

EXPERIMENT NO.4

Name:Aryan Anil Patankar

Class:D20A

Roll No:38

Batch:B

Aim:Hands on Solidity Programming Assignments for creating Smart Contracts

Theory:-

1. Primitive Data Types, Variables, Functions - pure, view

Primitive (Value) Data Types in Solidity include:

- bool → true/false
- int / uint (signed/unsigned integers, e.g., uint256 is most common)
- address → 20-byte Ethereum address
- bytes1 to bytes32 → fixed-size byte arrays
- string → dynamic UTF-8 encoded text (reference type, but often grouped here)

Variables are declared with a type and can be:

- **State variables** → stored permanently on the blockchain (expensive)
- **Local variables** → exist only during function execution

Functions can be marked as:

- pure → does not read or write state (computes only from inputs; cheapest gas)
- view → reads state but does not modify it (e.g., getters; no gas when called externally)

Example

```
function getResult() public view returns (uint product, uint sum) {  
    product = num1 * num2; // reads state  
    sum = num1 + num2;  
}
```

```
function pureCalc(uint a, uint b) public pure returns (uint) {  
    return a + b; // no state access  
}
```

2. Inputs and Outputs to Functions

Functions in Solidity can take **parameters** (inputs) and return **values** (outputs).

- **Inputs:** Declared in parentheses; can use data locations like memory or calldata for reference types.
- **Outputs:** Declared after returns keyword; can return multiple values.

Example:

```
function add(uint a, uint b) public pure returns (uint sum) {  
    sum = a + b;  
}  
  
// Multiple outputs  
  
function getValues() public view returns (uint, string memory) {  
    return (age, name);  
}
```

- Use calldata for external calls (cheaper, read-only).
- Use memory for temporary copies inside functions.

3. Visibility, Modifiers and Constructors

Visibility Specifiers (for functions and state variables):

- **public** → anyone can call/read (default for state vars creates getter)
- **private** → only inside current contract
- **internal** → current + derived (child) contracts
- **external** → only external calls (cheaper for large data)

Modifiers → reusable code blocks that run before/after function body (e.g., access control).

Use `_;` to insert function body.

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}
```

```
function restricted() public onlyOwner { ... }
```

Constructors → special function that runs once on deployment (initializes state).

- Syntax: `constructor() { ... }` (older: same name as contract)

4. Control Flow: if-else, loops

Solidity supports standard control structures:

if-else:

```
if (condition) {  
    // true  
} else if (another) {  
    // else-if  
} else {  
    // false  
}
```

Loops:

- `for (uint i = 0; i < 10; i++) { ... }`
- `while (condition) { ... }`
- `do { ... } while (condition);`

Avoid unbounded loops (gas limit risk). Use `break` / `continue` when needed.

5. Data Structures: Arrays, Mappings, structs, enums

- **Arrays**
 - Fixed: `uint[5] arr;`
 - Dynamic: `uint[] arr;` (use `.push()`, `.pop()`, `.length`)
- **Mappings** → key-value store (like hash table)
`mapping(address => uint) public balances;`

Keys can be most types; no iteration possible.

Structs → custom composite types

```
struct User {  
    address wallet;  
    uint balance;  
    bool active;  
}  
User public owner;
```

Enums → named constants (integers under the hood)

