

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:
 - o **public:** available both inside and outside the contract.
 - o **private:** only accessible within the same contract.
 - o **internal:** accessible within the contract and its child contracts.
 - o **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.

Implementation:

Tutorial - 1 (compile)

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract Counter {
5     uint public count;
6
7     // Function to get the current count
8     function get() public view returns (uint) { 2453 gas
9         return count;
10    }
11
12    // Function to increment count by 1
13    function inc() public { infinite gas
14        count += 1;
15    }
16
17    // Function to decrement count by 1
18    function dec() public { infinite gas
19        count -= 1;
20    }
21 }
```

Welcome to Remix 1.5.1

Your files are stored in indexedDB, 2.65 KB / 108.86 GB used

You can use this terminal to:

- Check transactions details and start debugging.
- Execute JavaScript scripts:
 - Input a script directly in the command line interface
 - Select a JavaScript file in the file explorer and then run ``remix.execute()`` or ``remix.executeCurrent()`` 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.

creation of Counter pending...



[vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0 hash: 0x955...0d83c

Debug



Tutorial - 1 (get)

COUNTER AT 0XD91...39138 (MEMORY)

Balance: 0 ETH

dec

inc

count

get

0: uint256: 0

Low level interactions

CALLDATA

Transact

You can use this terminal to:

- Check transactions details and start debugging.
- Execute JavaScript scripts:
 - Input a script directly in the command line interface
 - Select a JavaScript file in the file explorer and then run `remix.execute()` or `remix.executeCurrent()` 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.
creation of Counter pending...

✓

[vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0 hash: 0x955...0d83c

Debug

call to Counter.get

CALL

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

Debug

Tutorial - 1 (inc)

COUNTER AT 0XD91...39138 (MEMORY)

Balance: 0 ETH

dec

inc

count

get

0: uint256: 0

Low level interactions

CALLDATA

Transact

- 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.
creation of Counter pending...

✓

[vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0 hash: 0x955...0d83c

Debug

call to Counter.get

CALL

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

Debug

transact to Counter.inc pending ...

✓

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei
data: 0x371...303c0 logs: 0 hash: 0x3ec...ee5b5

Debug

Tutorial - 1 (dec)

COUNTER AT 0XD91...39138 (MEMORY)

Balance: 0 ETH

dec

inc

count

get

0: uint256: 0

Low level interactions

CALLDATA

Transact

Type the library name to see available commands.
creation of Counter pending...

✓

[vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei
data: 0x608...f0033 logs: 0 hash: 0x955...0d83c

Debug

call to Counter.get

CALL

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

Debug

transact to Counter.inc pending ...

✓

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei
data: 0x371...303c0 logs: 0 hash: 0x3ec...ee5b5

Debug

transact to Counter.dec pending ...

✓

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei
data: 0xb3b...cfa82 logs: 0 hash: 0xcc...79d46

Debug

Tutorial - 2

LEARNETH

Tutorials list

Syllabus

2. Basic Syntax

2 / 19

variable when you declare it. In this case, `greet` is a `string`.

We also define the *visibility* of the variable, which specifies from where you can access it. In this case, it's a `public` variable that you can access from inside and outside the contract.

Don't worry if you didn't understand some concepts like *visibility*, *data types*, or *state variables*. We will look into them in the following sections.

To help you understand the code, we will link in all following sections to video tutorials from the [creator](#) of the Solidity by Example contracts.

[Watch a video tutorial on Basic Syntax.](#)

★ **Assignment**

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

Check Answer

Show answer

Next

Well done! No errors.

Compile

introduction.sol

basicSyntax.sol

```
1 // SPDX-License-Identifier: MIT
2 // compiler version must be greater than or equal to 0.8.3 and
3 pragma solidity ^0.8.3;
4 //Ansh Sarfare D20A51
5 contract MyContract {
6 |   string public name = "Alice";
7 }
```

Tutorial - 3

LEARNETH

Tutorials list

Syllabus

3. Primitive Data Types

3 / 19

You can learn more about these data types as well as *Fixed Point Numbers*, *Byte Arrays*, *Strings*, and more in the [Solidity documentation](#).

Later in the course, we will look at data structures like **Mappings**, **Arrays**, **Enums**, and **Structs**.

[Watch a video tutorial on Primitive Data Types.](#)

★ **Assignment**

1. Create a new variable `newAddr` that is a `public` `address` and give it a value that is not the same as the available variable `addr`.
2. Create a `public` variable called `neg` that is a negative number, decide upon the type.
3. Create a new variable, `newU` that has the smallest `uint` size type and the smallest `uint` value and is `public`.

Tip: Look at the other address in the contract or search the internet for an Ethereum address.

Check Answer

Show answer

Next

Well done! No errors.

Compile

introduction.sol

basicSyntax.sol

primitiveD

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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
22   */
23   int8 public i8 = -1;
24   int public i256 = 456;
25   int public i = -123; // int is same as int256
26
27   address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;
28
29   // Default values
30   // Unassigned variables have a default value
31   bool public defaultBoo; // false
32   uint public defaultUInt; // 0
```

