

EXPERIMENT NO.4

Name:Vaishnal Mali

Class:D20A

Roll No:32

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:

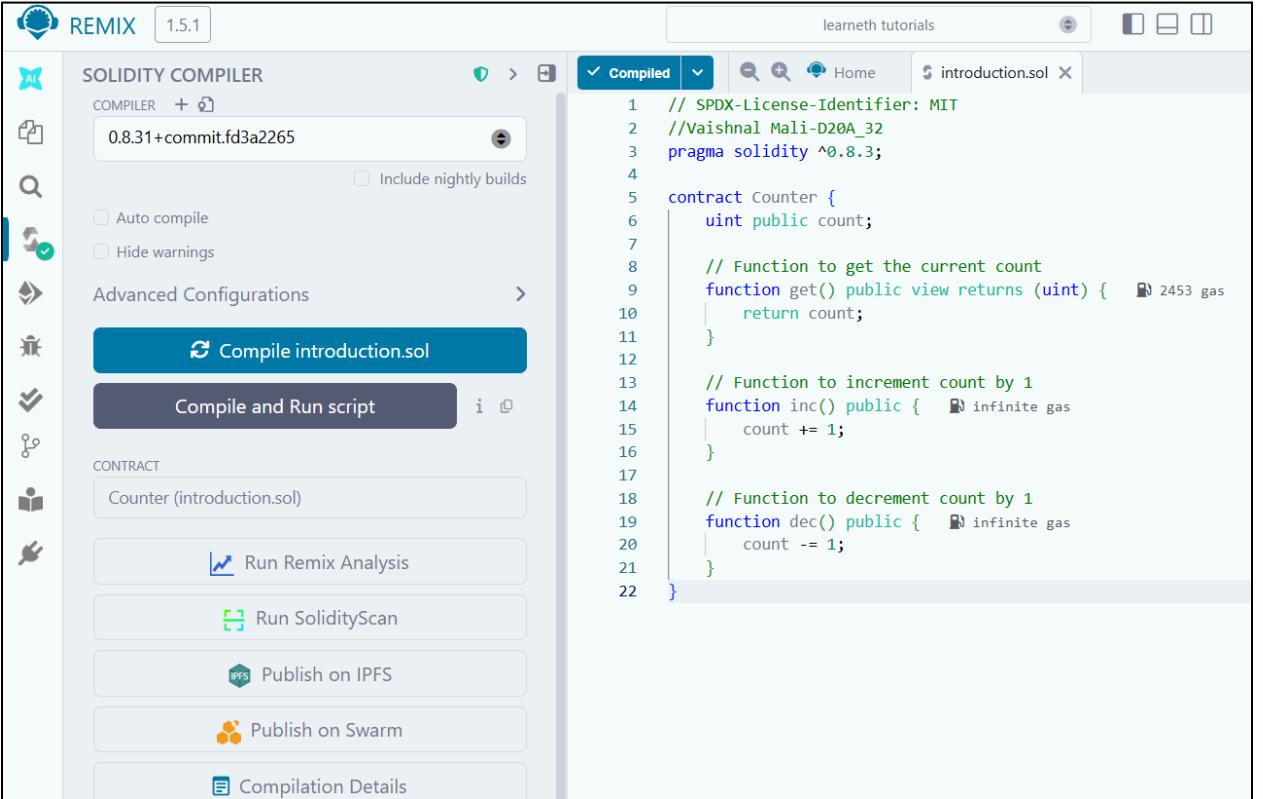
```
function sendEther(address payable recipient) public payable {
```

```
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



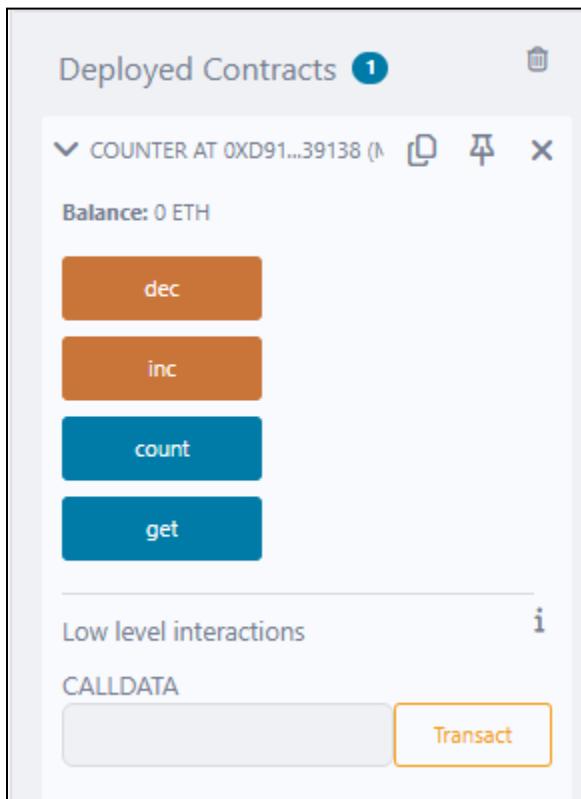
The screenshot shows the REMIX IDE interface. On the left, the SOLIDITY COMPILER panel displays the version 0.8.31+commit.fd3a2265 and includes options for Auto compile, Hide warnings, and Advanced Configurations. A prominent blue button labeled "Compile introduction.sol" is visible. On the right, the CONTRACT panel shows the "Counter (introduction.sol)" contract. The code editor contains the following Solidity code:

```
// SPDX-License-Identifier: MIT  
//Vaishnal Mali-D20A_32  
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

- Tutorial no. 1 - get

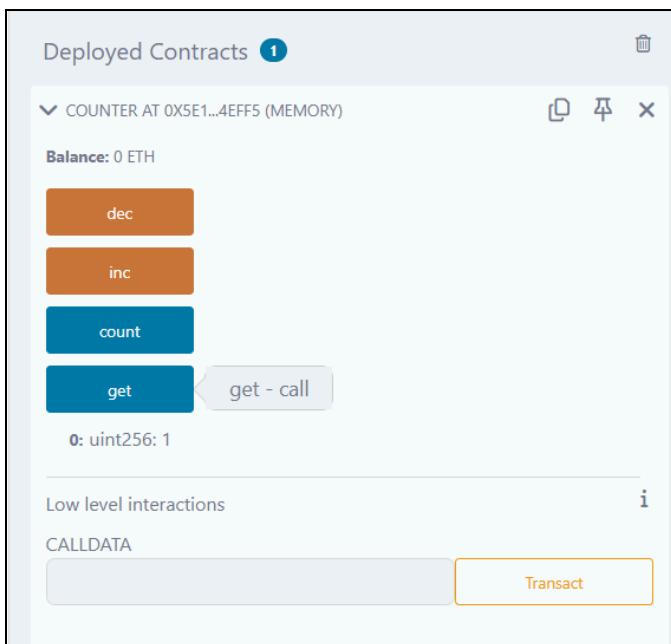
[call]	from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4	to: Counter.get()	data: 0x6d4...ce63c
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4		🔗
to	Counter.get()	0xd9145CCE52D386f254917e481e044e9943F39138	🔗
execution cost	2453 gas (Cost only applies when called by a contract)		🔗
input	0x6d4...ce63c		🔗
output	0x00		🔗
decoded input	{}	🔗	
decoded output	{ "0": "uint256: 0" }	🔗	
logs	[]	🔗	
raw logs	[]	🔗	



- Tutorial no. 1 – Increment



- Tutorial no. 1 – Decrement



● Tutorial no.2

The screenshot shows the REMIX IDE interface with a Solidity tutorial titled "2. Basic Syntax". The code editor on the right contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 // compiler version must be greater than or equal to 0.8.3 and
3 // Vaishnal Mali-D20A_32
4 pragma solidity ^0.8.3;
5
6 contract MyContract {
7     string public name = "Alice";
8 }

```

The left panel displays the tutorial content, which includes text about variable declaration, visibility, and a list of assignment tasks.

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 at the bottom include "Check Answer", "Show answer", and "Next". A message at the bottom says "Well done! No errors."

● Tutorial no.3

The screenshot shows the REMIX IDE interface with a Solidity tutorial titled "3. Primitive Data Types". The code editor on the right contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 // Vaishnal Mali-D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Primitives {
6     // Given variable (existing one)
7     address public addr = 0x11111111111111111111111111111111;
8
9     // 1 New public address variable (different from addr)
10    address public newAddr = 0x22222222222222222222222222222222;
11
12    // 2 Public negative number variable
13    // Using int8 (small signed integer type)
14    int8 public neg = -10;
15
16    // 3 Smallest uint size type with smallest uint value
17    // Smallest uint type is uint8
18    // Smallest possible uint value is 0
19    uint8 public newU = 0;
20 }

```

The left panel displays the tutorial content, which includes text about primitive data types and a list of assignment tasks.

