

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

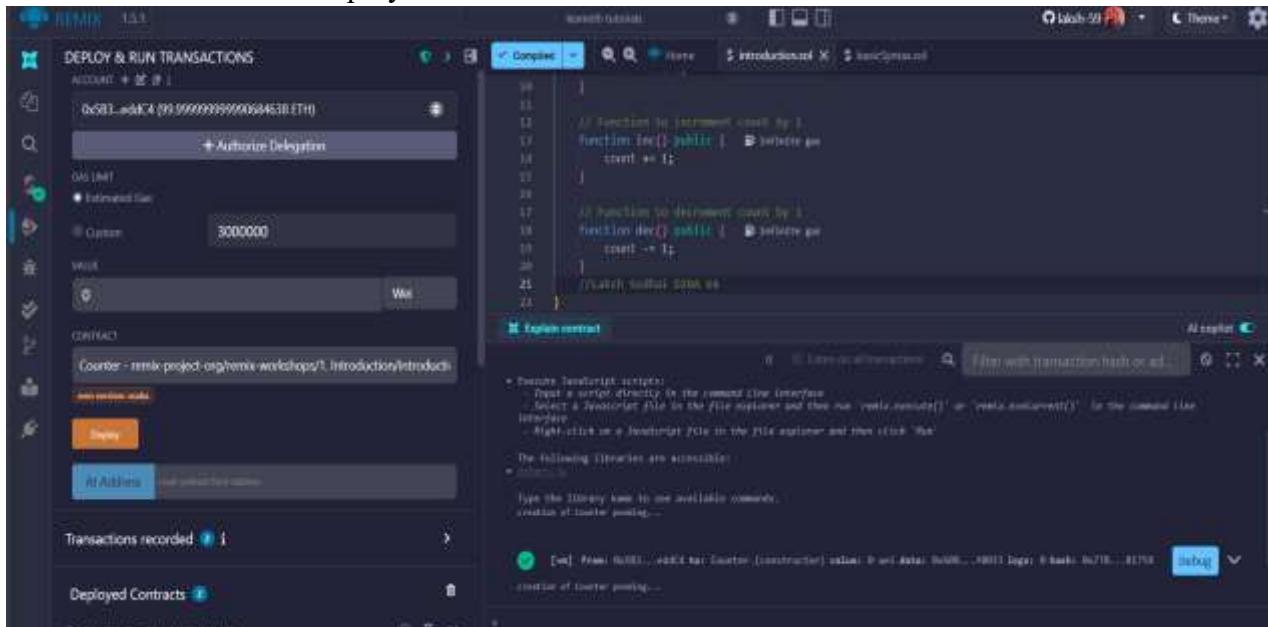
The screenshot shows the Remix IDE interface with the following details:

- Title Bar:** REMIX 1.5.1
- File Tabs:** Compiled (selected), Home, 5 introduction.sol, 5 basicSyntax.sol
- Code Editor:** Shows a Solidity contract with two functions: inc() and dec().

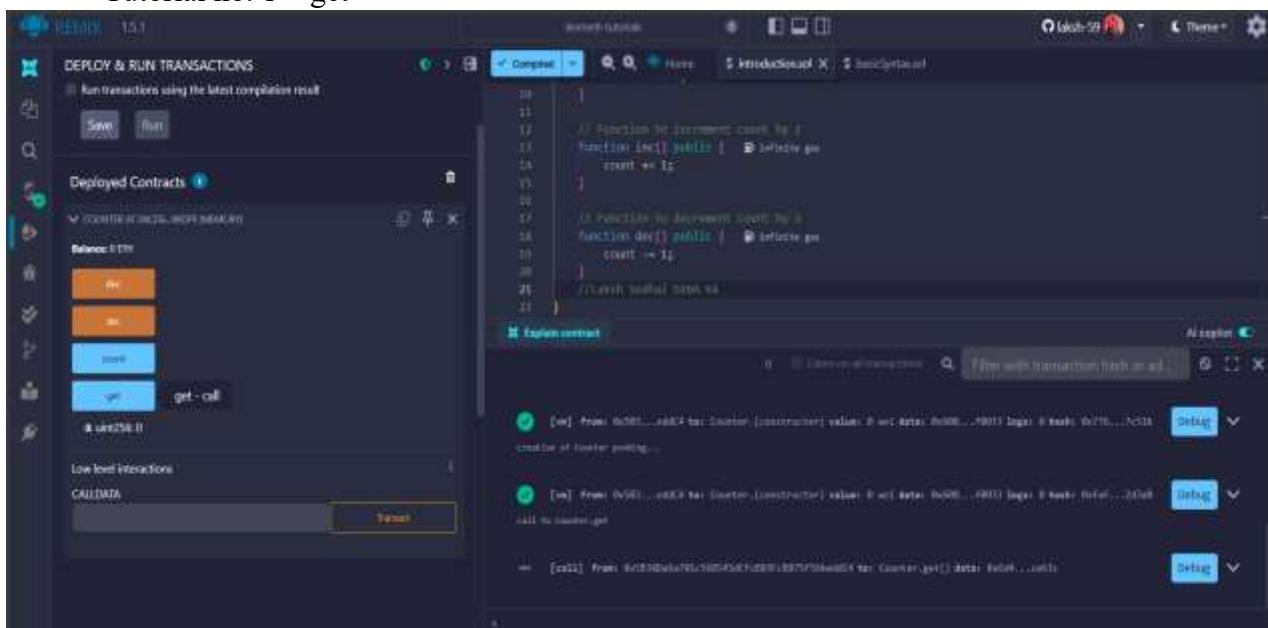
```
// Function to increment count by 1
function inc() public {
    count += 1;
}

// Function to decrement count by 1
function dec() public {
    count -= 1;
}
//Laksh Sodhai D20A 64.
```
- Toolbars and Buttons:** Includes icons for Run, Deploy, and other development tools.
- Terminal:** Welcome to Remix 1.5.1. Your files are stored in indexeIDR, 1.22 MB / 2 GB used.
- Help:** 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.execute()` in the command line interface
 - Right-click on a JavaScript file in the file explorer and then click 'Run'
- Libraries:** The following libraries are accessible:
 - ether.js
- Commands:** Type the library name to see available commands.

- Tutorial no. 1 – Deploy the contract



- Tutorial no. 1 – get



Tutorial no. 1 – Increment

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

- Deploy & Run Transactions:** Deployed Contracts: inc -> inc - transaction (not payable), inc, inc, inc.
- Code Editor:** Shows the Solidity code for the increment tutorial:

```
1 // SPDX-License-Identifier: MIT
2
3 // Function to increment count by 1
4 function inc() public {
5     count += 1;
6 }
7
8 // Function to decrement count by 1
9 function dec() public {
10    count -= 1;
11 }
12
13 // Function to retrieve current value
14 function getCount() public view returns (uint) {
15     return count;
16 }
```
- Deployed Contracts:** inc -> inc - transaction (not payable), inc, inc, inc.
- Low-level Interactions:** CALLDATA, inc, inc, inc.
- Transactions:** inc -> inc - transaction (not payable), inc, inc, inc.

● Tutorial no. 1 – Decrement

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

- Deploy & Run Transactions:** Deployed Contracts: dec -> dec - transaction (not payable), dec, dec, dec.
- Code Editor:** Shows the Solidity code for the decrement tutorial:

```
1 // SPDX-License-Identifier: MIT
2
3 // Function to increment count by 1
4 function inc() public {
5     count += 1;
6 }
7
8 // Function to decrement count by 1
9 function dec() public {
10    count -= 1;
11 }
12
13 // Function to retrieve current value
14 function getCount() public view returns (uint) {
15     return count;
16 }
```
- Deployed Contracts:** dec -> dec - transaction (not payable), dec, dec, dec.
- Low-level Interactions:** CALLDATA, inc, inc, inc.
- Transactions:** inc -> inc - transaction (not payable), inc, inc, inc.

- Tutorial no.

2

The screenshot shows the Truffle IDE interface. On the left, there's a sidebar titled 'LEARNETH' with a 'Tutorials list' section. The current tutorial is '2. Basic Syntax' at step 2/10. The main area displays a Solidity code editor with the following content:

```

pragma solidity ^0.8.6;
contract MyContract {
    string public name;
    string private alias;
    string internal address;
}

```

Below the code editor, there's a note about visibility:

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, state types, or state variables. We will look into them in the following sections. To help you understand the code, we will link to all following sections to video tutorials from the course of the Solidity by Example contracts.

