

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 Tab:** introduction.sol
- Code Editor:** Contains the following Solidity code:

```
// 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;
}

// Laksh Sodhai D20A 64
```
- Compiler Status:** Compiled (green)
- Toolbars and Buttons:** Includes icons for Home, Search, and various development tools.
- Bottom Panel:** Shows the terminal output, which includes a welcome message, file storage information, and instructions for using the terminal.

- Tutorial no. 1 – Deploy the contract

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, showing the environment as 'Remix VM (Osaka)', account as '0x5B3...addC4 (100.0 ETH)', and gas limit set to 'Custom' (3000000). The 'Deploy' button is highlighted. The main area displays the Solidity code for the Counter contract:

```
// 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;
}

// Laksh Sodhai D20A 64

```

The 'Explain contract' panel on the right provides instructions for interacting with the deployed contract.

- Tutorial no. 1 – get

The screenshot shows the Remix IDE interface after deployment. The 'DEPLOY & RUN TRANSACTIONS' sidebar now shows the deployed contract at address '0x5B3...addC4'. The 'Transactions recorded' section shows one transaction. The 'Deployed Contracts' section lists the deployed Counter contract with its address and balance (0 ETH). The 'Interactions' section shows the available functions: inc, dec, count, get, and their respective gas costs. The 'Explain contract' panel shows the creation of a pending transaction to call the 'get' function.

Laksh Sodhai D20A 64
Tutorial no. 1 – Increment

The screenshot shows the REMIX IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar has a 'Deploy' button and a dropdown set to 'At Address'. Below it, 'Transactions recorded' shows one transaction with hash 0x091...39138. The 'Deployed Contracts' sidebar shows a deployed contract named 'COUNTER' at address 0x091...39138, with a balance of 0 ETH. It lists four functions: 'dec', 'inc', 'count', and 'get', each with a uint256 return type. The 'Explain contract' section details the creation of the contract and its pending transactions. One transaction is shown: [vm] From: 0x5B...e0dCA to: Counter.(constructor) value: 0 wei data: 0x000...10033. Logs: 0 hash: 0xc10...aa3ff. Call to Counter.gas.

```
14: count += 1;
15:
16:
17: // Function to decrement count by 1
18: function dec() public {
19:     count -= 1;
20: }
21: //Laksh Sodhai D20A 64
22:
```

creation of counter pending...

[vm] From: 0x5B...e0dCA to: Counter.(constructor) value: 0 wei data: 0x000...10033
Logs: 0 hash: 0xc10...aa3ff
call to Counter.gas

[call] From: 0x5B...e0dCA to: Counter.get() value: 0 wei data: 0x004...1003c
transaction to Counter.inc pending...

[vm] From: 0x5B...e0dCA to: Counter.inc() value: 0 wei data: 0x371...383c8 Logs: 0 hash: 0xd55...84620
transaction to Counter.dec pending...

Debug Debug Debug

- Tutorial no. 1 – Decrement

The screenshot shows the REMIX IDE interface, identical to the previous one but with different transaction history. The 'Transactions recorded' sidebar now shows two transactions: one from step 1 and another from step 2. The second transaction is a call to the 'inc' function. The 'Explain contract' section details the creation of the contract and its pending transactions. One transaction is shown: [vm] From: 0x5B...e0dCA to: Counter.(constructor) value: 0 wei data: 0x000...10033. Logs: 0 hash: 0xc10...aa3ff. Call to Counter.gas.

```
14: count += 1;
15:
16:
17: // Function to decrement count by 1
18: function dec() public {
19:     count -= 1;
20: }
21: //Laksh Sodhai D20A 64
22:
```

creation of counter pending...

[vm] From: 0x5B...e0dCA to: Counter.(constructor) value: 0 wei data: 0x000...10033
Logs: 0 hash: 0xc10...aa3ff
call to Counter.gas

[call] From: 0x5B...e0dCA to: Counter.get() value: 0 wei data: 0x004...1003c
transaction to Counter.inc pending...

[vm] From: 0x5B...e0dCA to: Counter.inc() value: 0 wei data: 0x371...383c8 Logs: 0 hash: 0xd55...84620
transaction to Counter.dec pending...

[vm] From: 0x5B...e0dCA to: Counter.dec() value: 0 wei data: 0x001...fae2 Logs: 0 hash: 0xf87...3f8a7
transaction to Counter.inc pending...

Debug Debug Debug

- Tutorial no.

2

you can access it. In this case, it's a `state` variable that you can access from inside and outside the contract.

Don't worry if you don'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 to 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**

- Tutorial no. 3

- Tutorial no.

4

```

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

contract Variables {
    uint public blockNumber;

    function something() public view returns (uint) {
        return blockNumber;
    }

    function setSomething() public {
        blockNumber = 123;
    }
}

```

Assignment

1. Create a new public state variable called `blockNumber`.
2. Inside the function `something()`, assign the value of the current block number to the state variable `blockNumber`.

