# EXPERIMENT NO: 01 DATE:

**AIM:** Write a solidity program to demonstrate Access modifiers in solidity

# DESCRIPTION:

Access modifiers in Solidity define the visibility and accessibility of functions and state variables within a smart contract. There are four main types:

1. **public**
   * Accessible from **anywhere** (inside the contract, derived contracts, and external accounts).

## Example:

* + uint public number; // Can be accessed by anyone

1. **private**
   * Accessible **only within the contract** where it is defined.
   * Not inherited by derived contracts.
2. **internal**
   * Accessible **within the contract and derived contracts** but **not externally**.
   * Default visibility for state variables.
3. **external**
   * Can be accessed **only from outside** the contract (not from within).
   * Useful for saving gas in public functions.

Understanding access modifiers is crucial for ensuring security and proper contract design in Solidity. ˙s.¸•'7

# PROGRAM:

****

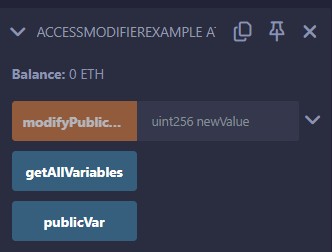
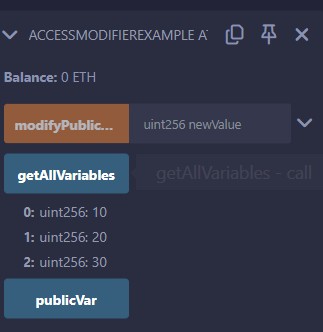
* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy.
* **Modify Public Variable** – Call modifyPublicVariable(newValue), then verify with

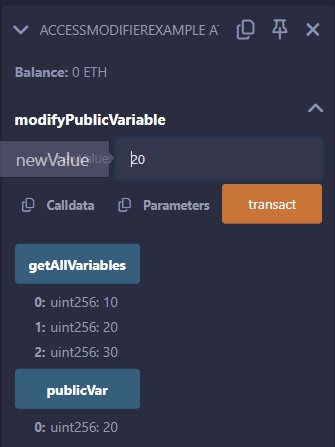
getAllVariables().

## Modify Internal & Private Variables –

* + modifyInternalVariable() needs a derived contract.
  + modifyPrivateVariable() is inaccessible outside the contract.
* **Verify Changes** – Call getAllVariables() again; only publicVar will change.

OUTPUT:



# RESULT:

The Solidity smart contract **AccessModifierExample** was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution, the contract correctly enforced access control using different visibility modifiers. The publicVar was accessible and modifiable externally, while internalVar was only accessible within the contract and derived contracts. The privateVar remained restricted to the contract itself. The getAllVariables function successfully retrieved the values of all variables, demonstrating the correct application of public, internal, and private access modifiers. This experiment effectively showcased access control in Solidity.

# EXPERIMENT NO: 02 DATE:

**AIM:** Write a solidity program to demonstrate all arithmetic operations

# DESCRIPTION:

## Introduction:

Solidity supports various arithmetic operations such as addition, subtraction, multiplication, division, and modulus. It also provides increment and decrement operations. This program demonstrates all arithmetic operations using a Solidity smart contract.

## Concepts Used:

* Arithmetic Operators: +, -, \*, /, %
* Increment and Decrement Operators: ++, --
* Functions in Solidity

## Implementation:

The Solidity contract Arithmetic defines a function arithop that takes two unsigned integers (uint a and uint b) as input and returns the results of arithmetic operations.

# PROGRAM:

// SPDX-License-Identifier: GPL-3.0 pragma solidity >=0.7.0 <0.9.0;

// Solidity program to demonstrate Arithmetic Operations contract Arithmetic {

function arithop(uint a, uint b) public pure returns (uint, uint, uint, uint, uint, uint, uint) {

uint sum = a + b; // Addition uint sub = a - b; // Subtraction

uint mul = a \* b; // Multiplication uint div = a / b; // Division

uint mod = a % b; // Modulus uint inc = a + 1; // Increment uint dec = b - 1; // Decrement

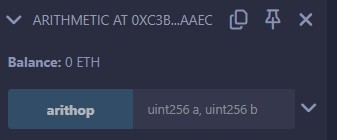
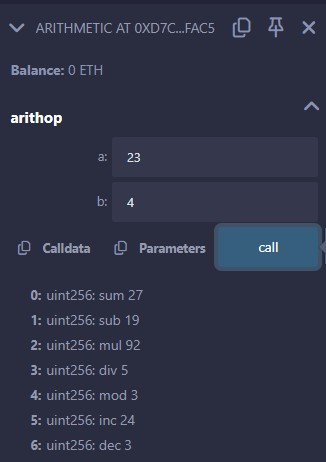
return (sum, sub, mul, div, mod, inc, dec);

}

}

* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy..
  + Select any Account.
  + Set “GAS LIMIT” as per requirement.
  + Set “VALUE” (amount to be sent along with the smart contract transaction).
  + Finally, Deploy the smart contract.

# OUTPUT:

** **

**RESULT:**

The Solidity smart contract Arithmetic was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the arithop function with valid inputs, the contract correctly returned the results of various arithmetic operations, including addition, subtraction, multiplication, division, modulus, increment, and decrement.

This experiment successfully demonstrated the implementation and execution of arithmetic operations in Solidity.

# EXPERIMENT NO: 03 DATE:

**AIM:** Write a solidity program to demonstrate Static & Dynamic Byte array

# DESCRIPTION:

In Solidity, **byte arrays** are used to handle sequences of bytes, and they come in two types: **Static** and **Dynamic**.

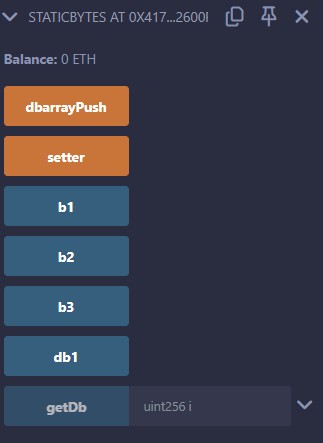
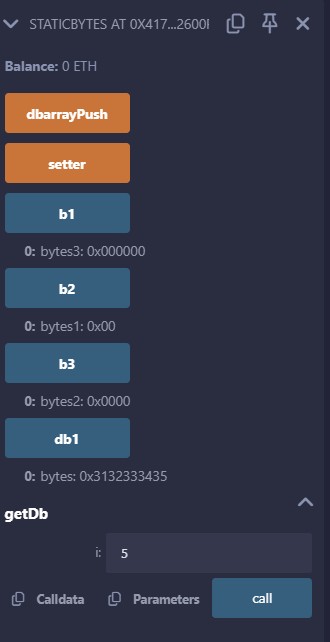
1. Static Byte Arrays:
   * Declared with a fixed size at compile time.
   * Uses the bytes1 to bytes32 types.
   * Memory-efficient for handling fixed-size data like addresses or hashes.
   * Example: bytes32 fixedData;
2. Dynamic Byte Arrays:
   * Declared as bytes (no size specified), meaning they can grow or shrink at runtime.
   * Useful for handling variable-length data, like strings or dynamic payloads.
   * Example: bytes dynamicData;

# PROGRAM:

****

* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy.
* **Check Initial Values:** b1(), b2(), b3() → 0x00…; db1() → "12345".
* **Set Values:** Call setter() → Updates b1 = "abc", b2 = 0x12, b3 = "ab".
* **Push to Dynamic Array:** Call dbarrayPush() → db1() becomes "123456".
* **Access Bytes:** Use getDb(i) to get each byte (getDb(0) → "1", getDb(5) → "6").

# OUTPUT:

# RESULT:

The Solidity smart contract for Static and Dynamic Arrays was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of functions handling static and dynamic arrays with valid inputs, the contract correctly performed operations such as array initialization, element insertion, retrieval, updating, and deletion. Additionally, the dynamic array demonstrated the ability to resize as elements were added or removed.

This experiment successfully demonstrated the implementation and manipulation of static and dynamic arrays in Solidity.

# EXPERIMENT NO: 04 DATE:

**AIM:** Write a solidity program to demonstrate Creation of static & dynamic arrays

# DESCRIPTION:

Arrays in Solidity are used to store multiple values of the same data type. They can be **static** (fixed-size) or

**dynamic** (resizable).

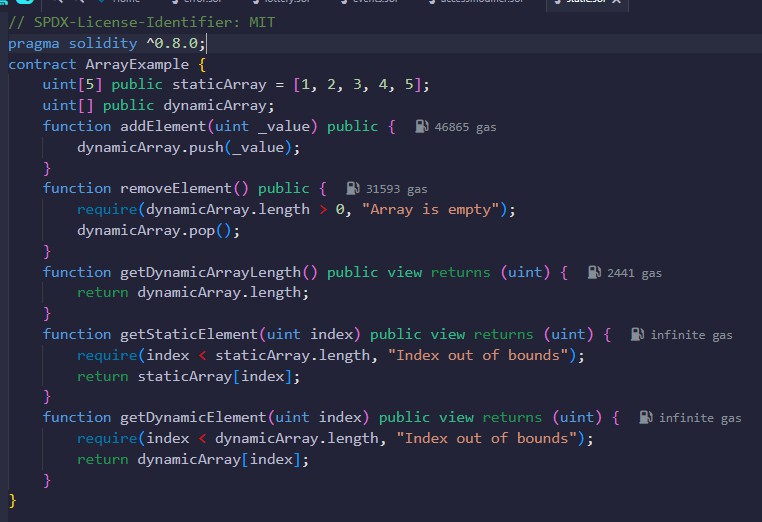
## Static Arrays

* + Have a fixed size defined at the time of declaration.
  + Cannot be resized after deployment.

## Dynamic Arrays

* + Do not have a predefined size and can grow or shrink dynamically.
  + Can be modified using functions like push() and pop().