Tutorial - 4

LEARNETH

< Tutorials list

Syllabus

<

4. Variables

4 / 19

>

Global variables, also called *special variables*, exist in the global namespace. They don't need to be declared but can be accessed from within your contract. Global Variables are used to retrieve information about the blockchain, particular addresses, contracts, and transactions.

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

A list of all Global Variables is available in the [Solidity documentation](#).

Watch video tutorials on [State Variables](#), [Local Variables](#), and [Global Variables](#).

★ Assignment

- Create a new public state variable called `blockNumber`.
- 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.

Compile

Home

introduction.sol

variables.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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
16        address sender = msg.sender; // address of the caller
17        blockNumber = block.number;
18    }
19 }
```

✖ Explain contract

Tutorial - 5

LEARNETH

< Tutorials list

Syllabus

<

5.1 Functions - Reading and Writing to a State Variable

5 / 19

>

To define a function, use the `function` keyword followed by a unique name.

If the function takes inputs like our `set` function (line 9), you must specify the parameter types and names. A common convention is to use an underscore as a prefix for the parameter name to distinguish them from state variables.

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

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

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

★ Assignment

- Create a public state variable called `b` that is of type `bool` and initialize it to `true`.
- Create a public function called `get_b` that returns the value of `b`.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

readAndWrite.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7     bool public b = true;
8
9
10    // You need to send a transaction to write to a state variable
11    function set(uint _num) public { 22536 gas
12        num = _num;
13    }
14
15    // You can read from a state variable without sending a transaction
16    function get() public view returns (uint) { 2475 gas
17        return num;
18    }
19
20    function get_b() public view returns (bool) { 2539 gas
21        return b;
22    }
23 }
```

✖ Explain contract

Tutorial - 6

LEARNETH

Tutorials list

Syllabus

5.2 Functions - View and Pure

6 / 19

4. Calling any function not marked pure.

5. Using inline assembly that contains certain opcodes."

From the [Solidity documentation](#).

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

In Solidity development, you need to optimise your code for saving computation cost (gas cost). Declaring functions `view` and `pure` can save gas cost and make the code more readable and easier to maintain. Pure functions don't have any side effects and will always return the same result if you pass the same arguments.

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

★ Assignment

Create a function called `addTox2` that takes the parameter `y` and updates the state variable `x` with the sum of the parameter and the state variable `x`.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

viewAndPure.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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 {
18        x = x + y;
19    }
20 }
```

Tutorial - 7

LEARNETH

Tutorials list

Syllabus

5.3 Functions - Modifiers and Constructors

7 / 19

Constructor

A constructor function is executed upon the creation of a contract. You can use it to run contract initialization code. The constructor can have parameters and is especially useful when you don't know certain initialization values before the deployment of the contract.

You declare a constructor using the `constructor` keyword. The constructor in this contract (line 11) sets the initial value of the owner variable upon the creation of the contract.

[Watch a video tutorial on Function Modifiers.](#)

★ Assignment

1. Create a new function, `increaseX` in the contract. The function should take an input parameter of type `uint` and increase the value of the variable `x` by the value of the input parameter.

2. Make sure that `x` can only be increased.

3. The body of the function `increaseX` should be empty.

Tip: Use modifiers.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

visibility.sol

modifiersAndConstructors.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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
22         // execute the rest of the code.
23         _;
24     }
25
26     // Modifiers can take inputs. This modifier checks that the
27     // address passed in is not the zero address.
28     modifier validAddress(address _addr) {
29         require(_addr != address(0), "Not valid address");
30         _;
31     }
32 }
```

Tutorial - 8

LEARNETH

Tutorials list

Syllabus

5.4 Functions - Inputs and Outputs

8 / 19

Input and Output restrictions

There are a few restrictions and best practices for the input and output parameters of contract functions.

"[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible." From the Solidity documentation.

Arrays can be used as parameters, as shown in the function `arrayInput` (line 71). Arrays can also be used as return parameters as shown in the function `arrayOutput` (line 76).

