

EXPERIMENT NO 04

Name: Tejas Gunjal

Class: D20A

Roll: No: 24

Batch: B

Aim: Hands on Solidity Programming Assignments for creating Smart Contracts.

Theory:

1. Introduction to Solidity and Smart Contracts

Solidity is a high-level programming language used to develop smart contracts on the Ethereum blockchain. It is designed specifically for creating decentralized applications (DApps) and has a syntax similar to JavaScript and C++, making it easier for developers to understand. Solidity allows programmers to define rules, manage data, and handle transactions securely on the blockchain network.

A smart contract is a self-executing digital program stored on the blockchain that automatically performs actions when predefined conditions are met. Once deployed, it becomes immutable, meaning it cannot be easily modified, which ensures transparency and security. Smart contracts eliminate the need for intermediaries and are widely used in applications such as digital payments, token creation, voting systems, and supply chain management.

2. Primitive Data Types, Variables, and Functions (pure & view)

In Solidity, primitive data types are the fundamental elements used to store and manage information inside a smart contract. Choosing the correct data type is important because it affects storage size, gas cost, and overall efficiency of the contract.

Common Data Types

- **uint / int** – Used to store whole numbers.

uint (unsigned integer) stores only positive numbers, while int (signed integer) stores both positive and negative numbers. Different sizes such as uint8, uint128, and uint256 are available. uint256 is most commonly used because it matches the Ethereum Virtual Machine (EVM) word size and is gas-efficient.

- **bool** – Stores logical values (true or false). It is mainly used for condition checking, such as verifying whether a user is registered or whether a transaction is approved.
- **address** – Stores a 20-byte Ethereum account address. It is commonly used to represent users, contract accounts, and to send or receive Ether. Addresses also have built-in properties like .balance and functions such as .transfer().
- **string / bytes** – Used to store text and binary data. string is generally used for readable text, while bytes is more gas-efficient for handling raw data.

Types of Variables

- **State Variables** – Declared outside functions and stored permanently on the blockchain. Their values are saved in contract storage and remain available as long as the contract exists.
- **Local Variables** – Declared inside functions and exist only during function execution. They do not consume permanent storage, making them cheaper in terms of gas.
- **Global Variables** – Built-in variables provided by Solidity that give information about the blockchain and transaction. Examples include:
 - msg.sender – The address of the account that called the function.
 - msg.value – The amount of Ether (in Wei) sent with the transaction.
 - block.timestamp – The current block's timestamp.
 - block.number – The current block number.

These global variables help contracts interact with blockchain data securely.

Types of Functions

- **pure** – Functions that neither read nor modify the blockchain state. They only perform calculations using input parameters or internal variables. These functions are highly gas-efficient because they do not access storage.
- **view** – Functions that can read state variables but cannot modify them. They are commonly used to check balances or retrieve stored information without changing any data.

Using pure and view improves clarity, security, and gas optimization, as it clearly defines how a function interacts with blockchain data.

3. Inputs and Outputs in Functions

Functions in Solidity can accept inputs and return outputs.

- **Inputs (Parameters)** allow users to pass data into the contract.
- **Outputs (Return values)** send results back after execution.

For example, a function can accept an amount and return whether a transaction was successful. Solidity also allows named return values, which make the code easier to read and understand.

4. Visibility, Modifiers, and Constructors

Function Visibility

Visibility defines who can access a function:

- **public** – Can be accessed inside and outside the contract.
- **private** – Only accessible inside the same contract.

- **internal** – Accessible within the contract and its derived contracts.
- **external** – Can only be called from outside the contract.

Modifiers

Modifiers are used to add extra rules before executing a function. For example, an `onlyOwner` modifier restricts certain functions so that only the contract owner can use them.

Constructors

A constructor is a special function that runs only once when the contract is deployed. It is mainly used to initialize important values like setting the owner of the contract.

5. Control Flow: if-else and Loops

Control flow statements decide how a smart contract executes code based on conditions.