# PROGRAM:

****

* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy

1. **Check Static Array** – Call staticArray(index) to get a value.
2. **Add to Dynamic Array** – Use addElement(value), then verify with

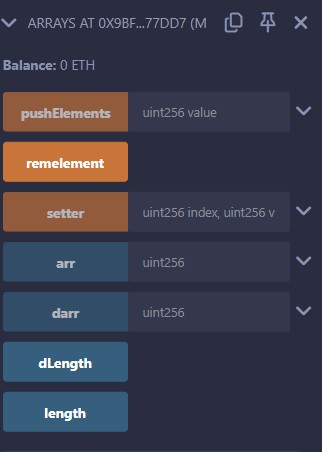
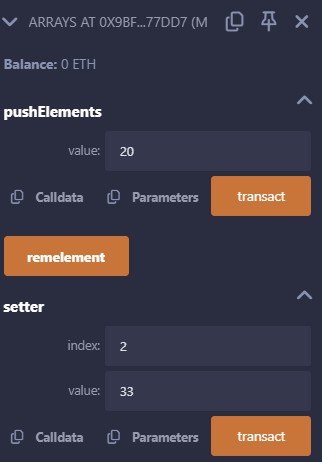
getDynamicElement(index).

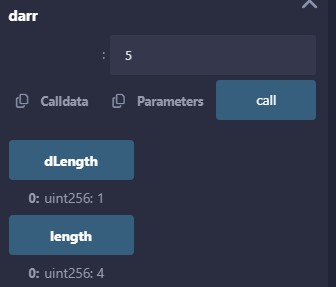
1. **Check Dynamic Array Length** – Call getDynamicArrayLength().
2. **Remove Last Element** – Use removeElement(), then check length again.

## Outcome:

* + Static array remains unchanged.
  + Dynamic array grows with push() and shrinks with pop().
  + Retrieved values match stored values.OUTPUT:

# OUTPUT:

** **

****

**RESULT:**

The Solidity smart contract for **Static & Dynamic Arrays** was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of functions handling both arrays with valid inputs, the contract correctly performed operations such as element insertion, retrieval, length calculation, and element removal.

* + The **static array** maintained a fixed size and allowed retrieval of elements by index.
  + The **dynamic array** demonstrated the ability to dynamically resize using push() and pop().
  + The contract correctly returned array lengths and elements when queried.

This experiment successfully demonstrated the implementation and manipulation of **static and dynamic arrays** in Solidity.

# EXPERIMENT NO: 05 DATE:

**AIM:** Write a solidity program to demonstrate Operations on Dynamic Arrays

# DESCRIPTION:

The Dynamic arrays are the arrays that are allocated memory at the runtime and the memory is allocated from the heap.

Syntax:

*// declaration of dynamic array*

## uint[ ] public arr; int [ ] private arr;

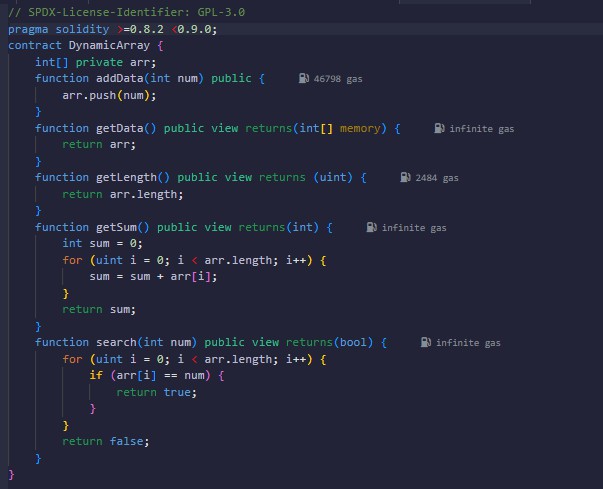
How They Are Different From Fixed Size Arrays?

The fixed-size array has a fixed memory size whereas, in dynamic arrays, the size can be randomly updated during the run time which may be considered efficient with respect to the memory complexity of the code.

Problem: How to create a dynamic array in solidity and perform its associated operations? Solution: In this article, we will create dynamic arrays in solidity language and will perform the following operations on it:

1. Add data in an array
2. Get data of an array
3. Get length of an array
4. Get sum of elements of an array
5. Search a particular element in an array.

**PROGRAM:**

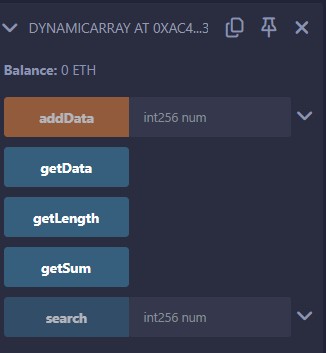
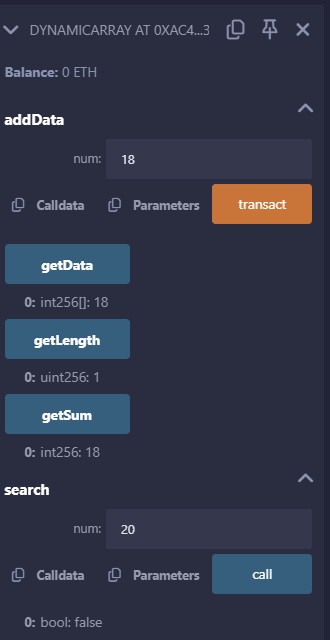
****

* + Press Ctrl+S to Save and Compile.
  + If no errors occurred proceed to Deploy.
  + **Deploy:** Use Remix to compile and deploy

## Test Functions:

* + **addData:** Enter a number and click **addData** to add elements.
  + **getData:** Click to view all elements.
  + **getLength:** Click to get the total number of elements.
  + **getSum:** Click to get the sum of all elements.
  + **search:** Enter a number and click **search** to check if it exists.
  + **Verify:** Check the transaction logs to confirm proper execution.

# OUTPUT:

** **

**RESULT:**

The Solidity smart contract for Dynamic Arrays was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of functions handling the dynamic array with valid inputs, the contract correctly performed operations such as element insertion, retrieval, length calculation, sum calculation, and element search. The dynamic array demonstrated the ability to dynamically resize as elements were added. This experiment successfully demonstrated the implementation and manipulation of dynamic arrays in Solidity.

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 06 **DATE**:  **AIM:** Write a solidity program to perform hashing with keccack-256  **DESCRIPTION:**  **Hashing** refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.  A cryptographic hash function is an algorithm that takes an arbitrary amount of data as input and produces the enciphered text of fixed size. Even a slight change in the input gives a completely different output.  Ethereum uses Keccak for hashing which is similar but not the same as SHA-256. For proof of work, it uses a custom scheme called ethash which is designed to be ASIC-resistant.  **PROGRAM:** | |
| // SPDX-License-Identifier: GPL-3.0  pragma solidity >=0.8.2 <0.9.0;  // Creating a contract contract hash {  // We want hash to be of 8 digits  // hence we store 10^8 which is  // used to extract first 8 digits  // later by Modulus uint hashDigits = 8;  // Equivalent to 10^8 = 8  uint hashModulus = 10 \*\* hashDigits;  // Function to generate the hash value  function \_generateRandom(string memory \_str) public view returns (uint) {  // "packing" the string into bytes and  // then applying the hash function.  // This is then typecasted into uint.  uint random = uint(keccak256(abi.encodePacked(\_str)));  // Returning the generated hash value return random % hashModulus;  }  function \_generateRandom1(string memory \_str1) public view returns (uint) {  // "packing" the string into bytes and  // then applying the hash function.  // This is then typecasted into uint.  uint random1 = uint(sha256(abi.encodePacked(\_str1)));  // Returning the generated hash value return random1 % hashModulus;  }  } |  |
| 11 | |

* + Press Ctrl+S to Save and Compile.
  + If no errors occurred proceed to Deploy.
  + **Deploy:** Use Remix to compile and deploy

## Test Functions:

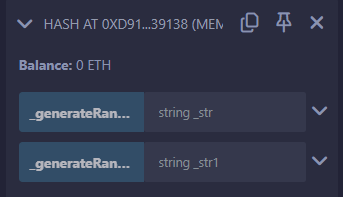
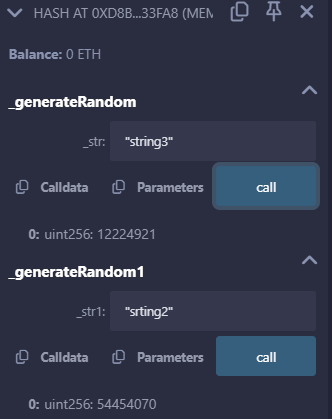
Once deployed, use the buttons under "Deployed Contracts":

* + **\_generateRandom:** Enter a string (e.g., "testString") and click the button.
    - It returns an 8-digit hash using keccak256.
  + **\_generateRandom1:** Enter a string and click the button.
    - It returns an 8-digit hash using sha256.

## Verify:

* + Check the returned values in the Remix terminal to ensure both hash functions work correctly.
  + Test with different inputs to observe unique hash values each time.

**OUTPUT:**

# RESULT:

The Solidity smart contract for performing hashing with keccak-256 was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the \_generateRandom function with valid string inputs, the contract correctly computed an 8-digit hash value by applying the keccak-256 hashing algorithm. This experiment successfully demonstrated the implementation of keccak-256 hashing in Solidity.

12

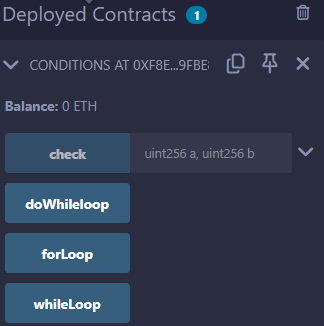
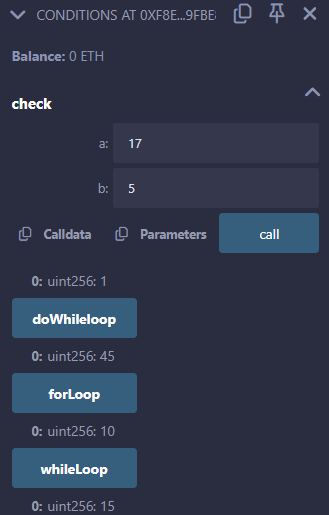
|  |  |
| --- | --- |
| **EXPERIMENT NO:** 07 **DATE**:  **AIM:** Write a solidity program to illustrate the use of loops  As we know, the loops are used when we have to perform an action over and over again. While writing a contract there may be a situation when we have to do some action repeatedly, In this situation, loops are implemented to reduce the number of lines of the statements. Solidity supports following loops too ease down the programming  **While Loop**  This is the most basic loop in solidity, Its purpose is to execute a statement or block of statements repeatedly as far as the condition is true and once the condition becomes false the loop terminates.  **Syntax:** | |
| while (condition) {  statement or block of code to be executed if the condition is True  } |  |
| **Do-While Loop**  This loop is very similar to while loop except that there is a condition check which happens at the end of loop i.e. the loop will always execute at least one time even if the condition is false.  **Syntax:** | |
| do  {  block of statements to be executed  } while (condition); |  |
| **For Loop**  This is the most compact way of looping. It takes three arguments separated by a semi-colon to run. The first one is ‘loop initialization’ where the iterator is initialized with starting value, this statement is executed before the loop starts. Second is ‘test statement’ which checks whether the condition is true or not, if the condition is true the loop executes else terminates. The third one is the ‘iteration statement’ where the iterator is increased or decreased. Below is the syntax of for loop:  **Syntax:** | |
| for (initialization; test condition; iteration statement) {  statement or block of code to be executed if the condition is True  }  PROGRAM: |  |
| 13 | |

|  |  |  |
| --- | --- | --- |
|  | | |
| // SPDX-License-Identifier: GPL-3.0 | |  |
| pragma solidity >=0.7.0 <0.9.0; | |
| contract conditions{ | |
| function check(uint a, uint b) public pure returns(uint){ | |
| if (a>b) { | |
| return 1; | |
| }else if(a==b){ | |
| return 0; | |
| } else{ | |
| return 2; | |
| } | |
| } | |
| function whileLoop() public pure returns (uint){ | |
| uint sum; | |
| uint count; | |
| while(count<=5){ | |
| sum=sum+count; | |
| count=count+1; | |
| } | |
| return sum; | |
| } | |
| function forLoop() public pure returns(uint){ | |
| uint c; | |
| uint s; | |
| for( c=0; c<5; c++){ | |
| s=s+c; | |
| } | |
| return s; | |
| } | |
| function doWhileloop() public pure returns(uint){ | |
| uint cc; | |
| uint ss; | |
| do{ | |
| ss=ss+cc; | |
| cc=cc+1; | |
| }while(cc<10); | |
| return ss; | |
| } | |
| } | |
| } | |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy | 14 |  |

## Test:

* + **check(a, b):** Returns 1 if a > b, 0 if a == b, 2 if a < b.
  + **whileLoop():** Returns 15.
  + **forLoop():** Returns 10.
  + **doWhileloop():** Returns 45.

# OUTPUT:

**RESULT:**

The Solidity smart contract for conditional statements and loops was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the functions with valid inputs, the contract correctly performed the following operations:

* + **check(a, b):** Compared two numbers, returning 1 if a > b, 0 if a == b, and 2 if a < b.
  + **whileLoop():** Accurately calculated the sum of numbers from 0 to 5, returning 15.
  + **forLoop():** Successfully iterated from 0 to 4, returning the sum 10.
  + **doWhileloop():** Correctly executed the do-while loop, summing numbers from 0 to 9, returning 45.

This experiment effectively demonstrated the use of conditional statements and loop structures in Solidity.

# EXPERIMENT NO: 08 DATE:

**AIM:** Write a solidity program to perform Modifiers in solidity

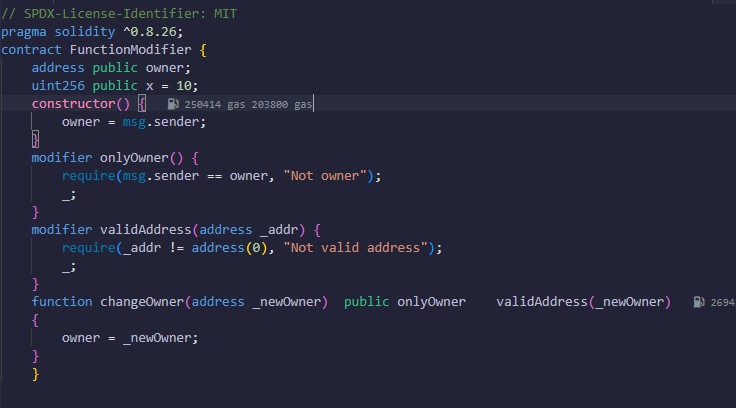
**DESCRIPTION:**

The Solidity smart contract **FunctionModifier** demonstrates the use of function modifiers to enforce access control and input validation.

* + The contract defines an **owner** variable, initialized to the deployer’s address in the constructor.
  + The **onlyOwner** modifier ensures that only the contract owner can execute specific functions.
  + The **validAddress** modifier checks that a given address is not the zero address (0x0).
  + The changeOwner() function allows transferring ownership but only if the caller is the owner and the new address is valid.

This contract effectively implements **function modifiers** to improve security and code reusability in Solidity.

# PROGRAM:

****

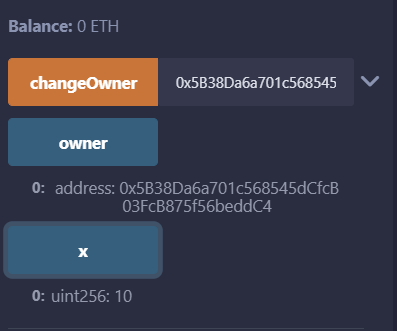
* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy

1. **Check Owner** – Call owner() to verify the initial owner.
2. **Unauthorized Change** – Try changeOwner(newAddress) from a different account (should fail: "Not owner").
3. **Valid Owner Change** – Owner calls changeOwner(newAddress) with a valid address (should succeed).
4. **Invalid Address Test** – Try changeOwner(0x0...) (should fail: "Not valid address").

Outcome:

* + Ownership can only be transferred by the owner.
  + Invalid addresses are rejected.
  + Function modifiers enforce security and validation.

# OUTPUT:

****

RESULT:

The Solidity smart contract for **Function Modifiers** was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the changeOwner function, the contract correctly enforced access control using the onlyOwner modifier, ensuring that only the contract owner could update the owner address.

Additionally, the validAddress modifier prevented the assignment of an invalid (zero) address. Unauthorized ownership transfer attempts were rejected, while valid ownership changes were successfully executed.

This experiment successfully demonstrated the implementation of **function modifiers** in Solidity for

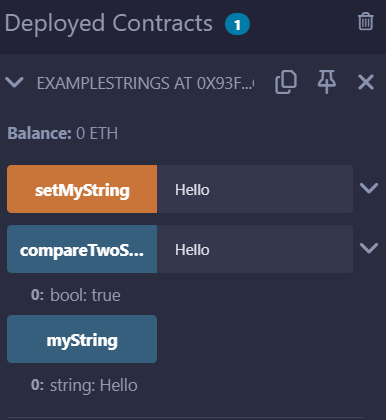
## access control and input validation.

|  |  |  |
| --- | --- | --- |
| **EXPERIMENT NO:** 09 **DATE**:  **AIM:** Write a solidity program to compare two strings  **DESCRIPTION:**  Often in programming, we want to compare two strings. By comparing, I mean we want to identify whether the two strings are equivalent to each other or not.  This is done by using the following operators in other languages:   ==: This checks whether two strings are equal   * !=: This checks if two strings are not equal   In Solidity, there is no native function to compare strings. It is also not possible to compare the equality of two strings by just using the == operator.  Currently, the == operator is only supported for booleans, integers, addresses, and static byte arrays. Solidity does not support == for dynamic arrays, and strings are dynamic arrays.   * keccak256() is a hashing function supported by Solidity * abi.encodePacked()encodes a value using the Application Binary Interface (ABI)   Keccak256() is a cryptographic function built into Solidity. This function takes in any amount of inputs and converts it to a unique 32-byte hash.  It takes a single bytes parameter as input, so we have to convert our string(s) to bytes first. Therefore, we use the abi.encodePacked() function.  It converts input into bytes array.  **PROGRAM:** | | |
| //SPDX-License-Identifier: MIT pragma solidity ^0.8.15; contract ExampleStrings {  string public myString = "Hello World";  function setMyString(string memory \_myString) public { myString = \_myString;  }  function compareTwoStrings(string memory \_myString) public view returns(bool) {  return keccak256(abi.encodePacked(myString)) == keccak256(abi.encodePacked(\_myString));  }  } | |  |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy | 18 |  |

**Test Functions:**

* + **myString:** Click to view the initial value "Hello World".
  + **setMyString:** Enter a new string (e.g., "Blockchain") and click the button to update myString.
  + **compareTwoStrings:** Enter a string and click the button to compare it with myString. It returns:
    - true if the strings are equal.
    - false if they are different.

# OUTPUT:

****

**RESULT:**

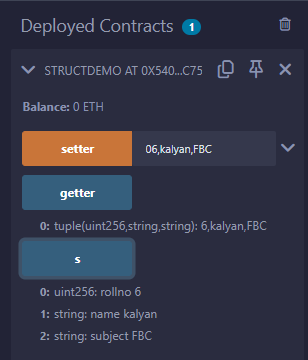
The Solidity smart contract for string manipulation was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the setMyString function, the contract correctly updated the myString variable with the provided input. The compareTwoStrings function accurately compared the stored string with the input string using the Keccak-256 hash function, returning true for matching strings and false for non-matching strings.

This experiment successfully demonstrated string storage, updating, and comparison in Solidity using Keccak-256 hashing.

19

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 10 **DATE**:  **AIM:** Write a solidity program that consist of student data  **DESCRIPTION:**  Structs in Solidity allows you to create more complicated data types that have multiple properties. You can define your own type by creating a **struct**.  They are useful for grouping together related data.  Structs can be declared outside of a contract and imported in another contract. Generally, it is used to represent a record. To define a structure *struct* keyword is used, which creates a new data type.  Syntax: | |
| struct <structure\_name> {  <data type> variable\_1;  <data type> variable\_2;  } |  |
| For accessing any element of the structure, ‘dot operator’ is used, which separates the struct variable and the element we wish to access. To define the variable of structure data type structure name is used.  **PROGRAM:**     * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy * **Set Data:** Use the setter function — input roll number, name, subject, and click **Transact**. * **Get Data:** Click the getter function to view stored student details.   20 | |

# OUTPUT:



RESULT:

The Solidity smart contract for struct demonstration was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the setter and getter functions with valid inputs, the contract correctly stored and retrieved student details, including roll number, name, and subject. This experiment effectively showcased the use of structs for organizing related data in Solidity.

