

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.

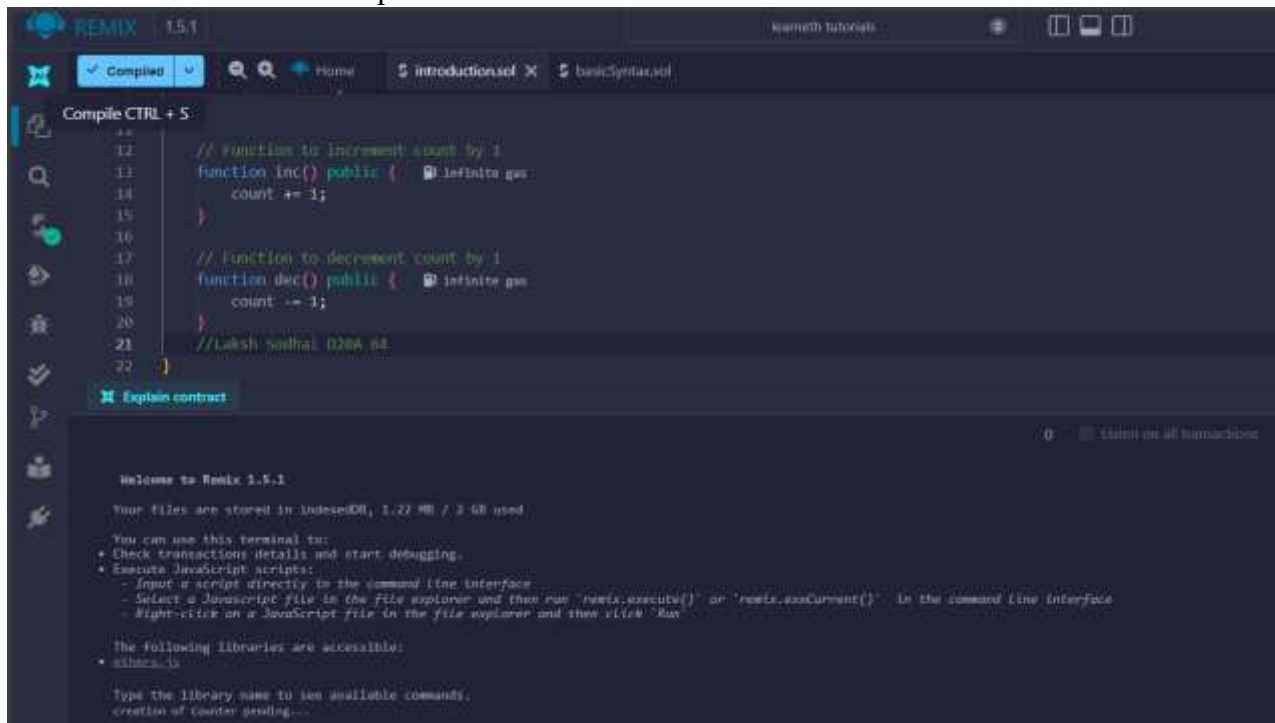
Understanding data locations is essential, as they directly impact gas costs and performance.