- **if-else Statement**

The if-else statement is used for decision-making. It checks whether a condition is true or false and executes code accordingly. For example, a contract can check if a user has enough balance before allowing an Ether transfer. If the condition is not satisfied, the transaction can be rejected using `require()`.

- **Loops (for, while, do-while)**

Loops are used to repeat a block of code multiple times. A for loop is commonly used to iterate through arrays, while while and do-while run as long as a condition remains true.

Since every loop consumes gas, developers must keep loops small and efficient to avoid high transaction costs or failure due to gas limits.

6. Data Structures in Solidity

Solidity provides different data structures to organize and manage information efficiently inside smart contracts.

- **Arrays:** It is used to store a list of elements of the same type. They can be fixed-size or dynamic & are commonly used to store multiple values like a list of registered users.
- **Mappings:** Mappings store data in key-value pairs, allowing fast and efficient data lookup. They are widely used for storing balances or ownership details, but they cannot be directly iterated (looped) through.
- **Structs:** Structs allow grouping multiple related variables into a single custom data type. They are useful for representing real-world objects like students or transactions.
- **Enums:** Enums define a set of predefined constant values. They improve code readability and are commonly used to represent statuses such as Pending, Active, or Closed.

7. Data Locations

Solidity uses different data locations that affect performance and gas cost.

- **storage** – Permanent data stored on the blockchain (state variables).
- **memory** – Temporary storage used during function execution.
- **calldata** – Special read-only location used for external function parameters. It is more gas-efficient than memory.

Understanding data locations helps in writing optimized and cost-effective smart contracts.

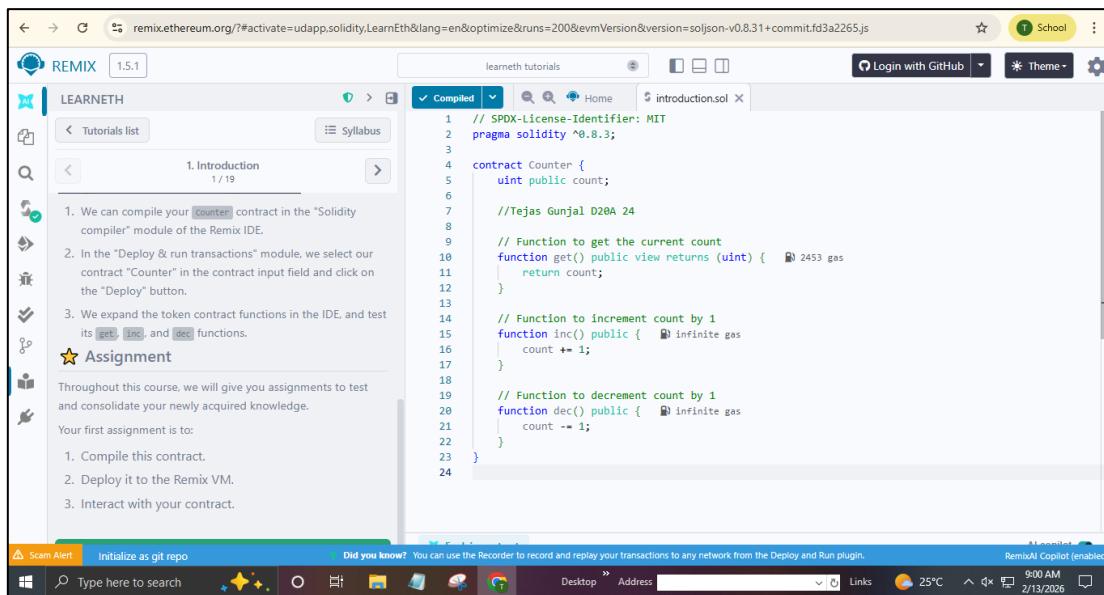
8. Transactions: Ether, Wei, Gas and Sending Transactions

- **Ether and Wei:** In Ethereum, Ether is the primary cryptocurrency used for transactions. However, all calculations are performed in Wei, the smallest unit of Ether (1 Ether = 10^{18} Wei), to maintain accuracy and avoid decimal issues in financial operations.
- **Gas and Gas Price:** Every action performed on the blockchain requires gas, which measures the computational work needed to execute a transaction. The gas price specifies how much Ether a user is willing to pay per unit of gas, and higher gas prices generally result in faster transaction processing.
- **Sending Transactions:** Transactions are executed to transfer Ether or to interact with smart contracts. Methods such as `transfer()`, `send()`, and `call()` are used for sending Ether, with `call()` offering greater flexibility. Since each transaction consumes gas, writing optimized and efficient contracts is essential.

