**Practical No. 03**

**Solidity Programming**

**Q. 1 Write a solidity smart contract to display hello world message.**

**Code :**

// Hello World Smart Contract

// SPDX-License-Identifier: MIT

pragma solidity >= 0.5.0 < 0.8.27;

contract HelloWorld {

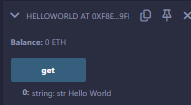
    function get()public pure returns (string memory str) {

        str = "Hello World";

    }

}

Output :





Q. 2 Write a solidity smart contract to demonstrate state variable, local variable and global variable.

Code:

// SPDX-License-Identifier: MIT

pragma solidity >= 0.5.0 < 0.8.27;

contract DemonstrateFunction{

    uint16 num = 10; // state variable

    function getInput3(uint16 n1, uint16 n2) public returns (uint16){

        num = 100;

        return n1 + n2 + num;

    }

    function getInput2(uint16 n1, uint16 n2) public view returns (uint16){

        return n1 + n2 + num;

    }

    function getInput(uint16 n1, uint16 n2) public pure returns (uint16){

        return n1 + n2;

    }

    function getInfo() public returns (uint16){

        return num += 11;

    }

    function getData() public view returns (uint16){

        return num;

    }

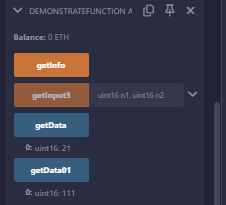
    function getData01() public pure returns (uint16){

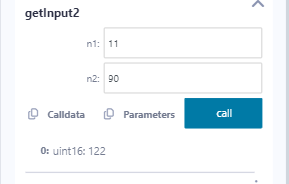
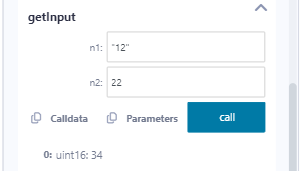
        return 111;

    }

}

Output:





**Q.3 Write a solidity smart contract to demonstrate getter and setter methods.**

Code :

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract StructureDemo1{

    // structure like C program

    struct Book{

        string title;

        string author;

        uint book\_id;

    }

    // variable object created

    Book b1;

    // setter method

    function setBook() public {

         b1=Book('Hello JS', 'Onkar', 101);

     }

    // add values to it

    function enterBkDtls(string memory title, string memory auth, uint bid) public {

        b1=Book(title, auth, bid);

    }

    // Getter method

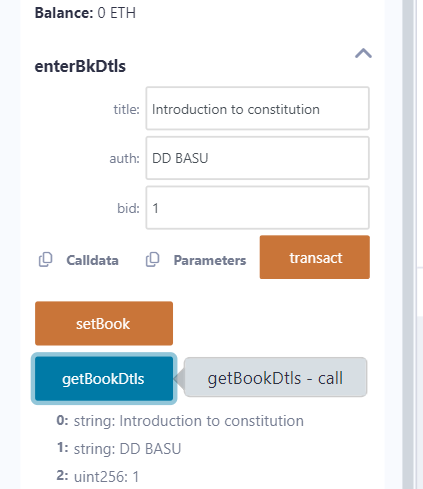
    function getBookDtls() public view returns(string memory,string memory,uint) {

        return (b1.title,b1.author,b1.book\_id);

    }

}

Output :



**Q.4 Write a solidity smart contract to demonstrate function modifier.**

Code:

// SPDX-License-Identifier: MIT

pragma solidity >= 0.5.0 < 0.8.27;

contract ModifierDemo {

    // We will use the modifier to limit the function changePrice to only the owner of the contract

    address public owner;

    uint price;

    constructor() {

        owner = msg.sender;

    }

    modifier onlyOwner{

        require(msg.sender == owner, 'Only Owner is allowed to modify the price!');

        \_;   // Asterisk is used to indicate that this function will be executed even if there is an exception.

        // This will allow us to do any other modification.

    }

    function changePrice(uint \_price) public onlyOwner{

        price=\_price;

    }

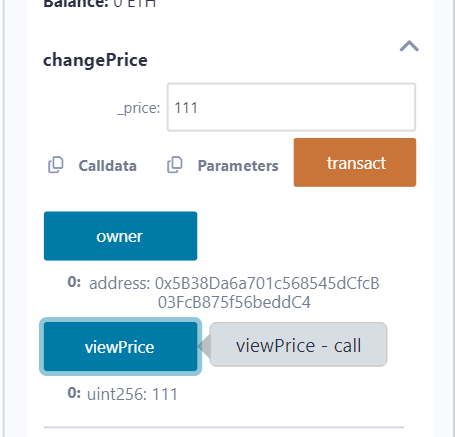
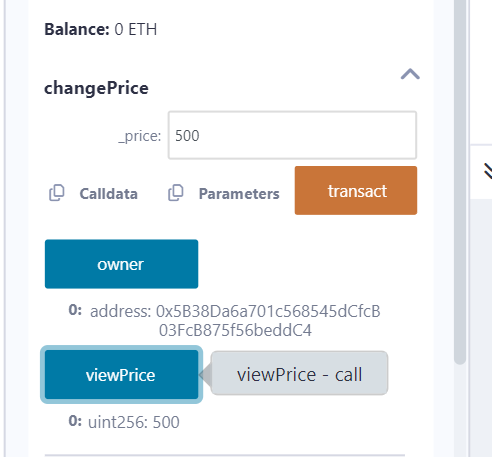
    function viewPrice() public view returns (uint){

        return price;

    }

}

Output:



**Q.5 Write a Solidity program to demonstrate arrays Push operation and Pop operation.**

Code:

// Push pop Array

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract PushPop {

    uint[] data = [10, 20, 30, 40, 50];

    function array\_push() public returns(uint[] memory) {

        data.push(60);

        data.push(70);

        data.push(80);

        return data;