```
enum Status { Pending, Active, Cancelled }  
Status public state = Status.Pending;
```

6. Data Locations

Solidity has three main **data locations** for reference types (arrays, structs, mappings, strings):

- **storage** → permanent blockchain storage (persistent, expensive)
Default for state variables.
- **memory** → temporary, function lifetime only (deleted after execution)
Cheap; used for local variables & function args/returns.
- **calldata** → read-only, non-modifiable area for function call data
Cheapest for external function parameters (immutable copy of tx data).

Rule:

- State vars → always storage
- Function args → prefer calldata (external) or memory
- Local reference vars → must specify location

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

- **Ether units** → smallest is **wei** ($1 \text{ ETH} = 10^{18} \text{ wei}$) Other: gwei (10^9 wei), commonly used for gas prices.
- **Gas** → computational effort unit
 - **Gas Limit** → max gas willing to spend (set by sender)
 - **Gas Price** → price per gas unit (in wei/gwei; set by sender)
 - Total fee = gas used \times gas price
- **msg.value** → amount of wei sent with transaction
- **Sending Ether** → use `.transfer()`, `.send()`, or `.call{value: amount}("")` (recommended low-level call)

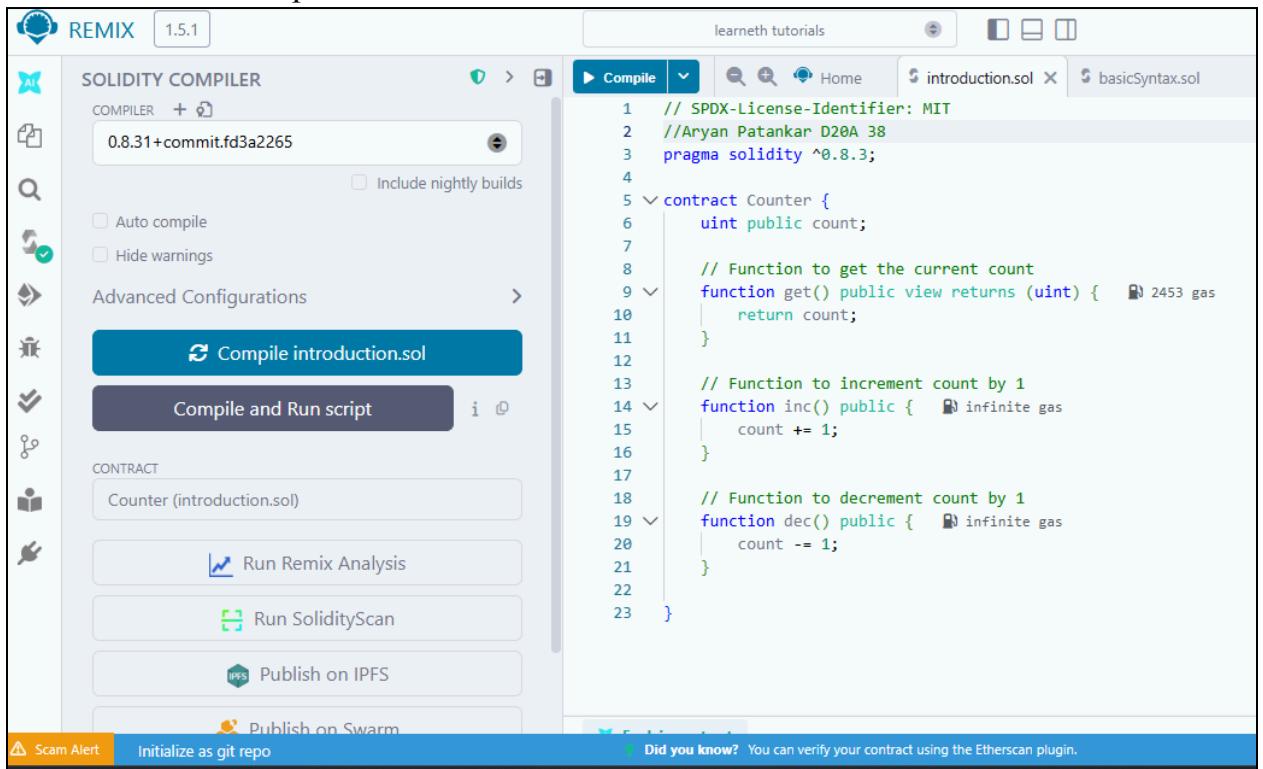
Example:

```
function sendEther(address payable recipient) public payable {  
    recipient.transfer(msg.value);  
}
```

- Use `payable` for addresses/functions that receive Ether.
- `tx.gasprice` → current gas price (global var)

Implementation:-

- Tutorial no. 1 – Compile the code



```
// SPDX-License-Identifier: MIT
//Aryan Patankar D20A 38
pragma solidity ^0.8.3;