7. Transactions: Ether and Wei, Gas and Gas Price, Sending Transactions

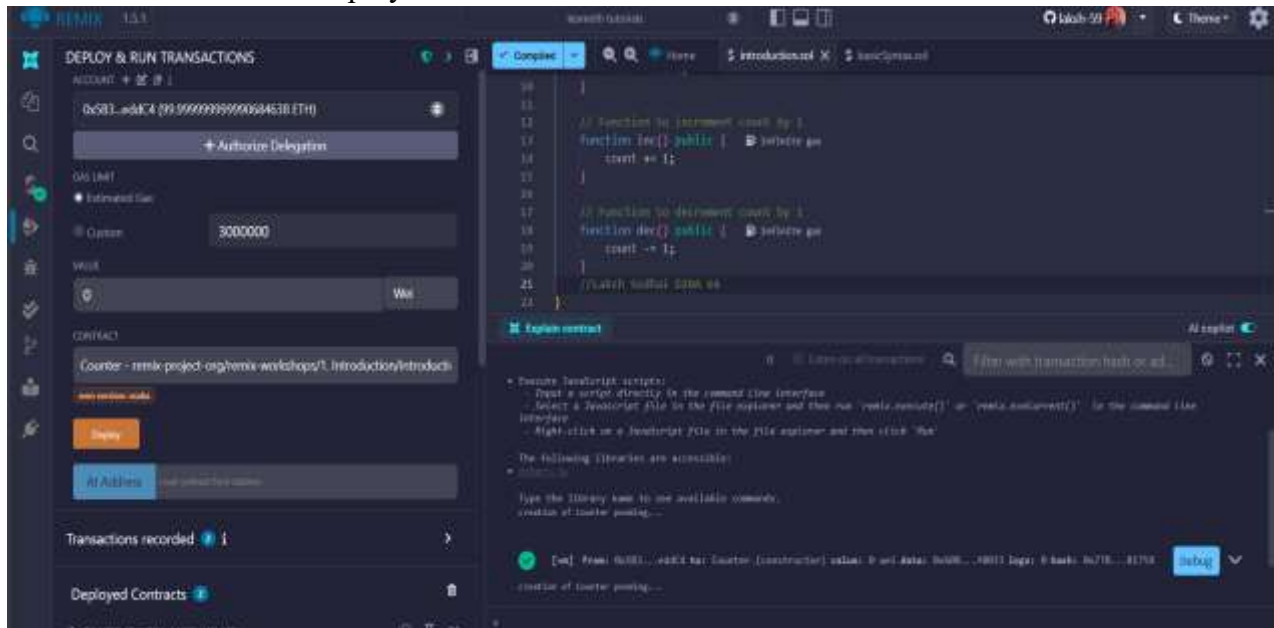
- **Ether and Wei:** Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.
- **Gas and Gas Price:** Every transaction consumes gas, which represents computational effort. The gas price 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:

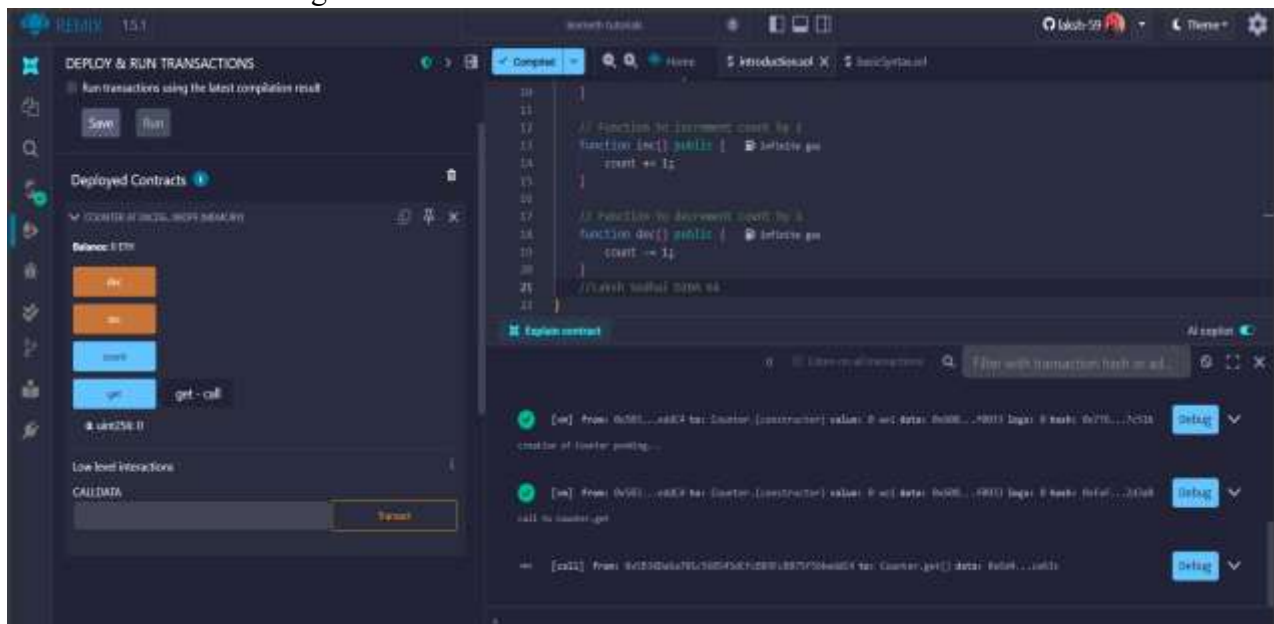
- Tutorial no. 1 – Compile the code



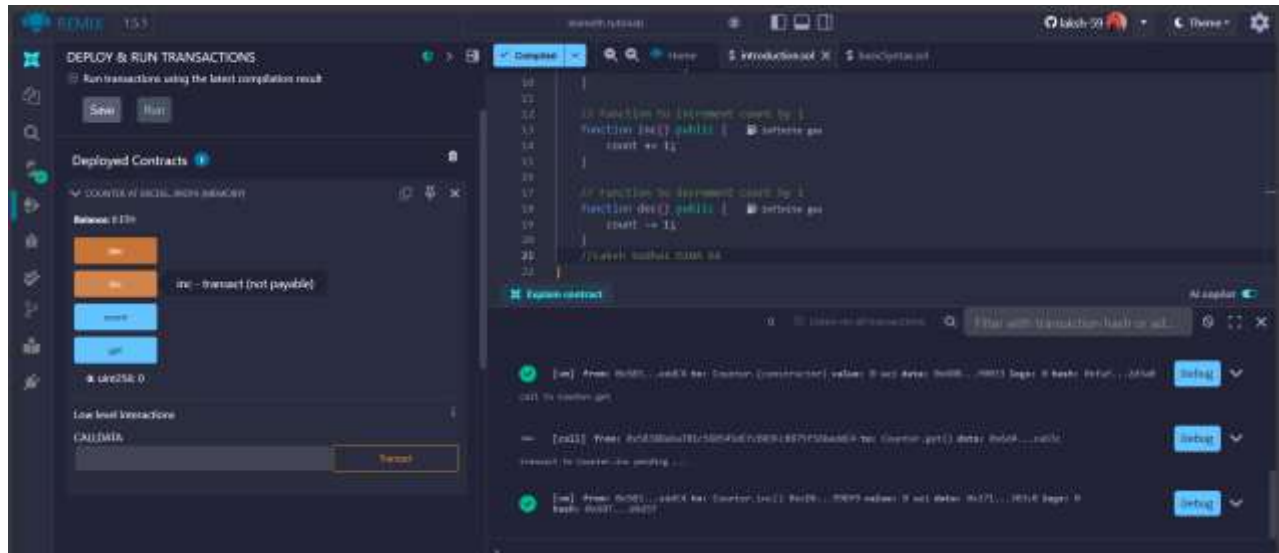
- Tutorial no. 1 – Deploy the contract



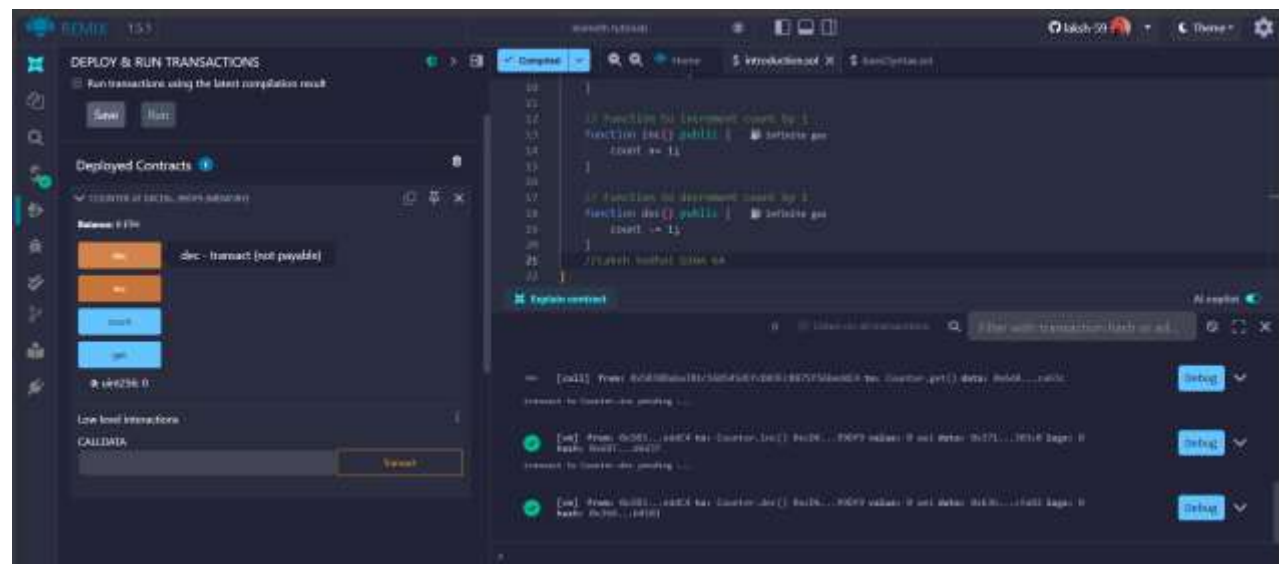
- Tutorial no. 1 – get



Tutorial no. 1 – Increment

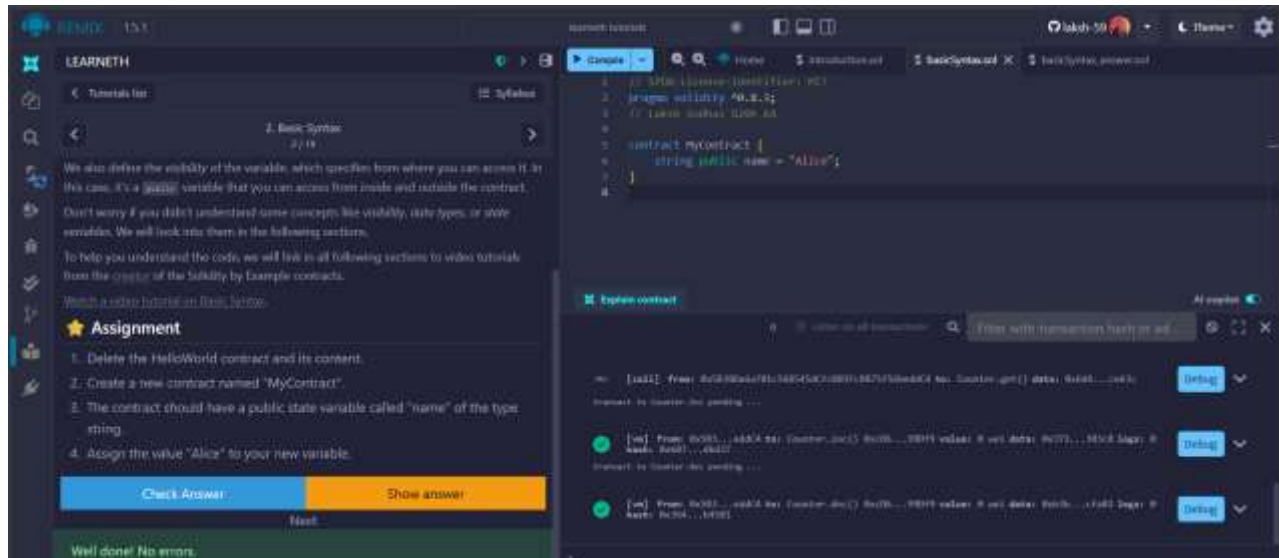


- Tutorial no. 1 – Decrement

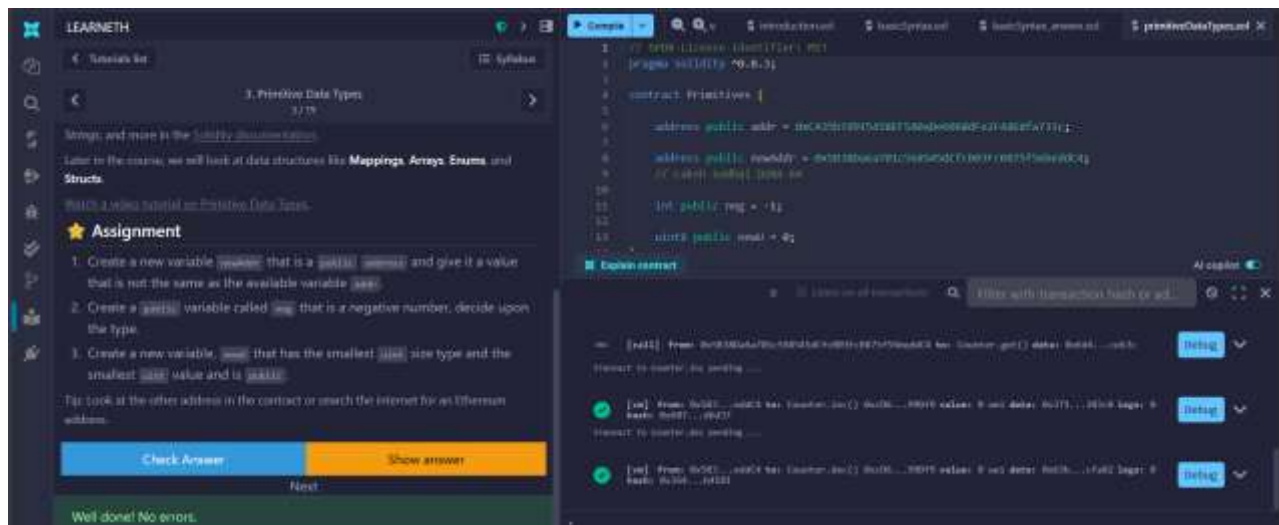


- Tutorial no.

2



- Tutorial no. 3



- Tutorial no.

4

LEARNETH 1.3.1

4. Variables

used to retrieve information about the blockchain, particular addresses, contracts, and transactions.

In this example, we use `block.timestamp` (line 14) to get a Unix timestamp of when the current block was generated and `msg.sender` (line 15) to get the caller of the contract function's address.

A list of all Global Variables is available in the [Solidity documentation](#).

Watch video tutorials on [State Variables](#), [Local Variables](#), and [Global Variables](#).

Assignment

1. Create a new public state variable called `blockNumber`.
2. Inside the function `doSomething()`, assign the value of the current block's number to the state variable `blockNumber`.

Tip: Look into the global variables section of the Solidity documentation to find out how to read the current block number.