You have to be cautious with arrays of arbitrary size because of their gas consumption. While a function using very large arrays as inputs might fail when the gas costs are too high, a function using a smaller array might still be able to execute.

[Watch a video tutorial on Function Outputs.](#)

★ Assignment

Create a new function called `returnTwo` that returns the values `2` and `true`, without using a return statement.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

visibility.sol

inputsAndOutputs.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20AS1
4 contract Function {
5     // Functions can return multiple values.
6     function returnMany() infinite gas
7     {
8         public
9         pure
10        returns (
11            uint,
12            bool,
13            uint
14        )
15    {
16        return (1, true, 2);
17    }
18
19    // Return values can be named.
20    function named() infinite gas
21    {
22        public
23        pure
24        returns (
25            uint x,
26            bool b,
27            uint y
28        )
29    {
30        return (1, true, 2);
31    }
32
33    // Return values can be assigned to their name.
34    // In this case the return statement can be omitted.
```

Tutorial - 9

LEARNETH

Tutorials list

Syllabus

6. Visibility

9 / 19

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`.

[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.

Compile

Home

introduction.sol

visibility.sol

inputsAndOutputs.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20AS1
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) { infinite gas
9         return "private function called";
10    }
11
12    function testPrivateFunc() public pure returns (string memory) { infinite gas
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) { infinite gas
20        return "internal function called";
21    }
22
23    function testInternalFunc() public pure virtual returns (string memory) { infinite gas
24        return internalFunc();
25    }
26
27    // Public functions can be called
28    // - inside this contract
29    // - inside contracts that inherit this contract
30    // - by other contracts and accounts
31    function publicFunc() public pure returns (string memory) { infinite gas
32        return "public function called";
```


Tutorial - 10

LEARNETH

Tutorials list

Syllabus

7.1 Control Flow - If/Else

10 / 19

in this contract, the `foo` function uses the `if-else` statement (line 10) to return `1` 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.
- 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.

Compile

Home

introduction.sol

IfElse.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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 y) public pure returns (bool) {
26        return y%2 == 0 ? true : false;
27    }
28 }
```

Tutorial - 11

LEARNETH

Tutorials list

Syllabus

7.2 Control Flow - Loops

11 / 19

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

- 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

Next

Well done! No errors.

Compile

Home

introduction.sol

loops.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract Loop {
5     uint public count;
6     function loop() public {
7         // for loop
8         for (uint i = 0; i < 10; i++) {
9             if (i == 5) {
10                // Skip to next iteration with continue
11                continue;
12            }
13            if (i == 5) {
14                // Exit loop with break
15                break;
16            }
17            count++;
18        }
19
20        // while loop
21        uint j;
22        while (j < 10) {
23            j++;
24        }
25    }
26 }
27
```

Tutorial - 12

LEARNETH

Tutorials list

Syllabus

8.1 Data Structures - Arrays

12 / 19

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`.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

arrays.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract Array {
5     // Several ways to initialize an array
6     uint[] public arr;
7     uint[] public arr2 = [1, 2, 3];
8     // Fixed sized array, all elements initialize to 0
9     uint[10] public myFixedSizeArr;
10    uint[3] public arr3 = [0, 1, 2];
11
12    function get(uint i) public view returns (uint) {
13        return arr[i];
14    }
15
16    // Solidity can return the entire array.
17    // But this function should be avoided for
18    // arrays that can grow indefinitely in length.
19    function getArr() public view returns (uint[3] memory) {
20        return arr3;
21    }
22
23    function push(uint i) public {
24        // Append to array
25        // This will increase the array length by 1.
26        arr.push(i);
27    }
28
29    function pop() public {
30        // Remove last element from array
31        // This will decrease the array length by 1
32        arr.pop();
33    }
```

Tutorial - 13

LEARNETH

Tutorials list

Syllabus

8.2 Data Structures - Mappings

13 / 19

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 we have seen in the arrays section.

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

★ Assignment

1. Create a public mapping `balances` that associates the key type `address` with the value type `uint`.
2. Change the functions `get` and `remove` to work with the mapping `balances`.
3. Change the function `set` to create a new entry to the `balances` mapping, where the key is the address of the parameter and the value is the balance associated with the address of the parameter.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

mappings.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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     // Nested mapping (mapping from address to another mapping)
27     mapping(address => mapping(uint => bool)) public nested;
28
29     function get(address _addr1, uint _i) public view returns (bool) {
30        // You can get values from a nested mapping
31        // even when it is not initialized
32        return nested[_addr1][_i];
33    }
```

Tutorial - 14

LEARNETH

Tutorials list

Syllabus

8.3 Data Structures - Structs

14 / 19

members as parameters in parentheses (line 16).