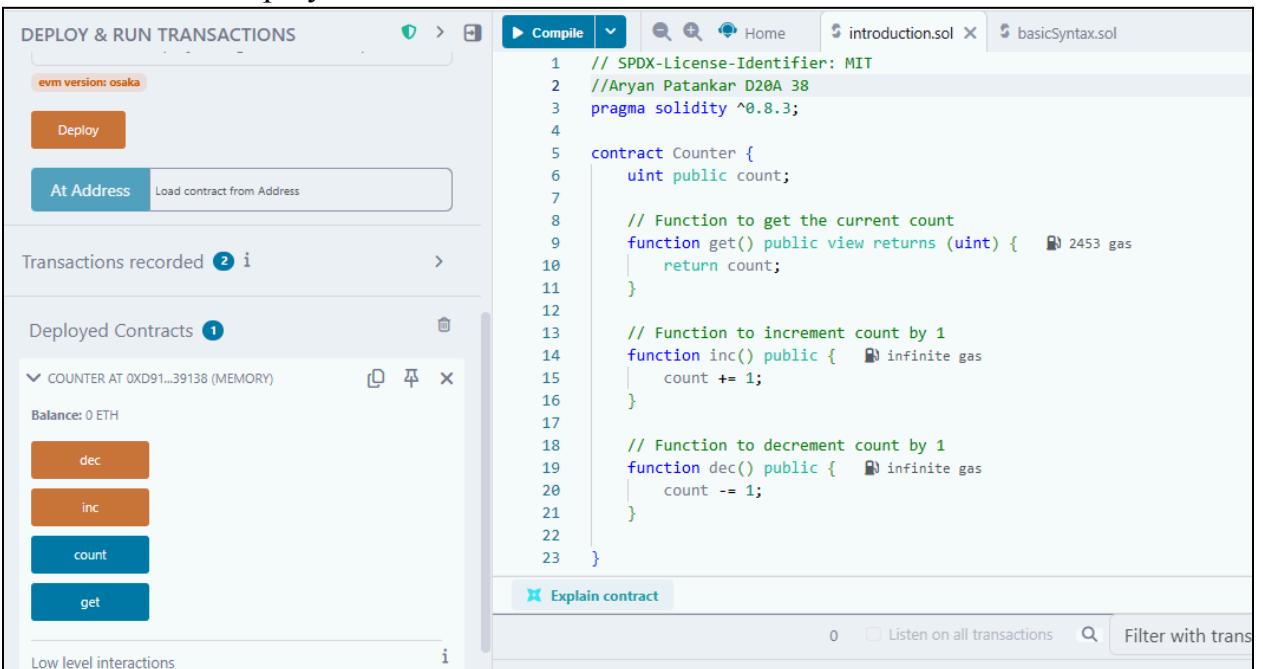
contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) { 2453 gas
        return count;
    }

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

    // Function to decrement count by 1
    function dec() public { infinite gas
        count -= 1;
    }
}
```

- Tutorial No.1-Deploy the contract



```
// SPDX-License-Identifier: MIT
//Aryan Patankar D20A 38
pragma solidity ^0.8.3;

contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) { 2453 gas
        return count;
    }

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

    // Function to decrement count by 1
    function dec() public { infinite gas
        count -= 1;
    }
}
```

- Tutorial no. 1 - get