On the right side of the interface, there are three tabs: 'Deploy contract', 'Transactions', and 'Logs'. The 'Logs' tab shows three log entries:

- [txid] From: 0x00 to: 0x00 gas: 2000000 value: 0 wei status: 0x00
- [txid] From: 0x00 to: 0x00 gas: 2000000 value: 0 wei status: 0x00
- [txid] From: 0x00 to: 0x00 gas: 2000000 value: 0 wei status: 0x00

At the bottom of the interface, there are buttons for 'Check Answer', 'Previous', 'Show answer', and 'Next'.

- Tutorial no. 3

The screenshot shows the Truffle IDE interface. On the left, there's a sidebar titled 'LEARNETH' with a 'Tutorials list' section. The current tutorial is '3. Primitive Data Types' at step 3/7. The main area displays a Solidity code editor with the following content:

```

address public addr = 0x23E0000000000000000000000000000000000000;
address public regular = 0x0000000000000000000000000000000000000000;
int public reg = 10;
uint public result = 0;

```

Below the code editor, there's a note about mappings, arrays, enums, and structs:

Learn 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:

- Create a new variable `regular` that is a `public` `address` and give it a value that is not the same as the available variable `addr`.
- Create a `public` variable called `reg` that is a negative number; decide upon the type.
- Create a new variable `result` that has the smallest `uint` size type and the smallest `uint` value and is `public`.

For look at the other address in the contract or search the internet for an Ethereum address.

At the bottom of the interface, there are buttons for 'Check Answer', 'Previous', 'Show answer', and 'Next'.

- Tutorial no.

4

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Variables {
    uint public blocknumber;
    function getBlockNumber() public view returns (uint) {
        blocknumber = block.number;
    }
}

```

Assignment

- Create a new public state variable called `blocknumber`.
- Inside the function `getBlockNumber()`, 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.

- Tutorial no. 5

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint x;
    function getX() public view returns (uint) {
        return x;
    }
}

```

Assignment

- Create a public state variable called `x` that is of type `uint` and initialize it to `100`.
- Create a public function called `getX()` that returns the value of `x`.

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

Well done! No errors.

- Tutorial no.

6

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Adder {
    function addTwo(uint x, uint y) public view returns (uint256) {
        return x + y;
    }

    function addThree(uint x, uint y, uint z) public view returns (uint256) {
        return x + y + z;
    }
}

```

Assignment

Create a function called `increment` 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.

- Tutorial no. 7

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract increment {
    uint x = 17;

    constructor() {
        x = 17;
    }

    function increment(uint x) public {
        x += x;
    }
}

```

Assignment

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

By the modifiers

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

Well done! No errors.

- Tutorial no.

8

```

contract Function {
    function determine() public pure returns (int a, int b) {
        a = 2;
        b = 3;
    }
}

// Watch local, kovan or mainnet

```

Assignment

Create a new function called `internal` that returns the values `a` and `b` without using a return statement.

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

Well done! No errors.

- Tutorial no. 9

```

// Internal function can't be called outside contracts
function testInternal() public view returns (string memory) {
    return internalFunc();
}

function testInternal() public view returns (string memory) {
    return internalFunc();
}

// LEND

```

Assignment

Create a new function in the `Borrow` contract called `internalFunction` that returns the values of all state variables from the `Lend` contract that are possible to return.

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

Well done! No errors.

- Tutorial no.

10

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract T1 {
    function checkEven(uint y) public pure returns (bool) {
        return (y % 2 == 0);
    }
}

// Solidity way to write an if-else statement
if (x > 0) {
    // ...
} else if (x < 0) {
    // ...
} else {
    // ...
}

```

Assignment

Create a new function called `isEven` in the `base` 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 `checkEven` function.

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

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

Well done! No errors.

- Tutorial no. 11

```

for (uint i = 0; i < 10; i++) {
    count += 1;
}

// while loop
while (true) {
    count += 1;
}

```

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.

Assignment

1. Create a public `uint` state variable called `count` in the `base` contract.
2. At the end of the for loop, increment the `count` variable by 1.
3. Try to get the `count` variable to be equal to 9, but make sure you don't edit the `base` statement.

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

Well done! No errors.

- Tutorial no.

12