    }

    function array\_pop() public returns (uint[] memory){

        data.pop();

        return data;

    }

}

Output:

Push Button Click:



Pop Button Click:



**Q.6 Write a Solidity program to demonstrate creating a fixed-size array and access array element.**

Code:

// Demonstrate Array

// SPDX-License-Identifier: MIT

pragma solidity >= 0.5.0 < 0.8.27;

// create a Contract

contract FixedSizeArray {

    // Declaring state variable of Array

    uint[6] data1;

    //Defining the functions

    function array\_example() public returns (int[5] memory, uint[6] memory){

        int[5] memory data = [int(50), -63, 77, -28, 90];

        // local variable type

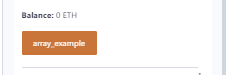
        data1 = [uint(10), 20, 30, 40, 50, 60];

        return (data, data1); // Returning the values

    }

}

Output:





**Q.7 Write a Solidity program to demonstrate creating a dynamic array and accessing array element.**

Code:

// Dynamic Smart Contract

// SPDX-License-Identifier: MIT

pragma solidity >= 0.5.0 < 0.8.27;

contract Types{

    // dynamic array

    int[] data1;

    // static array

    uint[] data = [10, 20, 30, 40, 50, 60];

    function dynamic\_array() public returns (uint[] memory, int[] memory) {

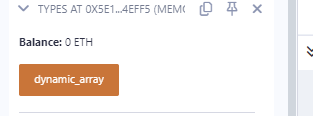
        data1 = [int(-60), 70, 120, -120];

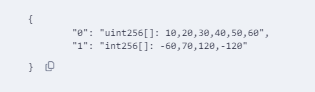
        return (data, data1);

    }

}

Output:





**Q.8 Write a solidity smart contract to demonstrate use of structure.**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract StructureDemo1{

    // structure like C program

    struct Book{

        string title;

        string author;

        uint book\_id;

    }

    // variable object created

    Book b1;

    // setter method

    function setBook() public {

         b1=Book('Hello JS', 'Onkar', 101);

     }

    // add values to it

    function enterBkDtls(string memory title, string memory auth, uint bid) public {

        b1=Book(title, auth, bid);

    }

    // Getter method

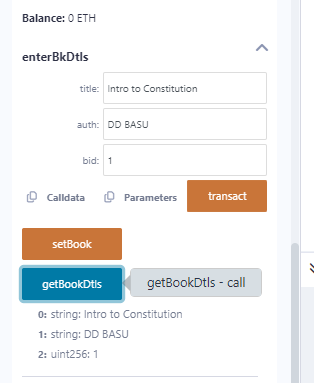
    function getBookDtls() public view returns(string memory,string memory,uint) {

        return (b1.title,b1.author,b1.book\_id);

    }

}

Output :



**Q.9 Write a solidity smart contract to calculate percentage of marks obtained by students for six subjects in final examination.**

**Code:**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract StudentMarks {

    struct Student {

        uint256[6] marks; // Array to hold marks for six subjects

    }

    mapping(address => Student) private students;

    // Function to set marks for a student

    function setMarks(uint256[6] memory \_marks) public {

        require(\_marks.length == 6, "Must provide marks for six subjects");

        students[msg.sender].marks = \_marks;

    }

    // Function to calculate the percentage of marks obtained

    function calculatePercentage() public view returns (uint256) {

        Student storage student = students[msg.sender];

        uint256 totalMarks = 0;

        uint256 maxMarks = 600; // Assuming each subject is out of 100

        for (uint256 i = 0; i < 6; i++) {

            totalMarks += student.marks[i];

        }

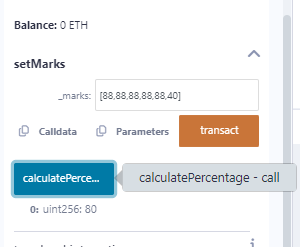
        // Calculate percentage

        return (totalMarks \* 100) / maxMarks;

    }

}

Output:



**Q.10 Write a solidity smart contract to find the factorial of entered number.**

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract FactorialCalculator {

    // Function to calculate factorial of a number

    function factorial(uint16 n) public pure returns (uint16) {

        require(n >= 0, "Input must be a non-negative integer");

        if (n == 0) {

            return 1; // Base case: 0! = 1

        } else {

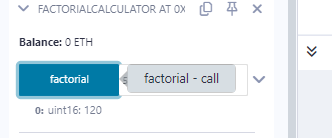
            return n \* factorial(n - 1); // Recursive case

        }

    }

}

Output:



**Q.11 Write a solidity smart contract to check whether entered number is palindrome or not.**

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract PalindromeChecker {

    // Function to check if a number is a palindrome

    function isPalindrome(uint256 n) public pure returns (bool) {

        uint256 reversed = reverse(n);

        return n == reversed;

    }

    // Function to reverse a number

    function reverse(uint256 n) internal pure returns (uint256) {

        uint256 reversed = 0;

        while (n > 0) {

            reversed = reversed \* 10 + n % 10;

            n /= 10;

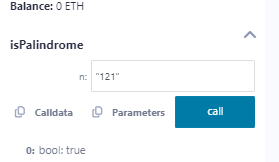
        }

        return reversed;

    }

}

Output:



Q.12 Write a solidity smart contract to generate Fibonacci Series up to given number.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract FibonacciGenerator {

    // Function to generate Fibonacci series up to a given number

    function generateFibonacci(uint256 n) public pure returns (uint256[] memory) {

        require(n > 0, "Input must be a positive integer");

        uint256[] memory fibSequence = new uint256[](n);

        fibSequence[0] = 0;

        if (n == 1) {

            return fibSequence;

