

Experiment No. 4

Aim: Hands on Solidity programming assignments for creating smart contracts

Theory:

Q1: Data Types, Variables, and Function Modifiers

Solidity is a statically typed language, meaning the type of each variable must be specified at compile time.

- **Integers:** uint (unsigned) and int (signed) come in steps of 8 bits (e.g., uint8 to uint256). Using the correct size can save gas, though uint256 is the most common as the EVM is optimized for 256-bit words.
- **Address:** This is a unique type to Solidity. A payable address is a subtype that allows you to send Ether to it using .transfer() or .send().
- **State vs. Local:** State variables are written into the contract's storage (the blockchain's "hard drive"), making them expensive to change. Local variables exist only in the "stack" during execution and are much cheaper.
- **Pure vs. View:** View: You are looking at the state (e.g., checking a balance) but not touching it.
Pure: You aren't even looking at the state (e.g., a math utility like $2 + 2$).

Q2: Inputs and Outputs

Functions are the primary way users interact with the blockchain.

- **Multiple Returns:** Unlike many languages, Solidity supports returning multiple values natively: return (uint a, bool b, string memory c);.
- **Named Returns:** You can define the variable name in the function signature: returns (uint sum). This allows you to simply assign a value to sum inside the function without an explicit return statement at the end.

Q3: Visibility, Modifiers, and Constructors

Visibility is the first line of security in a smart contract.

- **External vs. Public:** external functions are sometimes more gas-efficient than public when receiving large arrays because the data is read directly from calldata rather than being copied to memory.
- **Modifiers:** These act as "gatekeepers." A common pattern is the onlyOwner modifier, which checks if msg.sender == owner before allowing the rest of the function code (represented by the _ symbol) to run.
- **Constructors:** If you don't define one, a default constructor is used. Once the contract is deployed, the constructor code is discarded and can never be called again.

Q4: Control Flow

While Solidity supports standard loops, the Gas Limit introduces a unique constraint.

- **The Denial of Service (DoS) Risk:** If you create a for loop that iterates through a dynamic array of users, and that array grows too large, the gas required to finish the loop might exceed the block gas limit. This would make the function impossible to execute, effectively "bricking" the contract.
- **Recommendation:** Favor if-else logic over heavy loops whenever possible.

Q5: Data Structures

- **Mappings:** Think of these as a Hash Table. They are incredibly gas-efficient for lookups. However, you cannot "list" all keys in a mapping; if you need to know who all the users are, you usually pair a mapping with an address[] array.
- **Structs:** These allow you to create complex records. For example, a Request struct in a crowdfunding contract might contain a description, a value, and a recipient.
- **Enums:** These restrict a variable to have one of only a few predefined values, which reduces human error and makes the code self-documenting.

Q6: Data Locations (Storage, Memory, Calldata)

This is often the most confusing part for beginners but is vital for gas optimization.

- Storage: Permanent and expensive. Use this for data that must persist between transactions.
- Memory: Temporary and cheaper. Use this for data that you need to manipulate during a function call.
- Calldata: The cheapest. It is an immutable (read-only) area where function arguments are stored.

Pro Tip: Changing a variable from storage to memory creates a copy. Changing a pointer to storage affects the actual state variable on the blockchain.

Q7: Transactions and Gas

- Wei: Just as a Dollar has Cents, an Ether has Wei.
 - $1 \text{ {Ether}} = 1,000,000,000,000,000 \text{ {Wei}} (10^{18})$.
- Gas: Think of Gas as the "fuel" and Gas Price as the "price per gallon."
 - $\text{{Total Cost}} = \text{{Gas Used}} \times \text{{Gas Price}}$
- The call Method: While transfer() was once the standard, the Ethereum community now recommends using .call{value: amount}("") for sending Ether, as it is more flexible and handles gas more gracefully for modern contract interactions.

