

## 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:
  - **public**: available both inside and outside the contract.
  - **private**: only accessible within the same contract.
  - **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. The top bar displays 'REMX 1.5.1' and 'learneth tutorials'. Below the bar, there are tabs for 'Compiled' (selected), 'Home', and two other files: 'introduction.sol' and 'basicSyntax.sol'. The main area contains the Solidity code for the 'introduction.sol' file:

```
// Function to increment count by 1
function inc() public {
    count += 1;
}

// Function to decrement count by 1
function dec() public {
    count -= 1;
}
//Laksh Sodhai D20A 64
```

On the left side, there are various icons for interacting with the contract, such as Deploy, Run, and Explain contract. The 'Explain contract' button is highlighted. At the bottom, there is a terminal window with the following text:

Welcome to Remix 1.5.1  
Your files are stored in indexedDB, 1.22 MB / 2 GB used  
You can use this terminal to:  
• Check transactions details and start debugging.  
• Execute JavaScript scripts:  
 - Input a script directly in the command Line interface  
 - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.executeCurrent()` in the command Line interface  
 - Right-click on a JavaScript file in the file explorer and then click "Run"  
The following libraries are accessible:  
• ethers.js  
Type the library name to see available commands.  
creation of Counter pending...

## Laksh Sodhai D20A 64

- Tutorial no. 1 – Deploy the contract

The screenshot shows the REMIX IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, showing account details (0x5B3...eddC4), gas limit (Custom, 3000000), and value (0 Wei). The 'Deploy' button is highlighted. Below it, 'Transactions recorded' and 'Deployed Contracts' sections are visible. The main right panel displays the Solidity code for a Counter contract:

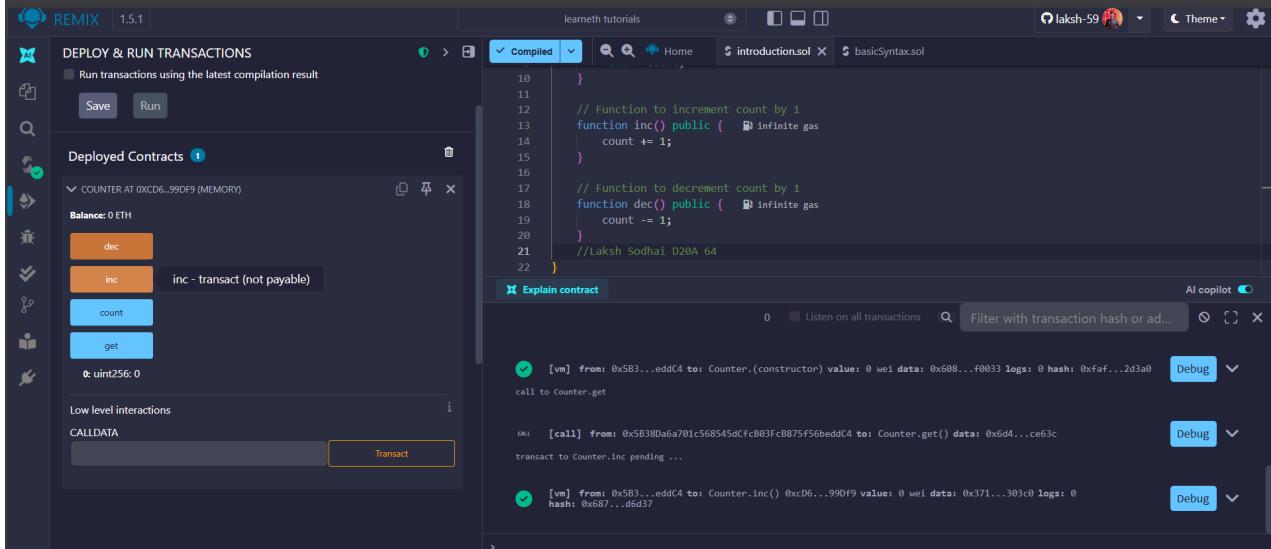
```
10 }
11
12 // Function to increment count by 1
13 function inc() public {
14     count += 1;
15 }
16
17 // Function to decrement count by 1
18 function dec() public {
19     count -= 1;
20 }
21
22 //Laksh Sodhai D20A 64
```

The transaction history pane at the bottom shows a successful deployment of the Counter contract with a pending creation log.