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 at the bottom include "Check Answer", "Show answer", and "Next". A message at the bottom says "Well done! No errors."

- Tutorial no.4

Global Variables, also called *special variables*, exist in the global namespace. They don't need to be declared but can be accessed from within your contract. Global Variables are 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 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.

[Explain contract](#)

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Variables {
6     // 1 Public state variable
7     uint256 public blockNumber;
8
9     // 2 Function to store current block number
10    function doSomething() public {
11        blockNumber = block.number;
12    }
13 }

```

- Tutorial no.5

To define a function, use the `function` keyword followed by a unique name.

If the function takes inputs like our `set` function (line 9), 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 `view` or `pure` 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 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

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

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

Next

Well done! No errors.

[Explain contract](#)

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract SimpleStorage {
6     // 1 Public state variable initialized to true
7     bool public b = true;
8
9     // 2 Public function that returns the value of b
10    function get_b() public view returns (bool) {
11        return b;
12    }
13 }

```

- Tutorial no.6

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

- Title Bar:** REMIX 1.5.1, learneth tutorials
- Left Sidebar:** Icons for file operations (New, Open, Save, etc.), search, and navigation.
- Top Navigation:** Compile, Home, readAndWrite.sol, viewAndPure.sol
- Content Area:**
 - Section:** LEARNETH, Tutorials list, 5.2 Functions - View and Pure, 6 / 19
 - Text:** 4. Calling any function not marked pure.
5. Using inline assembly that contains certain opcodes."
 - Text from documentation:** You can declare a pure function using the keyword `pure`. In this contract, `add` (line 13) is a pure function. This function takes the parameters `i` and `j`, and returns the sum of them. It neither reads nor modifies the state variable `x`.
 - Description:** 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.
 - Link:** Watch a video tutorial on View and Pure Functions.
 - Assignment Section:**
 - Section:** ★ Assignment
 - Text:** 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`.
 - Buttons:** Check Answer, Show answer, Next
 - Feedback:** Well done! No errors.
 - Contract View:** Shows the Solidity code for the `ViewAndPure` contract.

- Tutorial no.7

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

- Title Bar:** REMIX 1.5.1, learneth tutorials, Login
- Left Sidebar:** Icons for file operations, search, and navigation.
- Top Navigation:** Compile, Home, modifiersAndConstructors.sol, modifiersAndConstructors.sol
- Content Area:**
 - Section:** LEARNETH, Tutorials list, 5.3 Functions - Modifiers and Constructors, 7 / 19
 - Section:** Constructor
 - Text:** A constructor function is executed upon the creation of a contract. You can use it to run contract initialization code. The constructor can have parameters and is especially useful when you don't know certain initialization values before the deployment of the contract.
 - Text:** You declare a constructor using the `constructor` keyword. The constructor in this contract (line 11) sets the initial value of the `owner` variable upon the creation of the contract.
 - Text:** Watch a video tutorial on Function Modifiers.
 - Assignment Section:**
 - Section:** ★ Assignment
 - List:**
 - 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.
 - Buttons:** Check Answer, Show answer, Next
 - Feedback:** Well done! No errors.
 - Contract View:** Shows the Solidity code for the `FunctionModifier` contract.

● Tutorial no.8

The screenshot shows the REMIX IDE interface with a Solidity contract named `modifiersAndConstructors.sol`. The code defines a function `returnTwo()` that returns two values: `-2` and `true`. The gas cost for this function is 428 gas.