# EXPERIMENT NO: 11 DATE:

**AIM:** Write a solidity program to demonstrate Mapping in solidity

# DESCRIPTION:

Mapping in Solidity acts like a hash table or dictionary in any other language. These are used to store the data in the form of key-value pairs, a key can be any of the built-in data types but reference types are not allowed while the value can be of any type. Mappings are mostly used to associate the unique Ethereum address with the associated value type.

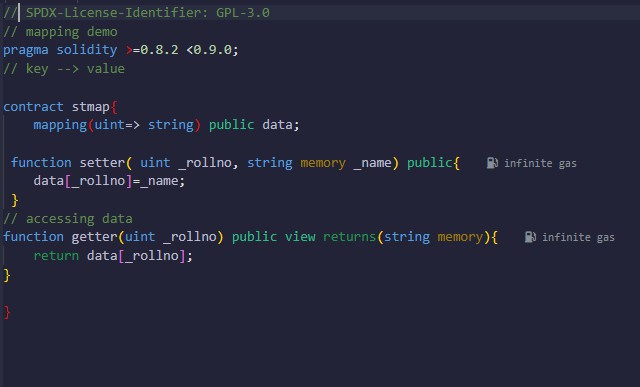
Syntax:

mapping(key => value) <access specifier> <name>; Creating a Mapping

Mapping is defined as any other variable type, which accepts a key type and a value type.

Example: In the below example, the contract mapping\_example a structure is defined and mapping is created.

# PROGRAM:

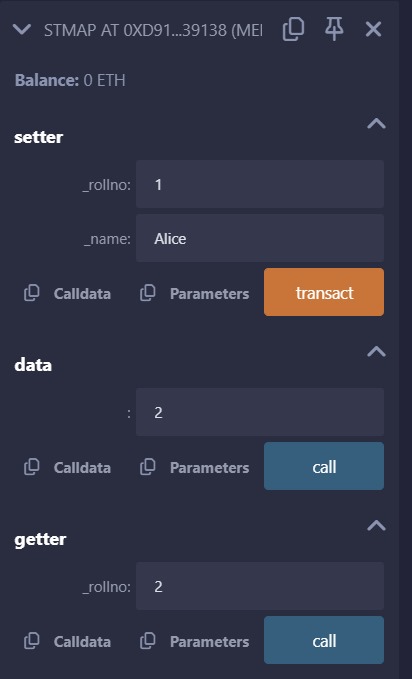
****

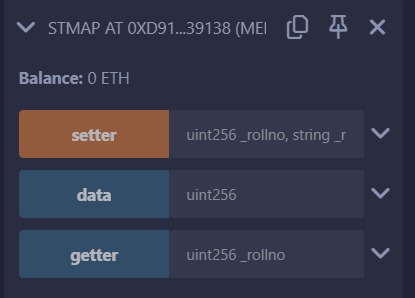
* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy
* **Set Data:** Use the setter function to add data (\_rollno = 101, \_name = "Alice"), then click

## transact.

* **Get Data:** Use the getter function with \_rollno = 101 and click **call** to see "Alice".
* **Direct Access:** Use data[101] to get "Alice" directly.

# OUTPUT:





**RESULT:**

The Solidity smart contract for mapping demonstration was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the setter and getter functions with valid inputs, the contract correctly stored and retrieved student names using roll numbers as keys. This experiment effectively showcased the use of mappings for associating unique identifiers with corresponding values in Solidity.

|  |  |  |
| --- | --- | --- |
| **EXPERIMENT NO:** 12 **DATE**:  **AIM:** Write a solidity program to demonstrate importance of global variables  **DESCRIPTION:**  Solidity offers various global variables, each serving a distinct purpose. Some of the essential global variables include:  **Block Variables**: These variables provide information about the current block on the Ethereum blockchain. Examples include block.number, block.timestamp, block.difficulty, and block.coinbase.  **Transaction Variables**: They offer insights into the current transaction’s details, such as msg.sender, msg.value, and msg.data.  **Contract Variables**: These variables provide information about the current contract itself. Examples include this.balance, which indicates the contract’s Ether balance.  **4. Gas Variables**: Solidity also includes global variables related to gas, such as gasleft(), which returns the amount of gas remaining in the current transaction.  These variables cannot be changed by smart contracts. | | |
| Variable | Type | Description |
| msg.sender | address | The address of the account that sent the current transaction. |
| msg.value | uint | The amount of Ether sent with the current transaction. |
| block.coinbase | address | The address of the miner who mined the current block. |
| block. Difficulty | uint | The difficulty of the current block. |
| block.gaslimit | uint | The maximum amount of gas that can be used in the current block. |
| block. Number | uint | The number of the current block. |
| block.  Timestamp | uint | The timestamp of the current block. |
| now | uint | An alias for block.timestamp. |
| TX. Origin | address | The address of the account that originally created the transaction (i.e., the sender of the first transaction in the call chain). |
| tx.gasprice | uint | The gas price (in Wei) of the current transaction. |
| 25 | | |

1.

2.

3.

.

|  |
| --- |
| **PROGRAM:** |
| / SPDX-License-Identifier: GPL-3.0  // Solidity program to implement  // global variables pragma solidity ^0.8.0;  contract GlobalVariablesExample {  address public owner; constructor()  {  // set the contract owner to the address  // that deployed the contract owner = msg.sender;  }  function getOwner() public view returns (address)  {  // return the contract owner address return owner;  }  function isOwner(address \_address) public view returns (bool)  {  // check if the provided address matches  // the contract owner return \_address == owner;  }  function sendEther(address payable \_recipient) public payable  {  // send ether to the specified recipient require(msg.sender == owner,  "Only the contract owner can send ether.");  \_recipient.transfer(msg.value);  }  function getCurrentBlock() public view returns (uint, uint, address)  {  // return the current block number, timestamp,  // and coinbase address  return (block.number, block.timestamp, block.coinbase);  }  } |  |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy * **Get Owner:** Click owner() or getOwner() to see the deployer's address. * **Verify Owner:** Use isOwner(address) to check if an address is the owner.   26 |

* **Send Ether:** Enter the recipient’s address, set the value, and call sendEther().
* **Get Block Info:** Click getCurrentBlock() for the block number, timestamp, and coinbase address.

# OUTPUT:

**RESULT:**

The Solidity smart contract for global variables was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of various functions, the contract accurately retrieved the deployer's address, verified ownership, transferred Ether to specified recipients, and fetched current block details such as block number, timestamp, and coinbase address. This experiment effectively demonstrated the use of global variables in Solidity to access blockchain-specific data and manage contract ownership.

# EXPERIMENT NO: 13 DATE:

**AIM:** Write a solidity program to demonstrate Error handling

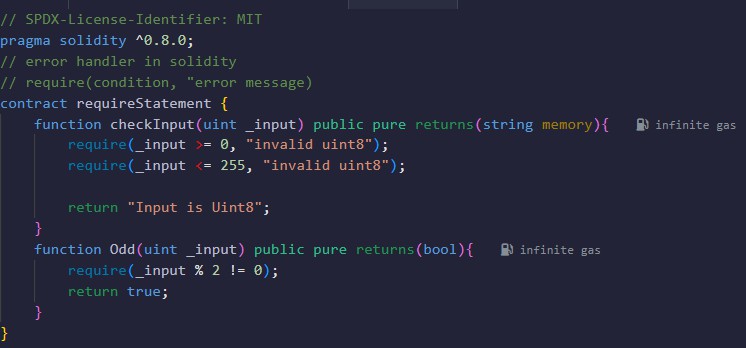
**DESCRIPTION:**

Solidity has many functions for error handling. Errors can occur at compile time or runtime. Solidity is compiled to byte code and there a syntax error check happens at compile-time, while runtime errors are difficult to catch and occurs mainly while executing the contracts. Some of the runtime errors are out-of- gas error, data type overflow error, divide by zero error, array-out-of-index error, etc. Until version 4.10 a single throw statement was there in solidity to handle errors, so to handle errors multiple if…else statements, one has to implement for checking the values and throw errors which consume more gas. After version 4.10 new error handling construct assert, require, revert statements were introduced and the throw was made absolute.

**Require Statements**

The ‘require’ statements declare prerequisites for running the function i.e. it declares the constraints which should be satisfied before executing the code. It accepts a single argument and returns a boolean value after evaluation, it also has a custom string message option. If false then exception is raised and execution is terminated. The unused gas is returned back to the caller and the state is reversed to its original state.

