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:
    - o public: available both inside and outside the contract. o private: only accessible within the same contract. o internal: accessible within the contract and its child contracts.
    - o 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:

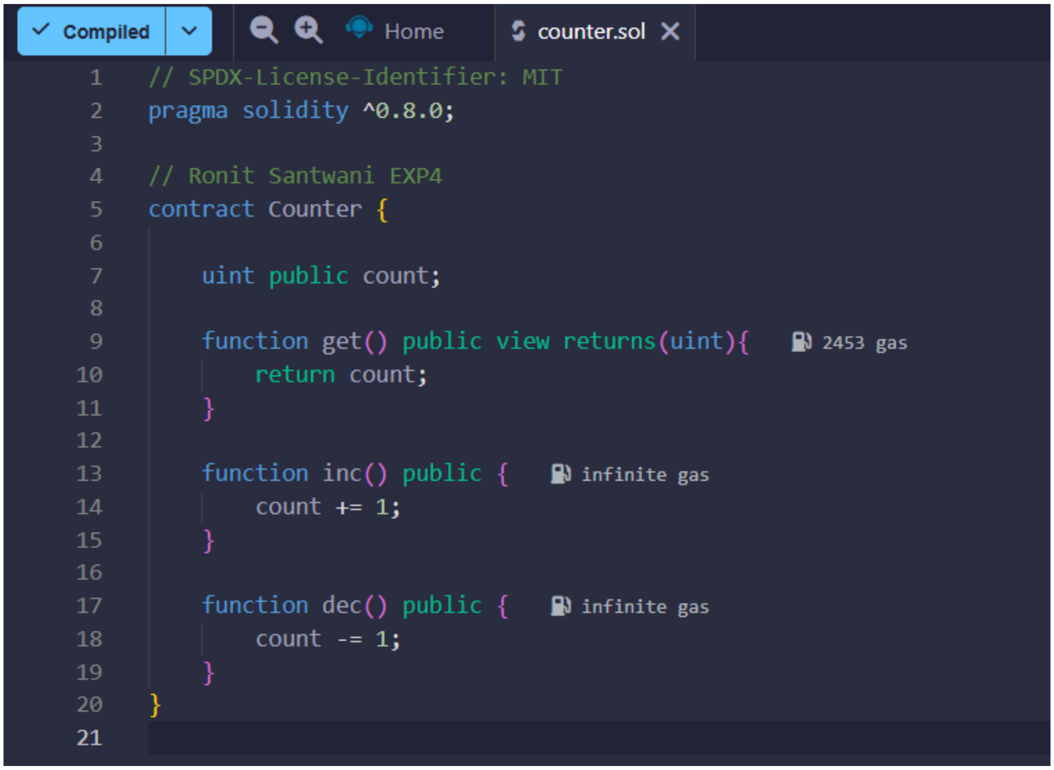
- **storage**: Data stored permanently on the blockchain. Examples: state variables.
- **memory**: Temporary data storage that exists only while a function is executing. Used for local variables and function inputs.
- **calldata**: A non-modifiable and non-persistent location used for external function 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.

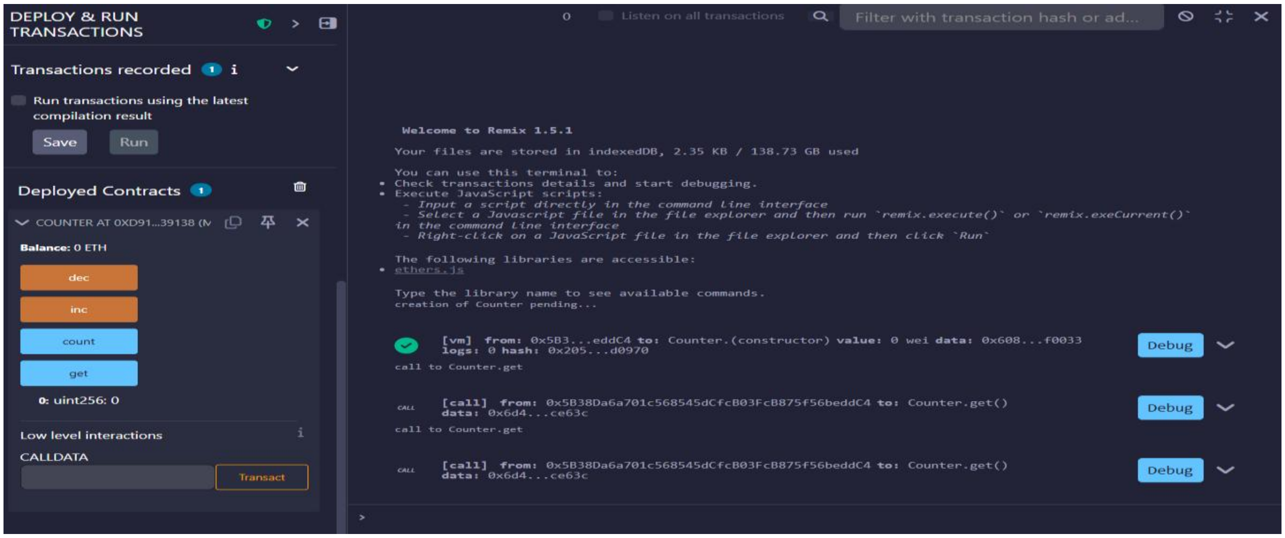
Implementation:

- Tutorial no. 1 – Compile the code



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// Ronit Santwani EXP4
contract Counter {

    uint public count;

    function get() public view returns(uint){
        return count;
    }

    function inc() public {
        count += 1;
    }

    function dec() public {
        count -= 1;
    }
}

```

- Tutorial no. 1 – get

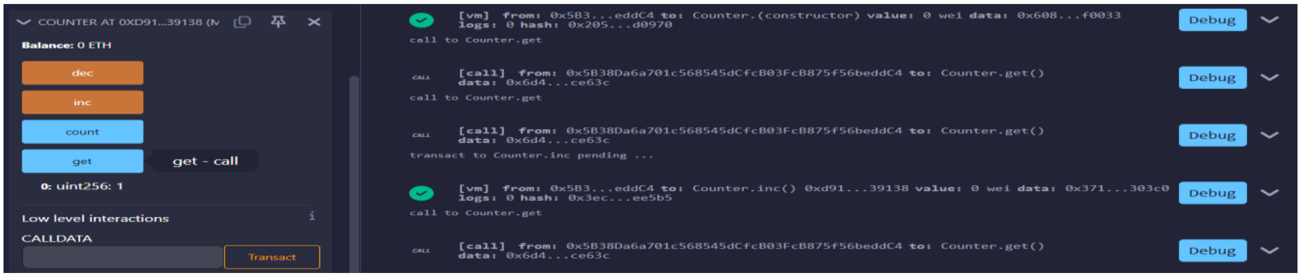

DEPLOY & RUN TRANSACTIONS
Transactions recorded 1
Run transactions using the latest compilation result
Save
Run
Deployed Contracts 1
COUNTER AT 0XD91...39138 (M
dec
inc
count
get
Low level interactions
CALLDATA
Transact
0
Listen on all transactions
Filter with transaction hash or ad...
Debug

Balance: 0 ETH

0: uint256: 0

```
 Welcome to Remix 1.5.1

Your files are stored in indexedDB, 2.35 KB / 138.73 GB used

You can use this terminal to:
• Check transactions details and start debugging.
• Execute JavaScript scripts:
  - Input a script directly in the command line interface
  - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.exeCurrent()`
in the command line interface
  - Right-click on a JavaScript file in the file explorer and then click `Run`

The following libraries are accessible:
• ethers.js

Type the library name to see available commands.
creation of Counter pending...

      [vm]  from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei data: 0x608...f0033
      logs: 0 hash: 0x205...d0970
call to Counter.get

CALL  [call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
      data: 0x6d4...ce63c
call to Counter.get

CALL  [call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
      data: 0x6d4...ce63c

>
```

Tutorial no. 1 – Increment

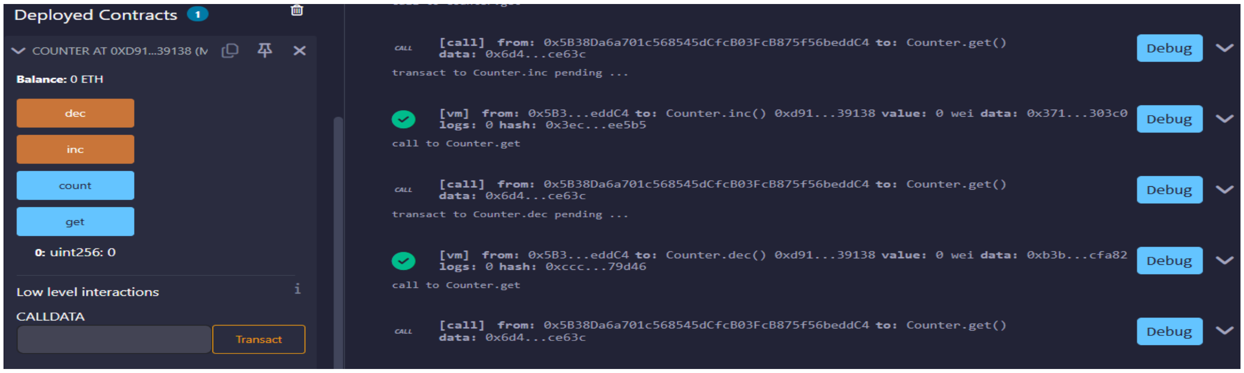


Balance: 0 ETH

0: uint256: 1

```
      [vm]  from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei data: 0x608...f0033
      logs: 0 hash: 0x205...d0970
call to Counter.get

CALL  [call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
      data: 0x6d4...ce63c
call to Counter.get

CALL  [call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
      data: 0x6d4...ce63c
transact to Counter.inc pending ...

      [vm]  from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371...303c0
      logs: 0 hash: 0x3ec...ee5b5
call to Counter.get

CALL  [call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
      data: 0x6d4...ce63c
```

- Tutorial no. 1 – Decrement

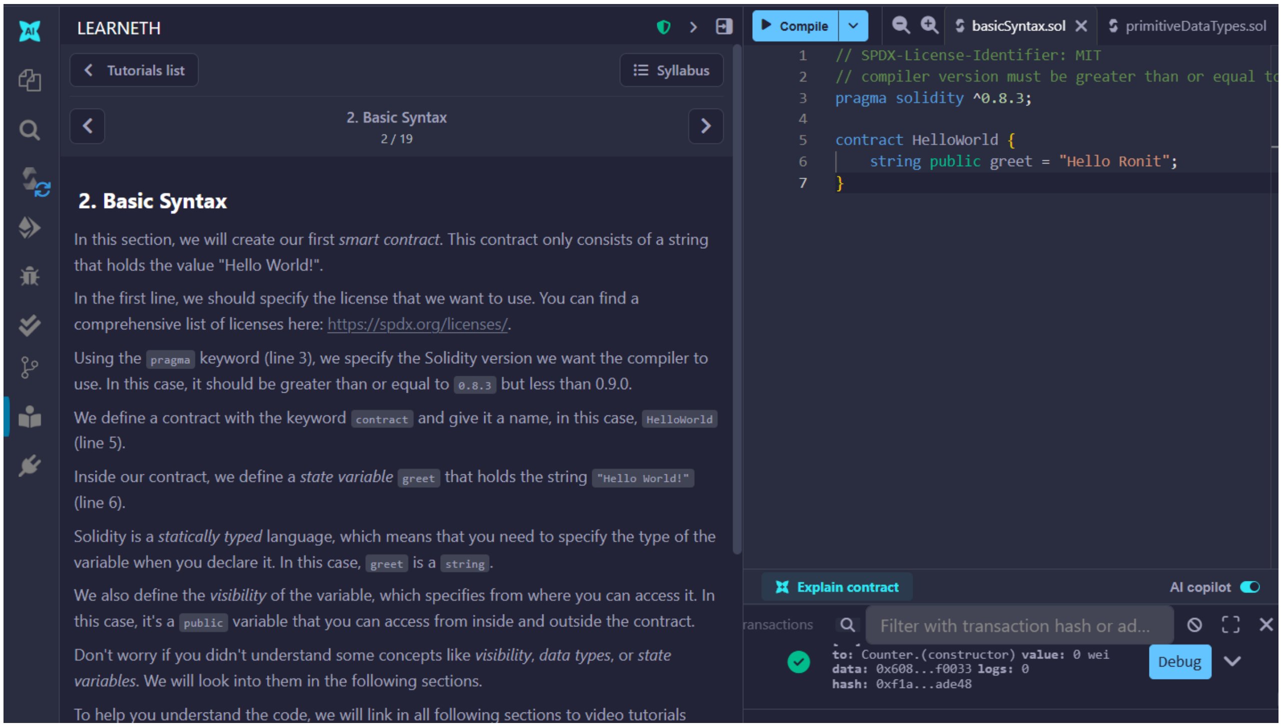


Balance: 0 ETH

0: uint256: 0

```
[call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
data: 0x6d4...ce63c
transact to Counter.inc pending ...

[vm]  from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371...303c0
logs: 0 hash: 0x3ec...ee5b5
call to Counter.get

[call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
data: 0x6d4...ce63c
transact to Counter.dec pending ...

[vm]  from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82
logs: 0 hash: 0xccc...79d46
call to Counter.get

[call]  from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get()
data: 0x6d4...ce63c
```

- Tutorial no. 2



2. Basic Syntax

In this section, we will create our first *smart contract*. This contract only consists of a string that holds the value "Hello World!".

In the first line, we should specify the license that we want to use. You can find a comprehensive list of licenses here: https://spdx.org/licenses/.

Using the `pragma` keyword (line 3), we specify the Solidity version we want the compiler to use. In this case, it should be greater than or equal to `0.8.3` but less than 0.9.0.

We define a contract with the keyword `contract` and give it a name, in this case, `HelloWorld` (line 5).

Inside our contract, we define a *state variable* `greet` that holds the string `"Hello World!"` (line 6).

Solidity is a *statically typed* language, which means that you need to specify the type of the variable when you declare it. In this case, `greet` is a `string`.

We also define the *visibility* of the variable, which specifies from where you can access it. In this case, it's a `public` variable that you can access from inside and outside the contract.

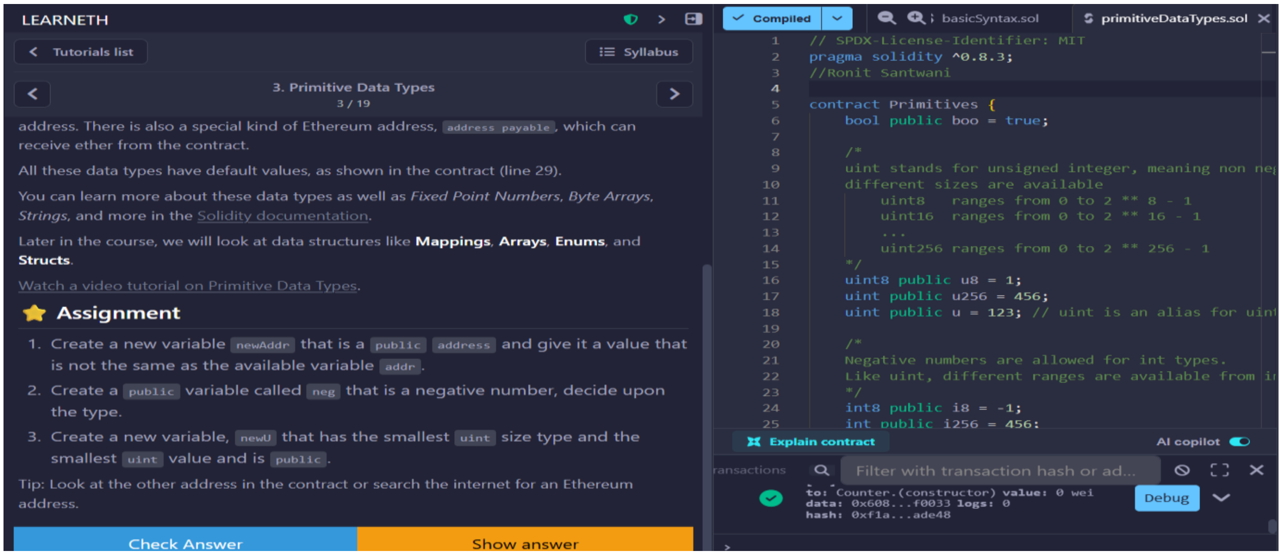
Don't worry if you didn't understand some concepts like *visibility, data types*, or *state variables*. We will look into them in the following sections.

To help you understand the code, we will link in all following sections to video tutorials

```
1  // SPDX-License-Identifier: MIT
2  // compiler version must be greater than or equal to
3  pragma solidity ^0.8.3;
4
5  contract HelloWorld {
6      string public greet = "Hello Ronit";
7  }
```

```
to: Counter.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0xf1a...ade48
```

- Tutorial no. 3



3. Primitive Data Types
3 / 19

address. There is also a special kind of Ethereum address, `address payable`, which can receive ether from the contract.

All these data types have default values, as shown in the contract (line 29).

You can learn more about these data types as well as *Fixed Point Numbers*, *Byte Arrays*, *Strings*, and more in the Solidity documentation.

Later in the course, we will look at data structures like **Mappings**, **Arrays**, **Enums**, and **Structs**.

Watch a video tutorial on Primitive Data Types.

⭐ Assignment

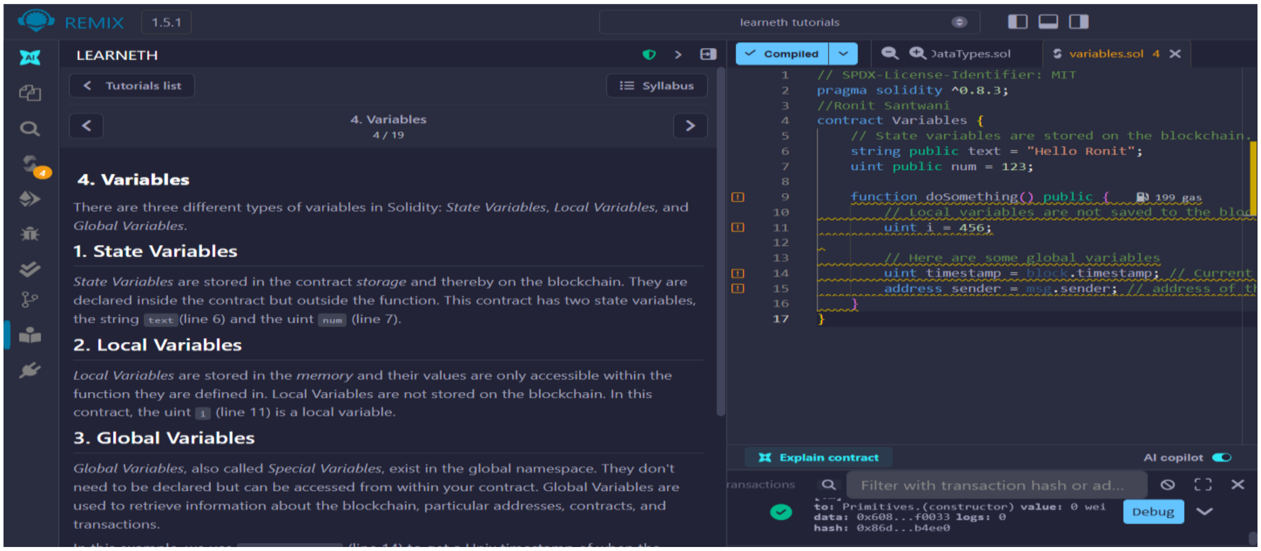
1. Create a new variable `newAddr` that is a `public` `address` and give it a value that is not the same as the available variable `addr`.
2. Create a `public` variable called `neg` that is a negative number, decide upon the type.
3. Create a new variable, `newU` that has the smallest `uint` size type 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.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Primitives {
6       bool public boo = true;
7
8       /*
9       uint stands for unsigned integer, meaning non neg
10      different sizes are available
11          uint8   ranges from 0 to 2 ** 8 - 1
12          uint16  ranges from 0 to 2 ** 16 - 1
13          ...
14          uint256 ranges from 0 to 2 ** 256 - 1
15      */
16      uint8 public u8 = 1;
17      uint public u256 = 456;
18      uint public u = 123; // uint is an alias for uint
19
20      /*
21      Negative numbers are allowed for int types.
22      Like uint, different ranges are available from in
23      */
24      int8 public i8 = -1;
25      int public i256 = 456;
```

```
to: Counter.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0xf1a...ade48
```

- Tutorial no. 4



4. Variables
4 / 19

4. Variables

There are three different types of variables in Solidity: *State Variables*, *Local Variables*, and *Global Variables*.

1. State Variables

State Variables are stored in the contract *storage* and thereby on the blockchain. They are declared inside the contract but outside the function. This contract has two state variables, the string `text`(line 6) and the uint `num` (line 7).

2. Local Variables

Local Variables are stored in the *memory* and their values are only accessible within the function they are defined in. Local Variables are not stored on the blockchain. In this contract, the uint `i` (line 11) is a local variable.

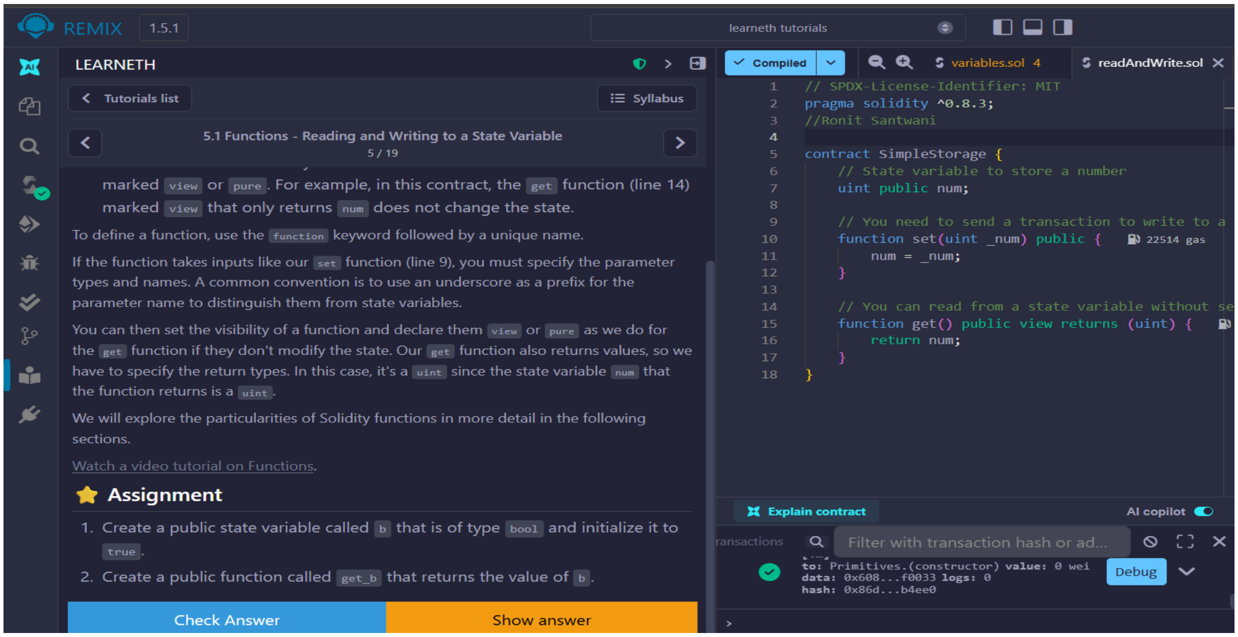
3. Global Variables

Global Variables, also called *Special Variables*, exist in the global namespace. They don't need to be declared but can be accessed from within your contract. Global Variables are used to retrieve information about the blockchain, particular addresses, contracts, and transactions.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4   contract Variables {
5       // State variables are stored on the blockchain.
6       string public text = "Hello Ronit";
7       uint public num = 123;
8
9       function doSomething() public {     199 gas
10          // Local variables are not saved to the bloc
11          uint i = 456;
12
13          // Here are some global variables
14          uint timestamp = block.timestamp; // Current
15          address sender = msg.sender; // address of th
16      }
17  }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 5



5.1 Functions - Reading and Writing to a State Variable
5 / 19

marked `view` or `pure`. For example, in this contract, the `get` function (line 14) marked `view` that only returns `num` does not change the state.

To define a function, use the `function` keyword followed by a unique name.

If the function takes inputs like our `set` function (line 9), you must specify the parameter types and names. A common convention is to use an underscore as a prefix for the parameter name to distinguish them from state variables.

You can then set the visibility of a function and declare them `view` or `pure` as we do for the `get` function if they don't modify the state. Our `get` function also returns values, so we have to specify the return types. In this case, it's a `uint` since the state variable `num` that the function returns is a `uint`.

We will explore the particularities of Solidity functions in more detail in the following sections.

Watch a video tutorial on Functions.

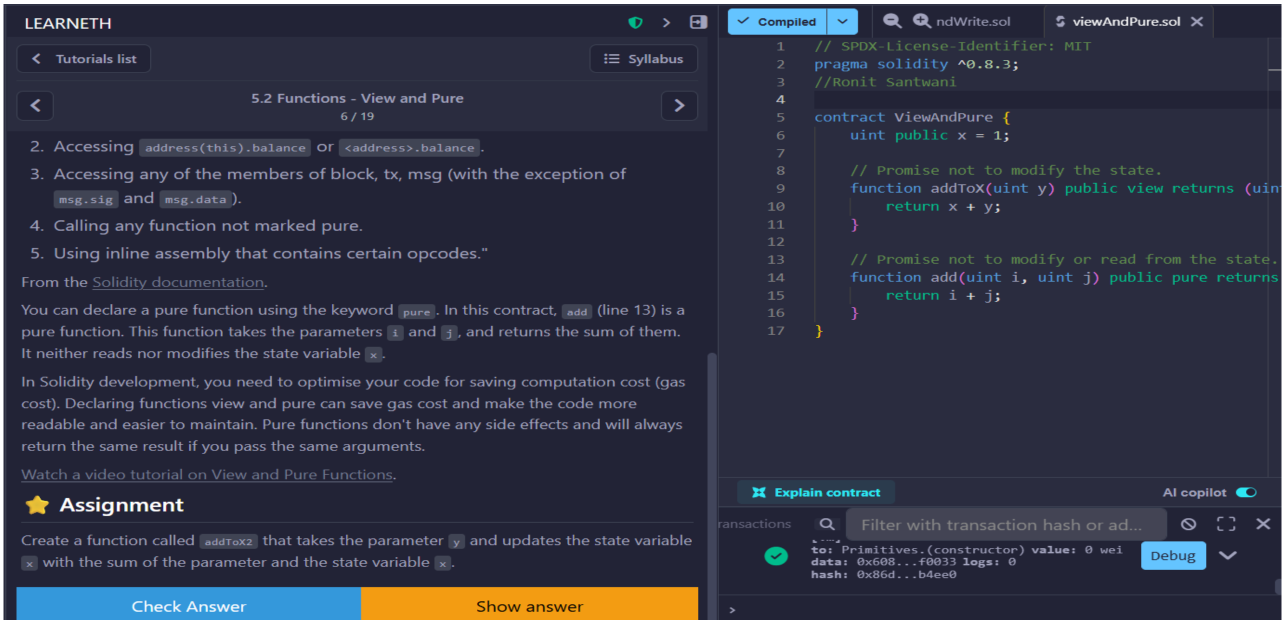
⭐ Assignment

1. Create a public state variable called `b` that is of type `bool` and initialize it to `true`.
2. Create a public function called `get_b` that returns the value of `b`.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract SimpleStorage {
6       // State variable to store a number
7       uint public num;
8
9       // You need to send a transaction to write to a
10      function set(uint _num) public {     22514 gas
11          num = _num;
12      }
13
14      // You can read from a state variable without se
15      function get() public view returns (uint) {
16          return num;
17      }
18  }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 6



5.2 Functions - View and Pure
6 / 19

2. Accessing `address(this).balance` or `<address>.balance`.
3. Accessing any of the members of block, tx, msg (with the exception of `msg.sig` and `msg.data`).
4. Calling any function not marked pure.
5. Using inline assembly that contains certain opcodes."

From the Solidity documentation.

You can declare a pure function using the keyword `pure`. In this contract, `add` (line 13) is a pure function. This function takes the parameters `i` and `j`, and returns the sum of them. It neither reads nor modifies the state variable `x`.

In Solidity development, you need to optimise your code for saving computation cost (gas cost). Declaring functions view and pure can save gas cost and make the code more readable and easier to maintain. Pure functions don't have any side effects and will always return the same result if you pass the same arguments.

Watch a video tutorial on View and Pure Functions.

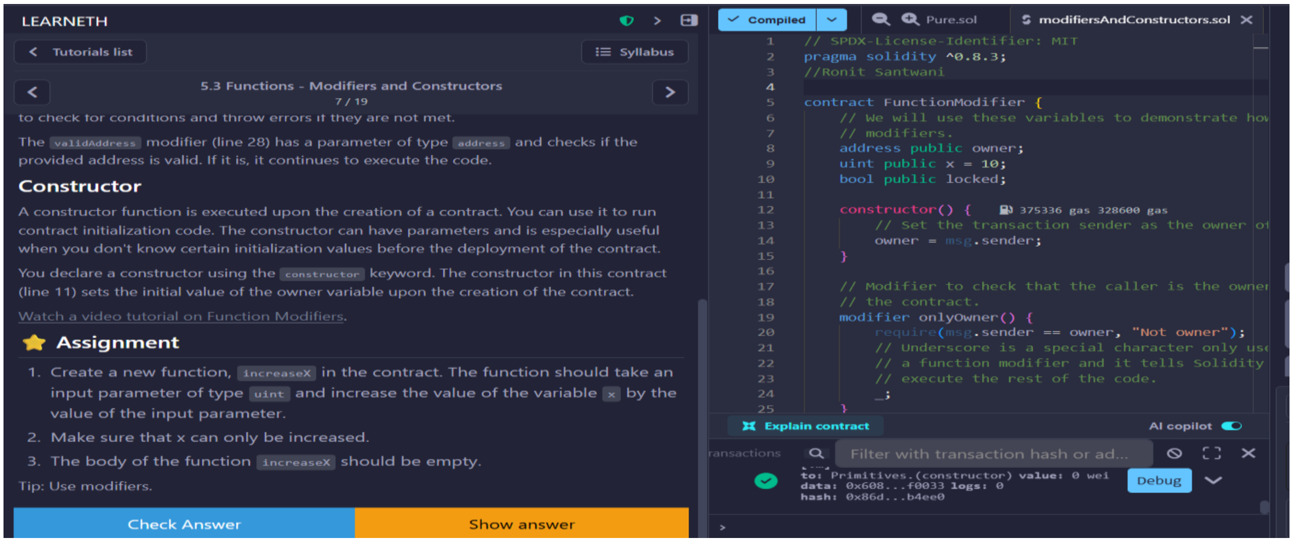
⭐ Assignment

Create a function called `addToX2` that takes the parameter `y` and updates the state variable `x` with the sum of the parameter and the state variable `x`.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract ViewAndPure {
6       uint public x = 1;
7
8       // Promise not to modify the state.
9       function addToX(uint y) public view returns (uin
10          return x + y;
11      }
12
13      // Promise not to modify or read from the state.
14      function add(uint i, uint j) public pure returns
15          return i + j;
16      }
17  }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 7



5.3 Functions - Modifiers and Constructors
7 / 19

to check for conditions and throw errors if they are not met.

The `validAddress` modifier (line 28) has a parameter of type `address` and checks if the provided address is valid. If it is, it continues to execute the code.

Constructor

A constructor function is executed upon the creation of a contract. You can use it to run contract initialization code. The constructor can have parameters and is especially useful when you don't know certain initialization values before the deployment of the contract.

You declare a constructor using the `constructor` keyword. The constructor in this contract (line 11) sets the initial value of the owner variable upon the creation of the contract.

Watch a video tutorial on Function Modifiers.

⭐ Assignment

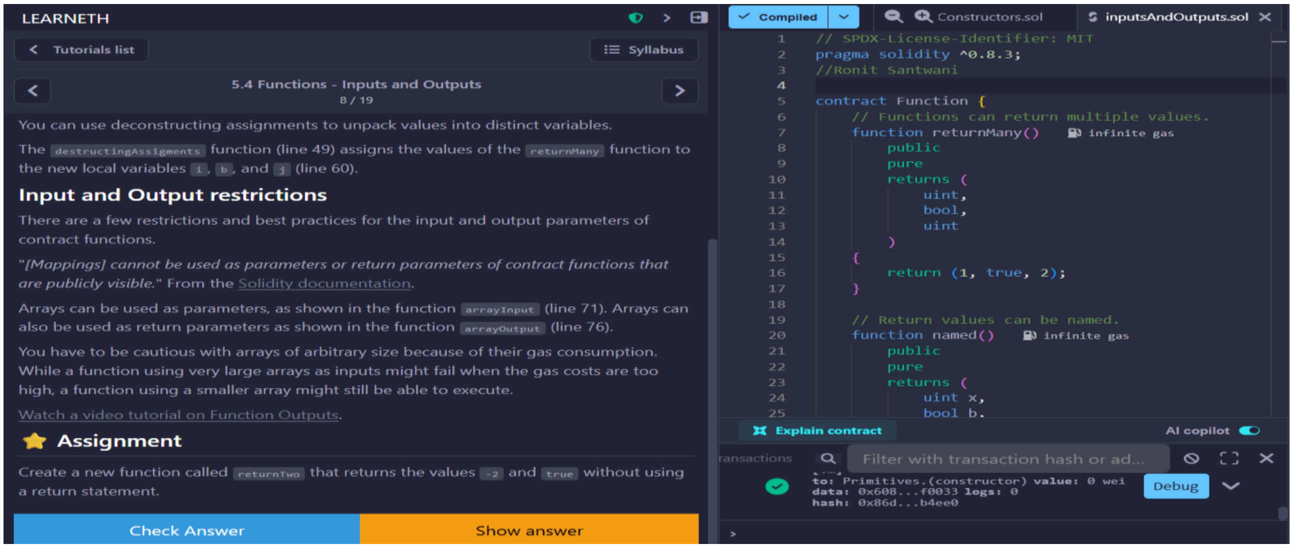
1. Create a new function, `increaseX` in the contract. The function should take an input parameter of type `uint` and increase the value of the variable `x` by the value of the input parameter.
2. Make sure that x can only be increased.
3. The body of the function `increaseX` should be empty.

Tip: Use modifiers.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract FunctionModifier {
6       // We will use these variables to demonstrate how
7       // modifiers.
8       address public owner;
9       uint public x = 10;
10      bool public locked;
11
12      constructor() {     375336 gas 328600 gas
13          // Set the transaction sender as the owner of
14          owner = msg.sender;
15      }
16
17      // Modifier to check that the caller is the owner
18      // the contract.
19      modifier onlyOwner() {
20          require(msg.sender == owner, "Not owner");
21          // Underscore is a special character only use
22          // a function modifier and it tells Solidity
23          // execute the rest of the code.
24          _;
25      }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 8



5.4 Functions - Inputs and Outputs
8 / 19

You can use deconstructing assignments to unpack values into distinct variables.

The `destructingAssigments` function (line 49) assigns the values of the `returnMany` function to the new local variables `i`, `b`, and `j` (line 60).

Input and Output restrictions

There are a few restrictions and best practices for the input and output parameters of contract functions.

"*[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible.*" From the Solidity documentation.

Arrays can be used as parameters, as shown in the function `arrayInput` (line 71). Arrays can also be used as return parameters as shown in the function `arrayOutput` (line 76).

You have to be cautious with arrays of arbitrary size because of their gas consumption. While a function using very large arrays as inputs might fail when the gas costs are too high, a function using a smaller array might still be able to execute.

Watch a video tutorial on Function Outputs.

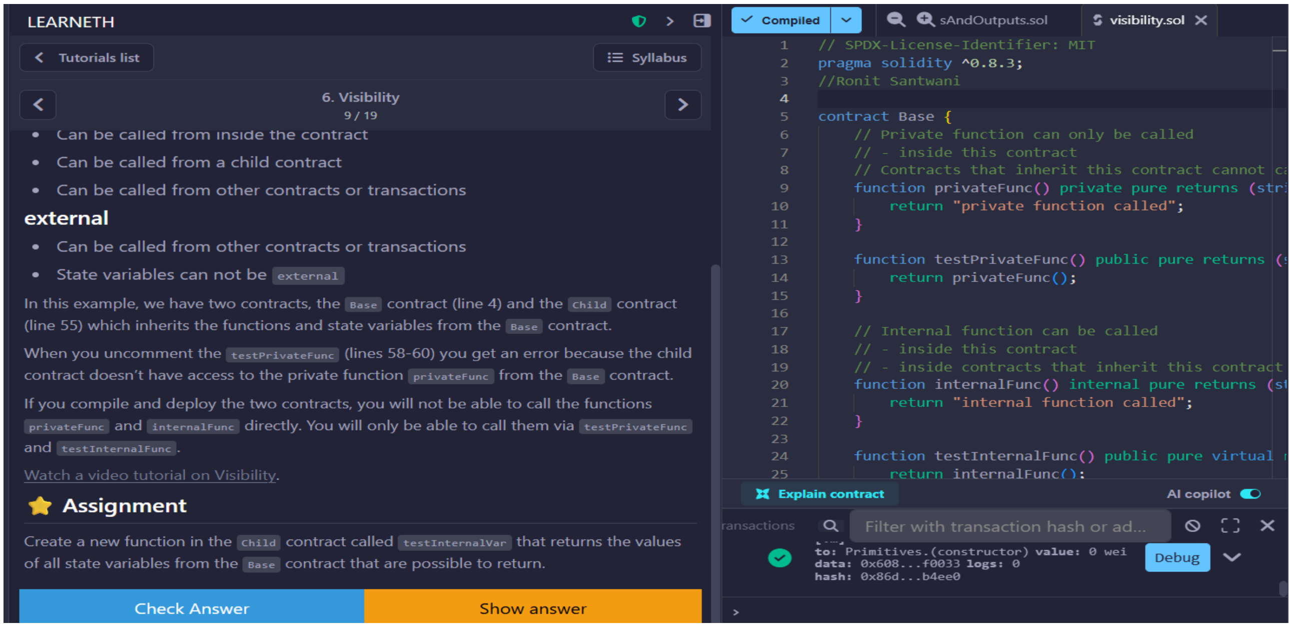
⭐ Assignment

Create a new function called `returnTwo` that returns the values `-2` and `true` without using a return statement.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Function {
6       // Functions can return multiple values.
7       function returnMany()     infinite gas
8           public
9           pure
10          returns (
11              uint,
12              bool,
13              uint
14          )
15      {
16          return (1, true, 2);
17      }
18
19      // Return values can be named.
20      function named()     infinite gas
21          public
22          pure
23          returns (
24              uint x,
25              bool b.
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 9



- Can be called from inside the contract
- Can be called from a child contract
- Can be called from other contracts or transactions

external

- Can be called from other contracts or transactions
- State variables can not be `external`

In this example, we have two contracts, the `Base` contract (line 4) and the `Child` contract (line 55) which inherits the functions and state variables from the `Base` contract.

When you uncomment the `testPrivateFunc` (lines 58-60) you get an error because the child contract doesn't have access to the private function `privateFunc` from the `Base` contract.

If you compile and deploy the two contracts, you will not be able to call the functions `privateFunc` and `internalFunc` directly. You will only be able to call them via `testPrivateFunc` and `testInternalFunc`.

Watch a video tutorial on Visibility.

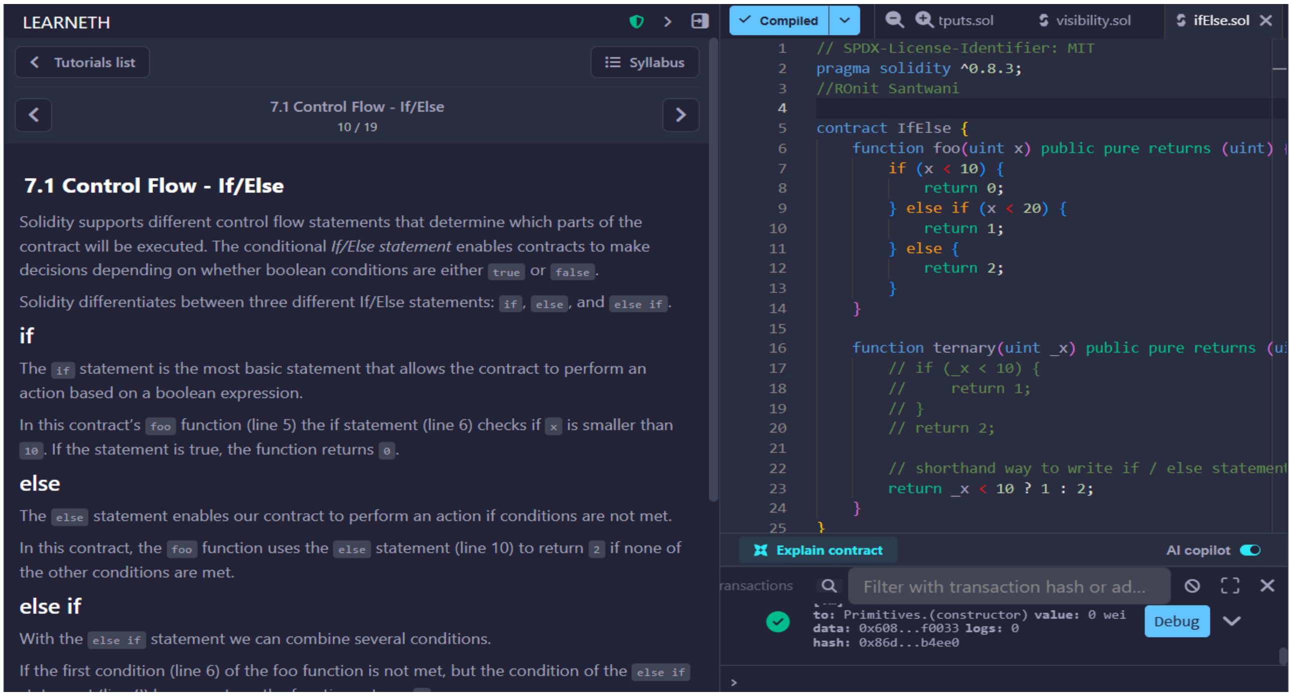
Assignment

Create a new function in the `Child` contract called `testInternalVar` that returns the values of all state variables from the `Base` contract that are possible to return.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Base {
6       // Private function can only be called
7       // - inside this contract
8       // Contracts that inherit this contract cannot ca
9       function privateFunc() private pure returns (stri
10          return "private function called";
11      }
12
13      function testPrivateFunc() public pure returns (s
14          return privateFunc();
15      }
16
17      // Internal function can be called
18      // - inside this contract
19      // - inside contracts that inherit this contract
20      function internalFunc() internal pure returns (st
21          return "internal function called";
22      }
23
24      function testInternalFunc() public pure virtual r
25          return internalFunc();
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 10



7.1 Control Flow - If/Else

Solidity supports different control flow statements that determine which parts of the contract will be executed. The conditional *If/Else statement* enables contracts to make decisions depending on whether boolean conditions are either `true` or `false`.

Solidity differentiates between three different If/Else statements: `if`, `else`, and `else if`.

if

The `if` statement is the most basic statement that allows the contract to perform an action based on a boolean expression.

In this contract's `foo` function (line 5) the if statement (line 6) checks if `x` is smaller than `10`. If the statement is true, the function returns `0`.

else

The `else` statement enables our contract to perform an action if conditions are not met.

In this contract, the `foo` function uses the `else` statement (line 10) to return `2` if none of the other conditions are met.

else if

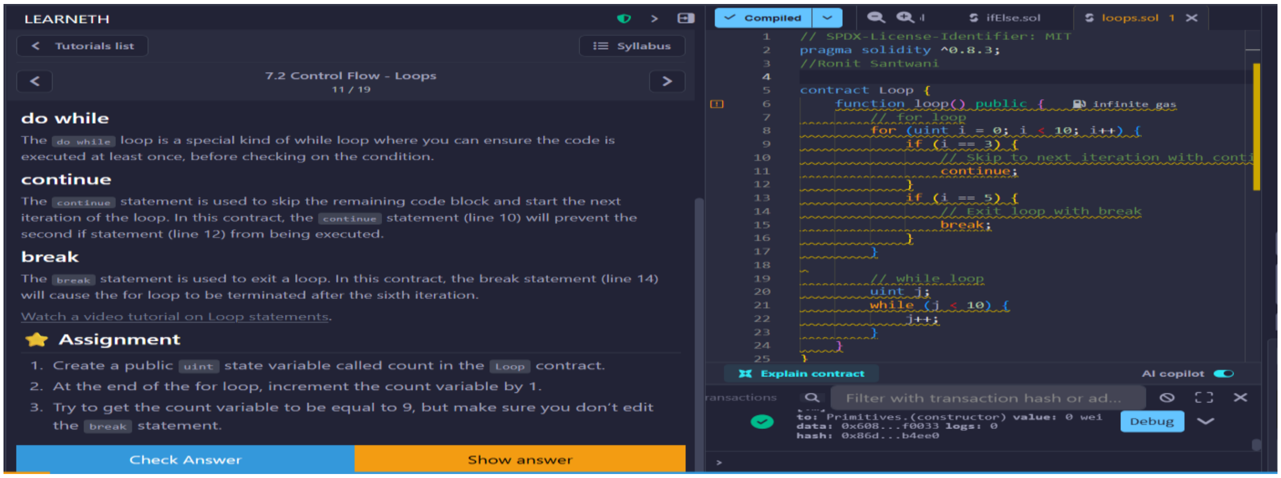
With the `else if` statement we can combine several conditions.

If the first condition (line 6) of the foo function is not met, but the condition of the `else if`

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //ROnit Santwani
4
5   contract IfElse {
6       function foo(uint x) public pure returns (uint) {
7           if (x < 10) {
8               return 0;
9           } else if (x < 20) {
10              return 1;
11          } else {
12              return 2;
13          }
14      }
15
16      function ternary(uint _x) public pure returns (ui
17          // if (_x < 10) {
18          //     return 1;
19          // }
20          // return 2;
21
22          // shorthand way to write if / else statement
23          return _x < 10 ? 1 : 2;
24      }
25  }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 11



do while

The `do while` loop is a special kind of while loop where you can ensure the code is executed at least once, before checking on the condition.

continue

The `continue` statement is used to skip the remaining code block and start the next iteration of the loop. In this contract, the `continue` statement (line 10) will prevent the second if statement (line 12) from being executed.

break

The `break` statement is used to exit a loop. In this contract, the break statement (line 14) will cause the for loop to be terminated after the sixth iteration.

Watch a video tutorial on Loop statements.

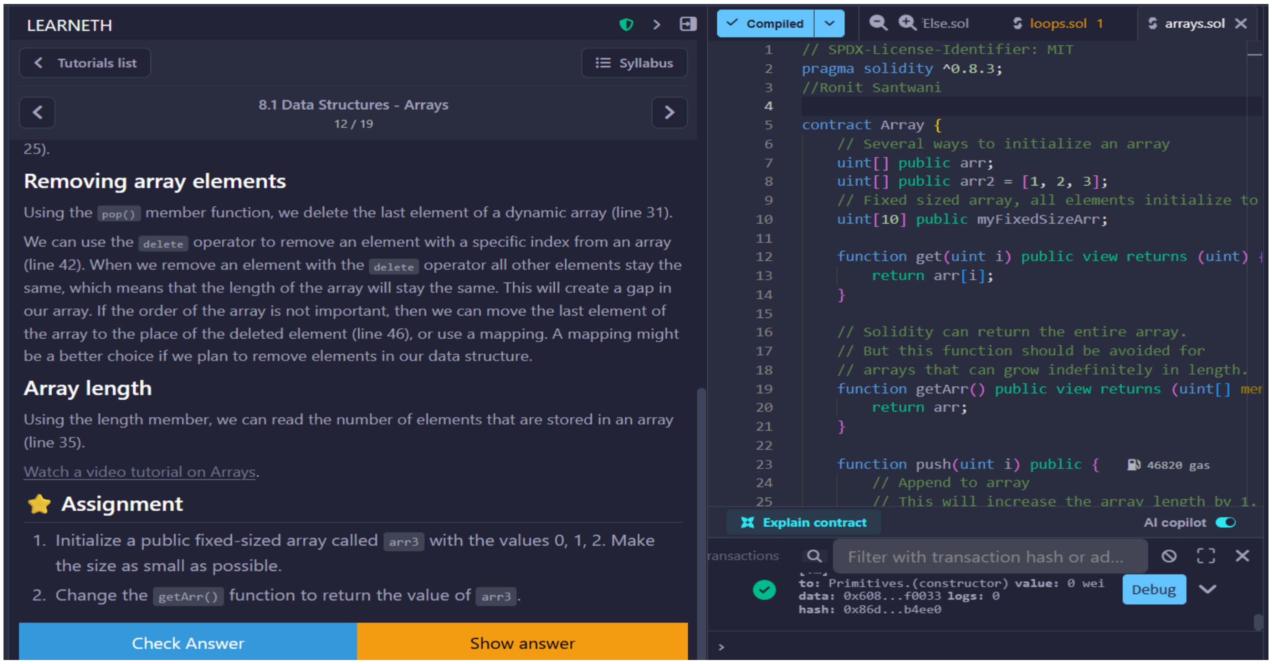
Assignment

1. Create a public `uint` state variable called count in the `Loop` contract.
2. At the end of the for loop, increment the count variable by 1.
3. Try to get the count variable to be equal to 9, but make sure you don't edit the `break` statement.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Loop {
6       function loop() public {    infinite gas
7           // for loop
8           for (uint i = 0; i < 10; i++) {
9               if (i == 3) {
10                  // Skip to next iteration with conti
11                  continue;
12              }
13              if (i == 5) {
14                  // Exit loop with break
15                  break;
16              }
17          }
18
19          // while loop
20          uint j;
21          while (j < 10) {
22              j++;
23          }
24      }
25  }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 12



8.1 Data Structures - Arrays
12 / 19

25).

Removing array elements

Using the `pop()` member function, we delete the last element of a dynamic array (line 31).

We can use the `delete` operator to remove an element with a specific index from an array (line 42). When we remove an element with the `delete` operator all other elements stay the same, which means that the length of the array will stay the same. This will create a gap in our array. If the order of the array is not important, then we can move the last element of the array to the place of the deleted element (line 46), or use a mapping. A mapping might be a better choice if we plan to remove elements in our data structure.

Array length

Using the length member, we can read the number of elements that are stored in an array (line 35).

Watch a video tutorial on Arrays.

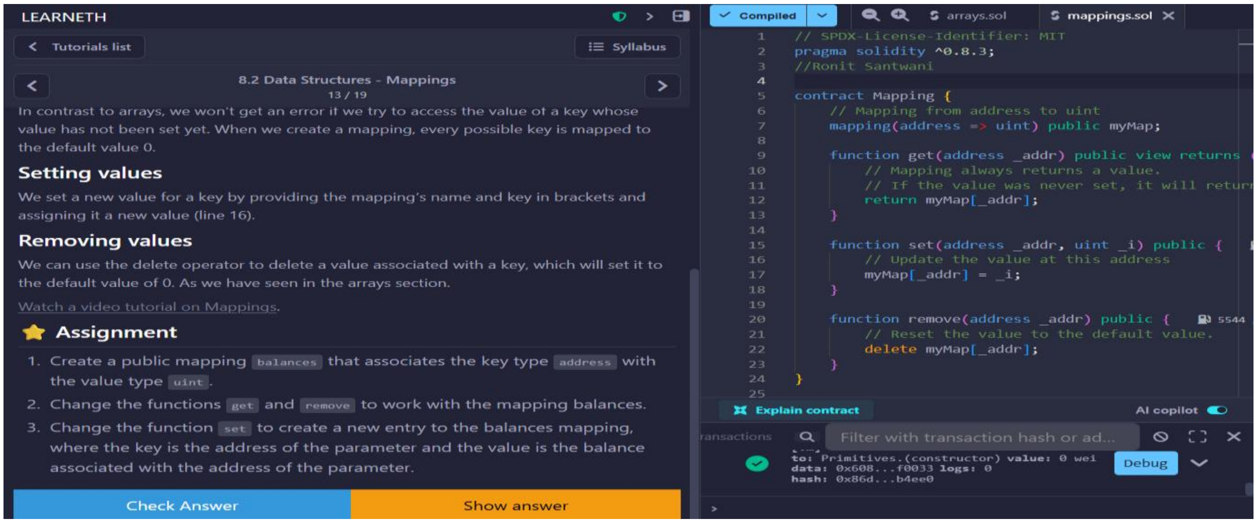
Assignment

1. Initialize a public fixed-sized array called `arr3` with the values 0, 1, 2. Make the size as small as possible.
2. Change the `getArr()` function to return the value of `arr3`.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Array {
6       // Several ways to initialize an array
7       uint[] public arr;
8       uint[] public arr2 = [1, 2, 3];
9       // Fixed sized array, all elements initialize to
10      uint[10] public myFixedSizeArr;
11
12      function get(uint i) public view returns (uint) {
13          return arr[i];
14      }
15
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[] mem
20          return arr;
21      }
22
23      function push(uint i) public {    46820 gas
24          // Append to array
25          // This will increase the array length by 1.
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 13



8.2 Data Structures - Mappings
13 / 19

In contrast to arrays, we won't get an error if we try to access the value of a key whose value has not been set yet. When we create a mapping, every possible key is mapped to the default value 0.

Setting values

We set a new value for a key by providing the mapping's name and key in brackets and assigning it a new value (line 16).

Removing values

We can use the delete operator to delete a value associated with a key, which will set it to the default value of 0. As we have seen in the arrays section.

Watch a video tutorial on Mappings.

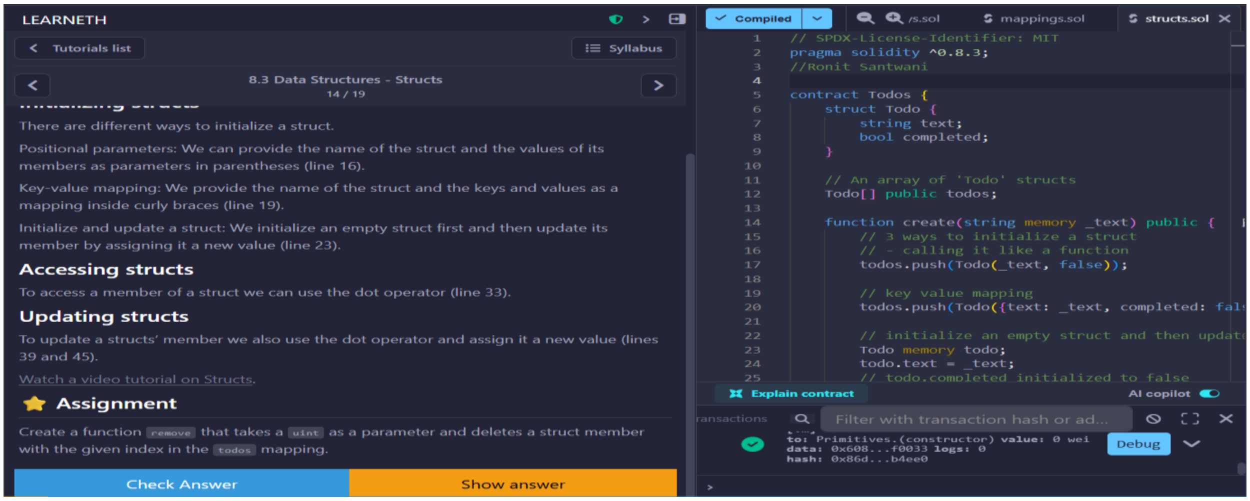
Assignment

1. Create a public mapping `balances` that associates the key type `address` with the value type `uint`.
2. Change the functions `get` and `remove` to work with the mapping balances.
3. Change the function `set` to create a new entry to the balances mapping, where the key is the address of the parameter and the value is the balance associated with the address of the parameter.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Mapping {
6       // Mapping from address to uint
7       mapping(address => uint) public myMap;
8
9       function get(address _addr) public view returns (
10          // Mapping always returns a value.
11          // If the value was never set, it will return
12          return myMap[_addr];
13      }
14
15      function set(address _addr, uint _i) public {
16          // Update the value at this address
17          myMap[_addr] = _i;
18      }
19
20      function remove(address _addr) public {    5544
21          // Reset the value to the default value.
22          delete myMap[_addr];
23      }
24  }
25
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 14



8.3 Data Structures - Structs
14 / 19

Initializing structs

There are different ways to initialize a struct.

Positional parameters: We can provide the name of the struct and the values of its members as parameters in parentheses (line 16).

Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 19).

Initialize and update a struct: We initialize an empty struct first and then update its member by assigning it a new value (line 23).

Accessing structs

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

Updating structs

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

Watch a video tutorial on Structs.

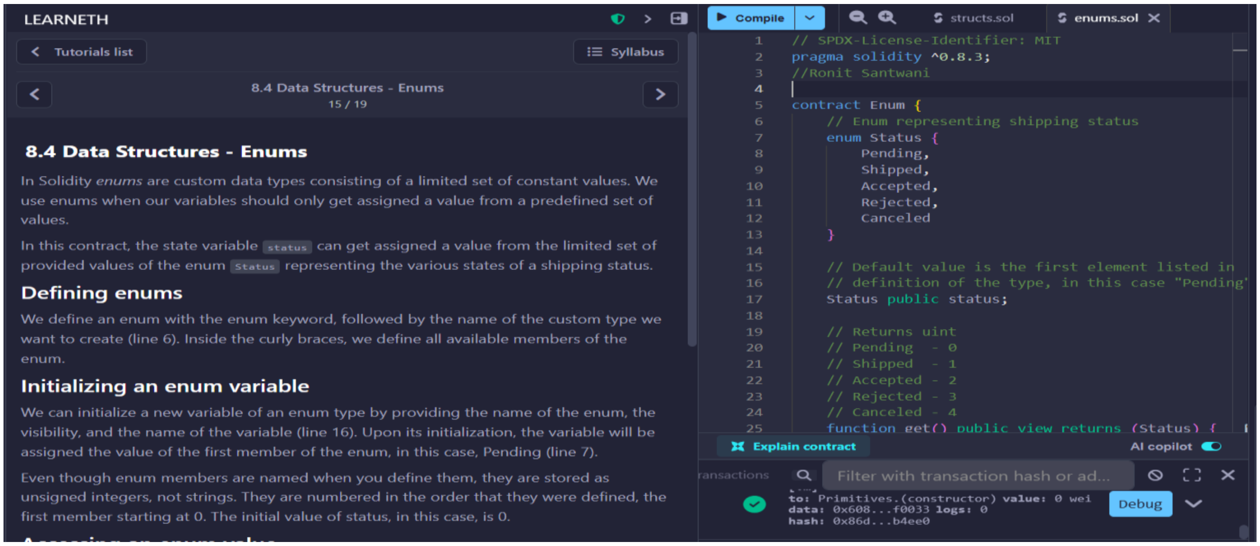
Assignment

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

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4
5   contract Todos {
6       struct Todo {
7           string text;
8           bool completed;
9       }
10
11      // An array of 'Todo' structs
12      Todo[] public todos;
13
14      function create(string memory _text) public {
15          // 3 ways to initialize a struct
16          // - calling it like a function
17          todos.push(Todo(_text, false));
18
19          // key value mapping
20          todos.push(Todo({text: _text, completed: fal
21
22          // initialize an empty struct and then updat
23          Todo memory todo;
24          todo.text = _text;
25          // todo.completed initialized to false
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 15



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 enum.

Initializing an enum variable

We can initialize a new variable of an enum type by providing the name of the enum, the visibility, and the name of the variable (line 16). Upon its initialization, the variable will be assigned the value of the first member of the enum, in this case, Pending (line 7).

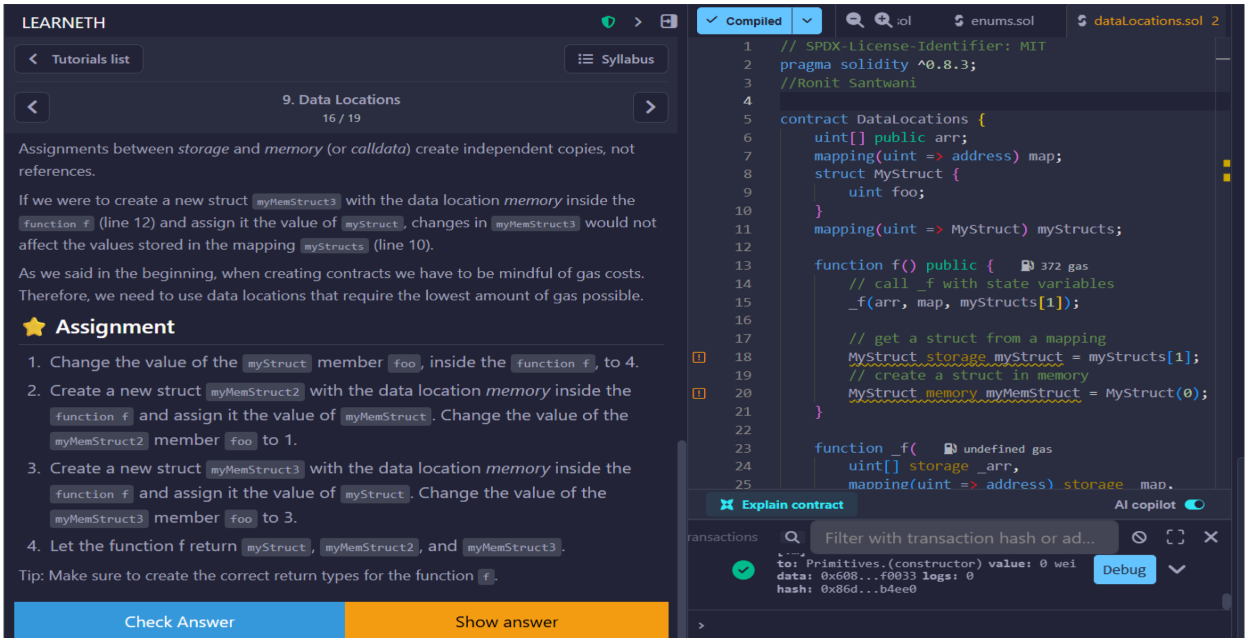
Even though enum members are named when you define them, they are stored as unsigned integers, not strings. They are numbered in the order that they were defined, the first member starting at 0. The initial value of status, in this case, is 0.

Accessing an enum value

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4   
5   contract Enum {
6       // Enum representing shipping status
7       enum Status {
8           Pending,
9           Shipped,
10          Accepted,
11          Rejected,
12          Canceled
13      }
14  
15      // Default value is the first element listed in
16      // definition of the type, in this case "Pending'
17      Status public status;
18  
19      // Returns uint
20      // Pending  - 0
21      // Shipped  - 1
22      // Accepted - 2
23      // Rejected - 3
24      // Canceled - 4
25      function get() public view returns (Status) {
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 16



Assignments between *storage* and *memory* (or *calldata*) create independent copies, not references.

If we were to create a new struct `myMemStruct3` with the data location *memory* inside the `function f` (line 12) and assign it the value of `myStruct`, changes in `myMemStruct3` would not affect the values stored in the mapping `myStructs` (line 10).

As we said in the beginning, when creating contracts we have to be mindful of gas costs. Therefore, we need to use data locations that require the lowest amount of gas possible.

⭐ Assignment

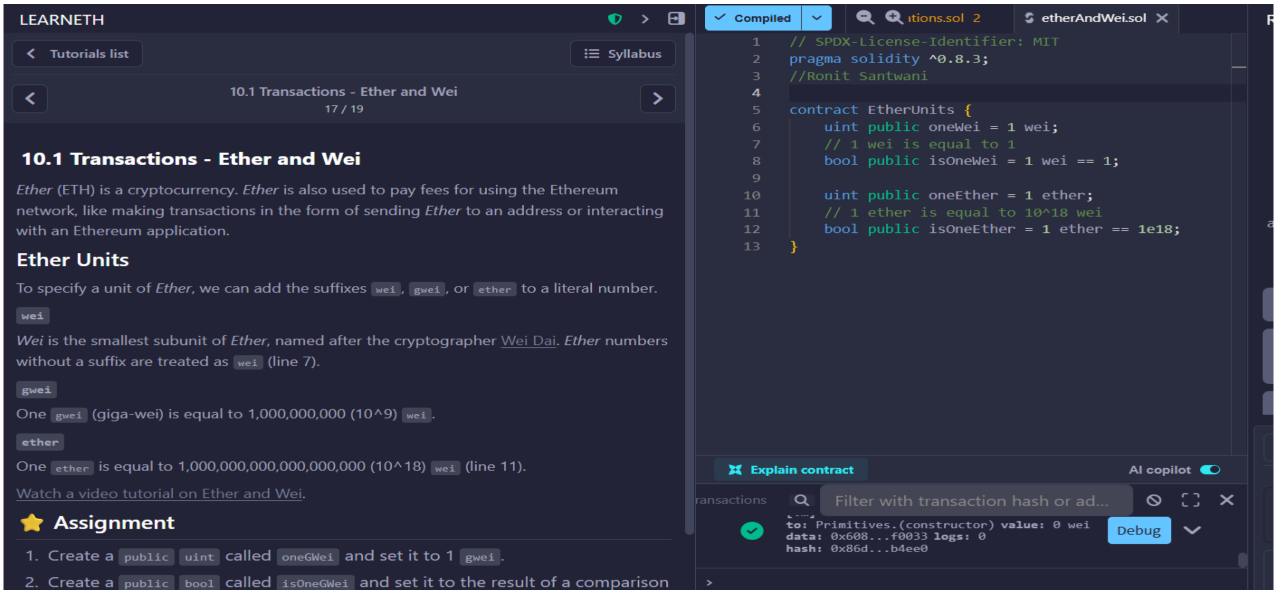
1. Change the value of the `myStruct` member `foo`, inside the `function f`, to 4.
2. Create a new struct `myMemStruct2` with the data location *memory* inside the `function f` and assign it the value of `myMemStruct`. Change the value of the `myMemStruct2` member `foo` to 1.
3. Create a new struct `myMemStruct3` with the data location *memory* inside the `function f` and assign it the value of `myStruct`. Change the value of the `myMemStruct3` member `foo` to 3.
4. Let the function f return `myStruct`, `myMemStruct2`, and `myMemStruct3`.

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

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4   
5   contract DataLocations {
6       uint[] public arr;
7       mapping(uint => address) map;
8       struct MyStruct {
9           uint foo;
10      }
11      mapping(uint => MyStruct) myStructs;
12  
13      function f() public {    372 gas
14          // call _f with state variables
15          _f(arr, map, myStructs[1]);
16  
17          // get a struct from a mapping
18          MyStruct storage myStruct = myStructs[1];
19          // create a struct in memory
20          MyStruct memory myMemStruct = MyStruct(0);
21      }
22  
23      function _f(    undefined gas
24          uint[] storage _arr,
25          mapping(uint => address) storage _map,
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 17



10.1 Transactions - Ether and Wei

Ether (ETH) is a cryptocurrency. *Ether* is also used to pay fees for using the Ethereum network, like making transactions in the form of sending *Ether* to an address or interacting with an Ethereum application.

Ether Units

To specify a unit of *Ether*, we can add the suffixes `wei`, `gwei`, or `ether` to a literal 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.

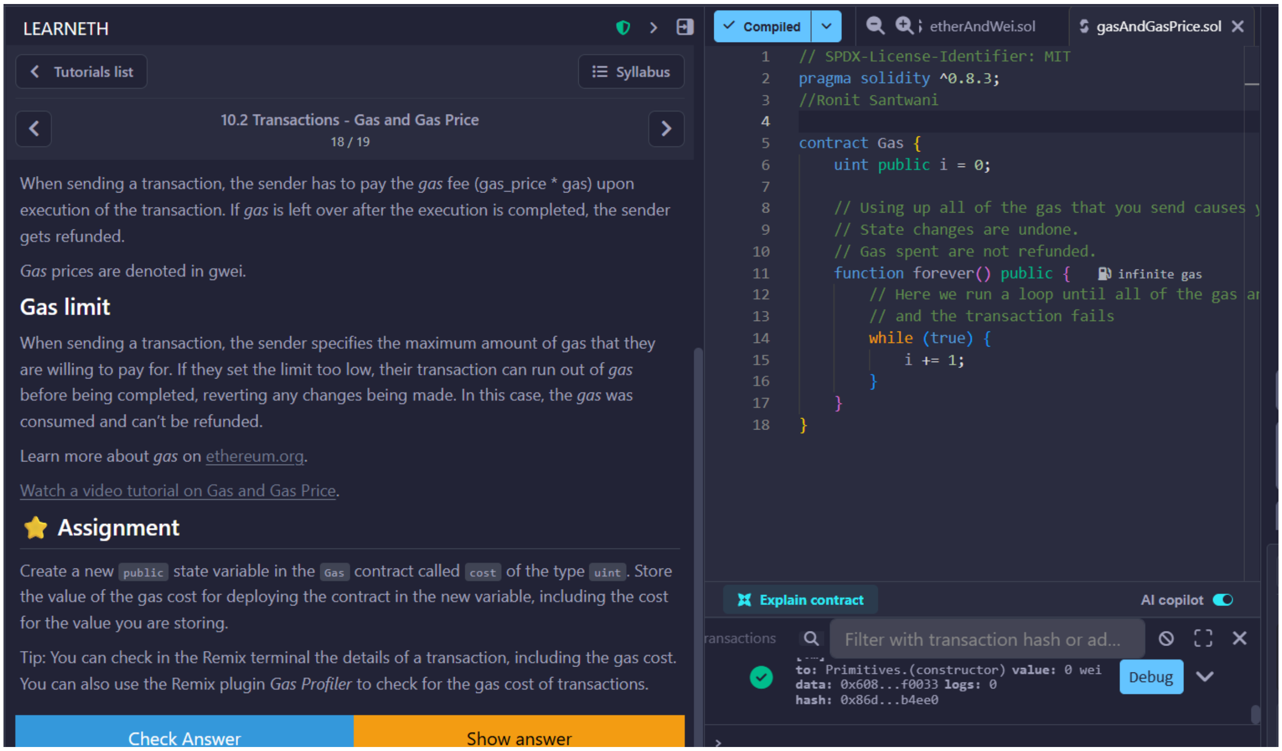
⭐ Assignment

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

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   //Ronit Santwani
4   
5   contract EtherUnits {
6       uint public oneWei = 1 wei;
7       // 1 wei is equal to 1
8       bool public isOneWei = 1 wei == 1;
9   
10      uint public oneEther = 1 ether;
11      // 1 ether is equal to 10^18 wei
12      bool public isOneEther = 1 ether == 1e18;
13  }
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 18



10.2 Transactions - Gas and Gas Price
18 / 19

When sending a transaction, the sender has to pay the *gas* fee (gas_price * gas) upon execution of the transaction. If *gas* is left over after the execution is completed, the sender gets refunded.

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

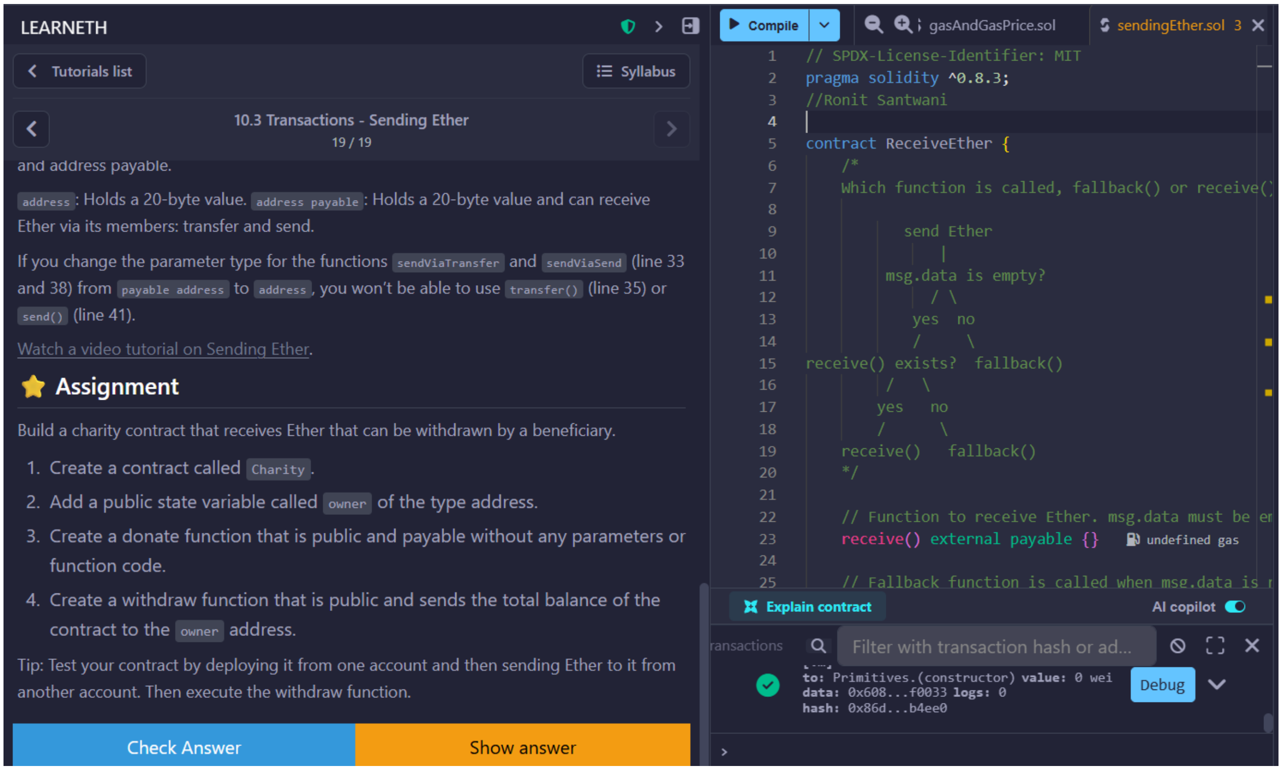
Create a new `public` state variable in the `Gas` contract called `cost` of the type `uint`. Store the value of the gas cost for deploying the contract in the new variable, including the cost for the value you are storing.

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.

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

contract Gas {
    uint public i = 0;

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

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

- Tutorial no. 19



10.3 Transactions - Sending Ether
19 / 19

and address payable.

`address`: Holds a 20-byte value. `address payable`: Holds a 20-byte value and can receive Ether via its members: transfer and send.

If you change the parameter type for the functions `sendViaTransfer` and `sendViaSend` (line 33 and 38) from `payable address` to `address`, you won't be able to use `transfer()` (line 35) or `send()` (line 41).

Watch a video tutorial on Sending Ether.

⭐ Assignment

Build a charity contract that receives Ether that can be withdrawn by a beneficiary.

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.

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

contract ReceiveEther {
    /*
    Which function is called, fallback() or receive(

           send Ether
               |
         msg.data is empty?
              / \
            yes  no
            /     \
receive() exists?  fallback()
        /   \
       yes   no
       /      \
    receive()   fallback()
    */

    // Function to receive Ether. msg.data must be e
    receive() external payable {}    undefined gas

    // Fallback function is called when msg.data is
```

```
to: Primitives.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0
hash: 0x86d...b4ee0
```

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.