        }

        fibSequence[1] = 1;

        for (uint256 i = 2; i < n; i++) {

            fibSequence[i] = fibSequence[i - 1] + fibSequence[i - 2];

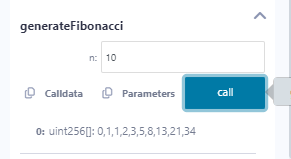
        }

        return fibSequence;

    }

}

Output:



Q.13 Write a solidity smart contract to check whether entered number is prime number or not.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract PrimeChecker {

    // Function to check if a number is prime

    function isPrime(uint256 n) public pure returns (bool) {

        require(n > 0, "Input must be a positive integer");

        if (n == 1) {

            return false; // 1 is not a prime number

        }

        for (uint256 i = 2; i \* i <= n; i++) {

            if (n % i == 0) {

                return false; // n is divisible by i, so it's not prime

            }

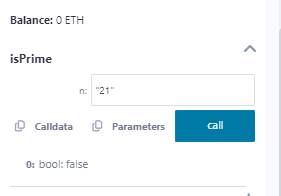
        }

        return true; // n is prime

    }

}

Output:



Q.14 Write a solidity smart contract to create arithmetic calculator which includes functions for operations addition, subtraction, multiplication, division etc.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract ArithmeticCalculator {

    // Function to add two numbers

    function add(uint256 a, uint256 b) public pure returns (uint256) {

        return a + b;

    }

    // Function to subtract two numbers

    function subtract(uint256 a, uint256 b) public pure returns (uint256) {

        require(b <= a, "Subtraction result would be negative");

        return a - b;

    }

    // Function to multiply two numbers

    function multiply(uint256 a, uint256 b) public pure returns (uint256) {

        return a \* b;

    }

    // Function to divide two numbers

    function divide(uint256 a, uint256 b) public pure returns (uint256) {

        require(b > 0, "Division by zero is not allowed");

        return a / b;

    }

    // Function to calculate modulus of two numbers

    function modulus(uint256 a, uint256 b) public pure returns (uint256) {

        require(b > 0, "Modulus by zero is not allowed");

        return a % b;

    }

    // Function to calculate exponentiation of two numbers

    function exponentiation(uint256 base, uint256 exponent) public pure returns (uint256) {

        uint256 result = 1;

        for (uint256 i = 0; i < exponent; i++) {

            result \*= base;

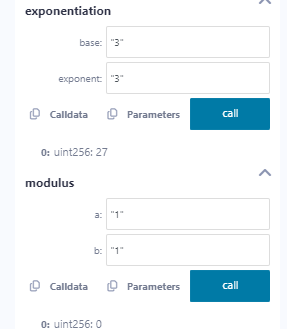
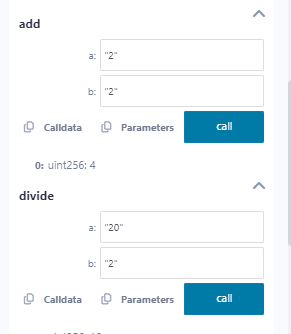
        }

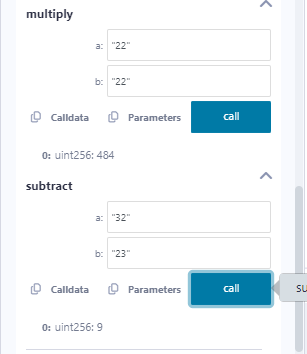
        return result;

    }

}

Output:





Q.15 Write a solidity smart contract to demonstrate view function and pure function.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract ViewAndPureFunctions {

    // State variable to store a number

    uint256 private storedNumber;

    // Constructor to initialize the stored number

    constructor(uint256 initialNumber) {

        storedNumber = initialNumber;

    }

    // View function to get the stored number

    function getStoredNumber() public view returns (uint256) {

        return storedNumber;

    }

    // Pure function to add two numbers

    function add(uint256 a, uint256 b) public pure returns (uint256) {

        return a + b;

    }

    // Pure function to multiply two numbers

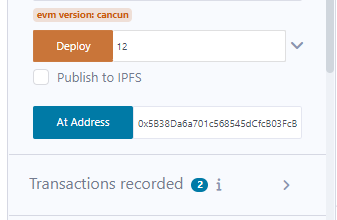
    function multiply(uint256 a, uint256 b) public pure returns (uint256) {

        return a \* b;

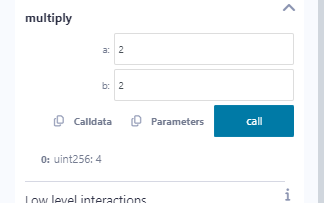
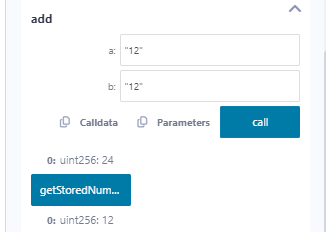
    }

}

Output:







Q.16 Write a solidity smart contract to demonstrate inbuilt mathematical functions.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract MathematicalFunctions {

// Function to demonstrate addition

function add(uint256 a, uint256 b) public pure returns (uint256) {

return a + b;

}

// Function to demonstrate subtraction

function subtract(uint256 a, uint256 b) public pure returns (uint256) {

return a - b;

}

// Function to demonstrate multiplication

function multiply(uint256 a, uint256 b) public pure returns (uint256) {

return a \* b;

}

// Function to demonstrate division

function divide(uint256 a, uint256 b) public pure returns (uint256) {

require(b > 0, "Division by zero is not allowed");

return a / b;

}

// Function to demonstrate modulus

function modulus(uint256 a, uint256 b) public pure returns (uint256) {

require(b > 0, "Modulus by zero is not allowed");

return a % b;

