**Practical No.3**

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

pragma solidity ^0.5.0; contract HelloWorld {

constructor () public {

}

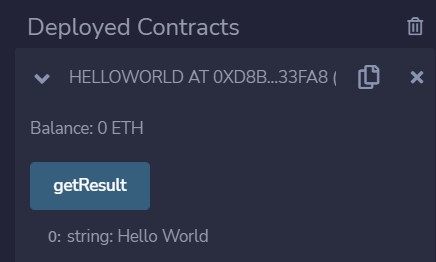
function getResult() public view returns(string memory){

return 'Hello World';

}

}

Output:



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

pragma solidity ^0.5.0; contract SolidityTest { uint storedData; // State variable constructor() public {

storedData = 10;

}

function getResult() public view returns(uint){

uint a = 1; // local variable uint b = 2; uint result = a + b;

return storedData; //access the state variable

}

}

Output:



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

pragma solidity ^0.5.0; contract GetAndSet{ string name; uint age;

function GetandSet() public {

}

function set(string memory newName, uint newAge) public { name = newName;

age = newAge;

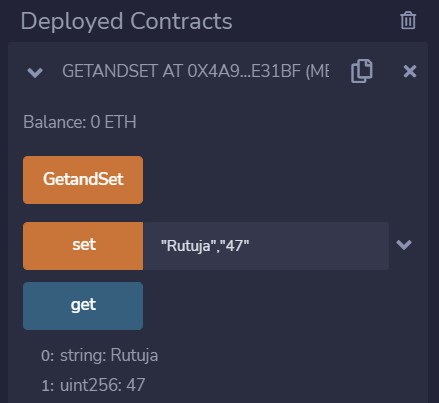
}

function get() public view returns (string memory, uint) { return (name,age);

}

}

Output:



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

pragma solidity ^0.5.0;

contract Owner { address owner; constructor() public { owner = msg.sender;

}

modifier onlyOwner {

require(msg.sender == owner);

}

modifier costs(uint price) {

if (msg.value >= price) {

}

}

function getPrice() public view returns(uint price){ return price;

}

}

contract Register is Owner {

mapping (address => bool) registeredAddresses; uint price;

constructor(uint initialPrice) public { price = initialPrice; }

function register() public payable costs(price) { registeredAddresses[msg.sender] = true;

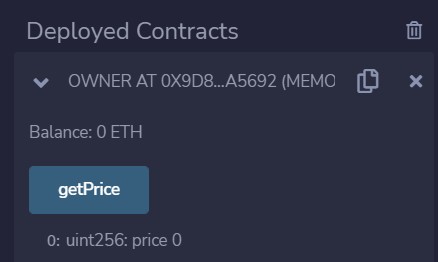
}

function changePrice(uint \_price) public onlyOwner { price = \_price;

}

}

Output:



5.Write a solidity smart contract to demonstrate use of structure.

pragma solidity ^0.5.0; contract test { struct Book { string title; string author;

uint book\_id; }

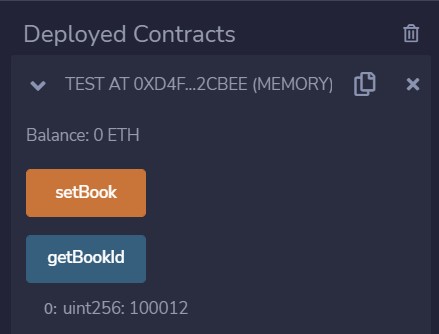
Book book;

function setBook() public {

book = Book('Learn Java', 'TP', 100012); }

function getBookId() public view returns (uint) { return book.book\_id; } }

Output:



6. Write a solidity smart contract to calculate percentage of marks obtained by students for six subject in final examination.

pragma solidity ^0.5.0; contract percentage{

uint sub\_1;uint sub\_2; uint sub\_3;uint sub\_4;uint sub\_5;uint sub\_6;uint total=600; uint marksObtained;

function set(uint s1,uint s2 ,uint s3,uint s4,uint s5,uint s6) public { sub\_1=s1; sub\_2=s2; sub\_3=s3; sub\_4=s4; sub\_5=s5; sub\_6=s6;

marksObtained=sub\_1+sub\_2+sub\_3+sub\_4+sub\_5+sub\_6; marksObtained=marksObtained\*100;

}

function getPercentage() public view returns (uint) {

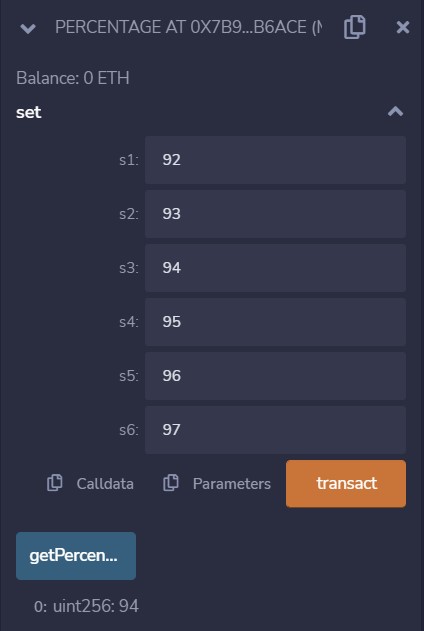
uint percent=marksObtained/total;

return percent;

}

}

Output:



7.Write a solidity smart contract to find the factorial of entered number.

pragma solidity ^0.5.0; contract factorial{ uint number; function set(uint n) public {

number=n;

}

function getFactorial() public view returns (uint) { uint f=1;

for(uint i=2;i<=number;i++){ f=f\*i;

}

return f;

}

}

Output:



8.Write a solidity smart contract to check whether entered number is palindrome or not.

pragma solidity ^0.5.0; contract palindrome{ uint number;

function set(uint n) public {

number=n;

}

function getPalindrome() public view returns (bool ) { uint r;

uint n=number; uint reverseNumber=0;

while(n>0){ r=n%10;

reverseNumber=reverseNumber\*10+r; n=n/10;

}

if(reverseNumber==number){

return true;

}

else

return false;

}

}

Output:



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

pragma solidity ^0.5.0; contract fibonacci{ uint number\_of\_terms; function set (uint n) public {

number\_of\_terms=n;

}

function getFiboSeries() public view returns (uint[] memory ) { uint a=0; uint b=1; uint c;

uint[] memory result=new uint[](number\_of\_terms); result[0]=a; result[1]=b;

for(uint i=2;i<number\_of\_terms;i++){ c=a+b; result[i]=c; a=b;

b=c;

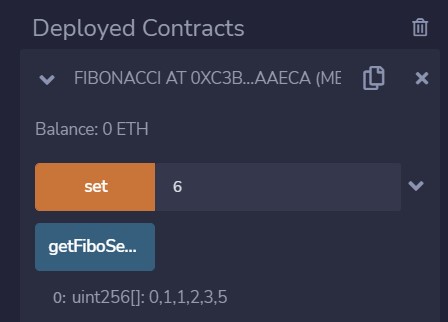
}

return result;

}

}

Output:



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

pragma solidity ^0.5.0; contract prime{

function isPrime(uint n) public view returns (string memory ) { string memory message=""; if(n==0){

return "Invalid input.";

}

else if (n==1){

return "1 is neither prime nor composite.";

}

else if(n==2){

return "Entered Number is prime.";

}

else{ bool flag=true; for(uint i=2;i<=n/2;i++ ){ if(n%i==0){ flag=false;

break;

}

}

if(flag){

return "Entered Number is prime.";

}

else{

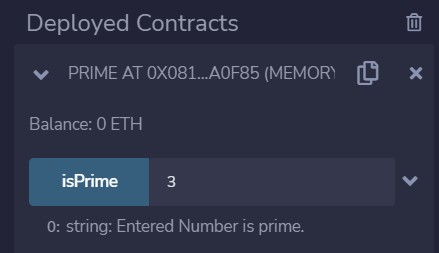
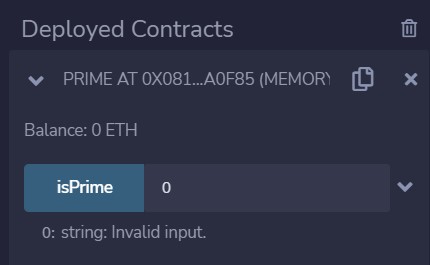
return "Entered Number is not prime.";

}

}

} }

Output:



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

pragma solidity ^0.5.0; contract arithmetic\_calci{

function add(uint n1,uint n2) public view returns (uint result ) { return n1+n2;

}

function sub(uint n1,uint n2) public view returns (uint result ) { return n1-n2;

}

function mul(uint n1,uint n2) public view returns (uint result ) { return n1\*n2;

}

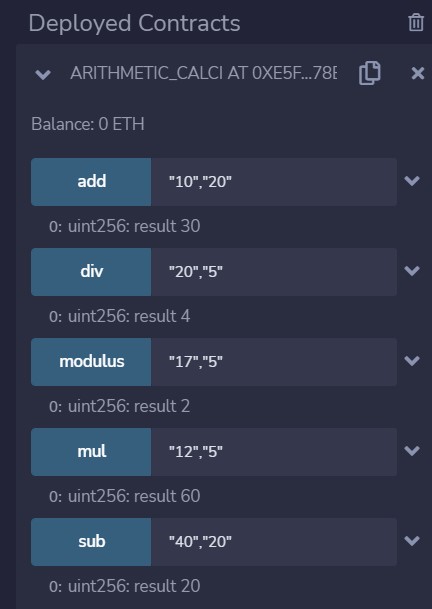
function div(uint n1,uint n2) public view returns (uint result ) { return n1/n2;

}

function modulus(uint n1,uint n2) public view returns (uint result ) { return n1%n2;

}

Output:



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

pragma solidity ^0.5.0; contract inbuilt\_function\_demo{

function callAddMod(uint n1,uint n2,uint n3) public pure returns(uint){ return addmod(n1,n2,n3);

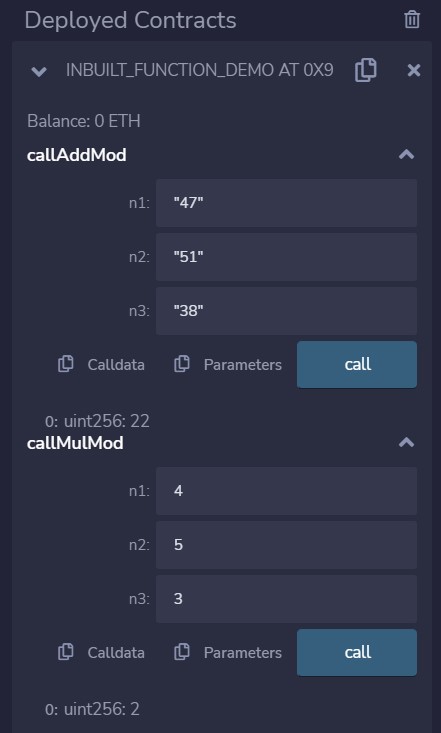
}

function callMulMod(uint n1,uint n2,uint n3) public pure returns(uint){ return mulmod(n1,n2,n3);

}

}

Output:



13.Write a solidity smart contract to demonstrate inbuilt mathematical functions.

pragma solidity ^0.5.0; contract C {

//private state variable

uint private data;

//public state variable uint public info; //constructor constructor() public {

info = 10;

}

//private function

function increment(uint a) private pure returns(uint) { return a + 1; }

//public function

function updateData(uint a) public { data = a; } function getData() public view returns(uint) { return data; }

function compute(uint a, uint b) internal pure returns (uint) { return a + b; }

}

//Derived Contract contract E is C { uint private result; C private c; constructor() public {

c = new C();

}

function getComputedResult() public { result = compute(3, 5);

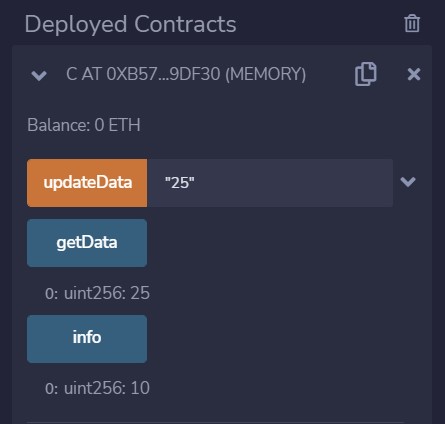
}

function getResult() public view returns(uint) { return result; }

function getData() public view returns(uint) { return c.info(); }

}

Output:



14.Write a solidity smart contract to demonstrate inheritance in contract.

pragma solidity ^0.5.0; contract C {

//private state variable

uint private data;

//public state variable uint public info; //constructor constructor() public {

info = 10;

}

//private function

function increment(uint a) private pure returns(uint) { return a + 1; }

//public function

function updateData(uint a) public { data = a; } function getData() public view returns(uint) { return data; }

function compute(uint a, uint b) internal pure returns (uint) { return a + b; }

}

//Derived Contract contract E is C { uint private result; C private c; constructor() public {

c = new C();

}

function getComputedResult() public { result = compute(3, 5);

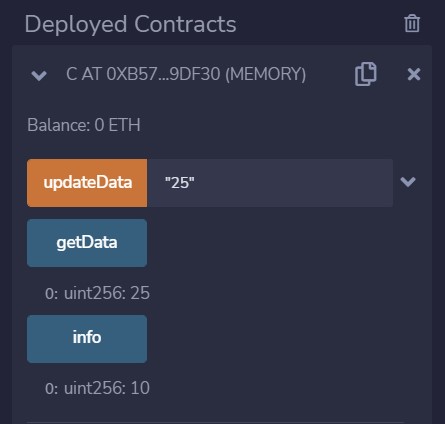
}

function getResult() public view returns(uint) { return result; }

function getData() public view returns(uint) { return c.info(); }

}

Output:



15.Write a solidity smart contract to demonstrate events.

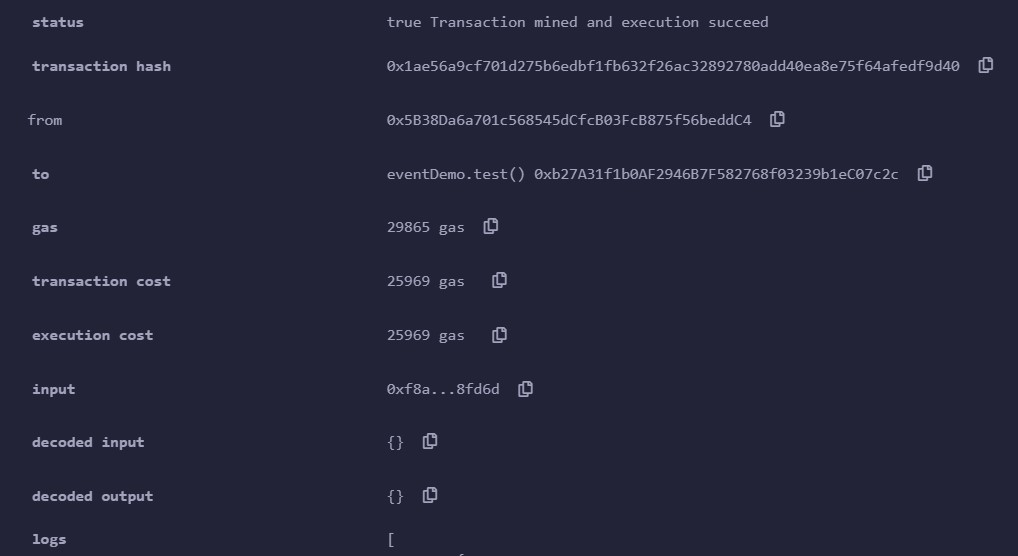
pragma solidity ^0.5.0; contract eventDemo{

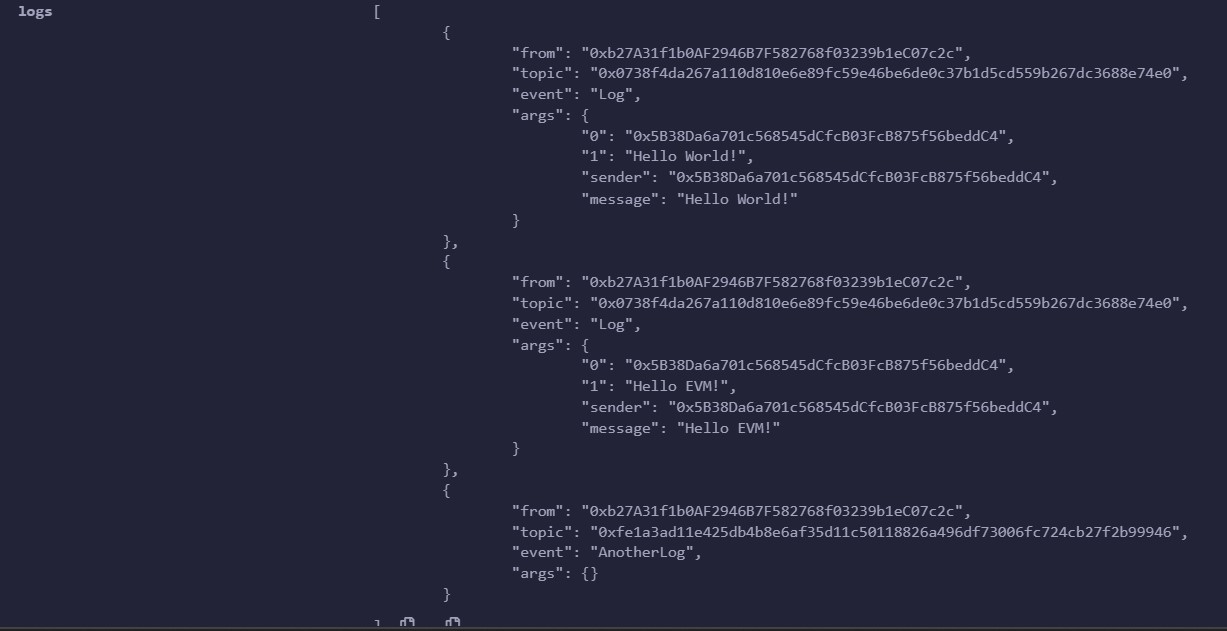
event Log(address indexed sender, string message); event AnotherLog(); function test() public { emit Log(msg.sender, "Hello World!"); emit Log(msg.sender, "Hello EVM!"); emit AnotherLog();

}

}

Output:





16.Write a solidity smart contract to demonstrate error handling.

pragma solidity 0.5.0;

contract ErroHandling {

function checkInput(uint \_input) public view returns(string memory)

{

require(\_input >= 0, "invalid uint8"); require(\_input <= 255, "invalid uint8"); return "Input is Uint8";

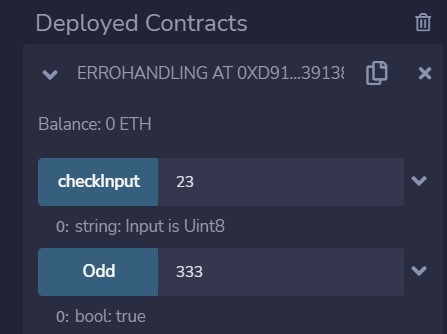
}

function Odd(uint \_input) public view returns(bool)

{

require(\_input % 2 != 0); return true; } }

Output:



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

pragma solidity ^0.5.0; contract Banking{

mapping(address=>uint)public userAccount;

mapping(address=>bool)public userExist;

function createAcc() public payable returns(string memory){ require(userExist[msg.sender]==false,'Account Alread Created'); if(msg.value==0){

userAccount[msg.sender]=0;

}

userAccount[msg.sender]=msg.value; userExist[msg.sender]=true;

return 'account Created';

}

function deposite()public payable returns(string memory){ require(userExist[msg.sender]==true,'Account does not exist!'); require(msg.value>0,'value for deposite is zero');

userAccount[msg.sender]=userAccount[msg.sender]+msg.value; return 'Amount deposited successfully!';

}

function withdraw(uint amount)public payable returns(string memory){ require(userExist[msg.sender]==true,'Account does not exist!'); require(msg.value>0,'deposite value should greater than zero');

require(msg.value>= amount,'Amount shound be equal to or greater then balance'); userAccount[msg.sender]=userAccount[msg.sender]-amount; return 'Amount withdraw successfully!';

}

function transferAmount(address payable userAddress,uint amount)public payable returns(string memory){ require(userAccount [msg.sender]>amount,'insufficent balance in bank'); require(userExist[msg.sender]==true,'Account does not created'); require(userExist[userAddress]==true,'transfer Amount does not efficient'); require(amount>0,'Enter non zero value for sending');

userAccount[msg.sender]=userAccount[msg.sender]-amount;

userAccount[userAddress]=userAccount[userAddress]+amount; return 'transfer successfully';

}

function sendAmount(address payable toAddress , uint256 amount)public payable returns(string memory){ require(amount>0,'Enter non zero value for withdrawal'); require(userExist[msg.sender]==true,'Account does not created'); require(userAccount [msg.sender]>amount,'insufficent balance in bank');

userAccount[msg.sender]=userAccount[msg.sender]-amount; toAddress.transfer(amount);

return'transfer successfully';

}

function userAccountBalance()public view returns(uint) { return userAccount[msg.sender];

}

function accountExist()public view returns(bool) { return userExist[msg.sender];

}

}

Output:

