

## Experiment No. 4

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

### Theory:

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

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

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

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

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

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

#### 2. Inputs and Outputs to Functions

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

#### 3. Visibility, Modifiers and Constructors

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

- o external: can be called only by external accounts or other contract
- **Modifiers** are reusable code blocks that change the behavior of functions. They are often used for access control, such as restricting sensitive functions to the contract owner (onlyOwner).
- **Constructors** are special functions executed only once during contract deployment. They initialize important values, such as setting the deploying account as the owner of the contract.

### 3. Control Flow: if-else, loops

Control flow in Solidity is similar to traditional programming languages:

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

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

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

### 6. Data Locations

Solidity uses three primary data locations for storing variables:

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

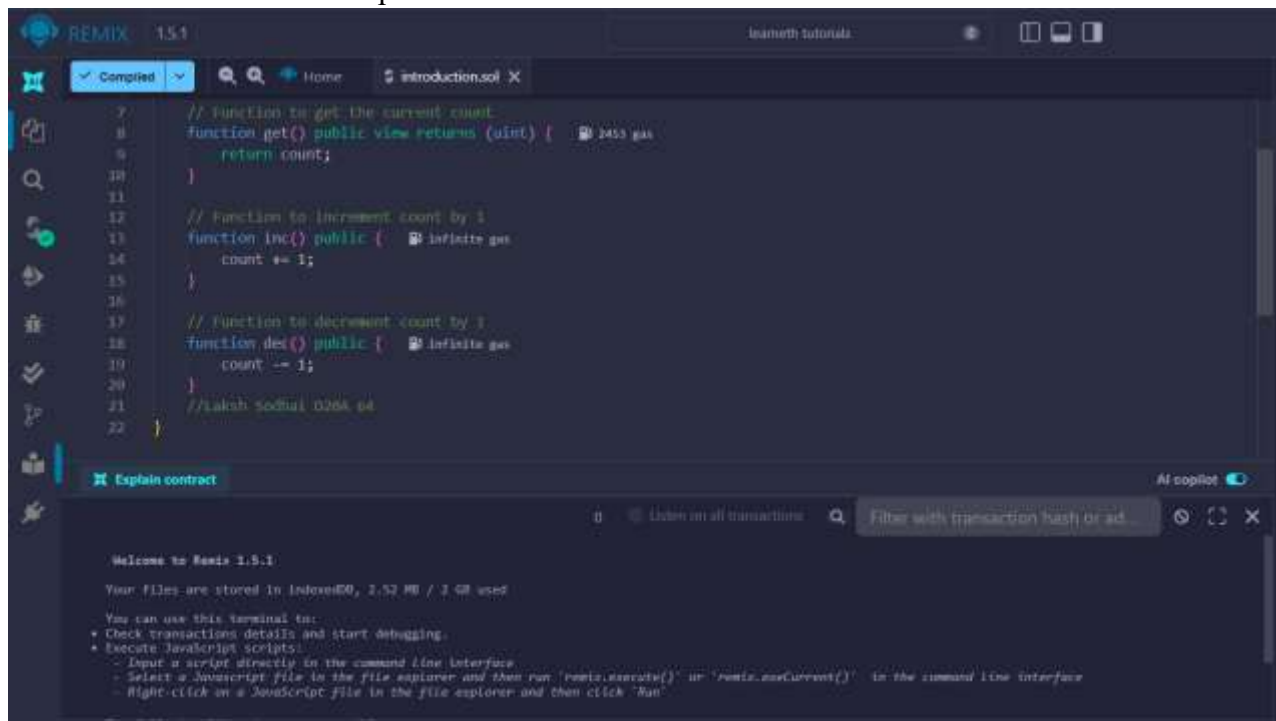
Understanding data locations is essential, as they directly impact gas costs and performance.