Tasks Performed:-

Assignment 1:

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is visible, showing the account '0x5B3...eddC4' and the gas limit '3000000'. The 'VALUE' is set to '0' Wei. The 'CONTRACT' is 'Counter - remix-project-org/remix-w'. The 'Deploy' button is highlighted. On the right, the 'introduction.sol' file is open, showing the Solidity code for the 'Counter' contract. The code includes a 'uint public count;' variable and a 'get' function. Below the code, the 'Explain contract' panel shows the available libraries and the transaction details, including the transaction hash and the contract address.

The two screenshots show the 'Counter' contract interface. The left screenshot shows the 'get' function being called, resulting in a value of 0. The right screenshot shows the 'get' function being called, resulting in a value of 1. Both screenshots show the 'Balance: 0 ETH' and the 'Low level interactions' section with a 'Transact' button.

2: Basic Syntax

2. Basic Syntax

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.

Watch a video tutorial on Basic Syntax.

★ 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.

Check Answer Show answer

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 // compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
3 //Shreyash Dhekane D20A 23
4 pragma solidity ^0.8.3;
5
6 contract MyContract {
7     string public name = "Alice";
8 }
```

3: Primitive Data Types

3. Primitive Data Types

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`.

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.

```
19 //
20 Negative numbers are allowed for int types.
21 Like uint, different ranges are available from int8 to int256
22 */
23 int8 public i8 = -1;
24 int public i256 = 456;
25 int public i = -123; // int is same as int256
26
27 address public defaultAddr;
28
29 // Default
30 remix-project-org/remix-workshops/3. Primitive Data Types/primitiveDataTypes
31 bool public defaultBool;
32 uint public defaultUint; // 0
33 int public defaultInt; // 0
34 address public defaultAddr; // 0x0000000000000000000000000000000000000000
35
36 // New values
37 address public newAddr = 0x0000000000000000000000000000000000000000;
38 int public neg = -12;
39 uint8 public newU = 0;
40
41 }
42 // Shreyash Dhekane D20A 23
```

4: Variables

4. Variables

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.

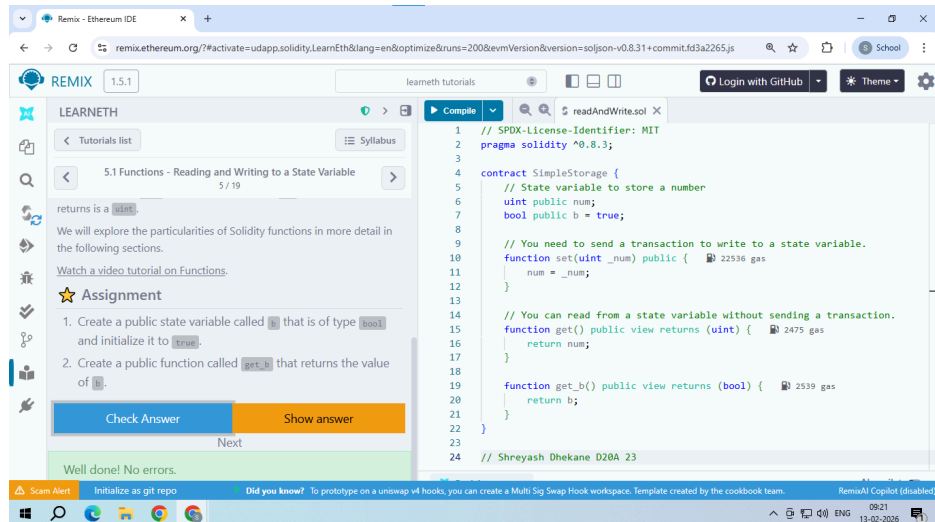
Check Answer Show answer

Next

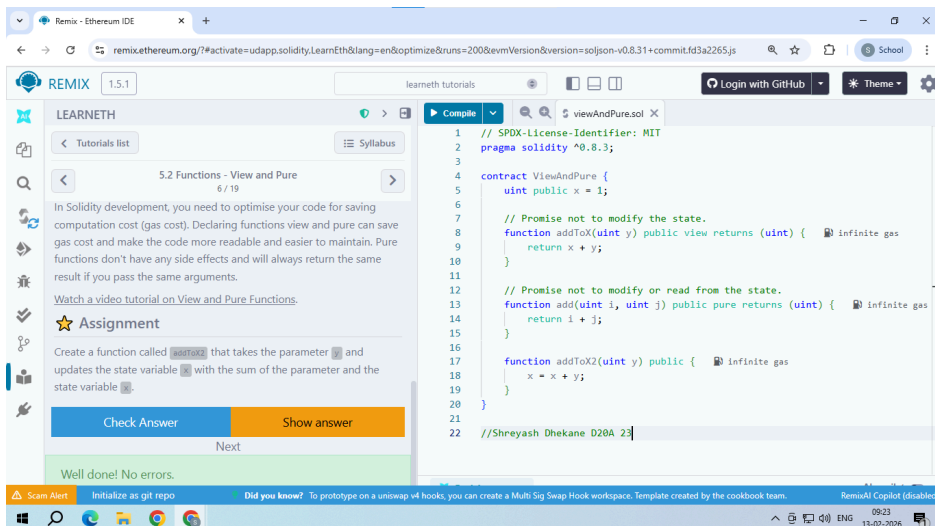
Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Variables {
5     // State variables are stored on the blockchain.
6     string public text = "Hello";
7     uint public num = 123;
8     uint public blockNumber;
9
10    function doSomething() public { 22334 gas
11        // Local variables are not saved to the blockchain.
12        uint i = 456;
13
14        // Here are some global variables
15        uint timestamp = block.timestamp; // Current block timestamp
16        address sender = msg.sender; // address of the caller
17        blockNumber = block.number;
18    }
19 }
20 // Shreyash Dhekane D20A 23
```