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Function {
6     function returnTwo() public pure returns (int256 a, bool b) { 428 gas
7         a = -2;
8         b = true;
9     }
10 }
```

Input and Output restrictions

There are a few restrictions and best practices for the input and output parameters of contract functions.

"[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible." From the [Solidity documentation](#).

Arrays can be used as parameters, as shown in the function `arrayInput` (line 71). Arrays can also be used as return parameters as shown in the function `arrayOutput` (line 76).

You have to be cautious with arrays of arbitrary size because of their gas consumption. While a function using very large arrays as inputs might fail when the gas costs are too high, a function using a smaller array might still be able to execute.

[Watch a video tutorial on Function Outputs.](#)

Assignment

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

Check Answer **Show answer**

Well done! No errors.

● Tutorial no.9

The screenshot shows the REMIX IDE interface with a Solidity contract named `modifiersAndConstructors.sol`. The code demonstrates visibility rules between contracts. It includes an `external` modifier and a `modifier` for checking ownership.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Vaishnal Mali D20A_32
4
5 contract FunctionModifier {
6     // We will use these variables to demonstrate how to use
7     // modifiers.
8     address public owner;
9     uint public x = 10;
10    bool public locked;
11
12    constructor() { 461217 gas 414400 gas
13        // Set the transaction sender as the owner of the contract.
14        owner = msg.sender;
15    }
16
17    // Modifier to check that the caller is the owner of
18    // the contract.
19    modifier onlyOwner() {
20        require(msg.sender == owner, "Not owner");
21        // Underscore is a special character only used inside
22        // a function modifier and it tells Solidity to
23        // execute the rest of the code.
24        ;
25    }
26
27    // Modifiers can take inputs. This modifier checks that the
28    // address passed in is not the zero address.
29    modifier validAddress(address _addr) {
30        require(_addr != address(0), "Not valid address");
31        ;
32    }
}
```

external

- Can be called from other contracts or transactions
- State variables can not be `external`

In this example, we have two contracts, the `Base` contract (line 4) and the `Child` contract (line 55) which inherits the functions 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.

● Tutorial no.10

LEARNETH

7.1 Control Flow - If/Else

In this contract, the `foo` function uses the `else if` statement (line 10) to return 0 if none of the other conditions are met.

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.

Code View (Right Panel):

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract IfElse {
6     function foo(uint x) public pure returns (uint) { infinite gas
7         if (x < 10) {
8             return 0;
9         } else if (x < 20) {
10            return 1;
11        } else {
12            return 2;
13        }
14    }
15
16    function ternary(uint _x) public pure returns (uint) { infinite gas
17        return _x < 10 ? 1 : 2;
18    }
19
20    // ★ Assignment Solution
21    function evenCheck(uint _x) public pure returns (bool) { infinite gas
22        return _x % 2 == 0 ? true : false;
23    }
24}
25

```

Action Buttons:

- Check Answer
- Show answer

Well done! No errors.

Explain contract

● Tutorial no.11

LEARNETH

7.2 Control Flow - Loops

11 / 19

executed at least once, before checking on the condition.

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.

Code View (Right Panel):

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Loop {
6     uint public count;
7
8     function loop() public { infinite gas
9         for (uint i = 0; i < 10; i++) {
10             if (i == 3) {
11                 continue;
12             }
13             if (i == 5) {
14                 break;
15             }
16             // Increment inside loop
17             count += 2;
18         }
19         // Increment once more after loop
20         count += 1;
21
22         // while loop (unchanged)
23         uint j;
24         while (j < 10) {
25             j++;
26         }
27
28         // while loop (unchanged)
29         uint k;
30         while (k < 10) {
31             k++;
32         }
33
34     }
35
36 }
37

```

Action Buttons:

- Check Answer
- Show answer

Next

Well done! No errors.