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

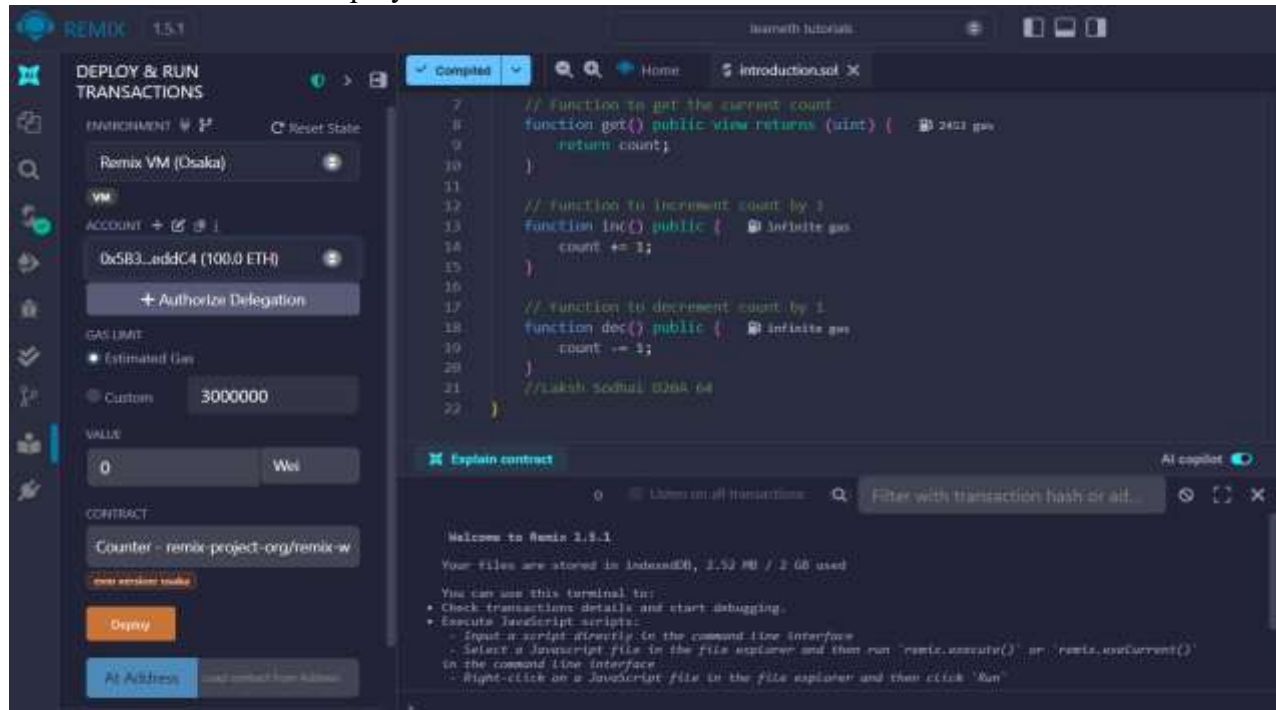
- **Ether and Wei:** Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether =  $10^{18}$  Wei). This ensures high precision in financial transactions.
- **Gas and Gas Price:** Every transaction consumes gas, which represents computational effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.
- **Sending Transactions:** Transactions are used for transferring Ether or interacting with contracts. Functions like `transfer()` and `send()` are commonly used, while `call()` provides more flexibility. Each transaction requires gas, making efficiency in contract design very important.

