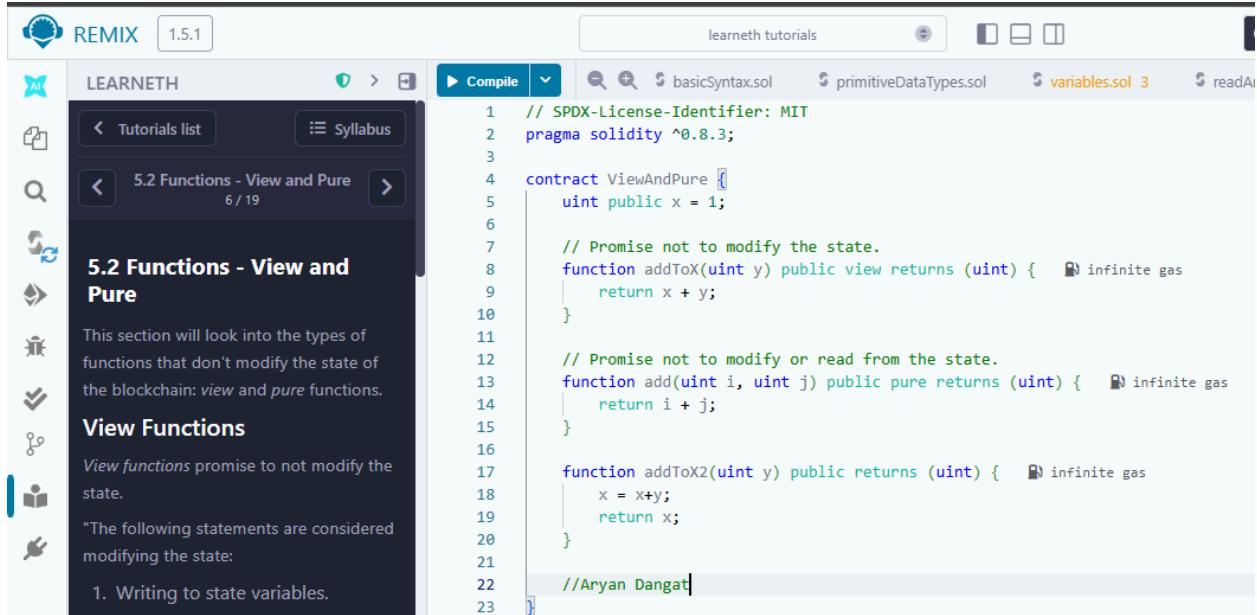
- Tutorial no. 5

```

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 { 22536 gas
11        num = _num;
12    }
13
14    // You can read from a state variable without sending a transaction.
15    function get() public view returns (uint) { 2475 gas
16        return num;
17    }
18
19    function get_b() public view returns (bool){ 2539 gas
20        return b;
21    }
22
23
24 } //Aryan Dangat
25