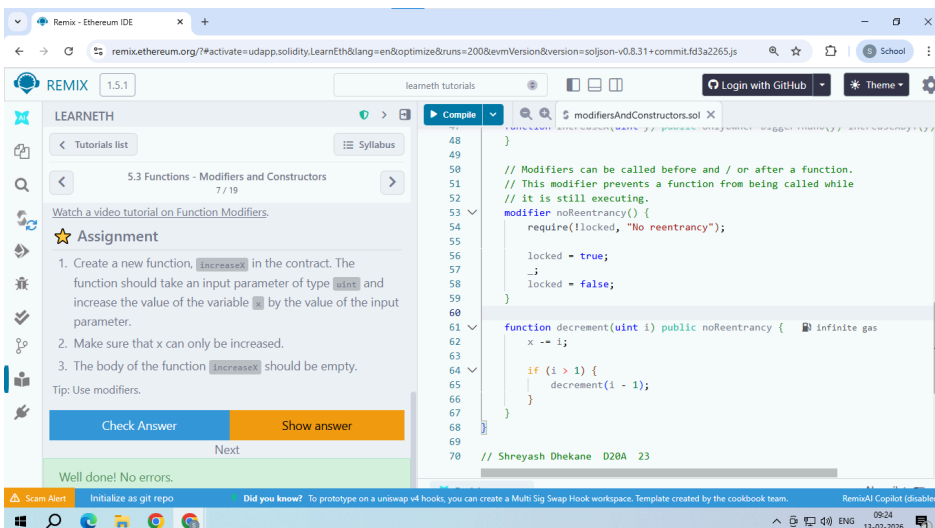
5: Functions



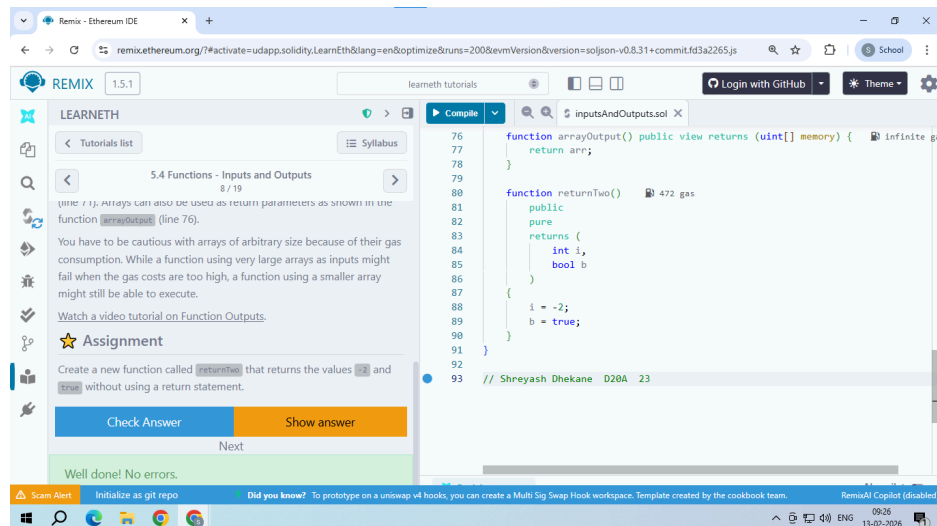
6: Functions



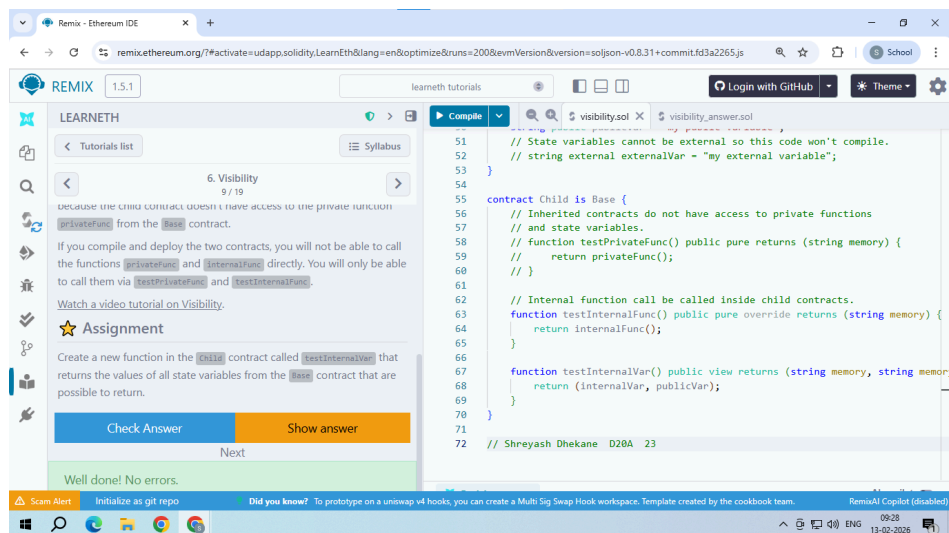
7: Functions



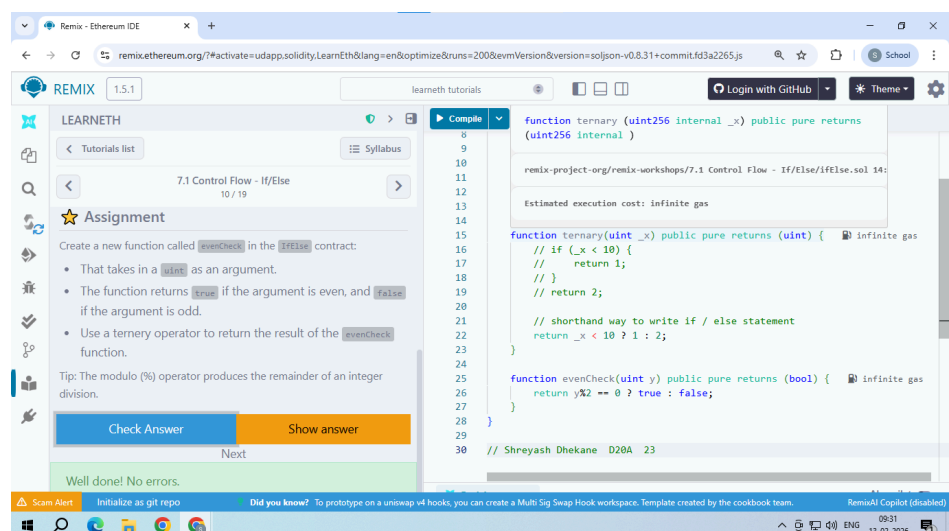
8: Functions



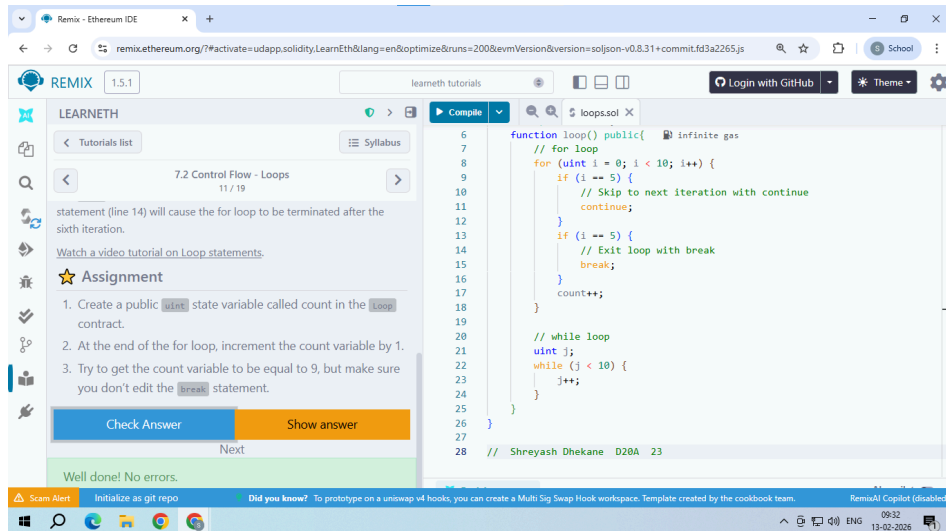