### Implementation:

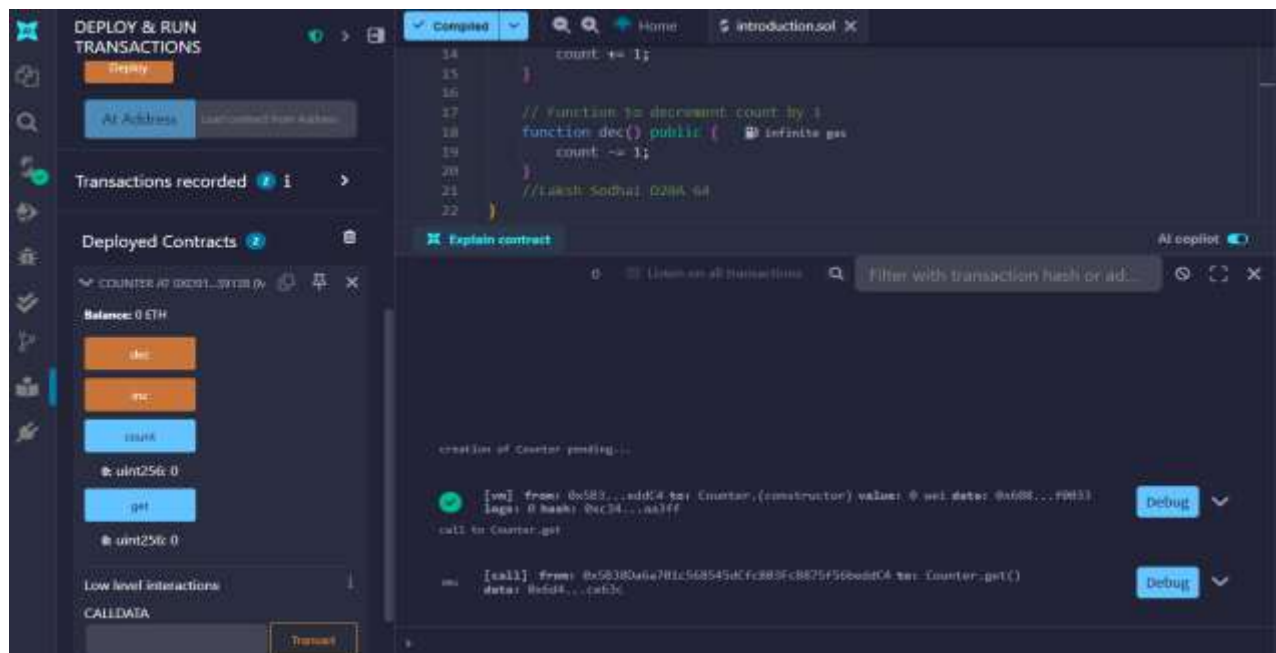
- Tutorial no. 1 – Compile the code



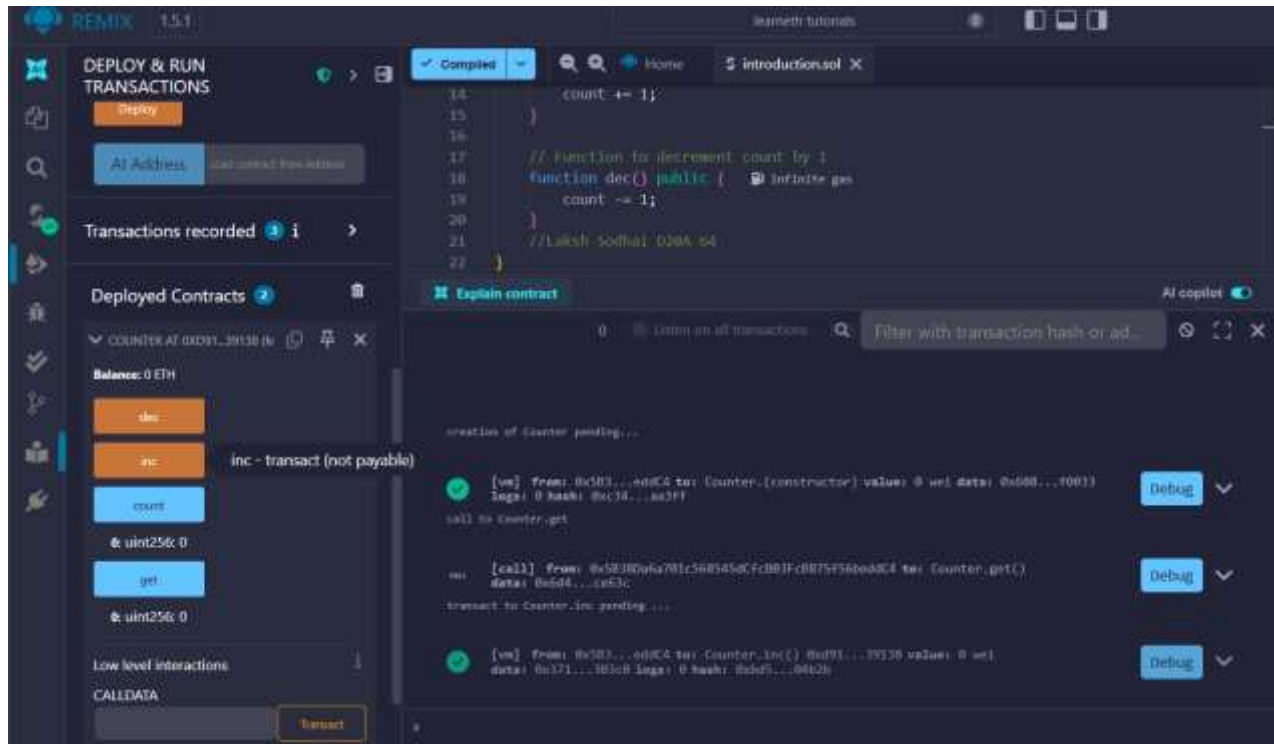
- Tutorial no. 1 – Deploy the contract



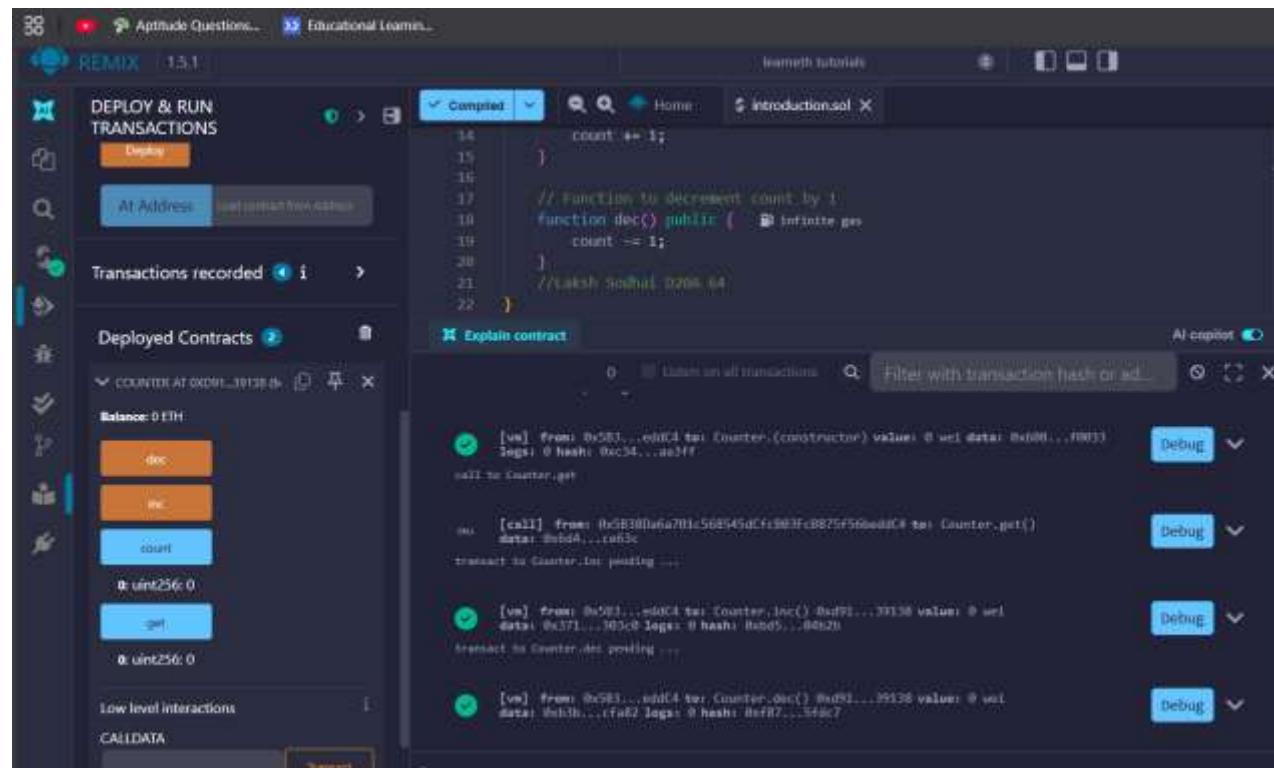
- Tutorial no. 1 – get



Laksh Sodhai D20A 64  
Tutorial no. 1 – Increment



- Tutorial no. 1 – Decrement



- Tutorial no.