```

- Tutorial no. 6



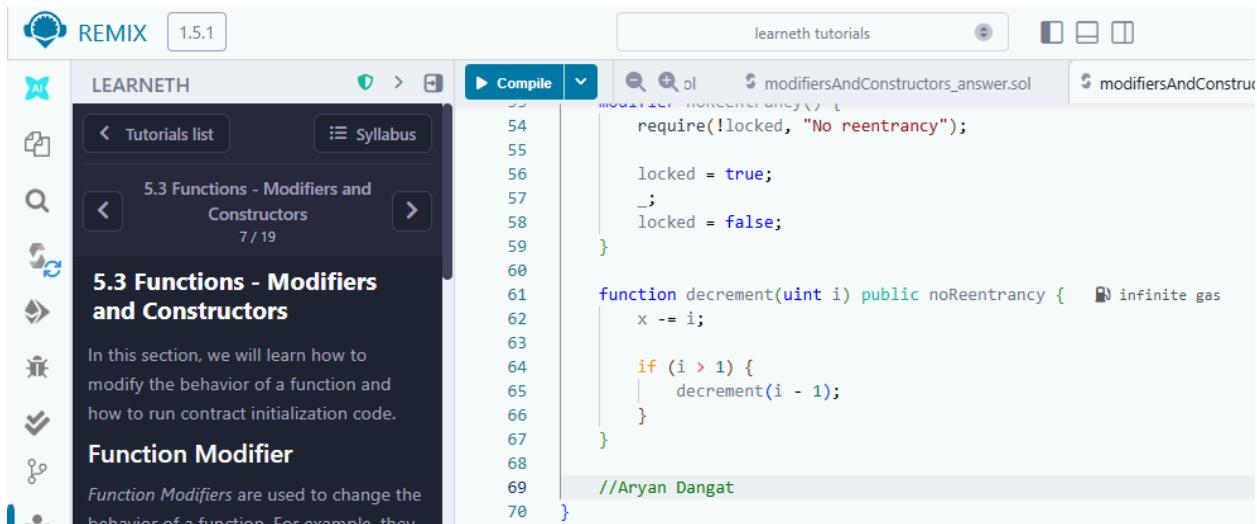
The screenshot shows the REMIX IDE interface. The left sidebar displays a navigation tree for 'LEARNETH' tutorials, specifically '5.2 Functions - View and Pure'. The main code editor area contains the following Solidity code:

```

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



The screenshot shows the REMIX IDE interface. The left sidebar displays a navigation tree for 'LEARNETH' tutorials, specifically '5.3 Functions - Modifiers and Constructors'. The main code editor area contains the following Solidity code:

```

33
34     require(!locked, "No reentrancy");
35
36     locked = true;
37     ;
38     locked = false;
39 }
40
41 function decrement(uint i) public noReentry {    infinite gas
42     x -= i;
43
44     if (i > 1) {
45         decrement(i - 1);
46     }
47 }
48
49 //Aryan Dangat
50

```

- Tutorial no. 8

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons and a main panel titled "LEARNETH" displaying a tutorial on "5.4 Functions - Inputs and Outputs". The main workspace contains the following Solidity code:

```
//
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93 }
```

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

```
//Aryan Dangat
```

The code defines a function `returnTwo()` that returns two values: `i` (an integer) and `b` (a boolean). The `i` variable is initialized to `-2`, and `b` is set to `true`. A comment at the end of the file reads `//Aryan Dangat`.

- Tutorial no. 9

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons and a main panel titled "LEARNETH" displaying a tutorial on "6. Visibility". The main workspace contains the following Solidity code:

```
// 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);
}
```

```
//Aryan Dangat
```

The code demonstrates the use of visibility specifiers. It includes three functions: `testPrivateFunc()` (private), `testInternalFunc()` (internal), and `testInternalVar()` (public). A comment at the end of the file reads `//Aryan Dangat`.

- Tutorial no. 10

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' section, specifically the '7.1 Control Flow - If/Else' tutorial, which covers control flow statements like if/else. The main workspace contains the following Solidity code:

```
// return 2;
// shorthand way to write if / else statement
return _x < 10 ? 1 : 2;

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

//Aryan Dangat
```

- Tutorial no. 11

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' section, specifically the '7.2 Control Flow - Loops' tutorial, which covers loops like for and while. The main workspace contains the following Solidity code:

```
// Skip to next iteration with continue
continue;
}
if (i == 5) {
    // Exit loop with break
    break;
}
count++;

}

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

//Aryan Dangat
```

- Tutorial no. 12

```

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     arr.push(4);

```

- Tutorial no. 13

```

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

```

- Tutorial no. 14

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, a search bar, and a syllabus section. The main area displays a tutorial titled "8.3 Data Structures - Structs" with the subtitle "14 / 19". The code editor contains the following Solidity code:

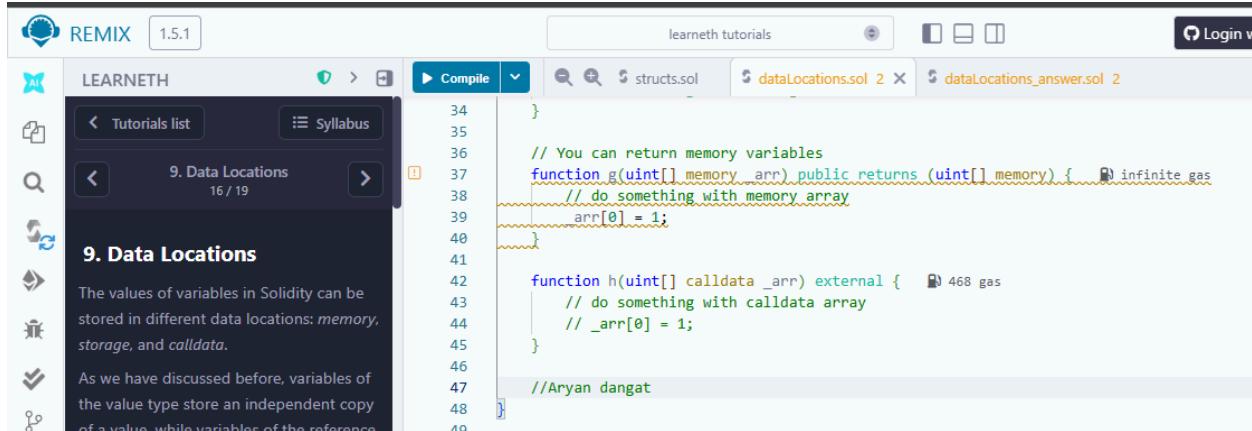
```
44 todo Storage todo = todos[_index],  
45 todo.completed = !todo.completed;  
46 }  
47  
48 function remove(uint _index) public { infinite gas  
49 | delete todos[_index];  
50 }  
51  
52 //Aryan Dangat|  
53 }
```

- Tutorial no. 15

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, a search bar, and a syllabus section. The main area displays a tutorial titled "8.4 Data Structures - Enums" with the subtitle "15 / 19". The code editor contains the following Solidity code:

```
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



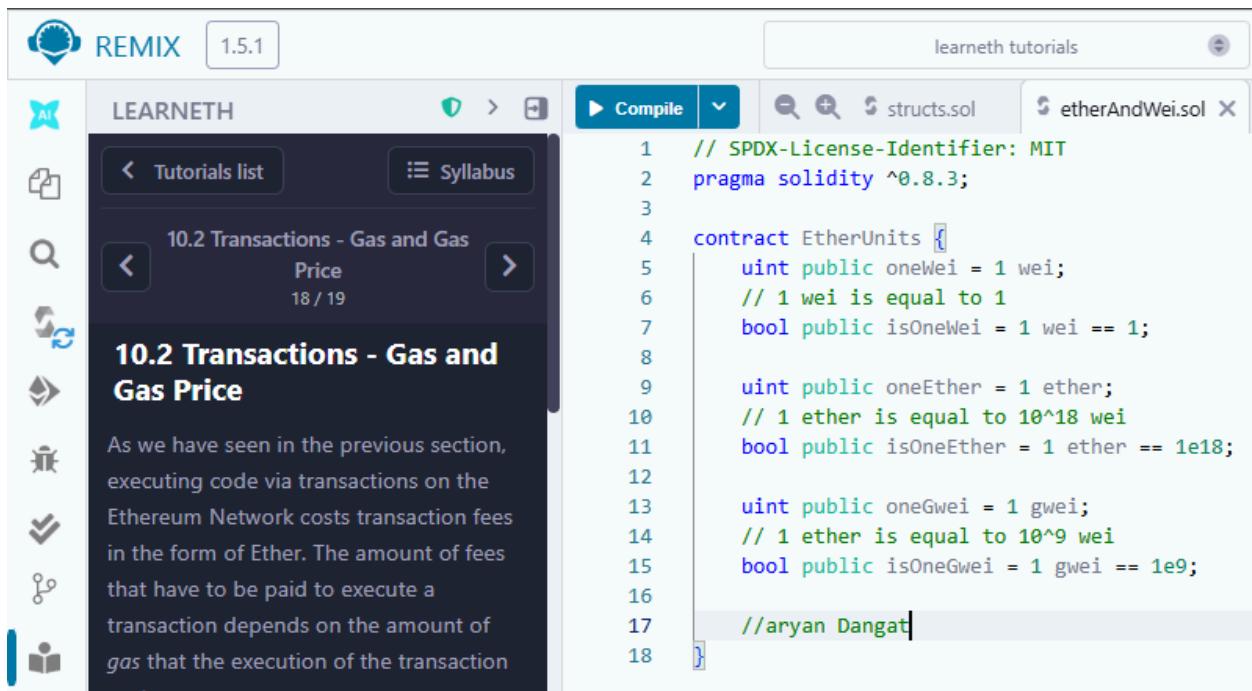
The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, search, and navigation. The main area displays a tutorial titled "9. Data Locations" from "LEARNETH". The code editor contains the following Solidity code:

```

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



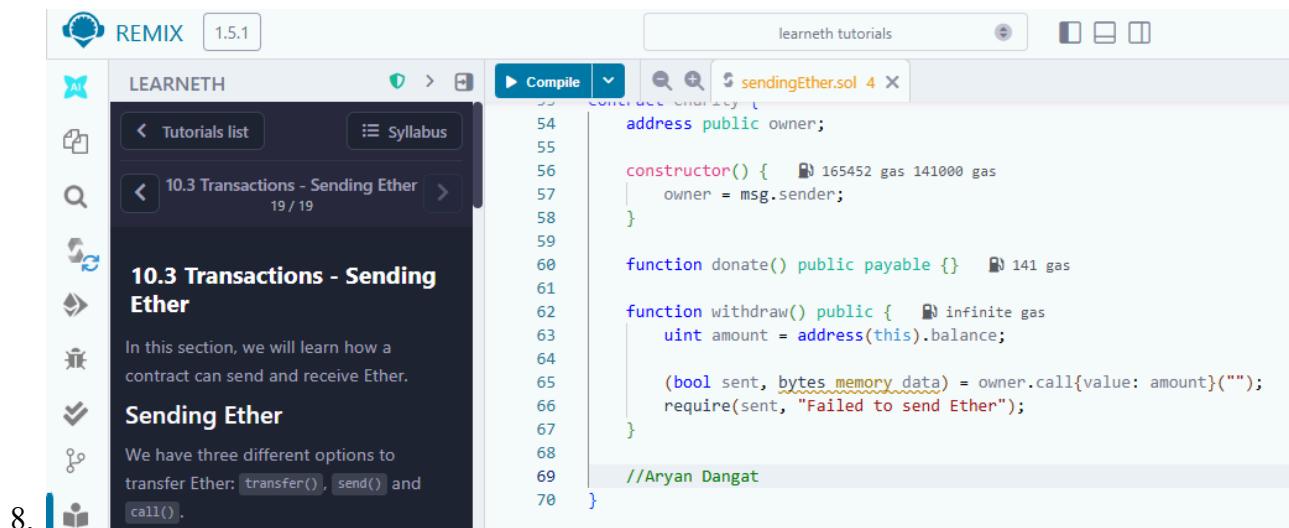
The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, search, and navigation. The main area displays a tutorial titled "10.2 Transactions - Gas and Gas Price" from "LEARNETH". The code editor contains the following Solidity code:

```

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 gwei is equal to 10^9 wei
15    bool public isOneGwei = 1 gwei == 1e9;
16
17
18 //aryan Dangat

```

- Tutorial no. 19



The screenshot shows the Remix IDE interface. On the left, there's a sidebar with icons for AI, Remix, and version 1.5.1. The main area has tabs for 'LEARNETH' and 'Syllabus'. Under 'LEARNETH', it says 'Tutorials list' and '10.3 Transactions - Sending Ether 19 / 19'. The right side is the code editor with the following Solidity code:

```
contract Charly {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function donate() public payable {
        require(msg.value > 0);
    }

    function withdraw() public {
        uint amount = address(this).balance;
        (bool sent, bytes memory data) = owner.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}

//Aryan Dangat
```

**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