}

// Function to demonstrate exponentiation

function exponentiation(uint256 base, uint256 exponent) public pure returns (uint256) {

uint256 result = 1;

for (uint256 i = 0; i < exponent; i++) {

result \*= base;

}

return result;

}

// Function to demonstrate absolute value

function absoluteValue(int256 a) public pure returns (uint256) {

return a >= 0 ? uint256(a) : uint256(-a);

}

// Function to demonstrate minimum value

function minimum(uint256 a, uint256 b) public pure returns (uint256) {

return a < b ? a : b;

}

// Function to demonstrate maximum value

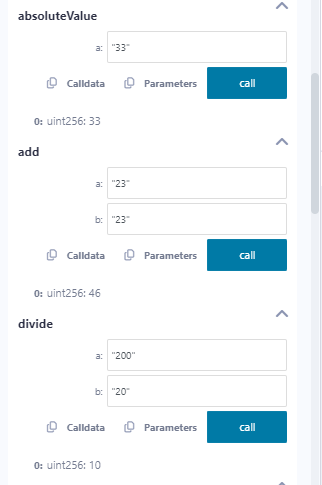
function maximum(uint256 a, uint256 b) public pure returns (uint256) {

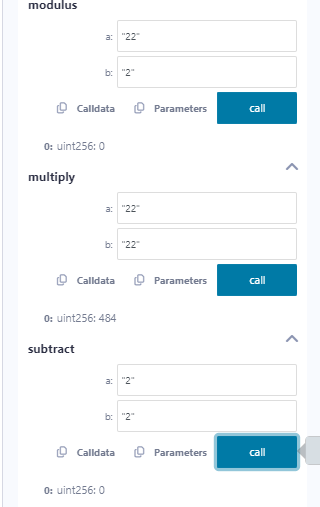
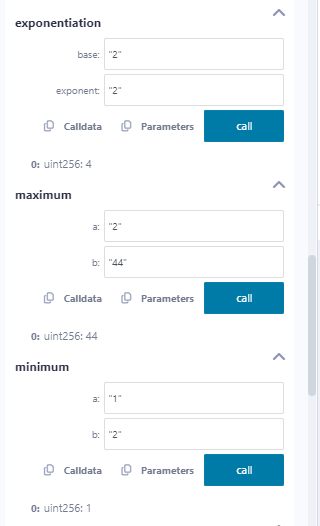
return a > b ? a : b;

}

}

Output

:



Q.17 Write a solidity smart contract to demonstrate inheritance in contract.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

// Base contract

contract Animal {

// State variable to store the name of the animal

string private name;

// Constructor to initialize the name of the animal

constructor(string memory \_name) {

name = \_name;

}

// Function to get the name of the animal

function getName() public view returns (string memory) {

return name;

}

// Function to make a sound

function makeSound() public virtual returns (string memory) {

return "The animal makes a sound";

}

}

// Derived contract that inherits from Animal

contract Dog is Animal {

// Constructor to initialize the name of the dog

constructor(string memory \_name) Animal(\_name) {}

// Override the makeSound function to make a dog sound

function makeSound() public override returns (string memory) {

return "The dog barks";

}

// Function to wag the tail

function wagTail() public pure returns (string memory) {

return "The dog wags its tail";

}

}

// Derived contract that inherits from Animal

contract Cat is Animal {

// Constructor to initialize the name of the cat

constructor(string memory \_name) Animal(\_name) {}

// Override the makeSound function to make a cat sound

function makeSound() public override returns (string memory) {

return "The cat meows";

}

// Function to scratch

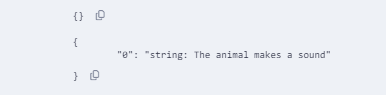
function scratch() public pure returns (string memory) {

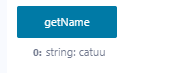
return "The cat scratches";

}

}

Output:





Q.18 Write a solidity smart contract to demonstrate events.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract EventDemo{

    uint256 public data = 0;

    event Increment(address owner);

    function getValue(uint \_a, uint \_b) public returns (uint256){

        emit Increment(msg.sender);

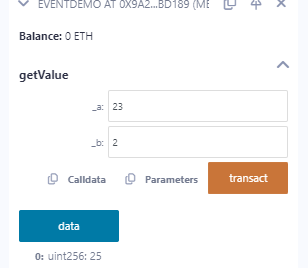
        data = \_a + \_b;

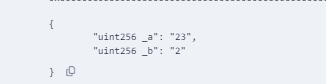
        return data;

    }

}

Output:





Q.19 Write a solidity smart contract to demonstrate assert statement and revert statement.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract AssertDemo {

    uint public count;

    function increment() public returns (uint) {

        count += 1;

        return count;

    }

    function decrement() public returns (uint) {

        require(count > 0, "Count must be greater than zero");

        count -= 1;

        return count;

    }

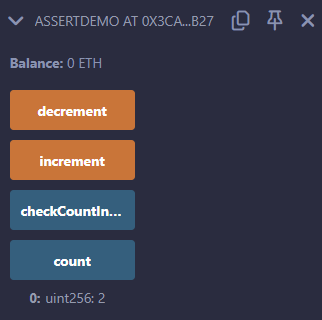
    function checkCountInVarient() public view {

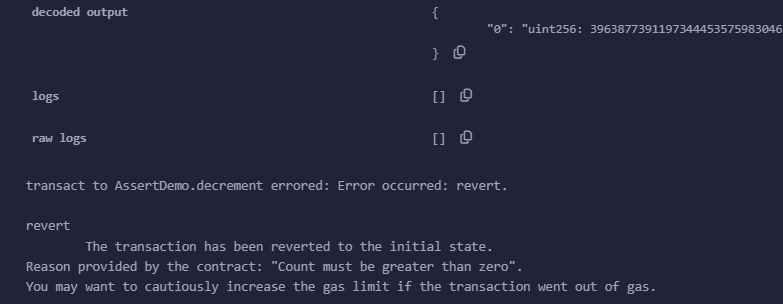
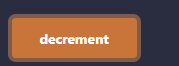
        assert(count >= 1);

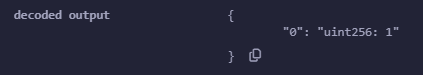
    }

}

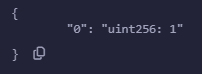
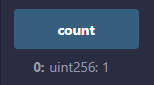
Output:











Q.20 Write a solidity smart contract for Bank Account which provides operations such as check account balance, withdraw amount and deposit amount etc.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract SimpleBank {

    // State variable to store the balance

    uint256 private balance;

    // Constructor to initialize balance

    constructor() {

        balance = 0;

    }

    // Function to add (deposit) amount to the balance

    function addAmount(uint256 amount) public {

        balance += amount;

    }

    // Function to withdraw amount from the balance

    function withdrawAmount(uint256 amount) public {

        require(amount <= balance, "Insufficient balance");

        balance -= amount;

    }

    // Function to check the remaining balance

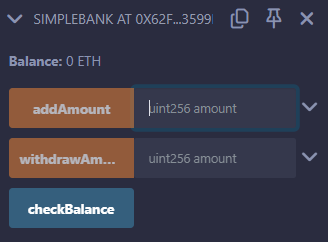
    function checkBalance() public view returns (uint256) {

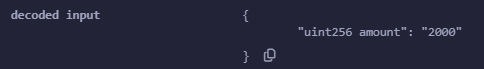
        return balance;

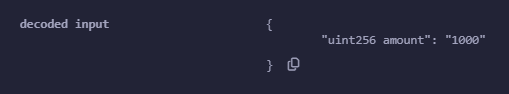
    }

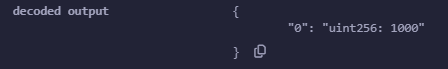
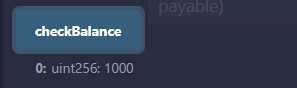
}

Output:









Q.21 Write a program in solidity to create a structured student with Roll no, Name,Class, Department, Course enrolled as variables.

I) Add information of 5 students.

ii) Search for a student using Roll no

iii) Display all information

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract StudentInformation {

    // Struct to represent a student

    struct Student {

        uint256 rollNo;

        string name;

        string class\_;

        string department;

        string courseEnrolled;

    }

    // Mapping to store students by roll number

    mapping(uint256 => Student) private students;

    // Function to add a new student

    function addStudent(uint256 rollNo, string memory name, string memory class\_, string memory department, string memory courseEnrolled) public {

        require(rollNo > 0, "Roll number must be greater than zero");

        require(bytes(name).length > 0, "Name cannot be empty");

        require(bytes(class\_).length > 0, "Class cannot be empty");

        require(bytes(department).length > 0, "Department cannot be empty");

        require(bytes(courseEnrolled).length > 0, "Course enrolled cannot be empty");

        students[rollNo] = Student(rollNo, name, class\_, department, courseEnrolled);

    }

    // Function to search for a student by roll number

    function searchStudent(uint256 rollNo) public view returns (Student memory) {

        require(students[rollNo].rollNo > 0, "Student not found");

        return students[rollNo];

    }

    // Function to display all student information

    function displayAllStudents() public view returns (Student[] memory) {

        uint256 count = 0;

        for (uint256 i = 1; i <= 100; i++) {

            if (students[i].rollNo > 0) {

                count++;

            }

        }

        Student[] memory allStudents = new Student[](count);

        uint256 index = 0;

        for (uint256 i = 1; i <= 100; i++) {

            if (students[i].rollNo > 0) {

                allStudents[index] = students[i];

                index++;

            }

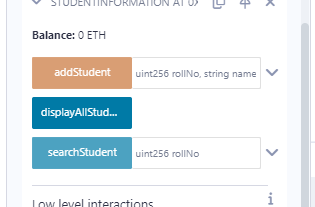
        }

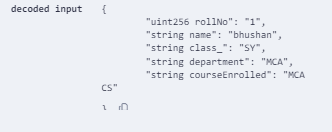
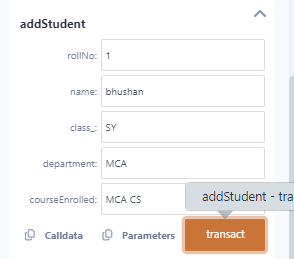
        return allStudents;

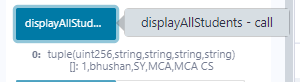
    }

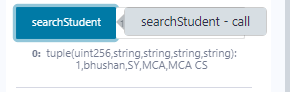
}

Output:









Q.22 Create a structure Consumer with Name, Address, Consumer ID, Units and Amount as members. Write a program in solidity to calculate the total electricity bill according to the given condition:

For first 50 units Rs. 0.50/unit. For next 100 units Rs. 0.75/unit. For next 100 units Rs. 1.20/unit. For unit above 250 Rs. 50/unit. An additional surcharge of 20% is added to the bill. Display the information of 5 such consumers along with their units consumed and amount.

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract ElectricityBillCalculator {

    // Struct to represent a consumer

    struct Consumer {

        string name;

        string address\_;

        uint256 consumerID;

        uint256 units;

        uint256 amount;

    }

    // Mapping to store consumers by consumer ID

    mapping(uint256 => Consumer) private consumers;

    // Function to add a new consumer

    function addConsumer(uint256 consumerID, string memory name, string memory address\_, uint256 units) public {

        require(consumerID > 0, "Consumer ID must be greater than zero");

        require(bytes(name).length > 0, "Name cannot be empty");

        require(bytes(address\_).length > 0, "Address cannot be empty");

        require(units >= 0, "Units cannot be negative");

        uint256 amount = calculateBill(units);

        consumers[consumerID] = Consumer(name, address\_, consumerID, units, amount);

    }

    // Function to calculate the total electricity bill

    function calculateBill(uint256 units) internal pure returns (uint256) {

        uint256 amount;

        if (units <= 50) {

            amount = units \* 50; // Rs. 0.50/unit

        } else if (units <= 150) {

            amount = 50 \* 50 + (units - 50) \* 75;

// Rs. 0.50/unit for first 50 units, Rs. 0.75/unit for next 100 units

        } else if (units <= 250) {

            amount = 50 \* 50 + 100 \* 75 + (units - 150) \* 120;

// Rs. 0.50/unit for first 50 units, Rs. 0.75/unit for next 100 units, Rs. 1.20/unit for next 100 units

        } else {

            amount = 50 \* 50 + 100 \* 75 + 100 \* 120 + (units - 250) \* 150;

// Rs. 0.50/unit for first 50 units, Rs. 0.75/unit for next 100 units, Rs. 1.20/unit for next 100 units, Rs. 1.50/unit for units above 250

        }

        // Additional surcharge of 20%

        amount = amount + (amount \* 20 / 100);

        return amount;

    }

    // Function to display the information of a consumer

    function displayConsumer(uint256 consumerID) public view returns (Consumer memory) {

        require(consumers[consumerID].consumerID > 0, "Consumer not found");

        return consumers[consumerID];

    }

    // Function to display the information of all consumers

    function displayAllConsumers() public view returns (Consumer[] memory) {

        uint256 count = 0;

        for (uint256 i = 1; i <= 100; i++) {

            if (consumers[i].consumerID > 0) {

                count++;

            }

        }

        Consumer[] memory allConsumers = new Consumer[](count);

        uint256 index = 0;

        for (uint256 i = 1; i <= 100; i++) {

            if (consumers[i].consumerID > 0) {

                allConsumers[index] = consumers[i];

                index++;

            }

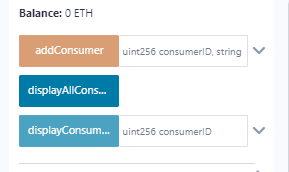
        }

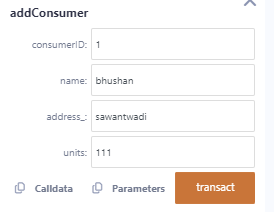
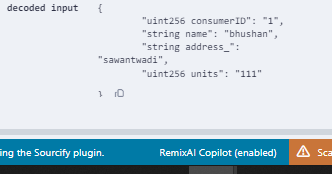
        return allConsumers;

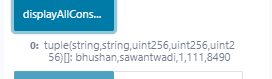
    }

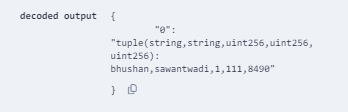
}

Output:





Exercise :

Solidity program to demonstrate Type Conversion : Implicit conversion

**Program :**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract ImplicitConversionDemo {

// Function to demonstrate implicit conversion

function implicitConversion() public pure returns (uint, uint) {

// Declare a uint variable

uint myUint = 10;

// Implicitly convert uint to int

int myInt = int(myUint); // Explicit conversion to int for clarity

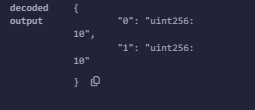
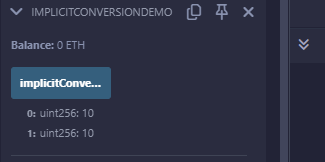
// Convert back to uint for return

return (myUint, uint(myInt));

}

}

**Output :**



Solidity program to demonstrate Type Conversion : Explicit conversion   
  
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;  
  
contract ExplicitConversionDemo {

// Function to demonstrate explicit conversion

function explicitConversion() public pure returns (uint, int) {

// Declare an int variable

int myInt = -10;

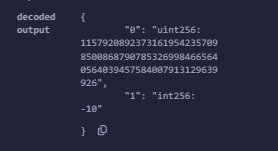
// Explicitly convert int to uint (this will revert if myInt is negative)

uint myUint = uint(myInt); // Unsafe conversion; will revert if myInt is negative

return (myUint, myInt);

}

}



Solidity program to demonstrate do while loop

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DoWhileLoopDemo {

// Declare a state variable to hold the sum

uint public sum = 0;

// Function to demonstrate do-while loop

function calculateSum(uint limit) public {

uint i = 1; // Initialize loop counter

// Use do-while loop to sum numbers from 1 to the given limit

do {

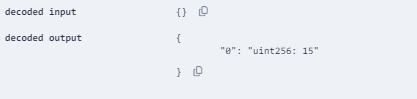
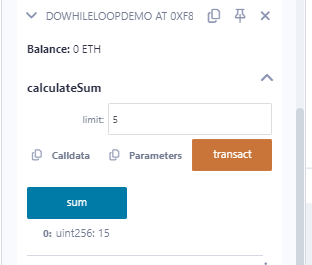
sum += i; // Add current value of i to the sum

i++; // Increment the counter

} while (i <= limit); // Loop will continue until i exceeds the limit

}

}



Solidity program to demonstrate while loop

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract WhileLoopDemo {

// Declare a state variable to hold the sum

uint public sum = 0;