2

**LEARNETH**

**2. Basic Syntax**

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.

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

Well done! No errors.

**Explain contract:**

Welcome to Basic 1.2.1.

Your files are stored in `learneth`, 1.50 MB / 2.0 GB used.

You can use this terminal to:

- Check transaction details and start debugging.
- Execute JavaScript scripts.

Drop a script directly in the command line interface.

• Select a JavaScript file in the file explorer and then run `run:execute()` or `run:execute()` in the command line interface.

• Right click on a JavaScript file in the file explorer and then click "Run".

The following libraries are accessible:

- `ethers.js`

Type the library name to see available commands.

- Tutorial no. 3

**LEARNETH**

**3. Primitive Data Types**

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 `balance` that is a `uint256` and give it a value that is not the same as the available variable `balance`.
2. Create a `public` variable called `age` that is a negative number, decide upon the type.
3. Create a new variable, `addr`, that has the smallest `address` 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**

Well done! No errors.

**Explain contract:**

Welcome to Basic 1.3.1.

Your files are stored in `learneth`, 1.52 MB / 2.0 GB used.

You can use this terminal to:

- Check transaction details and start debugging.
- Execute JavaScript scripts.

Drop a script directly in the command line interface.

• Select a JavaScript file in the file explorer and then run `run:execute()` or `run:execute()` in the command line interface.

• Right click on a JavaScript file in the file explorer and then click "Run".

The following libraries are accessible:

- `ethers.js`

Type the library name to see available commands.

- Tutorial no.

4

**LEARNETH** 1.5.1

**Tutorial no. 4**

**4 Variables**

about the blockchain, private addresses, contracts, and transactions.  
In this example, we use `block.timestamp` (line 34) to get a time timestamp of when the current block was generated and `block.number` (line 35) to get the value of the current block'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 `blackNumber`.
2. Inside the function `showSomething()`, assign the value of the current block number to the state variable `blackNumber`.

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

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

Next

Well done! No errors.

**Explain contract**

**Variables no. Basic 1.3.1**

Your files are stored in localStorage, 1.32 MB / 1 GB used.

You can use this terminal to:

- Check transaction details and start debugging.
- Create JavaScript scripts.
- Inject a script directly in the console like interface.
- Select a JavaScript file in the file explorer and then run `webpack.execute()` or `webpack.execute()` in the command line interface.
- Right-click on a JavaScript file in the file explorer and then click "Run".

The following libraries are accessible:

- `ethers.js`

Type the library name to use available comments.

- Tutorial no. 5

**LEARNETH** 1.5.1

**Tutorial no. 5**

**5.1 Functions - Reading and Writing to a State Variable**

Public state variables

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

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

Watch a video tutorial on [Functions](#).

**Assignment**

1. Create a public state variable called `h` that is of type `uint` and initialize it to `0`.
2. Create a public function called `get_h` that returns the value of `h`.

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

Next

Well done! No errors.

**Explain contract**

**SimpleStorage no. Basic 1.3.1**

Your files are stored in localStorage, 1.32 MB / 1 GB used.

You can use this terminal to:

- Check transaction details and start debugging.
- Create JavaScript scripts.
- Inject a script directly in the console like interface.
- Select a JavaScript file in the file explorer and then run `webpack.execute()` or `webpack.execute()` in the command line interface.
- Right-click on a JavaScript file in the file explorer and then click "Run".

The following libraries are accessible:

- `ethers.js`

Type the library name to use available comments.



- Tutorial no.

6

**LEARNETH** 1.3.1

**5.2 Functions - View and Pure** 2/19

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

In Solidity development, you need to optimize 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 `addTwo` that takes the parameter `x` 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.

**Explain contract**

Welcome to Remix 1.3.1.

Your files are stored in localStorage, 1.32 MB / 2 GB used.

You can use this tutorial too:

- Check transaction details and start debugging.
- Enable Immortals script.
- Deploy a contract directly to the command line interface.
- Select a transaction file in the file explorer and then use "repl.execute()" or "repl.execute()" in the command line interface.
- Signatures in a Solidity file to the file explorer and then click "Run".

