Ria Chaudhari D20A14

Blockchain Experiment 4

AIM: Hands on Solidity Programming Assignments for creating Smart Contracts

THEORY:

Q1: 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

● 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.

Q2: 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.

Q3: 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 contracts.

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.

Q4: 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.

Q5: 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.

Example: 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 }.

Q6: 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.

Q7: 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.

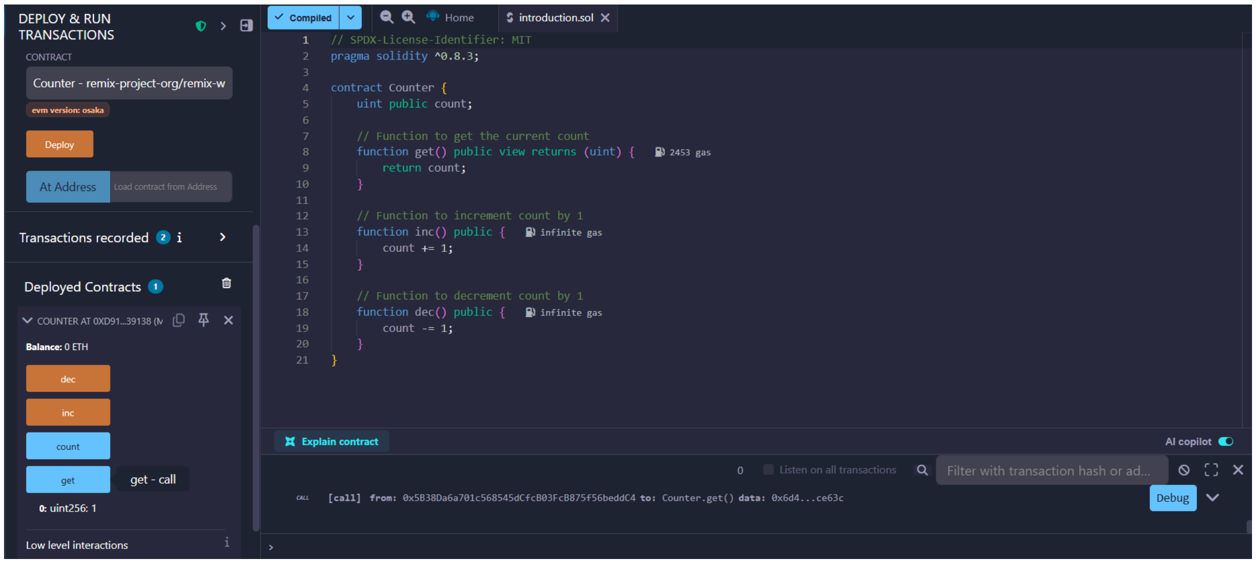
● Gas and Gas Price: Every transaction consumes gas, which represents computational effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.

● 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.

TASKS PERFORMED:

Tutorial 1: Introduction

a)get



DEPLOY & RUN TRANSACTIONS

CONTRACT: Counter - remix-project-org/remix-w

evm version: osaka

Transactions recorded 2

Deployed Contracts 1

COUNTER AT 0XD91...39138

Balance: 0 ETH

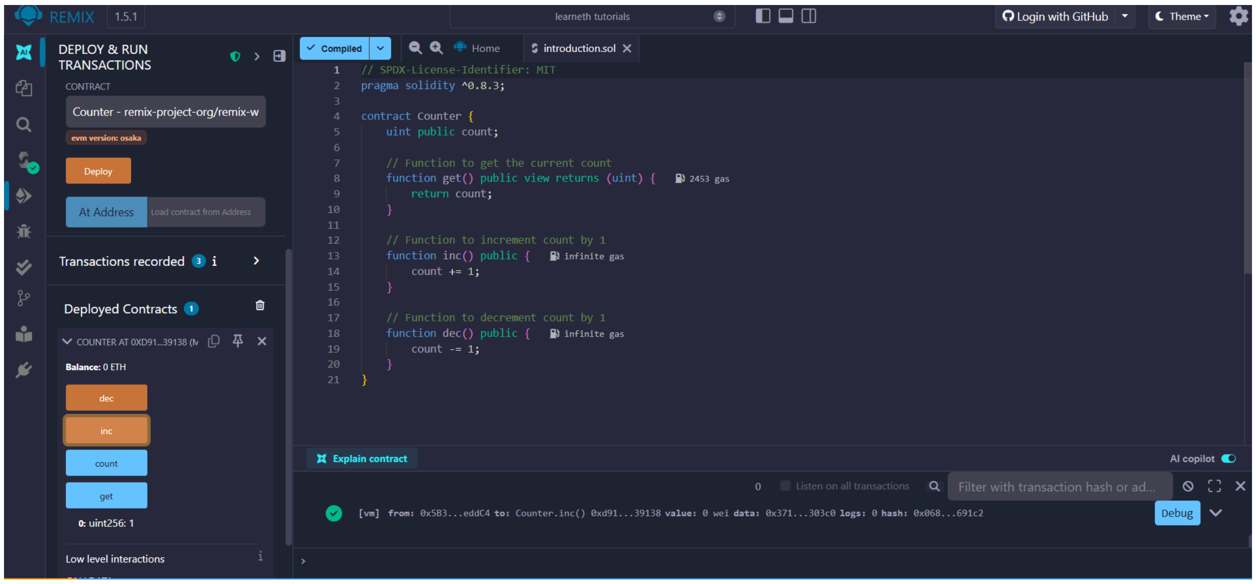
get - call

0: uint256: 1

```solidity
// 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) {    2453 gas
        return count;
    }

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

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

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

b)inc



DEPLOY & RUN TRANSACTIONS

CONTRACT: Counter - remix-project-org/remix-w

evm version: osaka

Transactions recorded 3

Deployed Contracts 1

COUNTER AT 0XD91...39138

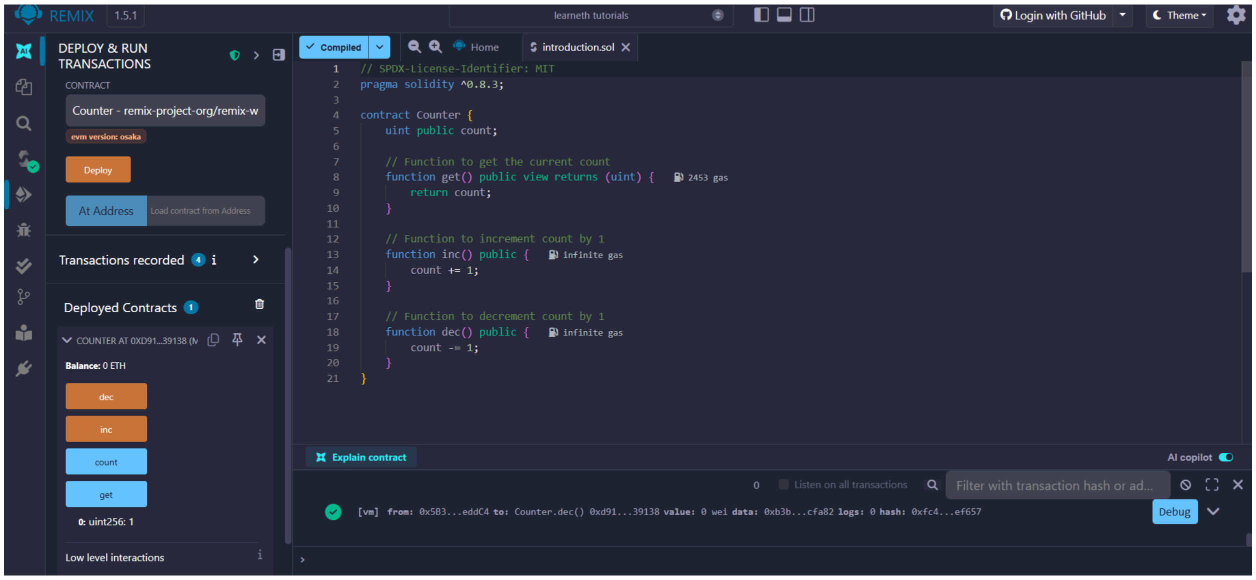
Balance: 0 ETH

0: uint256: 1

```solidity
// 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) {    2453 gas
        return count;
    }

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

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

```
[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0x068...691c2
```

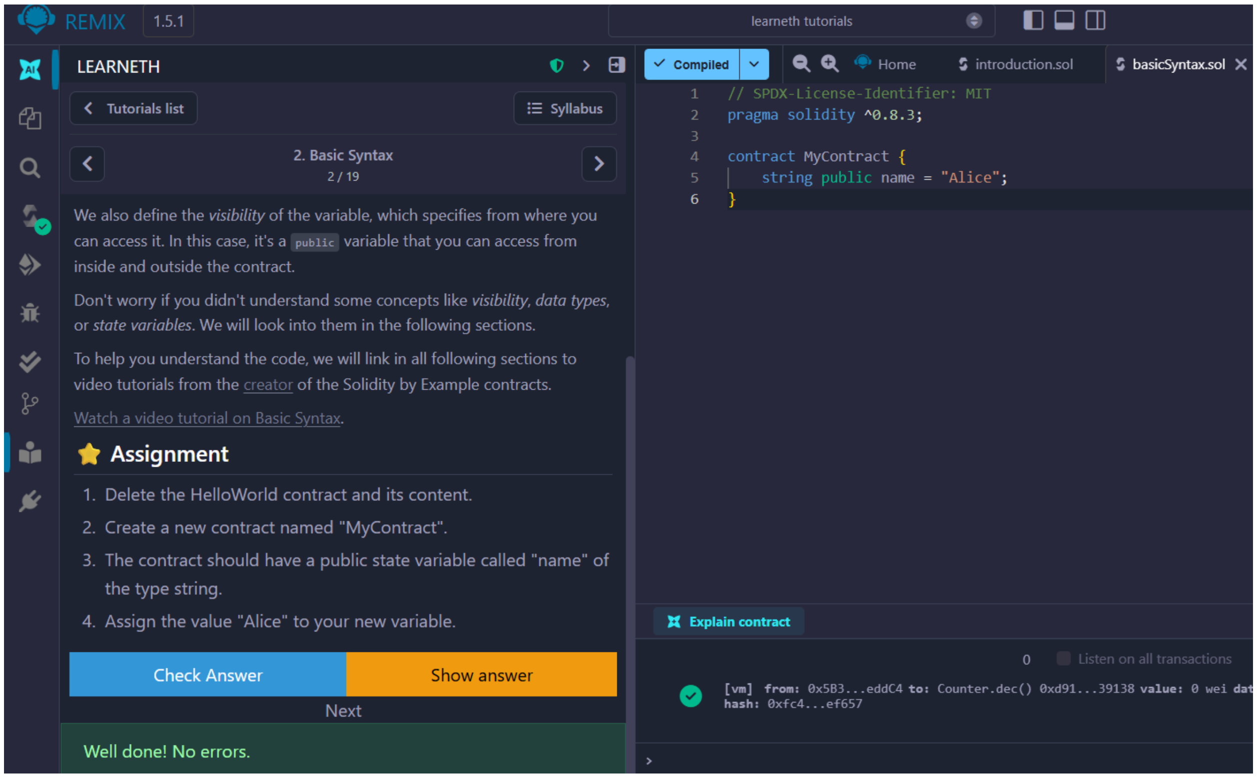
c)dec



```
// 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) {    2453 gas
        return count;
    }

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

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

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0 hash: 0xfc4...ef657
```

Tutorial 2:Basic Syntax



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 from the creator of the Solidity by Example contracts.

Watch a video tutorial on Basic Syntax.

Assignment

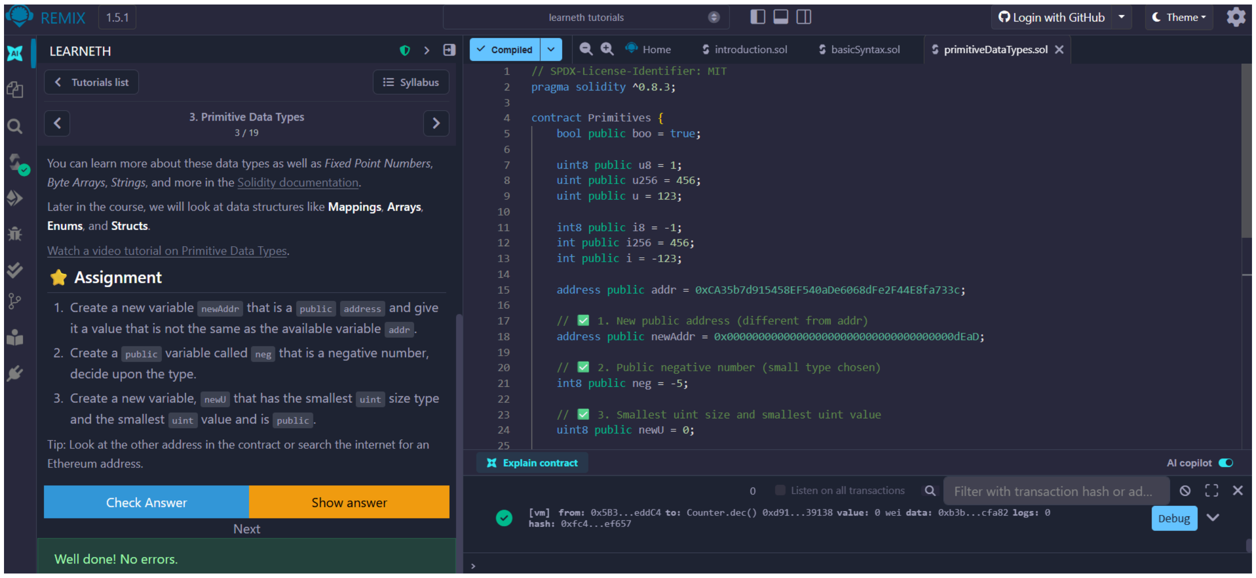
1. Delete the HelloWorld contract and its content.
2. Create a new contract named "MyContract".
3. The contract should have a public state variable called "name" of the type string.
4. Assign the value "Alice" to your new variable.

Well done! No errors.

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

contract MyContract {
    string public name = "Alice";
}
```

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei dat
hash: 0xfc4...ef657
```

Tutorial 3:Primitive Data Types



3. Primitive Data Types
3 / 19

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.

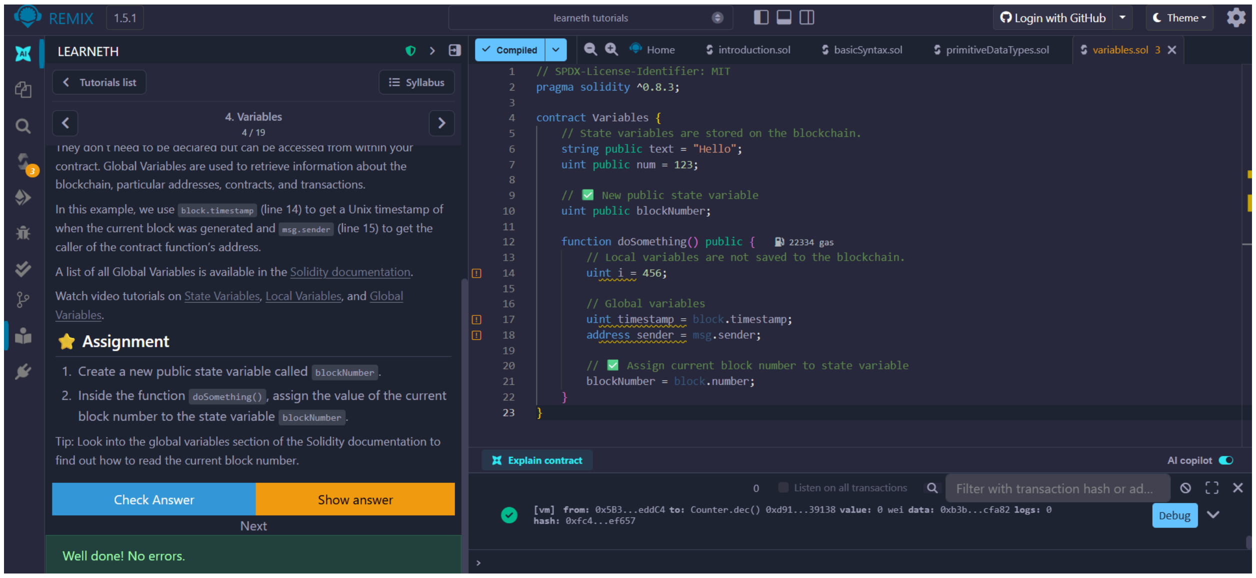
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract Primitives {
    bool public boo = true;

    uint8 public u8 = 1;
    uint public u256 = 456;
    uint public u = 123;

    int8 public i8 = -1;
    int public i256 = 456;
    int public i = -123;

    address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

    // ✅ 1. New public address (different from addr)
    address public newAddr = 0x000000000000000000000000000000000000dEaD;

    // ✅ 2. Public negative number (small type chosen)
    int8 public neg = -5;

    // ✅ 3. Smallest uint size and smallest uint value
    uint8 public newU = 0;

```

Check Answer
Show answer
Next
Explain contract
AI copilot
Listen on all transactions
Filter with transaction hash or ad...
Debug

Well done! No errors.

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Tutorial 4:Variables



4. Variables
4 / 19

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.

In this example, we use `block.timestamp` (line 14) to get a Unix timestamp of when the current block was generated and `msg.sender` (line 15) to get the caller of the contract function's address.

A list of all Global Variables is available in the Solidity documentation.

Watch video tutorials on State Variables, Local Variables, and Global Variables.

Assignment

1. Create a new public state variable called `blockNumber`.
2. Inside the function `doSomething()`, assign the value of the current block number to the state variable `blockNumber`.

Tip: Look into the global variables section of the Solidity documentation to find out how to read the current block number.

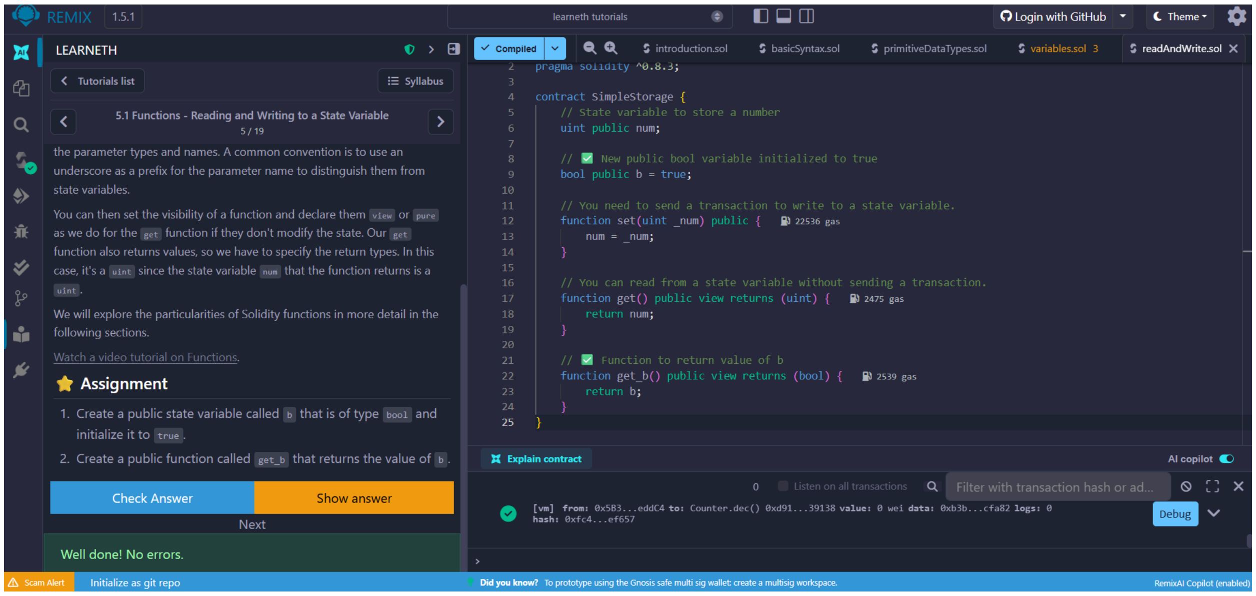
```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   contract Variables {
5       // State variables are stored on the blockchain.
6       string public text = "Hello";
7       uint public num = 123;
8
9       // ✅ New public state variable
10      uint public blockNumber;
11
12      function doSomething() public {     22334 gas
13          // Local variables are not saved to the blockchain.
14          uint i = 456;
15
16          // Global variables
17          uint timestamp = block.timestamp;
18          address sender = msg.sender;
19
20          // ✅ Assign current block number to state variable
21          blockNumber = block.number;
22      }
23  }
```

Check Answer
Show answer
Next
Explain contract
AI copilot
Listen on all transactions
Filter with transaction hash or ad...
Debug

Well done! No errors.

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Tutorial 5: Functions - Reading and Writing to a State Variable



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

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`.

Check Answer | Show answer

Next

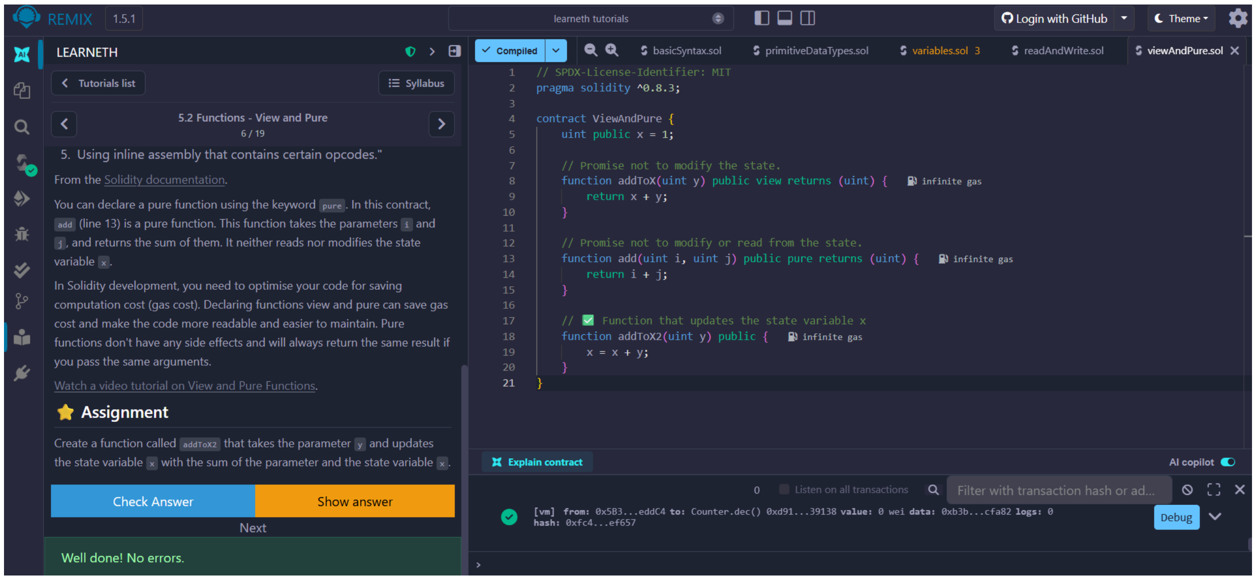
Well done! No errors.

```
pragma solidity ^0.8.3;

contract SimpleStorage {
    // State variable to store a number
    uint public num;

    // ✅ New public bool variable initialized to true
    bool public b = true;

    // You need to send a transaction to write to a state variable.
    function set(uint _num) public {    22536 gas
        num = _num;
    }

    // You can read from a state variable without sending a transaction.
    function get() public view returns (uint) {    2475 gas
        return num;
    }

    // ✅ Function to return value of b
    function get_b() public view returns (bool) {    2539 gas
        return b;
    }
}
```

```
0
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Did you know? To prototype using the Gnosis safe multi sig wallet: create a multisig workspace.

Tutorial 6: Functions - View and Pure



5.2 Functions - View and Pure
6 / 19

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`.

Check Answer | Show answer

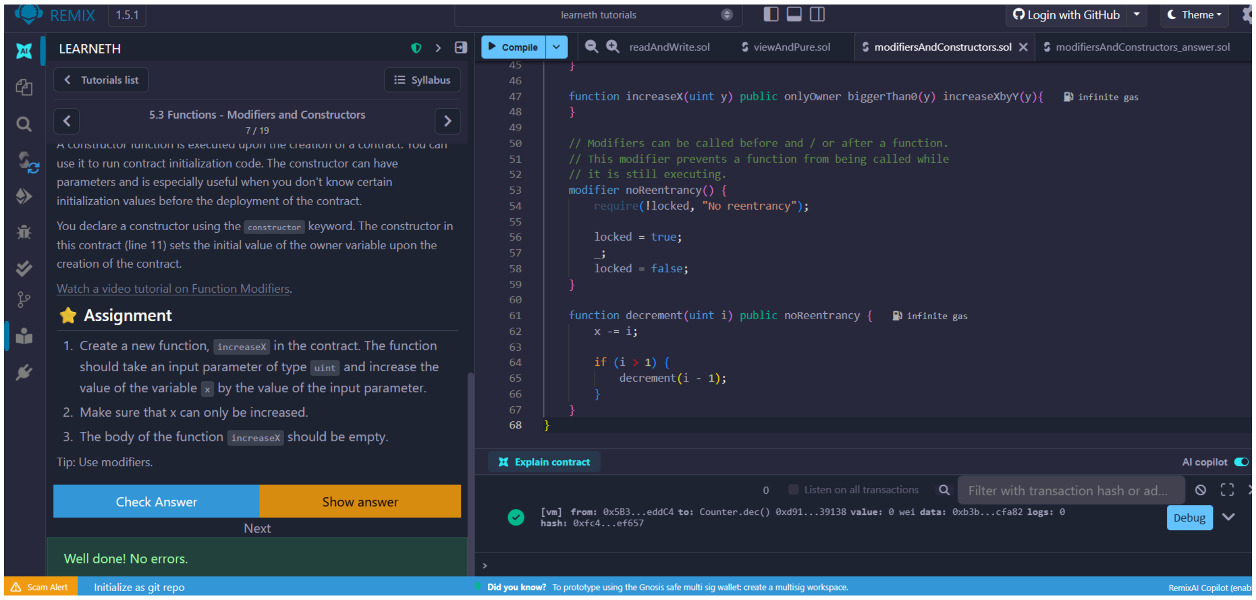
Next

Well done! No errors.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   contract ViewAndPure {
5       uint public x = 1;
6
7       // Promise not to modify the state.
8       function addToX(uint y) public view returns (uint) {    infinite gas
9           return x + y;
10      }
11
12      // Promise not to modify or read from the state.
13      function add(uint i, uint j) public pure returns (uint) {    infinite gas
14          return i + j;
15      }
16
17      // ✅ Function that updates the state variable x
18      function addToX2(uint y) public {    infinite gas
19          x = x + y;
20      }
21  }
```

```
0
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Tutorial 7:Modifiers and Constructors



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.

Well done! No errors.

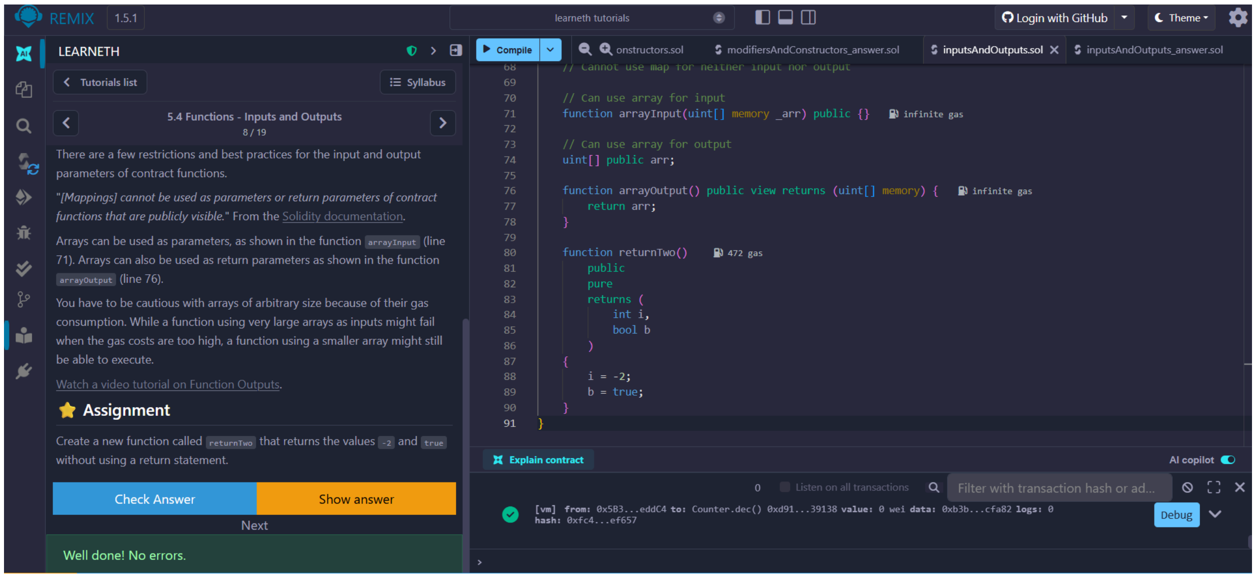
```
    }

    function increaseX(uint y) public onlyOwner biggerThan0(y) increaseXbyY(y){    infinite gas
    }

    // Modifiers can be called before and / or after a function.
    // This modifier prevents a function from being called while
    // it is still executing.
    modifier noReentrancy() {
        require(!locked, "No reentrancy");

        locked = true;
        _;
        locked = false;
    }

    function decrement(uint i) public noReentrancy {    infinite gas
        x -= i;

        if (i > 1) {
            decrement(i - 1);
        }
    }
}
```

Explain contract
AI copilot
0
Listen on all transactions
Filter with transaction hash or ad...
Debug

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Did you know? To prototype using the Gnosis safe multi sig wallet: create a multisig workspace.

Tutorial 8:Inputs and Outputs



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.

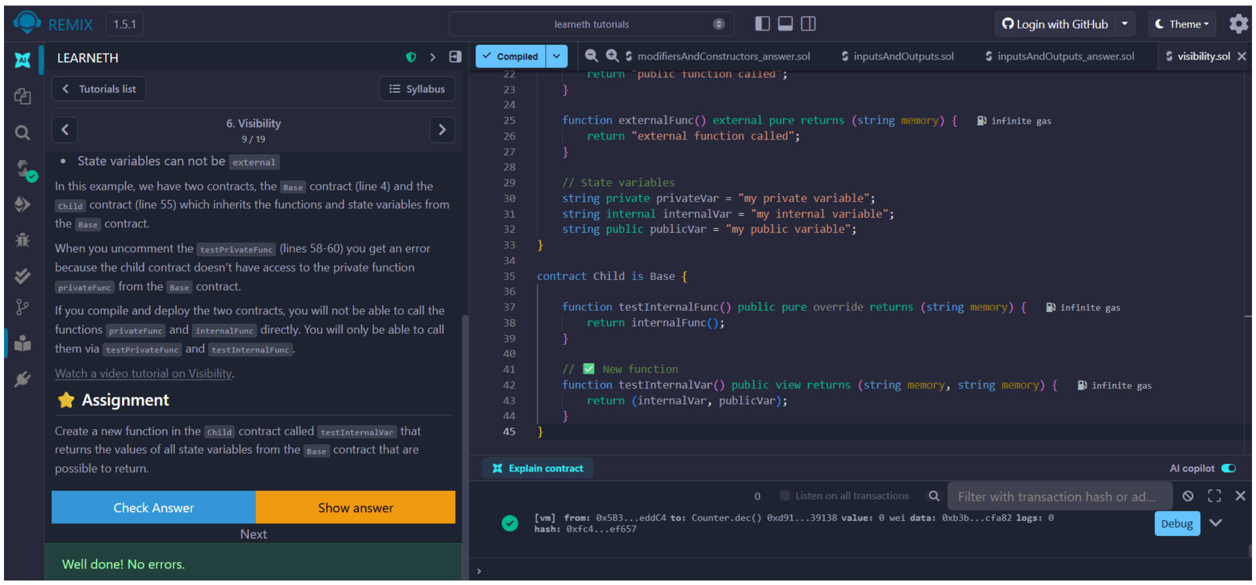
Well done! No errors.

```
68      // Cannot use map for neither input nor output
69
70      // Can use array for input
71      function arrayInput(uint[] memory _arr) public {}    infinite gas
72
73      // Can use array for output
74      uint[] public arr;
75
76      function arrayOutput() public view returns (uint[] memory) {    infinite gas
77          return arr;
78      }
79
80      function returnTwo()    472 gas
81          public
82          pure
83          returns (
84              int i,
85              bool b
86          )
87      {
88          i = -2;
89          b = true;
90      }
91  }
```

Explain contract
AI copilot
0
Listen on all transactions
Filter with transaction hash or ad...
Debug

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Tutorial 9:Visibility



6. Visibility
9 / 19

- 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.

Check Answer | Show answer

Next

Well done! No errors.

```
22          return "public function called";
23      }
24
25      function externalFunc() external pure returns (string memory) {    infinite gas
26          return "external function called";
27      }
28
29      // State variables
30      string private privateVar = "my private variable";
31      string internal internalVar = "my internal variable";
32      string public publicVar = "my public variable";
33  }
34
35  contract Child is Base {
36
37      function testInternalFunc() public pure override returns (string memory) {    infinite gas
38          return internalFunc();
39      }
40
41      // ✅ New function
42      function testInternalVar() public view returns (string memory, string memory) {    infinite gas
43          return (internalVar, publicVar);
44      }
45  }
```

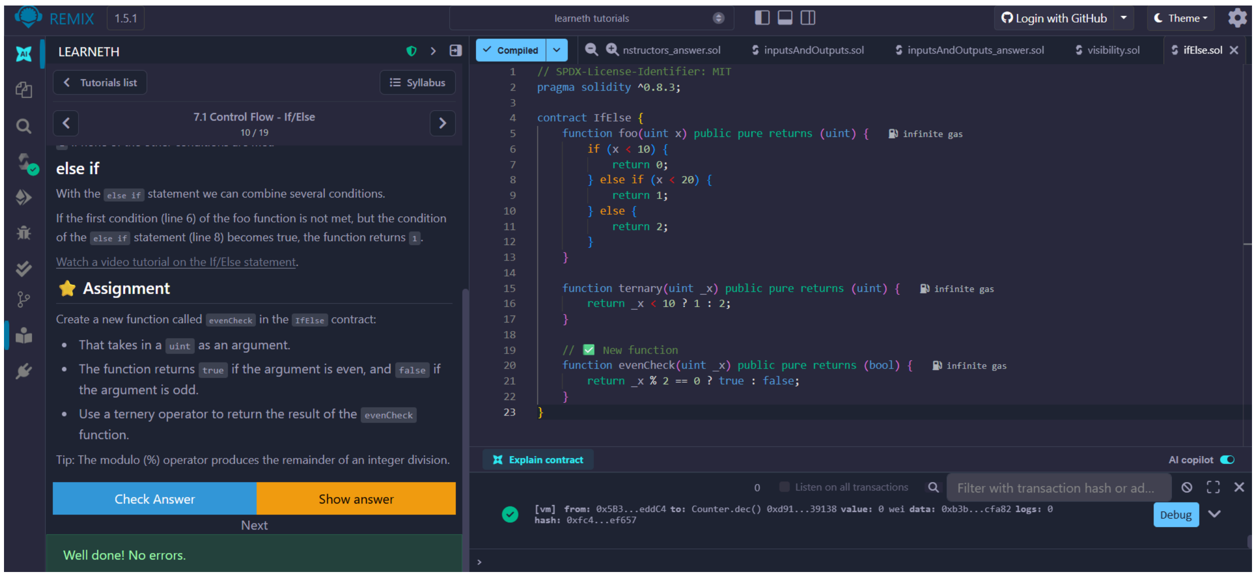
Explain contract   AI copilot

0   Listen on all transactions   Filter with transaction hash or ad...

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Debug

Tutorial 10: Control Flow-If/Else



7.1 Control Flow - If/Else
10 / 19

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` statement (line 8) becomes true, the function returns `1`.

Watch a video tutorial on the If/Else statement.

⭐ **Assignment**

Create a new function called `evenCheck` in the `IfElse` contract:

- That takes in a `uint` as an argument.
- The function returns `true` if the argument is even, and `false` if the argument is odd.
- Use a ternery operator to return the result of the `evenCheck` function.

Tip: The modulo (%) operator produces the remainder of an integer division.

Check Answer | Show answer

Next

Well done! No errors.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   contract IfElse {
5       function foo(uint x) public pure returns (uint) {    infinite gas
6           if (x < 10) {
7               return 0;
8           } else if (x < 20) {
9               return 1;
10          } else {
11              return 2;
12          }
13      }
14
15      function ternary(uint _x) public pure returns (uint) {    infinite gas
16          return _x < 10 ? 1 : 2;
17      }
18
19      // ✅ New function
20      function evenCheck(uint _x) public pure returns (bool) {    infinite gas
21          return _x % 2 == 0 ? true : false;
22      }
23  }
```

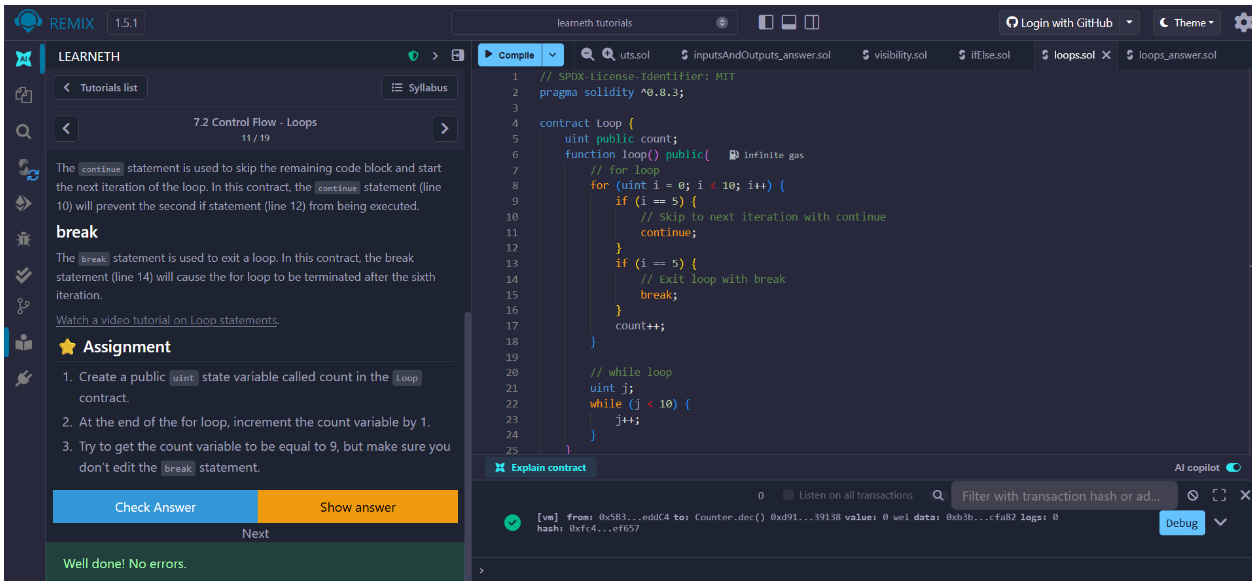
Explain contract   AI copilot

0   Listen on all transactions   Filter with transaction hash or ad...

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Debug

Tutorial 11:Loops



REMIX 1.5.1 · learneth tutorials · Login with GitHub · Theme

LEARNETH · Tutorials list · Syllabus

7.2 Control Flow - Loops
11 / 19

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.

Check Answer · Show answer · Next

Well done! No errors.

Compile · uts.sol · inputsAndOutputs_answer.sol · visibility.sol · ifElse.sol · loops.sol · loops_answer.sol

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

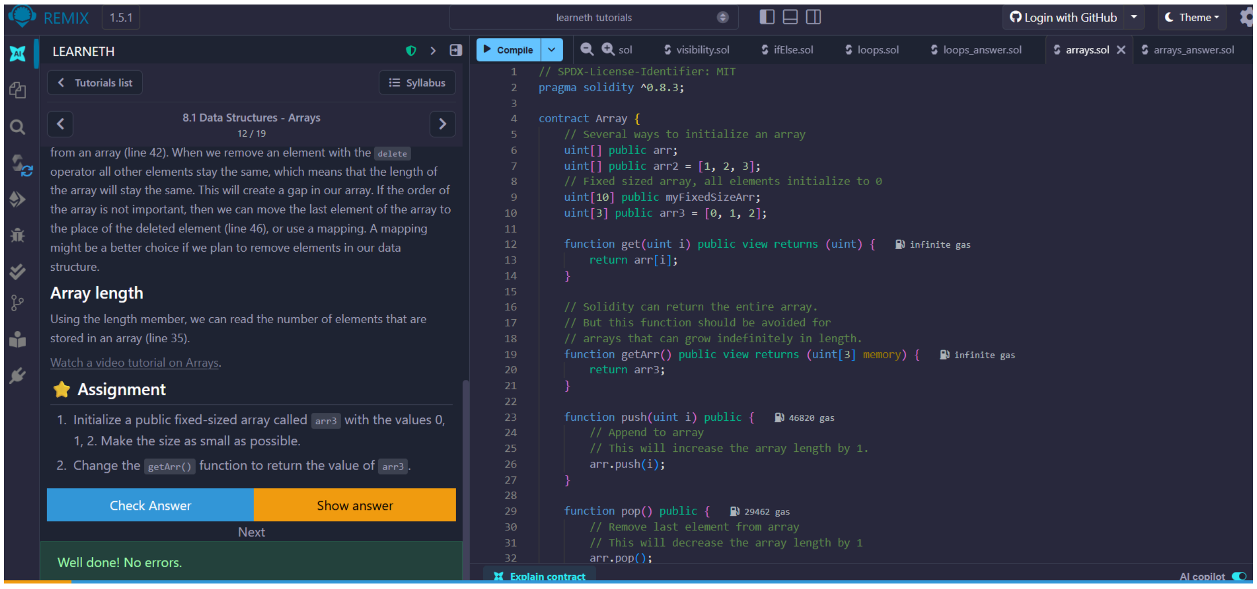
Explain contract · AI copilot

0 · Listen on all transactions · Filter with transaction hash or ad...

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0
hash: 0xfc4...ef657
```

Debug

Tutorial 12:Data Structures:Arrays



REMIX 1.5.1 · learneth tutorials · Login with GitHub · Theme

LEARNETH · Tutorials list · Syllabus

8.1 Data Structures - Arrays
12 / 19

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`.

Check Answer · Show answer · Next

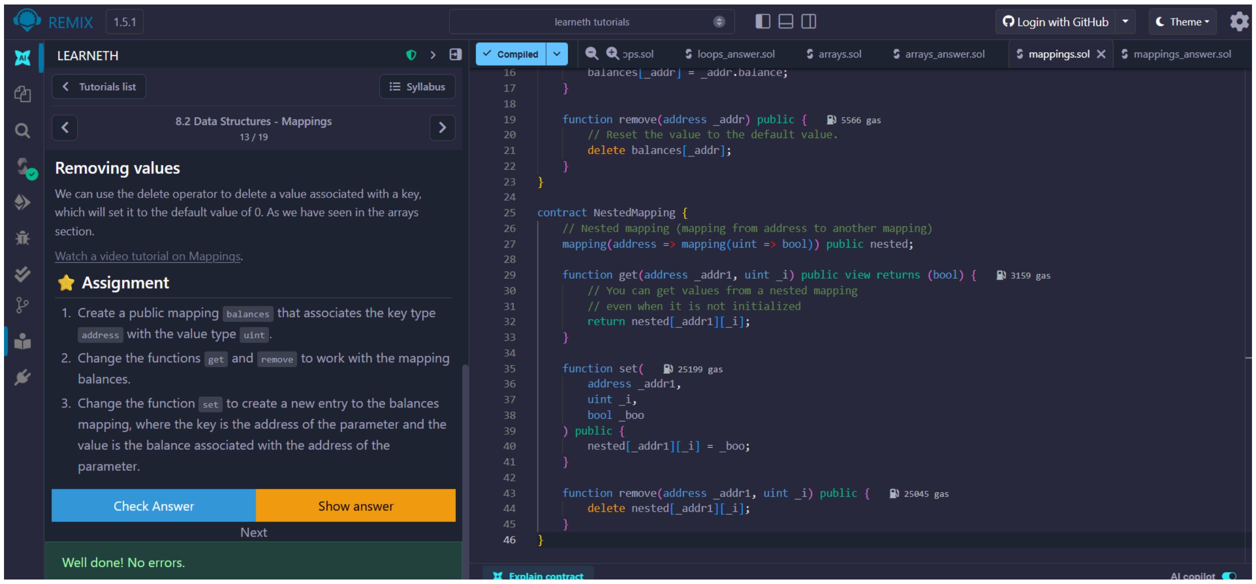
Well done! No errors.

Compile · sol · visibility.sol · ifElse.sol · loops.sol · loops_answer.sol · arrays.sol · arrays_answer.sol

```
 1  // SPDX-License-Identifier: MIT
 2  pragma solidity ^0.8.3;
 3
 4  contract Array {
 5      // Several ways to initialize an array
 6      uint[] public arr;
 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 get(uint i) public view returns (uint) {    infinite gas
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[3] memory) {    infinite gas
20          return arr3;
21      }
22
23      function push(uint i) public {    46820 gas
24          // Append to array
25          // This will increase the array length by 1.
26          arr.push(i);
27      }
28
29      function pop() public {    29462 gas
30          // Remove last element from array
31          // This will decrease the array length by 1
32          arr.pop();
```

Explain contract · AI copilot

Tutorial 13:Data Structures:Mappings



8.2 Data Structures - Mappings
13 / 19

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.

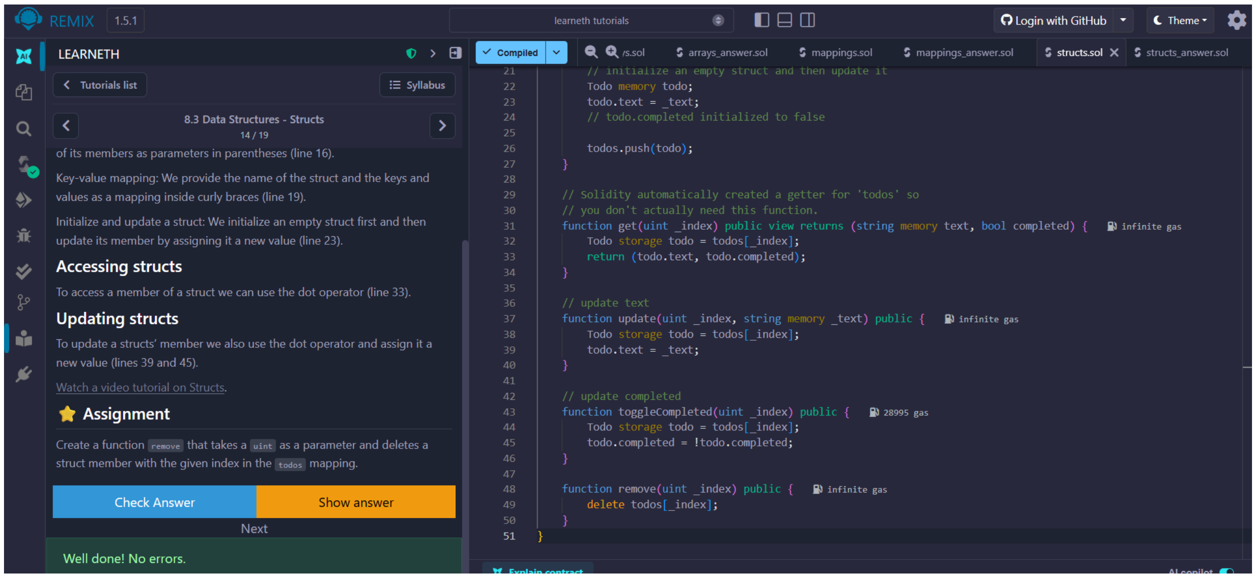
Check Answer | Show answer

Next

Well done! No errors.

```
        balances[_addr] = _addr.balance;
    }

    function remove(address _addr) public {    5566 gas
        // 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 _addr1, uint _i) public view returns (bool) {    3159 gas
        // You can get values from a nested mapping
        // even when it is not initialized
        return nested[_addr1][_i];
    }

    function set(    25199 gas
        address _addr1,
        uint _i,
        bool _boo
    ) public {
        nested[_addr1][_i] = _boo;
    }

    function remove(address _addr1, uint _i) public {    25045 gas
        delete nested[_addr1][_i];
    }
}
```

Tutorial 14:Data Structures:Structs



8.3 Data Structures - Structs
14 / 19

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.

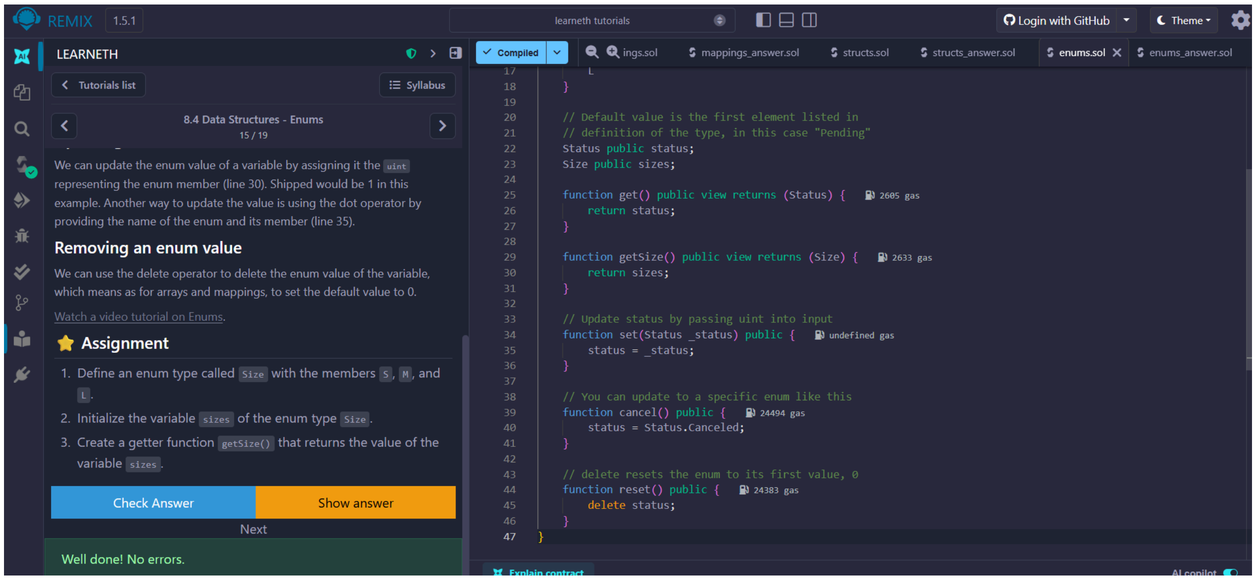
Check Answer | Show answer

Next

Well done! No errors.

```
21          // initialize an empty struct and then update it
22          Todo memory todo;
23          todo.text = _text;
24          // todo.completed initialized to false
25  
26          todos.push(todo);
27      }
28  
29      // Solidity automatically created a getter for 'todos' so
30      // you don't actually need this function.
31      function get(uint _index) public view returns (string memory text, bool completed) {    infinite gas
32          Todo storage todo = todos[_index];
33          return (todo.text, todo.completed);
34      }
35  
36      // update text
37      function update(uint _index, string memory _text) public {    infinite gas
38          Todo storage todo = todos[_index];
39          todo.text = _text;
40      }
41  
42      // update completed
43      function toggleCompleted(uint _index) public {    28995 gas
44          Todo storage todo = todos[_index];
45          todo.completed = !todo.completed;
46      }
47  
48      function remove(uint _index) public {    infinite gas
49          delete todos[_index];
50      }
51  }
```

Tutorial 15: Data Structures:Enums



8.4 Data Structures - Enums
15 / 19

We can update the enum value of a variable by assigning it the `uint` representing the enum member (line 30). Shipped would be 1 in this example. Another way to update the value is using the dot operator by providing the name of the enum and its member (line 35).

Removing an enum value

We can use the delete operator to delete the enum value of the variable, which means as for arrays and mappings, to set the default value to 0.

Watch a video tutorial on Enums.

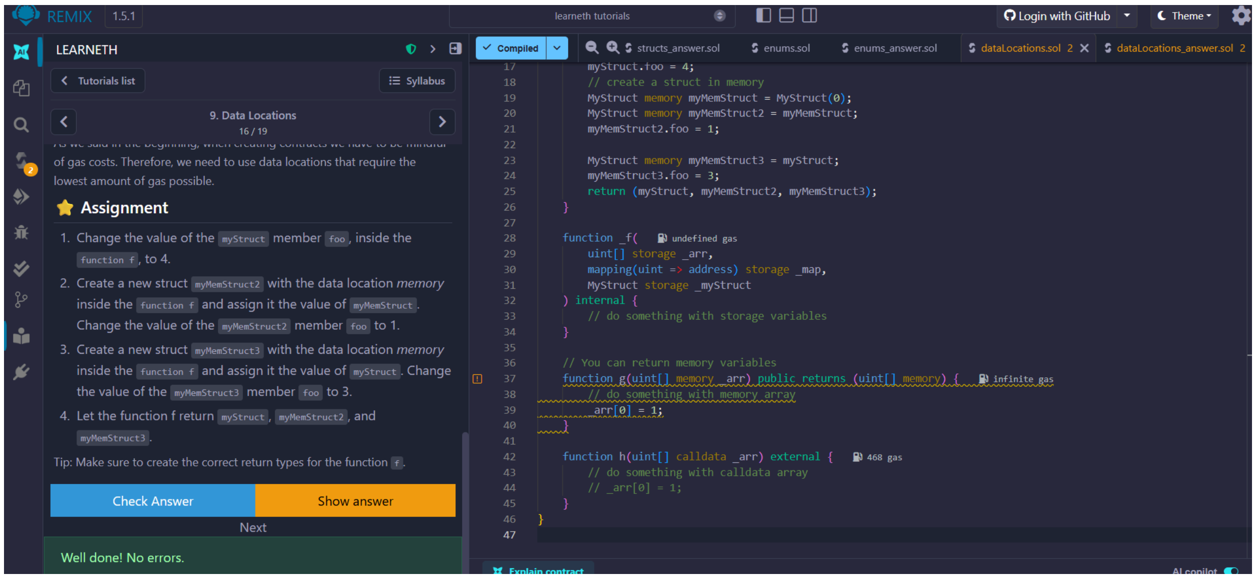
Assignment

1. Define an enum type called `Size` with the members `S`, `M`, and `L`.
2. Initialize the variable `sizes` of the enum type `Size`.
3. Create a getter function `getSize()` that returns the value of the variable `sizes`.

Well done! No errors.

```
        L
    }

    // 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 set(Status _status) public {
        status = _status;
    }

    // You can update to a specific enum like this
    function cancel() public {
        status = Status.Canceled;
    }

    // delete resets the enum to its first value, 0
    function reset() public {
        delete status;
    }
}
```

Tutorial 16:Data Locations



9. Data Locations
16 / 19

of gas costs. Therefore, we need to use data locations that require the lowest amount of gas possible.

Assignment

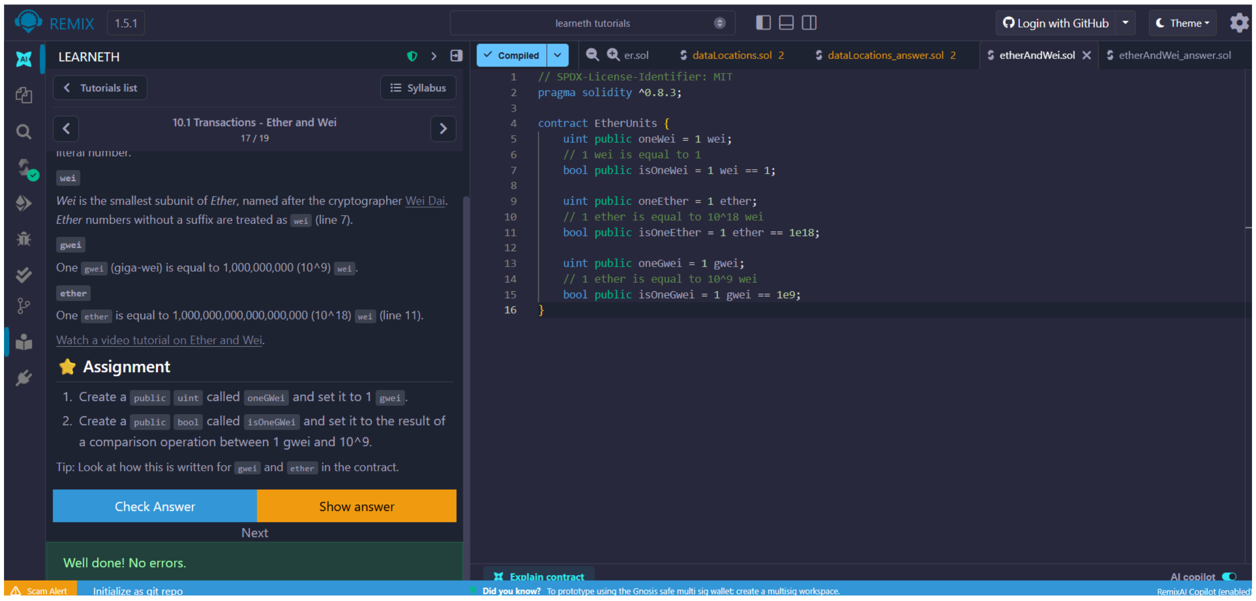
1. Change the value of the `myStruct` member `foo`, inside the `function f`, to 4.
2. Create a new struct `myMemStruct2` with the data location *memory* inside the `function f` and assign it the value of `myMemStruct`. Change the value of the `myMemStruct2` member `foo` to 1.
3. Create a new struct `myMemStruct3`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

Well done! No errors.

```
        myStruct.foo = 4;
        // create a struct in memory
        MyStruct memory myMemStruct = MyStruct(0);
        MyStruct memory myMemStruct2 = myMemStruct;
        myMemStruct2.foo = 1;

        MyStruct memory myMemStruct3 = myStruct;
        myMemStruct3.foo = 3;
        return (myStruct, myMemStruct2, myMemStruct3);
    }

    function _f(
        uint[] storage _arr,
        mapping(uint => address) storage _map,
        MyStruct storage _myStruct
    ) internal {
        // do something with storage variables
    }

    // You can return memory variables
    function g(uint[] memory _arr) public returns (uint[] memory) {
        // do something with memory array
        _arr[0] = 1;
    }

    function h(uint[] calldata _arr) external {
        // do something with calldata array
        // _arr[0] = 1;
    }
}
```

Tutorial 17:Transactions-Ether and Wei



10.1 Transactions - Ether and Wei
17 / 19

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 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

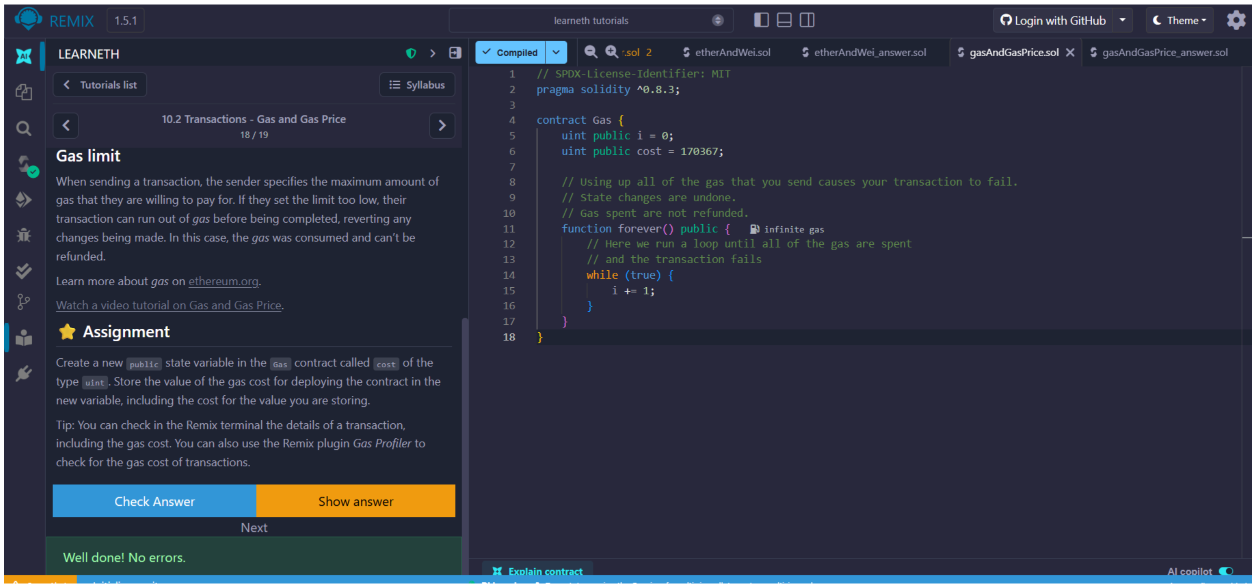
Next

Well done! No errors.

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

Did you know? To prototype using the Gnosis safe multi sig wallet: create a multisig workspace.

Tutorial 18:Transactions-Gas and Gas Price



10.2 Transactions - Gas and Gas Price
18 / 19

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.

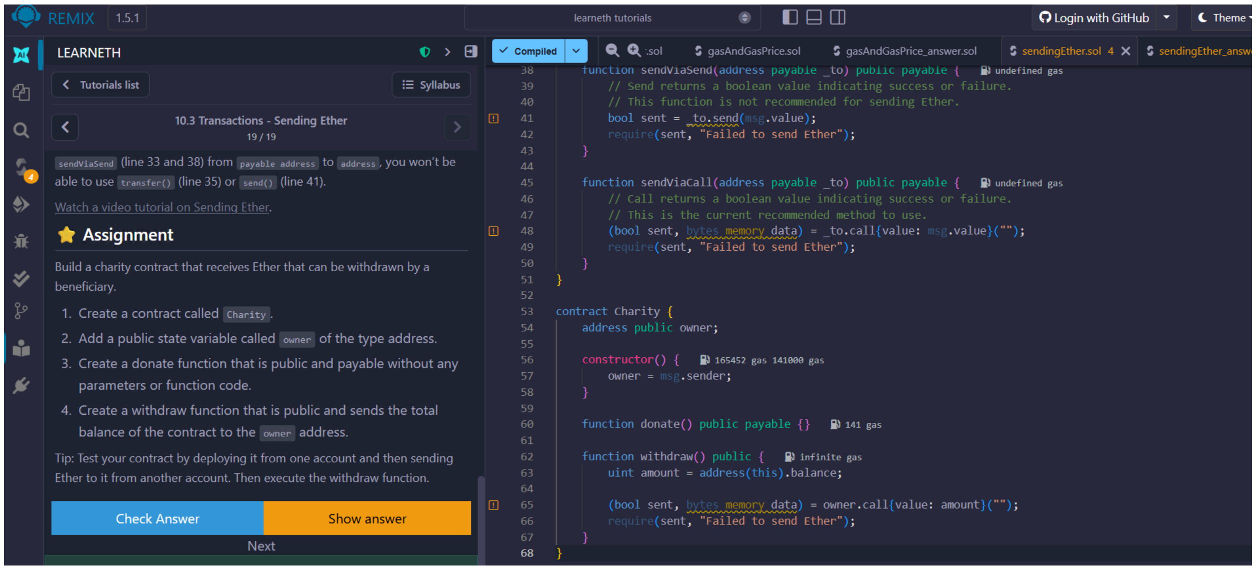
Check Answer | Show answer

Next

Well done! No errors.

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

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;
        }
    }
}
```

Tutorial 19:Transactions-Sending Ether



10.3 Transactions - Sending Ether
19 / 19

`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.

Check Answer Show answer

Next

```
38      function sendViaSend(address payable _to) public payable {    undefined gas
39          // Send returns a boolean value indicating success or failure.
40          // This function is not recommended for sending Ether.
41          bool sent = _to.send(msg.value);
42          require(sent, "Failed to send Ether");
43      }
44
45      function sendViaCall(address payable _to) public payable {    undefined gas
46          // Call returns a boolean value indicating success or failure.
47          // This is the current recommended method to use.
48          (bool sent, bytes memory data) = _to.call{value: msg.value}("");
49          require(sent, "Failed to send Ether");
50      }
51  }
52
53  contract Charity {
54      address public owner;
55
56      constructor() {    165452 gas 141000 gas
57          owner = msg.sender;
58      }
59
60      function donate() public payable {}    141 gas
61
62      function withdraw() public {    infinite gas
63          uint amount = address(this).balance;
64
65          (bool sent, bytes memory data) = owner.call{value: amount}("");
66          require(sent, "Failed to send Ether");
67      }
68  }
```

CONCLUSION:

Through this experiment, the basic concepts of Solidity programming were learned by completing practical assignments using the Remix IDE. Important topics such as data types, variables, different types of functions, visibility, modifiers, constructors, control flow statements, data structures, and transactions were studied and applied while creating smart contracts. The hands-on practice helped in understanding how to design, compile, and deploy contracts using the Remix VM. Overall, this experiment helped in building a clear understanding of blockchain concepts and provided a strong foundation for developing and managing smart contracts effectively.