**EXPERIMENT - 7**

**1.**

**AIM:** To understand the types and scope of variables in solidity

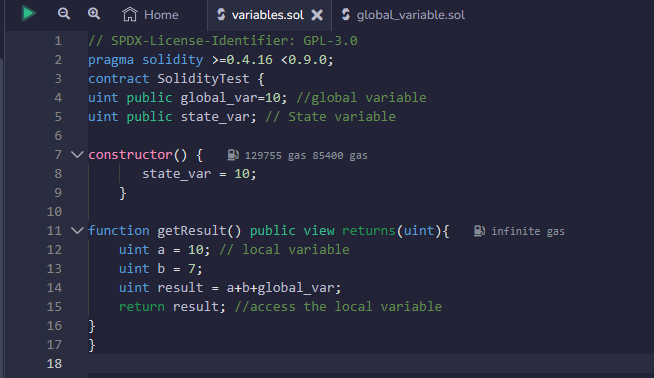
**DESCRIPTION**:

Solidity is an object-oriented programming language for implementing smart contracts on various blockchain platforms, most notably, Ethereum. Solidity is statically typed, supports inheritance, libraries, and complex user-defined types, among other features.

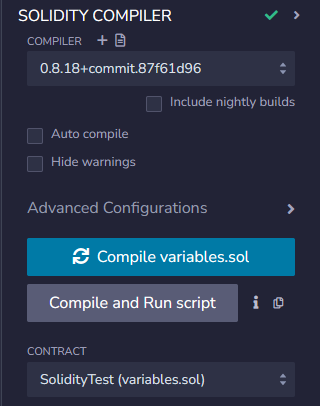
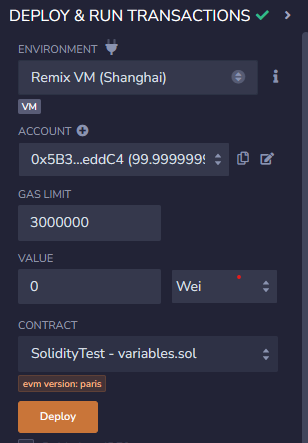
**Types of variables:**

* + - 1. **State Variables:** Values of these variables are permanently stored in the contract storage. Each function has its own scope, and state variables should always be defined outside of that scope.
      2. **Local Variable:** Values of these variables are present till the function executes and it cannot be accessed outside that function. This type of variable is usually used to store temporary values.
      3. **Global Variables:** These are some special variables that can be used globally and give information about the transactions and blockchain properties. Some of the global variables are listed below:
         1. block.coinbase
         2. block.gaslimit
         3. block.timestamp
         4. msg.sender, etc.

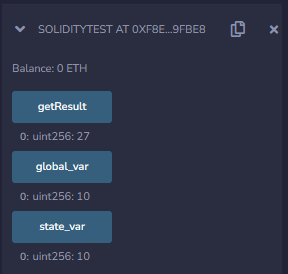
**PROCEDURE:**



Compile and deploy the code:

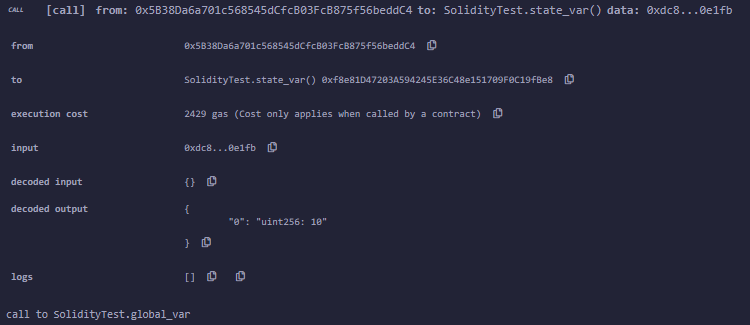
 

Giving Input and calling each function:



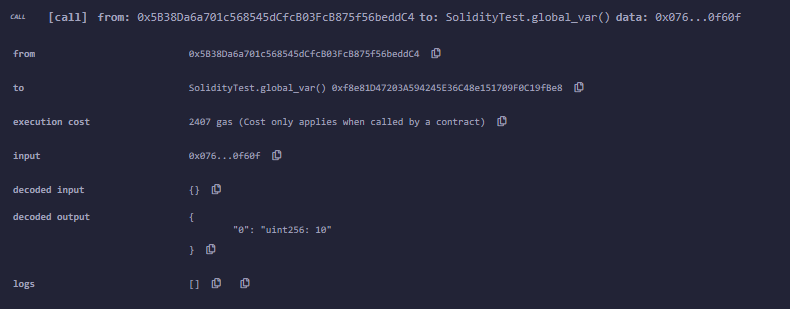
**OUTPUT:**

Calling the function SolidityTest.state\_var():

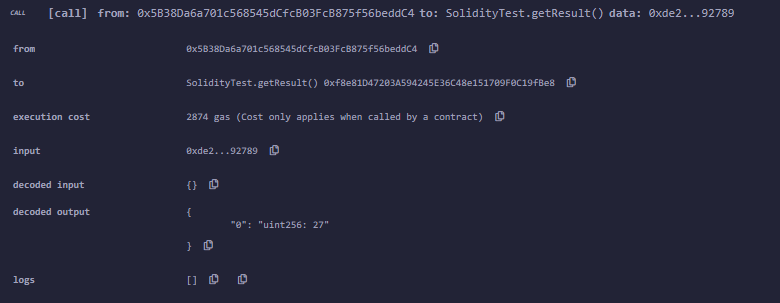


The above function stores the state variable state\_var in a constructor and returns the value when called.

Calling the function SolidityTest.global\_var():



Calling the function SolidityTest.getResult():



In the above function, two local variables are declared and the sum of the local variables along with the global variable is returned. Thus, the function demonstrates the scope of local and global variable.

**2.**

**AIM:** To understand the global variables in solidity

**DESCRIPTION**:

Solidity is an object-oriented programming language for implementing smart contracts on various blockchain platforms, most notably, Ethereum. Solidity is statically typed, supports inheritance, libraries, and complex user-defined types, among other features.

**Global Variables:** These are some special variables that can be used globally and give information about the transactions and blockchain properties. Some of the global variables are listed below:

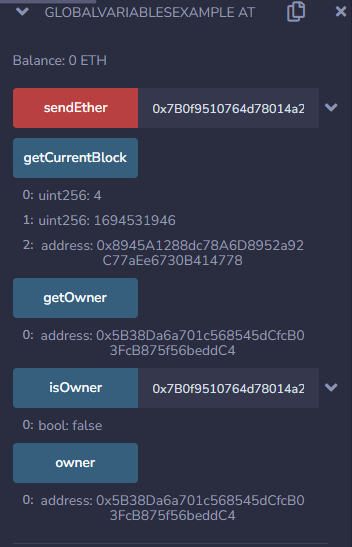
* + - * 1. **blockhash** - Hash of a given block, works for only 256 most recent transactions excluding current blocks
        2. **block.coinbase** - Address of current blocks miner
        3. **block.difficulty** - The difficulty of the current block
        4. **block.gaslimit** - Gaslimit of the current block
        5. **block.number** - Block number of the current block
        6. **block.timestamp** - The timestamp of the current block as seconds since Unix epoch
        7. **gasleft()** - Amount of gas left
        8. **msg.data** - Complete call data of block
        9. **msg.sender** - The sender of message i.e. current caller
        10. **msg.sig** - First four bytes of call data i.e. function identifier
        11. **msg.value** - Amount of Wei sent with a message
        12. **now** - The timestamp of the current block
        13. **tx.gasprice** - Price of gas for the transaction
        14. **tx.origin** - Transaction sender

**PROCEDURE:**

**Code:**

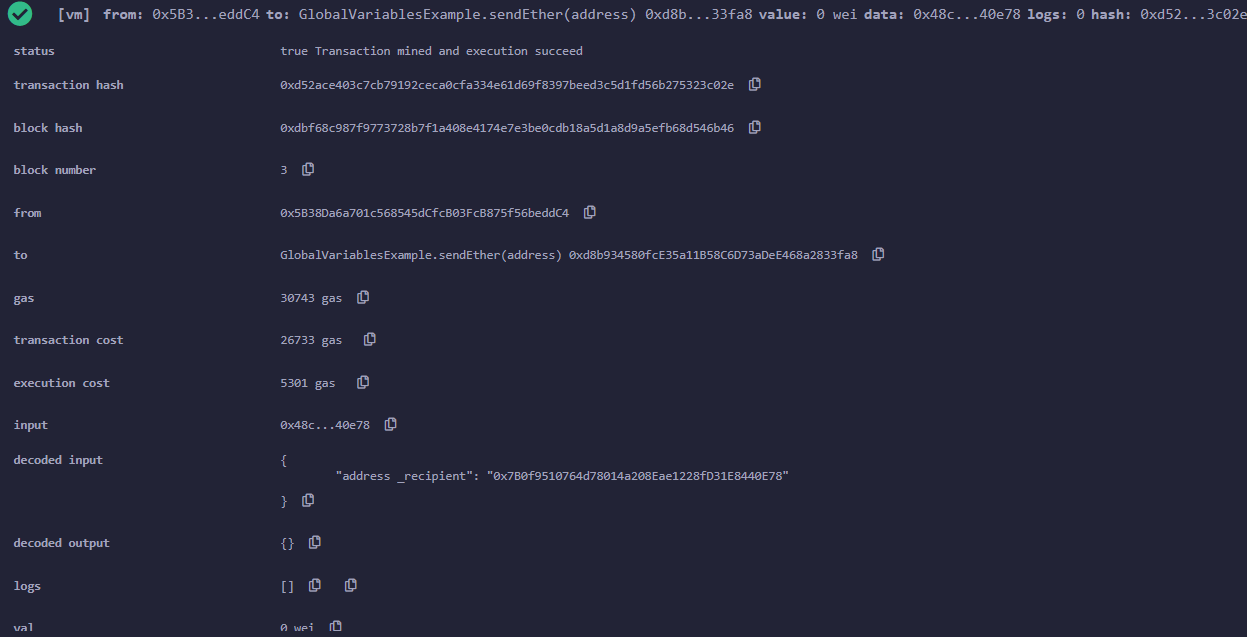


Giving Input and calling each function:



**OUTPUT:**

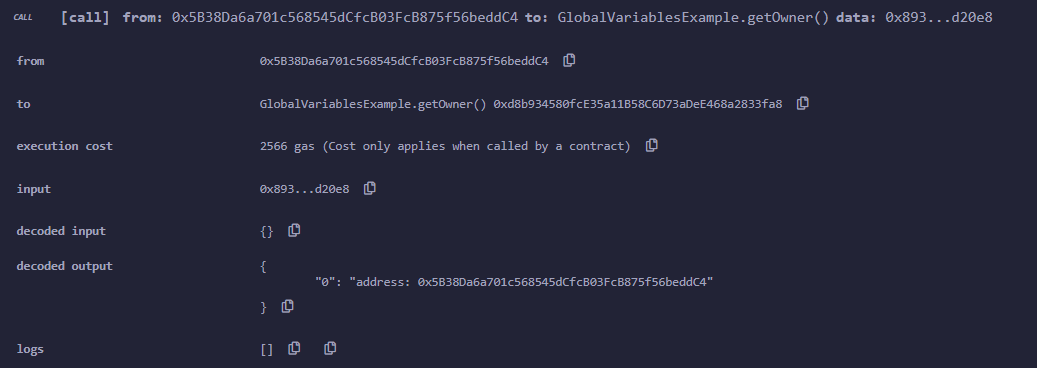
Calling the function GlobalvariableExample.sendether():



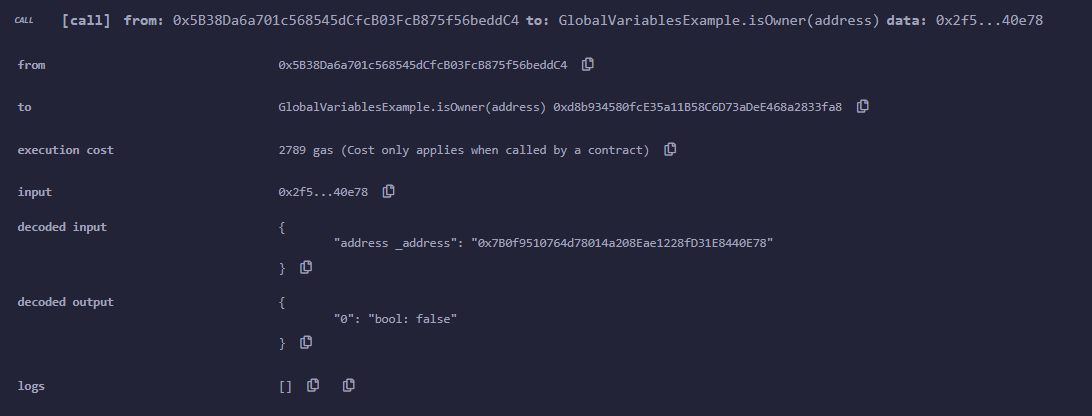
Calling the function GlobalvariableExample.getCurrentBlock():



Calling the function GlobalvariableExample.getowner():



Calling the function GlobalvariableExample.isowner():



Calling the function GlobalvariableExample.owner():



**3.**

**AIM:** To write a program to implement decision making statements in solidity

**DESCRIPTION**:

Solidity is an object-oriented programming language for implementing smart contracts on various blockchain platforms, most notably, Ethereum. Solidity is statically typed, supports inheritance, libraries, and complex user-defined types, among other features.

Decision making in programming is used when we have to adopt one out of a given set of paths for program flow. For this purpose, conditional statements are used which allows the program to execute the piece of code when the condition fulfills. Solidity uses control statements to control the flow of execution of the program to advance and branch-based changes to the state.

**If statement**

This is the most basic conditional statement. It is used to make a decision whether the statement or block of code will be executed or not. If the condition is true the statements will be executed, else no statement will execute.

**if…else statement**

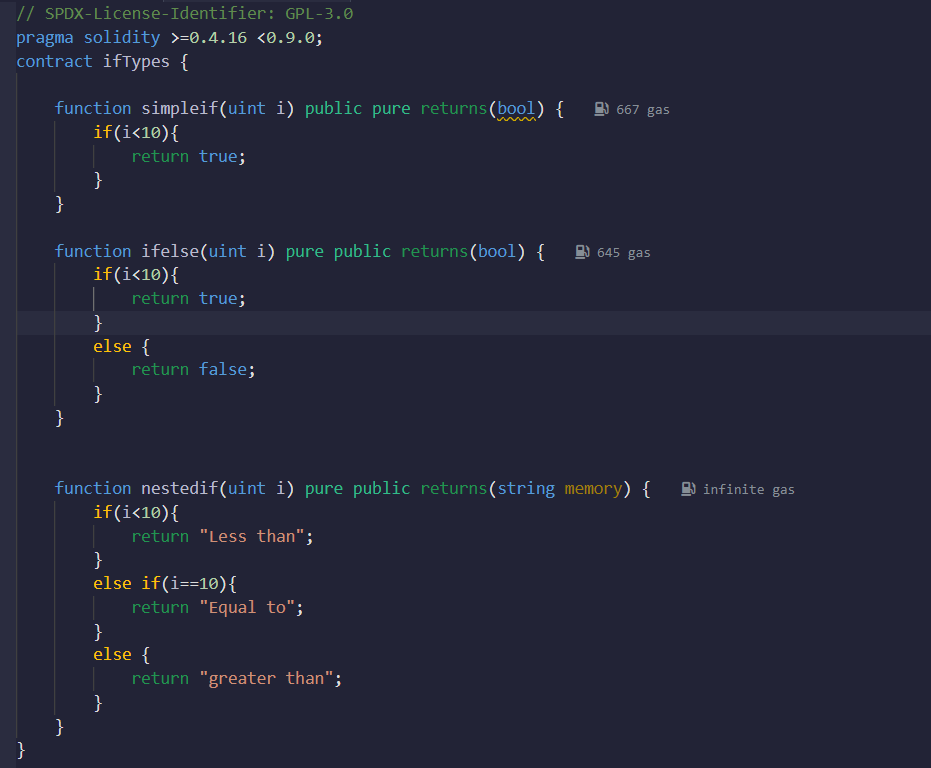
This statement is the next form of conditional statement which allows the program to execute in a more controlled way. Here if the condition is true then the, if block is executed while if the condition is false then else block, is executed.

**if…else if…else statement**

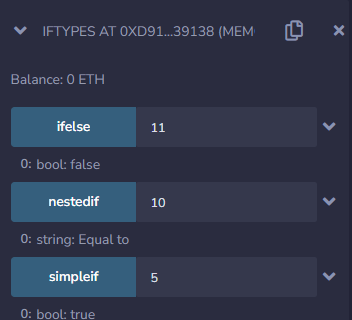
This is a modified form of if…else conditional statement which is used to make a decision among several options. The statements start execution from if statement and as the condition of any if block is true the block of code associated with it is executed and rest if are skipped, and if none of the condition is true then else block executes.

**PROCEDURE:**

**Code:**

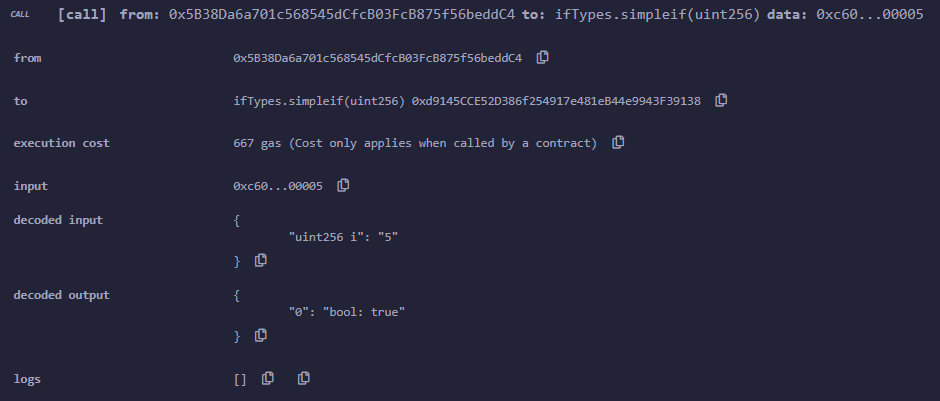


Giving Input and calling each function:

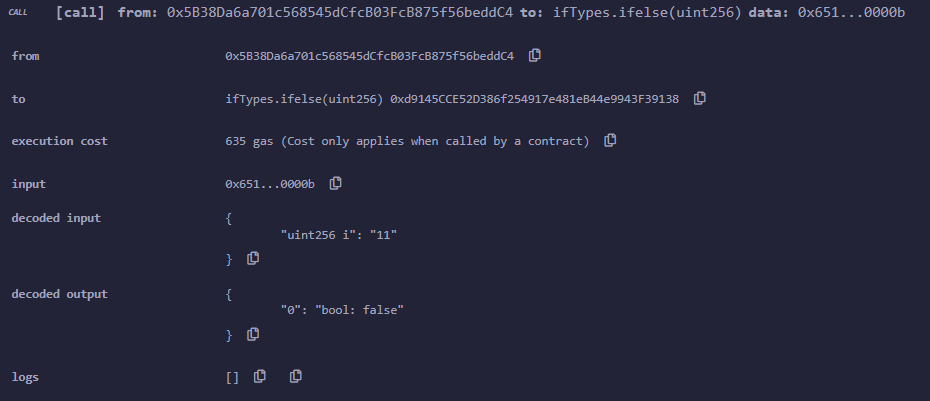


**OUTPUT:**

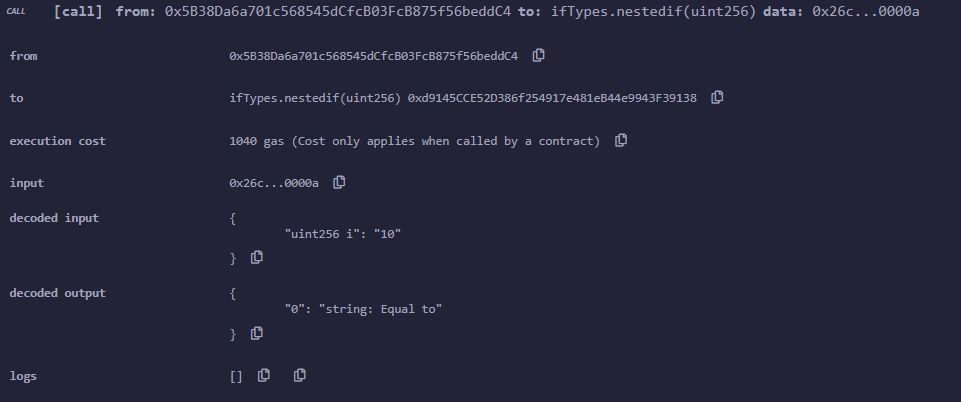
Calling the function ifTypes.simpleif():



Calling the function ifTypes.ifelse():



Calling the function ifTypes.nestedif():



**EXPERIMENT - 8**

**1.**

**AIM:** To get started with the Solidity programming language and deploy smart contracts.

**DESCRIPTION:**

**Smart Contract:**

A Smart Contract is a computer program that has been deployed onto a blockchain. These programs automatically execute, control or document events and actions according to the terms of a contract or an agreement. The main goal of smart contracts are to eliminate trusted intermediaries and avoid accidental exceptions and mishandling.

**Solidity:**

Solidity is an object-oriented programming language for implementing smart contracts on various blockchain platforms, most notably Ethereum. Programs written in Solidity run on the Ethereum Virtual Machine.

**Ganache, Truffle and Node.js:**

Ganache provides an easy one-click solution for creating a local blockchain testnet onto which a tool such as Truffle which runs on top of node.js can be used to deploy and interact with Smart Contracts.

**Procedure:**

1. Download ganache for your OS.
2. Install latest version of Node.js

For debian-based OS use the command:

sudo snap install node

1. Install truffle using npm

npm install truffle -g

1. Launch ganache, click on the Quickstart button and note the port number used.
2. Run the following command in your workspace folder.

truffle init

1. Edit truffle-config.js and uncomment the following lines:

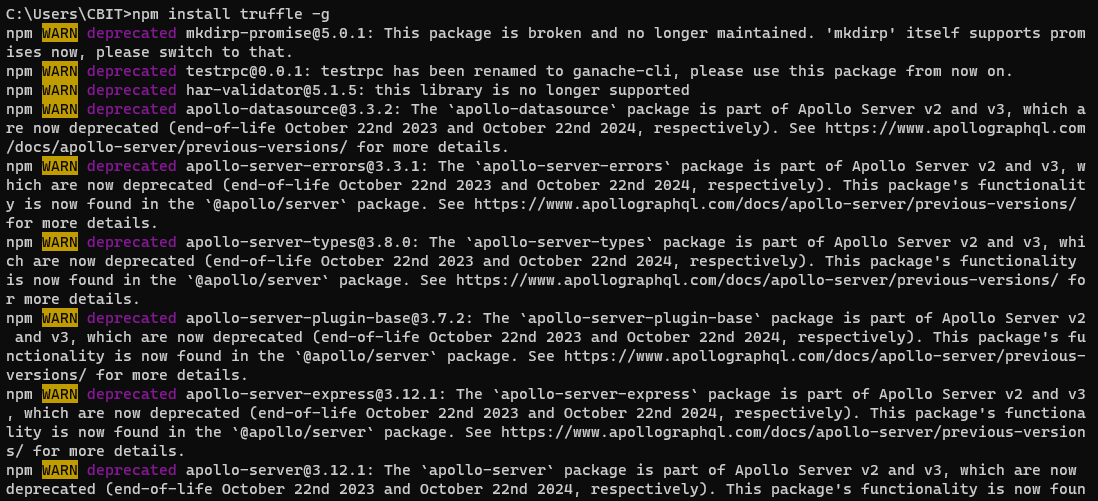
Replace the port with the one displayed in ganache.

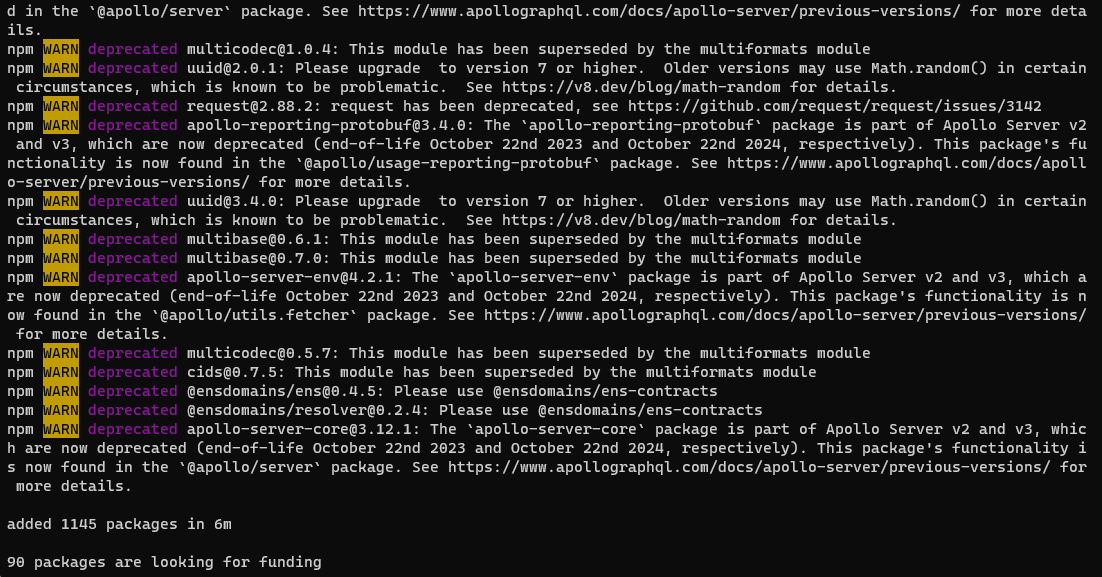
1. Write your smart contracts using your favourite text editor in the contracts directory.
2. Create a new .js file in the migrations directory and add the following lines for each smart contract file in contracts/.
   1. var MyContract = artifacts.require("MyContract");
   2. module.exports = function(deployer) {
   3. // deployment steps
   4. deployer.deploy(MyContract);
   5. };
3. Deploy the contracts using “truffle migrate” or “truffle deploy” or “truffle compile”.
4. Interact with the smart contracts by running “truffle console”.

