

BLOCKCHAIN EXP 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.
 - **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.

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

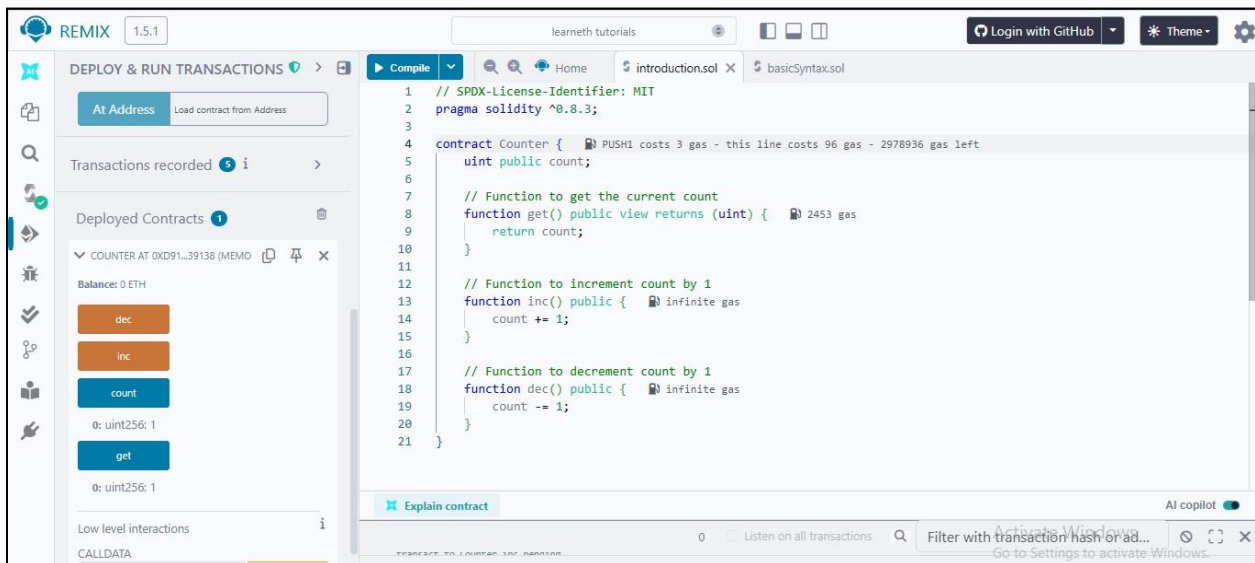
- **Ether and Wei:** Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.

- **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.

Code & Output -

Tutorial 1

1. Get counter value



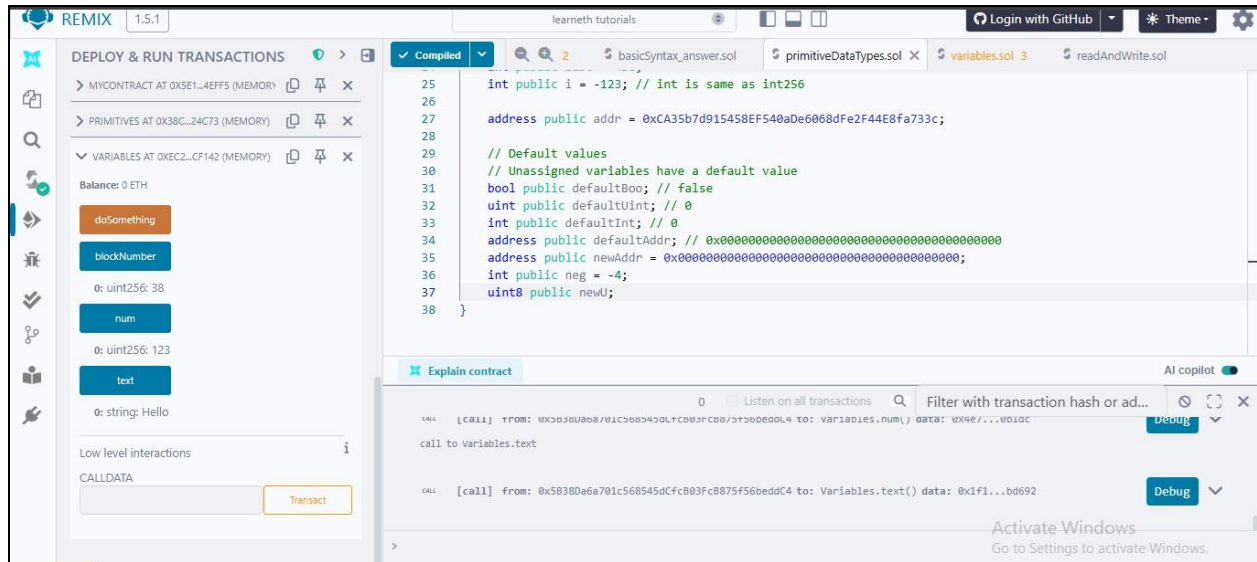
The screenshot displays the REMIX IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel shows a deployed contract named 'COUNTER AT 0xD91...39138 (MEMO)'. The contract's balance is 0 ETH. Below this, there are buttons for 'dec', 'inc', 'count', and 'get'. The 'count' button is highlighted, and the output shows '0: uint256: 1'. The main editor displays the Solidity code for the 'Counter' contract, which includes a public variable 'count' and three functions: 'get()', 'inc()', and 'dec()'. The 'get()' function returns the current value of 'count'. The 'inc()' function increments 'count' by 1, and the 'dec()' function decrements 'count' by 1. The code is as follows:

```

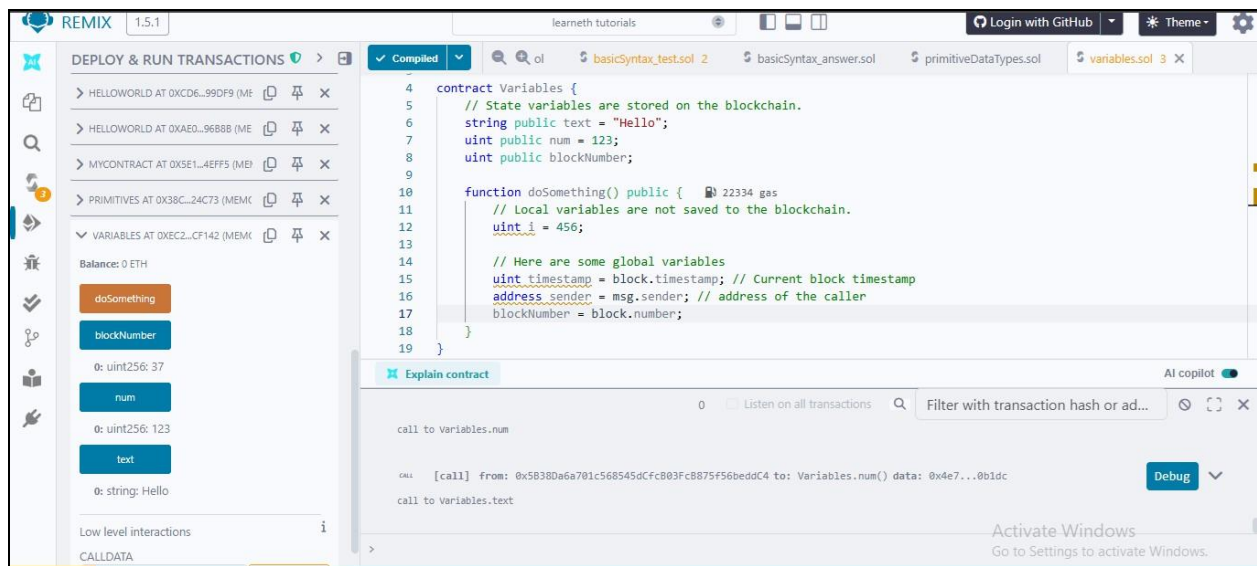
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     uint public count;
6
7     // Function to get the current count
8     function get() public view returns (uint) {
9         return count;
10    }
11
12    // Function to increment count by 1
13    function inc() public {
14        count += 1;
15    }
16
17    // Function to decrement count by 1
18    function dec() public {
19        count -= 1;
20    }
21 }

```

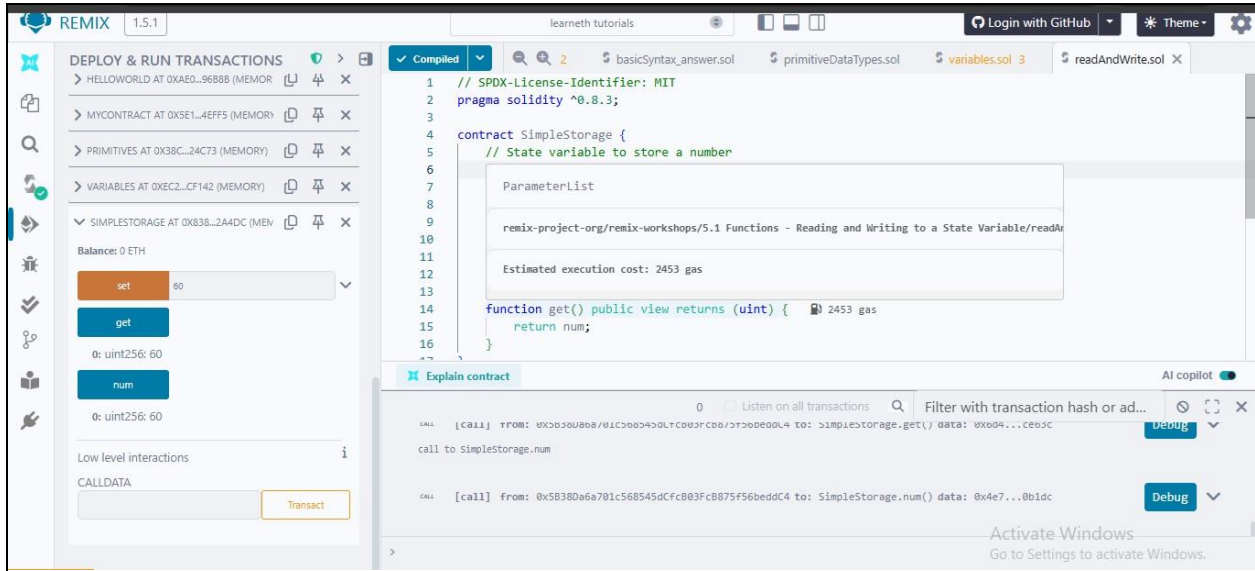
At the bottom of the interface, there is a search bar for transactions and a filter option.



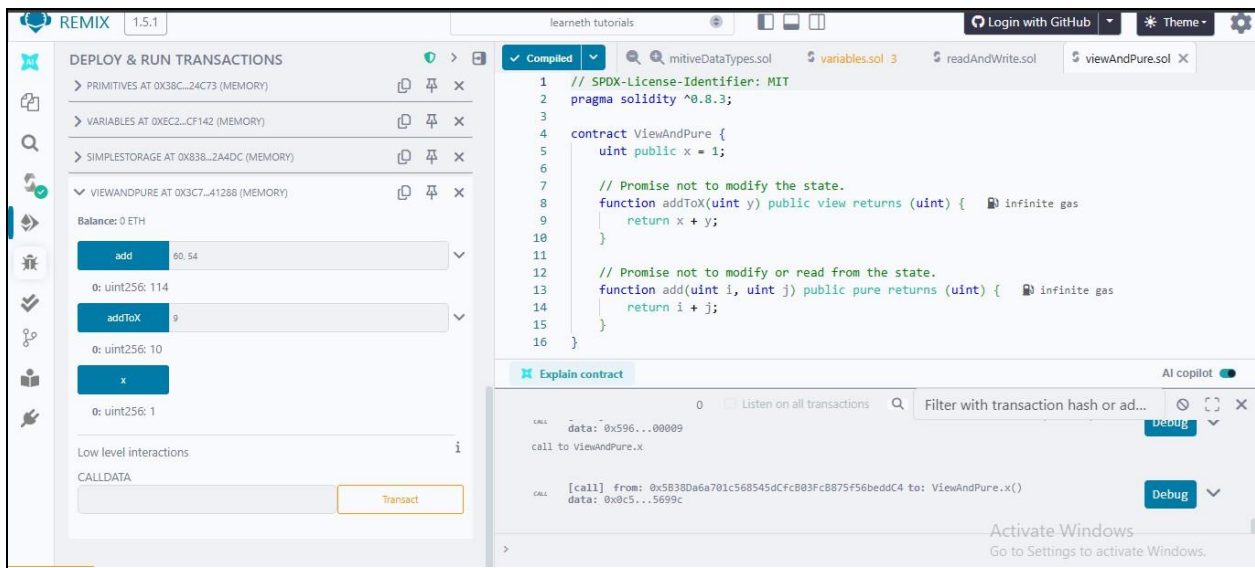
Tutorial 4



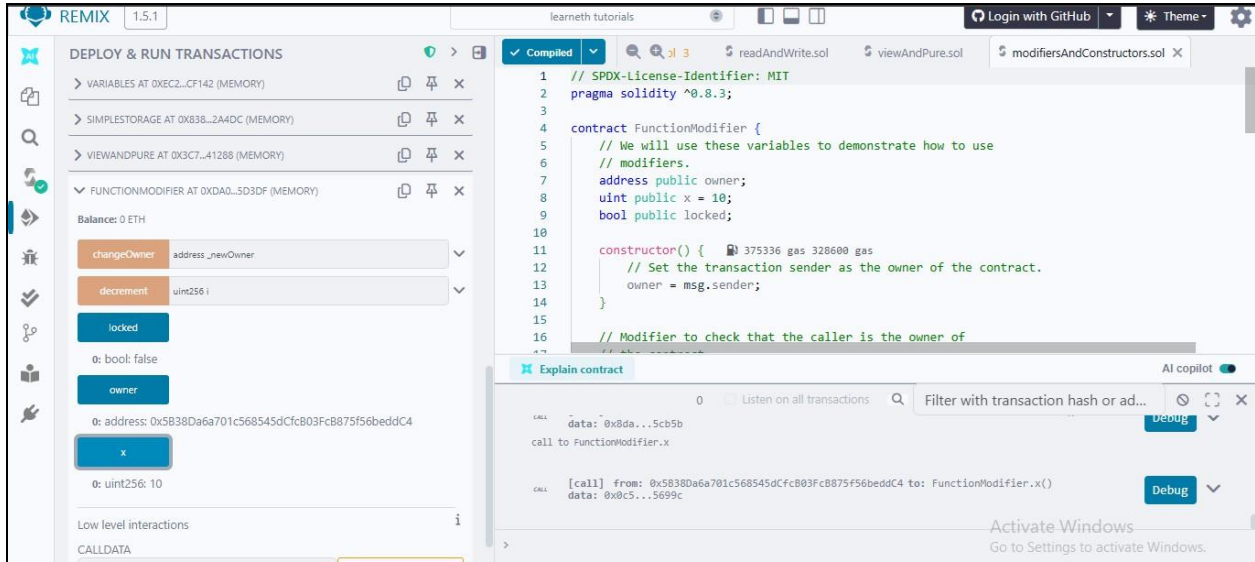
Tutorial 5



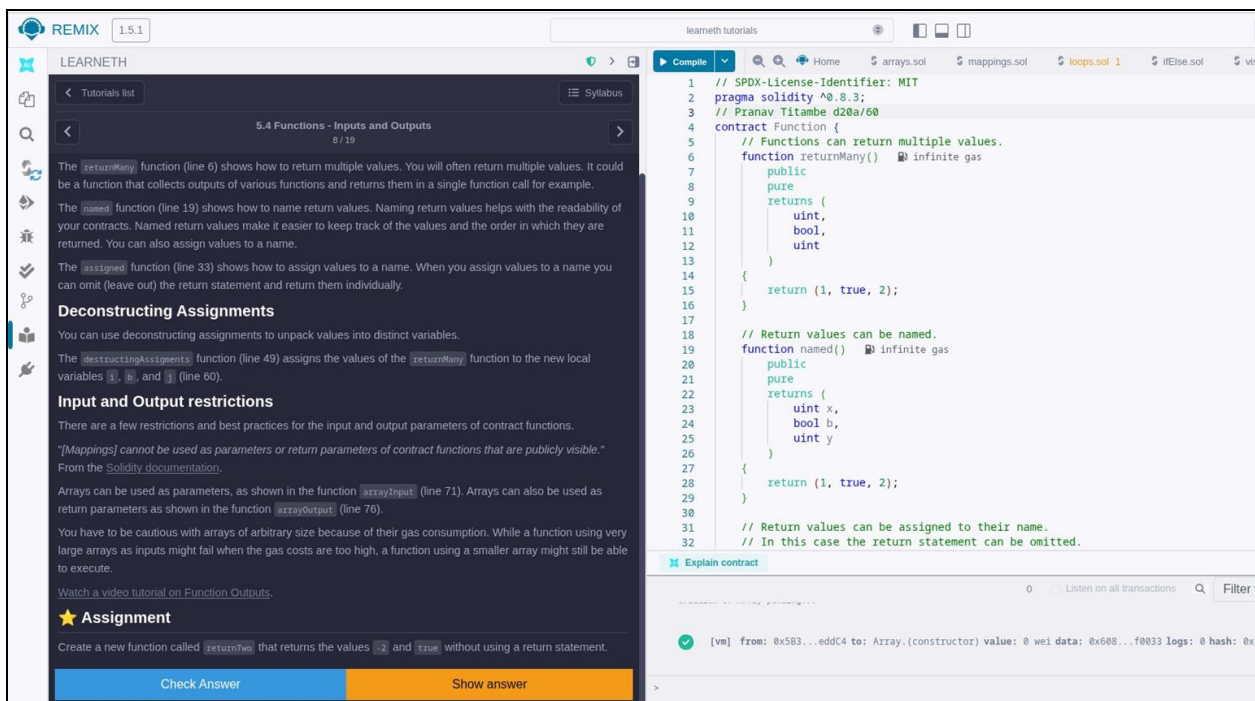
Tutorial 6



Tutorial 7



Tutorial 8



Tutorial 9

6. Visibility

The `visibility` specifier is used to control who has access to functions and state variables.

There are four types of visibilities: `external`, `public`, `internal`, and `private`.

They regulate if functions and state variables can be called from inside the contract, from contracts that derive from the contract (child contracts), or from other contracts and transactions.

private

- Can be called from inside the contract

internal

- Can be called from inside the contract
- Can be called from a child contract

public

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

external

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

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

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

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

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Base {
7     // Private function can only be called
8     // - inside this contract
9     // - Contracts that inherit this contract cannot call this function.
10    function privateFunc() private pure returns (string memory) {
11        return "private function called";
12    }
13
14    function testPrivateFunc() public pure returns (string memory) {
15        return privateFunc();
16    }
17
18    // Internal function can be called
19    // - inside this contract
20    // - inside contracts that inherit this contract
21    function internalFunc() internal pure returns (string memory) {
22        return "internal function called";
23    }
24
25    function testInternalFunc() public pure virtual returns (string memory) {
26        return internalFunc();
27    }
28
29    // Public functions can be called
30    // - inside this contract
31    // - inside contracts that inherit this contract
32    // - by other contracts and accounts
```

Tutorial 10

7.1 Control Flow - If/Else

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

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

if

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

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

else

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

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

else if

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

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

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract IfElse {
7     function foo(uint x) public pure returns (uint) {
8         if (x < 10) {
9             return 0;
10        } else if (x < 20) {
11            return 1;
12        } else {
13            return 2;
14        }
15    }
16
17    function ternary(uint _x) public pure returns (uint) {
18        // if (_x < 10) {
19        //     return 1;
20        // }
21        // return 2;
22
23        // shorthand way to write if / else statement
24        return _x < 10 ? 1 : 2;
25    }
26 }
```

Tutorial 11

REMX1.5.1

learneth tutorials

LEARNETH

Tutorials list

Syllabus

7.2 Control Flow - Loops11 / 19

7.2 Control Flow - Loops

Solidity supports iterative control flow statements that allow contracts to execute code repeatedly. Solidity differentiates between three types of loops: `for`, `while`, and `do while` loops.

for

Generally, `for` loops (line 7) are great if you know how many times you want to execute a certain block of code. In solidity, you should specify this amount to avoid transactions running out of gas and failing if the amount of iterations is too high.

while

If you don't know how many times you want to execute the code but want to break the loop based on a condition, you can use a `while` loop (line 20). Loops are seldom used in Solidity since transactions might run out of gas and fail if there is no limit to the number of iterations that can occur.

do while

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

continue

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

break

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

Watch a video tutorial on Loop statements.

★ Assignment

1. Create a public `uint` state variable called count in the `Loop` contract.

Compile

Home

arrays.sol

mappings.sol

loops.sol 1 x

ifElse.sol

1 // SPDX-License-Identifier: MIT

2 pragma solidity ^0.8.3;

3 // Pranav Titambe d20a/60

4

5 contract Loop {

6 function loop() public {

7 // for loop

8 for (uint i = 0; i < 10; i++) {

9 if (i == 3) {

10 // Skip to next iteration with continue

11 continue;

12 }

13 if (i == 5) {

14 // Exit loop with break

15 break;

16 }

17 }

18 }

19 }

20

21 // while loop

22 uint i;

23 while (i < 10) {

24 i++;

25 }

26 }

27

Explain contract

0 Listen on all transactions

Filter

[vm] from: 0x583...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash:

Tutorial 12

REMX1.5.1

learneth tutorials

LEARNETH

Tutorials list

Syllabus

8.1 Data Structures - Arrays12 / 19

8.1 Data Structures - Arrays

empty (line 6).

Initializing arrays

We can initialize the elements of an array all at once (line 7), or initiate new elements one by one (line 9). If we declare an array, we automatically initialize its elements with the default value 0 (line 9).

Accessing array elements

We access elements inside an array by providing the name of the array and the index in brackets (line 12).

Adding array elements

Using the `push()` member function, we add an element to the end of a dynamic array (line 25).

Removing array elements

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

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

Array length

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

Watch a video tutorial on Arrays.

★ Assignment

1. Initialize a public fixed-sized array called `arr3` with the values 0, 1, 2. Make the size as small as possible.

2. Change the `getArr()` function to return the value of `arr3`.

Compiled

Home

arrays.sol x

1 // SPDX-License-Identifier: MIT

2 pragma solidity ^0.8.3;

3 // Pranav Titambe d20a/60

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

11 function get(uint i) public view returns (uint) {

12 return arr[i];

13 }

14

15 // Solidity can return the entire array.

16 // But this function should be avoided for

17 // arrays that can grow indefinitely in length.

18 function getArr() public view returns (uint[] memory) {

19 return arr;

20 }

21

22 function push(uint i) public {

23 // Append to array

24 // This will increase the array length by 1.

25 arr.push(i);

26 }

27

28 function pop() public {

29 // Remove last element from array

30 // This will decrease the array length by 1

31 arr.pop();

32 }

33 }

Explain contract

0 Listen on all transactions

Filter with transaction hash or

[vm] from: 0x583...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6e...7f733

Check Answer

Show answer

Tutorial 13

8.2 Data Structures - Mappings

In Solidity, *mappings* are a collection of key types and corresponding value type pairs.

The biggest difference between a mapping and an array is that you can't iterate over mappings. If we don't know a key we won't be able to access its value. If we need to know all of our data or iterate over it, we should use an array.

If we want to retrieve a value based on a known key we can use a mapping (e.g. addresses are often used as keys). Looking up values with a mapping is easier and cheaper than iterating over arrays. If arrays become too large, the gas cost of iterating over it could become too high and cause the transaction to fail.

We could also store the keys of a mapping in an array that we can iterate over.

Creating mappings

Mappings are declared with the syntax `mapping(KeyType => ValueType) VariableName`. The key type can be any built-in value type or any contract, but not a reference type. The value type can be of any type.

In this contract, we are creating the public mapping `myMap` (line 6) that associates the key type `address` with the value type `uint`.

Accessing values

The syntax for interacting with key-value pairs of mappings is similar to that of arrays. To find the value associated with a specific key, we provide the name of the mapping and the key in brackets (line 11).

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

Setting values

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

Removing values

We can use the delete operator to delete a value associated with a key, which will set it to the default value of 0. As

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Mapping {
7     // Mapping from address to uint
8     mapping(address => uint) public myMap;
9
10    function get(address _addr) public view returns (uint) {
11        // Mapping always returns a value.
12        // If the value was never set, it will return the default value.
13        return myMap[_addr];
14    }
15
16    function set(address _addr, uint _i) public {
17        // Update the value at this address
18        myMap[_addr] = _i;
19    }
20
21    function remove(address _addr) public {
22        // Reset the value to the default value.
23        delete myMap[_addr];
24    }
25 }
26
27 contract NestedMapping {
28     // Nested mapping (mapping from address to another mapping)
29     mapping(address => mapping(uint => bool)) public nested;
30
31     function get(address _addr, uint _i) public view returns (bool) {
32         // You can get values from a nested mapping
```

Explain contract

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash:

Tutorial 14

8.3 Data Structures - Structs

In Solidity, we can define custom data types in the form of *structs*. Structs are a collection of variables that can consist of different data types.

Defining structs

We define a struct using the `struct` keyword and a name (line 5). Inside curly braces, we can define our struct's members, which consist of the variable names and their data types.

Initializing structs

There are different ways to initialize a struct.

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

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

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

Accessing structs

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

Updating structs

To update a 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 `todos` mapping.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Todos {
7     struct Todo {
8         string text;
9         bool completed;
10    }
11
12    // An array of 'Todo' structs
13    Todo[] public todos;
14
15    function create(string memory _text) public {
16        // 3 ways to initialize a struct
17        // - calling it like a function
18        todos.push(Todo(_text, false));
19
20        // key value mapping
21        todos.push(Todo({text: _text, completed: false}));
22
23        // initialize an empty struct and then update it
24        Todo memory todo;
25        todo.text = _text;
26        todo.completed = false;
27        todos.push(todo);
28    }
29
30    // Solidity automatically created a getter for 'todos' so
31    // you don't actually need this function.
```

Explain contract

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

Tutorial 15

8.4 Data Structures - Enums

In Solidity *enums* are custom data types consisting of a limited set of constant values. We use enums when our variables should only get assigned a value from a predefined set of values.

In this contract, the state variable `status` can get assigned a value from the limited set of provided values of the enum `Status` representing the various states of a shipping status.

Defining enums

We define an enum with the `enum` keyword, followed by the name of the custom type we want to create (line 6). Inside the curly braces, we define all available members of the enum.

Initializing an enum variable

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

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

Accessing an enum value

To access the enum value of a variable, we simply need to provide the name of the variable that is storing the value (line 25).

Updating an enum value

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.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Enum {
7     // Enum representing shipping status
8     enum Status {
9         Pending,
10        Shipped,
11        Accepted,
12        Rejected,
13        Canceled
14    }
15
16    // Default value is the first element listed in
17    // definition of the type, in this case "Pending"
18    Status public status;
19
20    // Returns uint
21    // Pending - 0
22    // Shipped - 1
23    // Accepted - 2
24    // Rejected - 3
25    // Canceled - 4
26    function get() public view returns (Status) { 2582 gas
27        return status;
28    }
29
30    // Update status by passing uint into input
31    function set(Status _status) public { undefined gas
32        status = _status;
33    }
34 }
```

Tutorial 16

9. Data Locations

The values of variables in Solidity can be stored in different data locations: *memory*, *storage*, and *calldata*.

As we have discussed before, variables of the value type store an independent copy of a value, while variables of the reference type (array, struct, mapping) only store the location (reference) of the value.

If we use a reference type in a function, we have to specify in which data location their values are stored. The price for the execution of the function is influenced by the data location; creating copies from reference types costs gas.

Storage

Values stored in *storage* are stored permanently on the blockchain and, therefore, are expensive to use.

In this contract, the state variables `arr`, `map`, and `myStructs` (lines 5, 6, and 10) are stored in storage. State variables are always stored in storage.

Memory

Values stored in *memory* are only stored temporarily and are not on the blockchain. They only exist during the execution of an external function and are discarded afterward. They are cheaper to use than values stored in *storage*.

In this contract, the local variable `myMemStruct` (line 19), as well as the parameter `_arr` (line 31), are stored in memory. Function parameters need to have the data location *memory* or *calldata*.

Calldata

Calldata stores function arguments. Like *memory*, *calldata* is only stored temporarily during the execution of an external function. In contrast to values stored in *memory*, values stored in *calldata* can not be changed. *Calldata* is the cheapest data location to use.

In this contract, the parameter `_arr` (line 35) has the data location *calldata*. If we wanted to assign a new value to the first element of the array `arr`, we could do that in the `function g` (line 31) but not in the `function h` (line 35). This is because `_arr` in `function g` has the data location *memory* and `function h` has the data location *calldata*.

Assignments

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract DataLocations {
7     uint[] public arr;
8     mapping(uint => address) map;
9     struct MyStruct {
10         uint foo;
11     }
12     mapping(uint => MyStruct) myStructs;
13
14     function f() public { 372 gas
15         // call _f with state variables
16         _f(arr, map, myStructs[1]);
17     }
18
19     // get a struct from a mapping
20     MyStruct storage myStruct = myStructs[1];
21     // create a struct in memory
22     MyStruct memory myMemStruct = MyStruct(0);
23
24     function _f( undefined gas
25         uint[] storage _arr,
26         mapping(uint => address) storage _map,
27         MyStruct storage _myStruct
28     ) internal {
29         // do something with storage variables
30     }
31
32     // You can return memory variables
33 }
```

Tutorial 17

REMIX 1.5.1

learnmeth tutorials

Compile

Else.sol

visibility.sol

inputsAndOutputs.sol

structs.sol

LEARNETH

Tutorials list

Syllabus

10.1 Transactions - Ether and Wei

17 / 19

10.1 Transactions - Ether and Wei

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

Ether Units

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

`wei`

Wei is the smallest subunit of *Ether*, named after the cryptographer Wei Dai. *Ether* numbers without a suffix are treated as `wei` (line 7).

`gwei`

One `gwei` (giga-wei) is equal to 1,000,000,000 (10^9) `wei`.

`ether`

One `ether` is equal to 1,000,000,000,000,000 (10^{18}) `wei` (line 11).

[Watch a video tutorial on Ether and Wei.](#)

★ Assignment

- Create a `public uint` called `oneGwei` and set it to 1 `gwei`.
- 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

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract EtherUnits {
7     uint public oneWei = 1 wei;
8     // 1 wei is equal to 1
9     bool public isOneWei = 1 wei == 1;
10
11     uint public oneEther = 1 ether;
12     // 1 ether is equal to 10^18 wei
13     bool public isOneEther = 1 ether == 1e18;
14 }

```

Explain contract

0 ☐ Listen on all transactions

from: 0x5B3...edc4 to: Array.constructor value: 0 wei data: 0x608...f0033

Tutorial 18

REMIX 1.5.1

learnmeth tutorials

Compile

inputsAndOutputs.sol

structs.sol

enums.sol

data

LEARNETH

Tutorials list

Syllabus

10.2 Transactions - Gas and Gas Price

18 / 19

10.2 Transactions - Gas and Gas Price

As we have seen in the previous section, executing code via transactions on the Ethereum Network costs transaction fees in the form of Ether. The amount of fees that have to be paid to execute a transaction depends on the amount of gas that the execution of the transaction costs.

Gas

Gas is the unit that measures the amount of computational effort that is required to execute a specific operation on the Ethereum network.

Gas price

The gas that fuels Ethereum is sometimes compared to the gas that fuels a car. The amount of gas your car consumes is mostly the same, but the price you pay for gas depends on the market.

Similarly, the amount of gas that a transaction requires is always the same for the same computational work that is associated with it. However the price that the sender of the transaction is willing to pay for the gas is up to them. Transactions with higher *gas prices* are going through faster; transactions with very low *gas prices* might not go through at all.

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

Gas prices are denoted in gwei.

Gas limit

When sending a transaction, the sender specifies the maximum amount of gas that they are willing to pay for. If they set the limit too low, their transaction can run out of gas before being completed, reverting any changes being made. In this case, the gas was consumed and can't be refunded.

[Learn more about gas on ethereum.org.](#)

[Watch a video tutorial on Gas and Gas Price.](#)

★ Assignment

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Gas {
7     uint public i = 0;
8
9     // Using up all of the gas that you send causes your transaction
10    // State changes are undone.
11    // Gas spent are not refunded.
12    function forever() public {
13        // Here we run a loop until all of the gas are spent
14        // and the transaction fails
15        while (true) {
16            i += 1;
17        }
18    }
19 }

```

Explain contract

0 ☐ Listen on all transactions

from: 0x5B3...edc4 to: Array.constructor value: 0 wei data: 0x608...f0033

Tutorial 19

The screenshot displays the Remix IDE interface. On the left, a sidebar shows the 'Tutorials List' with '10.3 Transactions - Sending Ether' selected. The main panel on the left contains the tutorial content, which explains three methods for sending Ether: `transfer()`, `send()`, and `call()`. It includes code snippets for each method and their respective behaviors, such as gas stipends and error handling. The right panel shows a Solidity contract named `ReceiveEther` with a `receive()` function and a `fallback()` function. The contract is compiled, and the bottom panel shows the 'Explain contract' section with a transaction log entry: '[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033'.

10.3 Transactions - Sending Ether

In this section, we will learn how a contract can send and receive Ether.

Sending Ether

We have three different options to transfer Ether: `transfer()`, `send()` and `call()`.

transfer

`<address payable>.transfer(uint256 amount)`

- `transfer()` throws an exception on failure
- Forwards a fixed 2300 gas stipend

An example of `transfer()` can be seen in the `SendEther` contract (line 35). `Transfer()` is not recommended to be used anymore.

send

`<address payable>.send(uint256 amount) returns (bool)`

- `send()` returns false on failure
- Forwards a fixed 2300 gas stipend

An example of `send()` can be seen in the `SendEther` contract (line 41). `Send()` is not recommended to be used anymore.

call

`<address>.call(bytes memory) returns (bool, bytes memory)`

- `call()` returns false on failure
- Forwards the maximum amount of gas, but this is adjustable

An example of `call()` can be seen in the `SendEther` contract (line 48). `Call()` is currently recommended if you are transferring Ether.

The reason `transfer()` and `send()` were introduced was to guard against reentry attacks by limiting the forwarded

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract ReceiveEther {
7     /**
8      * Which function is called
9      */
10    send Ether
11    |
12    msg.data is empty?
13    | \
14    yes no
15    | \
16    receive() exists? fallback()
17    | \
18    yes no
19    | \
20    receive() fallback()
21    */
22
23    // Function to receive Ether. msg.data must be empty
24    receive() external payable {} // undefined gas
25
26    // Fallback function is called when msg.data is not empty
27    fallback() external payable {} // undefined gas
28
29    function getBalance() public view returns (uint) { // 312 gas
30        return address(this).balance;
31    }
32 }
```

Explain contract

0 Listen on all transactions

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

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.