Note: 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 Functions {
    uint public balance;

    function getBalance() public view returns (uint) {
        return balance;
    }
}

```

Assignment

1. Create a public state variable called `b` that is of type `bool` and initialize it to `false`.
2. Create a public function called `getB()` that returns the value of `b`.

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

Check Answer **Show answer**

Next

Well done! No errors.

- Tutorial no.

6

```

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

contract MyContract {
    function pureSum(uint a, uint b) public pure returns (uint) {
        return a + b;
    }

    function pureProduct(uint a, uint b) public pure returns (uint) {
        return a * b;
    }
}

```

- Tutorial no. 7

```

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

contract MyContract {
    bool locked = false;

    constructor() {
        locked = false;
    }

    modifier locked() {
        require(locked == false);
        _;
    }

    function increment() public locked {
        x += 1;
    }
}

function increment() public {
    x += 1;
}

```

- Tutorial no.

Tutorial No: 08

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

contract Function {
    function returning() public pure returns (int a, bool b) {
        a = -2;
        b = true;
    }
}
```

InputsAndOutputs

Variables

Read/Write

Viewed/Passed

ModifiersAndConstructors

InputsAndOutputs

contract Function {

function returning() public pure returns (int a, bool b) {

a = -2;

b = true;

}

}

Explain contract

Welcome to Truffle 1.10.1

Your files are stored in `InputSolidity`, 3.52 MB / 2.08 MB

You can use this terminal to:

- Check transaction details and start debugging:
- Create Javascript scripts:
 - Drop a script directly in the command line interface
 - Select a `.js` file in the file explorer and then run `truffle migrate --network development` in the command line interface
 - Right-click on a `.development.js` file in the file explorer and then click "Run"

The following libraries are accessible:

`ethers.js`

Type the library name to see available methods.

Check Answer **Show answer**

Next

Well done! No errors.

- Tutorial no.

9

```

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

contract VISIBLE {
    string public visibleVar = "Visible";
}

contract INVISIBLE {
    string private privateVar = "Invisible";

    function privateFunction() internal pure returns (string memory) {
        return privateVar;
    }

    function allVisible() external view returns (string[] memory) {
        string[] memory values = new string[](1);
        values[0] = privateFunction();
        return values;
    }
}

```

10

```

function foo(bool a) public pure returns (bool) {
    if (a) {
        // If a is true
        // ...
        // return a;
    } 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.
        // ...
        // return b;
    }
}

function evenCheck(uint y) public pure returns (bool) {
    return y % 2 == 0 ? true : false;
}

```

- Tutorial no.

- Tutorial no. 11

```

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

contract Continue {
    uint count = 0;

    for (uint i = 0; i < 10; i++) {
        if (i == 5) {
            continue;
        }
        if (i == 10) {
            break;
        }
        count++;
    }
}

// SPDX-License-Identifier: MIT
// File: Continue.sol
// Contract: Continue
// Description: A simple contract demonstrating the use of the continue and break statements in loops.
// Author: Laksh Sodhai
// Version: 1.0.0
// Last Updated: 2023-09-15
// 
```

12

```

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

contract Array {
    uint[] arr;

    function test() public {
        arr.push(1);
        arr.push(2);
        arr.push(3);
        arr.push(4);
        // arr[3] = 5; // Error: Assignment to constant array element.

        remove(1);
        // arr[1] = 5; // Error: Assignment to constant array element.

        remove(2);
        // arr[2] = 5; // Error: Assignment to constant array element.

        remove(3);
        // arr[3] = 5; // Error: Assignment to constant array element.
    }

    function remove(uint index) public {
        arr.pop();
    }
}

// SPDX-License-Identifier: MIT
// File: Array.sol
// Contract: Array
// Description: A simple contract demonstrating array manipulation and mapping.
// Author: Laksh Sodhai
// Version: 1.0.0
// Last Updated: 2023-09-15
// 
```

- Tutorial no.

- Tutorial no. 13

```

1 // You can get values from a nested mapping
2 // when using the dot operator
3 return nested.address[0];
4
5
6 function add1() {
7     address.add("0x00000000000000000000000000000000", "1");
8 }
9
10 function remove(address, value) {
11     delete nested.address[address];
12 }
13
14 // Check address[0] after add1();
15 // It will be 1
16
17 // Check address[0] after remove();
18 // It will be null
  
```

Assignment

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

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

Well done! No errors.

14

```

1 // update field
2 function update(uint _index, string memory _text) public {
3     storage.toku[_index] = _text;
4 }
5
6 // update completed
7 function updateCompleted(uint _index) public {
8     storage.toku[_index] = "Completed";
9 }
10
11 function remove(uint _index) public {
12     delete storage.toku[_index];
13 }
14
15 // Check storage.toku[0]
16 // It will be null
  
```

Assignment

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

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

Well done! No errors.

- Tutorial no.