Implementation:

➤ Tutorial 1:

Tutorial No 1 – Compile the code



The screenshot shows the Remix IDE interface. On the left, there's a sidebar with 'LEARNETH' and a 'Tutorials list' section showing '1. Introduction' (1/19). The main workspace has tabs for 'Compiled' and 'introduction.sol'. The code editor contains the following Solidity code:

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

contract Counter {
    uint public count;

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

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

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

Below the code, there's a 'Did you know?' section with the text: 'You can compile your `Counter` contract in the "Solidity compiler" module of the Remix IDE.' There are also assignment instructions and a reminder to initialize a git repo.

Deploy the contract

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with options like 'DEPLOY & RUN TRANSACTIONS', 'GAS LIMIT', 'Estimated Gas', 'Custom', 'Value', 'CONTRACT', 'Deploy', and 'At Address'. The main area displays the Solidity code for the 'Counter' contract:

```
4 contract Counter {
5     uint public count;
6
7     //Tejas Gunjal DBA 24
8
9     // Function to get the current count
10    function get() public view returns (uint) { 2453 gas
11        return count;
12    }
13
14    // Function to increment count by 1
15    function inc() public { infinite gas
16        count += 1;
17    }
18
19    // Function to decrement count by 1
20    function dec() public { infinite gas
21        count -= 1;
22    }
}
```

Below the code, the 'Transactions recorded' section shows one transaction record. The 'Deployed Contracts' section shows the deployed contract 'COUNTER AT 0x5B3...eddc4 (MEMORY)'.

Get

This screenshot is similar to the previous one but shows a balance of 0 ETH in the sidebar. The 'Transactions recorded' section shows a single transaction record. The 'Deployed Contracts' section shows the deployed contract 'COUNTER AT 0x5B3...eddc4 (MEMORY)'.

Increment

This screenshot shows the state after an 'inc()' call. The 'Transactions recorded' section shows two transaction records. The first is a constructor call to the deployed contract. The second is an 'inc()' call. The 'Deployed Contracts' section still shows the deployed contract 'COUNTER AT 0x5B3...eddc4 (MEMORY)'. The 'Balance' is now 0 ETH.

Decrement

```
contract Counter {
    uint public count;
}

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

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

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

The screenshot shows the REMIX interface with the Solidity code for a 'Counter' contract. It includes functions for getting the current count, incrementing it by 1, and decrementing it by 1. The 'inc()' and 'dec()' functions both have a note indicating they cost infinite gas.

- Tutorial 2: Creating "MyContract" with a public string variable "name" set to "Alice".

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

//TEJAS GUNJAL D20A 24
contract MyContract {
    string public name = "Alice";
}
```

The screenshot shows the REMIX interface with the Solidity code for a 'MyContract' contract. It contains a single public string variable 'name' initialized to the value 'Alice'.

- Tutorial 3: Creating address and integer variables with specific values.

```
//TEJAS GUNJAL D20A 24
// Default values
// Unassigned variables have a default value
bool public defaultBool; // false
uint public defaultUint; // 0
int public defaultInt; // 0
address public defaultAddr; // 0x0000000000000000000000000000000000000000

// New values
address public newAddr = 0x0000000000000000000000000000000000000000;
int public newInt = -12;
```

The screenshot shows the REMIX interface with the Solidity code for a contract. It includes variables of type address (newAddr) and int (newInt), both assigned specific values.

➤ Tutorial 4: Creating a state variable and assigning the current block number.