**PROGRAM:**

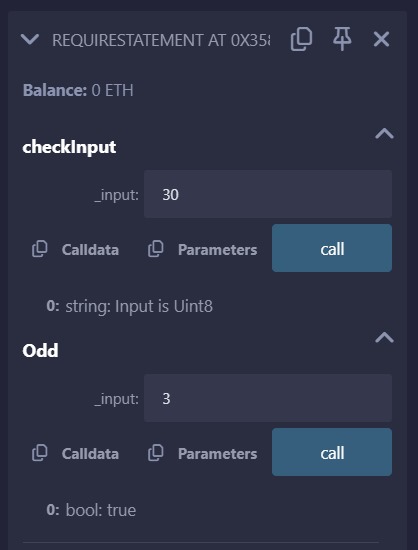
****

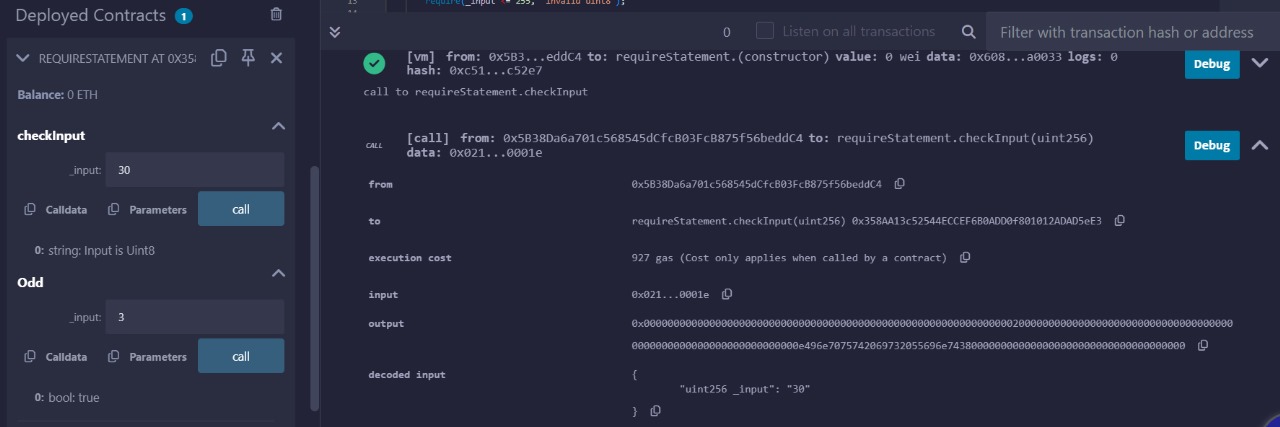
* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy
* **Test checkInput()**
  + Enter a number (0-255) and click "Transact."
  + If valid, returns "Input is Uint8".
  + If invalid, shows an error.
* **Test Odd()**
  + Enter an odd number → Returns true.
  + Enter an even number → Transaction fails.

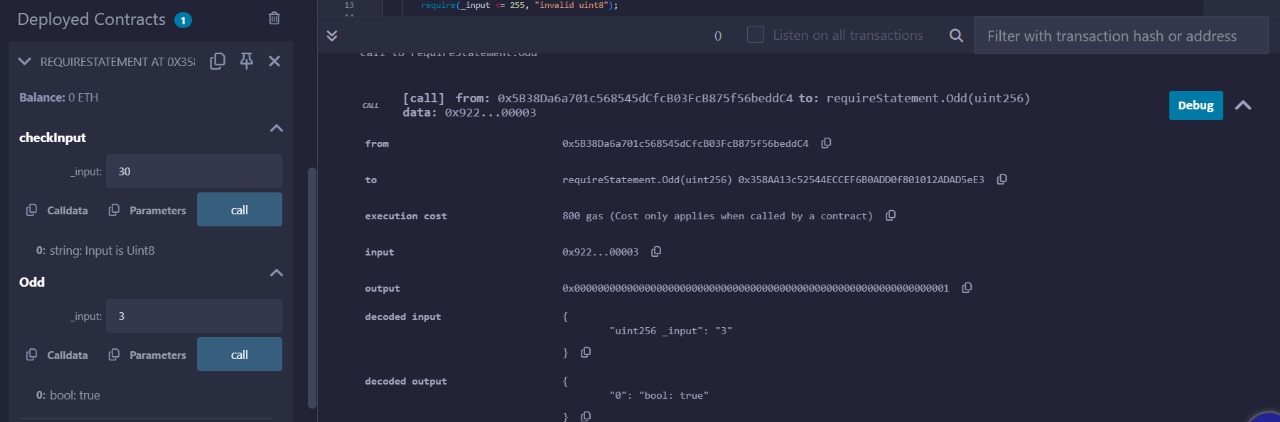
## Debug Errors

* + If require fails, Remix shows an error message.
  + Adjust input values and retry.

OUTPUT:







**RESULT:**

The Solidity smart contract for error handling was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of various functions, the contract correctly validated input conditions using the require statement. The checkInput() function ensured that the input was within the valid uint8 range (0-255), while the Odd() function accurately determined whether a number was odd, reverting the transaction for even numbers. This experiment effectively demonstrated the use of the require statement in Solidity to enforce conditions and prevent invalid operations.

# EXPERIMENT NO: 14 DATE:

**AIM:** Write a solidity program to demonstrate Lottery smart contract

**DESCRIPTION:**

Objective: Players play a lottery game by spending some ethers, and winner will get all the money. Prerequisites

1. Solidity >0.8.0
2. Ethereum development environment

Description: This smart contract has two entities..a manager and a player The manager can not participate in lottery game

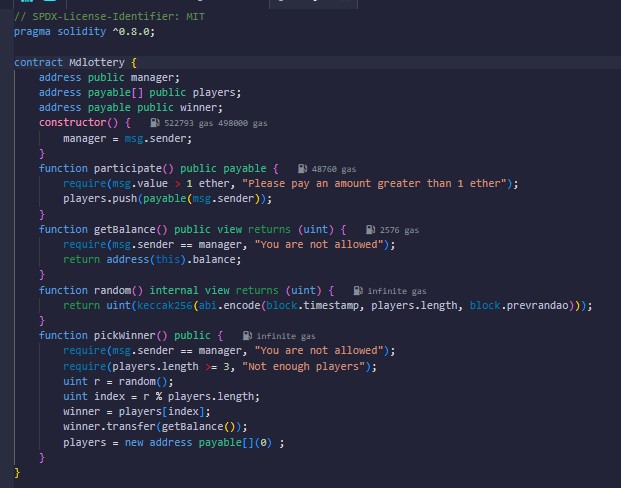
Players: Each player enters into lottery after paying minimum money

Manager: Manager account collects all the money Manager picks up winner among the players and transfers money to winner and restarts the game

Manager will decide the winner if there are more than three participants

Functions: Participate() Get balance() Pickwinner()

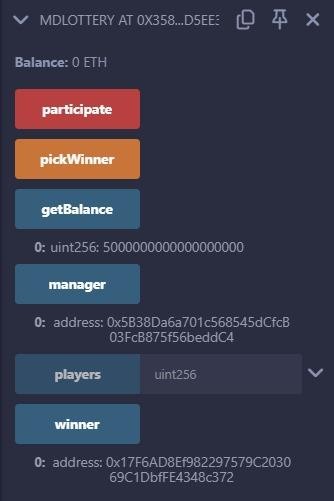
# PROGRAM :

****

* Press Ctrl+S to Save and Compile.
* If no errors occurred proceed to Deploy.
* **Deploy:** Use Remix to compile and deploy
* **Players Participate** by calling participate() with **at least 1 ETH**.
* **Check Balance** using getBalance() (only manager can call).
* **Pick a Winner** by calling pickWinner() (only manager can call).
* **Verify Winner** using winner() and check if the prize was transferred.
* **Lottery Resets** automatically, and players.length becomes **0**.

# OUTPUT:

# 

****

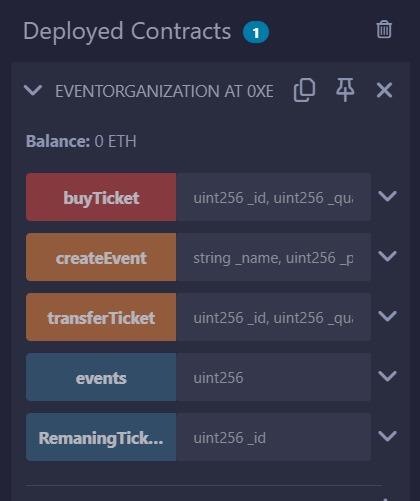
**RESULT:**

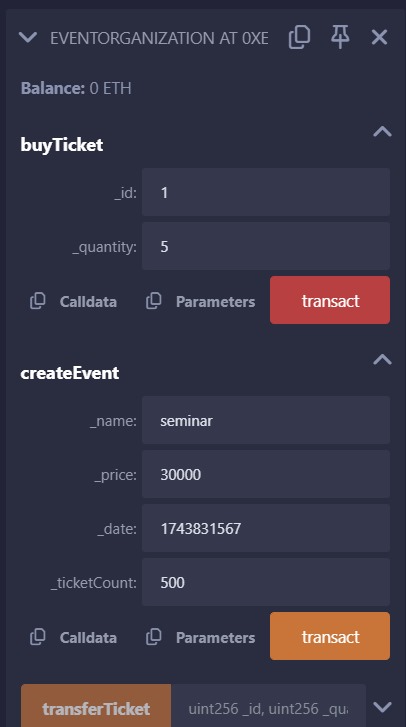
The Solidity smart contract for **Lottery Participation** was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the participate function, the contract correctly added players to the lottery, ensuring that each participant contributed more than 1 Ether.The getBalance function accurately returned the contract balance but was restricted to the manager. The pickWinner function ensured that only the manager could select a winner, requiring at least three participants. The winner was chosen using a **pseudo-random hash function**, and the contract correctly transferred the prize to the selected winner.

This experiment successfully demonstrated **lottery participation, fund management, and winner selection** in Solidity using **Ethereum smart contracts**.