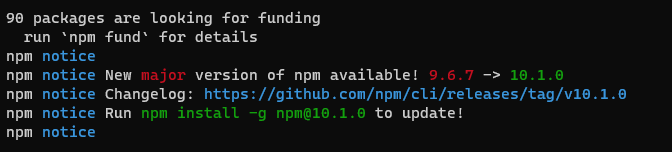
**PROGRAMS AND OUTPUTS:**



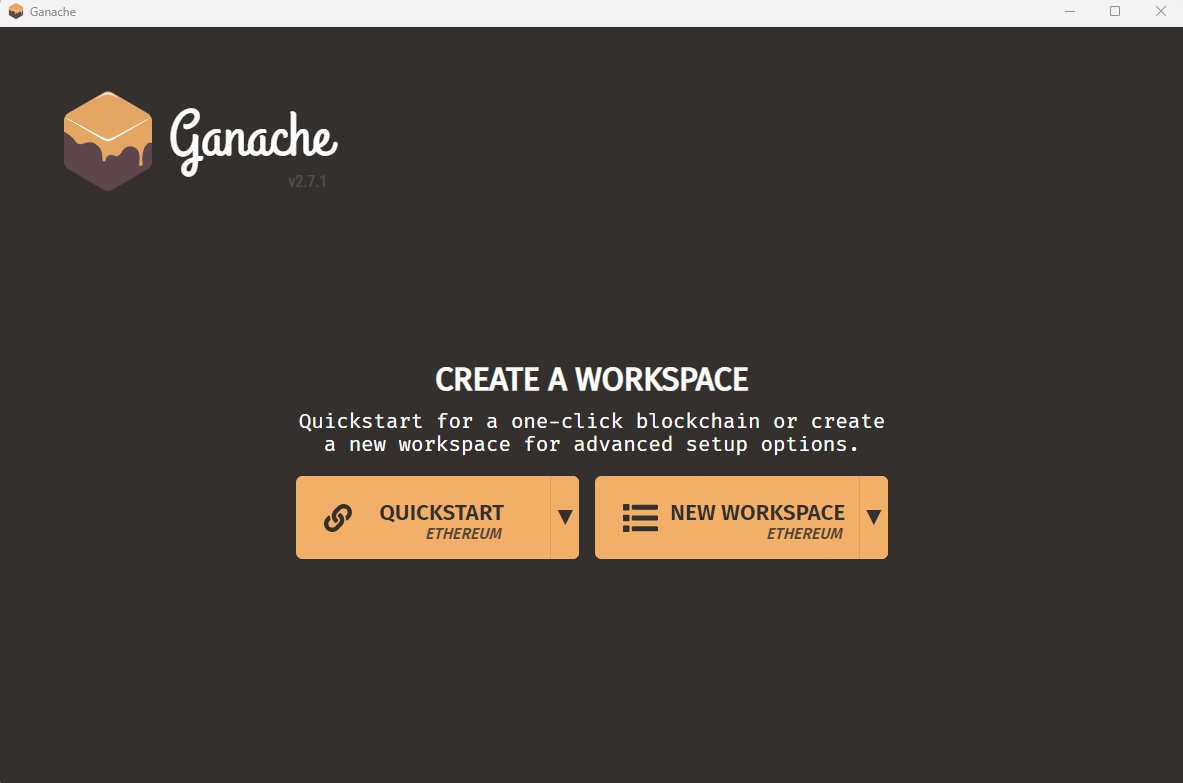
npm install truffle -g



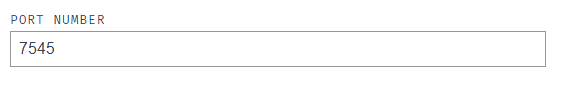




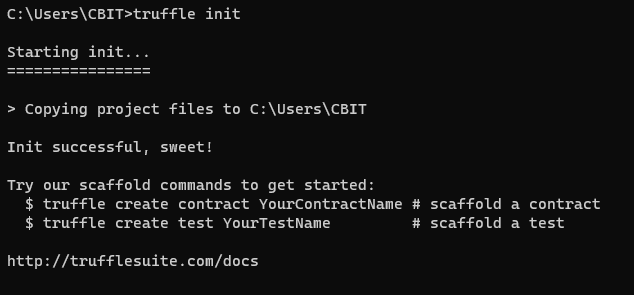
Ganache environment – Quickstart (Ethereum)



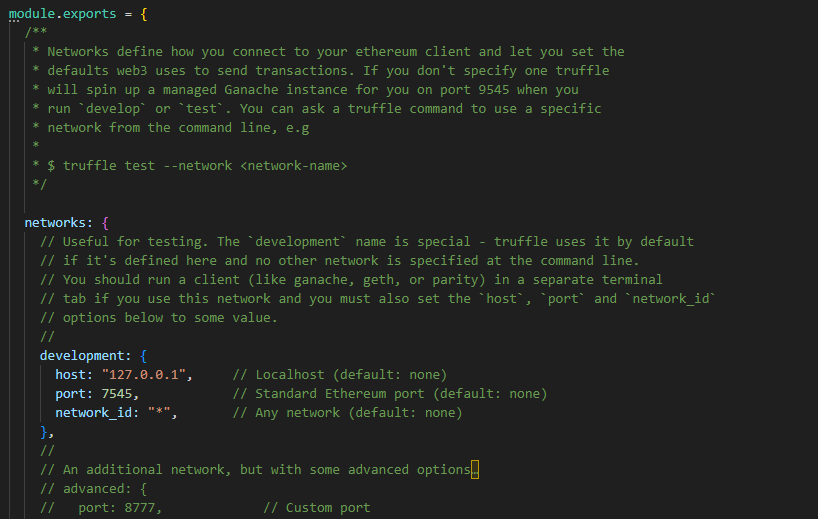
Noting the Port number



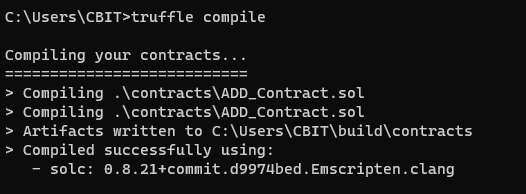
Truffle init



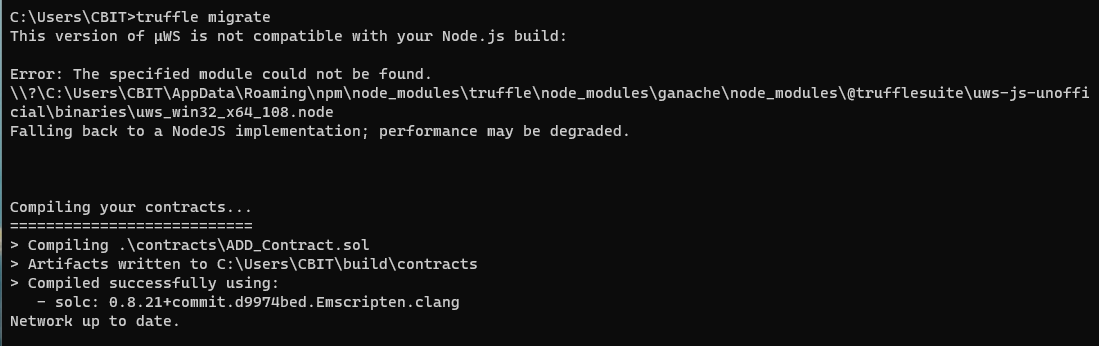
Truffle-config.js



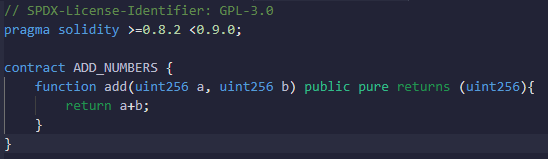
Truffle Compile :



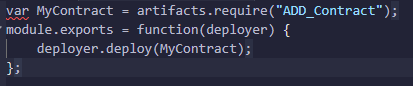
Truffle Migrate :



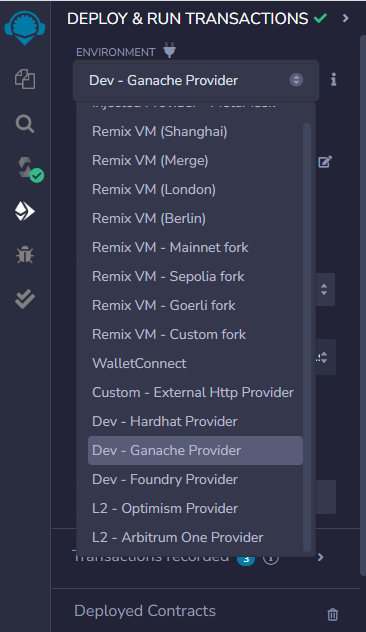
Solidity smart contract to add 2 numbers



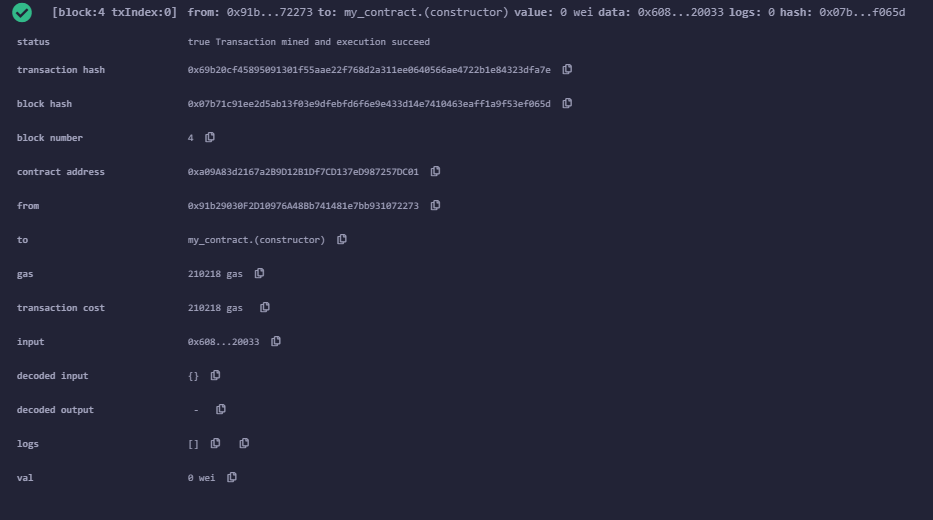
.js file in the migrations directory -



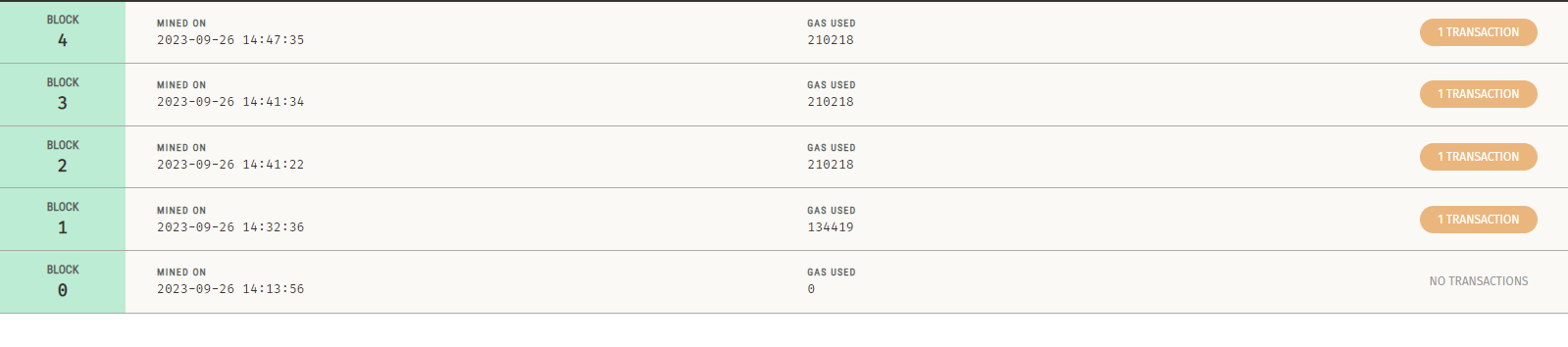
Interacting with smart contract:



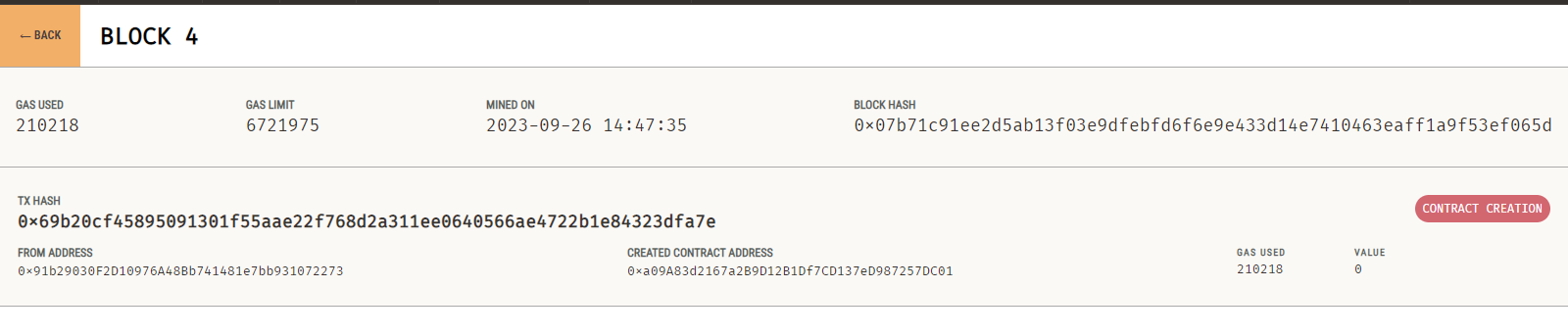
In the remix environment , after compiling the code In the deployment section select Dev-Ganache Provider in the



Viewing the deployed transactions on Ganache:

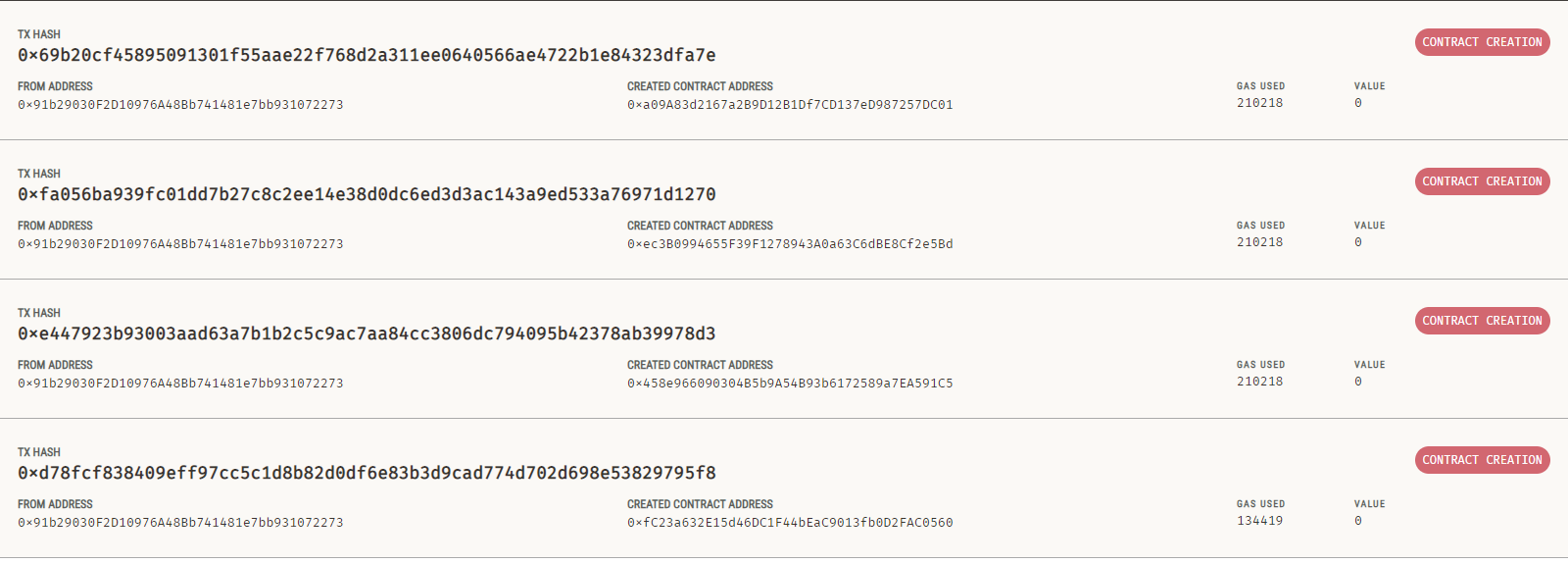


Above is the list of blocks having the transactions that have being executed on the local blockchain. It consists of all the data related to the transactions likes, the gas fee used and the timestamp.



Above are the details in each block . It consists of various transactions executed. It consists of the gas used, gas limit, timestamp, the block hash, and the transaction’s hash value.





**2.**

**AIM:** To understand and implement the concept of loops in solidity.

**DESCRIPTION:**

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:

**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 :

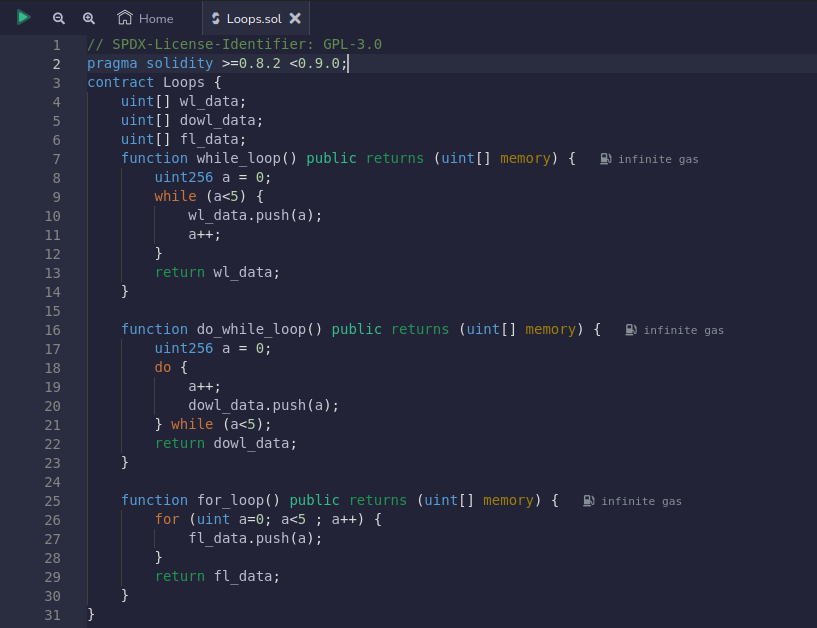
**Syntax:**

for (initialization; test condition; iteration statement) {

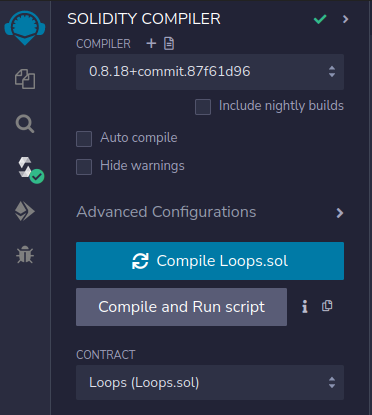
statement or block of code to be executed if the condition is True

}

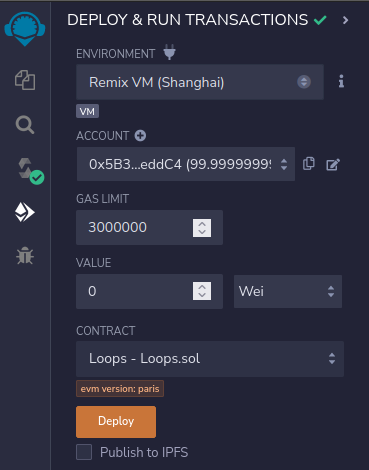
**PROCEDURE:**



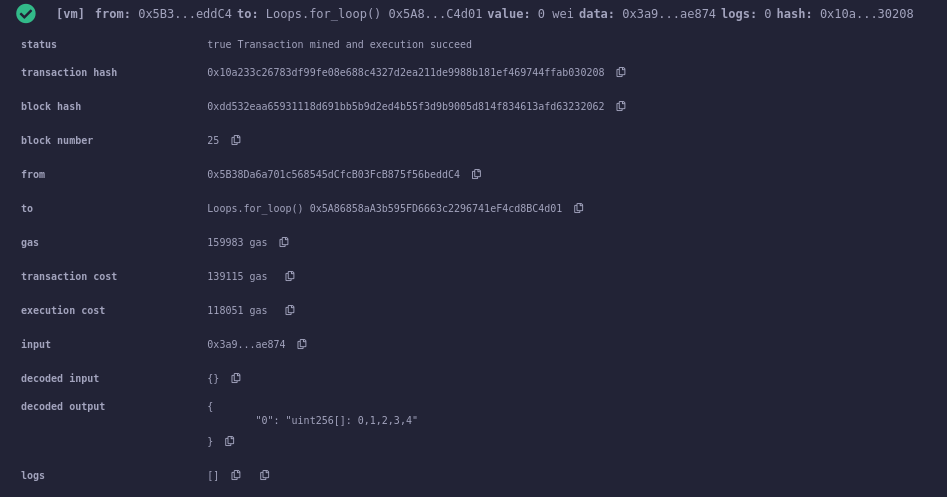
Compiling the smart contract :

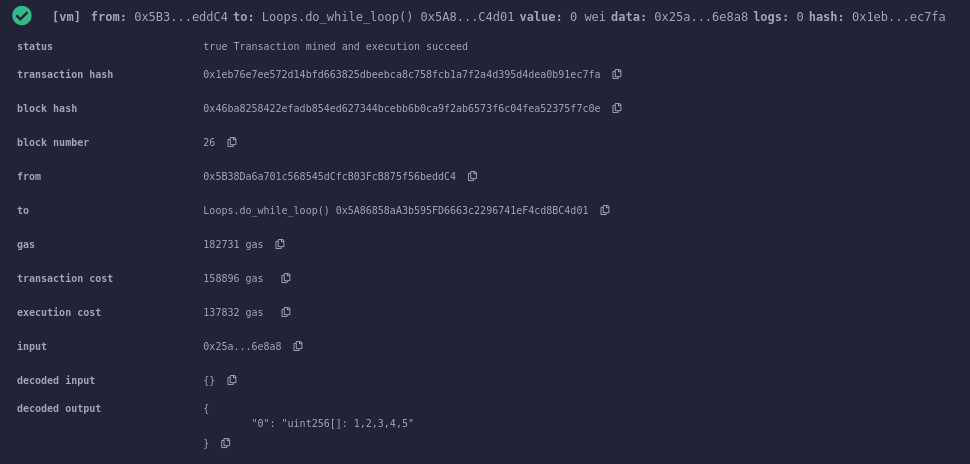


Deploying the smart contract:



**OUTPUT:**





**EXPERIMENT – 9**

**AIM:** To understand and implement the concept of arrays and structures in solidity.

**DESCRIPTION:**

**Arrays:**

* Arrays are data structures that store the fixed collection of elements of the same data types in which each and every element has a specific location called index.
* Instead of creating numerous individual variables of the same type, we just declare one array of the required size and store the elements in the array and can be accessed using the index.
* In Solidity, an array can be of fixed size or dynamic size.   
  Arrays have a continuous memory location, where the lowest index corresponds to the first element while the highest represents the last
* **1 .** **Fixed size array:** the size of an array is fixed (predefined) at the time of compilation
* data\_type[size] array\_name = <elements>;
* **2 . Dynamic size array:** dynamic means the size of an array can change
* data\_type[] array\_name = <elements>;

**Structures:**

* 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.
* Structs can be declared outside of a contract and imported in another contract.

**Syntax:**

struct <structure\_name> {

<data type> variable\_1;

<data type> variable\_2;

}

### Example:

struct Book {

string title;

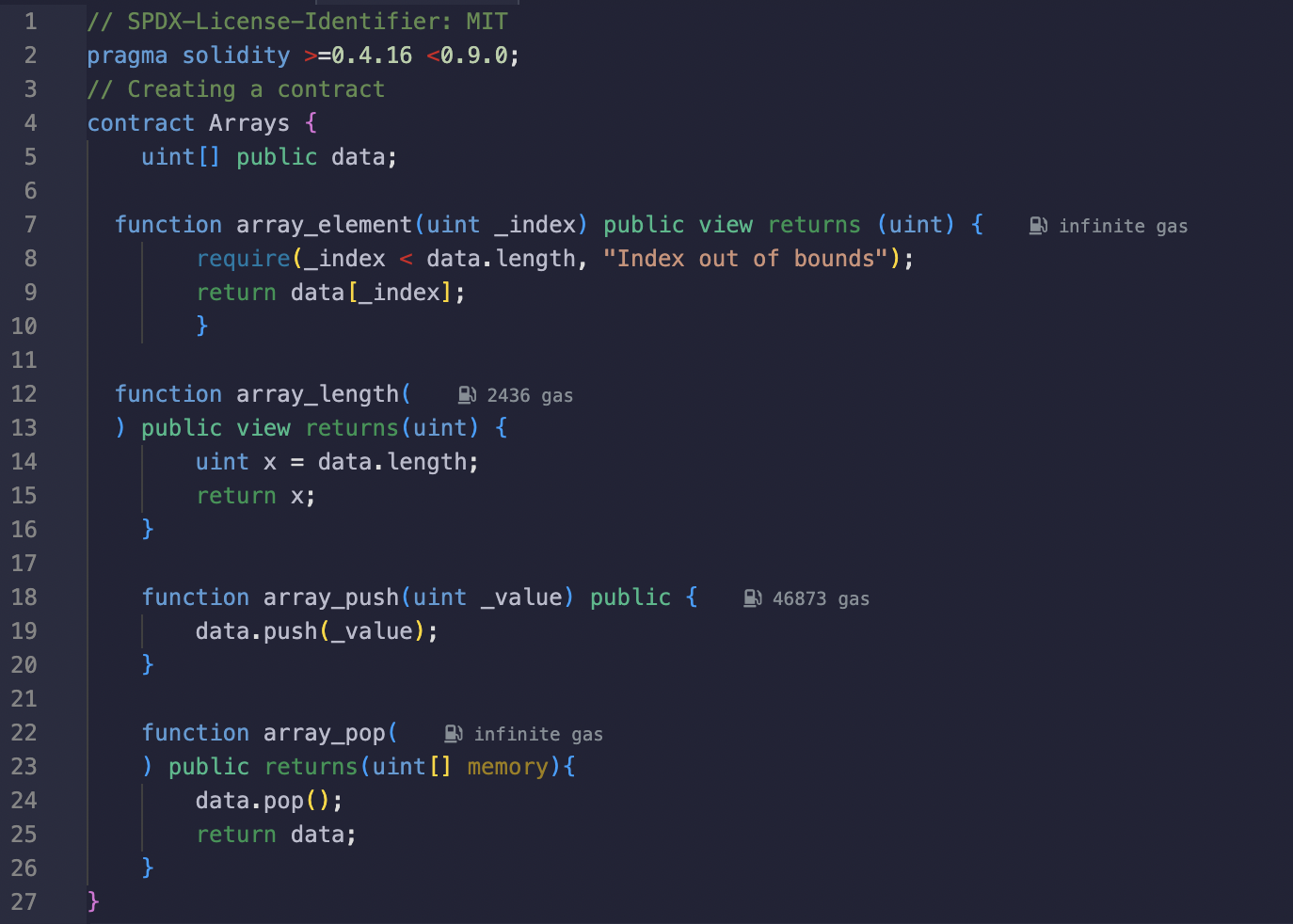
string author;

uint book\_id;