The screenshot shows the Remix IDE interface. The left sidebar displays a 'Tutorials list' for '4. Variables'. The main workspace shows the following Solidity code:

```

4 contract Variables {
5     // State variables are stored on the blockchain.
6     string public name = "Hello";
7     uint public num = 123;
8     uint public blockNumber;
9
10 //TEJAS GUNDAL 0208 24
11 function doSomething() public {
12     // Local variables are not saved to the blockchain.
13     uint j = 456;
14
15     // Here are some global variables
16     uint timestamp = block.timestamp; // Current block timestamp
17     address sender = msg.sender; // Address of the caller
18     blockNumber = block.number;
19 }
20

```

The 'Compile' tab is selected, and the 'variables.sol' file is open. Below the code, there's an 'Explain contract' section with an AI copilot button. The status bar at the bottom shows the date as 2/13/2026.

➤ Tutorial 5: Creating a boolean state variable and a function to return its value.

The screenshot shows the Remix IDE interface. The left sidebar displays a 'Tutorials list' for '5.1 Functions - Reading and Writing to a State Variable'. The main workspace shows the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7     bool public b = true;
8
9 //TEJAS GUNDAL 0208 24
10 // You need to send a transaction to write to a state variable.
11 function set(uint _num) public {
12     num = _num;
13 }
14
15 // You can read from a state variable without sending a transaction.
16 function get() public view returns (uint) {
17     return num;
18 }
19

```

The 'Compile' tab is selected, and the 'readAndWritesol' file is open. Below the code, there's an 'Explain contract' section with an AI copilot button. The status bar at the bottom shows the date as 2/13/2026.

➤ Tutorial 6: Creating functions that read state variables without modifying them.

The screenshot shows the Remix IDE interface. The left sidebar displays a 'Tutorials list' for '5.2 Functions - View and Pure'. The main workspace shows the following Solidity code:

```

4 contract ViewAndPure {
5     uint public x = 1;
6
7     //TEJAS GUNDAL 0208 24
8     // Promise not to modify the state.
9     function add(uint y) public view returns (uint) {
10         infinite gas
11         return x + y;
12     }
13
14     // Promise not to modify or read from the state.
15     function add(uint x, uint y) public pure returns (uint) {
16         infinite gas
17         return x + y;
18     }
19
20     function addToX2(uint y) public {
21         infinite gas
22         x = x + y;
23     }
24

```

The 'Compile' tab is selected, and the 'viewAndPure.sol' file is open. Below the code, there's an 'Explain contract' section with an AI copilot button. The status bar at the bottom shows the date as 2/13/2026.

➤ Tutorial 7: Creating a function with a modifier to restrict state variable modifications.

```

38 require(y > 0, "Not bigger than x");
39 ...
40 }
41 modifier increaseXby(uint y) {
42     ...
43     x = x + y;
44 }
45
46 //TEJAS GUNDAL 0204 24
47 function increaseXby(uint y) public owner biggerThan0(y) increaseXby(y) { infinite gas
48 }
49
50 // Modifiers can be called before / or after a function.
51 // This modifier prevents a function from being called while
52 // it is still executing.
53 modifier noReentry() {
54

```

➤ Tutorial 8: Creating function that returns multiple values without using a return statement.

```

76 function arrayOutput() public view returns (uint[] memory) { infinite gas
77     return arr;
78 }
79
80 //TEJAS GUNDAL 0204 24
81 function returnTwo() { 472 gas
82     public
83     pure
84     returns (
85         int i,
86         bool b
87     )
88     {
89         i = -2;
90         b = true;
91     }
92 }
```

➤ Tutorial 9: Creating a function in a child contract to access visible state variables from the base contract.

```

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     function testInternalFunc() public pure override returns (string memory) { infinite gas
64         return internalFunc();
65     }
66     //TEJAS GUNDAL 0204 24
67     function testInternalVar() public view returns (string memory, string memory) { infinite gas
68         return (internalVar, publicVar);
69     }
70 }
```

➤ Tutorial 10: Creating a function to check if a number is even using ternary operator.