```
CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 ⓘ

to Counter.get() 0xd9145CCE52D386f254917e481eB44e9943F39138 ⓘ

execution cost 2453 gas (Cost only applies when called by a contract) ⓘ

input 0x6d4...ce63c ⓘ

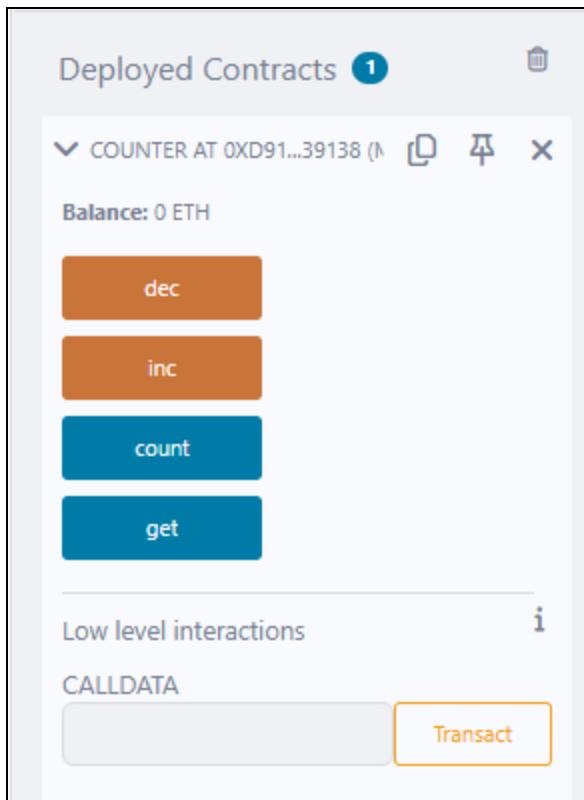
output 0x000000000000000000000000000000000000000000000000000000000000000 ⓘ

decoded input {} ⓘ

decoded output {
    "0": "uint256: 0"
} ⓘ

logs [] ⓘ

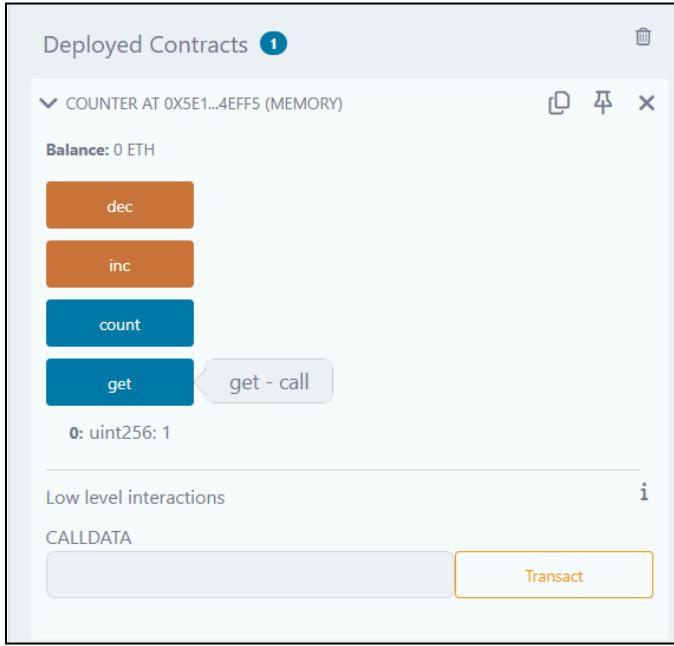
raw logs [] ⓘ
```



- Tutorial no. 1 – Increment



- Tutorial no. 1 – Decrement



- Tutorial no.2

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
//Aryan Patankar D20A 38
pragma solidity ^0.8.3;

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

● Tutorial no.3

LEARNETH

3. Primitive Data Types

Later in the course, we will look at data structures like [Mappings](#), [Arrays](#), [Enums](#), and [Structs](#).

Watch a video tutorial on [Primitive Data Types](#).

Assignment

- 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`.
- Create a `public` variable called `neg` that is a negative number, decide upon the type.
- Create a new variable, `newU` that has the smallest `uint` size type and the smallest `uint` value and is `public`.

Tip: Look at the other address in the contract or search the internet for an Ethereum address.

Check Answer **Show answer**

Next

Well done! No errors.