```

#include <avr/interrupt.h>
#include "array.h"

void array(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        arr[i] = i;
    }
}

int main() {
    int arr[5];
    array(arr, 5);
    for (int i = 0; i < 5; i++) {
        if (arr[i] != i) {
            return 1;
        }
    }
    return 0;
}

```

Assignment

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

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

Well done! No errors.

- Tutorial no. 13

```

#include <avr/interrupt.h>
#include "mapping.h"

void addBalance(address_t key, balance_t value) {
    nestedAddress_t temp;
    temp.key = key;
    temp.value = value;
    nestedAddress_t* nestedAddress = &temp;
    map[&temp] = nestedAddress;
}

balance_t remove(address_t address, uint16_t* result) {
    balance_t temp;
    temp = map[address];
    delete map[address];
    *result = temp.value;
    return temp.value;
}

void checkBalance(address_t address) {
    balance_t result;
    remove(address, &result);
    if (result == 0) {
        Serial.println("Balance removed");
    } else {
        Serial.print("Balance is ");
        Serial.println(result);
    }
}

```

Assignment

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

- Tutorial no.

14

6.3 Data Structures - Struct

Key-value mapping. We provide the name of the struct and the keys and values in 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 20).

Accessing structs

To access a member of a struct we can use the dot operator (line 23).

Updating structs

To update a struct's member we also use the dot operator and assign it a new value (lines 37 and 45).

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

Assignment

Create a function `update` that takes a `task` as a parameter and updates a struct member with the given index in the `tasks` mapping.

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

Well done! No errors.

- Tutorial no. 15

6.4 Data Structures - Enum

We can update the enum value of a variable by updating its `base`, representing the `status_initializer` (line 30). Shown 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 39).

Removing an enum value

We can use the `delete` operator to delete the enum value of the variable, which moves its arrays and mappings to set the default value to 0.

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

Assignment

- Define an enum type called `base` with the members `0`, `1`, and `2`.
- Initialize the variable `status` of the enum type `base`.
- Create a getter function `getBase()` that returns the value of the variable `base`.

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

Well done! No errors.

- Tutorial no.

16

```

mapping(address => uint256) public balances;
function add(uint256 amount) public {
    balances[msg.sender] += amount;
}

```

Assignment

- Change the value of the `balances` member `msg`, inside the `balances`, to 4.
- Create a new struct `balances`, with the data location memory inside the `balances`, and assign it the value of `balances`. Change the value of the `balances` member `msg` to 1.
- Create a new struct `balances`, with the data location memory inside the `balances`, and assign it the value of `balances`. Change the value of the `balances` member `msg` to 3.
- Let the function return `balances`, `balances`, and `balances`.
- Make sure to create the correct return types for the function `g`.

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

Well done! No errors.

- Tutorial no. 17

```

mapping(address => uint256) public balances;

```

Assignment

- Create a `public` `uint` called `omega` and set it to 1 `wei`.
- Create a `public` `uint` called `omega`, and set it to the result of a comparison operation between 1 `wei` and `10^18`.
- Tip: look at how this is written for `wei` and `ether` in the contract.

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

Well done! No errors.

- Tutorial no.

18

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

- Left Panel:** Shows the project structure under "LEARNETH" with a file named "Gas limit".
- Middle Panel:** Displays the Solidity code for the "Gas limit" contract. The code includes a function `pay` that loops until it reaches the gas limit, demonstrating how gas consumption is tracked and can run out.
- Right Panel:** Shows the transaction history with three entries, each detailing a transaction that failed due to gas limits.
- Bottom Bar:** Includes buttons for "Check Answer", "Show answer", and "Next". A message at the bottom says "Well done! No errors."

- Tutorial no. 19

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

- Left Panel:** Shows the project structure under "LEARNETH" with a file named "Charity Ether".
- Middle Panel:** Displays the Solidity code for the "Charity Ether" contract. It includes functions for depositing ether and withdrawing ether.
- Right Panel:** Shows the transaction history with three entries, each detailing a transaction that failed due to ether balance issues.
- Bottom Bar:** Includes buttons for "Check Answer", "Show answer", and "Next". A message at the bottom says "Well done! No errors."

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.