```

13 }
14
15 function ternary(uint _x) public pure returns (uint) {
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 //TEJAS GUNDAL D20A 24
25 function evenCheck(uint _y) public pure returns (bool) {
26     return _y % 2 == 0 ? true : false;
27 }

```

➤ Tutorial 11: Incrementing a counter in a for loop to reach 9 while preserving the break statement.

```

12
13 if (1 == 5) {
14     // Exit loop with break
15     break;
16 }
17 count++;
18
19 //TEJAS GUNDAL D20A 24
20 // While loop
21 uint j;
22 while (j < 10) {
23     j++;
24 }
25
26
27

```

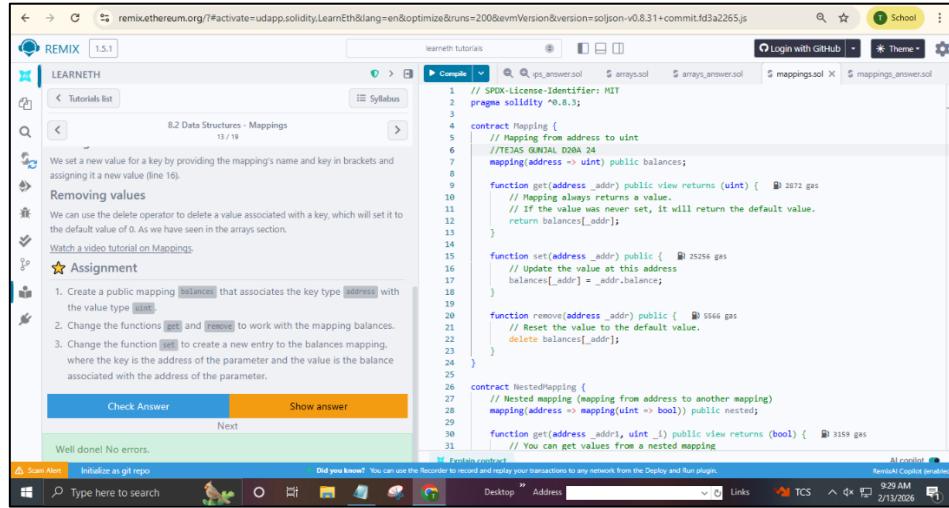
➤ Tutorial 12: Creating a fixed-size array and modifying a function to return its values.

```

7 uint[] public arr2 = [1, 2, 3];
8 // Fixed sized array, all elements initialize to 0
9 uint[10] public myFixedSizeArr;
10 uint[3] public arr3 = [0, 1, 2];
11
12 function getArr1() public view returns (uint) {
13     return arr1[1];
14 }
15 //TEJAS GUNDAL D20A 24
16 // Solidity can return the entire array.
17 // But this function should be avoided for
18 // arrays that can grow indefinitely in length.
19 function getArr() public view returns (uint[3] memory) {
20     return arr3;
21 }
22
23 function push(uint i) public {
24     // Append to array
25     // This will increase the array length by 1.
26     arr.push(i);
27 }
28
29 function pop() public {
30     // Remove last element from array
31     // This will decrease the array length by 1.
32     arr.pop();
33 }
34
35 function getLength() public view returns (uint) {
36     return arr.length;
37 }

```

- Tutorial 13: Creating an address-to-integer mapping and implementing functions to set, remove, and list balance entries.



```

// SPDX-License-Identifier: MIT
pragma solidity >0.8.3;

contract Mapping {
    // Mapping from address to uint
    //TEJAS GUNDAL D20A 24
    mapping(address => uint) public balances;

    function get(address _addr) public view returns (uint) {
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return balances[_addr];
    }

    function set(address _addr) public {
        // Update the value at this address
        balances[_addr] = _addr.balance;
    }