- Tutorial no. 1 – get

The screenshot shows the REMIX IDE interface after the contract has been deployed. The 'DEPLOY & RUN TRANSACTIONS' sidebar now shows 'Run transactions using the latest compilation result' with 'Save' and 'Run' buttons. The 'Deployed Contracts' section shows the deployed Counter contract at address 0xC6D6...99DF9 with a balance of 0 ETH. The 'Low level interactions' section shows the available functions: 'dec', 'inc', 'count', 'get', and 'get - call'. The transaction history pane at the bottom shows three successful interactions: the deployment of the contract, a call to its constructor, and a call to its 'get' function.

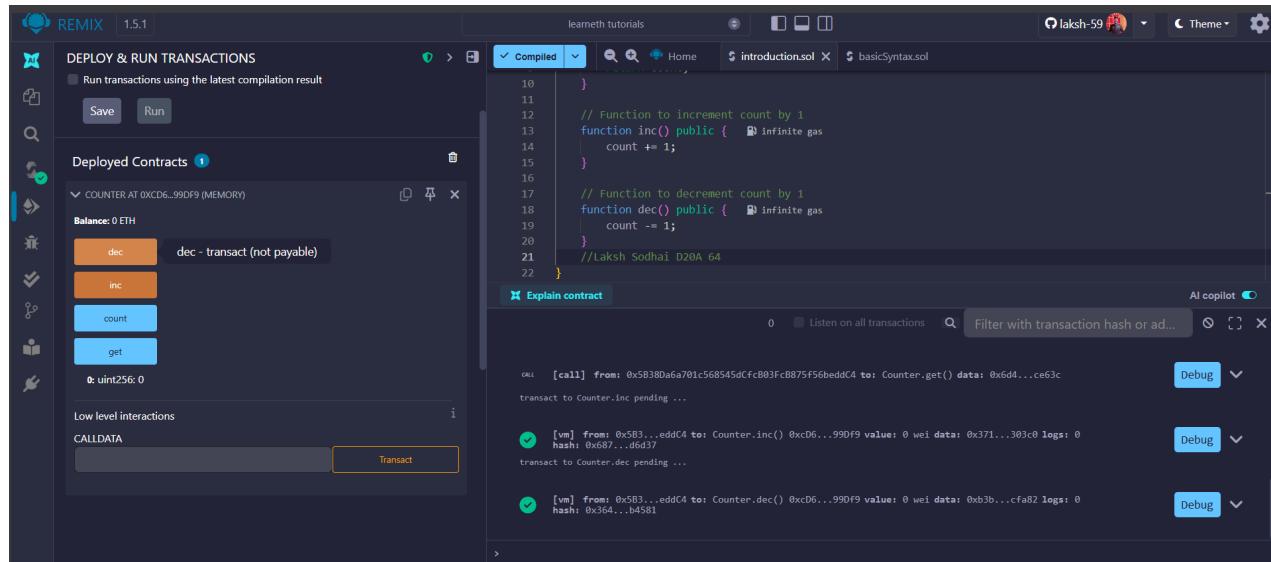
## Tutorial no. 1 – Increment



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

- Deploy & Run Transactions:** Shows the latest compilation result.
- Deployed Contracts:** Displays the `COUNTER` contract at address `0xCd6...990f9` (MEMORY). It has a balance of `0 ETH`.
  - Functions:** `dec`, `inc` (highlighted in orange), `count`, `get`.
  - Low level interactions:** `CALLDATA` section with a `Transact` button.
- Code Editor:** Shows the Solidity code for the `Counter` contract, including the `inc` function which increments the count by 1.
- Logs:** Shows transaction logs for calls to `get`, `inc`, and `dec` functions.

## ● Tutorial no. 1 – Decrement



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

- Deploy & Run Transactions:** Shows the latest compilation result.
- Deployed Contracts:** Displays the `COUNTER` contract at address `0xCd6...990f9` (MEMORY). It has a balance of `0 ETH`.
  - Functions:** `dec` (highlighted in orange), `inc`, `count`, `get`.
  - Low level interactions:** `CALLDATA` section with a `Transact` button.
- Code Editor:** Shows the Solidity code for the `Counter` contract, including the `dec` function which decrements the count by 1.
- Logs:** Shows transaction logs for calls to `get`, `inc`, and `dec` functions.

● Tutorial no.