9: Visibility



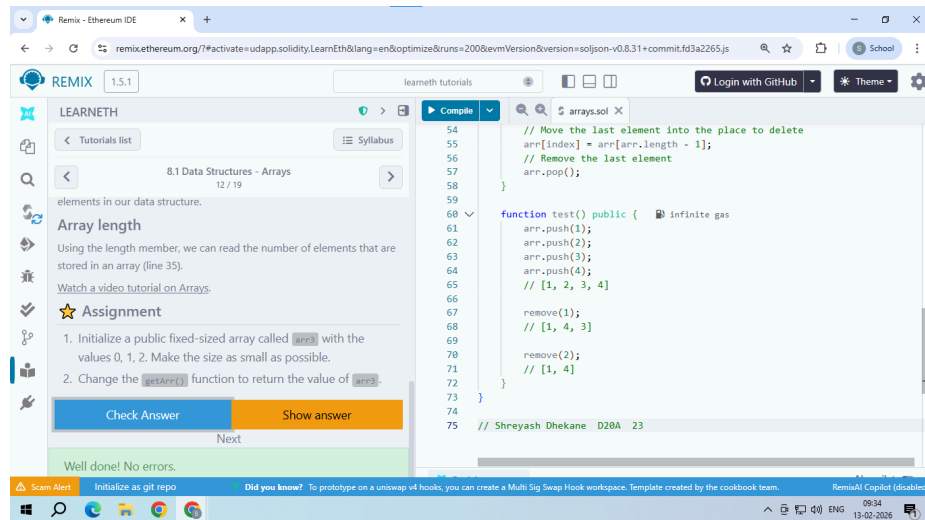
10: Control Flow



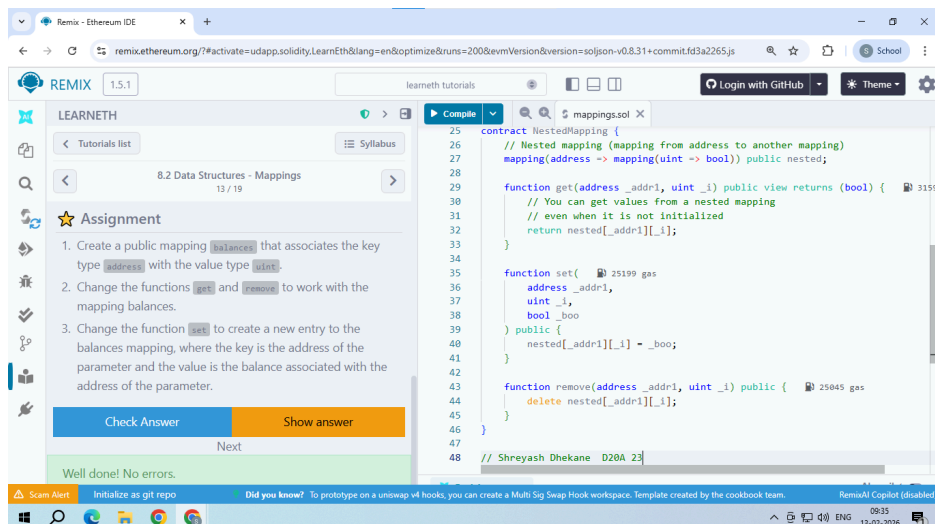
11: Control Flow



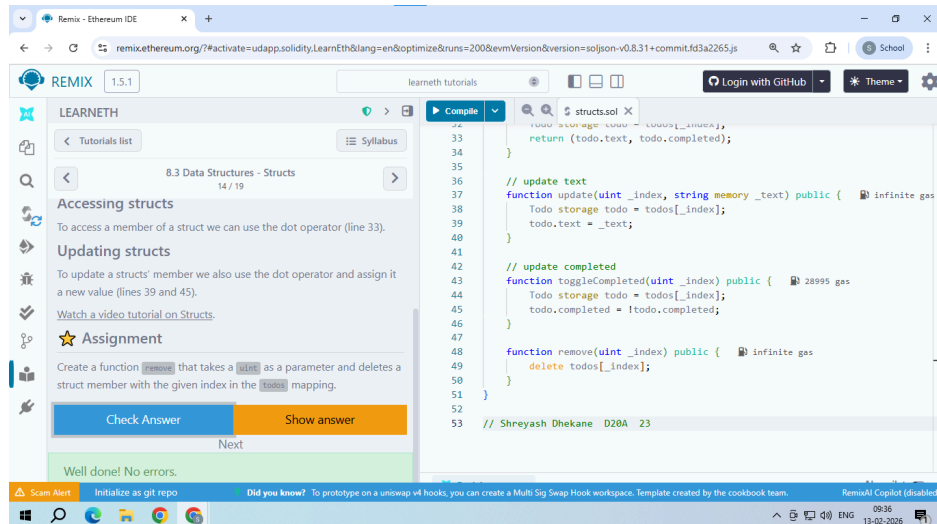
12: Data Structure - Arrays



13: Data Structures - Mapping



14: Data Structures - Structs



REMIX - Ethereum IDE

remix.ethereum.org/?#activate=udapp.solidity.LearnEth&lang=en&optimize&runs=200&evmVersion=soljson-v0.8.31+commit.f3a2265.js

LEARNETH

8.3 Data Structures - Structs

Accessing structs

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

Updating structs

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

Watch a video tutorial on Structs.

Assignment

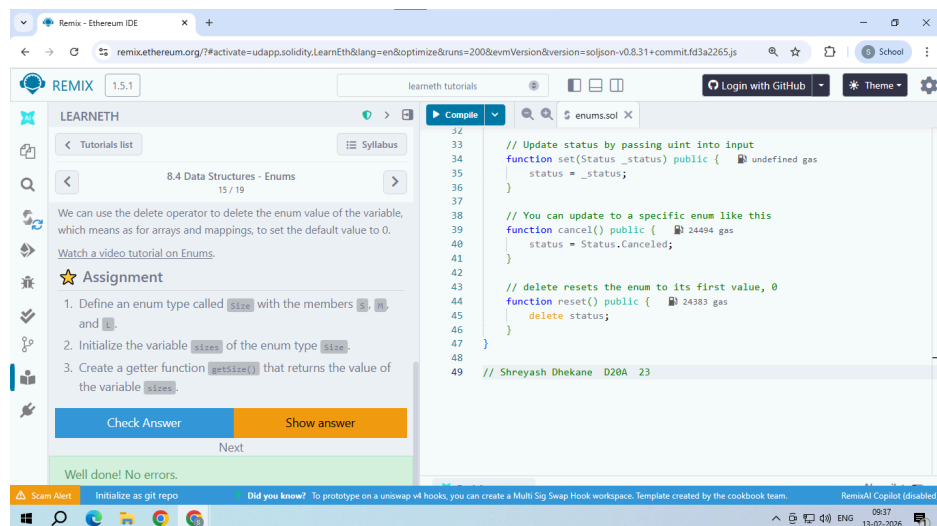
Create a function `remove` that takes a `uint` as a parameter and deletes a struct member with the given index in the `todos` mapping.

Check Answer Show answer

Well done! No errors.

```
33 // update text
34 function updateText(uint _index, string memory _text) public {
35     // update text
36     function update(uint _index, string memory _text) public {
37         // update text
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
48 function remove(uint _index) public {
49     delete todos[_index];
50 }
51 }
52
53 // Shreyash Dhekane D20A 23
```

15: Data structures - Enums



REMIX - Ethereum IDE

remix.ethereum.org/?#activate=udapp.solidity.LearnEth&lang=en&optimize&runs=200&evmVersion=soljson-v0.8.31+commit.f3a2265.js

LEARNETH

8.4 Data Structures - Enums

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 `Small` and `Big`.

2. Initialize the variable `size` of the enum type `Size`.

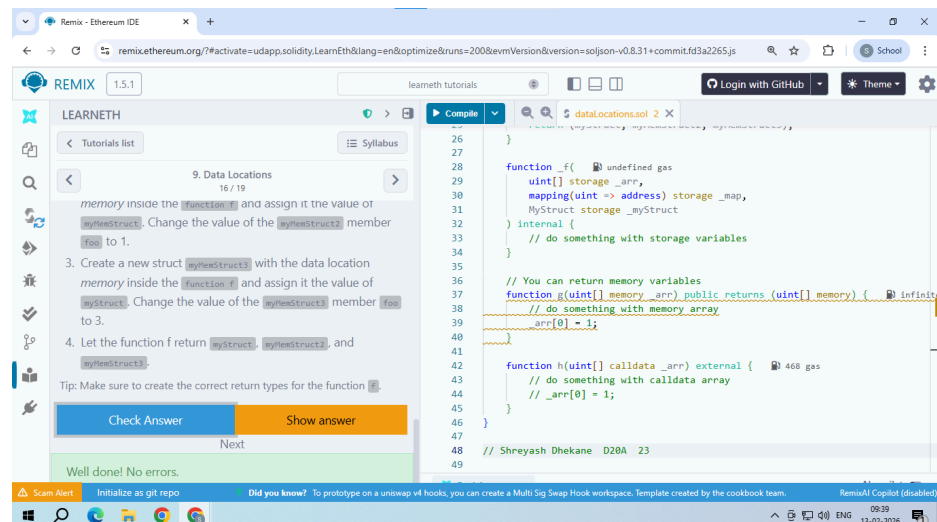
3. Create a getter function `getSize()` that returns the value of the variable `size`.

Check Answer Show answer

Well done! No errors.

```
33 // Update status by passing uint into input
34 function setStatus(uint _status) public {
35     status = _status;
36 }
37
38 // You can update to a specific enum like this
39 function cancel() public {
40     status = Status.Canceled;
41 }
42
43 // delete resets the enum to its first value, 0
44 function reset() public {
45     delete status;
46 }
47 }
48
49 // Shreyash Dhekane D20A 23
```

16: Data Locations



REMIX - Ethereum IDE

remix.ethereum.org/?#activate=udapp.solidity.LearnEth&lang=en&optimize&runs=200&evmVersion=soljson-v0.8.31+commit.f3a2265.js

LEARNETH

9. Data Locations

memory inside the function `f` and assign it the value of `myNewStruct`. Change the value of the `myNewStruct` member `foo` to 1.