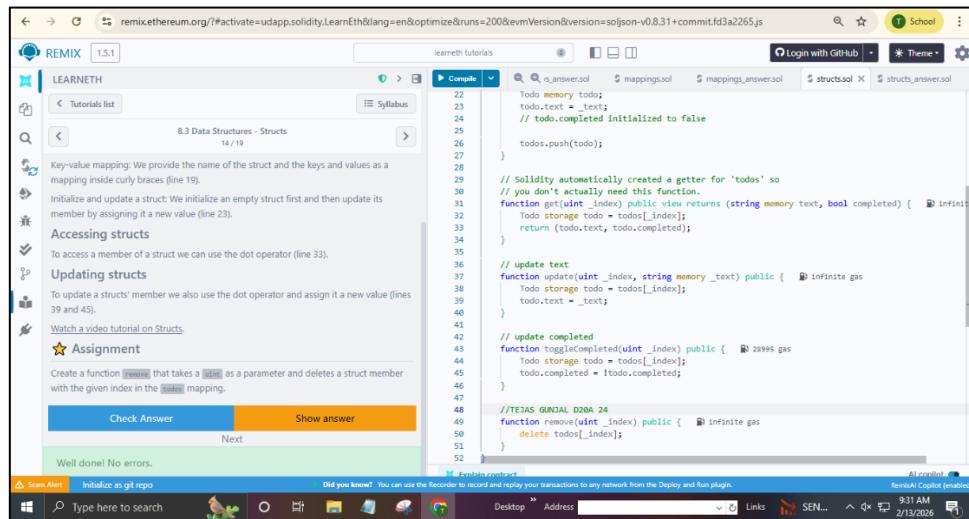
    function remove(address _addr) public {
        // Reset the value to the default value.
        delete balances[_addr];
    }
}

contract NestedMapping {
    // Nested mapping (mapping from address to another mapping)
    mapping(address => mapping(uint => bool)) public nested;

    function get(address _addr, uint _i) public view returns (bool) {
        // You can get values from a nested mapping
    }
}

```

- Tutorial 14: Creating a function to delete a struct member from a mapping using its index.



```

// SPDX-License-Identifier: MIT
pragma solidity >0.8.3;

contract Structs {
    struct Todo {
        string text;
        bool completed;
    }

    mapping(uint => Todo) todos;

    function addTodo(string memory _text) public {
        Todo memory todo;
        todo.text = _text;
        todo.completed = false;
        todos.push(todo);
    }

    // Solidity automatically created a getter for 'todos' so
    // you don't need to write that function.
    function getTodo(uint _index) public view returns (string memory text, bool completed) {
        Todo storage todo = todos[_index];
        return (todo.text, todo.completed);
    }

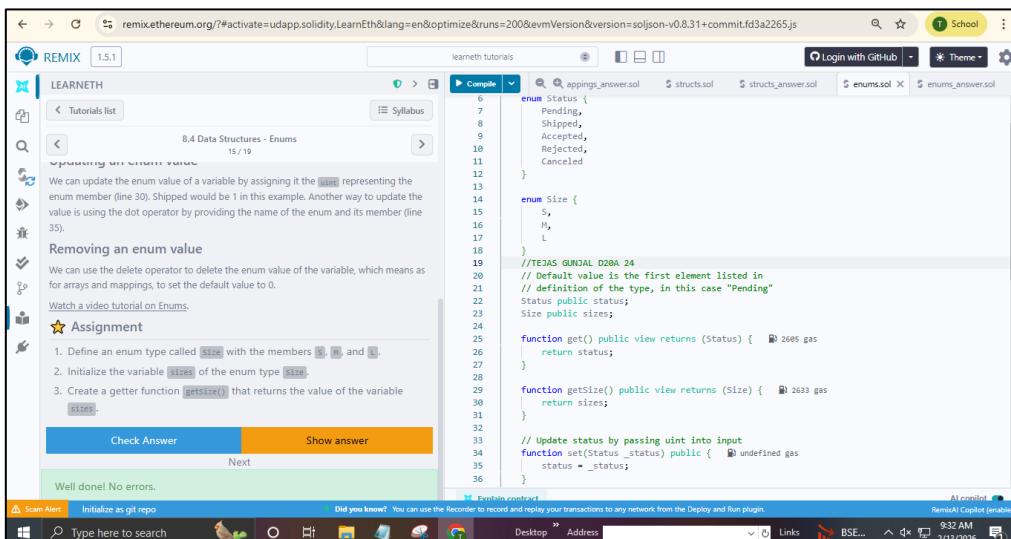
    // update text
    function update(uint _index, string memory _text) public {
        Todo storage todo = todos[_index];
        todo.text = _text;
    }

    // update completed
    function toggleCompleted(uint _index) public {
        Todo storage todo = todos[_index];
        todo.completed = !todo.completed;
    }

    //TEJAS GUNDAL D20A 24
    function remove(uint _index) public {
        delete todos[_index];
    }
}

```

- Tutorial 15: Defining an enum type, initializing a variable, and creating a getter function.



```

// SPDX-License-Identifier: MIT
pragma solidity >0.8.3;

enum Status {
    Pending,
    Shipped,
    Accepted,
    Rejected,
    Canceled
}

enum Size {
    S,
    M,
    L
}

//TEJAS GUNDAL D20A 24
// Default value is the first element listed in
// definition of the type, in this case "Pending"
Status public status;
Size public sizes;

function get() public view returns (Status) {
    return status;
}

function getSize() public view returns (Size) {
    return sizes;
}

// Update status by passing uint into input
function setStatus(uint _status) public {
    status = _status;
}

```

- Tutorial 16: Modifying struct values in different data locations and returning them from a function.

```

4 contract DataLocations {
5     uint[] public arr;
6     mapping(uint => address) map;
7     struct MyStruct {
8         uint foo;
9     }
10    mapping(uint => MyStruct) public myStructs;
11    //TEJAS GUNDAL D20A 24
12    function f() public returns (MyStruct memory, MyStruct memory, MyStruct memory){
13        // call _f with state variables
14        _f(arr, map, myStructs[1]);
15        // get a struct from a mapping
16        MyStruct storage myStruct = myStructs[1];
17        myStruct.foo = 4;
18        // create a struct in memory
19        MyStruct memory myMemStruct = MyStruct(0);
20        MyStruct memory myMemStruct2 = myMemStruct;
21        myMemStruct2.foo = 1;
22
23        MyStruct memory myMemStruct3 = myStruct;
24        myMemStruct3.foo = 3;
25        return (myStruct, myMemStruct2, myMemStruct3);
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    }
}

```

- Tutorial 17: Creating variables to represent 1 wei and comparing 1 gwei to 10^{18} .

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4
5 //TEJAS GUNDAL D20A 24
6 contract EtherUnits {
7     uint public oneWei = 1 wei;
8     // 1 wei is equal to 1
9     bool public isOneWei = 1 wei == 1;
10
11    uint public oneEther = 1 ether;
12    // 1 ether is equal to 10^18 wei
13    bool public isOneEther = 1 ether == 1e18;
14
15    uint public oneGwei = 1 gwei;
16    // 1 ether is equal to 10^9 wei
17    bool public isOneGwei = 1 gwei == 1e9;
18 }

```

➤ Tutorial 18: Creating a variable to store the gas cost of deploying the contract.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 //TEJAS GUNJAL D20A 24
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 your transaction to fail.
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 spent
14        // and the transaction fails
15        while (true) {
16            i += 1;
17        }
18    }
19 }

```

➤ Tutorial 19: Creating a charity contract that accepts donations and allows the owner to withdraw the balance.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 //TEJAS GUNJAL D20A 24
5 contract Charity {
6     address payable owner;
7
8     constructor() {
9         owner = msg.sender;
10    }
11
12    function donate() public payable {
13        require(msg.value > 0);
14    }
15
16    function withdraw() public {
17        require(owner != msg.sender);
18        owner.transfer(address(this).balance);
19    }
20 }

```

Conclusion:

In this practical, we explored the fundamental concepts of Solidity programming required to create smart contracts on the Ethereum blockchain. We understood primitive data types, variables, function types such as pure and view, visibility rules, modifiers, constructors, control flow statements, data structures, data locations, and transaction-related concepts like Ether, Wei, and gas.

By learning these core features, we gained a clear understanding of how smart contracts are written, deployed, and executed securely on the blockchain. These concepts form the foundation for building efficient, transparent, and decentralized applications.