Shreyash G Dhekane D20A Roll No.23

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.

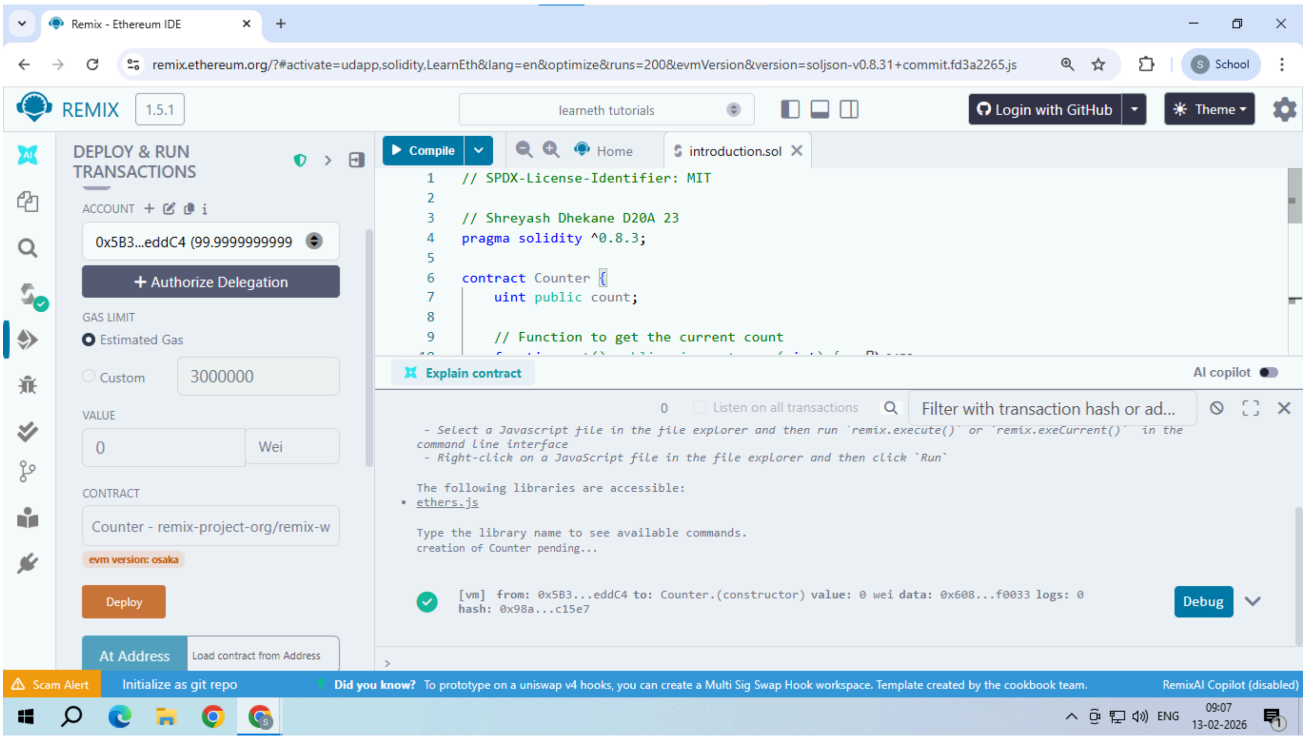
Pro Tip: Changing a variable from storage to memory creates a copy. Changing a pointer to storage affects the actual state variable on the blockchain.

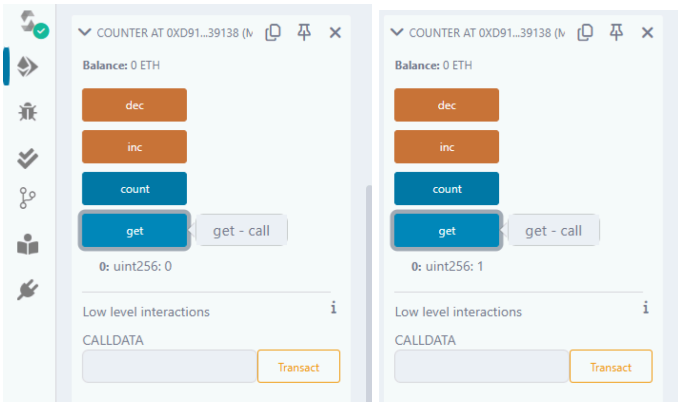
Q7: Transactions and Gas

- Wei: Just as a Dollar has Cents, an Ether has Wei.
  - 1 { Ether} = 1,000,000,000,000,000,000 { Wei} (10^{18}).
- Gas: Think of Gas as the "fuel" and Gas Price as the "price per gallon."
  - {Total Cost} = {Gas Used} \times {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:-