3. Create a new struct `myNewStruct3` with the data location memory inside the function `f` and assign it the value of `myStruct`. Change the value of the `myNewStruct3` member `foo` to 3.

4. Let the function `f` return `myStruct`, `myNewStruct3`, and `myNewStruct2`.

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

Check Answer Show answer

Well done! No errors.

```
26 }
27
28 function _f() {
29     uint[] storage _arr;
30     mapping(uint => address) storage _map;
31     MyStruct storage _myStruct;
32     internal {
33         // do something with storage variables
34     }
35 }
36
37 // You can return memory variables
38 function g(uint[] memory _arr) public returns (uint[] memory) {
39     // do something with memory array
40     _arr[0] = 1;
41 }
42
43 function h(uint[] calldata _arr) external {
44     // do something with calldata array
45     _arr[0] = 1;
46 }
47
48 // Shreyash Dhekane D20A 23
```


17: Transactions

REMUX 1.5.1

learnth tutorials

Learn with GitHub

Theme

LEARNTETH

Tutorials list

Syllabus

10.1 Transactions - Ether and Wei

17 / 19

One `wei` (giga-wei) is equal to 1,000,000,000 (10^9) `wei`.

One `ether` is equal to 1,000,000,000,000,000 (10^{18}) `wei` (line 11).

Watch a video tutorial on Ether and Wei.

★ Assignment

1. Create a `public uint` called `oneWei` and set it to 1 `wei`.
2. Create a `public bool` called `isOneWei` and set it to the result of a comparison operation between 1 `wei` and 10^9 .

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

Check Answer Show answer

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract EtherUnits {
5     uint public oneWei = 1 wei;
6     // 1 wei is equal to 1
7     bool public isOneWei = 1 wei == 1;
8
9     uint public oneEther = 1 ether;
10    // 1 ether is equal to 10^18 wei
11    bool public isOneEther = 1 ether == 1e18;
12
13    uint public oneGwei = 1 gwei;
14    // 1 ether is equal to 10^9 wei
15    bool public isOneGwei = 1 gwei == 1e9;
16 }
17
18 // Shreyash Dhekane D20A 23
```

18: Transactions

REMUX 1.5.1

learnth tutorials

Learn with GitHub

Theme

LEARNTETH

Tutorials list

Syllabus

10.2 Transactions - Gas and Gas Price

18 / 19

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.

Check Answer Show answer

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Gas {
5     uint public i = 0;
6     uint public cost = 170367;
7
8     // Using up all of the gas that you send causes your transaction to fail.
9     // State changes are undone.
10    // Gas spent are not refunded.
11    function forever() public {
12        // Here we run a loop until all of the gas are spent
13        // and the transaction fails
14        while (true) {
15            i += 1;
16        }
17    }
18 }
19
20 // Shreyash Dhekane D20A 23
```

19: Transactions

REMUX 1.5.1

learnth tutorials

Learn with GitHub

Theme

LEARNTETH

Tutorials list

Syllabus

10.3 Transactions - Sending Ether

19 / 19

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.

Check Answer Show answer

Next

Well done! No errors.

```
51 }
52
53 contract Charity {
54     address public owner;
55
56     constructor() {
57         owner = msg.sender;
58     }
59
60     function donate() public payable {
61         // 141 gas
62     }
63
64     function withdraw() public {
65         // infinite gas
66         uint amount = address(this).balance;
67         (bool sent, bytes memory data) = owner.call{value: amount}("");
68         require(sent, "Failed to send Ether");
69     }
70 }
71
72 // Shreyash Dhekane D20A 23
```


Conclusion:

This experiment provided a comprehensive immersion into the Solidity programming language, bridging the gap between theoretical blockchain architecture and practical software engineering. By utilizing the Remix IDE, a sophisticated development environment, a wide array of technical constructs were explored and implemented, including state variables, complex data structures like mappings and structs, and the nuances of visibility specifiers and custom modifiers. This hands-on approach allowed for a granular examination of the smart contract lifecycle—from the initial state defined by constructors to the execution of complex logic via control flow statements—ensuring a robust understanding of how code behaves within a decentralized context.