2

The screenshot shows the REMIX IDE interface. On the left, the LEARNETH tutorial list indicates the user is on section 2. Basic Syntax, step 2/19. The main area displays the following Solidity code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
// Laksh Sodhai D20A 64
contract MyContract {
    string public name = "Alice";
}
```

Below the code, a note states: "We also define the *visibility* of the variable, which specifies from where you can access it. In this case, it's a `public` variable that you can access from inside and outside the contract." A note below that says: "Don't worry if you didn't understand some concepts like *visibility*, *data types*, or *state variables*. We will look into them in the following sections." Another note says: "To help you understand the code, we will link in all following sections to video tutorials from the *creator* of the Solidity by Example contracts." A link to a video tutorial on Basic Syntax is provided.

**Assignment**

1. Delete the HelloWorld contract and its content.
2. Create a new contract named "MyContract".
3. The contract should have a public state variable called "name" of the type string.
4. Assign the value "Alice" to your new variable.

Buttons for "Check Answer" and "Show answer" are present, along with a "Next" button at the bottom.

On the right, the Explain contract interface is open, showing three transaction logs:

- [call] from: 0x5B38Da6a701c568545dCfcB03FcB875F56beddC4 to: Counter.get() data: 0x6d4...ce63c
- [vm] from: 0x5B3...eddC4 to: Counter.inc() 0xcD6...990f9 value: 0 wei data: 0x371...303c0 logs: 0
- [vm] from: 0x5B3...eddC4 to: Counter.dec() 0xcD6...990f9 value: 0 wei data: 0xb3b...cfab82 logs: 0

● Tutorial no. 3

The screenshot shows the REMIX IDE interface. On the left, the LEARNETH tutorial list indicates the user is on section 3. Primitive Data Types, step 3/19. The main area displays the following Solidity code:

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

contract Primitives {
    address public addr = 0xCA35b7d915458EF540a0e6068dFe2F44E8fa733c;
    address public newAddr = 0x5B38Da6a701c568545dCfcB03FcB875F56beddC4;
    // Laksh Sodhai D20A 64

    int public neg = -1;
    uint8 public newU = 0;
```

Below the code, a note says: "Later in the course, we will look at data structures like **Mappings**, **Arrays**, **Enums**, and **Structs**." A link to a video tutorial on Primitive Data Types is provided.

**Assignment**

1. Create a new variable `newAddr` that is a `public address` and give it a value that is not the same as the available variable `addr`.
2. Create a `public` variable called `neg` that is a negative number, decide upon the type.
3. Create a new variable, `newU` that has the smallest `uint` size type and the smallest `uint` value and is `public`.

A tip at the bottom says: "Tip: Look at the other address in the contract or search the internet for an Ethereum address."

Buttons for "Check Answer" and "Show answer" are present, along with a "Next" button at the bottom.

On the right, the Explain contract interface is open, showing three transaction logs:

- [call] from: 0x5B38Da6a701c568545dCfcB03FcB875F56beddC4 to: Counter.get() data: 0x6d4...ce63c
- [vm] from: 0x5B3...eddC4 to: Counter.inc() 0xcD6...990f9 value: 0 wei data: 0x371...303c0 logs: 0
- [vm] from: 0x5B3...eddC4 to: Counter.dec() 0xcD6...990f9 value: 0 wei data: 0xb3b...cfab82 logs: 0

● Tutorial no.

4

The screenshot shows the REMIX IDE interface. On the left, the 'Tutorials list' shows '4. Variables' as the current section. The main area displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Variables {
5     uint public blockNumber;
6
7     function doSomething() public {
8         blockNumber = block.number;
9     }
10    // Laksh Sodhai D20A 64
11 }
12
13

```

The right side of the interface shows the 'Explain contract' panel with three transaction logs:

- [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
- [vm] from: 0x5B3...eddC4 to: Counter.inc() 0xcD6...990f9 value: 0 wei data: 0x371...303c0 logs: 0
- [vm] from: 0x5B3...eddC4 to: Counter.dec() 0xcD6...990f9 value: 0 wei data: 0xb3b...cfa82 logs: 0

At the bottom, a message says 'Well done! No errors.'

● Tutorial no. 5

The screenshot shows the REMIX IDE interface. On the left, the 'Tutorials list' shows '5.1 Functions - Reading and Writing to a State Variable' as the current section. The main area displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract SimpleStorage {
5     bool public b = true;
6
7     function get_b() public view returns (bool) {
8         return b;
9     }
10    //Laksh Sodhai D20A 64
11 }
12
13

```

The right side of the interface shows the 'Explain contract' panel with three transaction logs:

- [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
- [vm] from: 0x5B3...eddC4 to: Counter.inc() 0xcD6...990f9 value: 0 wei data: 0x371...303c0 logs: 0
- [vm] from: 0x5B3...eddC4 to: Counter.dec() 0xcD6...990f9 value: 0 wei data: 0xb3b...cfa82 logs: 0

At the bottom, a message says 'Well done! No errors.'

● Tutorial no.

