Aim: Hands on Solidity Programming Assignments for creating Smart Contracts

Theory:

1. **Primitive Data Types, Variables, Functions – pure, view**

In Solidity, primitive data types form the foundation of smart contract development. Commonly used types include:

- **uint / int**: unsigned and signed integers of different sizes (e.g., uint256, int128).
- **bool**: represents logical values (true or false).
- **address**: holds a 20-byte Ethereum account address, often used for storing user accounts or contract addresses.
- **bytes / string**: store binary data or textual data.

Variables in Solidity can be **state variables** (stored on the blockchain permanently), **local variables** (temporary, created during function execution), or **global variables** (special predefined variables such as msg.sender, msg.value, and block.timestamp).

Functions allow execution of contract logic. Special types of functions include:

- **pure**: cannot read or modify blockchain state; they work only with inputs and internal computations.
- **view**: can read state variables but cannot alter them. This classification helps optimize gas usage and enforces function integrity.

2. **Inputs and Outputs to Functions**

Functions in Solidity can accept input arguments and return one or more output values. Inputs enable users or other contracts to pass data into the contract, while outputs make it possible to return results after computation. For example, a function can accept an amount in Ether and return whether the transfer was successful. Solidity also allows named return variables, which improve readability and debugging.

3. **Visibility, Modifiers and Constructors**

- **Function Visibility** defines who can access a function:
  - public: available both inside and outside the contract.
  - private: only accessible within the same contract.
  - internal: accessible within the contract and its child contracts.
  - external: can be called only by external accounts or other contract

- **Modifiers** are reusable code blocks that change the behavior of functions. They are often used for access control, such as restricting sensitive functions to the contract owner (onlyOwner).

- **Constructors** are special functions executed only once during contract deployment. They initialize important values, such as setting the deploying account as the owner of the contract.

4. **Control Flow: if-else, loops**

Control flow in Solidity is similar to traditional programming languages:

- **if-else** allows conditional decision-making in contract logic, e.g., checking if a balance is sufficient before transferring funds.
- **Loops** (for, while, do-while) enable repeated execution of code. For example, iterating through an array of users. However, loops must be used carefully, as excessive iterations increase gas consumption, potentially making the contract expensive to execute.

5. Data Structures: Arrays, Mappings, Structs, Enums

- **Arrays**: Can be fixed or dynamic and are used to store ordered lists of elements. Example: an array of addresses for registered users.
- **Mappings**: Key-value pairs that allow quick lookups. Example: mapping(address => uint) for storing balances. Unlike arrays, mappings do not support iteration.
- **Structs**: Allow grouping of related properties into a single data type, such as creating a struct Player {string name; uint score;}.
- **Enums**: Used to define a set of predefined constants, making code more readable. Example: enum Status { Pending, Active, Closed }.

6. Data Locations

Solidity uses three primary data locations for storing variables:

- **storage**: Data stored permanently on the blockchain. Examples: state variables.
- **memory**: Temporary data storage that exists only while a function is executing. Used for local variables and function inputs.
- **calldata**: A non-modifiable and non-persistent location used for external function parameters. It is gas-efficient compared to memory. Understanding data locations is essential, as they directly impact gas costs and performance.

7. Transactions: Ether and Wei, Gas and Gas Price, Sending Transactions

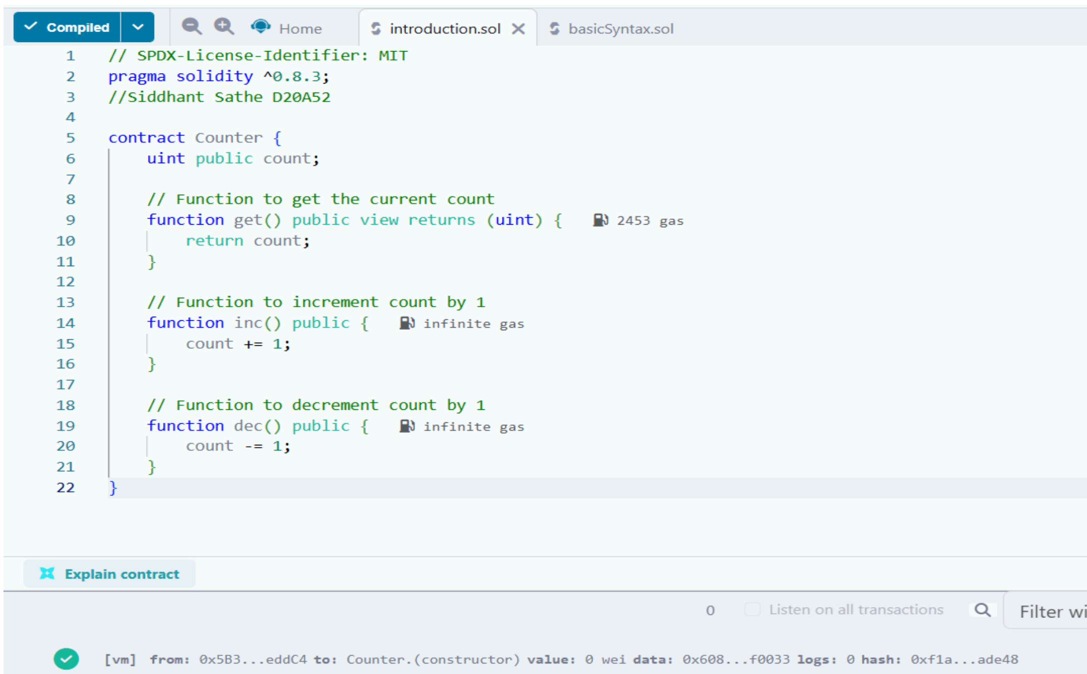
- **Ether and Wei**: Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.
- **Gas and Gas Price**: Every transaction consumes gas, which represents computational

effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.

- **Sending Transactions**: Transactions are used for transferring Ether or interacting with contracts. Functions like transfer() and send() are commonly used, while call() provides more flexibility. Each transaction requires gas, making efficiency in contract design very important.

Implementation:

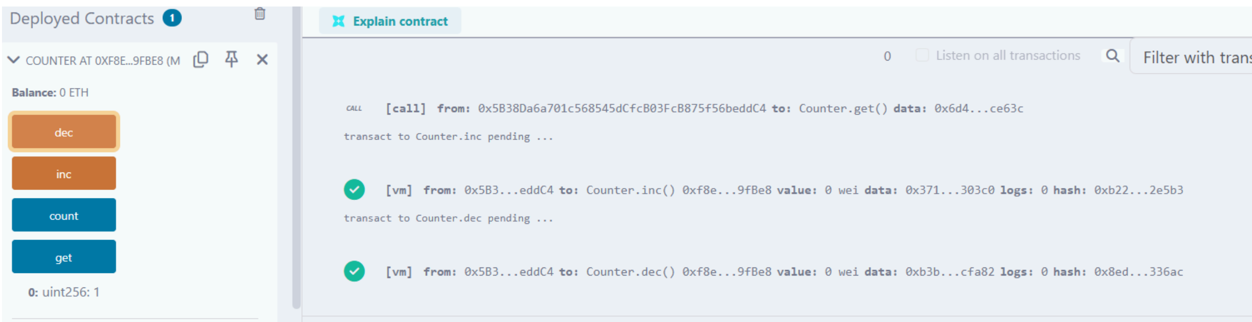
- Tutorial no. 1 – Compile the code



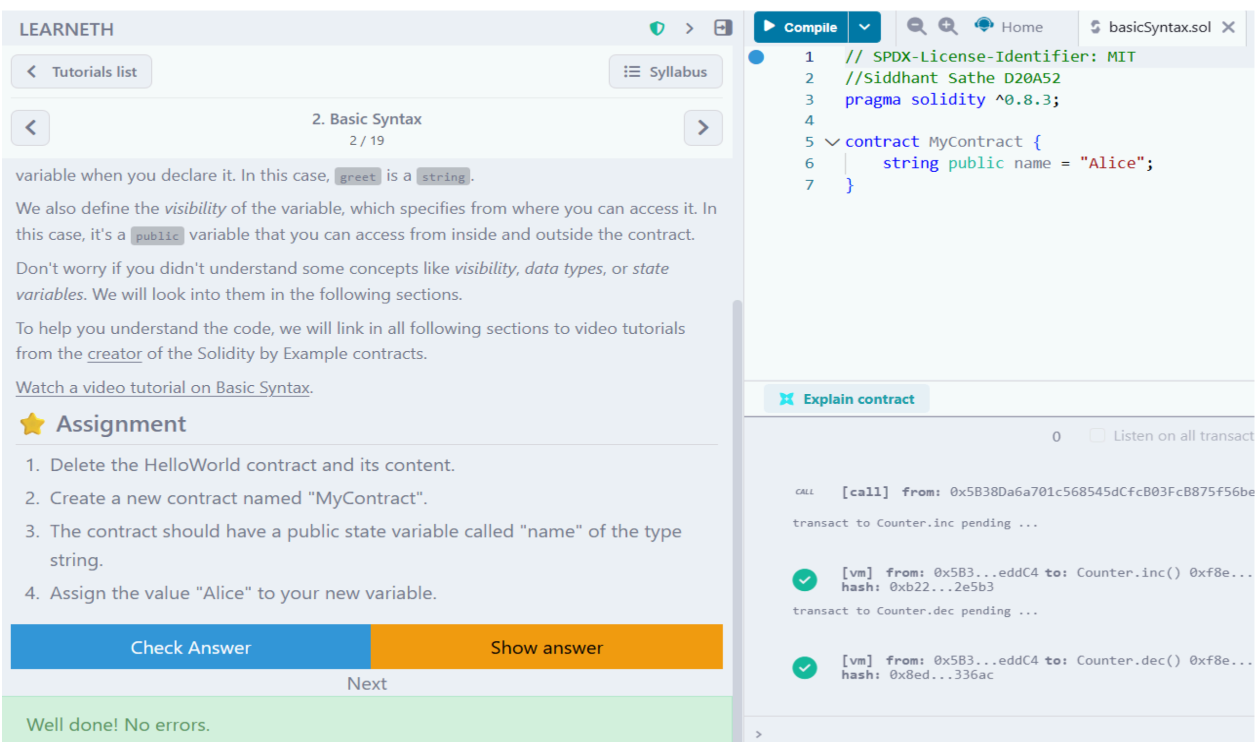
- Tutorial no. 1 – get

Tutorial no. 1 – Increment

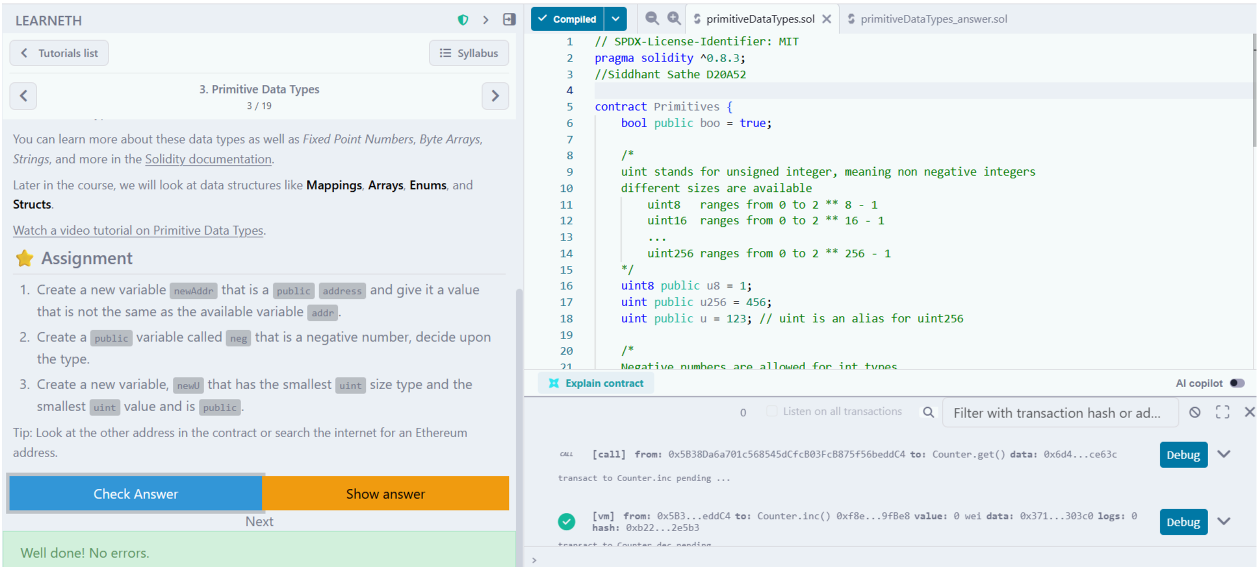
- Tutorial no. 1 – Decrement



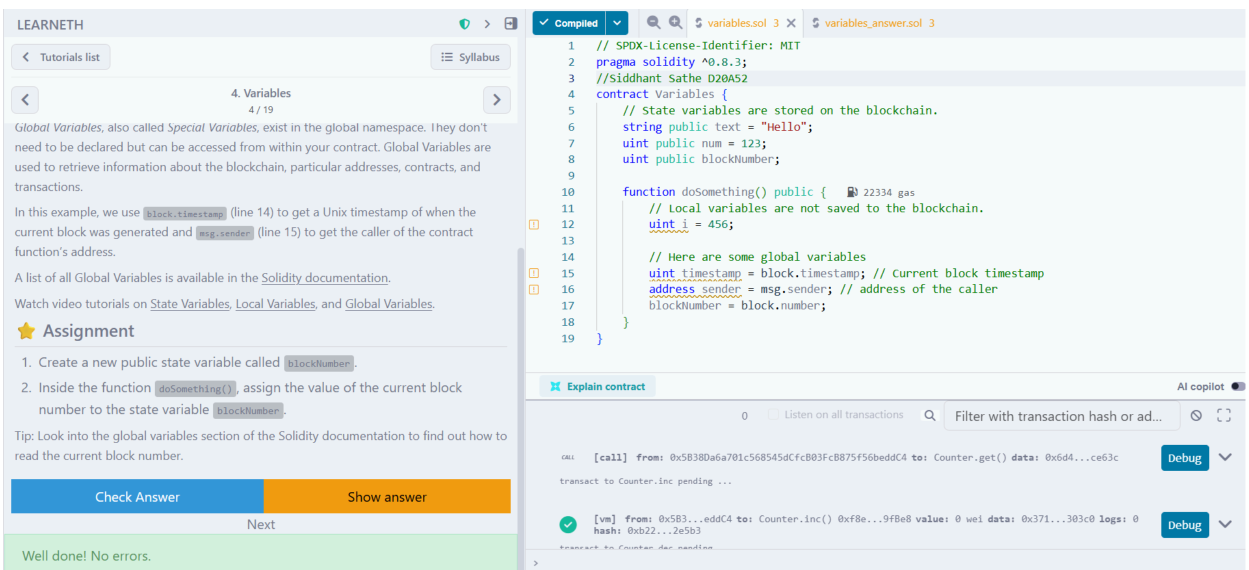
- Tutorial no. 2



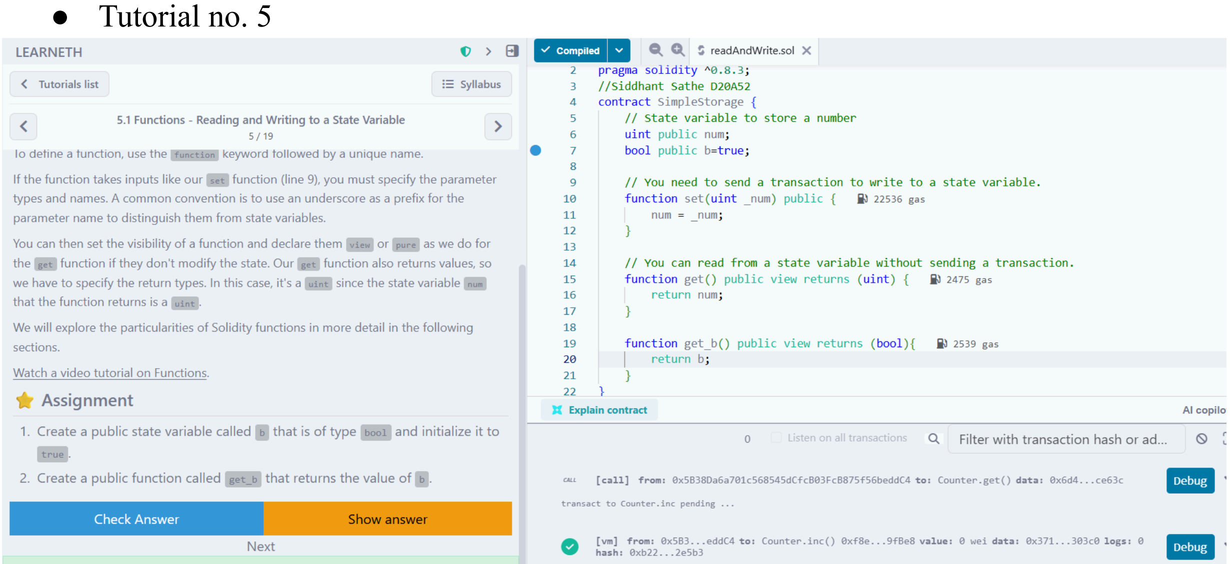
- Tutorial no. 3



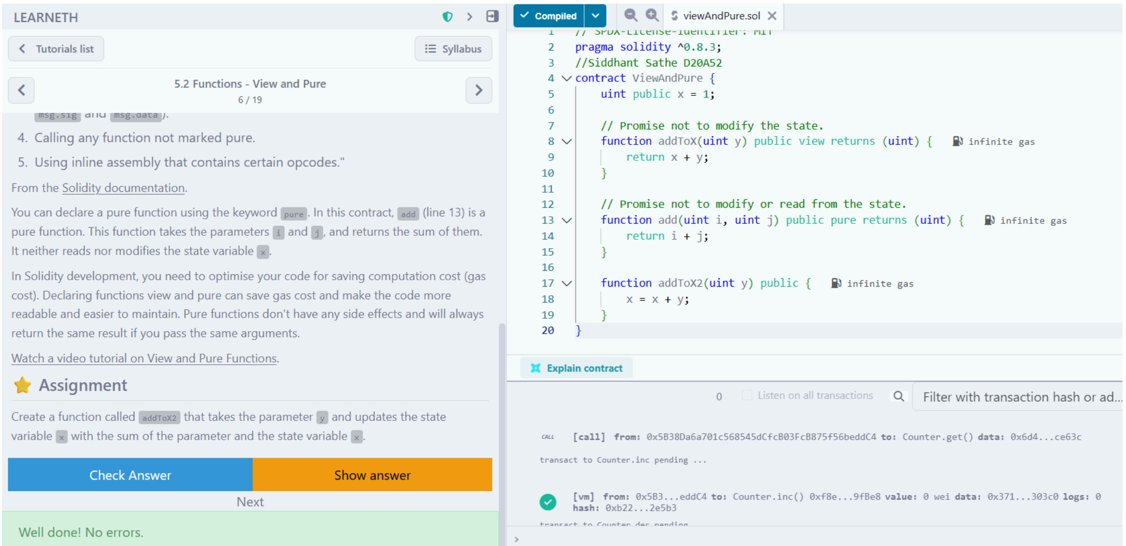
- Tutorial no. 4



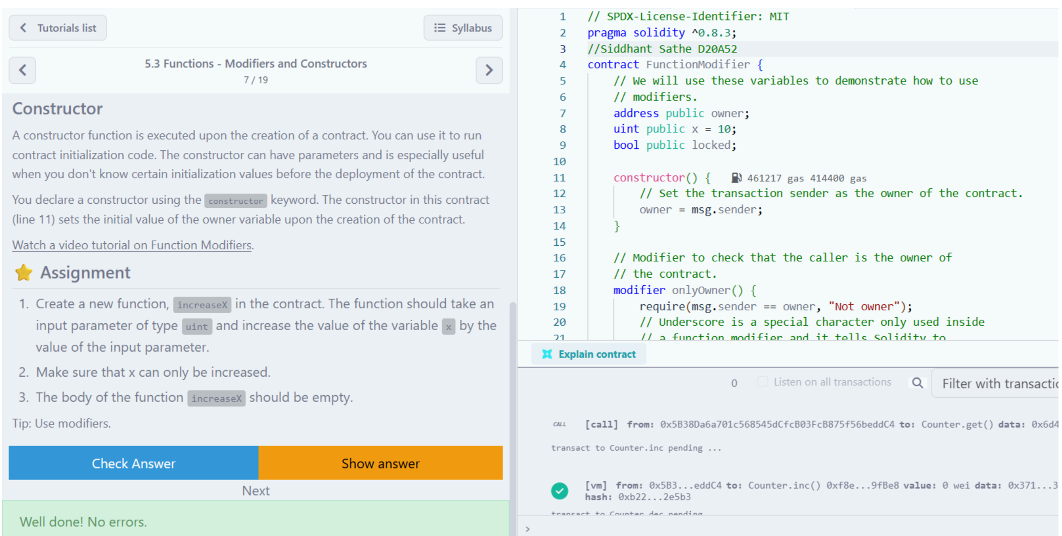
- Tutorial no. 5



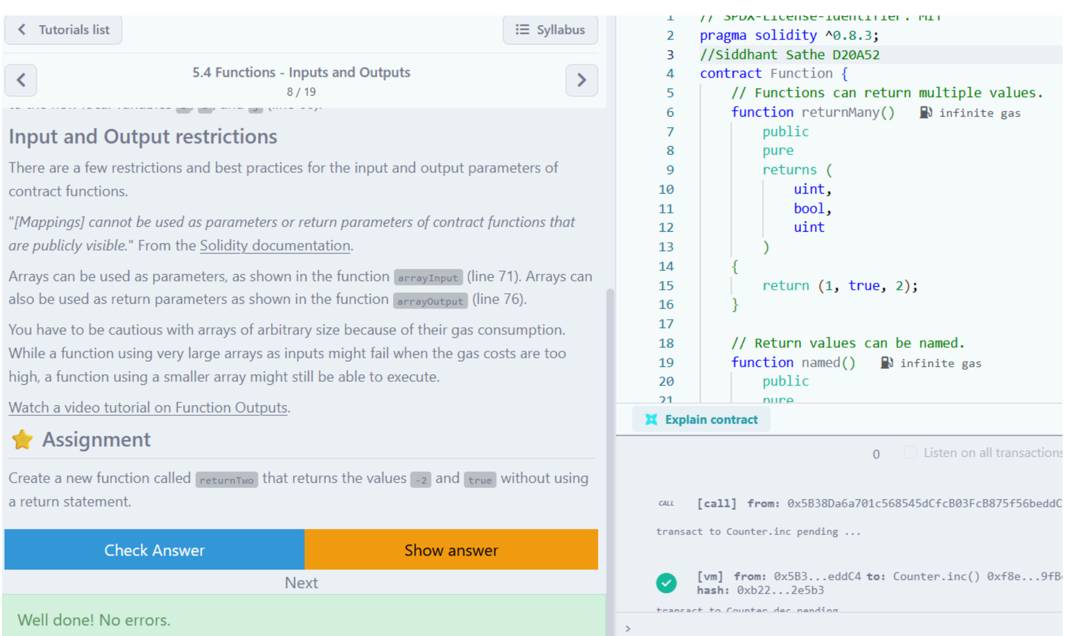
- Tutorial no. 6



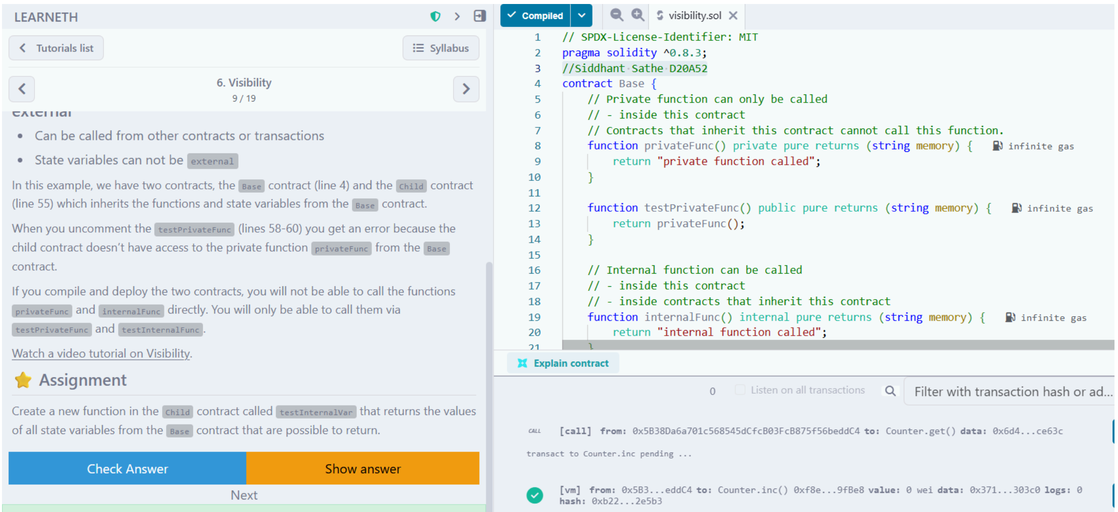
- Tutorial no. 7



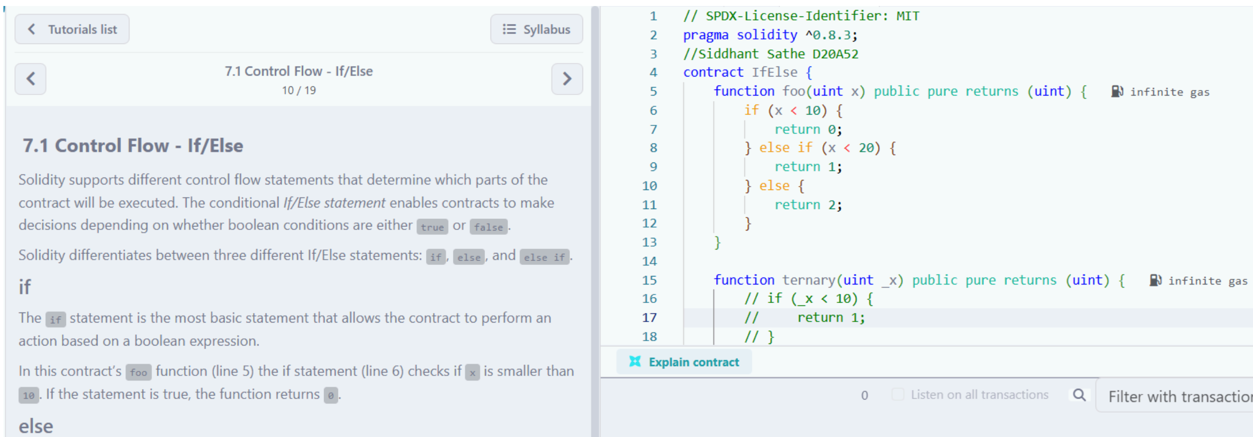
- Tutorial no. 8



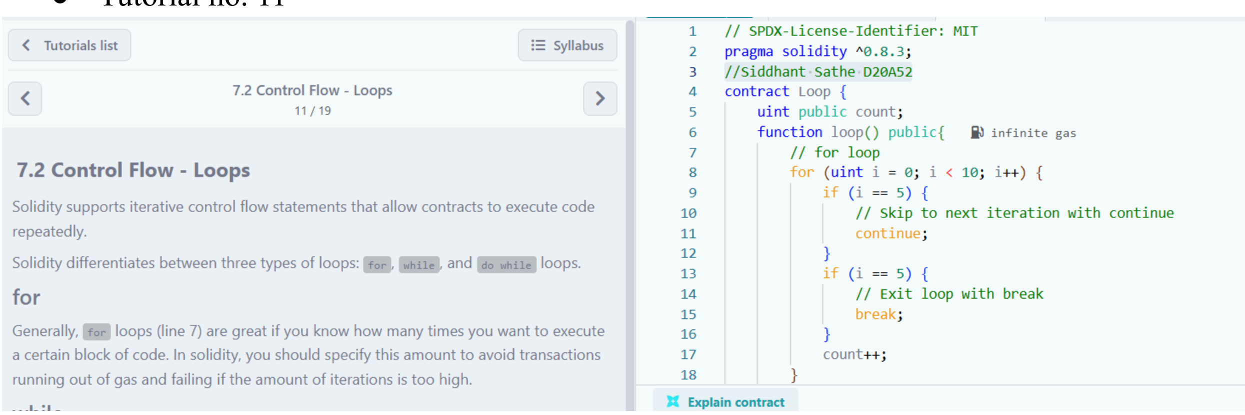
- Tutorial no. 9



- Tutorial no. 10



- Tutorial no. 11



- Tutorial no. 12

Tutorials list | Syllabus

8.1 Data Structures - Arrays
12 / 19

8.1 Data Structures - Arrays

In the next sections, we will look into the data structures that we can use to organize and store our data in Solidity.

Arrays, *mappings* and *structs* are all *reference types*. Unlike *value types* (e.g. *booleans* or *integers*) reference types don't store their value directly. Instead, they store the location where the value is being stored. Multiple reference type variables could reference the same location, and a change in one variable would affect the others, therefore they need to be handled carefully.

In Solidity, an array stores an ordered list of values of the same type that are indexed numerically.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
contract Array {
    // Several ways to initialize an array
    uint[] public arr;
    uint[] public arr2 = [1, 2, 3];
    // Fixed sized array, all elements initialize to 0
    uint[10] public myFixedSizeArr;

    function get(uint i) public view returns (uint) {    infinite gas
        return arr[i];
    }

    // Solidity can return the entire array.
    // But this function should be avoided for
    // arrays that can grow indefinitely in length.
    function getArr() public view returns (uint[] memory) {    infinite gas
```

Explain contract

- Tutorial no. 13

Tutorials list | Syllabus

8.2 Data Structures - Mappings
13 / 19

8.2 Data Structures - Mappings

In Solidity, *mappings* are a collection of key types and corresponding value type pairs.

The biggest difference between a mapping and an array is that you can't iterate over mappings. If we don't know a key we won't be able to access its value. If we need to know all of our data or iterate over it, we should use an array.

If we want to retrieve a value based on a known key we can use a mapping (e.g. addresses are often used as keys). Looking up values with a mapping is easier and cheaper than iterating over arrays. If arrays become too large, the gas cost of iterating over it could become too high and cause the transaction to fail.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public myMap;

    function get(address _addr) public view returns (uint) {    2872 gas
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return myMap[_addr];
    }

    function set(address _addr, uint _i) public {    22842 gas
        // Update the value at this address
        myMap[_addr] = _i;
    }

```

Explain contract

- Tutorial no. 14

Tutorials list | Syllabus

8.3 Data Structures - Structs
14 / 19

8.3 Data Structures - Structs

In Solidity, we can define custom data types in the form of *structs*. Structs are a collection of variables that can consist of different data types.

Defining structs

We define a struct using the `struct` keyword and a name (line 5). Inside curly braces, we can define our struct's members, which consist of the variable names and their data types.

Initializing structs

There are different ways to initialize a struct.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.3;
3  //Siddhant Sathe D20A52
4  contract Todos {
5      struct Todo {
6          string text;
7          bool completed;
8      }
9
10     // An array of 'Todo' structs
11     Todo[] public todos;
12
13     function create(string memory _text) public {    infinite gas
14         // 3 ways to initialize a struct
15         // - calling it like a function
16         todos.push(Todo(_text, false));
17
18         // key value mapping
```

Explain contract

- Tutorial no. 15

Tutorials list | Syllabus

8.4 Data Structures - Enums
15 / 19

8.4 Data Structures - Enums

In Solidity *enums* are custom data types consisting of a limited set of constant values. We use enums when our variables should only get assigned a value from a predefined set of values.

In this contract, the state variable `status` can get assigned a value from the limited set of provided values of the enum `Status` representing the various states of a shipping status.

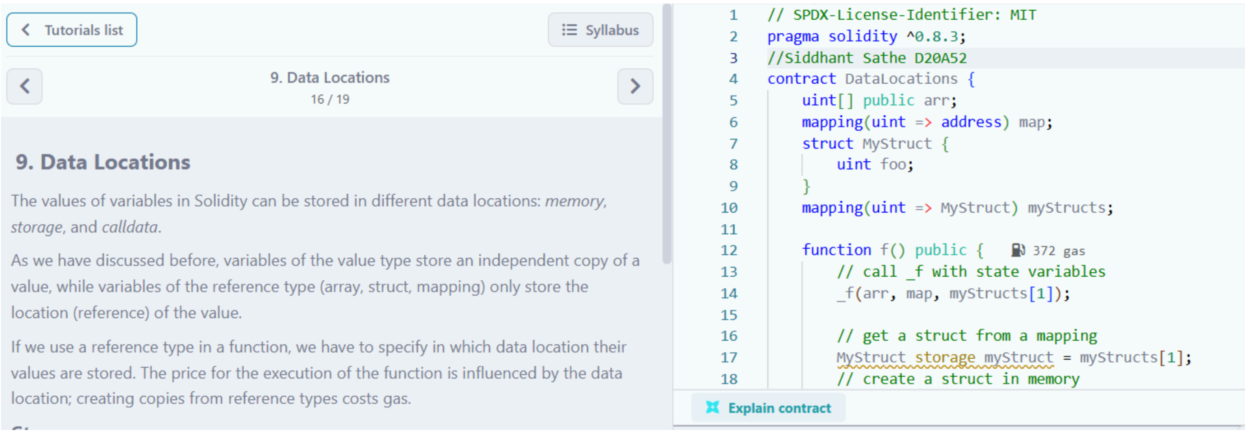
Defining enums

We define an enum with the enum keyword, followed by the name of the custom type we want to create (line 6). Inside the curly braces, we define all available members of the

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.3;
3  //Siddhant Sathe D20A52
4  contract Enum {
5      // Enum representing shipping status
6      enum Status {
7          Pending,
8          Shipped,
9          Accepted,
10         Rejected,
11         Canceled
12     }
13
14     // Default value is the first element listed in
15     // definition of the type, in this case "Pending"
16     Status public status;
17
18     // Returns uint
```

Explain contract

- Tutorial no. 16



Tutorials list | Syllabus

9. Data Locations
16 / 19

9. Data Locations

The values of variables in Solidity can be stored in different data locations: *memory*, *storage*, and *calldata*.

As we have discussed before, variables of the value type store an independent copy of a value, while variables of the reference type (array, struct, mapping) only store the location (reference) of the value.

If we use a reference type in a function, we have to specify in which data location their values are stored. The price for the execution of the function is influenced by the data location; creating copies from reference types costs gas.

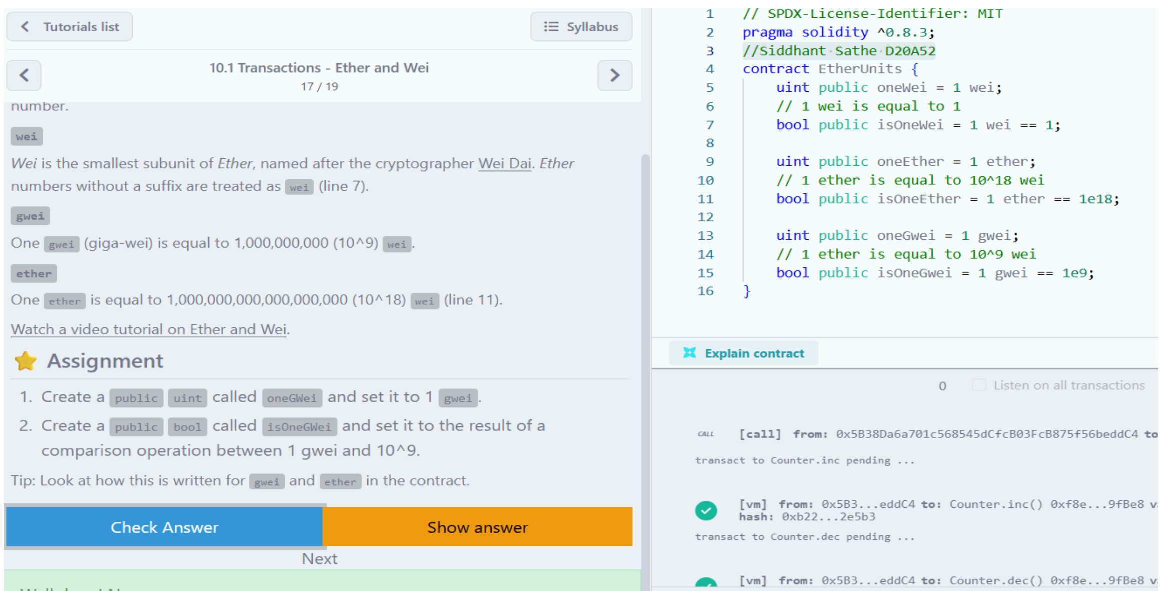
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
contract DataLocations {
    uint[] public arr;
    mapping(uint => address) map;
    struct MyStruct {
        uint foo;
    }
    mapping(uint => MyStruct) myStructs;

    function f() public {    372 gas
        // call _f with state variables
        _f(arr, map, myStructs[1]);

        // get a struct from a mapping
        MyStruct storage myStruct = myStructs[1];
        // create a struct in memory
```

Explain contract

- Tutorial no. 17



Tutorials list | Syllabus

10.1 Transactions - Ether and Wei
17 / 19

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

Next

Well done! No errors

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Siddhant Sathe D20A52
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  }
```

Explain contract

0 Listen on all transactions

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to

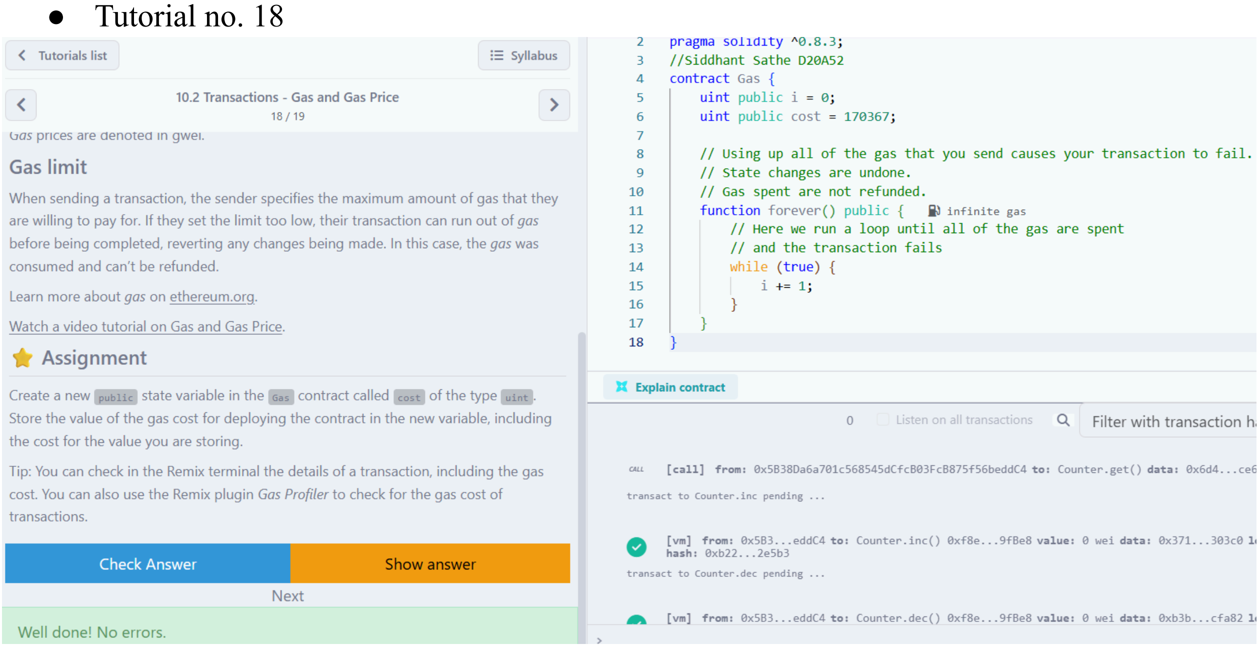
transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 v
hash: 0xb22...2e5b3

transact to Counter.dec pending ...

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xf8e...9fBe8 v

- Tutorial no. 18



Tutorials list | Syllabus

10.2 Transactions - Gas and Gas Price
18 / 19

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.

Check Answer | Show answer

Next

Well done! No errors.

```
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
contract Gas {
    uint public i = 0;
    uint public cost = 170367;

    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {    infinite gas
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

Explain contract

0 Listen on all transactions Filter with transaction h

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce6

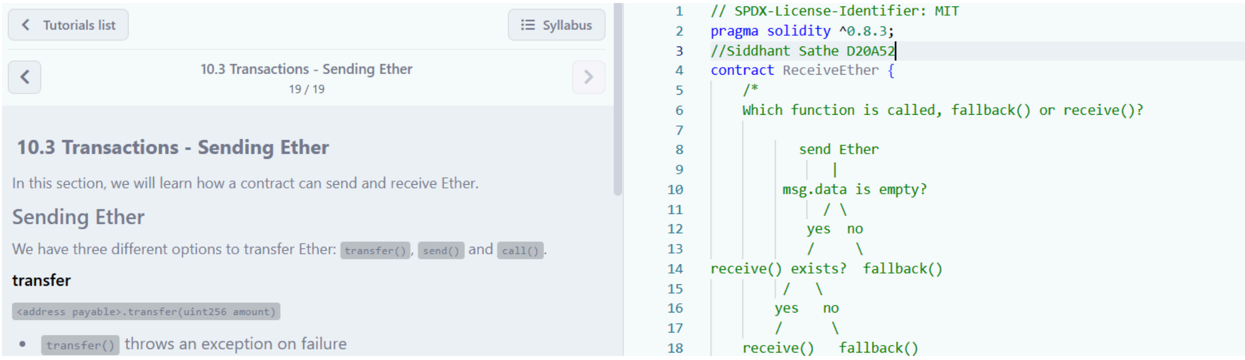
transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 l
hash: 0xb22...2e5b3

transact to Counter.dec pending ...

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xf8e...9fBe8 value: 0 wei data: 0xb3b...cfa82 l

- Tutorial no. 19



Conclusion : Through this experiment, the fundamentals of Solidity programming were explored by completing practical assignments in Remix IDE. This hands-on approach improved understanding of smart contract creation, deployment, and execution on the Ethereum blockchain.