

## Blockchain Experiment 4

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

### THEORY:

#### **Q1: 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
- **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.

#### **Q2: 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.

#### **Q3: 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.

## **Q4: 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.

## **Q5: 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.  
Example: 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 }.

## **Q6: 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.

## **Q7: 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.

## **TASKS PERFORMED:**

### **Tutorial 1: Introduction**

#### a. get

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays 'DEPLOY & RUN TRANSACTIONS' with a 'Deploy' button and an 'At Address' dropdown set to 'Load contract from Address'. Below this is a 'Transactions recorded' section with a balance of '0 ETH'. On the right, the main area shows the Solidity code for the Counter contract:

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

contract Counter {
    uint public count;

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

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

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

Below the code, there's an 'Explain contract' section with a 'CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c' entry. At the bottom, there are 'Scan Alert' and 'Initialize as git repo' buttons.

#### b. inc

This screenshot is nearly identical to the previous one, showing the REMIX IDE interface. The 'Transactions recorded' section now shows a balance of '0 ETH' and a recent transaction record for an 'inc' call. The Solidity code for the Counter contract is identical to the previous screenshot.

### c. dec

```

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

contract Counter {
    uint public count;

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

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

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

    function count() public view returns (uint) {
        return count;
    }
}

```

## Tutorial 2: Basic Syntax

```

// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
pragma solidity ^0.8.3;

contract MyContract {
    string public name = "Alice";
}

```

**Assignment**

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

**Check Answer**   **Show answer**

**Well done! No errors.**

## Tutorial 3: Primitive Data Types

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons and a search bar. The main area has tabs for 'Compiled' and 'primitiveDataTypes.sol'. The code editor contains the following Solidity code:

```
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 defaultBool; // false
32 uint public defaultUint; // 0
33 int public defaultInt; // 0
34 address public defaultAddr; // 0x000000000000000000000000000000000000000000000000000000000000000
35
36 // New values
37 address public newAddr = 0x000000000000000000000000000000000000000000000000000000000000000;
38 int public neg = -12;
39 uint8 public newU = 0;
40 }
```

Below the code editor, there's an 'Explain contract' section with a transaction log:

```
CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', 'Next', and 'Well done! No errors.'

## Tutorial 4: Variables

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons and a search bar. The main area has tabs for 'Compile' and 'variables.sol'. The code editor contains the following Solidity code:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Variables {
5     // State variables are stored on the blockchain.
6     string public text = "Hello";
7     uint public num = 123;
8     uint public blockNumber;
9
10    function doSomething() public {
11        // Local variables are not saved to the blockchain.
12        uint i = 456;
13
14        // Here are some global variables
15        uint timestamp = block.timestamp; // Current block timestamp
16        address sender = msg.sender; // Address of the caller
17        blockNumber = block.number;
18    }
19 }
```

Below the code editor, there's an 'Explain contract' section with a transaction log:

```
CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', 'Next', and 'Well done! No errors.'

## Tutorial 5: Functions - Reading and Writing to a State Variable

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons and a 'Tutorials list' section. The main area has tabs for 'Compile' and 'View'. The code editor contains the following Solidity code:

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

contract SimpleStorage {
    // State variable to store a number
    uint public num;
    bool public b = true;

    // You need to send a transaction to write to a state variable.
    function set(uint _num) public {
        num = _num;
    }

    // You can read from a state variable without sending a transaction.
    function get() public view returns (uint) {
        return num;
    }

    function get_b() public view returns (bool) {
        return b;
    }
}
```

Below the code, there's an 'Explain contract' section with a transaction log:

```
call [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xd4...ce63c
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A green bar at the very bottom says 'Well done! No errors.'

## Tutorial 6: Functions - View and Pure

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons and a 'Tutorials list' section. The main area has tabs for 'Compiled' and 'View'. The code editor contains the following Solidity code:

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

contract ViewAndPure {
    uint public x = 1;

    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }

    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }

    function addToX2(uint y) public {
        x = x + y;
    }
}
```