// Function to demonstrate while loop

function calculateSum(uint limit) public {

uint i = 1; // Initialize loop counter

// Use while loop to sum numbers from 1 to the given limit

while (i <= limit) {

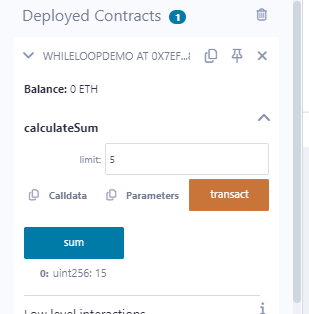
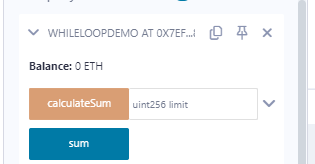
sum += i; // Add current value of i to the sum

i++; // Increment the counter

}

}

}





Solidity program to demonstrate for loop

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract ForLoopDemo {

// Declare a state variable to hold the sum

uint public sum = 0;

// Function to demonstrate for loop

function calculateSum(uint limit) public {

sum = 0; // Reset the sum before calculation

// Use for loop to sum numbers from 1 to the given limit

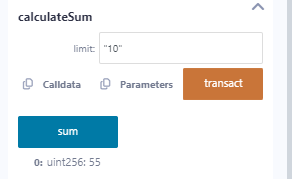
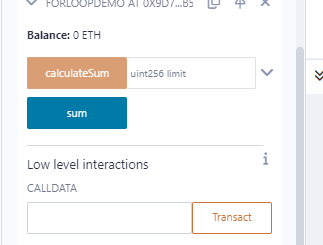
for (uint i = 1; i <= limit; i++) {

sum += i; // Add current value of i to the sum

}

}

}



Solidity program to demonstrate if else statement

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract IfElseDemo {

// Declare a state variable to hold a result

string public result;

// Function to demonstrate if-else statement

function checkNumber(int num) public {

if (num > 0) {

result = "The number is positive";

} else if (num == 0) {

result = "The number is zero";

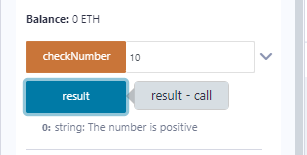
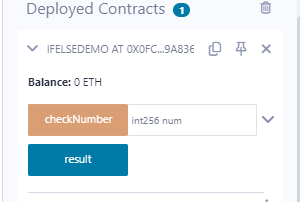
} else {

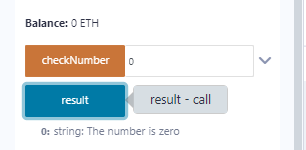
result = "The number is negative";

}

}

}





Solidity program to demonstrate **else if**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract ElseIfDemo {

// Declare a state variable to store the result

string public grade;

// Function to determine the grade based on marks

function checkGrade(uint marks) public {

if (marks >= 90) {

grade = "A+";

} else if (marks >= 80) {

grade = "A";

} else if (marks >= 70) {

grade = "B";

} else if (marks >= 60) {

grade = "C";

} else if (marks >= 50) {

grade = "D";

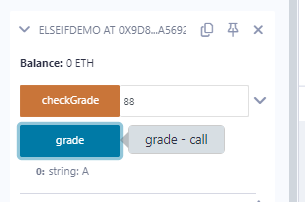
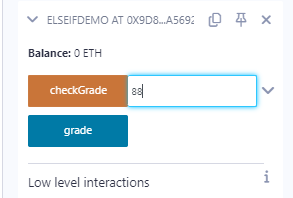
} else {

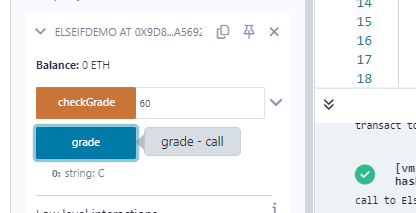
grade = "F";

}

}

}





Solidity program to demonstrate **jumping statements**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract BreakContinueDemo {

// Declare a state variable to hold the sum

uint public sum = 0;

// Function to demonstrate break and continue in for loop

function calculateSum(uint limit) public {

sum = 0; // Reset the sum before calculation

for (uint i = 1; i <= limit; i++) {

if (i == 3) {

continue; // Skip the number 3 (do not add it to sum)

}

if (i == 7) {

break; // Stop the loop when i reaches 7

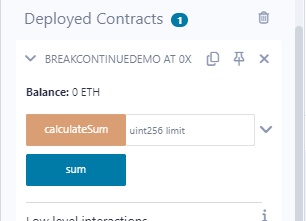
}

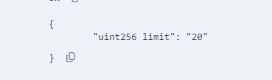
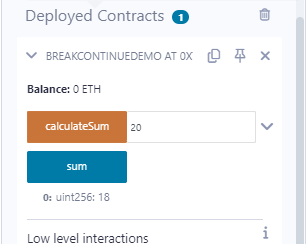
sum += i; // Add current value of i to the sum

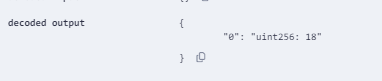
}

}

}







Solidity program to demonstrate operators

Code :

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract OperatorsDemo {

uint public a = 10;

uint public b = 5;

bool public result;

uint public bitwiseResult;

// Function to demonstrate Arithmetic operators

function arithmeticOperations() public view returns (uint, uint, uint, uint, uint) {

uint sum = a + b; // Addition

uint diff = a - b; // Subtraction

uint prod = a \* b; // Multiplication

uint div = a / b; // Division

uint mod = a % b; // Modulus

return (sum, diff, prod, div, mod);

}

// Function to demonstrate Assignment operators

function assignmentOperations() public returns (uint) {

uint c = a; // Assign value of a to c

c += b; // c = c + b

c -= 2; // c = c - 2

c \*= 2; // c = c \* 2

c /= 2; // c = c / 2

c %= 3; // c = c % 3

return c;

}

// Function to demonstrate Logical operators

function logicalOperations() public view returns (bool, bool, bool) {

bool andOp = (a > b) && (b > 3); // Logical AND

bool orOp = (a > b) || (b < 2); // Logical OR

bool notOp = !(a == b); // Logical NOT

return (andOp, orOp, notOp);