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

Well done! No errors.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract Variables {
5
6     uint public blockNumber;
7
8     function doSomething() public {
9         blockNumber = block.number;
10    }
11 }
  
```

Explain contract

| Variable | Type | Value |
|-------------|------|-------|
| blockNumber | uint | 0 |

Functions

| Function | Parameters | Return Type |
|---------------|------------|-------------|
| doSomething() | None | None |

- Tutorial no. 5

LEARNETH 1.3.1

5.1 Functions - Reading and Writing to a State Variable

In our function calls, inputs are our `msg` functions (line 1), you must specify the parameter types and names. A common convention is to use an underscore as a prefix for the parameter name to distinguish them from state variables.

You can then set the visibility of a function and declare them `public` or `private` as we do for the `get()` function if they don't modify the state. Our `get()` function also returns values, so we have to specify the return type. In this case, it's a `uint` since the state variable `x` that the function returns is a `uint`.

We will explore the particularities of Solidity functions in more detail in the following sections.

Watch video tutorials on [Functions](#).

Assignment

1. Create a public state variable called `x` that is of type `uint` and initialize it to `0`.
2. Create a public function called `get()` that returns the value of `x`.

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

Well done! No errors.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract SimpleStorage {
5
6     uint public x = 0;
7
8     function get() public view returns (uint) {
9         return x;
10    }
11 }
  
```

Explain contract

| Variable | Type | Value |
|----------|------|-------|
| x | uint | 0 |

Functions

| Function | Parameters | Return Type |
|----------|------------|-------------|
| get() | None | uint |

- Tutorial no.

6

The screenshot shows the Remix IDE interface. On the left, the 'LEARNETH' sidebar is open, displaying a tutorial for '5.2 Functions - View and Pure'. The main editor area shows a Solidity contract with the following code:

```

function addWithGas(uint x) public returns (uint) {
    x += 1;
}

function addWithView(uint x) public view returns (uint) {
    return x + 1;
}

function addPure(uint x, uint y) public pure returns (uint) {
    return x + y;
}
// SPDX-License-Identifier: MIT

```

The bottom panel shows the 'Compiler' tab with a 'Compile' button and a 'Run on contract' button. The 'Run on contract' button is highlighted, and a transaction hash is visible.

- Tutorial no. 7

The screenshot shows the Remix IDE interface. On the left, the 'LEARNETH' sidebar is open, displaying a tutorial for '5.3 Functions - Modifiers and Constructors'. The main editor area shows a Solidity contract with the following code:

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
    uint public count;

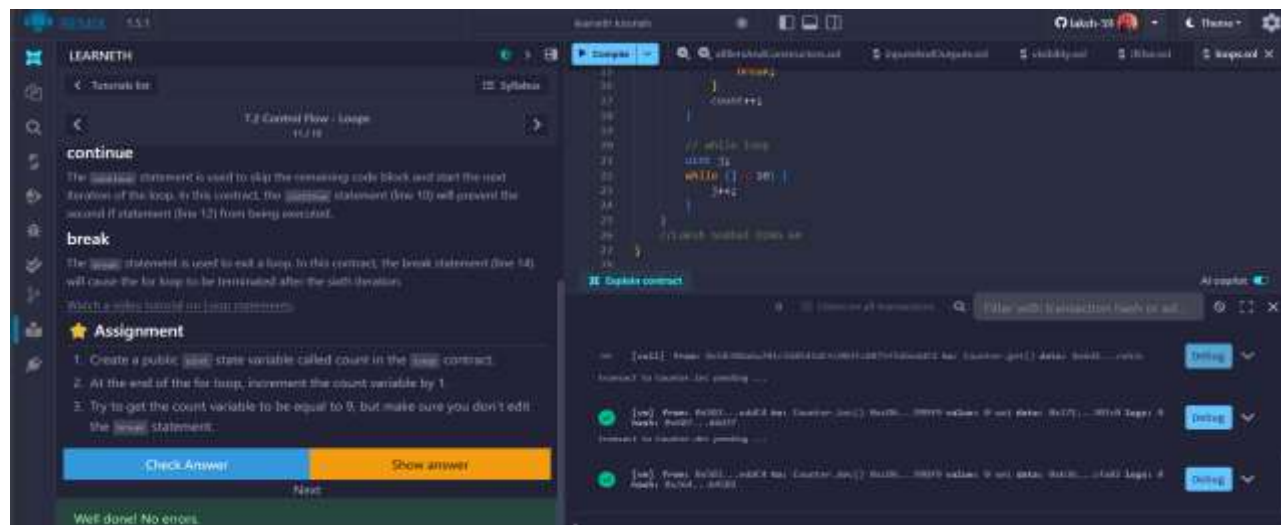
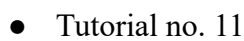
    function increment() public {
        count++;
    }

    function decrement() public {
        count--;
    }
}

```

The bottom panel shows the 'Compiler' tab with a 'Compile' button and a 'Run on contract' button. The 'Run on contract' button is highlighted, and a transaction hash is visible.

The screenshot shows the Learneth interface. On the left, a sidebar contains navigation icons. The main content area displays a lesson titled "LEARNETH" with a sub-header "1.4 Functions - Inputs and Outputs". The text explains that mapping cannot be used as a parameter or return parameter for contract functions that are publicly visible. It also mentions that arrays can be used as parameters or return parameters. Below the text is an "Assignment" section with a button to "Check Answer" and a "Next" button. On the right, a code editor shows a Go program with a function `contractFunction` that takes a `uint256` parameter and returns a `bool`. The program also includes a `main` function that calls `contractFunction` with the value `42`.



- Tutorial no.

12

LEARNETH 13.1

8.1 Data Structures - Arrays

We remove the `splice()` function to remove an element from a specific index, maintaining the order. When we remove an element with the `splice()` operation, 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 25).

Watch a video tutorial on Arrays.

Assignment

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

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

Next

Well done! No errors.

Debug Console

```

10 array.splice(2,1);
11 array.splice(3,1);
12 array.splice(4,1);
13 // [0, 1, 2, 3, 4]
14
15 function splice() {
16   // [0, 1, 2, 3, 4]
17
18   return array.splice(2,1);
19 }
20 // [0, 1, 2, 3, 4]
21
22 // Laksh Sodhai D20A 64
23
24 }
  
```

Debug Console

Filter with transaction hash or address

10 [0,1] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0,1]

11 [0] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0]

12 [0] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0]

13 [0] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0]

- Tutorial no. 13

LEARNETH 13.1

8.2 Data Structures - Mappings

We set a new value for a key by providing the mapping's name and key in `balances` 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 `add` and `remove` to work with the mapping `balances`.
3. Change the function `add` 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.

Debug Console