The following functions are available:

- `addTwo`

Type the library name to see available commands.

- Tutorial no. 7

**LEARNETH** 1.3.1

**5.3 Functions - Modifiers and Constructors** 2/19

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 typically 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 10) sets the initial value of the state variable upon the creation of the contract.

Watch a video tutorial on Function Modifiers.

**Assignment**

1. Create a new function, `increment`, 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 `increment` should be empty.

Tip: the modifiers.

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

Next

Well done! No errors.

**Explain contract**

Welcome to Remix 1.3.1.

Your files are stored in localStorage, 1.32 MB / 2 GB used.

You can use this tutorial too:

- Check transaction details and start debugging.
- Enable Immortals script.
- Deploy a contract directly to the command line interface.
- Select a transaction file in the file explorer and then use "repl.execute()" or "repl.execute()" in the command line interface.



- Tutorial no.

Tutorial No: 08

The screenshot displays the Remix IDE interface with a dark theme. On the left, the 'LEARNETH' sidebar is open, showing a 'Tutorial List' with '5.4 Functions - Inputs and Outputs' selected. Below this, an 'Assignment' section prompts the user to create a function named `returnTest` that returns the values `10` and `true` without using a `return` statement. The main editor area shows a Solidity file named `main.sol` with the following code:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Function {
5
6     function returnTest() public pure returns (int a, bool b) {
7         a = 10;
8         b = true;
9     }
10 }
11 // Laksh Sodhai D20A 64
12
```

Below the code editor, the 'EXPLAIN CONTRACT' panel is visible, displaying a welcome message for Remix 1.5.1 and instructions on how to use the IDE, including checking transaction details and running JavaScript scripts. The bottom status bar indicates 'Well done! No errors.'



- Tutorial no.
- Tutorial no. 11

**LEARNETH** 1.3.1

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

**1.2 Control Flow - Loops** (1 / 10)

**continue**

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

**break**

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

Watch a video tutorial on [break statements](#).

**Assignment**

1. Create a public `uint` state variable called `count` in the `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 5, but make sure you don't edit the `break` statement.

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

Next

Well done! No errors.

**Explain contract**

Welcome to Revmick 1.3.1.

Your files are stored in `datastore`, 1.32 MB / 2 GB used.

You can use this terminal to:

- Check transaction details and start debugging.
- Inspect JavaScript artifacts.
- Inject a script directly to the command line interface.
- Select a JavaScript file in the file explorer and then run `injectScript()` or `injectArtifact()` in the command line interface.

12

**LEARNETH** 1.3.1

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

**1.1 Data Structures - Arrays** (1 / 10)

We can use the `length` property to know the number of elements stored in an array. In this contract, we use `length` (line 42). When we remove an element with the `splice` 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 remove the last element of the array to the place of the deleted element (line 48), 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-size array called `arr` with the values 0, 1, 2. Make the size as small as possible.
2. Change the `getMessage()` function to return the value of `arr`.

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

Next

Well done! No errors.

**Explain contract**

Welcome to Revmick 1.3.1.

Your files are stored in `datastore`, 1.32 MB / 2 GB used.

You can use this terminal to:

- Check transaction details and start debugging.
- Inspect JavaScript artifacts.
- Inject a script directly to the command line interface.
- Select a JavaScript file in the file explorer and then run `injectScript()` or `injectArtifact()` in the command line interface.

- Tutorial no.
- Tutorial no. 13

**LEARNETH** 13.1

**8.2 Data Structures - Mappings** 13/19

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

**Removing values**

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

Watch a video tutorial on [Mappings](#).

**Assignment**

1. Create a public mapping `balances` that associates the key type `address` with the value type `uint`.
2. Change the functions `get` and `set` to work with the mapping balances.
3. Change the function `add` 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.

```

30 // You can get values from a stored mapping
31 // even when it is not initialized
32 return nested[_addr][_i];
33
34
35 function add() { // done go
36     address _addr;
37     uint _i;
38     bool _bnc;
39     public {
40         nested[_addr][_i] = _bnc;
41     }
42
43
44 function remove(address _addr, uint _i) public { // done go
45     delete nested[_addr][_i];
46 }
47
48 //Laksh Sodhai D20A 64
49
50

```

**Explain context**

Welcome to Reinx 13.1.

Your files are stored in /home/13.1 (0 / 3 GB used).

You can use this terminal for:

- Check transactions details web start debugging.
- Execute JavaScript scripts.
- Deploy a script directly to the current live interface.

Select a JavaScript file in the file explorer and then run "inject.execute()" or "inject.exclude()". In the console live interface.

14

**LEARNETH** 13.1

**8.3 Data Structures - Structs** 14/19

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 struct's 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 `tasks` mapping.

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

Next

Well done! No errors.

```

35 // update task
36 function update(uint _index, string memory _task) public { // initdone go
37     tasks[_index] = _task;
38 }
39
40
41 // update completed
42 function toggleCompleted(uint _index) public { // done go
43     tasks[_index].completed = !tasks[_index].completed;
44 }
45
46
47 function remove(uint _index) public { // initdone go
48     delete tasks[_index];
49 }
50
51 //Laksh Sodhai D20A 64
52

```

**Explain context**

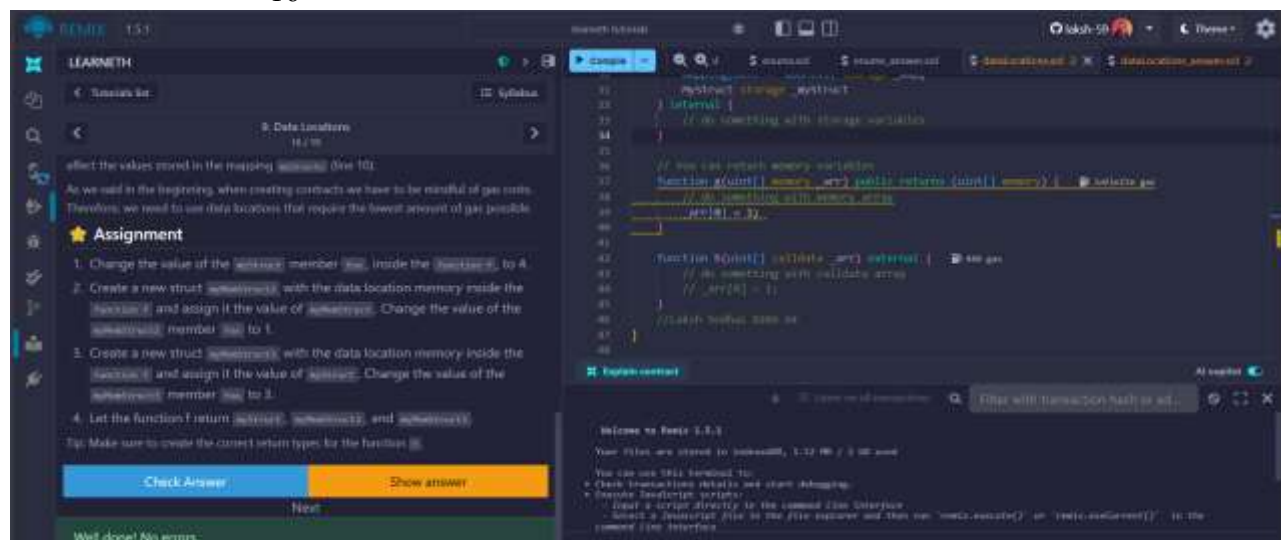
Welcome to Reinx 13.1.

Your files are stored in /home/13.1 (0 / 3 GB used).

You can use this terminal for:

- Check transactions details web start debugging.
- Execute JavaScript scripts.
- Deploy a script directly to the current live interface.

Select a JavaScript file in the file explorer and then run "inject.execute()" or "inject.exclude()". In the console live interface.



- Tutorial no.
- Tutorial no. 17

The screenshot shows the Remix IDE interface. On the left, the 'LEARNETH' tutorial is open, displaying a list of transactions and an assignment. The main editor shows a Solidity contract with the following code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract Etharthritis {
5     uint public count1 = 1 wei;
6     // 1 wei is equal to 1
7     bool public iscount1 = 1 wei == 1;
8
9     uint public count2 = 1 ether;
10    // 1 ether is equal to 10^18 wei
11    bool public iscount2 = 1 ether == 1e18;
12
13    uint public count3 = 1 gwei;
14    // 1 gwei is equal to 10^9 wei
15    bool public iscount3 = 1 gwei == 1e9;
16
17    // Learn Solidity 100%
18 }

```

Below the code, the 'Explain contract' section is visible, providing a summary of the contract's functionality.

18

The screenshot shows the Remix IDE interface. On the left, the 'LEARNETH' tutorial is open, displaying a list of transactions and an assignment. The main editor shows a Solidity contract with the following code:

```

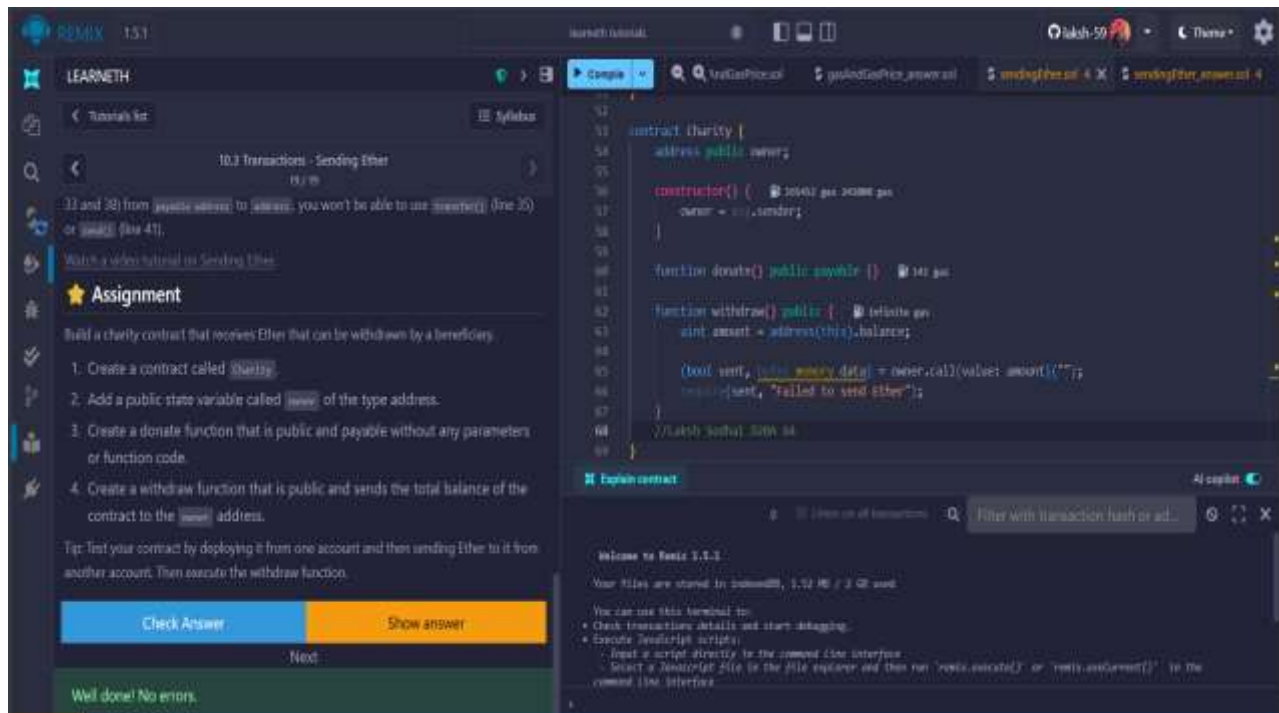
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract Gas {
5     uint public i = 0;
6     uint public cost = 100000;
7
8     // Using up all of the gas that you send causes your transaction to fail.
9     // State changes are undone.
10    // Gas spent are not refunded.
11    function forward() 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
19    // Learn Solidity 100%
20 }

```

Below the code, the 'Explain contract' section is visible, providing a summary of the contract's functionality.



- Tutorial no.
- 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.