}

// Function to demonstrate Conditional (Ternary) operator

function conditionalOperation() public view returns (string memory) {

return (a > b) ? "a is greater than b" : "a is not greater than b";

}

// Function to demonstrate Bitwise operators

function bitwiseOperations() public returns (uint) {

bitwiseResult = a & b; // Bitwise AND

bitwiseResult = a | b; // Bitwise OR

bitwiseResult = a ^ b; // Bitwise XOR

bitwiseResult = ~a; // Bitwise NOT

bitwiseResult = a << 1; // Left Shift

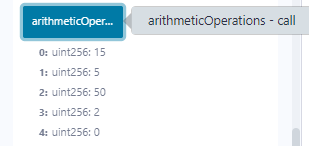
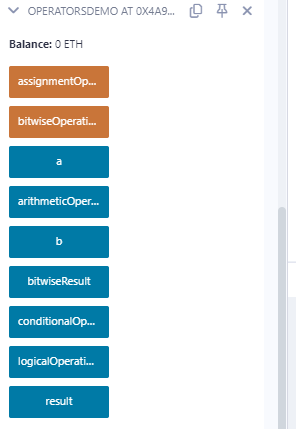
bitwiseResult = a >> 1; // Right Shift

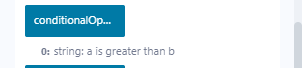
return bitwiseResult;

}

}

Output :





Solidity program to demonstrate Function overloading

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Test {

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

return a + b;

}

function getSum(uint a, uint b, uint c) public pure returns(uint){

return a + b + c;

}

function callSumWithTwoArguments() public pure returns(uint){

return getSum(1,2);

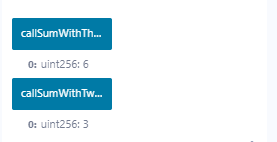
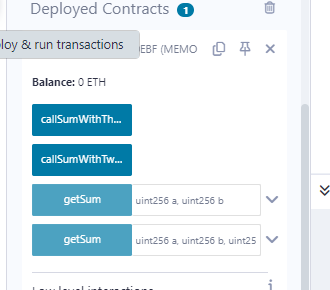
}

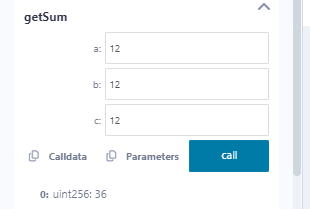
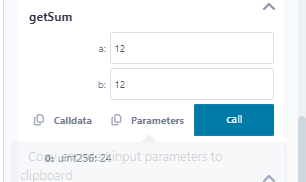
function callSumWithThreeArguments() public pure returns(uint){

return getSum(1,2,3);

}

}







Solidity program to demonstrate Units

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract TimeUnitsDemo {

// Event to log when an action is performed

event ActionPerformed(string message, uint timestamp);

// Function to perform an action with a delay

function performActionAfterDelay(uint delayInMinutes) public {

// Calculate the delay in seconds

uint delayInSeconds = delayInMinutes \* 1 minutes; // 1 minute = 60 seconds

// Emit event with a timestamp

emit ActionPerformed("Action performed after delay.", block.timestamp + delayInSeconds);

}

// Function to get the current time in different units

function getCurrentTime() public view returns (uint secondsSinceEpoch, uint minutesSinceEpoch, uint hoursSinceEpoch) {

// Get the current time in seconds since the Unix epoch

secondsSinceEpoch = block.timestamp;

// Convert to minutes and hours

minutesSinceEpoch = secondsSinceEpoch / 60; // Convert to minutes

hoursSinceEpoch = secondsSinceEpoch / 3600; // Convert to hours

}

// Function to calculate future time in a specific unit

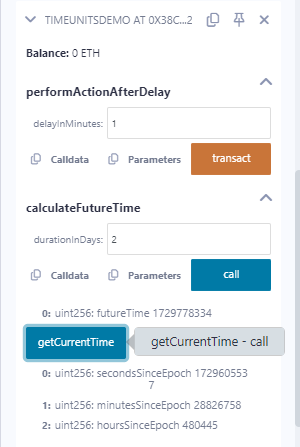
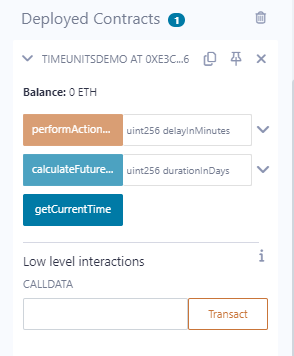
function calculateFutureTime(uint durationInDays) public view returns (uint futureTime) {

// Calculate future time in seconds

futureTime = block.timestamp + (durationInDays \* 1 days); // 1 day = 86400 seconds

}

}



Solidity program to demonstrate Mathematical Funtions

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract MathOperations {

// Function to add two unsigned integers

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

return a + b;

}

// Function to subtract two unsigned integers

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

require(a >= b, "Subtraction underflow"); // Prevents underflow

return a - b;

}

// Function to multiply two unsigned integers

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

return a \* b;

}

// Function to divide two unsigned integers

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

require(b > 0, "Division by zero"); // Prevents division by zero

return a / b;

}

// Function to get the modulus of two unsigned integers

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

require(b > 0, "Division by zero"); // Prevents division by zero

return a % b;

}

// Function to raise a number to a power

function power(uint base, uint exponent) public pure returns (uint) {

uint result = 1;

for (uint i = 0; i < exponent; i++) {

result \*= base;

}

return result;

}

// Function to calculate the square root of a number

function sqrt(uint x) public pure returns (uint) {

if (x == 0) return 0;

uint z = (x + 1) / 2;

uint y = x;

while (z < y) {

y = z;

z = (x / z + z) / 2;

}

return y;

}

}