Explain contract

- Tutorial no.12

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons. The main area displays a tutorial titled "8.1 Data Structures - Arrays". The text explains how to use the `pop()` function to delete the last element of a dynamic array. It also discusses using the `delete` operator to remove elements from an array. Below this, there's a section on "Array length" and a "Assignment" section with two tasks:

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

At the bottom of the tutorial area, there are "Check Answer" and "Show answer" buttons, along with a "Next" button. A message says "Well done! No errors." On the right side of the interface, the Solidity code for `arrays.sol` is shown:

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Array {
6     uint[] public arr;
7     uint[] public arr2 = [1, 2, 3];
8
9     // Fixed sized array, all elements initialize to 0
10    uint[10] public myFixedSizeArr;
11
12    uint[3] public arr3 = [0, 1, 2];
13
14    function get(uint i) public view returns (uint) {
15        return arr[i];
16    }
17
18    // Modified to return arr
19    function getArr() public view returns (uint[3] memory) {
20        return arr3;
21    }
22
23    function push(uint i) public {
24        arr.push(i);
25    }
26
27    function pop() public {
28        arr.pop();
29    }
30
31    function getLength() public view returns (uint) {
32        return arr.length;
33    }

```

- Tutorial no.13

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons. The main area displays a tutorial titled "8.2 Data Structures - Mappings". It covers setting values and removing values from a mapping. Below this, there's a "Setting values" section and a "Removing values" section. There's also a "Assignment" section with three tasks:

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.

At the bottom of the tutorial area, there are "Check Answer" and "Show answer" buttons, along with a "Next" button. A message says "Well done! No errors." On the right side of the interface, the Solidity code for `mappings.sol` is shown:

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
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    }
24
25    contract NestedMapping {
26        // Nested mapping (mapping from address to another mapping)
27        mapping(address => mapping(uint => bool)) public nested;
28
29        function get(address _addr1, uint _i) public view returns (bool) {
30            // You can get values from a nested mapping
31            // even when it is not initialized
32        }
33    }

```

● Tutorial no.14

LEARNETH

8.3 Data Structures - Structs

members as parameters in parentheses (line 16).
Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 16).
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.

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Todos {
6
7     struct Todo {
8         string text;
9         bool completed;
10    }
11
12    Todo[] public todos;
13
14    function create(string memory _text) public {
15        todos.push(Todo(_text, false));
16        todos.push(Todo({text: _text, completed: false}));
17
18        Todo memory todo;
19        todo.text = _text;
20        todos.push(todo);
21    }
22
23    function get(uint _index) public view returns (string memory text, bool completed) {
24        Todo storage todo = todos[_index];
25        return (todo.text, todo.completed);
26    }
27
28    function update(uint _index, string memory _text) public {
29        Todo storage todo = todos[_index];
30        todo.text = _text;
31    }
32

```

● Tutorial no.15

LEARNETH

8.4 Data Structures - Enums

15 / 19

Updating an enum value
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.

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Enum {
6
7     enum Status {
8         Pending,
9         Shipped,
10        Accepted,
11        Rejected,
12        Canceled
13    }
14
15    Status public status;
16
17    function get() public view returns (Status) {
18        return status;
19    }
20
21    function set(Status _status) public {
22        status = _status;
23    }
24
25    function cancel() public {
26        status = Status.Canceled;
27    }
28
29    function reset() public {
30        delete status;
31    }
32

```

Explain contract

- Tutorial no.16

LEARNETH

9. Data Locations

As we said in the beginning, when creating contracts we have to be mindful of gas costs. Therefore, we need to use data locations that require the lowest amount of gas possible.

Assignment

1. Change the value of the `[myStruct]` member `foo`, inside the `function f`, to 4.
2. 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.
3. 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.
4. Let the function `f` return `[myStruct]`, `[myMemStruct2]`, and `[myMemStruct3]`.

Tip: Make sure to create the correct return types for the function `f`.

Check Answer **Show answer**

Well done! No errors.

