

## Experiment No. 4

**Aim:** Hands on Solidity Programming Assignments for creating Smart Contracts

**Lab Objectives:** To explore Blockchain concepts.

**Lab Outcomes (LO):** Design Smart Contract using Solidity (LO2)

**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 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.

### 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:

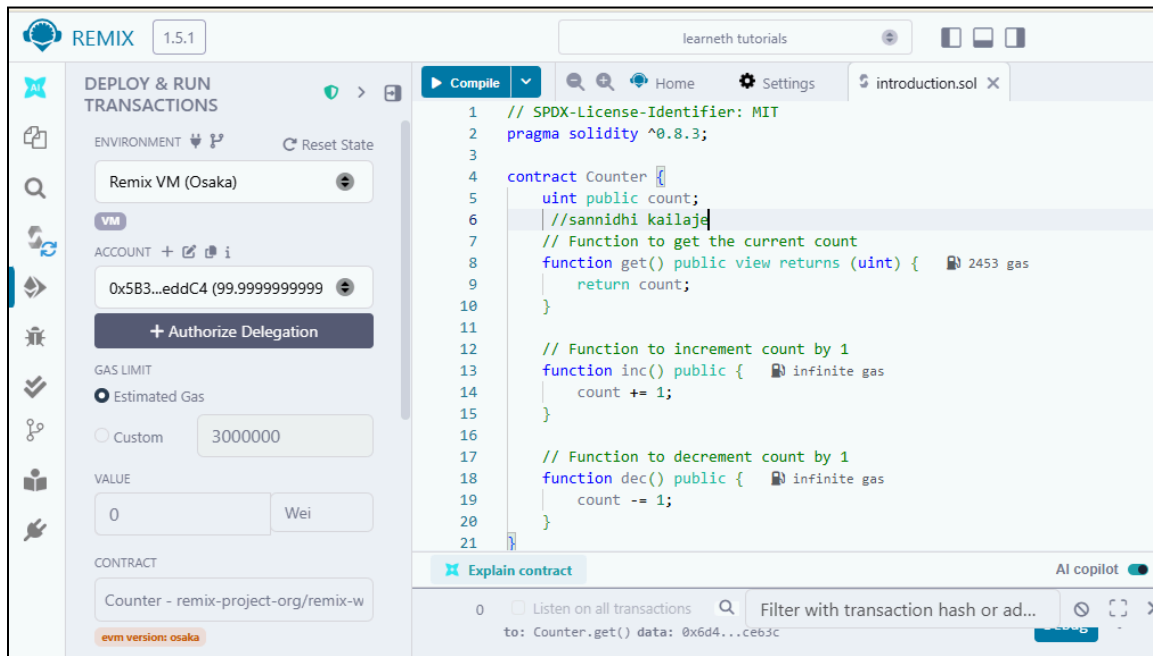
- storage: Data stored permanently on the blockchain. Examples: state variables.
- memory: Temporary data storage that exists only while a function is executing. Used for local variables and function inputs.
- calldata: A non-modifiable and non-persistent location used for external function parameters.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

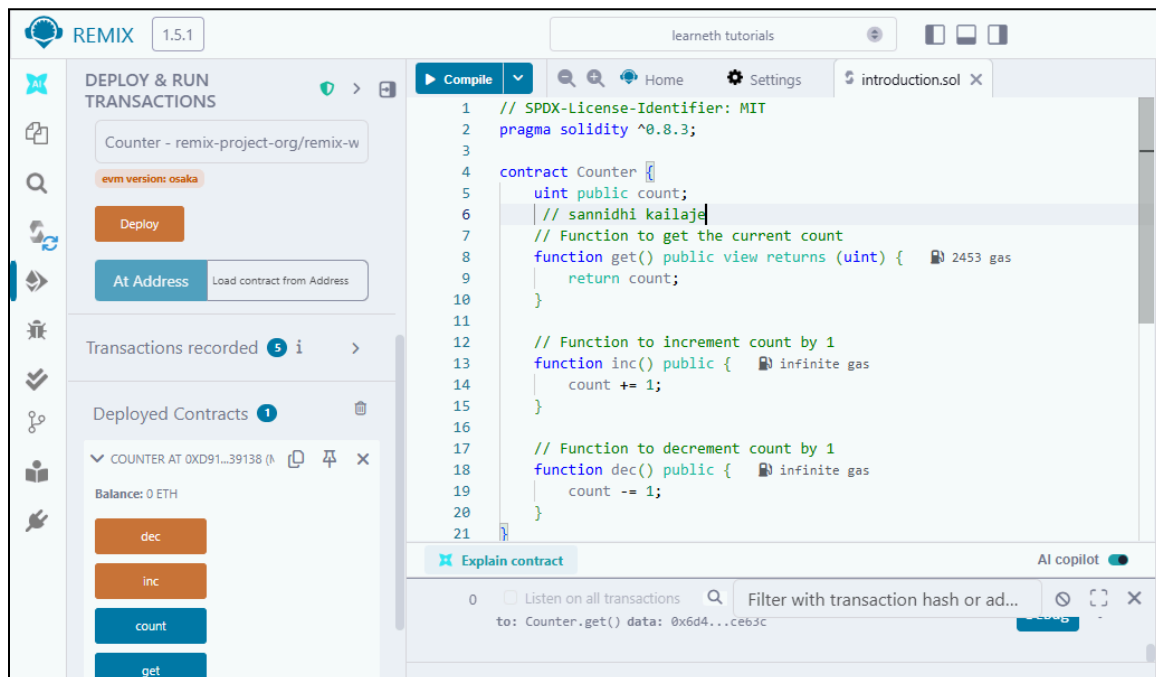
- Ether and Wei: Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit ( $1 \text{ Ether} = 10^{18} \text{ Wei}$ ). This ensures high precision in financial transactions.
- Gas and Gas Price: Every transaction consumes gas, which represents computational effort. The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

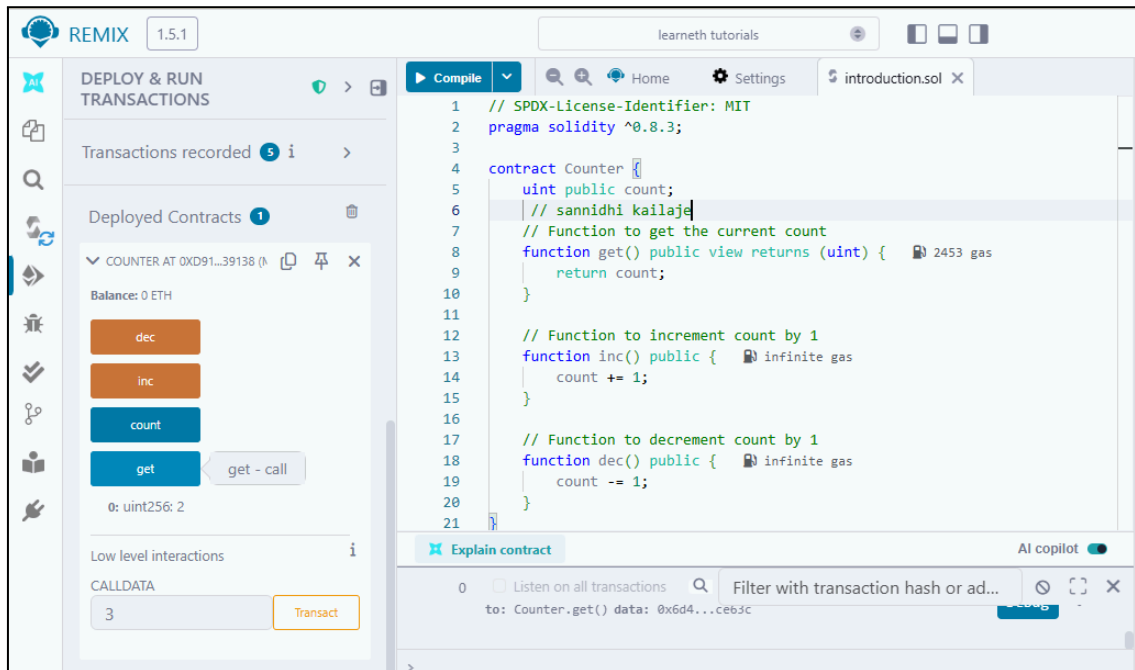
- Tutorial no. 1 – Compile the code



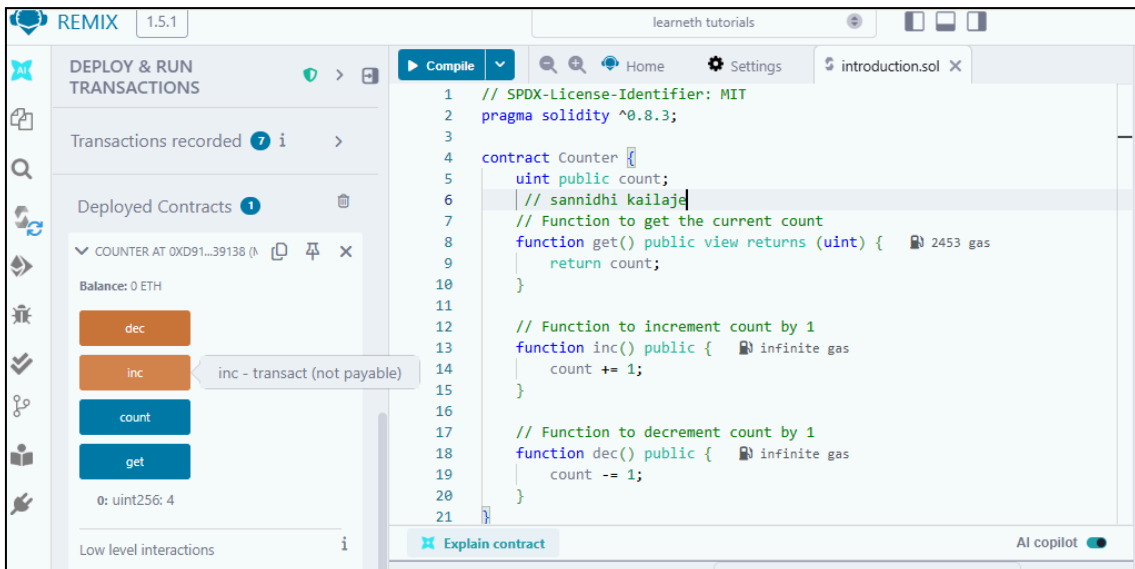
- Tutorial no. 1 – Deploy the contract



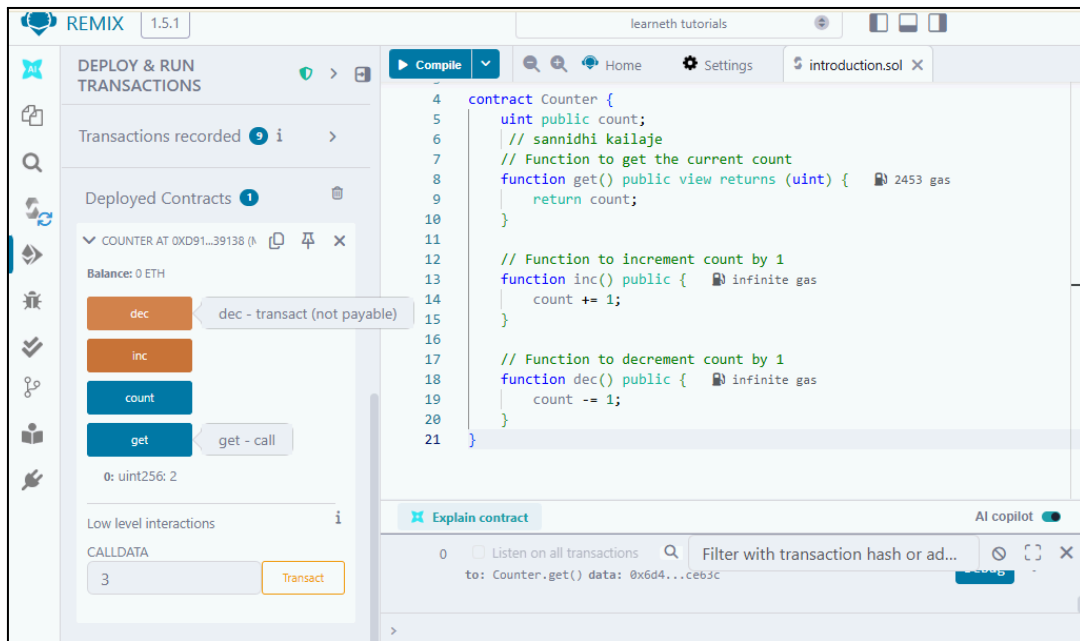
- Tutorial no. 1 – get



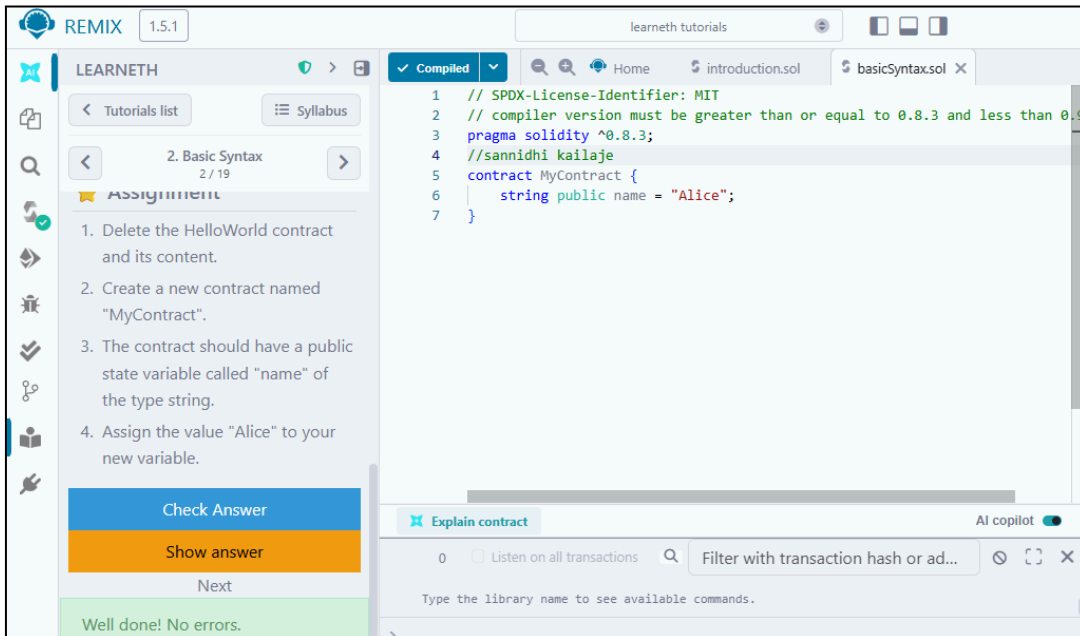
- Tutorial no. 1 – Increment



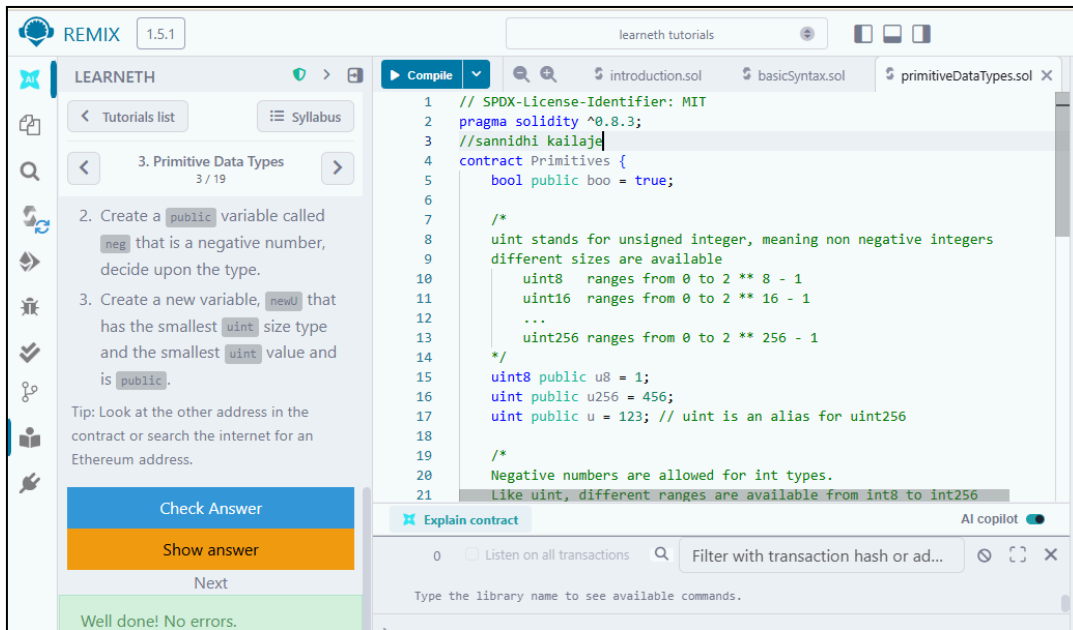
- Tutorial no. 1 – Decrement



- Tutorial no. 2



- Tutorial no. 3



REMIX 1.5.1

learneth tutorials

LEARNETH

Tutorials list Syllabus

3. Primitive Data Types 3 / 19

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.

Check Answer

Show answer

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract Primitives {
5     bool public boo = true;
6
7
8     /*
9     uint stands for unsigned integer, meaning non negative integers
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 uint256
19
20    /*
21    Negative numbers are allowed for int types.
22    Like uint, different ranges are available from int8 to int256
23    */
24 }
```

Compile

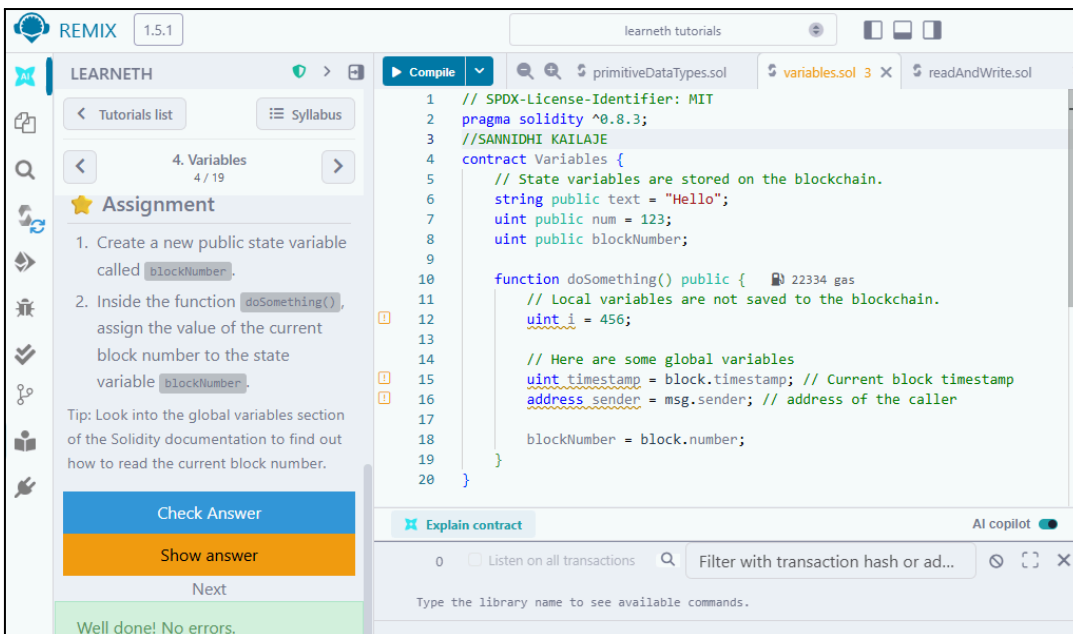
introduction.sol basicSyntax.sol primitiveDataTypes.sol

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

Type the library name to see available commands.

AI copilot

- Tutorial no. 4



REMIX 1.5.1

learneth tutorials

LEARNETH

Tutorials list Syllabus

4. Variables 4 / 19

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.

Check Answer

Show answer

Next

Well done! No errors.

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

Compile

primitiveDataTypes.sol variables.sol 3 x readAndWrite.sol

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

Type the library name to see available commands.

AI copilot

- Tutorial no. 5

REMIX 1.5.1

LEARNETH

Tutorials list Syllabus

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

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.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7
8     // State variable to store a boolean
9     bool public b = true;
10
11     // You need to send a transaction to write to a state variable.
12     function set(uint _num) public { 22536 gas
13         num = _num;
14     }
15
16     // You can read from a state variable without sending a transaction.
17     function get() public view returns (uint) { 2475 gas
18         return num;
19     }
20
21     // Function to get the value of b
```

Explain contract

AI copilot

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

Type the library name to see available commands.

- Tutorial no. 6

REMIX 1.5.1

LEARNETH

Tutorials list Syllabus

5.2 Functions - View and Pure 6 / 19

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

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //SANNIDHI KAILAJE
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 addToX2(uint y) public returns (uint) { infinite gas
18         x += y;
19         return x;
20     }
21 }
```

Explain contract

AI copilot

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

Type the library name to see available commands.



- Tutorial no. 7

**LEARNETH** 1.5.1

**5.3 Functions - Modifiers and Constructors** 7 / 19

or the owner variable upon the creation of the contract.

Watch a video tutorial on Function Modifiers.

★ **Assignment**

- 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.
- Make sure that `x` can only be increased.
- The body of the function `increaseX` should be empty.

Tip: Use modifiers.

[Check Answer](#) [Show answer](#)

Next

Well done! No errors.

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

- Tutorial no. 8

**LEARNETH** 1.5.1

**5.4 Functions - Inputs and Outputs** 8 / 19

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.

[Check Answer](#) [Show answer](#)

Next

Well done! No errors.

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

- Tutorial no. 9

**LEARNETH** 1.5.1

**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.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract Base {
5     // Private function can only be called
6     // - inside this contract
7     // Contracts that inherit this contract cannot call this function
8     function privateFunc() private pure returns (string memory) {
9         return "private function called";
10    }
11
12    function testPrivateFunc() public pure returns (string memory) {
13        return privateFunc();
14    }
15
16    // Internal function can be called
17    // - inside this contract
18    // - inside contracts that inherit this contract
19    function internalFunc() internal pure returns (string memory) {
20        return "internal function called";
21    }
22
23    function testInternalFunc() public pure virtual returns (string memory) {
24        return internalFunc();
25    }
26 }
```

- Tutorial no. 10

**LEARNETH** 1.5.1

**7.1 Control Flow - If/Else** 10 / 19

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 ternary operator to return the result of the `evenCheck` function.

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

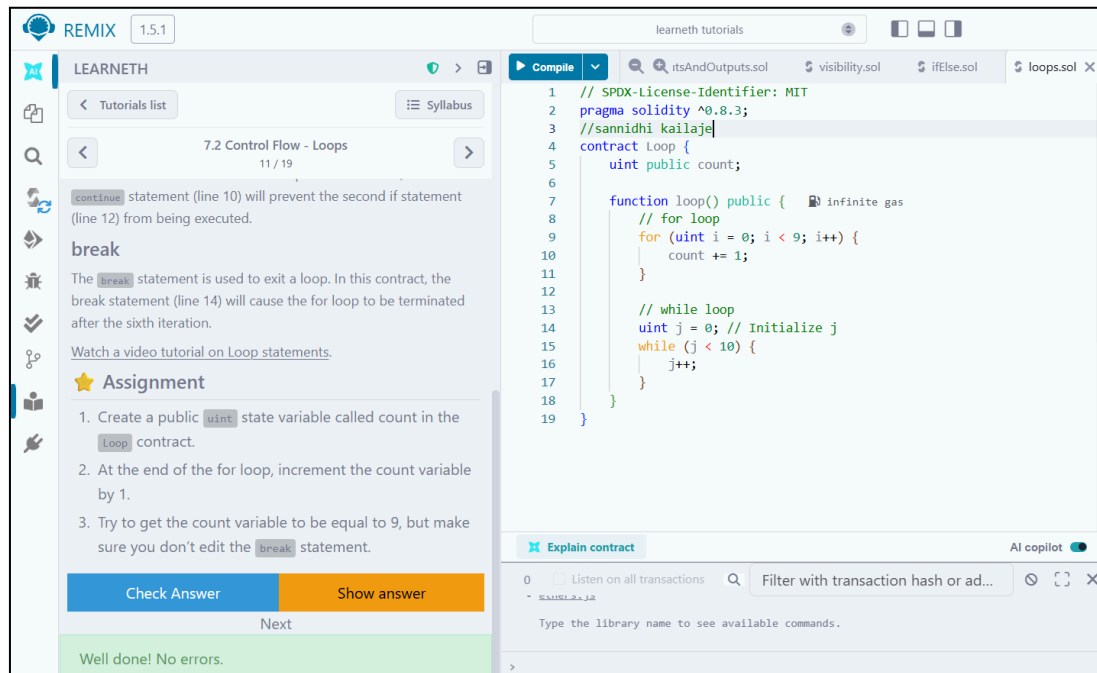
[Check Answer](#) [Show answer](#)

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract IfElse {
5     function foo(uint x) public pure returns (uint) {
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) {
16        // if (_x < 10) {
17        //     return 1;
18        // }
19        // return 2;
20
21        // shorthand way to write if / else statement
22        return _x < 10 ? 1 : 2;
23    }
24
25    function evenCheck(uint x) public pure returns (bool) {
26    }
```

- Tutorial no. 11



**REMUX 1.5.1**

**LEARNETH**

Tutorials list | Syllabus

7.2 Control Flow - Loops 11 / 19

`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.

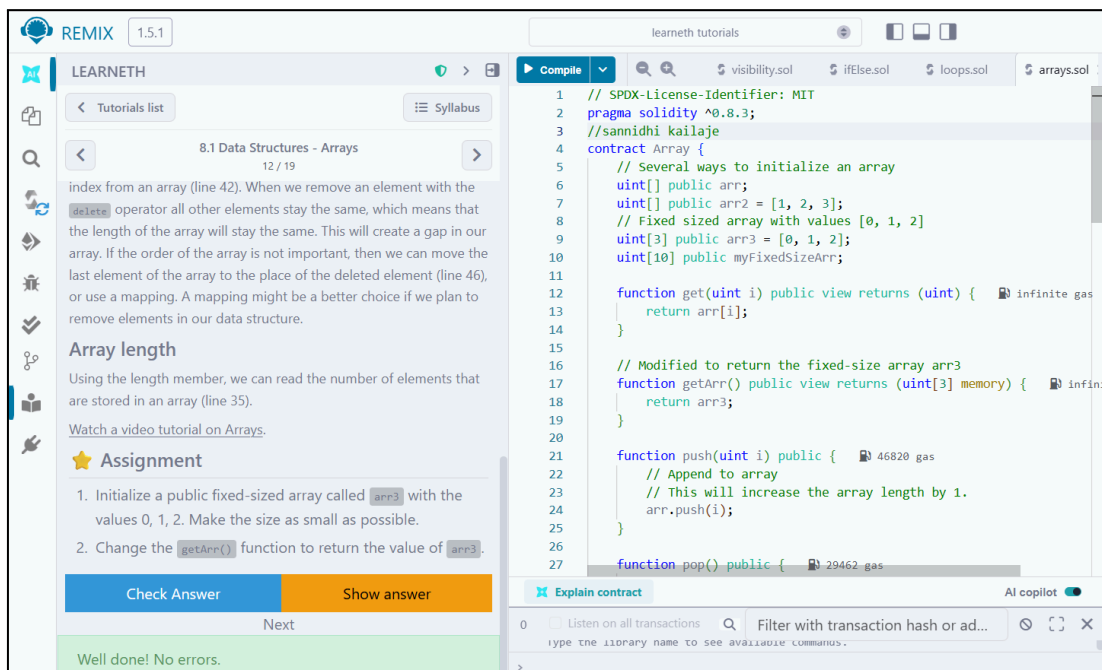
**Compile** | `visibility.sol` | `ifElse.sol` | `loops.sol`

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract Loop {
5     uint public count;
6
7     function loop() public {
8         // for loop
9         for (uint i = 0; i < 9; i++) {
10             count += 1;
11         }
12
13         // while loop
14         uint j = 0; // Initialize j
15         while (j < 10) {
16             j++;
17         }
18     }
19 }
```

**Explain contract** | AI copilot

0 ☐ Listen on all transactions | Filter with transaction hash or ad... | Type the library name to see available commands.

- Tutorial no. 12



**REMUX 1.5.1**

**LEARNETH**

Tutorials list | Syllabus

8.1 Data Structures - Arrays 12 / 19

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`.

[Check Answer](#) [Show answer](#)

Next

Well done! No errors.

**Compile** | `visibility.sol` | `ifElse.sol` | `loops.sol` | `arrays.sol`

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
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 with values [0, 1, 2]
9     uint[3] public arr3 = [0, 1, 2];
10    uint[10] public myFixedSizeArr;
11
12    function get(uint i) public view returns (uint) {
13        return arr[i];
14    }
15
16    // Modified to return the fixed-size array arr3
17    function getArr() public view returns (uint[3] memory) {
18        return arr3;
19    }
20
21    function push(uint i) public {
22        // Append to array
23        // This will increase the array length by 1.
24        arr.push(i);
25    }
26
27    function pop() public {
28        // Remove last element
29    }
30 }
```

**Explain contract** | AI copilot

0 ☐ Listen on all transactions | Filter with transaction hash or ad... | Type the library name to see available commands.

- Tutorial no. 13

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' sidebar displays '8.2 Data Structures - Mappings' with a progress indicator '13 / 19'. The main content area shows a tutorial titled 'Removing values' with instructions on using the delete operator. Below the tutorial is an 'Assignment' section with three steps: 1. Create a public mapping `balances` of type `address` to `uint`. 2. Change `get` and `remove` functions to work with the mapping. 3. Change the `set` function to create a new entry. At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a green message 'Well done! No errors.'.

The main editor shows a Solidity file named `mappings.sol` with the following code:

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

The bottom panel shows the 'Explain contract' feature with a search bar and a list of libraries including `ethers.js`.

- Tutorial no. 14

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' sidebar displays '8.3 Data Structures - Structs' with a progress indicator '14 / 19'. The main content area shows a tutorial titled 'Accessing structs' with instructions on using the dot operator. Below the tutorial is an 'Assignment' section with a task: Create a function `remove` that takes a `uint` as a parameter and deletes a struct member. At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a green message 'Well done! No errors.'.

The main editor shows a Solidity file named `structs.sol` with the following code:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
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 {
14        // 3 ways to initialize a struct
15        // - calling it like a function
16        todos.push(Todo(_text, false));
17
18        // key value mapping
19        todos.push(Todo({text: _text, completed: false}));
20
21        // initialize an empty struct and then update it
22        Todo memory todo;
23        todo.text = _text;
24        todo.completed = false;
25    }
```

The bottom panel shows the 'Explain contract' feature with a search bar and a list of libraries including `ethers.js`.

- Tutorial no. 15

**LEARNETH** 1.5.1

**Tutorials list** | **Syllabus**

**8.4 Data Structures - Enums** 15 / 19

enum, representing the enum member (line 35), 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`.

**Check Answer** **Show answer**

Next

Well done! No errors.

**enums.sol**

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

**enums\_answer.sol**

```
2605 eas
```

**Explain contract**

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

THE FOLLOWING COMMANDS ARE AVAILABLE:

- ethers.js

Type the library name to see available commands.

- Tutorial no. 16

**LEARNETH** 1.5.1

**Tutorials list** | **Syllabus**

**9. Data Locations** 16 / 19

**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`.

**Check Answer** **Show answer**

Next

Well done! No errors.

**dataLocations.sol**

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract DataLocations {
5     uint[] public arr;
6     mapping(uint => address) map;
7     struct MyStruct {
8         uint foo;
9     }
10    mapping(uint => MyStruct) myStructs;
11
12    function f() public returns (MyStruct memory, MyStruct memory, MyStruct memory) {
13        // call _f with state variables
14        _f(arr, map, myStructs[1]);
15
16        // get a struct from a mapping
17        MyStruct storage myStruct = myStructs[1];
18        // create a struct in memory
19        MyStruct memory myMemStruct = MyStruct(0);
20
21        // Change the value of the myStruct member foo to 4
22        myStruct.foo = 4;
23
24        // Create a new struct myMemStruct2 with the data location memory
25        // and assign it the value of myMemStruct
```

**Explain contract**

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

THE FOLLOWING COMMANDS ARE AVAILABLE:

- ethers.js

Type the library name to see available commands.

- Tutorial no. 17

The screenshot shows the REMIX IDE interface for Tutorial 17. The sidebar on the left displays the 'Tutorials list' with '10.1 Transactions - Ether and Wei' selected. The main editor shows the Solidity code for the 'EtherUnits' contract, which defines constants for wei, ether, and gwei, and includes a comparison function 'isOneWei'. The bottom panel features an 'Explain contract' tool with a search bar and a list of available commands.

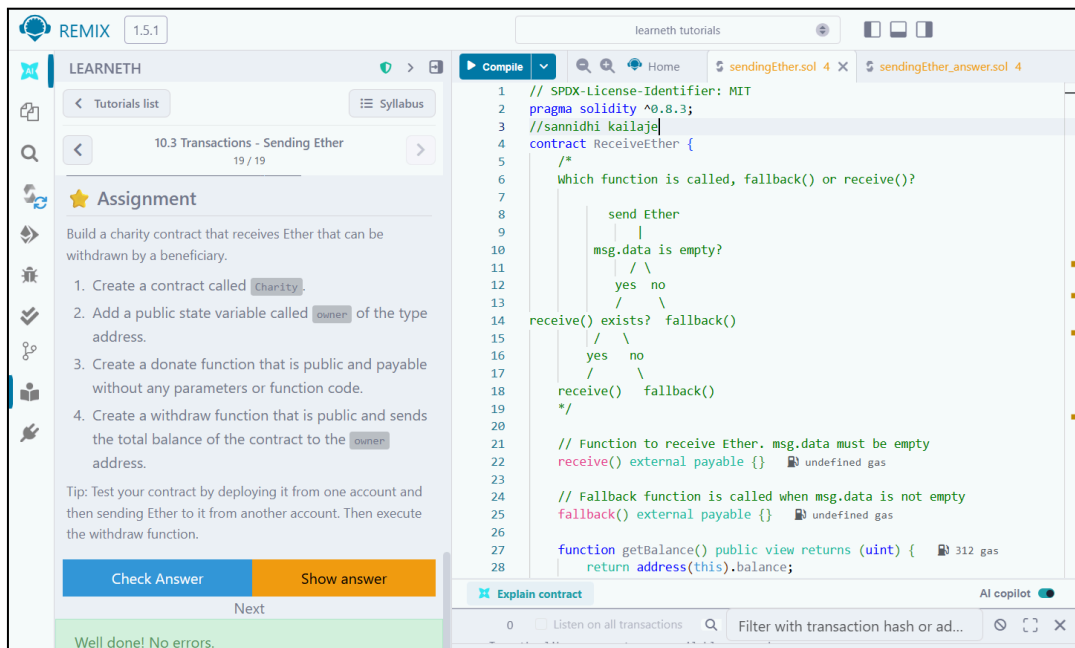
```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract EtherUnits {
5     uint public oneWei = 1 wei;
6     // 1 wei is equal to 1
7     bool public isOneWei = 1 wei == 1;
8
9     uint public oneEther = 1 ether;
10    // 1 ether is equal to 10^18 wei
11    bool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

- Tutorial no. 18

The screenshot shows the REMIX IDE interface for Tutorial 18. The sidebar on the left displays the 'Tutorials list' with '10.2 Transactions - Gas and Gas Price' selected. The main editor shows the Solidity code for the 'Gas' contract, which defines a state variable 'cost' and a function 'forever' that runs an infinite loop. The bottom panel features an 'Explain contract' tool with a search bar and a list of available commands.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje
4 contract Gas {
5     uint public i = 0;
6     uint public cost = 170367;
7
8     // Using up all of the gas that you send causes your transaction to fail.
9     // State changes are undone.
10    // Gas spent are not refunded.
11    function forever() public {
12        // Here we run a loop until all of the gas are spent
13        // and the transaction fails
14        while (true) {
15            i += 1;
16        }
17    }
18 }
```

- Tutorial no. 19



## Conclusion:

Through this experiment, the fundamentals of Solidity programming were explored by completing practical assignments in the Remix IDE. Concepts such as data types, variables, functions, visibility, modifiers, constructors, control flow, data structures, and transactions were implemented and understood. The hands-on practice helped in designing, compiling, and deploying smart contracts on the Remix VM, thereby strengthening the understanding of blockchain concepts. This experiment provided a strong foundation for developing and managing smart contracts efficiently.



