- Tutorial no. 15

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

- Left Panel:** Shows the project structure with a file named "8.4 Data Structures - Enums.sol".
- Middle Panel:**
 - Description:** A note about updating enum values using the dot operator.
 - Assignment:**
 - Define an enum type called `OrderStatus` with the members `Shipped`, `Cancelled`, and `Delivered`.
 - Initialize the variable `status` of the enum type `OrderStatus`.
 - Create a getter function `getOrderStatus()` that returns the value of the variable `status`.
 - Buttons:** "Check Answer" (blue), "Show answer" (orange), and "Next" (disabled).
 - Bottom Status:** "Well done! No errors."
- Right Panel:**
 - Compiler:** Shows the Solidity code for the contract.
 - Logs:** A message indicating "LearnETH is modifying remix-project-origenesis-workshop/8.4 Data Structures - Enums/Enums.sol.sol".


```
// update status by providing enum member name
function setOrderStatus(OrderStatus status) public {
    status = status;
}

// You can update to a specific enum like this
function cancel() public {
    status = OrderStatus.Cancelled;
}

// delete resets the enum to its first value
function reset() public {
    delete status;
}
```
 - Contracts:** Shows the current state of the contract with 0 pending transactions.

16

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

- Left Panel:** Shows the project structure with a file named "8. Data Locations.sol".
- Middle Panel:**
 - Description:** A note about setting values in memory.
 - Assignment:**
 - Change the value of the `internal` member `key` inside the `Settable` to 4.
 - Create a new struct `SettableWithKey` with the data location memory inside the `Settable` and assign it the value of `Settable`. Change the value of the `internal` member `key` to 5.
 - Create a new struct `SettableWithKey` with the data location memory inside the `Settable` and assign it the value of `Settable`. Change the value of the `internal` member `key` to 3.
 - Let the function return `Settable`, `SettableWithKey`, and `SettableWithKey`.
 - Tip:** Make sure to create the correct return types for the functions.
 - Buttons:** "Check Answer" (blue), "Show answer" (orange), and "Next" (disabled).
 - Bottom Status:** "Well done! No errors."
- Right Panel:**
 - Compiler:** Shows the Solidity code for the contract.
 - Logs:** A message indicating "LearnETH is modifying remix-project-origenesis-workshop/8.4 Data Structures - Enums/Enums.sol.sol".


```
internal storage internal {
    uint key;
}

// You can interact with memory variables
function get(uint _key) public pure returns (uint) {
    return key;
}

// You can interact with storage variables
function Settable() internal {
    key = 4;
}

function SettableWithKey() internal {
    key = 5;
}

function SettableWithKey() internal {
    key = 3;
}

function SettableWithKey() internal {
    key = 4;
}

function SettableWithKey() internal {
    key = 5;
}

function SettableWithKey() internal {
    key = 3;
}
```
 - Contracts:** Shows the current state of the contract with 0 pending transactions.

- Tutorial no.

- Tutorial no. 17

```

contract EtherAndWei {
    uint public wei = 1 ether;
    uint Wei = 10**18;
    bool public checkWei = 1 wei == 1;

    function checkWei() {
        if(wei >= Wei) {
            checkWei = true;
        }
    }
}

```

Assignment

- Create a `uint` variable called `weiWei` and set it to 1 `wei`.
- Create a `bool` variable called `checkWei` 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 `weiWei` in the contract.

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

Well done! No errors.

18

```

contract Gas {
    uint public gas = 0;
    uint public cost = 20000;

    // using all of the gas that you send causes gas transactions to fail.
    // state changes are undone.
    // any value are not returned.
    function forward(uint _gas) {
        // this will cause 100% of the gas to be spent
        // and no transaction fails.
        while(true) {
            _gas -= 1;
        }
    }
}

```

Assignment

Create a new `uint` state-variable in the `Gas` contract called `gas` of the type `uint256`. 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.

- Tutorial no.
- Tutorial no. 19

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with 'LEARNETH' and a 'Transactions - Sending Ether' section. Below it is an 'Assignment' section with instructions to build a charity contract with a 'donate' function and a 'withdraw' function. A 'Check Answer' button is present. The main area contains the Solidity code for the 'CharityFund' contract:

```
contract CharityFund {
    address public owner;
    uint256 public totalBalance;

    constructor() public {
        owner = msg.sender;
        totalBalance = 0;
    }

    function donate() public payable {
        totalBalance += msg.value;
    }

    function withdraw() public {
        uint amount = address(this).balance;
        (bool sent, bytes memory data) = owner.call{value: amount}("");
        require(sent, "Failed to send ether");
    }
}
```

The Remix interface also includes tabs for 'Contracts', 'Blocks', 'Transactions', and 'Logs'. A status bar at the bottom indicates 'Welcome to Remix 1.5.3' and shows file statistics: 'Your files are stored in /remix, 1.92 MB / 3 GB used'.

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.