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.

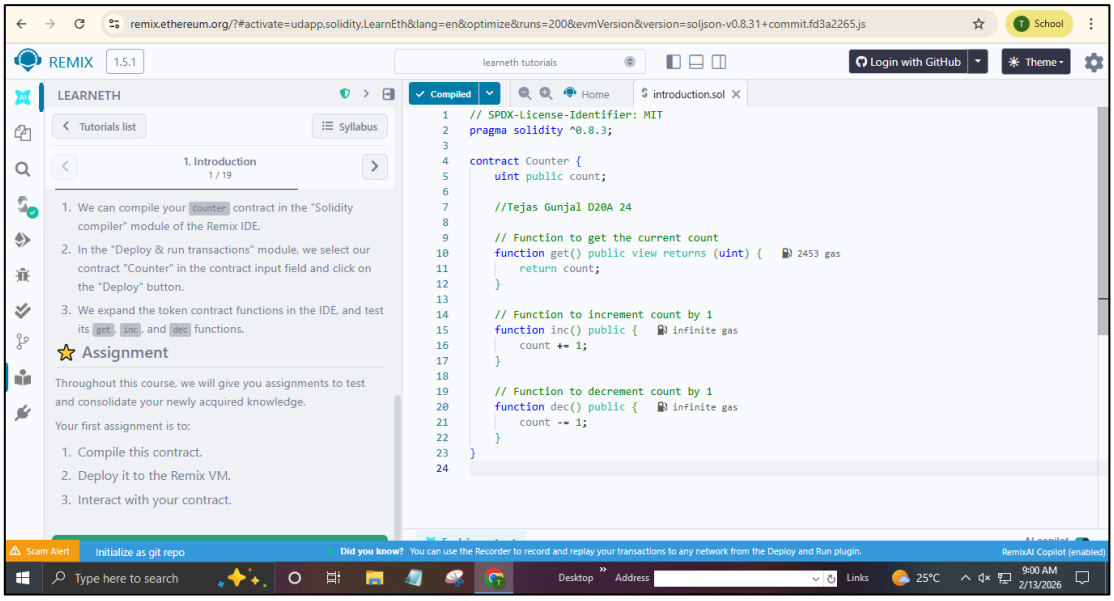
8. Transactions: Ether, Wei, Gas and Sending Transactions

- **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.

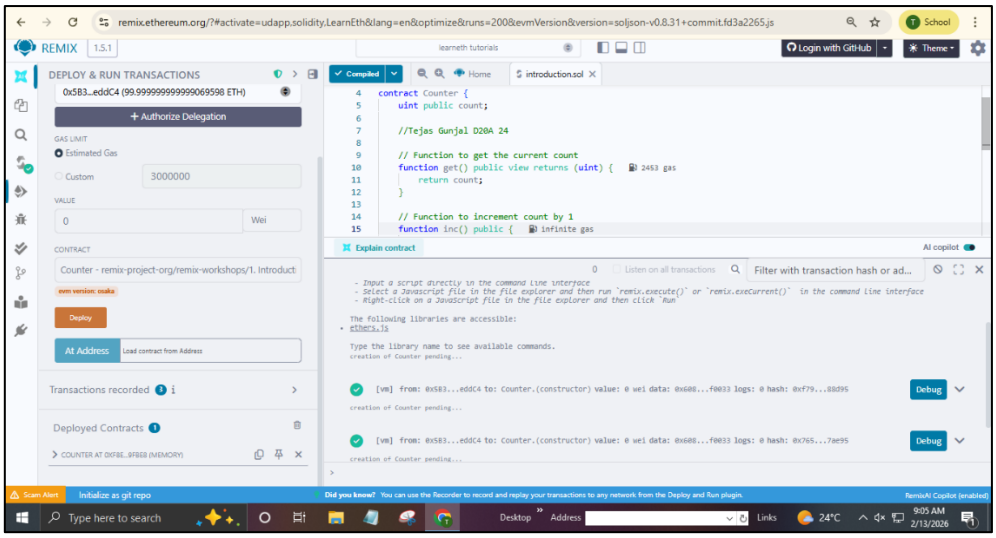
Implementation:

➢ Tutorial 1:

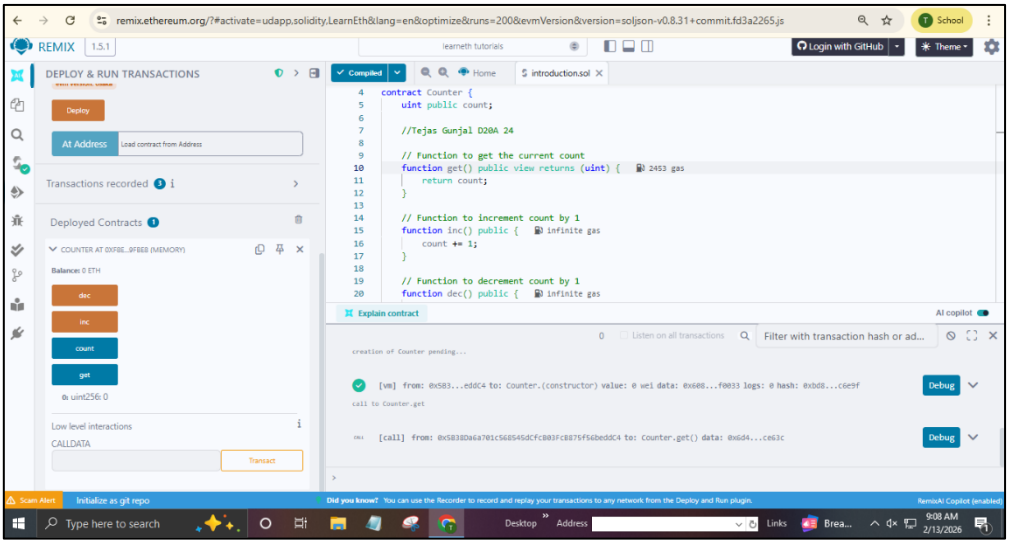
Tutorial No 1 – Compile the code



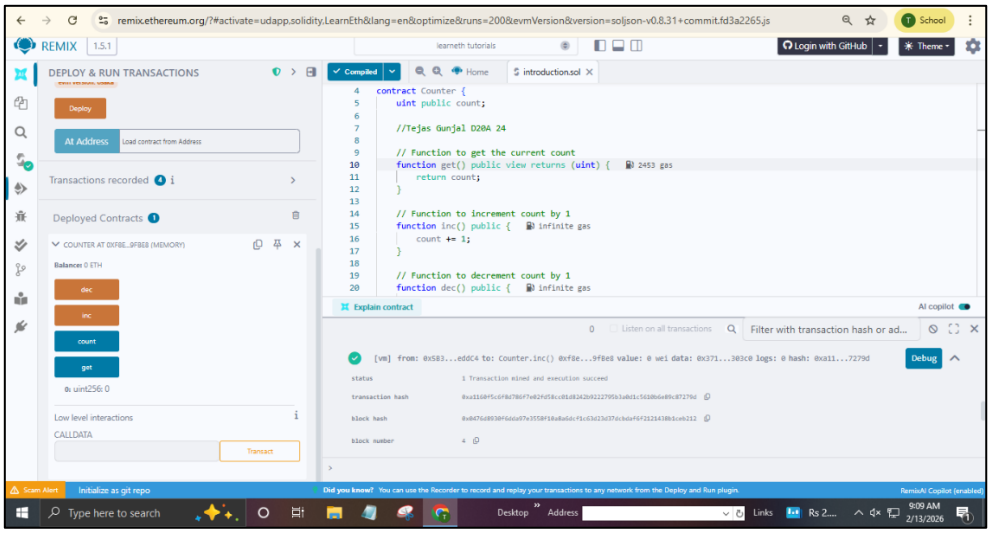
Deploy the contract



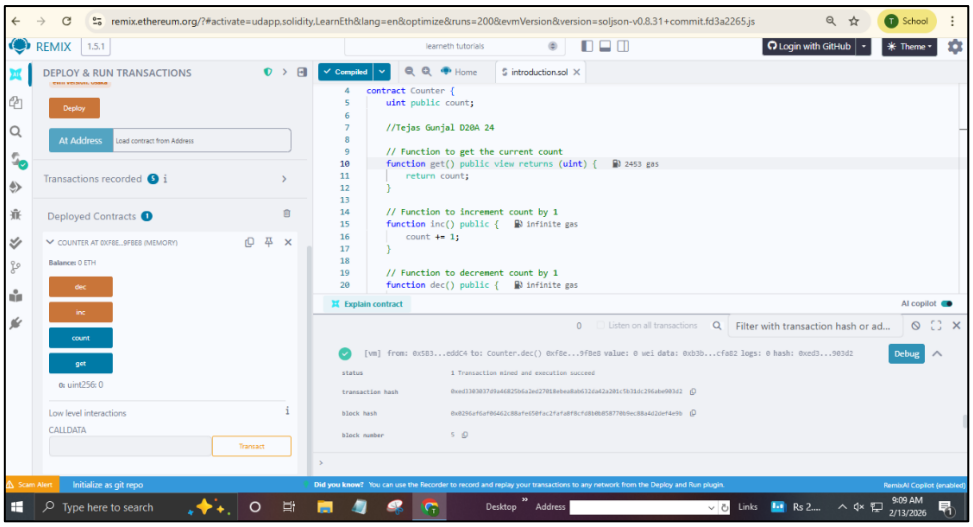
Get



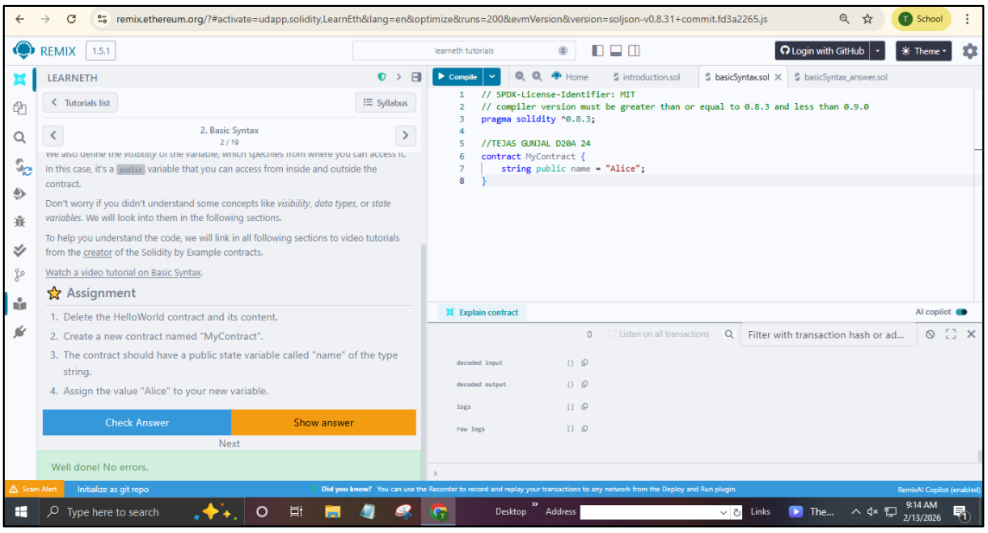
Increment



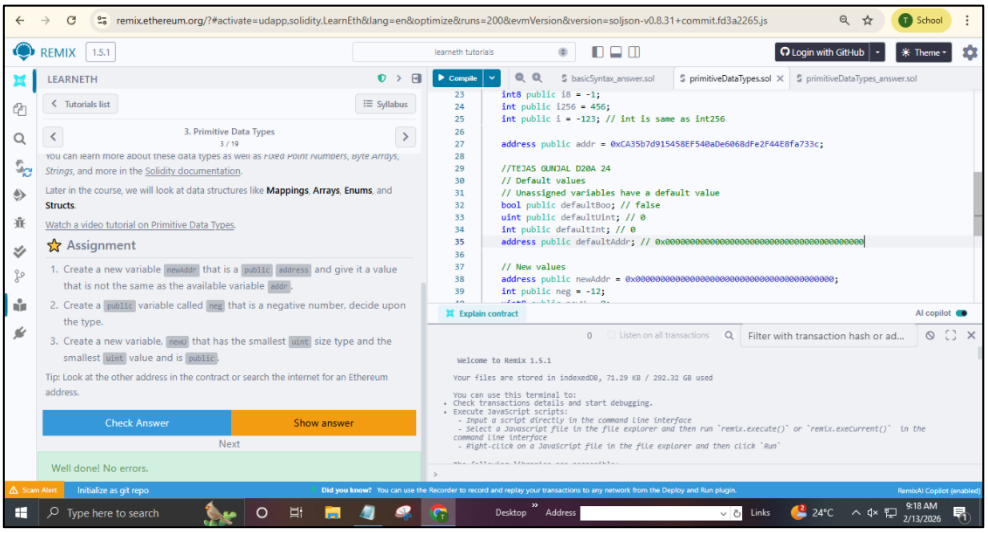
Decrement



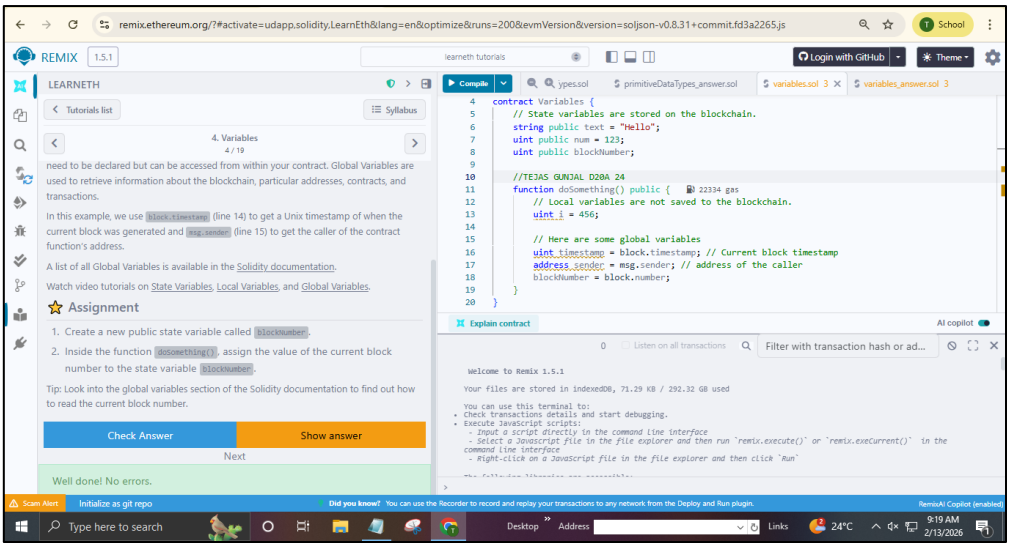
- Tutorial 2: Creating "MyContract" with a public string variable "name" set to "Alice".



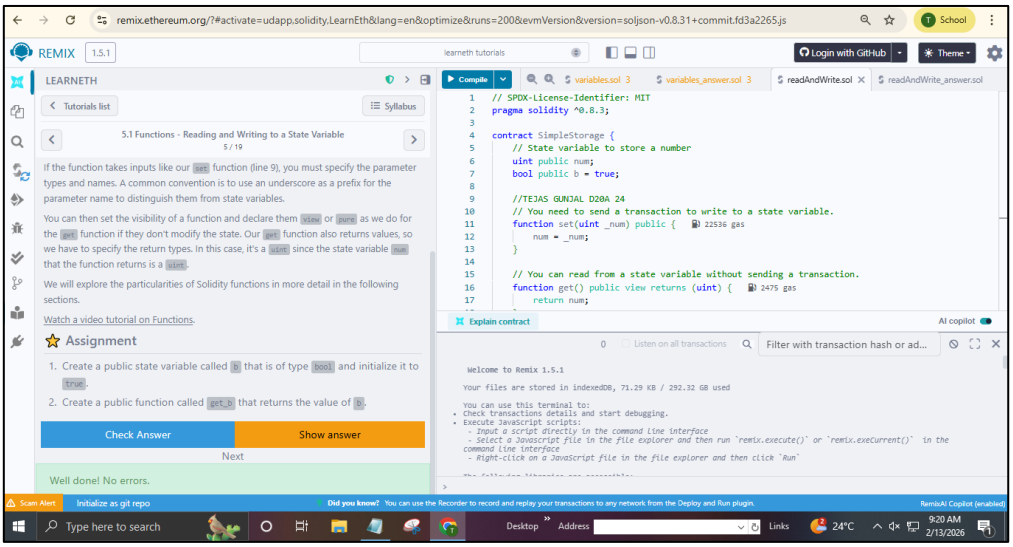
- Tutorial 3: Creating address and integer variables with specific values.



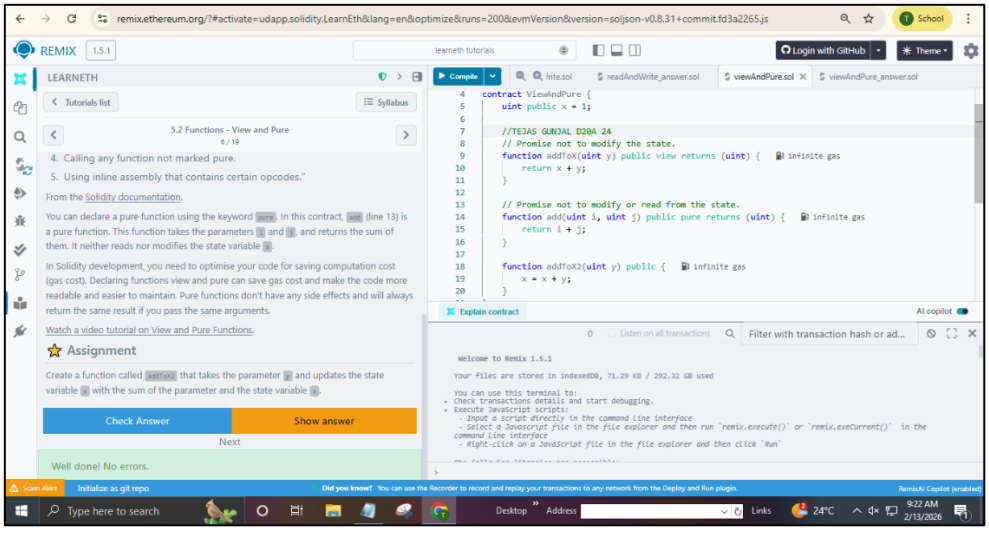
- Tutorial 4: Creating a state variable and assigning the current block number.



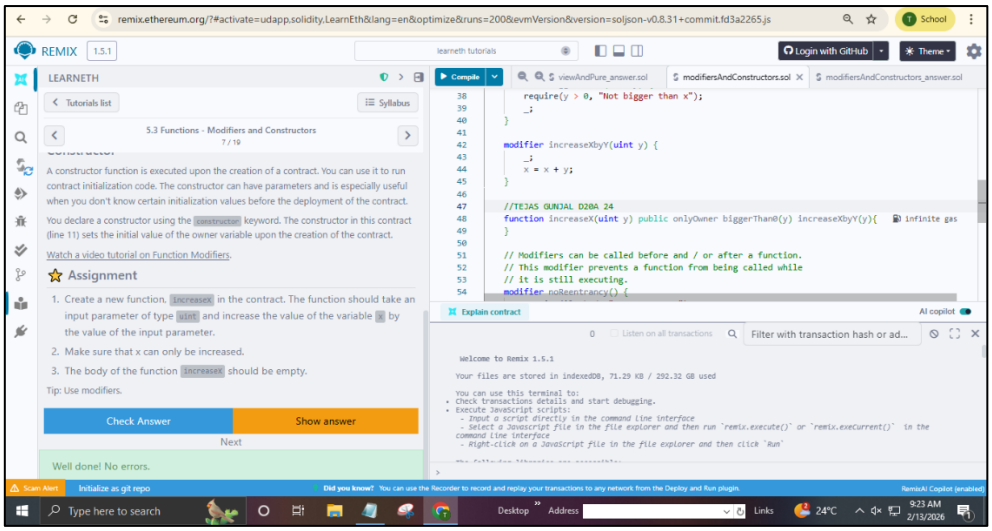
- Tutorial 5: Creating a boolean state variable and a function to return its value.



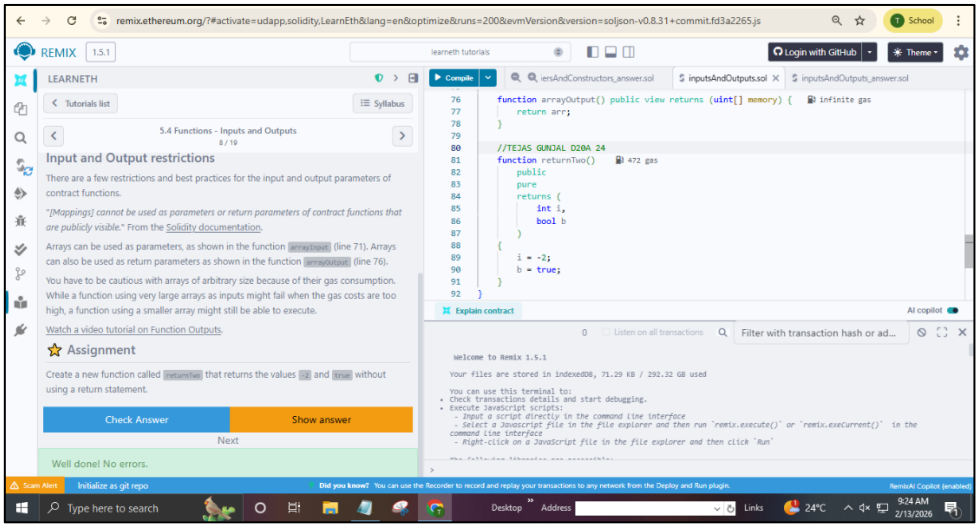
- Tutorial 6: Creating functions that read state variables without modifying them.



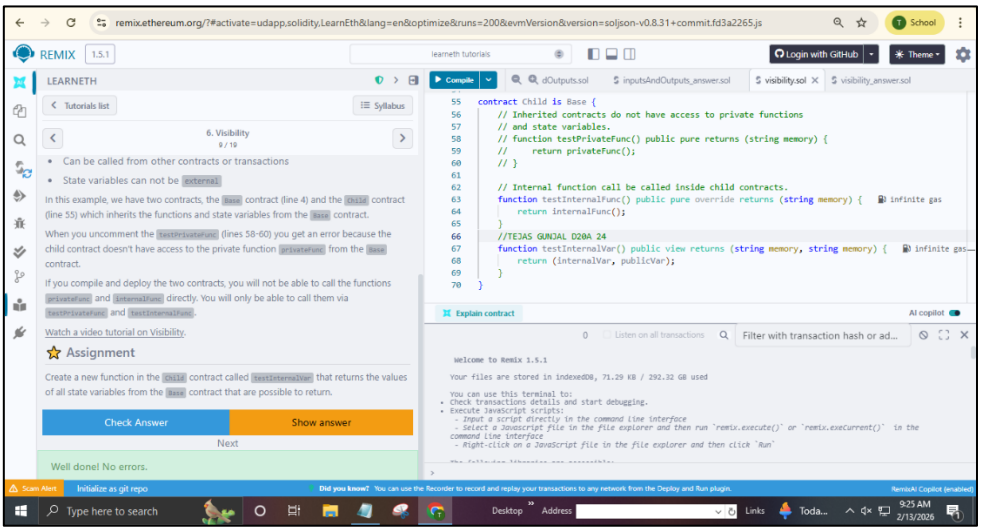
- Tutorial 7: Creating a function with a modifier to restrict state variable modifications.



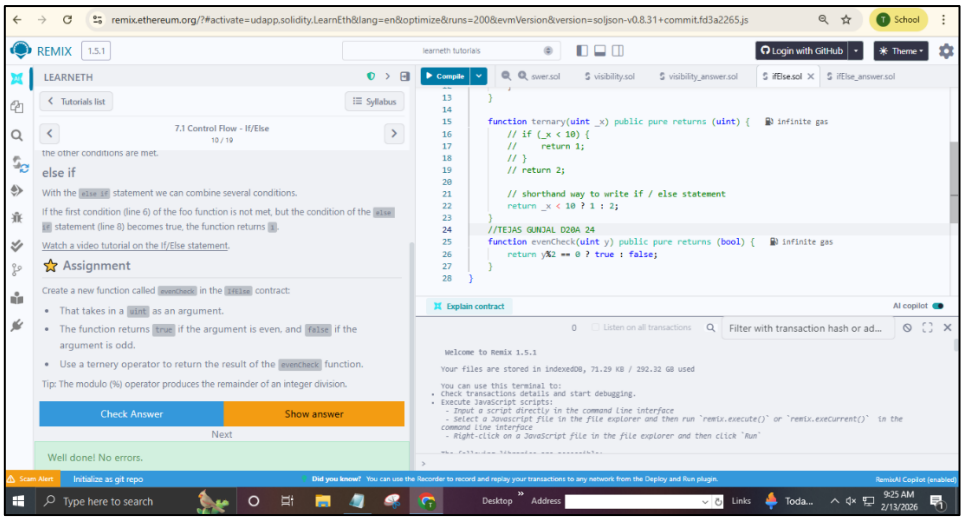
- Tutorial 8: Creating function that returns multiple values without using a return statement.



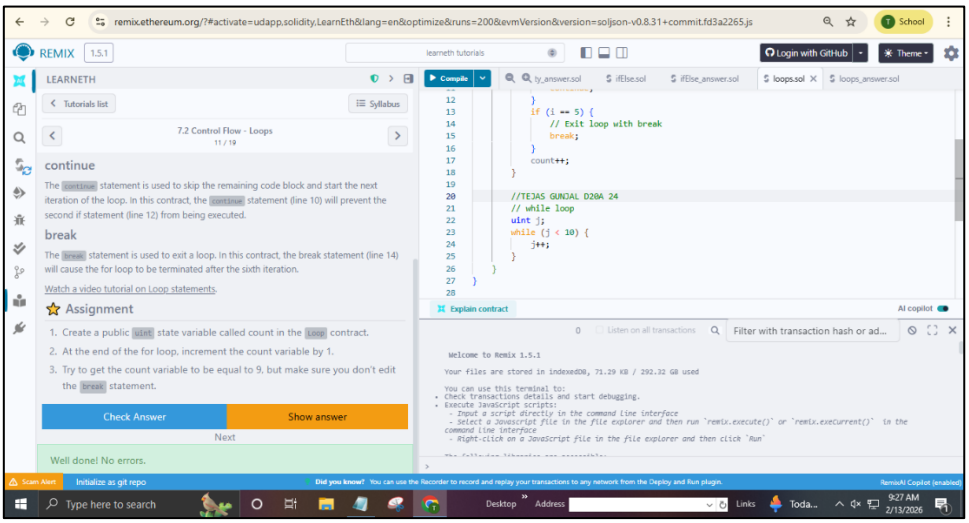
- Tutorial 9: Creating a function in a child contract to access visible state variables from the base contract.



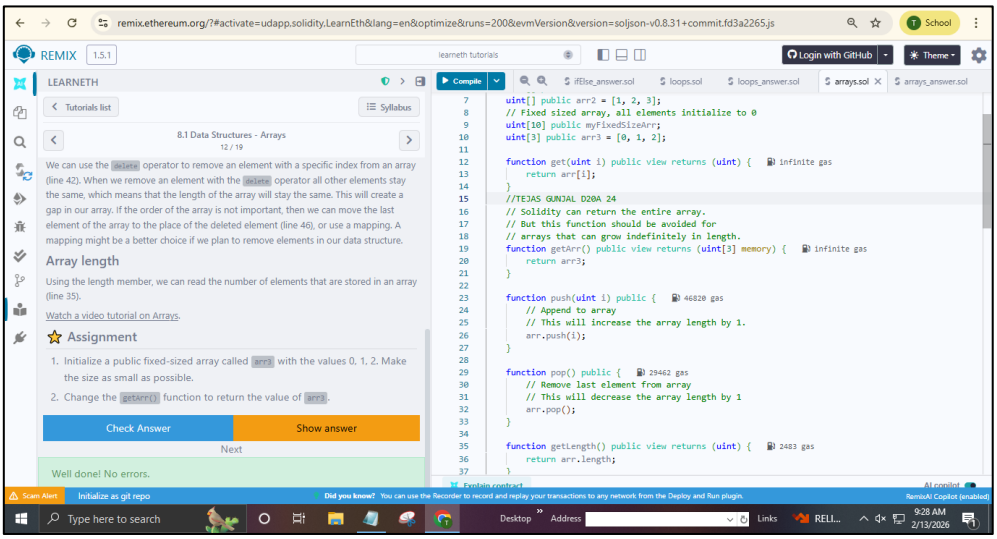
➢ Tutorial 10: Creating a function to check if a number is even using ternary operator.



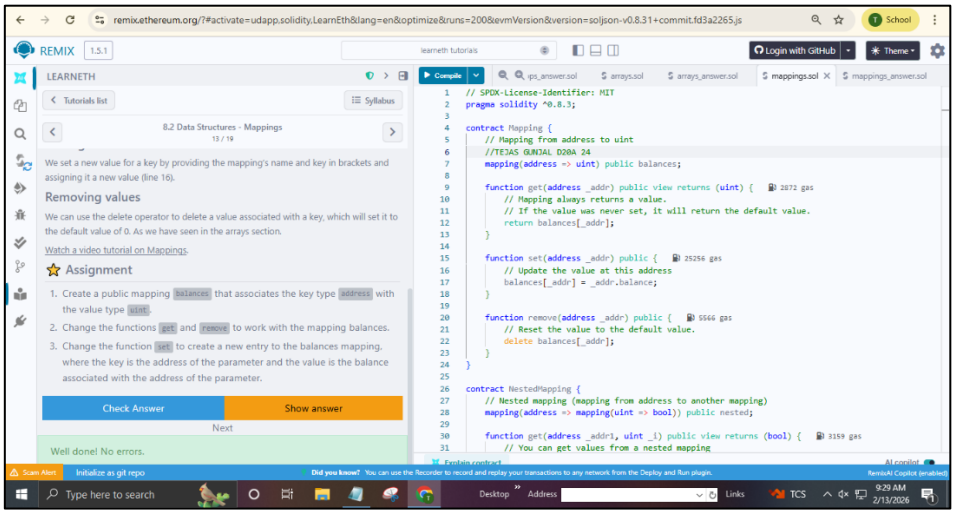
➢ Tutorial 11: Incrementing a counter in a for loop to reach 9 while preserving the break statement.



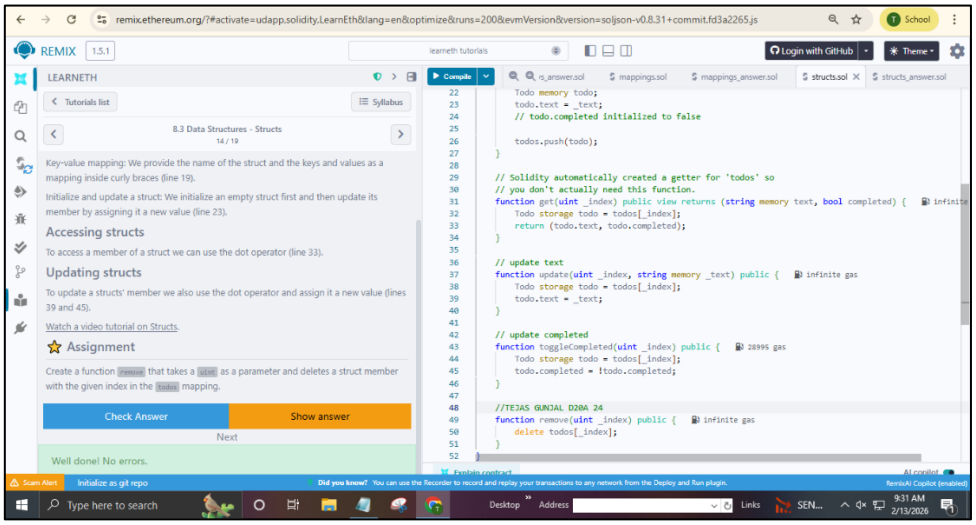
➢ Tutorial 12: Creating a fixed-size array and modifying a function to return its values.



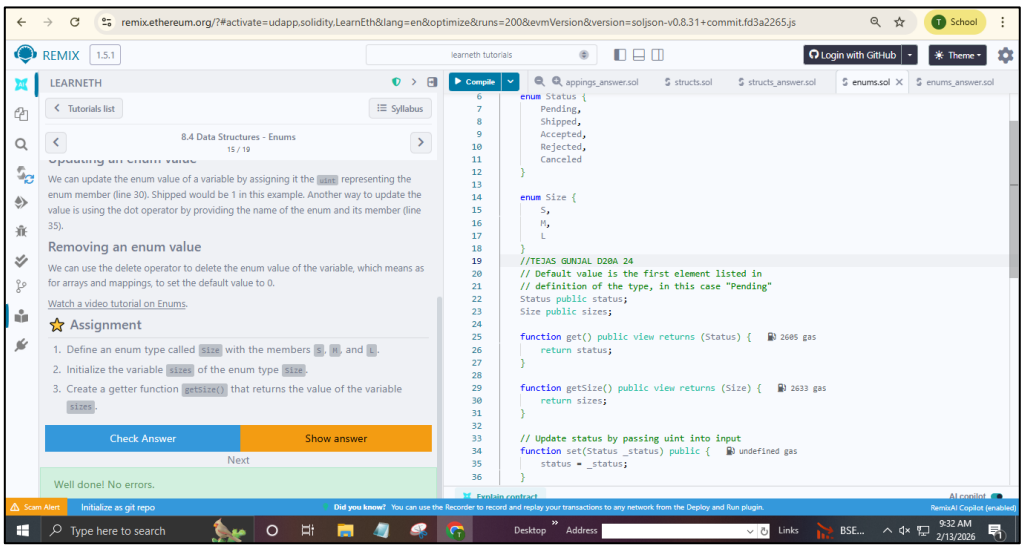
- Tutorial 13: Creating an address-to-integer mapping and implementing functions to set, remove, and list balance entries.



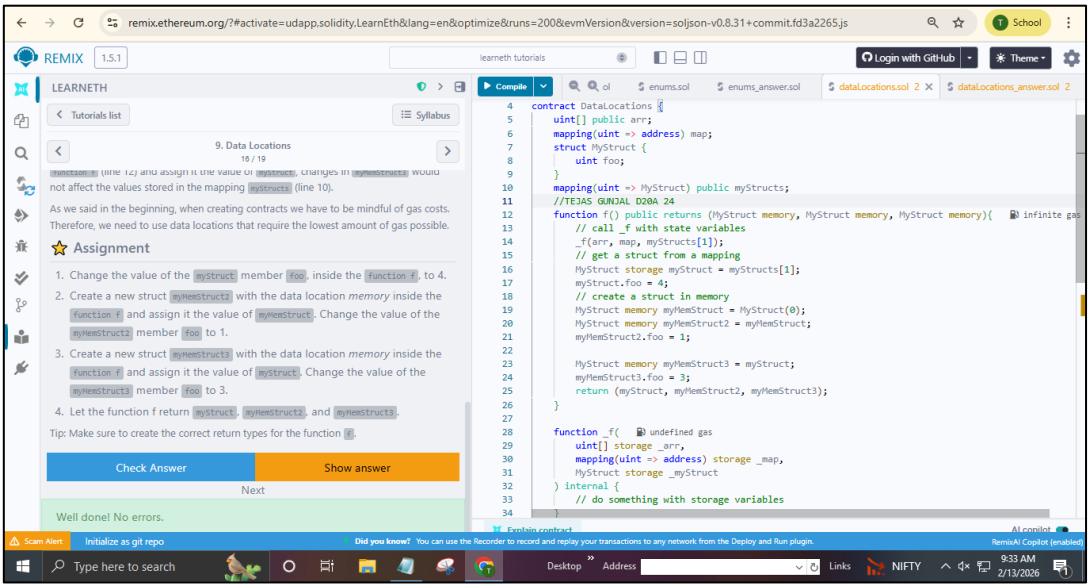
- Tutorial 14: Creating a function to delete a struct member from a mapping using its index.



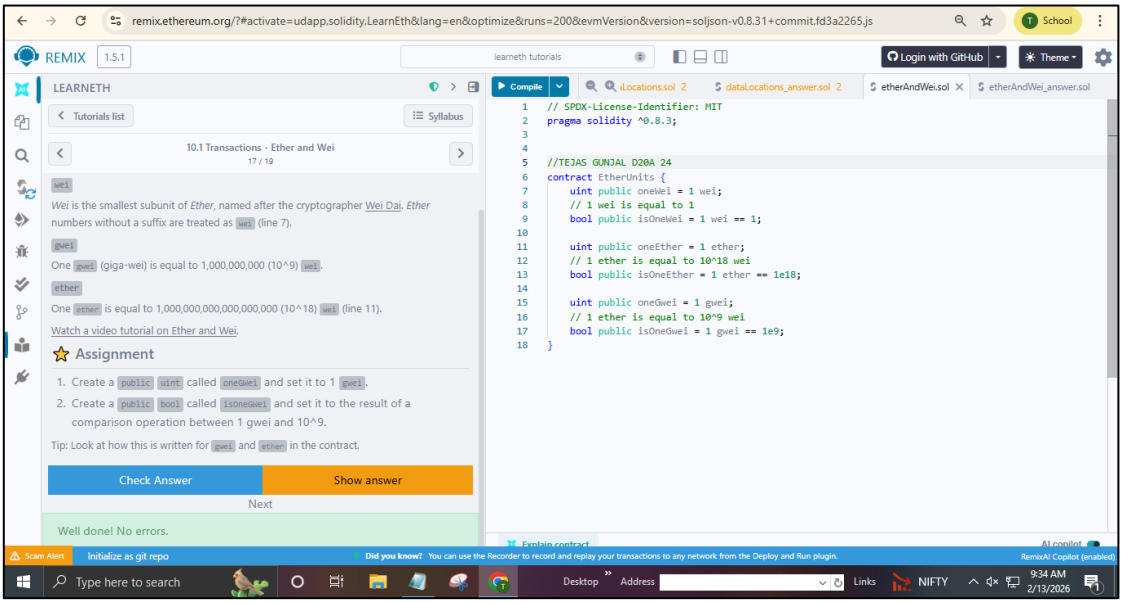
- Tutorial 15: Defining an enum type, initializing a variable, and creating a getter function.



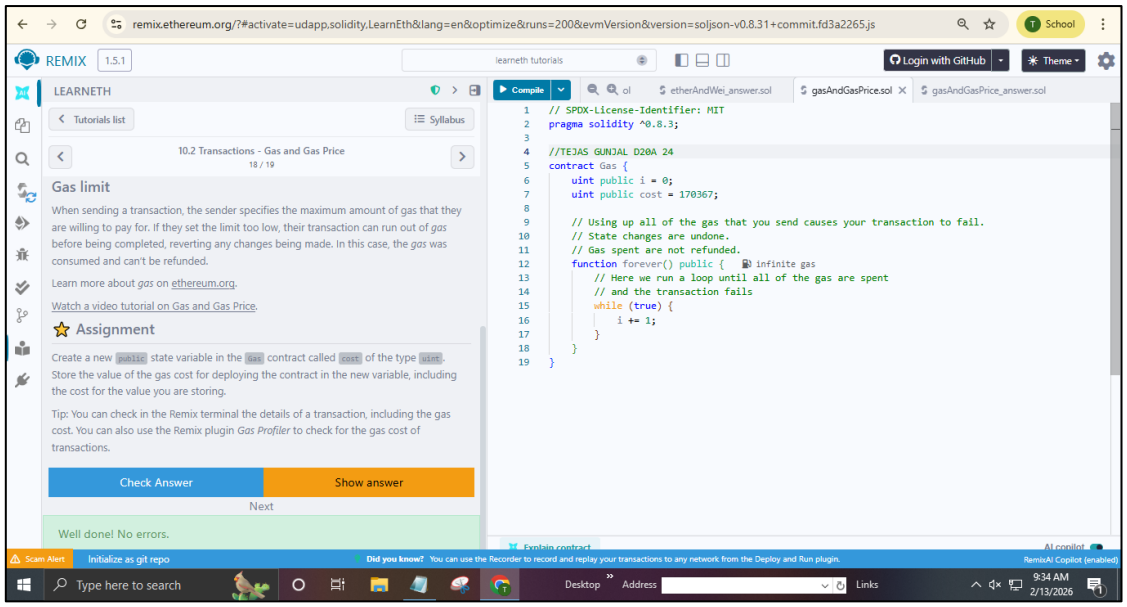
- Tutorial 16: Modifying struct values in different data locations and returning them from a function.



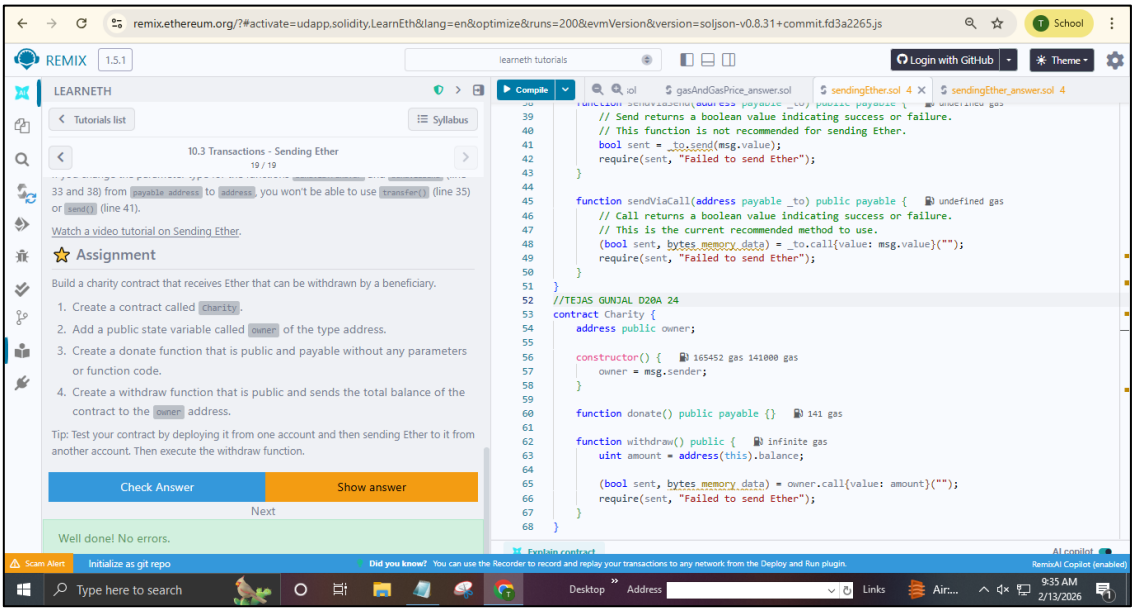
- Tutorial 17: Creating variables to represent 1 wei and comparing 1 gwei to 10^9.



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



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



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.