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

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

Accessing structs

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

Updating structs

To update a structs' member we also use the dot operator and assign it a new value (lines 39 and 45).

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

★ Assignment

Create a function `remove` that takes a `uint` as a parameter and deletes a struct member with the given index in the `todos` mapping.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

structs.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D28A51
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
26        todos.push(todo);
27    }
28
29    // Solidity automatically created a getter for 'todos' so
30    // you don't actually need this function.
31    function get(uint _index) public view returns (string memory text, bool completed) {
32        Todo storage todo = todos[_index];
```

Tutorial - 15

LEARNETH

Tutorials list

Syllabus

8.4 Data Structures - Enums

15 / 19

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.

[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.

Compile

Home

introduction.sol

enums.sol

```
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) {
26     return status;
27 }
28
29 function getSize() public view returns (Size) {
30     return sizes;
31 }
32
33 // Update status by passing uint into input
34 function set(Status _status) public {
35     status = _status;
36 }
37
38 // You can update to a specific enum like this
39 function cancel() public {
40     status = Status.Canceled;
41 }
42
43 // delete resets the enum to its first value, 0
44 function reset() public {
45     delete status;
46 }
47 }
```

Tutorial - 16

LEARNETH

Tutorials list

Syllabus

9. Data Locations

16 / 19

function f (line 12) and assign it the value of `myStruct`. changes in `myMemStruct3` would not affect the values stored in the mapping `myStructs` (line 10).

As we said in the beginning, when creating contracts we have to be mindful of gas costs. Therefore, we need to use data locations that require the lowest amount of gas possible.

★ Assignment

- Change the value of the `myStruct` member `foo`, inside the `function f`, to 4.
- 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.
- 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.
- 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.

Compile

Home

introduction.sol

dataLocations.sol 2

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract DataLocations {
5     uint[] public arr;
6     mapping(uint => address) map;
7     struct MyStruct {
8         uint foo;
9     }
10    mapping(uint => MyStruct) public 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        // get a struct from a mapping
16        MyStruct storage myStruct = myStructs[1];
17        myStruct.foo = 4;
18        // create a struct in memory
19        MyStruct memory myMemStruct = MyStruct(0);
20        MyStruct memory myMemStruct2 = myMemStruct;
21        myMemStruct2.foo = 1;
22
23        MyStruct memory myMemStruct3 = myStruct;
24        myMemStruct3.foo = 3;
25        return (myStruct, myMemStruct2, myMemStruct3);
26    }
27
28    function _f(
29        uint[] storage _arr,
30        mapping(uint => address) storage _map,
31        MyStruct storage _myStruct
32    ) internal {
```

Tutorial - 17

LEARNETH

Tutorials list

Syllabus

10.1 Transactions - Ether and Wei

17 / 19

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 (10⁹) `wei`.

`ether`

One `ether` is equal to 1,000,000,000,000,000 (10¹⁸) `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 10⁹.

Tip: Look at how this is written for `gwei` and `ether` in the contract.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

etherUnits.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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 - 18

LEARNETH

Tutorials list

Syllabus

10.2 Transactions - Gas and Gas Price

18 / 19

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

Create a new `public` state variable in the `Gas` contract called `cost` of the type `uint`. Store the value of the gas cost for deploying the contract in the new variable, including the cost for the value you are storing.

Tip: You can check in the Remix terminal the details of a transaction, including the gas cost. You can also use the Remix plugin *Gas Profiler* to check for the gas cost of transactions.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

gasAndGasPrice.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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 trans
9     // State changes are undone.
10    // Gas spent are not refunded.
11    function forever() public { // Infinite gas
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 - 19

LEARNETH

Tutorials list

Syllabus

10.3 Transactions - Sending Ether

19 / 19

If you change the parameter type for the functions `sendViaTransfer` and `sendViaSend` (line 33 and 38) from `payable address` to `address`, you won't be able to use `transfer()` (line 35) or `send()` (line 41).

Watch a video tutorial on Sending Ether.

★ Assignment

Build a charity contract that receives Ether that can be withdrawn by a beneficiary.

1. Create a contract called `Charity`.
2. Add a public state variable called `owner` of the type `address`.
3. Create a donate function that is public and payable without any parameters or function code.
4. Create a withdraw function that is public and sends the total balance of the contract to the `owner` address.

Tip: Test your contract by deploying it from one account and then sending Ether to it from another account. Then execute the withdraw function.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Home

introduction.sol

sendingEther.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract ReceiveEther {
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;
29     }
30 }
31
32 contract SendEther {
```

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.