6

```

function addToX2(uint y) public {
    x += y;
}

function addToView(uint y) public view returns (uint) {
    infinite gas
    return x + y;
}

function addPure(uint i, uint j) public pure returns (uint) {
    infinite gas
    return i + j;
}
//Laksh Sodhai D20A 64

```

**Assignment**

Create a function called `addToX2` that takes the parameter `y` and updates the state variable `x` with the sum of the parameter and the state variable `x`.

**Check Answer**   **Show answer**

Well done! No errors.

● Tutorial no. 7

```

locked = false;

function decrement(uint i) public noReentrancy {
    x -= i;
    if (i > 1) {
        decrement(i - 1);
    }
}
//Laksh Sodhai D20A 64

```

**Assignment**

- Create a new function, `increaseX`, in the contract. The function should take an input parameter of type `uint` and increase the value of the variable `x` by the value of the input parameter.
- Make sure that `x` can only be increased.
- The body of the function `increaseX` should be empty.

Tip: Use modifiers.

**Check Answer**   **Show answer**

Well done! No errors.

- Tutorial no.

8

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

contract Function {
    function returnTwo() public pure returns (int a, bool b) {
        a = -2;
        b = true;
    }
    //Laksh Sodhai D20A 64
}
```

The screenshot shows the REMIX interface with the following details:

- Left Panel (Tutorials):** Shows "5.4 Functions - Inputs and Outputs" and "Input and Output restrictions". It includes a note about arrays being used as parameters or return values.
- Middle Panel (Code Editor):** Displays the Solidity code for the `Function` contract.
- Right Panel (Logs):** Shows three transaction logs for calls to the `returnTwo()` function on the `Counter` contract.
- Bottom Panel (Feedback):** Says "Well done! No errors."

- Tutorial no. 9

```
// 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);
}
//Laksh Sodhai D20A 64
```

The screenshot shows the REMIX interface with the following details:

- Left Panel (Tutorials):** Shows "6. Visibility" and lists points about calling from other contracts and state variables.
- Middle Panel (Code Editor):** Displays the Solidity code for the contract, which includes internal and public functions.
- Right Panel (Logs):** Shows three transaction logs for calls to the `testInternalFunc()` and `testInternalVar()` functions on the `Counter` contract.
- Bottom Panel (Feedback):** Says "Well done! No errors."

● Tutorial no.

10

```

17     // return 1;
18     // }
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 }
//Laksh Sodhai D20A 64
28

```

**else if**  
With the `else if` statement we can combine several conditions.  
If the first condition (line 6) of the `foo` function is not met, but the condition of the `else if` statement (line 8) becomes true, the function returns 1.  
Watch a video tutorial on the If/Else statement.

**Assignment**  
Create a new function called `evenCheck` in the `ifElse` contract:

- That takes in a `uint` as an argument.
- The function returns `true` if the argument is even, and `false` if the argument is odd.
- Use a ternary operator to return the result of the `evenCheck` function.

Tip: The modulo (%) operator produces the remainder of an integer division.

**Check Answer**   **Show answer**

Next

Well done! No errors.

● Tutorial no. 11