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 15 **DATE**:  **AIM:** Write a solidity program to demonstrate Events in solidity  **DESCRIPTION:**  Solidity events give an abstraction on top of the EVM’s logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.  Smart contracts cannot handle logs Thus  Events in Solidity are a logging mechanism that allows smart contracts to communicate with external listeners (such as a dApp frontend or blockchain explorer). Events are stored on the blockchain but are more efficient and cost-effective than storing data in contract storage.  Events are useful for:   * Emitting logs that external applications can listen to. * Providing an efficient way to record contract activity. * Triggering actions in off-chain applications when specific conditions are met.   **PROGRAM:** | |
| / SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract EventExample {  uint public favouriteNumber;  // Declaring an event  event storedNumber(uint indexed oldNumber,uint indexed newNumber, uint addedNumber, address sender);  function store(uint \_newfavouriteNumber) public { favouriteNumber= \_newfavouriteNumber;  // Emitting the event  emit storedNumber(favouriteNumber,\_newfavouriteNumber, favouriteNumber+  \_newfavouriteNumber, msg.sender);  }  function retrieve() public view returns(uint){ return favouriteNumber;  }  } |  |
| 1. Press Ctrl+S to Save and Compile. 2. If no errors occurred proceed to Deploy. 3. **Deploy:** Use Remix to compile and deploy 4. **Check Initial Value** using retrieve() (default: 0).**Store a Number** via store(10). 5. **Verify Event Log** in Remix (shows old & new number, sum, sender). 6. **Retrieve Updated Value** via retrieve() (updated successfully). 7. Contract stores numbers and emits events correctly!   34 | |

# OUTPUT:





**35**

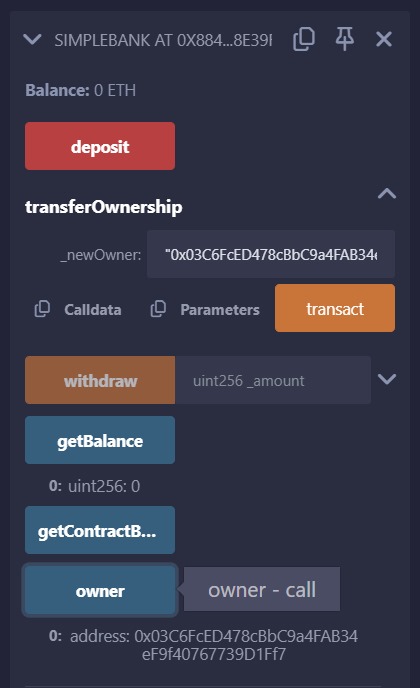
**RESULT:**

The Solidity smart contract for event handling was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the store() function, the contract correctly updated the favouriteNumber variable and emitted the storedNumber event. The event accurately logged the old number, new number, their sum, and the sender’s address. The retrieve() function successfully returned the stored number. This experiment effectively demonstrated the use of Solidity events for logging state changes and enhancing contract transparency.

36

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 16 **DATE**:  **AIM:** Write a solidity program to demonstrate Simple bank operations  **DESCRIPTION:**  p**ragma solidity ^0.8.0;**: Specifies the Solidity compiler version to use. **contract SimpleBank { ... }**: Defines the smart contract.  **mapping(address => uint256) public balances;**: Stores the balances of each user, using their Ethereum address as the key.  **address public owner;**: Stores the address of the contract owner.  **event Deposit(address indexed account, uint256 amount);** and **event Withdrawal(address indexed account, uint256 amount);**: Define events that are emitted when deposits or withdrawals occur. These events are crucial for tracking transactions.  **constructor() { ... }**: Sets the contract owner when the contract is deployed.  **modifier onlyOwner() { ... }**: A modifier that restricts access to certain functions to the contract owner. **deposit() public payable { ... }**: Allows users to deposit Ether into their accounts. The payable keyword is essential for receiving Ether.  **withdraw(uint256 amount) public { ... }**: Allows users to withdraw Ether from their accounts.  **getBalance() public view returns (uint256) { ... }**: Allows users to view their account balance. The view keyword indicates that this function does not modify the blockchain state.  **transfer(address recipient, uint256 amount) public { ... }**: Enables users to transfer funds to other users within the contract.  **Error Handling**: The require() statements are used to handle errors and prevent invalid transactions.  **Events**: events are used to log transactions.  **PROGRAM:** | |
| SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract SimpleBank { address public owner;  mapping(address => uint256) private balances;  event Deposit(address indexed user, uint256 amount); event Withdrawal(address indexed user, uint256 amount);  event OwnershipTransferred(address indexed oldOwner, address indexed newOwner);  modifier onlyOwner() {  require(msg.sender == owner, "Only owner can perform this action");  \_;  }  constructor() {  owner = msg.sender; // Set the deployer as the owner  }  // Function to deposit ETH into the bank function deposit() public payable {  require(msg.value > 0 ether, "Deposit amount must be greater than 0");  balances[msg.sender] += msg.value; |  |
| 37 | |

|  |  |
| --- | --- |
|  | |
| emit Deposit(msg.sender, msg.value);  }  // Function to withdraw ETH from the bank function withdraw(uint256 \_amount) public {  require(\_amount > 0 ether, "Withdraw amount must be greater than 0"); require(balances[msg.sender] >= \_amount, "Insufficient balance");  balances[msg.sender] -= \_amount; payable(msg.sender).transfer(\_amount); emit Withdrawal(msg.sender, \_amount);  }  // Function to check user's balance  function getBalance() public view returns (uint256) { return balances[msg.sender];  }  // Function to check total contract balance (only owner)  function getContractBalance() public view onlyOwner returns (uint256) { return address(this).balance;  }  // Function to transfer ownership  function transferOwnership(address \_newOwner) external onlyOwner { require(\_newOwner != address(0), "New owner cannot be zero address"); emit OwnershipTransferred(owner, \_newOwner);  owner = \_newOwner;  }  } |  |
| Press Ctrl+S to Save and Compile.   * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy.   **OUTPUT:**    38 | |



# RESULT:

The Solidity smart contract for a banking system was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the deposit() function, the contract correctly updated the user's balance and emitted the Deposited event, logging the sender’s address and deposit amount. The withdraw() function successfully deducted the requested amount and emitted the Withdrawn event, ensuring transparency. The checkBalance() function accurately returned the user's current balance. This experiment effectively demonstrated the use of Solidity events for tracking transactions and enhancing financial transparency in smart contracts.

# EXPERIMENT NO: 17 DATE:

**AIM:** Write a solidity program to demonstrate Event organization

**DESCRIPTION:**

This smart contract for event organization is written in solidity it leverages blockchain technology to helps us securely and efficiently manage event details such as ticket sales and payments. It eliminates the need for intermediaries and ensures the terms of the agreement are unalterable. In short, it makes event planning smoother and more secure.

To use this smart contract, you will need a platform that supports smart contract deployment and execution, such as Ethereum. You will also need a tool such as Remix or Truffle to deploy and test the contract.

Features

* Purchase tickets for events
* Transfer tickets to other individuals
* View remaining number of tickets for an event

**Contract Functions**

**Create Events: This method allows a organizer to organization their event**

**buy Ticket: This method allows a user to purchase a ticket for a specified event.**

**Transfer Ticket: This method allows a user to transfer a purchased ticket to another individual.**

**Remaining Tickets: This method returns the number of remaining tickets for a specified event.**

his Solidity smart contract, EventOrganization, manages the creation, purchase, and transfer of tickets for events. Let's break down its components:

## Structure Definition (Event)

Solidity

struct Event {

address organizer; string name;

uint price; uint date;

uint ticketcount; uint ticketRemaning; uint EventID;

}

* This struct defines the data associated with each event:
  + organizer: The address of the event's creator.
  + name: The event's name.
  + price: The ticket price.
  + date: The event's date (presumably a Unix timestamp).
  + ticketcount: The total number of tickets.
  + ticketRemaning: The number of tickets remaining.
  + EventID: A unique ID for the event.

## Mappings

Solidity

mapping (uint => Event) public events;

mapping (address => mapping(uint => uint)) internal Tickets;

* + events: A mapping that associates an event ID ( uint) with an Event struct. This allows you to retrieve event details using the ID.
  + Tickets: A nested mapping that tracks ticket ownership. The outer mapping associates an address with an inner mapping. The inner mapping then associates an event ID with the number of tickets that address holds for that event.

1. **State Variable (nextID)**

Solidity

uint internal nextID;

* + nextID: A counter used to generate unique event IDs. It's incremented each time a new event is created.

1. **createEvent Function**

Solidity

function createEvent(string memory \_name, uint \_price , uint \_date, uint \_ticketCount) external {

require(\_date > block.timestamp, " PLESE GIVE A PROPER DATE FOR THE EVENT"); require(\_ticketCount > 0, " THE TICKET COUNT SHOULD BE MORE THAN 0 "); events[nextID] = Event(msg.sender, \_name, \_price , \_date , \_ticketCount,

\_ticketCount, nextID); nextID++ ;

}

* + This function creates a new event.
  + It takes the event's name, price, date, and ticket count as input.
  + It uses require to ensure that the event date is in the future and that the ticket count is greater than zero.
  + It creates a new Event struct and stores it in the events mapping, using nextID as the key.
  + It increments nextID to prepare for the next event.

1. **Modifier (eventStatus)**

Solidity

modifier eventStatus(uint \_id){

require(events[\_id].date != 0, "Please enter a valid Event Details"); require(events[\_id].date > block.timestamp , "The event has already occured");

\_;

}

* + This modifier is used to validate the event ID and date before executing certain functions.
  + It ensures that the event exists and that it hasn't already occurred.
  + The \_; symbol represents the place where the modified function's code will be inserted.

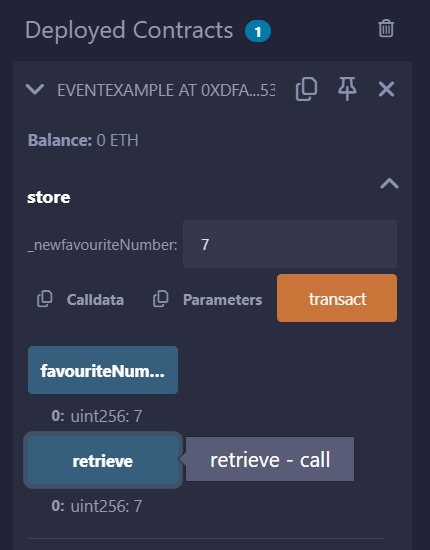
1. **buyTicket Function**

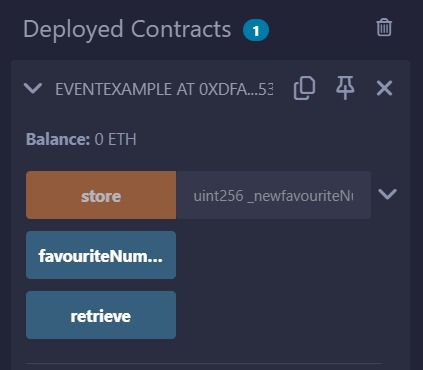
|  |  |
| --- | --- |
| Solidity  function buyTicket(uint \_id, uint \_quantity) external payable eventStatus(\_id) { Event storage \_event = events[\_id];  require(msg.value == \_event.price\*\_quantity , "Please enter a valid Amount"); require(\_event.ticketRemaning >= \_quantity, "Try for lower count or Tickets are  already sold out");  \_event.ticketRemaning -= \_quantity; Tickets[msg.sender][\_id] += \_quantity;  }   * This function allows users to buy tickets. * It takes the event ID and the quantity of tickets as input. * It's payable, meaning it can receive Ether. * It uses the eventStatus modifier for validation. * It checks if the correct amount of Ether was sent. * It checks if there are enough tickets remaining. * It updates the ticketRemaning count and records the ticket purchase in the Tickets mapping.  1. **transferTicket Function**   Solidity  function transferTicket(uint \_id, uint \_quantity, address \_to) external eventStatus(\_id){  require(Tickets[msg.sender][\_id] > \_quantity, "You dont have enough Tickets"); Tickets[msg.sender][\_id] -= \_quantity;  Tickets[\_to][\_id] += \_quantity;  }   * + This function allows users to transfer tickets to another address.   + It checks if the sender has enough tickets.   + It updates the Tickets mapping to reflect the transfer.  1. **RemaningTickets Function**   Solidity  function RemaningTickets(uint \_id) view public eventStatus(\_id) returns(uint){ return events[\_id].ticketRemaning;  }   * + This function returns the number of remaining tickets for a given event.   + It uses the eventStatus modifier for validation.   **PROGRAM:** | |
| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0; contract EventOrganization{  //Structure for details of the event  struct Event { address organizer; string name;  uint price; uint date;  uint ticketcount;  uint ticketRemaning; |  |
| 42 | |