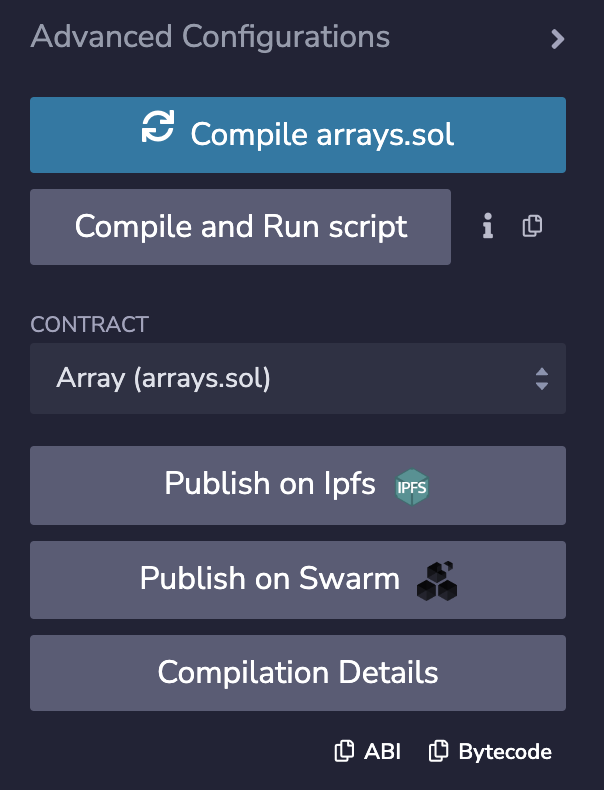
}

**PROCEDURE:**

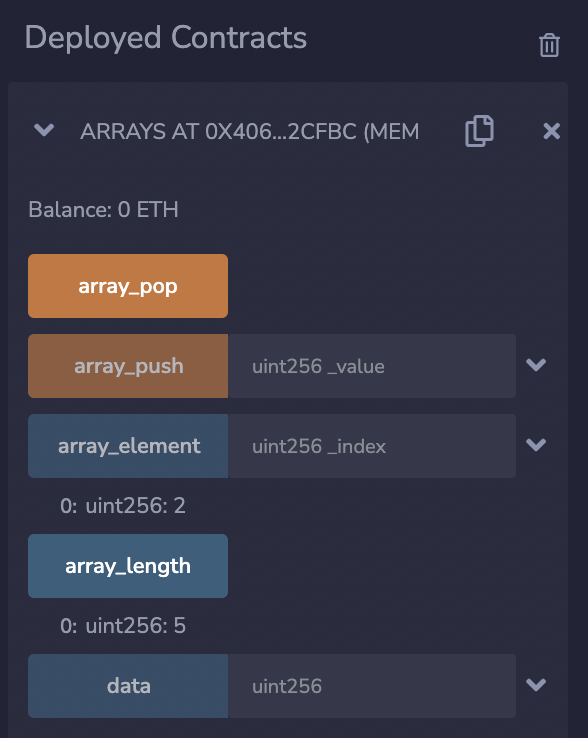
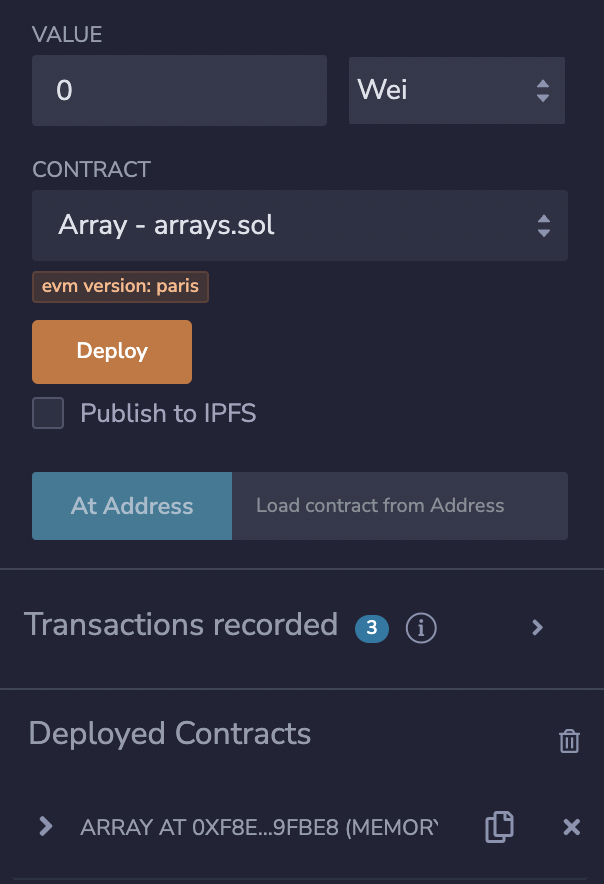
**(ARRAYS)**