```

15         break;
16     }
17     count++;
18 }
19
20 // while loop
21 uint j;
22 while (j < 10) {
23     j++;
24 }
25
26 //Laksh Sodhai D20A 64
27

```

**continue**  
The `continue` statement is used to skip the remaining code block and start the next iteration of the loop. In this contract, the `continue` statement (line 10) will prevent the second if statement (line 12) from being executed.

**break**  
The `break` statement is used to exit a loop. In this contract, the `break` statement (line 14) will cause the for loop to be terminated after the sixth iteration.  
Watch a video tutorial on Loop statements.

**Assignment**  
1. Create a public `uint` state variable called `count` in the `Loop` contract.  
2. At the end of the for loop, increment the `count` variable by 1.  
3. Try to get the `count` variable to be equal to 9, but make sure you don't edit the `break` statement.

**Check Answer**   **Show answer**

Next

Well done! No errors.

● Tutorial no.

12

**8.1 Data Structures - Arrays**

We can use the `delete` operator to remove an element with a specific index from an array (line 42). When we remove an element with the `delete` operator all other elements stay the same, which means that the length of the array will stay the same. This will create a gap in our array. If the order of the array is not 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 `getArr3()` function to return the value of `arr3`.

**Check Answer**    **Show answer**

Next

Well done! No errors.

● Tutorial no. 13

**8.2 Data Structures - Mappings**

We set a new value for a key by providing the mapping's name and key in brackets and assigning it a new value (line 16).

**Removing values**

We can use the `delete` operator to delete a value associated with a key, which will set it to the default value of 0. As we have seen in the arrays section.

Watch a video tutorial on Mappings.

**Assignment**

1. Create a public mapping `balances` that associates the key type `address` with the value type `uint`.
2. Change the functions `get` and `remove` to work with the mapping `balances`.
3. Change the function `set` to create a new entry to the `balances` mapping, where the key is the address of the parameter and the value is the balance associated with the address of the parameter.

**Check Answer**    **Show answer**

Next

Well done! No errors.

● Tutorial no.

14

**8.3 Data Structures - Structs**

Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 19).

Initialize and update a struct: We initialize an empty struct first and then update its member by assigning it a new value (line 23).

**Accessing structs**

To access a member of a struct we can use the dot operator (line 33).

**Updating structs**

To update a structs' member we also use the dot operator and assign it a new value (lines 39 and 45).

Watch a video tutorial on Structs.

**Assignment**

Create a function `remove` that takes a `uint` as a parameter and deletes a struct member with the given index in the `todos` mapping.

**Check Answer**   **Show answer**   **Next**

Well done! No errors.

● Tutorial no. 15

**8.4 Data Structures - Enums**

We can update the enum value of a variable by assigning it the `uint` representing the enum member (line 30). Shipped would be `1` in this example. Another way to update the value is using the dot operator by providing the name of the enum and its member (line 35).

**Removing an enum value**

We can use the delete operator to delete the enum value of the variable, which means as for arrays and mappings, to set the default value to `0`.

Watch a video tutorial on Enums.

**Assignment**

1. Define an enum type called `Size` with the members `S`, `M`, and `L`.
2. Initialize the variable `sizes` of the enum type `Size`.
3. Create a getter function `getSize()` that returns the value of the variable `sizes`.

**Check Answer**   **Show answer**   **Next**

Well done! No errors.

● Tutorial no.

16

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' tutorial list is visible, with '9. Data Locations' selected. The main area displays the Solidity code for the 'Data Locations' tutorial. The code includes two functions: `g` and `h`. The `g` function returns a memory array and the `h` function takes a calldata array. Both functions interact with a memory variable `arr[0]`. The right side of the interface shows the transaction history and the AI copilot feature.

```

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 //Laksh Sodhai D20A 64
47 }
48

```

● Tutorial no. 17

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' tutorial list is visible, with '10.1 Transactions - Ether and Wei' selected. The main area displays the Solidity code for the 'Transactions - Ether and Wei' tutorial. The code defines constants for Wei, Gwei, Ether, and OneWei. The right side of the interface shows the transaction history and the AI copilot feature.

```

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 //Laksh Sodhai D20A 64

```

● Tutorial no.

18

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial list, with '10.2 Transactions - Gas and Gas Price' selected. The main content area shows a code editor with the following Solidity code:

```

7 // Using up all of the gas that you send causes your transaction to fail.
8 // State changes are undone.
9 // Gas spent are not refunded.
10 function forever() public { infinite gas
11     // Here we run a loop until all of the gas are spent
12     // and the transaction fails
13     while (true) {
14         i += 1;
15     }
16 }
17 //Laksh Sodhai D20A 64
18
19

```

Below the code editor, there is an 'Explain contract' section with three transaction logs:

- [call] from: 0x5830a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xf6d4...ce63c Debug
- [vm] from: 0x583...eddC4 to: Counter.inc() 0xcd6...990f9 value: 0 wei data: 0x371...303c0 logs: 0 Debug
- [vm] from: 0x583...eddC4 to: Counter.dec() 0xcd6...990f9 value: 0 wei data: 0xb3b...cf082 logs: 0 Debug

At the bottom of the interface, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A message 'Well done! No errors.' is displayed.

● Tutorial no. 19

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial list, with '10.3 Transactions - Sending Ether' selected. The main content area shows a code editor with the following Solidity code:

```

57
58     owner = msg.sender;
59
60     function donate() public payable { 141 gas
61
62     function withdraw() public { infinite gas
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 //Laksh Sodhai D20A 64
69

```

Below the code editor, there is an 'Explain contract' section with three transaction logs:

- [call] from: 0x5830a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xf6d4...ce63c Debug
- [vm] from: 0x583...eddC4 to: Counter.inc() 0xcd6...990f9 value: 0 wei data: 0x371...303c0 logs: 0 Debug
- [vm] from: 0x583...eddC4 to: Counter.dec() 0xcd6...990f9 value: 0 wei data: 0xb3b...cf082 logs: 0 Debug

At the bottom of the interface, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A message 'Well done! No errors.' is displayed.

**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.