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract DataLocations {
6
7     uint[] public arr;
8     mapping(uint => address) map;
9
10    struct Mystruct {
11        uint foo;
12    }
13
14    mapping(uint => Mystruct) mystructs;
15
16    function f() public returns (
17        MyStruct memory,
18        MyStruct memory,
19        MyStruct memory
20    ) {
21
22        _f(arr, map, mystructs[1]);
23
24        MyStruct storage myStruct = myStructs[1];
25        MyStruct memory myMemStruct = MyStruct(0);
26
27        // 1 Change storage struct value
28        myStruct.foo = 4;
29
30        // 2 Create memory struct from myMemStruct
31        MyStruct memory myMemStruct2 = myMemStruct;
32        myMemStruct2.foo = 1;
33
34    }
35
36    function _f(uint[] memory arr, mapping(uint => address) memory map, mapping(uint => MyStruct) memory myStructs) internal {
37        for (uint i = 0; i < arr.length; i++) {
38            map[i] = arr[i];
39            myStructs[i].foo = arr[i];
40        }
41    }
42}

```

- Tutorial no.17

LEARNETH

10.1 Transactions - Ether and Wei

To specify a unit of *Ether*, we can add the suffixes `[wei]`, `[gwei]`, or `[ether]` to a literal number.

`wei`

Wei is the smallest subunit of *Ether*, named after the cryptographer Wei Dai. *Ether* numbers without a suffix are treated as `[wei]` (line 7).

`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).

Watch a video tutorial on Ether and Wei.

Assignment

1. Create a `public uint` called `oneGwei` and set it to 1 `[gwei]`.
2. 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.

Check Answer **Show answer**

Well done! No errors.

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract EtherUnits {
6     uint public oneWei = 1 wei;
7     // 1 wei is equal to 1
8     bool public isOneWei = 1 wei == 1;
9
10    uint public oneEther = 1 ether;
11    // 1 ether is equal to  $10^{18}$  wei
12    bool public isOneEther = 1 ether == 1e18;
13
14    uint public oneGwei = 1 gwei;
15    // 1 ether is equal to  $10^9$  wei
16    bool public isOneGwei = 1 gwei == 1e9;
17}

```

● Tutorial no.18

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with various icons. The main content area is titled "10.2 Transactions - Gas and Gas Price" and shows the following text:

Gas prices are denoted in gwei.

Gas limit

When sending a transaction, the sender specifies the maximum amount of gas that they are willing to pay for. If they set the limit too low, their transaction can run out of *gas* before being completed, reverting any changes being made. In this case, the *gas* was consumed and can't be refunded.

Learn more about *gas* on [ethereum.org](#).

Watch a video tutorial on Gas and Gas Price.

Assignment

Create a new `public` state variable in the `Gas` contract called `cost` of the type `uint`. Store the value of the gas cost for deploying the contract in the new variable, including the cost for the value you are storing.

Tip: You can check in the Remix terminal the details of a transaction, including the gas cost. You can also use the Remix plugin *Gas Profiler* to check for the gas cost of transactions.

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

On the right side of the interface, there is a code editor window showing Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 //Vaishnal Mali D20A_32
3 pragma solidity ^0.8.3;
4
5 contract Gas {
6     uint public i = 0;
7     uint public cost = 170367;
8
9     // Using up all of the gas that you send causes you
10    // State changes are undone.
11    // Gas spent are not refunded.
12    function forever() public {
13        // Here we run a loop until all of the gas are
14        // and the transaction fails
15        while (true) {
16            i += 1;
17        }
18    }
19 }
```

● Tutorial no.19

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with various icons. The main content area is titled "10.3 Transactions - Sending Ether" and shows the following text:

If you change the parameter type for the functions `sendViaTransfer` and `sendViaSend` (line 33 and 38) from `payable address` to `address`, you won't be able to use `transfer()` (line 35) or `send()` (line 41).

Watch a video tutorial on Sending Ether.

Assignment

Build a charity contract that receives Ether that can be withdrawn by a beneficiary.

1. Create a contract called `Charity`.
2. Add a public state variable called `owner` of the type `address`.
3. Create a donate function that is public and payable without any parameters or function code.
4. Create a withdraw function that is public and sends the total balance of the contract to the `owner` address.

Tip: Test your contract by deploying it from one account and then sending Ether to it from another account. Then execute the withdraw function.

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

On the right side of the interface, there is a code editor window showing Solidity code:

```

44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69 
```

```

function sendViaCall(address payable _to) public payable {
    // Call returns a boolean value indicating success or failure.
    // This is the current recommended method to use.
    (bool sent, bytes memory data) = _to.call{value: msg.value}("");
    require(sent, "Failed to send Ether");
}

// Vaishnal Mali D20A_32
contract Charity {
    address public owner;
}

constructor() {
    owner = msg.sender;
}

function donate() public payable {} 141 gas

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

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.