Below the code, there's an 'Explain contract' section with a transaction log:

```
call [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xd4...ce63c
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A green bar at the very bottom says 'Well done! No errors.'

## Tutorial 7: Functions - Modifiers and Constructors

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

- Header:** remix.ethereum.org/?#activate=udapp.solidity,LearnEth&lang=en&optimize&runs=200&evmVersion&version=soljson-v0.8.31+commit.fd3a2265.js
- Sidebar:** LEARNETH, Tutorials list, Syllabus, 5.3 Functions - Modifiers and Constructors (7 / 19).
- Code Editor:** Compiled code for `modifiersAndConstructors.sol`. The code includes a constructor, a modifier `noReentrancy()`, and two functions: `increaseX` and `decrement`.
- Assignment:** Create a new function `increaseX` that takes an input parameter of type `uint` and increases the value of the variable `x` by the value of the input parameter.
- Tip:** Use modifiers.
- Buttons:** Check Answer, Show answer, Next, Well done! No errors.
- Logs:** Explain contract, 0 transactions, Listen on all transactions, Filter.
- Bottom Bar:** Scan Alert, Initialize as git repo, Did you know? You can use the Recorder to record and replay your transactions to any network from the Deploy and Run plugin.

## Tutorial 8: Functions - Inputs and Outputs

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

- Header:** remix.ethereum.org/?#activate=udapp.solidity,LearnEth&lang=en&optimize&runs=200&evmVersion&version=soljson-v0.8.31+commit.fd3a2265.js
- Sidebar:** LEARNETH, Tutorials list, Syllabus, 5.4 Functions - Inputs and Outputs (8 / 19).
- Code Editor:** Compiled code for `inputsAndOutputs.sol`. The code includes functions for arrays and a `returnTwo` function.
- Assignment:** Create a new function called `returnTwo` that returns the values `-2` and `true` without using a return statement.
- Tip:** 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.
- Buttons:** Check Answer, Show answer, Next, Well done! No errors.
- Logs:** Explain contract, 0 transactions, Listen on all transactions, Filter.
- Bottom Bar:** Scan Alert, Initialize as git repo, Did you know? You can use the Recorder to record and replay your transactions to any network from the Deploy and Run plugin.

## Tutorial 9: Visibility

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

- Left Sidebar:** Titled "LEARNETH", it shows a "Tutorials list" with "6. Visibility" selected (9 / 19). It contains text about state variables and contracts, and a section titled "Assignment" with instructions to create a new function in the `child` contract.
- Code Editor:** Titled "Compiled", it displays the `visibility.sol` file with the following code:

```
43 // function testExternalFunc() public pure returns (string memory) {
44 //     return externalFunc();
45 // }
46
47 // State variables
48 string private privateVar = "my private variable";
49 string internal internalVar = "my internal variable";
50 string public publicVar = "my public variable";
51 // State variables cannot be external so this code won't compile.
52 // string external externalVar = "my external variable";
53 }
54
contract child is Base {
55     // Inherited contracts do not have access to private functions
56     // and state variables.
57     // function testPrivateFunc() public pure returns (string memory) {
58     //     return privateFunc();
59     // }
60
61     // Internal function call be called inside child contracts.
62     function testInternalFunc() public pure override returns (string memory) {
63         return internalFunc();
64     }
65
66     function testInternalVar() public view returns (string memory, string memory) {
67         return (internalVar, publicVar);
68     }
69 }
70 }
```
- Bottom Panel:** Shows a transaction log: "CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c". A note says "Did you know? You can use the Recorder to record and replay your transactions to any network from the Deploy and Run plugin."

## Tutorial 10: Control Flow - If/Else

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

- Left Sidebar:** Titled "LEARNETH", it shows a "Tutorials list" with "7.1 Control Flow - If/Else" selected (10 / 19). It contains text about the `else if` statement and a section titled "Assignment" with instructions to create a new function called `evenCheck`.
- Code Editor:** Titled "Compiled", it displays the `ifElse.sol` file with the following code:

```
1 // SPDX-LICENSE-IDENTIFIER: MIT
2 pragma solidity ^0.8.3;
3
4 contract IfElse {
5     function foo(uint x) public pure returns (uint) {    infinite gas
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) {    infinite gas
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) {    infinite gas
26        return y%2 == 0 ? true : false;
27    }
28 }
```
- Bottom Panel:** Shows a transaction log: "CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c". A note says "Did you know? You can use the Recorder to record and replay your transactions to any network from the Deploy and Run plugin."

## Tutorial 11: Control Flow - Loops

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, including 'Tutorials list' and 'Syllabus'. The main workspace shows the 'Compiled' tab for the file 'loops.sol'. The code implements two loops: a for loop that iterates from 0 to 9, skipping the value 5 with a `continue` statement; and a while loop that iterates from 0 to 9, exiting with a `break` statement when the value reaches 5. Below the code, the 'Explain contract' section shows a transaction log:

```
CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', and 'Next', along with a message 'Well done! No errors.'

## Tutorial 12: Data Structures - Arrays

The screenshot shows the REMIX IDE interface. The sidebar shows the 'LEARNETH' tutorial navigation. The main workspace shows the 'Compiled' tab for the file 'arrays.sol'. The code defines a `CompactArray` contract with an array of uints. It demonstrates how to remove elements from the array using the `delete` operator. The `remove` function shifts the last element into the position of the deleted element. The `test` function shows the array being populated with values 1, 2, 3, 4, then having its second and third elements removed, resulting in the array [1, 4]. Below the code, the 'Explain contract' section shows a transaction log:

```
CALL [call] from: 0x58380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', and 'Next', along with a message 'Well done! No errors.'

## Tutorial 13: Data Structures - Mappings

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '8.2 Data Structures - Mappings'. The main area shows the Solidity code for 'mappings.sol'. The code defines a mapping named 'balances' where the key is an address and the value is a uint. It includes functions for setting and removing values from this mapping. Below the code editor, there is a 'Check Answer' button and a 'Show answer' button.

```
11 // SPDX-License-Identifier: MIT
12 pragma solidity ^0.8.0;
13
14 contract LearnETH {
15     mapping(address => uint) balances;
16
17     function set(address _addr) public {
18         // Update the value at this address
19         balances[_addr] = _addr.balance;
20     }
21
22     function remove(address _addr) public {
23         // Reset the value to the default value.
24         delete balances[_addr];
25     }
26
27     function get(address _addr) public view returns (uint) {
28         return balances[_addr];
29     }
30
31     function nestedGet(address _addr1, uint _i) public view returns (bool) {
32         // You can get values from a nested mapping
33         // even when it is not initialized
34         return nested[_addr1][_i];
35     }
36
37     function setNested(address _addr1, uint _i, bool _boo) public {
38         nested[_addr1][_i] = _boo;
39     }
40
41     function removeNested(address _addr1, uint _i) public {
42         delete nested[_addr1][_i];
43     }
44
45 }
46 }
```

## Tutorial 14: Data Structures - Structs

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '8.3 Data Structures - Structs'. The main area shows the Solidity code for 'structs.sol'. The code defines a struct named 'Todo' with fields 'text' and 'completed'. It then creates a mapping named 'todos' that maps addresses to Todo structs. Functions are provided for adding, updating, and removing todos from this mapping. Below the code editor, there is a 'Check Answer' button and a 'Show answer' button.