```

36 address _addr1,
37 uint _val,
38 bool _bool
39 } public {
40   balances[_addr1][_val] = _bool;
41 }
42
43 function remove(address _addr1, uint _val) public {
44   delete balances[_addr1][_val];
45 }
46 // Laksh Sodhai D20A 64
47 }
  
```

Debug Console

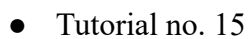
Filter with transaction hash or address

10 [0,1] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0,1]

11 [0] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0]

12 [0] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0]

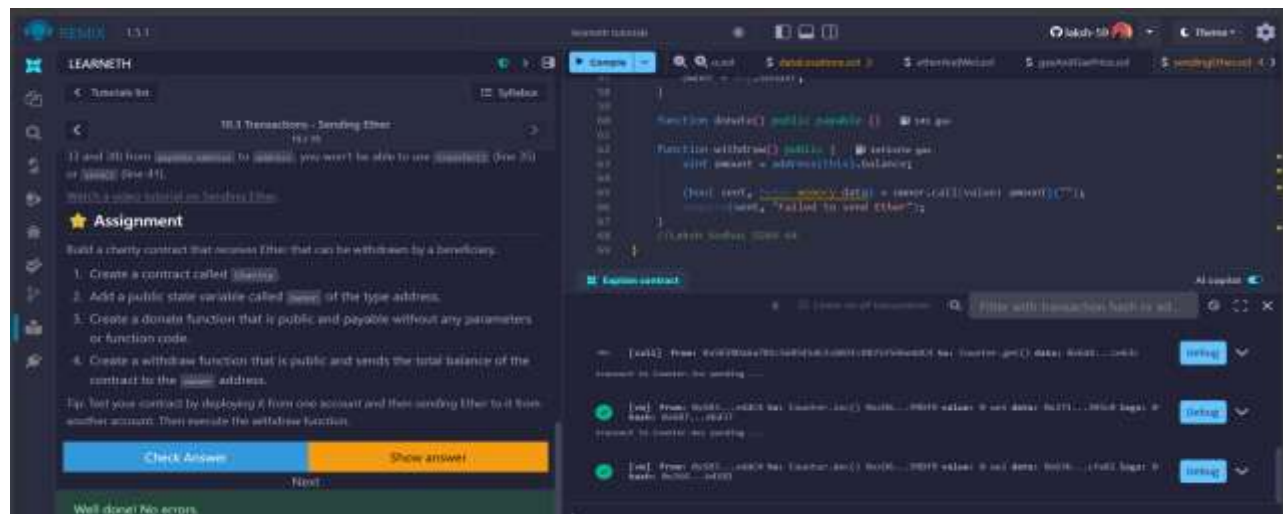
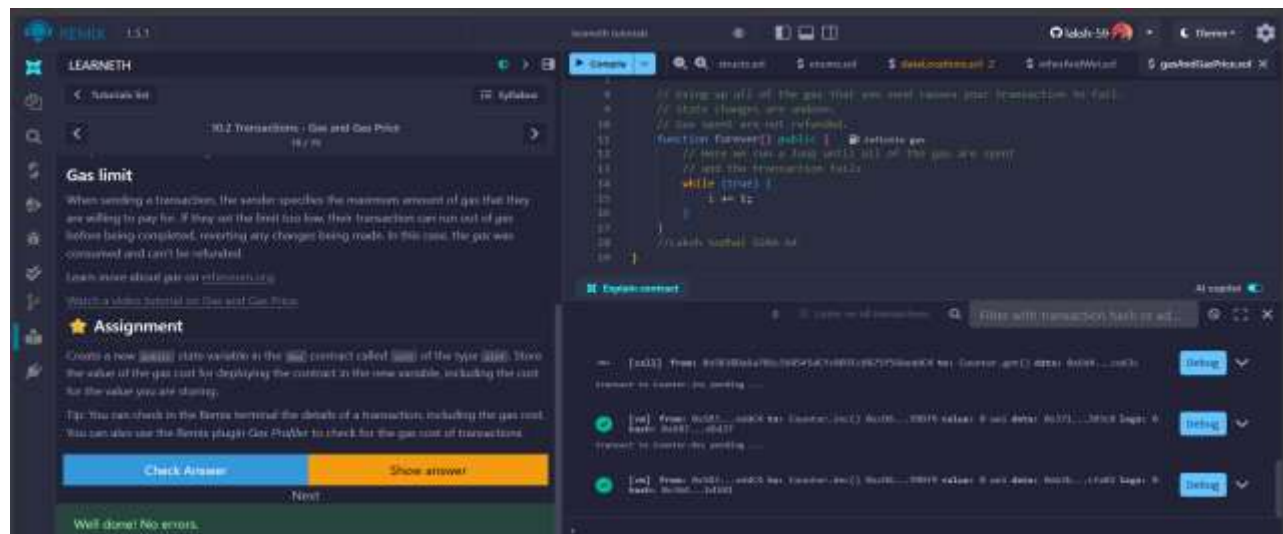
13 [0] From: 0x038b6a78c79d948c7d801d87c75b6a0c3a: Counter-act() detail: debug... [0]



[illegible]

- Tutorial no. 17

The screenshot shows the Remix IDE interface. On the left, the 'LearnETH' sidebar is open, displaying the '10.1 Transactions - Ether and Wei' section. The main editor area shows a Solidity contract snippet. The right pane shows the 'Explain contract' tool, which provides a detailed explanation of the contract's logic, including the 'Coin' struct definition and the 'Coin' function's behavior.



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.