|  |  |
| --- | --- |
|  | |
| uint EventID;  }  // maping uint with a event  // THIS WILL GIVE A PERTICULAR ORGANIZER TO ORGANIZE N NUMBER OF EVENTS.  mapping (uint => Event) public events;  // nested mapping to hold tickets against a perticluar address  // THIS SIGNIFES A PERSON CAN HOLD N NUMBER OF TICKETS OF N NUMBER FOR SHOWS  mapping (address => mapping(uint => uint)) internal Tickets;  // state variable for eventid (nextID) uint internal nextID;  // FUNCTION  // CREATEING EVENT  // -take all the input for the struct  // -it requires the date to be more then the current timestamp.  // -it requires more than 0 ticket for the event.  // -it will now add all the details from the parmeter to the event struct on the 0th index of the mapping by utilizing the initial value of the event id  // -it will add 1 in the event id so the next one  function createEvent( string memory \_name, uint \_price , uint \_date, uint  \_ticketCount) external {  require(\_date >block.timestamp, " PLESE GIVE A PROPER DATE FOR THE EVENT"); require(\_ticketCount > 0, " THE TICKET COUNT SHOULD BE MORE THAN 0 "); events[nextID] = Event(msg.sender, \_name, \_price , \_date , \_ticketCount,  \_ticketCount, nextID);  nextID++ ;  }  // Creating modifier to make the code more clean  // -it will requires to check the existence of the event  // -it will require the date of the event to be after the current block time  modifier eventStatus(uint \_id){  require(events[\_id].date != 0, "Please enter a valid Event Details"); require(events[\_id].date > block.timestamp , "The event has already occured");  \_;  }  // BUYING EVENT TICKET (payable function to receive funds )  // -initialized and make a copy of the event in any other event id  // -it requires the msg.value to be equal to the event price \* the no of tickets  required  // -it requires to check whether the remaing tickets are more then required  tickets  // -we will substract the quantity requried from the remaing tickets.  // -we will add the number of tickets to perticular show by using nested mapping. |  |
| 43 | |

|  |  |
| --- | --- |
|  | |
| function buyTicket(uint \_id, uint \_quantity) external payable eventStatus(\_id) { Event storage \_event = events[\_id];  require(msg.value == \_event.price\*\_quantity , "Please enter a valid Amount"); require(\_event.ticketRemaning >= \_quantity, "Try for lower count or Tickets are  already sold out");  \_event.ticketRemaning -= \_quantity; Tickets[msg.sender][\_id] += \_quantity;  }  // TRANSFAR FUNCTION TO TRANSFER N NUMBER OF TICKTS TO X ADDRESS  // -it will then check if the number of tickets are greater then the number of tickets to send.(by using the nexted mapping)  // -we will then substract the number of tickets to be sent from ther address  // -we will add the substracted tickets to the new given address function transferTicket(uint \_id, uint \_quantity, address \_to) external  eventStatus(\_id){  require(Tickets[msg.sender][\_id] > \_quantity, "You dont have enough Tickets"); Tickets[msg.sender][\_id] -= \_quantity;  Tickets[\_to][\_id] += \_quantity;  }  // Remaing Ticket  // First we check if the event is there or Not  // We take event ID as input and the total tickets remaning  function RemaningTickets(uint \_id) view public eventStatus(\_id) returns(uint){ return events[\_id].ticketRemaning;  }  } |  |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy * **Create an Event** → Call createEvent("Music Concert", 1 ETH in Wei, Future Timestamp, 100), then confirm the transaction. * **Verify Event Details** → Check events(0) to view stored event details. * **Buy Tickets** → Call buyTicket(0, 2), send 2 \* event price in ETH, then confirm. * **Check Remaining Tickets** → Call RemaningTickets(0) to view available tickets. * **Transfer Tickets** → Call transferTicket(0, 1, <Receiver Address>), then confirm. * **Check User Tickets** → Query Tickets(YourAddress, 0) to see your tickets. * **Final Validation** → Test for invalid ETH, ticket limits, and expired events.   44 | |

# OUTPUT:





**RESULT:**

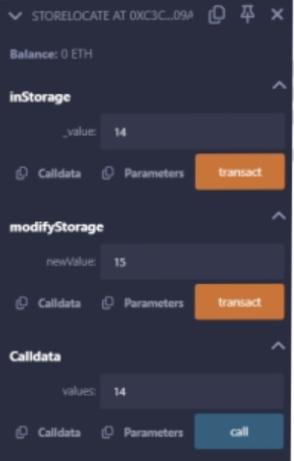
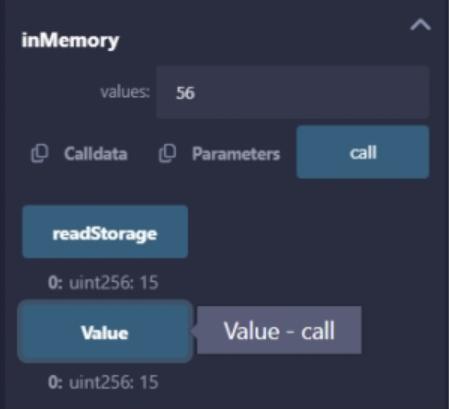
The Solidity smart contract for a banking system was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution of the createEvent() function, the contract correctly stored event details, including the organizer’s address, event name, ticket price, date, and available tickets. The buyTicket() function allowed users to purchase tickets by sending the required amount, updating the remaining ticket count, and emitting an event for transparency. The transferTicket() function enabled ticket transfers between users, ensuring secure ownership changes. The RemaningTickets() function accurately returned the updated number of available tickets. This experiment effectively demonstrated the use of Solidity events for logging transactions and enhancing event management transparency in smart contracts.

45

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 18 **DATE**:  **AIM:** Write a solidity program to demonstrate Illustration of storage locations  **DESCRIPTION:**  When working with Solidity, it's important to understand the differences between storage, [memory,](https://docs.alchemy.com/docs/what-is-the-difference-between-memory-and-calldata-in-solidity) [and calldata](https://docs.alchemy.com/docs/what-is-the-difference-between-memory-and-calldata-in-solidity). These are the three different types of data locations that can be used to store and manipulate data within a smart contract.  Memory is used to store temporary data that is needed during the execution of a function. Calldata is used to store function arguments that are passed in from an external caller. Storage is used to store data permanently on the blockchain.  function addNumbers(uint a, uint b) public pure returns (uint) {  uint c = a + b;  return c;  }  In this example, variables "a", "b", and "c" are all stored in memory because they are temporary variables that are only needed during the execution of the function.  Calldata is also a temporary data storage location, but it is used to hold function arguments that are passed in from an external caller, such as a user or another contract. Calldata is read-only and cannot be modified by the function.  function transfer(address recipient, uint amount) public {  // send tokens to the recipient  }  In this example, the "recipient" and "amount" variables are stored in calldata because they are function arguments passed in from an external caller.  **PROGRAM:** | |
| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract storeLocate { uint256 public Value;  // Example using memory  // Used for temporary storage within a function.  // Data stored in memory is not persisted on the blockchain.  // In the inMemory function, the values array is stored in memory.  // The function calculates the sum of the array elements and returns it.  // The array itself is discarded after the function completes.  function inMemory(uint256[] memory values) public pure returns (uint256) { uint256 sum = 0;  for (uint256 i = 0; i < values.length; i++) { sum = sum +values[i];  } |  |
| 46 | |

|  |  |  |
| --- | --- | --- |
|  | | |
| return sum; | |  |
| //memory array is not stored on blockchain, only sum is returned. | |
| } | |
| // Example using calldata  //Used for function arguments passed from external sources (e.g., transactions).  // calldata array is not stored on blockchain, only sum is returned.  // Similar to memory, calldata is also temporary and not persisted on the blockchain.  // calldata is read-only, meaning you cannot modify the data within it. This optimizes gas usage.  // In the Calldata function, the values array is passed as calldata.  // The function calculates the sum and returns it.  function Calldata(uint256[] calldata values) public pure returns (uint256) { uint256 sum = 0;  for (uint256 i = 0; i < values.length; i++) { sum += values[i];  }  return sum; | |
| } | |
| //Example using storage  function inStorage(uint256 \_value) public { Value = \_value;  } | |
| function readStorage() public view returns (uint256){ return Value;  } | |
| function modifyStorage(uint256 newValue) public{ Value = newValue;  } | |
| } | |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy | 47 |  |

# OUTPUT:

****

**RESULT:**

The Solidity smart contract for storage location handling was successfully compiled and deployed in the Remix Ethereum IDE. The inMemory() function correctly processed an array of values passed as memory, computed the sum, and returned the result without storing data on the blockchain. The Calldata() function efficiently handled an external array using calldata, ensuring read-only access and optimized gas usage while computing the sum. The inStorage() function successfully stored a value in blockchain storage, which was accurately retrieved using the readStorage() function. The modifyStorage() function successfully updated the stored value. This experiment effectively demonstrated the use of Solidity’s memory, calldata, and storage data locations for efficient state management in smart contracts.