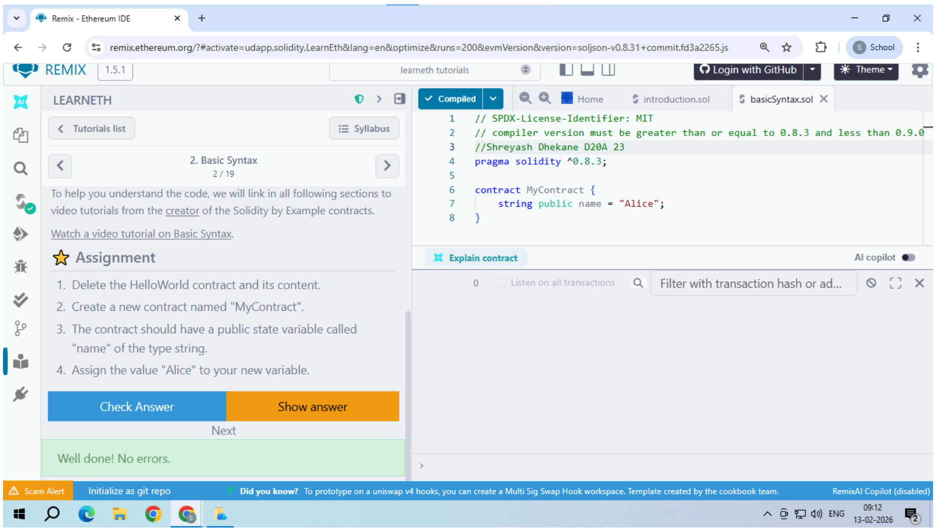
Assignment 1:



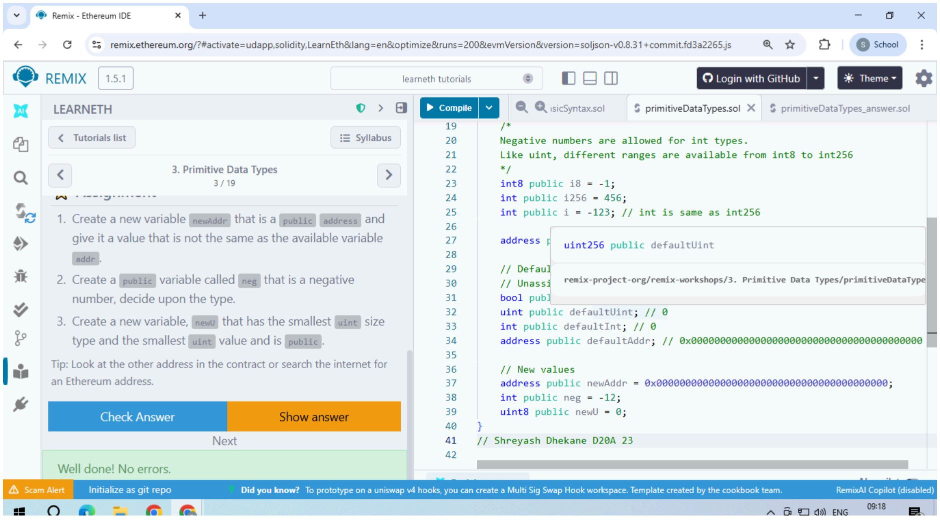


Shreyash G Dhekane D20A Roll No.23

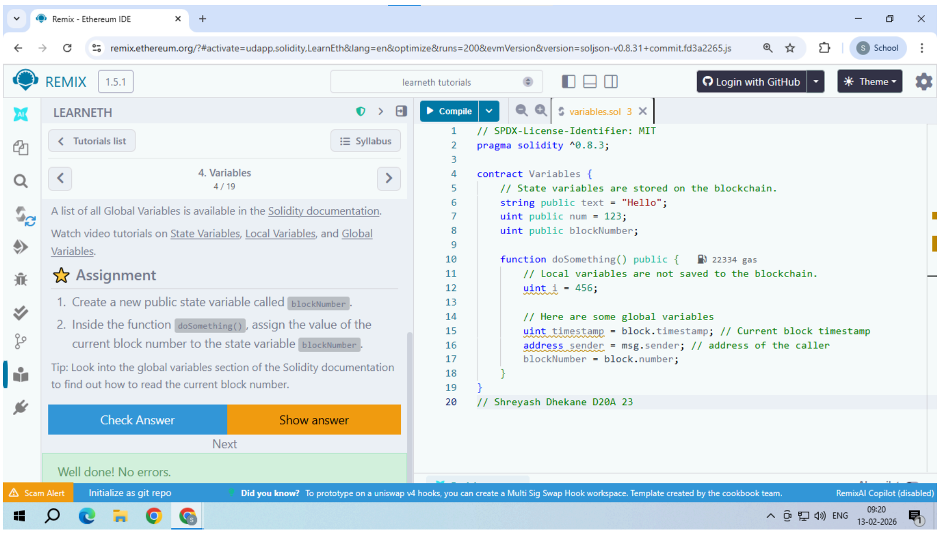
2: Basic Syntax



3: Primitive Data Types

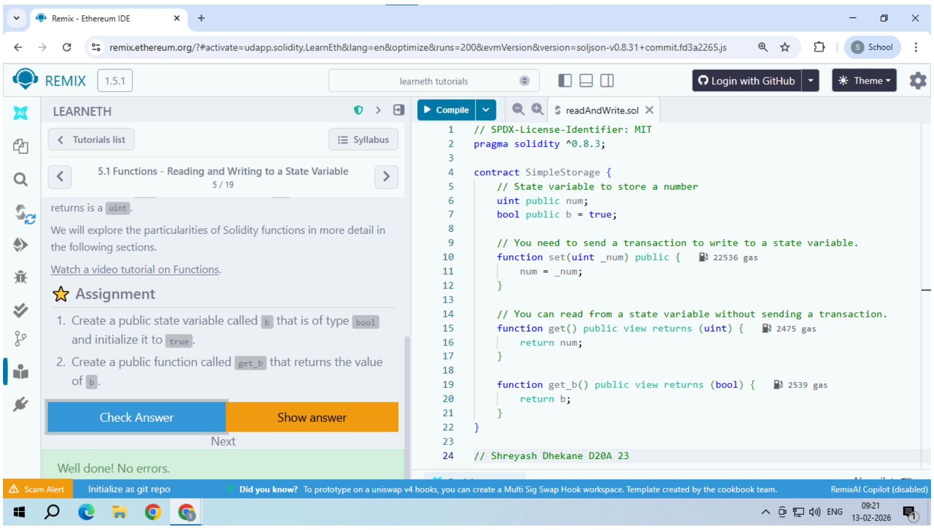


4: Variables

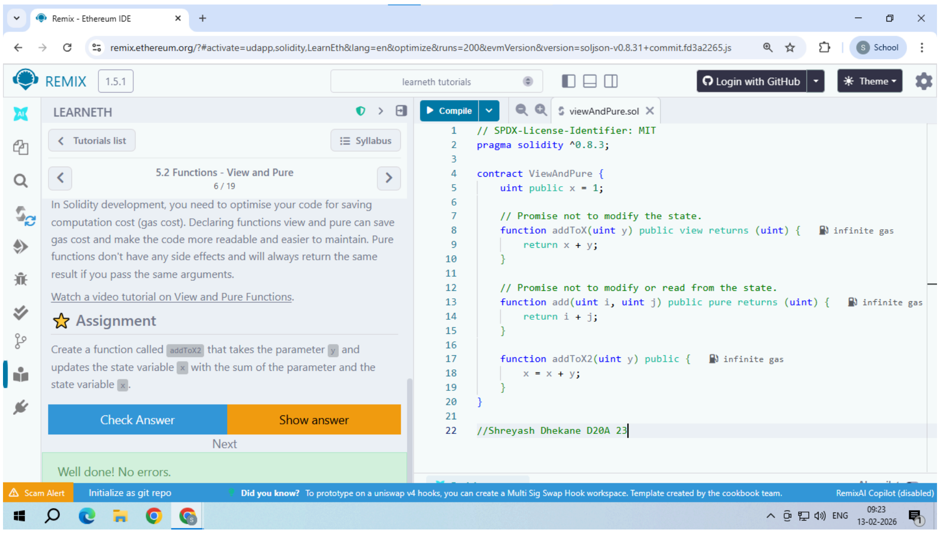


Shreyash G Dhekane D20A Roll No.23

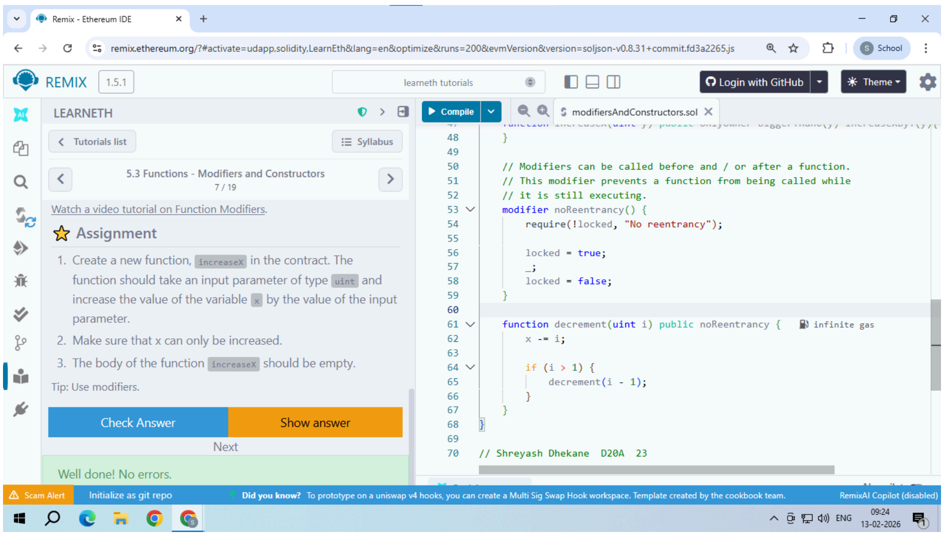
5: Functions



6: Functions

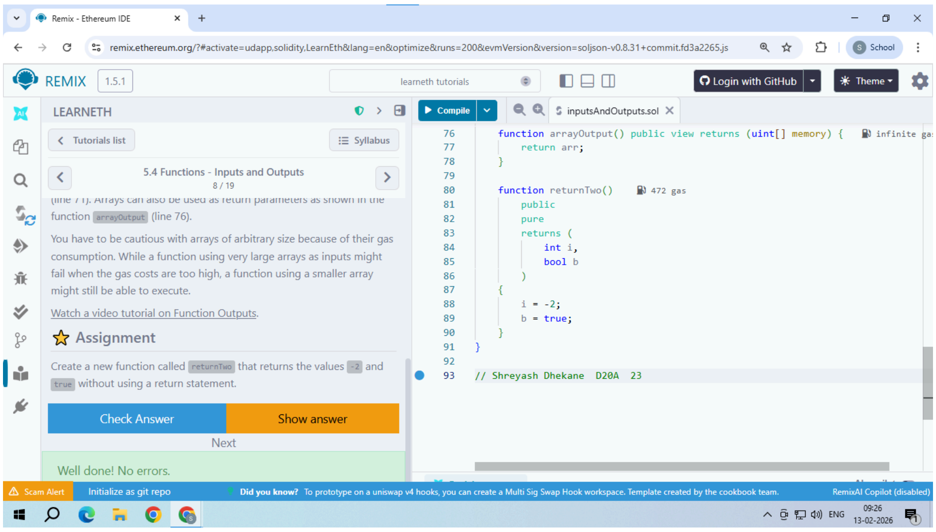


7: Functions

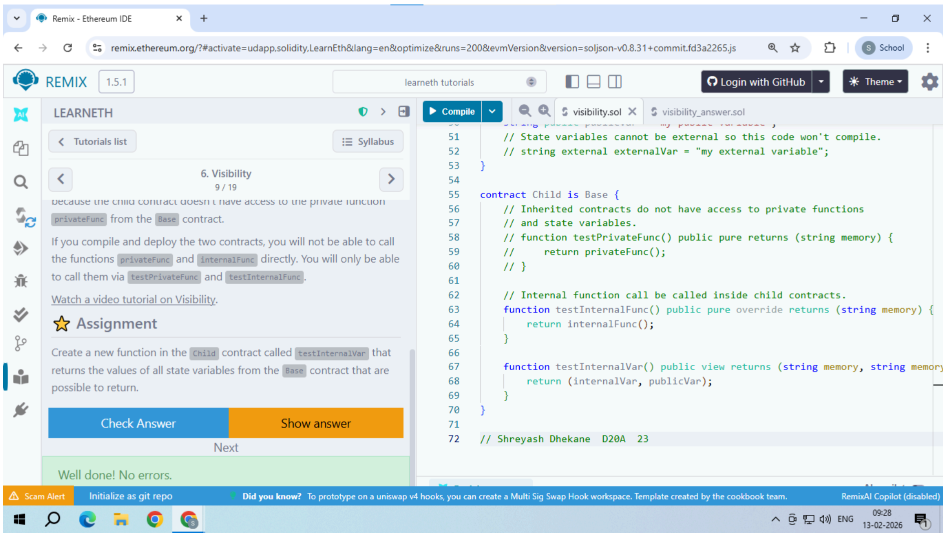


Shreyash G Dhekane D20A Roll No.23

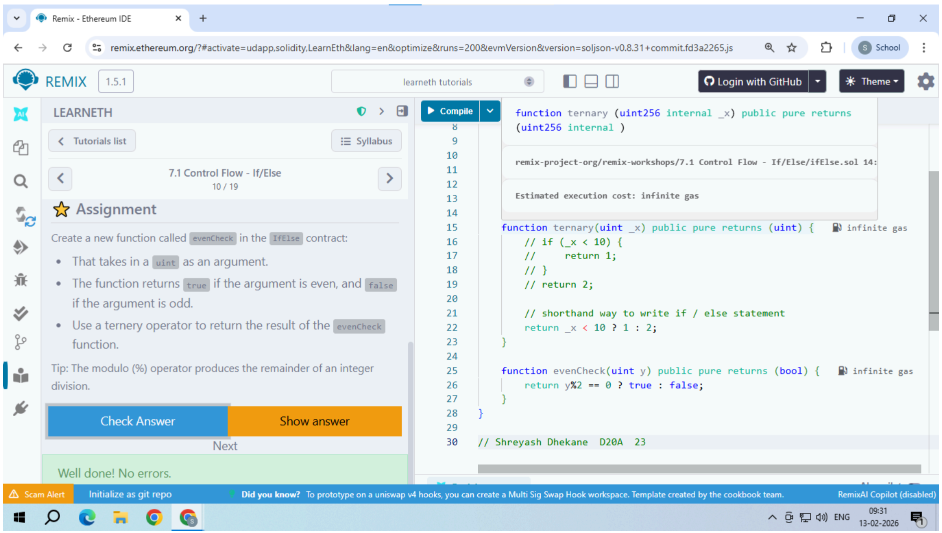
8: Functions



9: Visibility

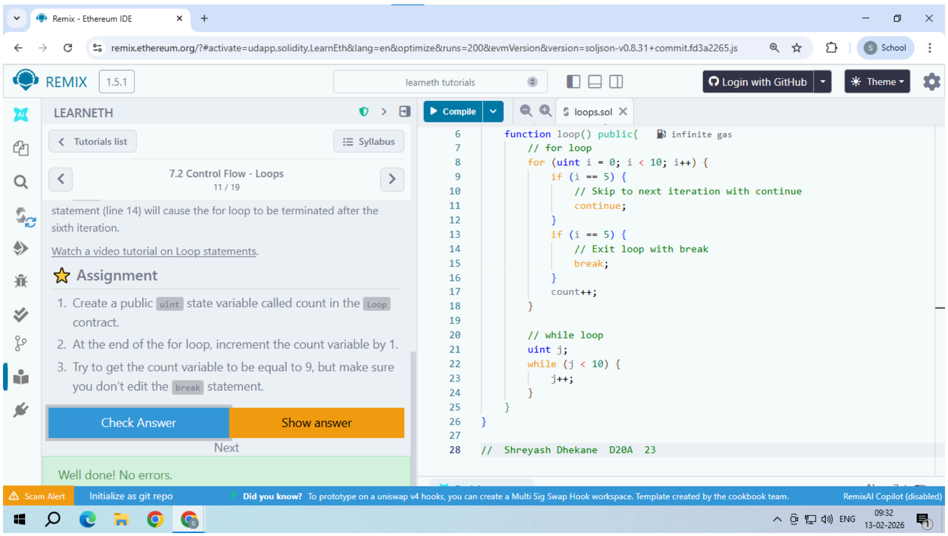


10: Control Flow

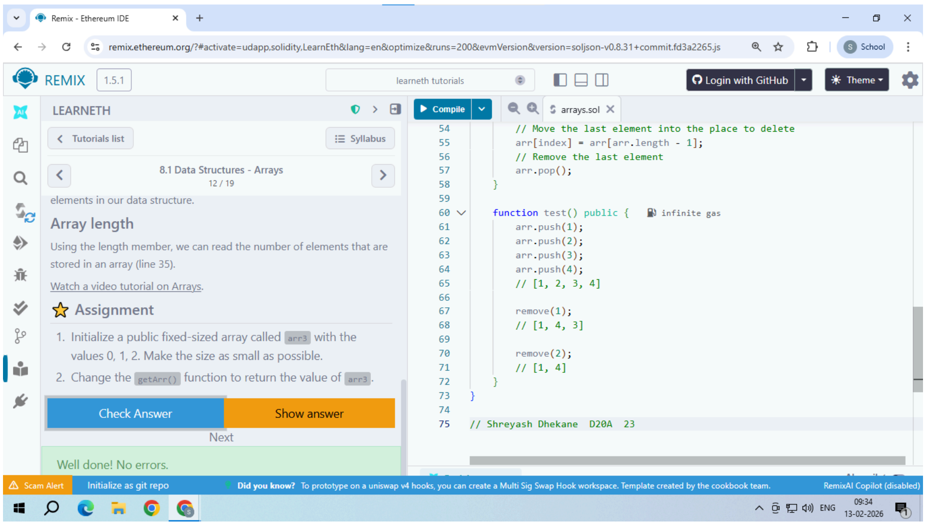


Shreyash G Dhekane D20A Roll No.23

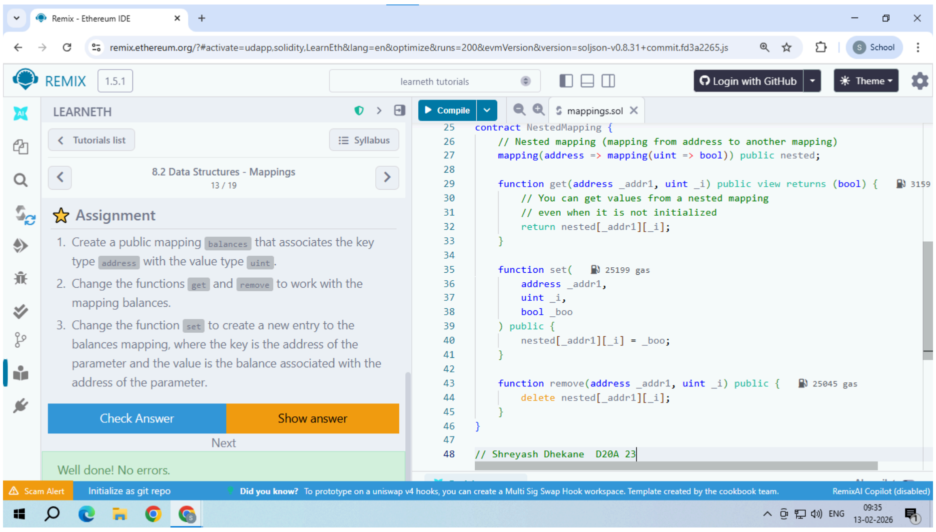
11: Control Flow



12: Data Structure - Arrays

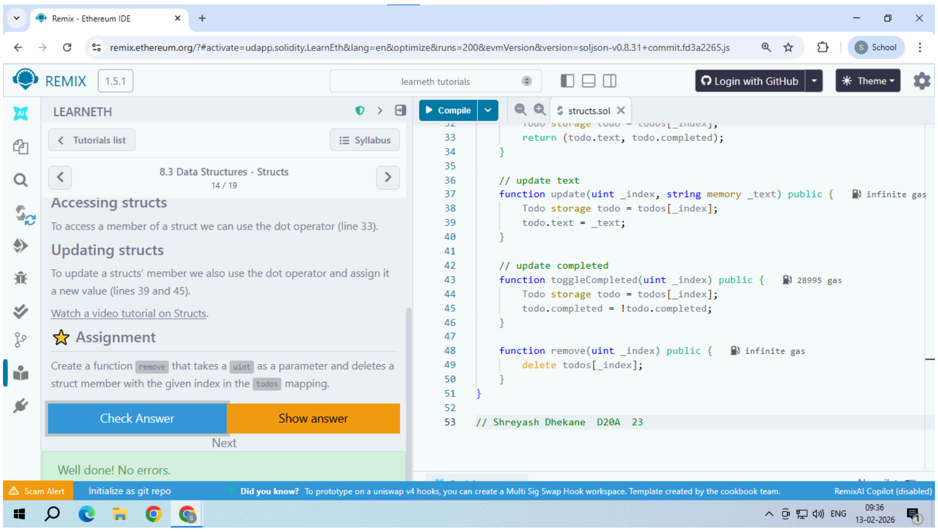


13: Data Structures - Mapping

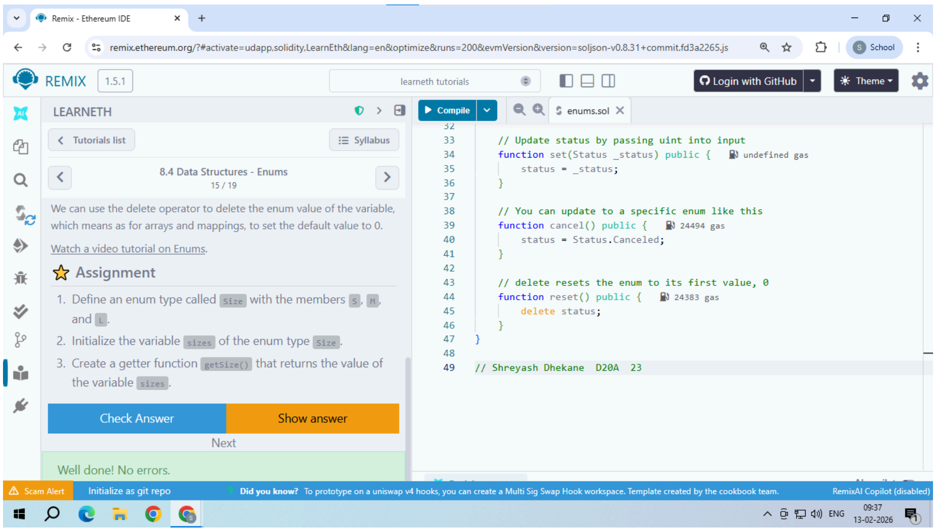


Shreyash G Dhekane D20A Roll No.23

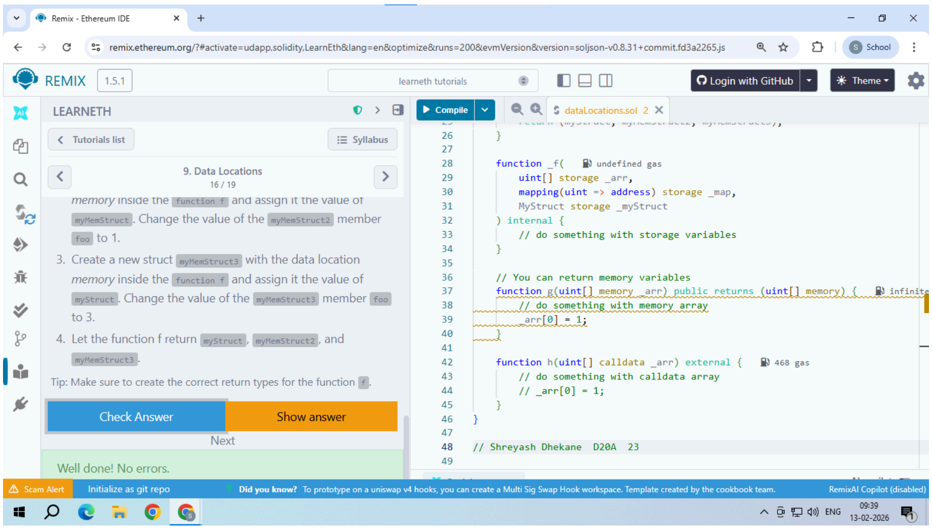
14: Data Structures - Structs



15: Data structures - Enums

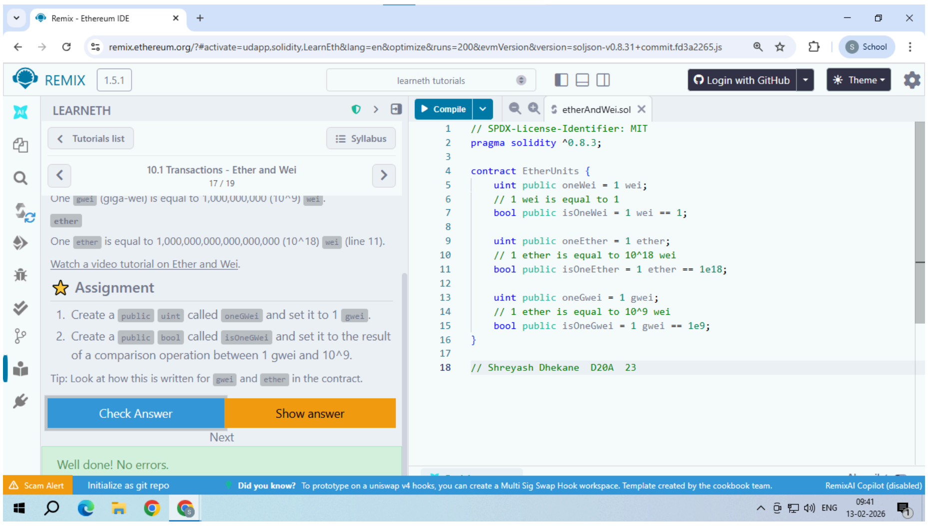


16: Data Locations

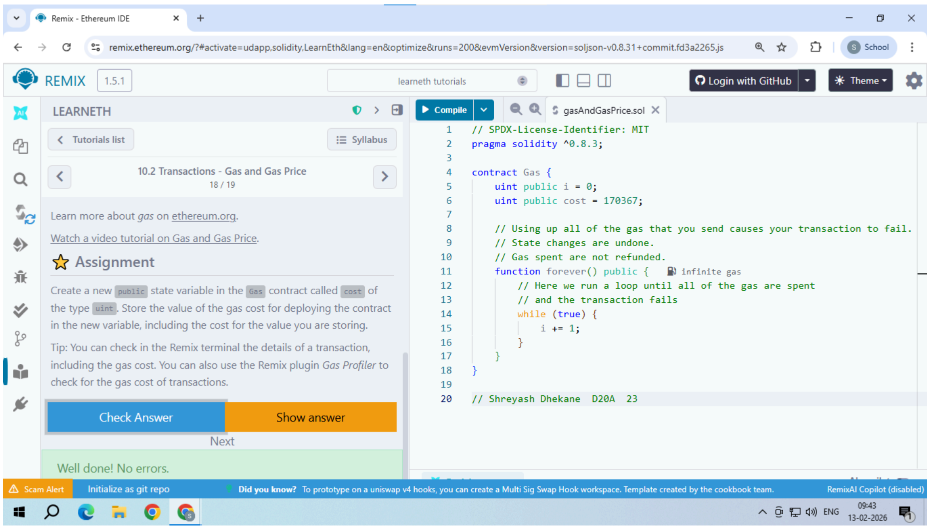


Shreyash G Dhekane D20A Roll No.23

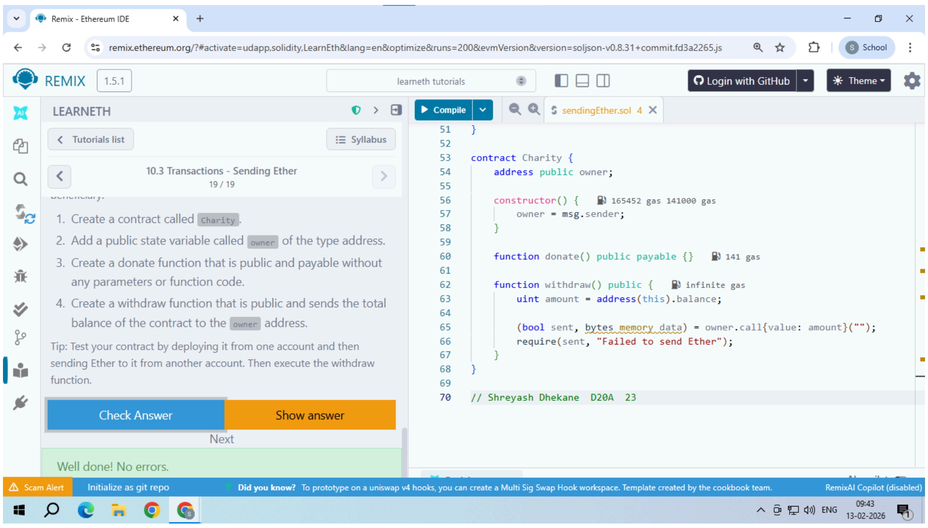
17: Transactions



18: Transactions



19: Transactions



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.