```

23 int8 public i8 = -1;
24 int public i256 = 456;
25 int public i = -123; // int is same as int256
26
27 address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa73c;
28
29 // Default values
30 // Unassigned variables have a default value
31 bool public defaultBool; // false
32 uint public defaultUint; // 0
33 int public defaultInt; // 0
34 address public defaultAddr; // 0x000000000000000000000000000000000000000000000000000000000000000
35
36 //Aryan Patankar D20A 38
37 address public newAddr=0x000000000000000000000000000000000000000000000000000000000000000;
38 int public neg=-15;
39 uint8 public newU=0;
40 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

[vm] from: 0x5B3...eddC4 to: MyContract.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xF5a...52c7f Debug

● Tutorial no.4

LEARNETH

4. Variables

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

Assignment

- Create a new public state variable called `blockNumber`.
- Inside the function `doSomething()`, assign the value of the current block 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**

Next

Well done! No errors.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 // Aryan Patankar D20A 38
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 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

[vm] from: 0x5B3...eddC4 to: MyContract.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xF5a...52c7f

● Tutorial no.5

LEARNETH

Tutorials list Syllabus

5.1 Functions - Reading and Writing to a State

Variable 5 / 19

function if they don't modify the state. Our `get` function also returns values, so we have to specify the return types. In this case, it's a `uint` since the state variable `num` that the function returns is a `uint`.

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

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

Assignment

- Create a public state variable called `b` that is of type `bool` and initialize it to `true`.
- Create a public function called `get_b` that returns the value of `b`.

Check Answer **Show answer**

Next

Well done! No errors.

Compiled

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7     bool public b=true;
8     // You need to send a transaction to write to a state variable.
9     function set(uint _num) public { 22536 gas
10        num = _num;
11    }
12
13    // You can read from a state variable without sending a transaction.
14    function get() public view returns (uint) { 2475 gas
15        return num;
16    }
17    function get_b() public view returns(bool){ 2539 gas
18        return b;
19    }
20 }
```

Explain contract

0 Listen on all transactions

[vm] from: 0x5B3...eddC4 to: MyContract.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 has

● Tutorial no.6

Tutorials list Syllabus

5.2 Functions - View and Pure

6 / 19

the state variable `x`.

In Solidity development, you need to optimise your code for saving computation cost (gas cost). Declaring functions `view` and `pure` can save gas cost and make the code more readable and easier to maintain. `Pure` functions don't have any side effects and will always return the same result if you pass the same arguments.

[Watch a video tutorial on View and Pure Functions.](#)

Assignment

Create a function called `addToX` 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**

Next

Well done! No errors.

Compiled

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
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    function addToX2(uint y) public{ infinite gas
17        x=x+y;
18    }
19 }
```

Explain contract

0 Listen on all transactions

[vm] from: 0x5B3...eddC4 to: MyContract.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash

● Tutorial no.7

The screenshot shows the LearnETH platform interface. On the left, there's a sidebar with "Tutorials list" and "Syllabus". The main content area is titled "5.3 Functions - Modifiers and Constructors" (7 / 19). It contains text about the constructor setting the initial value of the owner variable and a link to a video tutorial on Function Modifiers. Below this is a "Assignment" section with three tasks:

- 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.

A tip says "Tip: Use modifiers." Below the assignment are "Check Answer" and "Show answer" buttons, and a "Next" button. At the bottom, it says "Well done! No errors."

On the right, the Solidity code editor shows the following code:

```

1   9
2   10
3   11
4   12
5   13
6   14
7   15
8   16
9   17
10  18
11  19
12  20
13  21
14  22
15  23
16  24
17  25
18  26
19  27
20  28
21  29
22  30
    bool public locked;
constructor() {
    // Set the transaction sender as the owner of the contract.
    owner = msg.sender;
}
// Aryan Patankar D20A 38
modifier onlyPositive(uint _value) {
    require(_value > 0, "Increase value must be greater than zero");
}
function increaseX(uint _amount) public onlyPositive(_amount) {
    x += _amount;
}

// Modifier to check that the caller is the owner of
// the contract.
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    // Underscore is a special character only used inside

```

Below the code editor is an "Explain contract" section with a search bar and filter options.

● Tutorial no.8

The screenshot shows the LearnETH platform interface. On the left, there's a sidebar with "Tutorials list" and "Syllabus". The main content area is titled "5.4 Functions - Inputs and Outputs" (8 / 19). It contains text about parameters of contract functions and arrays being used as parameters or return values. Below this is a "Assignment" section with a task:

Create a new function called `returnTwo` that returns the values `-2` and `true` without using a return statement.

Below the assignment are "Check Answer" and "Show answer" buttons, and a "Next" button. At the bottom, it says "Well done! No errors."

On the right, the Solidity code editor shows the following code:

```

1   71
2   72
3   73
4   74
5   75
6   76
7   77
8   78
9   79
10  80
11  81
12  82
13  83
14  84
15  85
16  86
17  87
18  88
function arrayInput(uint[] memory _arr) public {} infinite gas
// Can use array for output
uint[] public arr;
function arrayOutput() public view returns (uint[] memory) {} infinite gas
    return arr;
//Aryan Patankar D20A 38
function returnTwo() public pure returns(
    int i,
    bool b
)
{
    i=-2;
    b=true;
}

```

Below the code editor is an "Explain contract" section with a search bar and filter options.

● Tutorial no.9

6. Visibility

and state variables from the `Base` contract.

When you uncomment the `testPrivateFunc` (lines 58-60) you get an error because the child contract doesn't have access to the private function `privateFunc` from the `Base` contract.

If you compile and deploy the two contracts, you will not be able to call the functions `privateFunc` and `internalFunc` directly. You will only be able to call them via `testPrivateFunc` and `testInternalFunc`.

Watch a video tutorial on Visibility.

Assignment

Create a new function in the `Child` contract called `testInternalVar` that returns the values of all state variables from the `Base` contract that are possible to return.

Check Answer **Show answer**

Well done! No errors.

Compile

```

52 // string external externalVar = "my external variable";
53 }
54
55 contract Child is Base {
56     // Inherited contracts do not have access to private functions
57     // and state variables.
58     // function testPrivateFunc() public pure returns (string memory) {
59     //     return privateFunc();
60     // }
61
62     // Internal function call be called inside child contracts.
63     //Aryan Patankar D20A 38
64     function testInternalFunc() public pure override returns (string memory) { infinite gas
65         return internalFunc();
66     }
67
68     function testInternalVar() public view returns(string memory x,string memory y){ infinite gas
69         return(internalVar,publicVar);
70     }
71 }
```

Explain contract

● Tutorial no.10

7.1 Control Flow - If/Else

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**

Well done! No errors.

Compile

```

10 } else {
11     return 2;
12 }
13
14
15 function ternary(uint _x) public pure returns (uint) { infinite gas
16     // if (_x < 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 //Aryan Patankar D20A 38
26 function evenCheck(uint _number) public pure returns (bool) { infinite gas
27     return (_number % 2 == 0) ? true : false;
28 }
```

Explain contract

● Tutorial no.11

LEARNETH

Tutorials list Syllabus

7.2 Control Flow - Loops 11 / 19

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

- Create a public `uint` state variable called `count` in the `Loop` contract.
- At the end of the for loop, increment the `count` variable by 1.
- 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.

Contract Code:

```

15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
        break;
    }
    count++;
}

// while loop
//Aryan Patankar D20A 38
uint j;
while (j < 10) {
    j++;
    if (count < 9) {
        count++;
    }
}

```

Explain contract

Type the library name to see available commands.

● Tutorial no.12

LEARNETH

Tutorials list Syllabus

8.1 Data Structures - Arrays 12 / 19

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

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

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

Next

Well done! No errors.

Contract Code:

```

4 contract Array {
5     // Several ways to initialize an array
6     //Aryan Patankar D20A 38
7     uint[] public arr;
8     uint[] public arr2 = [1, 2, 3];
9     uint[3] public arr3 = [0, 1, 2];
10
11
12
13
14     function get(uint i) public view returns (uint) {    infinite gas
15         return arr[i];
16     }
17
18
19     // Solidity can return the entire array.
20     // But this function should be avoided for
21     // arrays that can grow indefinitely in length.
22     function getArr() public view returns (uint[3] memory) {    infinite gas
23         return arr3;
24     }
25

```

Explain contract

Type the library name to see available commands.

● Tutorial no.13

The screenshot shows a Solidity code editor interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main area displays a tutorial titled '8.2 Data Structures - Mappings' (page 13 of 19). The text explains that Mappings associate keys with values, with a key setting the value to 0 by default. It includes a link to a video tutorial on Mappings.

Assignment:

- Create a public mapping `balances` that associates the key type `address` with the value type `uint`.
- Change the functions `get` and `remove` to work with the mapping `balances`.
- 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.

At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a message 'Well done! No errors.'

On the right, the Solidity code editor shows the following code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
4
5 contract Mapping {
6     // Mapping from address to uint
7     mapping(address => uint) public balances;
8
9     function get(address _addr) public view returns (uint) {
10        // Mapping always returns a value.
11        // If the value was never set, it will return the default value.
12        return balances[_addr];
13    }
14
15    function set(address _addr) public {
16        // Update the value at this address
17        balances[_addr] = _addr.balance;
18    }
19
20    function remove(address _addr) public {
21        // Reset the value to the default value.
22        delete balances[_addr];
23    }
}

```

Below the code editor, there's an 'Explain contract' button and a search/filter bar.

● Tutorial no.14

The screenshot shows a Solidity code editor interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main area displays a tutorial titled '8.3 Data Structures - Structs' (page 14 of 19). The text discusses accessing and updating struct members using the dot operator.

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.

At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a message 'Well done! No errors.'

On the right, the Solidity code editor shows the following code:

```

35
36 // update text
37 function update(uint _index, string memory _text) public {
38     Todo storage todo = todos[_index];
39     todo.text = _text;
40 }
41
42 // update completed
43 function toggleCompleted(uint _index) public {
44     Todo storage todo = todos[_index];
45     todo.completed = !todo.completed;
46 }
47 //Aryan Patankar D20A 38
48 function remove(uint _index) public {
49     delete todos[_index];
50 }
51

```

Below the code editor, there's an 'Explain contract' button and a search/filter bar.

● Tutorial no.15

The screenshot shows a web-based development environment for Ethereum smart contracts. On the left, there's a sidebar with 'LEARNETH' branding, a 'Tutorials list' button, and a 'Syllabus' button. The main content area displays a tutorial titled '8.4 Data Structures - Enums' (page 15 / 19). The text discusses removing an enum value using the delete operator. It includes a link to a video tutorial on Enums. Below this is a section titled 'Assignment' with three tasks:

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

At the bottom of the assignment section are 'Check Answer' and 'Show answer' buttons. A green banner at the bottom says 'Well done! No errors.'

On the right side of the interface, there's a code editor window titled 'Compile'. It shows a Solidity code snippet by 'Aryan Patankar D20A 38' with line numbers 14 to 36. The code defines an enum `Size` with values `S`, `M`, and `L`, and a public variable `sizes`. It also contains a function `getSize()` that returns the value of `sizes`. Below the code editor is a 'Explain contract' button.

At the bottom of the right panel, there are buttons for 'Listen on all transactions', a search icon, and a 'Filter with transaction hash or address' input field.

● Tutorial no.16

This screenshot shows another web-based Ethereum development environment. The sidebar has 'LEARNETH' branding, a 'Tutorials list' button, and a 'Syllabus' button. The main content area displays a tutorial titled '9. Data Locations' (page 16 / 19). The text notes that when creating contracts, it's important to be mindful of gas costs. It includes a section titled 'Assignment' with four tasks:

- Change the value of the `myStruct` member `foo` inside the `function f` to 4.
- Create a new struct `myMemStruct2` with the data location `memory` inside the `function f` and assign it the value of `myStruct`. Change the value of the `myMemStruct2` member `foo` to 1.
- Create a new struct `myMemStruct3` with the data location `memory` inside the `function f` and assign it the value of `myStruct`. Change the value of the `myMemStruct3` member `foo` to 3.
- Let the function `f` return `myStruct`, `myMemStruct2`, and `myMemStruct3`.

A tip at the bottom says: 'Tip: Make sure to create the correct return types for the function `f`'.

At the bottom of the assignment section are 'Check Answer' and 'Show answer' buttons.

On the right side, there's a code editor window titled 'Compile' showing a Solidity code snippet by 'Aryan Patankar D20A 38'. The code defines a function `f` that performs the tasks listed in the assignment. It uses `MyStruct` and `MyStruct memory` types, and creates `myMemStruct2` and `myMemStruct3` in memory. Below the code editor is a 'Explain contract' button.

At the bottom of the right panel, there are buttons for 'Listen on all transactions', a search icon, and a 'Filter with transaction hash or address' input field.

● Tutorial no.17

The screenshot shows a web-based development environment for Ethereum smart contracts. On the left, there's a sidebar with 'LEARNETH' branding, a 'Tutorials list' button, and a 'Syllabus' button. The main content area is titled '10.1 Transactions - Ether and Wei' (17 / 19). It contains two code snippets:

- gwei:** One `gwei` (giga-wei) is equal to 1,000,000,000 (10^9) `wei`.
- ether:** One `ether` is equal to 1,000,000,000,000,000,000 (10^{18}) `wei` (line 11).

A link to 'Watch a video tutorial on Ether and Wei.' is provided.

Assignment

- Create a `public uint` called `oneGWei` and set it to 1 `gwei`.
- Create a `public bool` called `isOneGWei` and set it to the result of a comparison operation between 1 `gwei` and 10^9 .

Tip: Look at how this is written for `gwei` and `ether` in the contract.

At the bottom of the assignment section are 'Check Answer' and 'Show answer' buttons, and a 'Next' button.

Well done! No errors.

On the right side, the code editor shows the Solidity contract source code:

```
// SPDX-License-Identifier: MIT
//Aryan Patankar D20A 38
pragma solidity ^0.8.3;

contract EtherUnits {
    uint public oneWei = 1 wei;
    bool public isOneWei = 1 wei == 1;

    uint public oneEther = 1 ether;
    bool public isOneEther = 1 ether == 1e18;

    //1. Create oneGWei and set it to 1 gwei
    uint public oneGWei = 1 gwei;

    // 2. Compare 1 gwei to 10^9 (10**9)
    bool public isOneGWei = 1 gwei == 10**9;
}
```

The code editor has tabs for 'Compile', '.sol', 'arrays.sol', 'mappings.sol', 'structs.sol', and 'enum'. Below the code editor is an 'Explain contract' section with a command input field and a 'Listen on all transaction' checkbox.

● Tutorial no.18

The screenshot shows a web-based Ethereum development environment. On the left, the 'Tutorials list' indicates '10.2 Transactions - Gas and Gas Price' is selected. The main content area displays a snippet of Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
4
5 contract Gas {
6     uint public i = 0;
7
8
9     uint public cost = 170367;
10
11     function forever() public {
12         while (true) {
13             i += 1;
14         }
15     }
16 }

```

Below the code, there's an 'Explain contract' section with a green checkmark and the message: '[vm] from: 0x5B3...eddC4 to: Gas.(constructor) value: 0 wei data: 0x608...f0033 logs: 0'. The status bar shows '1 Transaction mined and execution succeed'.

At the bottom, there are 'Check Answer', 'Show answer', and 'Next' buttons, along with a green message: 'Well done! No errors.'

● Tutorial no.19

The screenshot shows a web-based Ethereum development environment. On the left, the 'Tutorials list' indicates '10.3 Transactions - Sending Ether' is selected. The main content area displays a snippet of Solidity code:

```

53 }
54 //Aryan Patankar D20A 38
55 contract Charity {
56     address public owner;
57
58     constructor() {
59         owner = msg.sender;
60     }
61
62     function donate() public payable {} 141 gas
63
64     function withdraw() public {
65         uint amount = address(this).balance;
66
67         (bool sent, bytes memory data) = owner.call{value: amount}("");
68         require(sent, "Failed to send Ether");
69     }
70 }

```

Below the code, there's an 'Explain contract' section with a green checkmark and the message: '[vm] from: 0x5B3...eddC4 to: Gas.(constructor) value: 0 wei data: 0x608...f0033 logs: 0'. The status bar shows '1 Transaction mined and execution succeed'.

At the bottom, there are 'Check Answer', 'Show answer', and 'Next' buttons, along with a green message: 'Well done! No errors.'

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.