# EXPERIMENT NO: 19 DATE:

**AIM:** Write a solidity program to demonstrate **CrowdFunding**

**DESCRIPTION:**

Crowdfunding smart contracts provide a secure, transparent, and efficient way to raise funds for projects. By leveraging blockchain technology, they eliminate the need for traditional intermediaries, fostering greater trust and accessibility.

Global and State Variables

1. owner -> owner of the contract
2. minimumContribution -> minimum contribution required for contributor to do crowd funding.
3. deadline -> Final Time to do crowdfunding.
4. target -> crowd funding target (How much funds is targeted to be completed in the deadline);
5. raisedAmount -> Keep track of the total raised amount.
6. totalContributors -> total contributors who have contributed to crowd funding.
7. numRequests -> keep track of requets done by owner.
8. contributors -> key value pair of contributor to keep track of their address and funds they have contributed.
9. allRequests -> All requests done by owner for tranfering money to a perticular address for a perticular reason.

**Functions** :-

1. getBalance
2. contribute
3. refund
4. createRequest
5. voteRequest
6. Make payment

**How It works :-**

1. Initially using constructor the owner addres, deadline, target, and minimum contribution to be done by contributor is saved in state variable.
2. The contributor contributes using contribute() function in crowd funding. If the deadline is not over and the funds are meeting the minimum requirements then the address and value is stored in state variable mapping.
3. If the target is not met and contributor wants the refund then he can refund the money by using refund() function.
4. The owner can request the all contributors to withdraw the funds by uing createRequest() function and tell the description, address to which the funds will be sent, and the total funds which to be sent.
5. All the contributors vote using voteRequest() function and if the votes are more than the 50% of contributors then using the makePayment() function the money is sent to the perticular address which owner has provided.

|  |  |
| --- | --- |
| **PROGRAM:** | |
| // SPDX-License-Identifier: MIT pragma solidity ^0.8.7; contract CrowdFunding {  address public owner; // owner of the contract  uint256 public minimumContribution; // minimum contribution required for contributor to do crofunding.  uint256 public numRequests;  uint256 public deadline; // Final Time to do crowdfunding. uint256 public target;  uint256 public raisedAmount; // Keep track of the total raised amount.  uint256 public totalContributors; // total contributors who have contributed to crowd  struct Request { string description;  address payable recipient; uint256 value;  bool completed; uint256 noOfVoters;  mapping(address => bool) voters;  }  mapping(uint256 => Request) public allRequests;  mapping(address => uint256) public contributors; // key value pair of contributors address and fund they have contributed.  constructor(uint256 \_target, uint256 \_deadline) { owner = msg.sender;  deadline = block.timestamp + \_deadline; target = \_target;  minimumContribution = 100000 wei;  }  // All Modifiers starts here modifier onlyOwner() {  require(msg.sender == owner, "Only owner can access this function.");  \_;  }  modifier isDeadlinePassed() { require(  block.timestamp < deadline,"Crowd Funding deadline has passed. Please try again later.");  \_;  } |  |
| 51 | |

|  |  |
| --- | --- |
|  | |
| modifier isContributor() {  require(contributors[msg.sender] > 0, "Sorry, you are not a contributor. Try to contribute to crowd funding then try again. Thanks.");  \_;  }  // All Modifiers Ends here  // get balance  function getBalance() public view returns (uint256) { return address(this).balance;  }  // Contribute  function contribute() public payable isDeadlinePassed { require(  msg.value >= minimumContribution,  "Minimum 100 wei is required to contribute."  );  if (contributors[msg.sender] == 0) { totalContributors++;  }  contributors[msg.sender] += msg.value; raisedAmount += msg.value;  }  // Refund the money if the target is not fulfilled and deadline has passed. function refund() public isDeadlinePassed isContributor {  require(raisedAmount < target, "You are not eligible for refund"); require(contributors[msg.sender] > 0);  address payable user = payable(msg.sender); user.transfer(contributors[msg.sender]);  }  // Create Request function createRequest(  string memory \_description, address payable \_recipient, uint256 \_value  ) public onlyOwner {  Request storage newRequest = allRequests[numRequests]; numRequests++;  newRequest.description = \_description; newRequest.recipient = \_recipient; newRequest.value = \_value; newRequest.completed = false; newRequest.noOfVoters = 0;  }  // vote request.  function voteRequest(uint256 \_requestNo) public isContributor { |  |
| 52 | |

|  |  |  |
| --- | --- | --- |
|  | | |
| Request storage currentRequest = allRequests[\_requestNo]; require(currentRequest.voters[msg.sender] == false, "you have already voted"); currentRequest.voters[msg.sender] = true;  currentRequest.noOfVoters++;  }  // make payemnt only if the voters are greater than half of the contributors. function makePayment(uint \_requestNo) public onlyOwner{  require(raisedAmount >= target);  Request storage currentRequest = allRequests[\_requestNo]; require(currentRequest.completed == false, "This request has already completed"); require(currentRequest.noOfVoters > totalContributors/2, "Contributor does not  support. Votes are low.");  currentRequest.recipient.transfer(currentRequest.value);  }  } | |  |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy   **OUTPUT:** | 53 |  |

RESULT:

The Solidity smart contract for crowdfunding was successfully compiled and deployed in the Remix Ethereum IDE. The contribute() function correctly accepted contributions above the minimum threshold, updated the total raised amount, and tracked contributors. The refund() function successfully processed refunds if the target was not met and the deadline had passed. The createRequest() function allowed the contract owner to generate fund withdrawal requests, while the voteRequest() function enabled contributors to vote on fund release proposals. The makePayment() function successfully transferred funds to the recipient if more than half of the contributors approved the request. This experiment demonstrated effective implementation of decentralized crowdfunding with voting-based fund disbursement.

54

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 20 **DATE**:  **AIM:** Write a solidity program to demonstrate Usage of Fallback functions  **DESCRIPTION:**  In Solidity, the fallback() function is a special function that is executed under certain conditions when a contract receives a call. This function gets called when the contract has no desired function say for example flow().  **PROGRAM:** | |
| // SPDX-License-Identifier: MIT pragma solidity >=0.7.1 <0.9.0; contract Fallback{  event Log(string func, address sender, uint value, bytes data); fallback() external payable{  emit Log("fallback", msg.sender, msg.value, msg.data);  }  receive() external payable{  emit Log("receive", msg.sender, msg.value, "");  }  } |  |
| * Press Ctrl+S to Save and Compile. * If no errors occurred proceed to Deploy. * **Deploy:** Use Remix to compile and deploy   **OUTPUT:**      **RESULT:**  The Solidity smart contract for handling fallback and receive functions was successfully compiled and deployed in the Remix Ethereum IDE. Upon sending Ether to the contract:  55 | |

* If the transaction included data, the fallback() function was triggered, emitting the **Log** event with the function name "fallback," sender address, value sent, and calldata.
* If the transaction did not include data, the receive() function was executed, emitting the **Log**

event with the function name "receive," sender address, and value sent (with an empty data field).

This experiment demonstrated the correct handling of unexpected function calls and Ether transfers using the fallback() and receive() functions in Solidity.

56

|  |  |
| --- | --- |
| **EXPERIMENT NO:** 21 **DATE**:  **AIM:** Write a solidity program to demonstrate Call Other contracts  **DESCRIPTION:**  This Solidity program illustrates how one smart contract can call functions from another contract using external contract interactions. This is particularly useful in scenarios like **decentralized finance (DeFi), multi-contract systems, and cross-contract communication**.  *Key Concepts Demonstrated:*   1. **Contract Interaction:** The primary contract interacts with another contract by calling its functions. 2. **Calling External Functions:** The contract uses an interface or direct contract address reference to invoke functions from another deployed contract. 3. **State Updates Across Contracts:** It shows how one contract can modify or retrieve the state of another contract.   *Implementation Overview:*   * + **Contract A:** A simple contract that stores and retrieves a number.   + **Contract B:** A contract that interacts with Contract A by calling its functions.   *Execution Flow:*   1. Deploy **Contract A** first. 2. Deploy **Contract B** with the address of Contract A. 3. Call the functions in Contract B, which in turn interact with Contract A.   *Expected Outcome:*   * + Contract B successfully calls functions of Contract A and retrieves/updates data.   + Solidity’s call method ensures secure execution.   + The program demonstrates smart contract modularity and interoperability.   PROGRAM: | |
| // SPDX-License-Identifier: MIT pragma solidity >=0.7.1 <0.9.0;  contract testContract{ uint public x;  uint public value= 89;  function setX( uint \_x)external { x=\_x;  }  function getX( )external view returns(uint){ return x;  }  function setXandreceiveether( uint \_x)external payable{ x=\_x;  value= msg.value;  } |  |
| 57 | |

|  |  |  |
| --- | --- | --- |
|  | | |
| function getXandgetvalue( )external view returns(uint,uint) { return (x,value); | |  |
| } | |
| }  contract calltestContract{ | |
| /\* function setX(address \_test, uint \_x) external{ testcontract(\_test).setX(-x);  }\*/  function setX(testContract \_test, uint \_x) external{  \_test.setX(\_x);  }  function getX(testContract \_test) external view returns(uint){ uint x = \_test.getX();  return x;  } | |
| } | |
| OUTPUT: | 58 |  |



**RESULT:**

The Solidity smart contract for calling functions from another contract was successfully compiled and deployed in the Remix Ethereum IDE. Upon execution:

* The setX() function in testContract was called through calltestContract, correctly updating the value of x in testContract.
* The getX() function in calltestContract successfully retrieved the updated value of x from

testContract.

* The setXandreceiveether() function in testContract updated x while also storing the received Ether amount.
* The getXandgetvalue() function accurately returned both x and the stored Ether value.

This experiment successfully demonstrated contract-to-contract interactions in Solidity, showcasing how one contract can call and modify another contract’s state.

59