**Compiling the smart contract :**

****

**Deploying the smart contract :**

****

**OUTPUT:**

**Pushing an element into array**

****

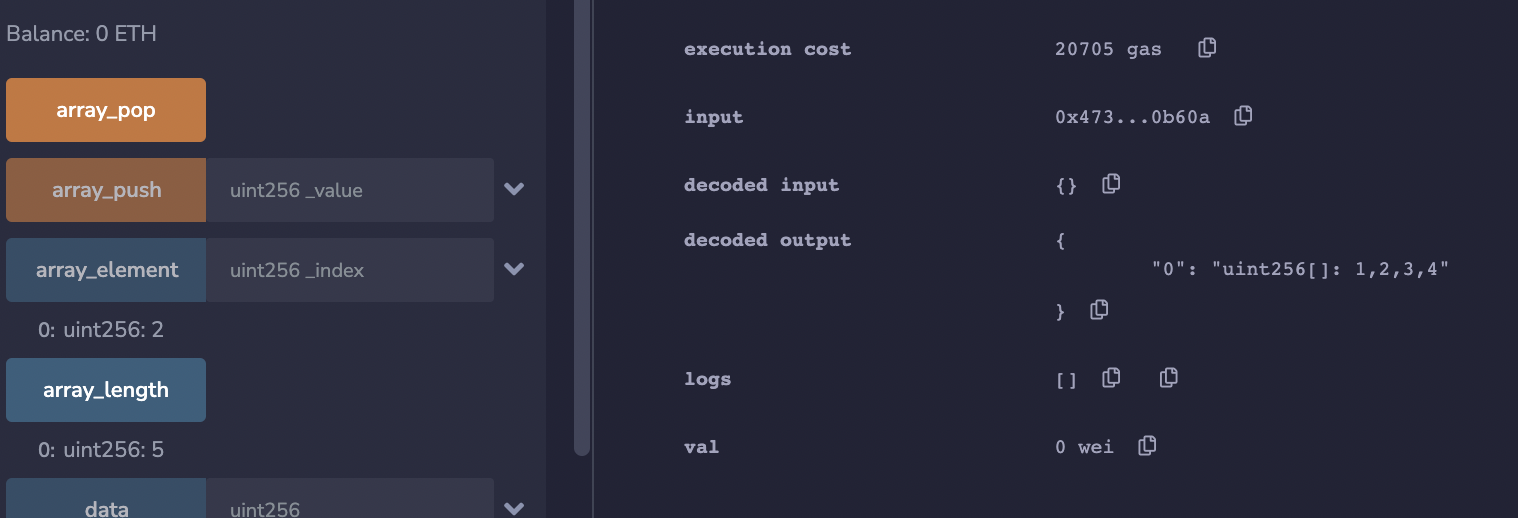
**Accessing an element**

****

**Length of array**

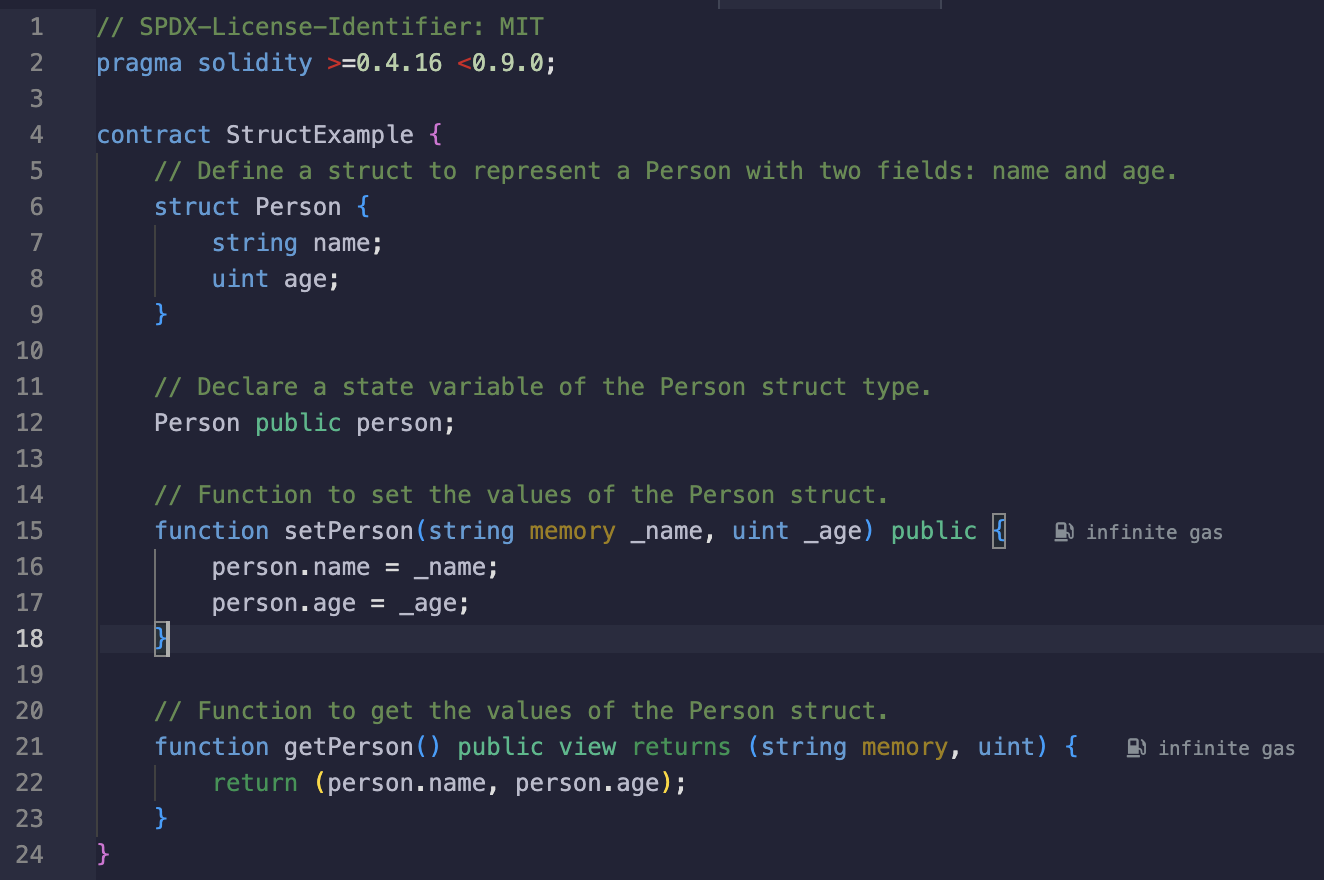
****

**Pop element from an array**

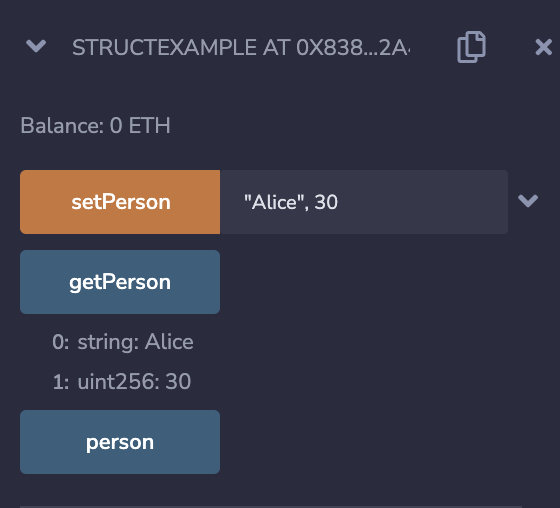
****

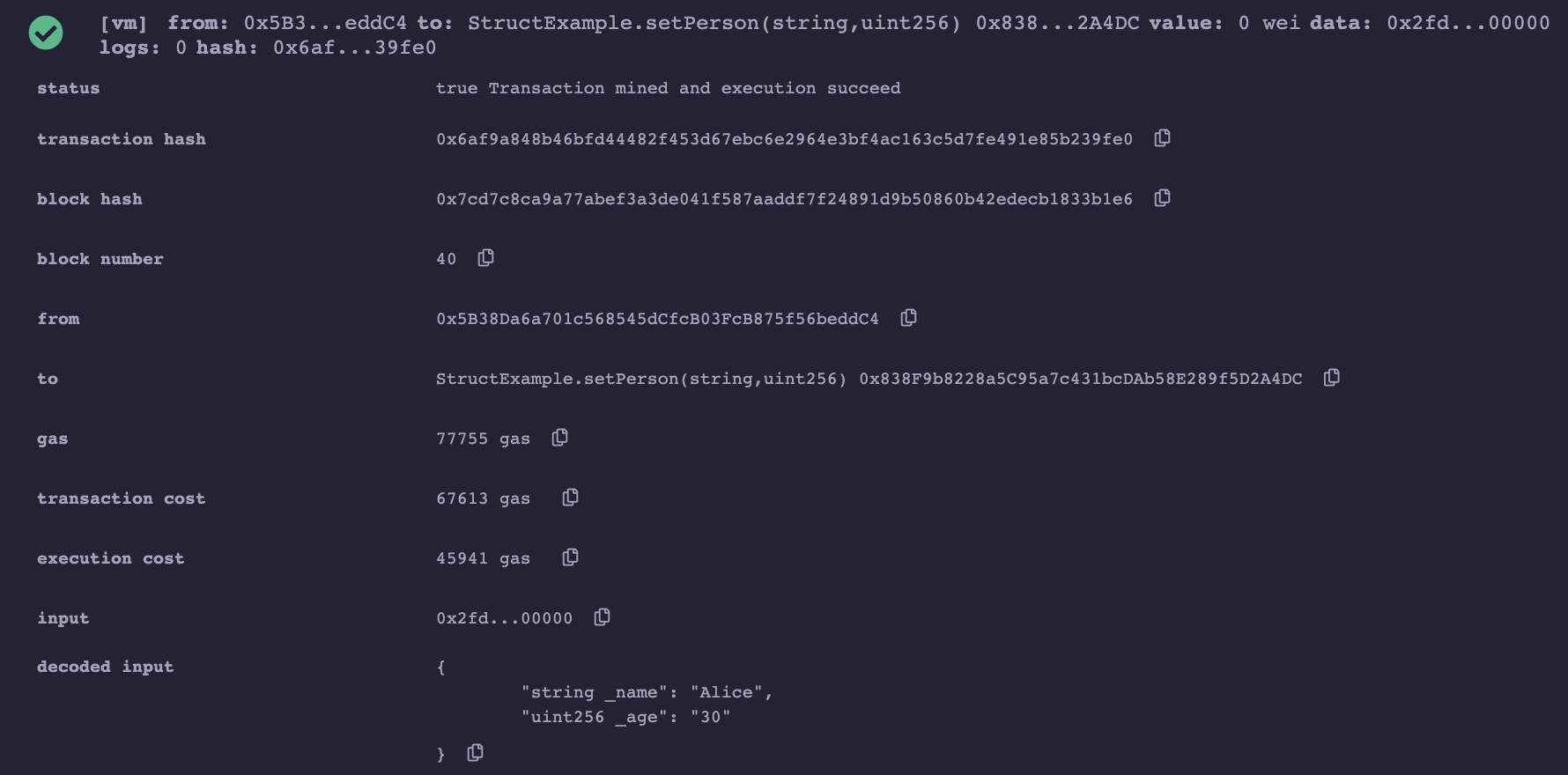
**PROCEDURE:**

**(STRUCTURES)**

****

**Deploying smart contract:**

****

**OUTPUT:  
  
  
**

****

**EXPERIMENT – 10**

**AIM :** To explore mathematical and cryptographic functions in Solidity.

**DESCRIPTION :**

1. **Mathematical Functions:**

Solidity provides inbuilt mathematical functions as well. Following are heavily used methods –

* addmod(uint x, uint y, uint k) returns (uint) − computes (x + y) % k where the addition is performed with arbitrary precision and does not wrap around at 2256.
* mulmod(uint x, uint y, uint k) returns (uint) − computes (x \* y) % k where the addition is performed with arbitrary precision and does not wrap around at 2256.

1. **Cryptographic Functions:**

Solidity provides inbuilt cryptographic functions as well. Following are important methods –

* keccak256(bytes memory) returns (bytes32) − computes the Keccak-256 hash of the input.
* ripemd160(bytes memory) returns (bytes20) − compute RIPEMD-160 hash of the input.
* sha256(bytes memory) returns (bytes32) − computes the SHA-256 hash of the input.
* ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address) − recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature: r - first 32 bytes of signature; s: second 32 bytes of signature; v: final 1 byte of signature. This method returns an address.

**CODE-1:**

pragma solidity ^0.5.0;

contract Test {

function callAddMod() public pure returns(uint){

return addmod(4, 5, 3);

}

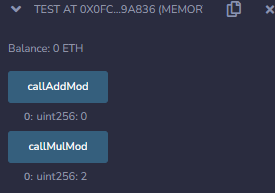
function callMulMod() public pure returns(uint){

return mulmod(4, 5, 3);

}

}

**OUTPUT-1:**



**CODE-2:**

**Keccak256**

pragma solidity ^0.5.0;

contract Test {

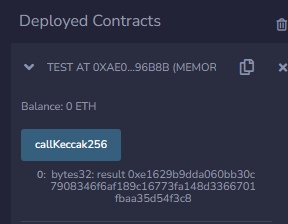
function callKeccak256() public pure returns(bytes32 result){

return keccak256("ABC");

}

}

**OUTPUT-2:**

****

**CODE-3:**

**SHA256**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract HashExample {

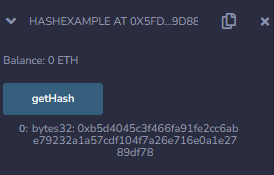
function getHash() public pure returns (bytes32) {

return sha256(“ABC”);

}

}

**OUTPUT-3:**

****

**EXPERIMENT – 11**

**AIM :** To implement smart contracts of Privacy and Security.

**DESCRIPTION :**

In the digital age, the quest for privacy and security has become paramount, leading to the advent of blockchain technology. This revolutionary system, known for its robust structure, offers a way to secure information exchange, validate transactions, and establish trust between parties in a decentralized manner. The concept of blockchain extends beyond cryptocurrencies like Bitcoin or Ethereum and permeates various sectors, including finance, healthcare, supply chain, and government services, promising a transparent, immutable, and secure environment.

**PROCEDURE :**

Creating a smart contract that incorporates both privacy and security features can be quite complex, especially since public blockchains are transparent by nature. However, we can provide a simple example of a smart contract written in Solidity (the most common language for Ethereum smart contracts) that demonstrates some basic security principles. For true privacy, advanced techniques and specific blockchain environments are needed, which are beyond the scope of this example.

Below is a basic example of a secure smart contract. This contract doesn't ensure privacy in the sense of hidden transactions, but it does demonstrate important security concepts like access control and state consistency.

**CODE :**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Importing OpenZeppelin's SafeMath library to prevent integer overflow and underflow

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

// Importing OpenZeppelin's Ownable contract to restrict access to certain functions

import "@openzeppelin/contracts/access/Ownable.sol";

contract SecureVault is Ownable {

using SafeMath for uint256;

// State variables

mapping(address => uint256) private \_balances;

// Events

event Deposited(address indexed user, uint256 amount);

event Withdrawn(address indexed user, uint256 amount);

// Modifier to check the condition before executing the function

modifier hasSufficientBalance(uint256 amount) {

require(\_balances[msg.sender] >= amount, "Insufficient balance");

\_;

}

// Function to deposit Ether into the contract

function deposit() external payable {

require(msg.value > 0, "Must send some ether");

\_balances[msg.sender] = \_balances[msg.sender].add(msg.value);

emit Deposited(msg.sender, msg.value);

}

// Function to withdraw Ether from the contract

function withdraw(uint256 amount) external hasSufficientBalance(amount) {

\_balances[msg.sender] = \_balances[msg.sender].sub(amount);

payable(msg.sender).transfer(amount);

emit Withdrawn(msg.sender, amount);

}

// Function to check the Ether balance of a user in the contract

function balanceOf(address user) external view returns (uint256) {

return \_balances[user];

}

// Secure function to withdraw all funds from the contract, accessible only by the owner

function emergencyWithdraw() external onlyOwner {

payable(owner()).transfer(address(this).balance);

}

}

**OUTPUT :**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated