

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:

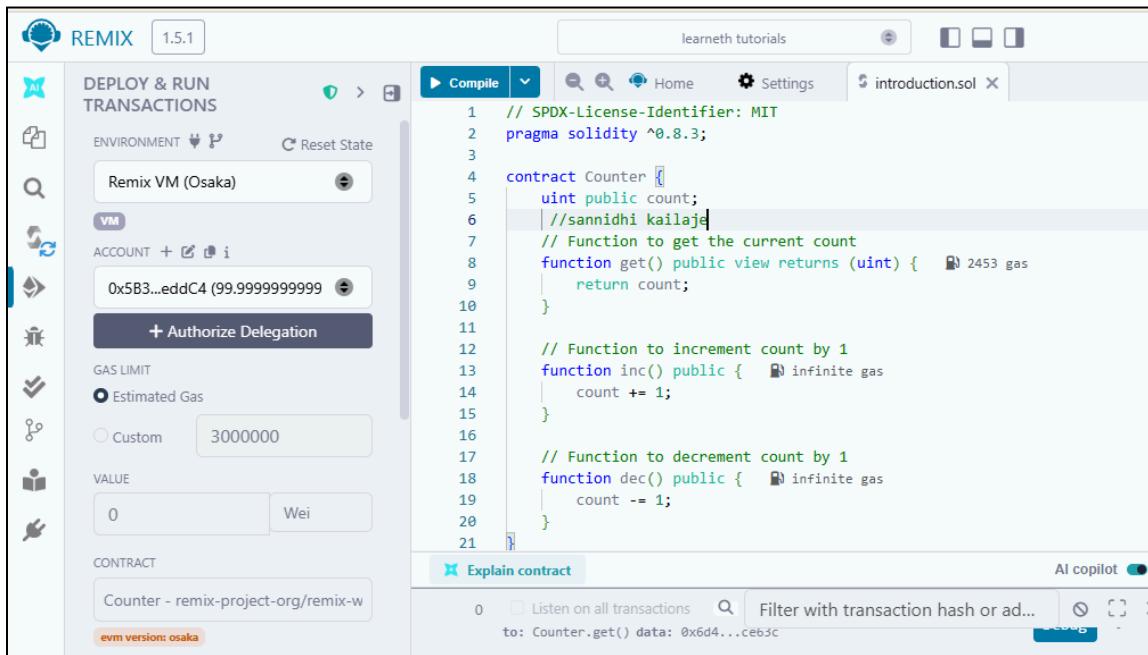
- storage: Data stored permanently on the blockchain. Examples: state variables.
- memory: Temporary data storage that exists only while a function is executing. Used for local variables and function inputs.
- calldata: A non-modifiable and non-persistent location used for external function parameters.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



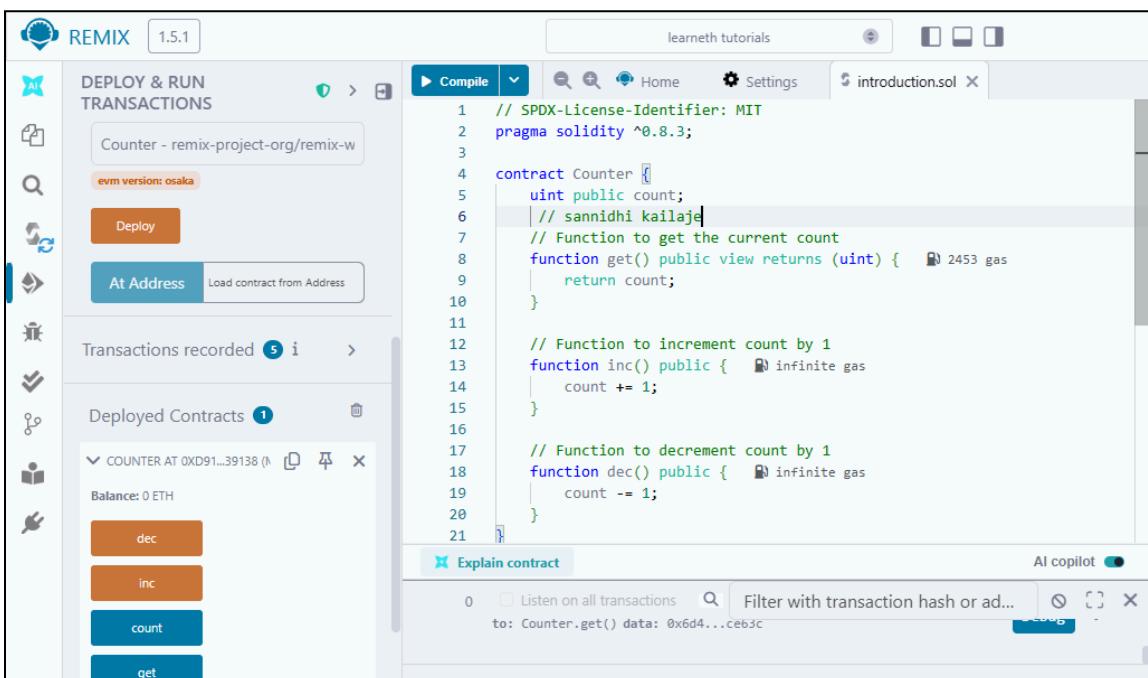
The screenshot shows the REMIX IDE interface with the following details:

- Left Panel (DEPLOY & RUN TRANSACTIONS):**
 - ENVIRONMENT: Remix VM (Osaka)
 - ACCOUNT: 0x5B3...eddC4 (99.999999999)
 - GAS LIMIT: Estimated Gas (radio button selected)
 - Custom: 3000000
 - VALUE: 0 Wei
 - CONTRACT: Counter - remix-project-org/remix-w
 - evm version: osaka
- Right Panel (Code Editor):**

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     uint public count;
6     // sannidhi kailaje
7     // Function to get the current count
8     function get() public view returns (uint) {
9         return count;
10    }
11
12    // Function to increment count by 1
13    function inc() public {
14        count += 1;
15    }
16
17    // Function to decrement count by 1
18    function dec() public {
19        count -= 1;
20    }
21 }
```
- Bottom Panel (Logs):**
 - 0 Listen on all transactions
 - Filter with transaction hash or address: to: Counter.get() data: 0xd4...ce63c

- Tutorial no. 1 – Deploy the contract



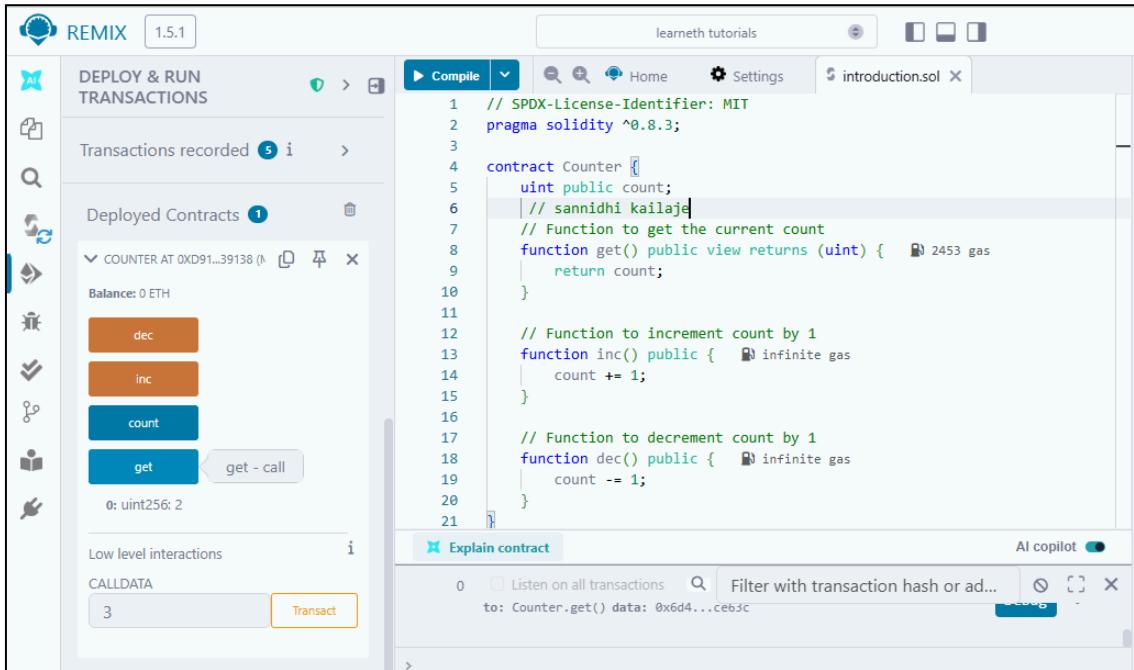
The screenshot shows the REMIX IDE interface after deployment, with the following details:

- Left Panel (DEPLOY & RUN TRANSACTIONS):**
 - Deployed Contracts: 1
 - COUNTER AT 0xD91...39138
 - Balance: 0 ETH
 - Functions: dec, inc, count, get
- Right Panel (Code Editor):**

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     uint public count;
6     // sannidhi kailaje
7     // Function to get the current count
8     function get() public view returns (uint) {
9         return count;
10    }
11
12    // Function to increment count by 1
13    function inc() public {
14        count += 1;
15    }
16
17    // Function to decrement count by 1
18    function dec() public {
19        count -= 1;
20    }
21 }
```
- Bottom Panel (Logs):**
 - 0 Listen on all transactions
 - Filter with transaction hash or address: to: Counter.get() data: 0xd4...ce63c

- Tutorial no. 1 – get



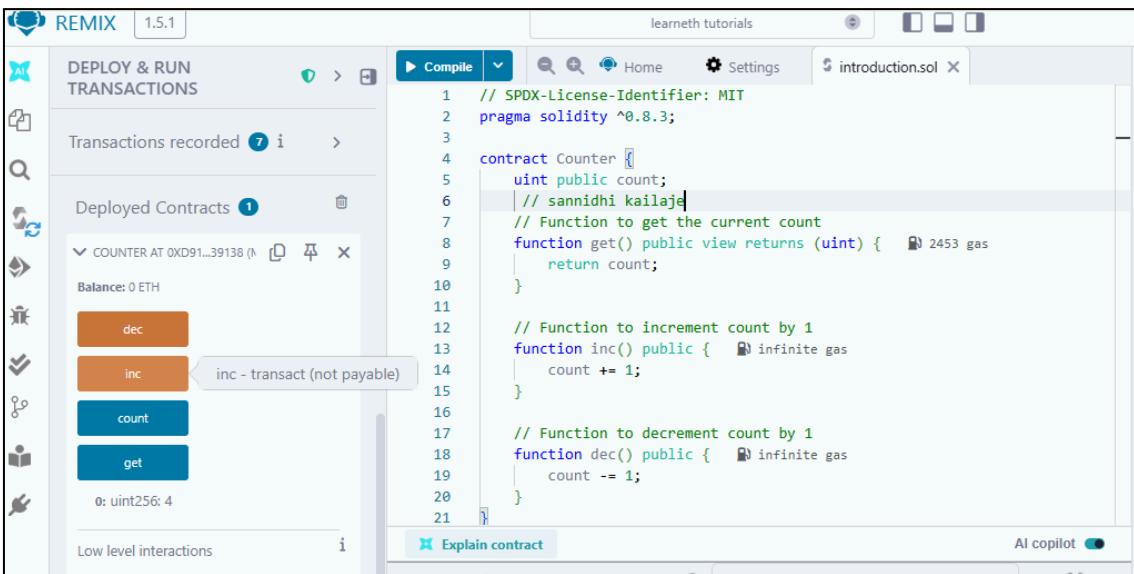
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract Counter {
    uint public count;
    // sannidhi kailaje
    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

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

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

- Tutorial no. 1 – Increment



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

contract Counter {
    uint public count;
    // sannidhi kailaje
    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

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

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

- Tutorial no. 1 – Decrement

```

contract Counter {
    uint public count;
    // sannidhi kailaje
    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }
    // Function to increment count by 1
    function inc() public {
        count += 1;
    }
    // Function to decrement count by 1
    function dec() public {
        count -= 1;
    }
}

```

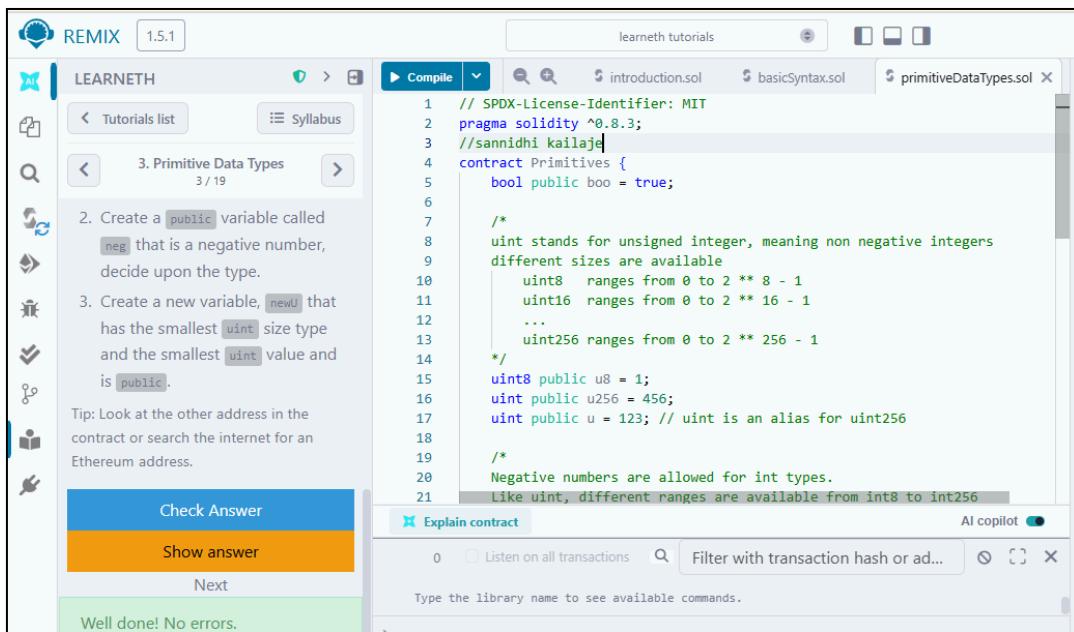
- Tutorial no. 2

```

// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9
pragma solidity ^0.8.3;
//sannidhi kailaje
contract MyContract {
    string public name = "Alice";
}

```

- Tutorial no. 3



The screenshot shows the REMIX IDE interface. The left sidebar displays a 'Tutorials list' for 'LEARNETH' with '3. Primitive Data Types' selected. The main workspace shows the following Solidity code:

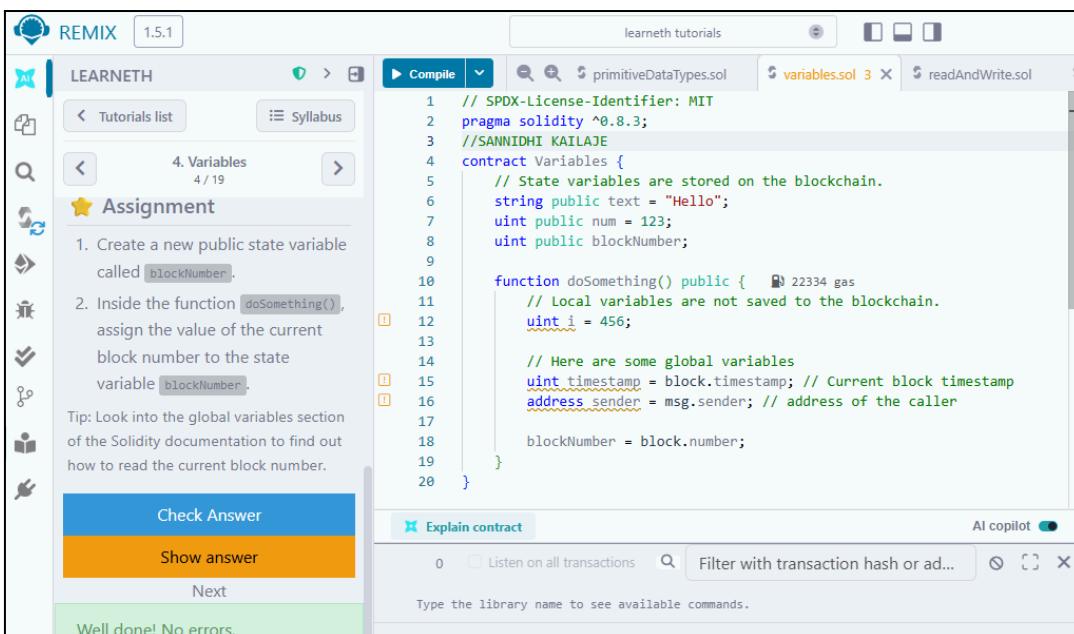
```

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      uint stands for unsigned integer, meaning non negative integers
9      different sizes are available
10     uint8 ranges from 0 to 2 ** 8 - 1
11     uint16 ranges from 0 to 2 ** 16 - 1
12     ...
13     uint256 ranges from 0 to 2 ** 256 - 1
14   */
15     uint8 public u8 = 1;
16     uint public u256 = 456;
17     uint public u = 123; // uint is an alias for uint256
18
19     /*
20      Negative numbers are allowed for int types.
21      Like uint, different ranges are available from int8 to int256

```

The code editor has a status bar at the bottom with 'Well done! No errors.'

- Tutorial no. 4



The screenshot shows the REMIX IDE interface. The left sidebar displays a 'Tutorials list' for 'LEARNETH' with '4. Variables' selected. The main workspace shows the following Solidity code:

```

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 {
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 }

```

The code editor has a status bar at the bottom with 'Well done! No errors.'

- Tutorial no. 5

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '5.1 Functions - Reading and Writing to a State Variable'. The main workspace contains the following Solidity code:

```

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 {
13        num = _num;
14    }
15
16    // You can read from a state variable without sending a transaction.
17    function get() public view returns (uint) {
18        return num;
19    }
20
21    // Function to get the value of b

```

Below the code editor, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A message at the bottom says 'Well done! No errors.'

- Tutorial no. 6

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '5.2 Functions - View and Pure'. The main workspace contains the following Solidity code:

```

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) {
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) {
14        return i + j;
15    }
16
17    function addToX2(uint y) public returns (uint) {
18        x += y;
19        return x;
20    }

```

Below the code editor, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A message at the bottom says 'Well done! No errors.'

- Tutorial no. 7

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays 'LEARNETH' and 'Tutorials list'. The main workspace shows a Solidity contract named 'FunctionModifier'. The code includes a constructor that sets the transaction sender as the owner, and a modifier 'onlyOwner' that checks if the caller is the owner. A tip at the bottom suggests using modifiers.

```

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

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays 'LEARNETH' and 'Tutorials list'. The main workspace shows a Solidity contract named 'Function'. It includes a function 'returnMany' that returns three values: 1, true, and 2. It also includes a function 'named' that returns two values: public and pure.

```

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

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//sannidhi kailaje
contract Base {
    // Private function can only be called
    // - inside this contract
    // Contracts that inherit this contract cannot call this function
    function privateFunc() private pure returns (string memory) {
        return "private function called";
    }

    function testPrivateFunc() public pure returns (string memory) {
        return privateFunc();
    }

    // Internal function can be called
    // - inside this contract
    // - inside contracts that inherit this contract
    function internalFunc() internal pure returns (string memory) {
        return "internal function called";
    }

    function testInternalFunc() public pure virtual returns (string memory) {
        return internalFunc();
    }
}

```

- Tutorial no. 10

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//sannidhi kailaje
contract IfElse {
    function foo(uint x) public pure returns (uint) {
        infinite gas
        if (x < 10) {
            return 0;
        } else if (x < 20) {
            return 1;
        } else {
            return 2;
        }
    }

    function ternary(uint _x) public pure returns (uint) {
        infinite gas
        // if (_x < 10) {
        //     return 1;
        // }
        // return 2;

        // shorthand way to write if / else statement
        return _x < 10 ? 1 : 2;
    }

    function evenCheck(uint x) public pure returns (bool) {
        infinite gas
    }
}

```

- Tutorial no. 11

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//sannidhi kailaje
contract Loop {
    uint public count;

    function loop() public {
        // for loop
        for (uint i = 0; i < 9; i++) {
            count += 1;
        }

        // while loop
        uint j = 0; // Initialize j
        while (j < 10) {
            j++;
        }
    }
}

```

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

- Create a public `uint` state variable called `count` in the `Loop` contract.
- At the end of the for loop, increment the `count` variable by 1.
- 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**

Well done! No errors.

- Tutorial no. 12

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//sannidhi kailaje
contract Array {
    // Several ways to initialize an array
    uint[] public arr;
    uint[] public arr2 = [1, 2, 3];
    // Fixed sized array with values [0, 1, 2]
    uint[3] public arr3 = [0, 1, 2];
    uint[10] public myFixedSizeArr;

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

    // Modified to return the fixed-size array arr3
    function getArr() public view returns (uint[3] memory) {
        infinite gas
        return arr3;
    }

    function push(uint i) public {
        46820 gas
        // Append to array
        // This will increase the array length by 1.
        arr.push(i);
    }

    function pop() public {
        29462 gas
    }
}

```

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.

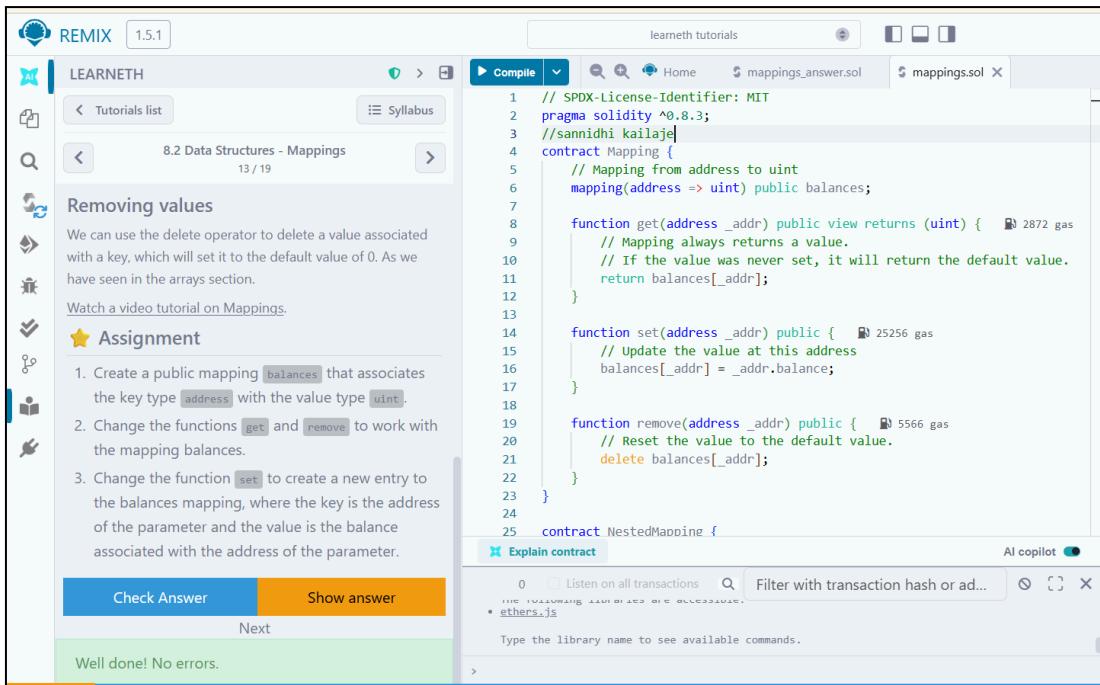
Assignment

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

Check Answer **Show answer**

Well done! No errors.

- Tutorial no. 13



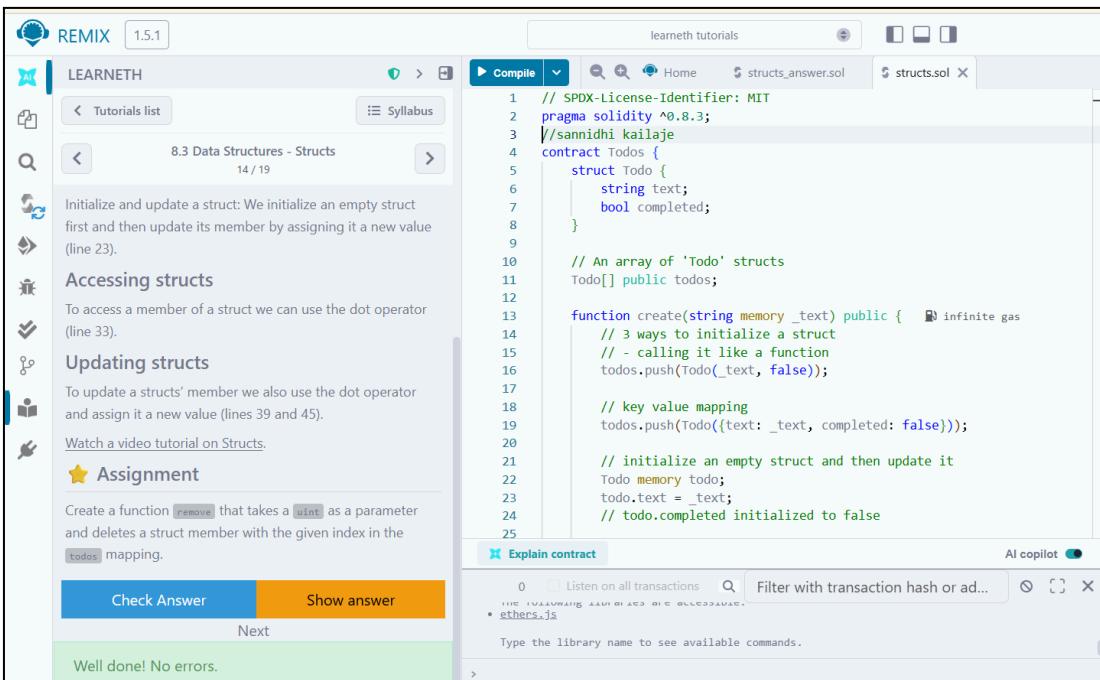
The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '8.2 Data Structures - Mappings'. The main workspace shows a Solidity contract named 'mappings.sol' with the following code:

```

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 {
26 }
```

The 'Compile' button is highlighted. Below the code editor, the 'Explain contract' panel is visible. At the bottom of the screen, a green message bar says 'Well done! No errors.'

- Tutorial no. 14



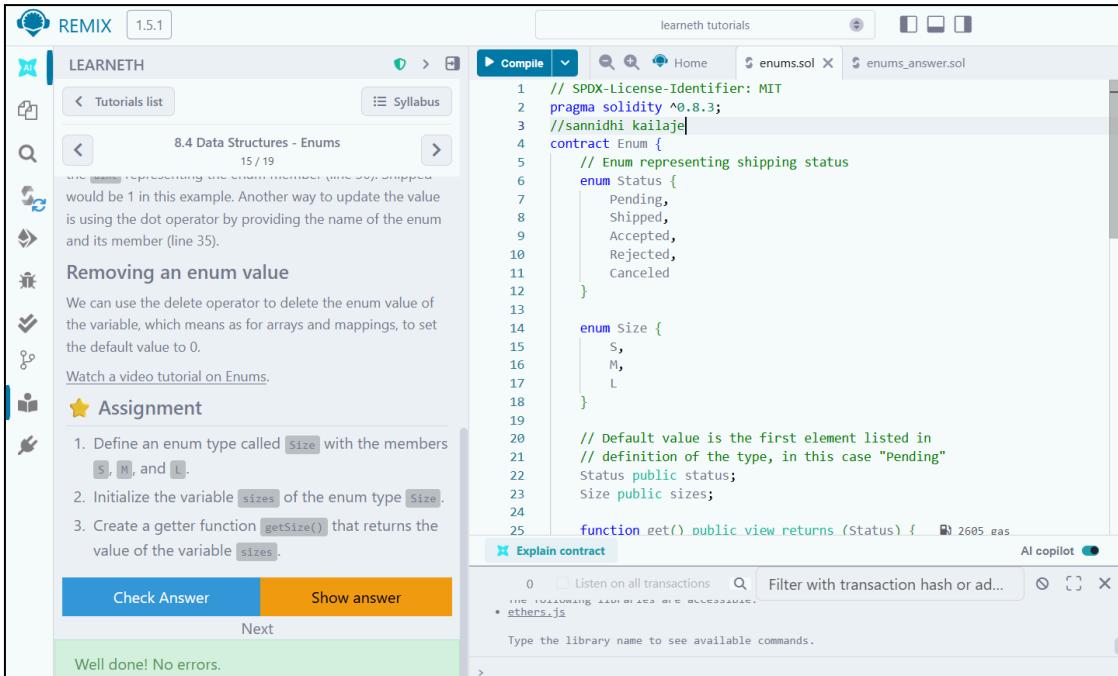
The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '8.3 Data Structures - Structs'. The main workspace shows a Solidity contract named 'structs.sol' with the following code:

```

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 initialized to false
25 }
```

The 'Compile' button is highlighted. Below the code editor, the 'Explain contract' panel is visible. At the bottom of the screen, a green message bar says 'Well done! No errors.'

- Tutorial no. 15



The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial list, with '8.4 Data Structures - Enums' selected. The main workspace shows the following Solidity code:

```

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) f... 2605_eas

```

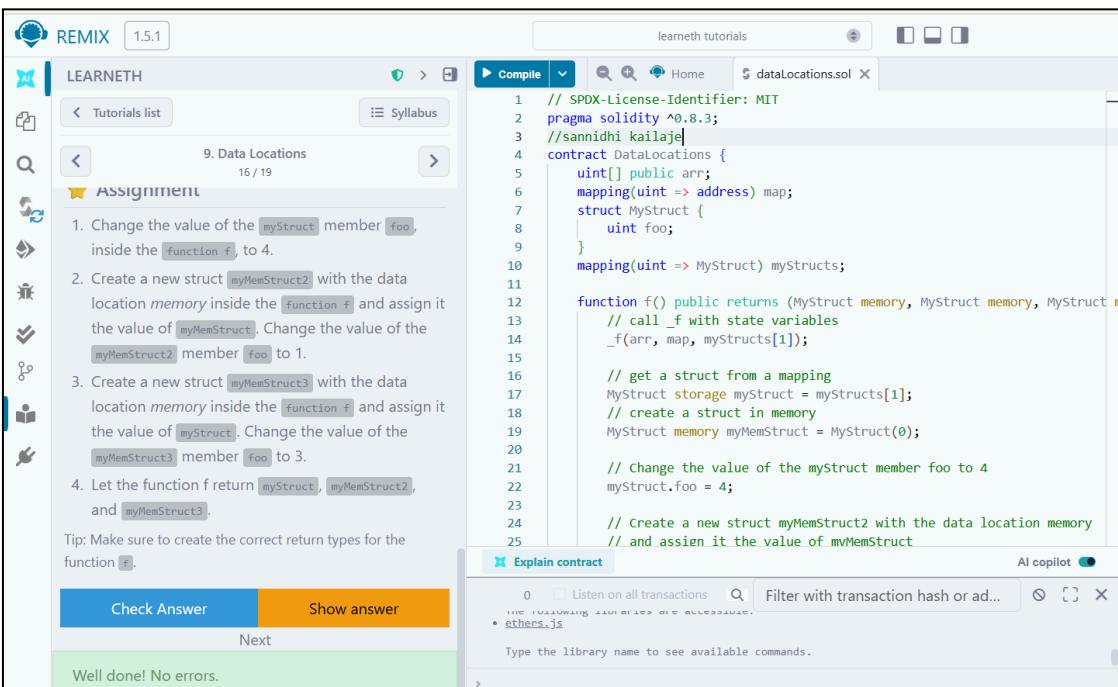
Below the code, there is an 'Assignment' section with three tasks:

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 `get()` that returns the value of the variable `sizes`.

At the bottom of the assignment section, there are 'Check Answer' and 'Show answer' buttons, and a 'Next' link.

The right side of the interface shows the transaction history and an AI copilot feature.

- Tutorial no. 16



The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial list, with '9. Data Locations' selected. The main workspace shows the following Solidity code:

```

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

```

Below the code, there is an 'Assignment' section with four tasks:

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`.

Below the assignment section, there is a tip: 'Tip: Make sure to create the correct return types for the function `f`'. At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a 'Next' link.

The right side of the interface shows the transaction history and an AI copilot feature.

- Tutorial no. 17

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//sannidhi kailaje
contract EtherUnits {
    uint public oneWei = 1 wei;
    // 1 wei is equal to 1
    bool public isOneWei = 1 wei == 1;

    uint public oneEther = 1 ether;
    // 1 ether is equal to 10^18 wei
    bool public isOneEther = 1 ether == 1e18;

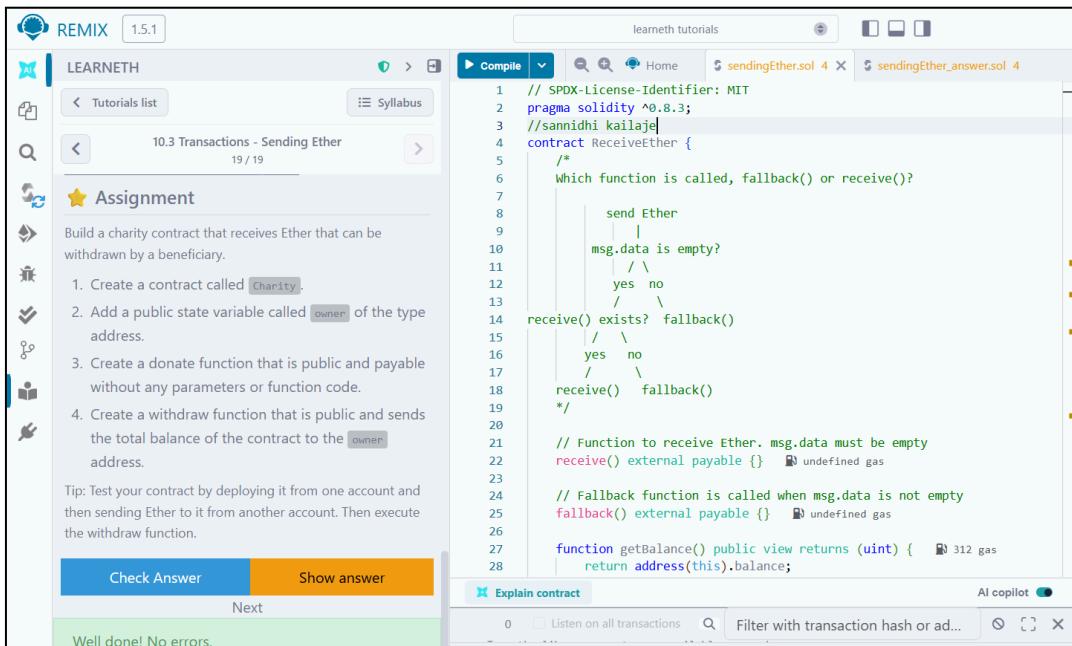
    uint public oneGwei = 1 gwei;
    // 1 ether is equal to 10^9 wei
    bool public isOneGwei = 1 gwei == 1e9;
}
```

- Tutorial no. 18

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//sannidhi kailaje
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 no. 19



The screenshot shows the Remix IDE interface. On the left, there's a sidebar with icons for file operations, search, and navigation. The main area has tabs for 'Tutorials list' and 'Syllabus'. Below that, it says '10.3 Transactions - Sending Ether' and '19 / 19'. A yellow star icon indicates an 'Assignment'. The assignment description is: 'Build a charity contract that receives Ether that can be withdrawn by a beneficiary.' It lists four tasks: 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. A tip at the bottom says: 'Tip: Test your contract by deploying it from one account and then sending Ether to it from another account. Then execute the withdraw function.' At the bottom of the assignment area, there are 'Check Answer' and 'Show answer' buttons, and a note 'Well done! No errors.'

On the right, the code editor shows the Solidity contract:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //sannidhi kailaje|
4 contract Receivether {
5     /*
6      * Which function is called, fallback() or receive()?
7      *
8      * send Ether
9      |
10     msg.data is empty?
11     / \
12     yes no
13     / \
14     receive() exists? fallback()
15     / \
16     yes no
17     / \
18     receive() fallback()
19 */
20
21 // Function to receive Ether. msg.data must be empty
22 receive() external payable {} // undefined gas
23
24 // Fallback function is called when msg.data is not empty
25 fallback() external payable {} // undefined gas
26
27 function getBalance() public view returns (uint) { // 312 gas
28     return address(this).balance;
}

```

Below the code editor, there are buttons for 'Explain contract' and 'AI copilot'. At the bottom, there are filters for 'Listen on all transactions', 'Filter with transaction hash or ad...', and a search bar.

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.