```
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51 
```

## Tutorial 15: Data Structures - Enums

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '8.4 Data Structures - Enums' (15 / 19). The main area shows the Solidity code for 'enums.sol'. The code defines an enum 'Size' with members S, M, and L. It includes functions to get the current status and size, and to set a new status. A note explains that the default value is the first element in the enum definition. The code also demonstrates how to update a specific enum value using 'this'. Below the code, there is an 'Assignment' section with four tasks:

- Define an enum type called `Size` with the members `S`, `M`, and `L`.
- Initialize the variable `sizes` of the enum type `Size`.
- Create a getter function `getSize()` that returns the value of the variable `sizes`.
- Change the status to `Canceled` using `this`.

At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a message 'Well done! No errors.'

## Tutorial 16: Data Locations

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays the 'LEARNETH' tutorial navigation, currently at '9. Data Locations' (16 / 19). The main area shows the Solidity code for 'dataLocations.sol'. The code demonstrates various memory locations and storage operations. It includes a function `f` that takes a mapping and returns a struct, and another function `f` that takes a mapping and returns a memory variable. It also shows how to create a struct in memory and return multiple memory variables. A tip at the bottom suggests creating the correct return types for the function `f`. Below the code, there is an 'Assignment' section with four tasks:

- 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`.

At the bottom, there are 'Check Answer' and 'Show answer' buttons, and a message 'Well done! No errors.'

## Tutorial 17: Transactions - Ether and Wei

The screenshot shows the Remix IDE interface with the title "Tutorial 17: Transactions - Ether and Wei". The left sidebar displays the "Tutorials list" with "10.1 Transactions - Ether and Wei" selected. The main area shows the Solidity code for "etherAndWei.sol":

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

contract EtherUnits {
    uint public oneWei = 1 wei;
    // 1 wei is equal to 1
    bool public isOneWei = 1 wei == 1;

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

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

The right sidebar contains an "Assignment" section with two tasks:

- Create a `public uint` called `oneGwei` and set it to 1 `gwei`.
- Create a `public bool` called `isOneWei` and set it to the result of a comparison operation between 1 `gwei` and  $10^9$ .

A tip notes: "Look at how this is written for `gwei` and `ether` in the contract."

Buttons at the bottom include "Check Answer", "Show answer", "Next", and "Well done! No errors." A "Scan Alert" and "Initialize as git repo" button are also present.

## Tutorial 18: Transactions - Gas and Gas Price

The screenshot shows the Remix IDE interface with the title "Tutorial 18: Transactions - Gas and Gas Price". The left sidebar displays the "Tutorials list" with "10.2 Transactions - Gas and Gas Price" selected. The main area shows the Solidity code for "gasAndGasPrice.sol":

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

contract Gas {
    uint public i = 0;
    uint public cost = 170367;

    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

The right sidebar contains an "Assignment" section with a task:

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.

A tip notes: "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."

Buttons at the bottom include "Check Answer", "Show answer", "Next", and "Well done! No errors." A "Scan Alert" and "Initialize as git repo" button are also present.

## Tutorial 19: Transactions - Sending Ether

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons and sections like 'Tutorials list', 'Syllabus', and 'Assignment'. The main area displays a Solidity contract named 'sendingEther.sol'. The code is as follows:

```
function sendViaTransfer(address payable _to) public payable {
    // This function is no longer recommended for sending Ether.
    _to.transfer(msg.value);
}

function sendViaSend(address payable _to) public payable {
    // Send returns a boolean value indicating success or failure.
    // This function is not recommended for sending Ether.
    bool sent = _to.send(msg.value);
    require(sent, "Failed to send Ether");
}

function sendViaCall(address payable _to) public payable {
    // Call returns a boolean value indicating success or failure.
    // This is the current recommended method to use.
    (bool sent, bytes memory data) = _to.call{value: msg.value}("");
    require(sent, "Failed to send Ether");
}

contract Charity {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function donate() public payable {}

    function withdraw() public {
        uint amount = address(this).balance;

        (bool sent, bytes memory data) = owner.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

Below the code, there are buttons for 'Check Answer' (blue), 'Show answer' (yellow), and 'Next' (green). A message 'Well done! No errors.' is displayed. At the bottom, there are links for 'Scan Alert' and 'Initialize as git repo', and a note about the Recorder.

## CONCLUSION:

Through this experiment, the basic concepts of Solidity programming were learned by completing practical assignments using the Remix IDE. Important topics such as data types, variables, different types of functions, visibility, modifiers, constructors, control flow statements, data structures, and transactions were studied and applied while creating smart contracts. The hands-on practice helped in understanding how to design, compile, and deploy contracts using the Remix VM. Overall, this experiment helped in building a clear understanding of blockchain concepts and provided a strong foundation for developing and managing smart contracts effectively.