Practical Journal

**BLOCKCHAIN**

**DEEP LEARNING**

**ROBOTIC PROCESS AUTOMATION**

**A Practical Report**

Submitted in partial fulfilment of the Requirements for the award of the Degree of

**MASTER OF SCIENCE (INFORMATION TECHNOLOGY)**

**Part II – SEM IV**

**Submitted by**

**NISHI JAIN – 53004230036**

****

**Department of Information Technology**

**Shri Vile Parle Kelavani Mandal’s**

**USHA PRAVIN GANDHI COLLEGE OF ARTS, SCIENCE AND COMMERCE**

*(Autonomous Affiliated to University of Mumbai)*

**NAAC Reaccredited ‘A+’ Grade**

**MUMBAI, 400056**

**2024-25**

**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr.No.** | | **Topic** | **Date** | **Pg.No.** | **Sign** |
| 1 | | Write a simple client class that generates the private and public keys by using the built-in Python RSA algorithm and test it. | 02/12/24 | 3 |  |
| 2 | | Setting up Ethereum network by using Geth command line interface (INSTALLATION). | 09/12/24 | 4 |  |
| 3 | | Transfer ethers from one contract to another on an Ethereum testnet. | 16/12/24 | 10 |  |
| 4 | | Transfer ethers from one account to another on an Ethereum testnet. | 02/01/25 | 11 |  |
| 5 |  | Implement and demonstrate the use of the following in Solidity: |  | 15 |  |
| a | Variable, Operators, Loops, Decision Making, Strings, Arrays, Enums, Structs, Mappings, Conversions, Ether Units, Special Variables. | 09/01/25 |  |  |
| b | Functions, Function Modifiers, View functions, Pure Functions, Fallback Function, Function Overloading, Mathematical functions, Cryptographic functions. | 16/01/25 |  |  |
| 6 |  | Implement and demonstrate the use of the following in Solidity. |  | 51 |  |
| a | Withdrawal Pattern, Restricted Access. | 23/01/25 |  |  |
| b | Contracts, inheritance, Constructors, Abstract Contracts, Interfaces. | 30/01/25 |  |  |
| c | Libraries, Assembly, Events, Error handling. | 06/02/25 |  |  |
| 7 | | Deploying a contract on an external blockchain by using Ganache and/or MyEtherwalllet, Metamask. | 13/02/25 | 72 |  |
| 8 | | Deploy a local private blockchain over a network with Ethereum or Rust (VM). | 20/02/25 | 74 |  |
| 9 | | Implement the mining module of Bitcoin client. The mining module, or miner, should produce blocks that solve proof-of-work puzzle. | 24/02/25 | 78 |  |
| 10 | | Compile and test smart contracts on a testing framework using the Ethereum Virtual Machine (EVM). | 27/02/25 | 79 |  |
| 11 | | Demonstrate the use of Bitcoin Core API. | 03/03/25 | 80 |  |
| 12 | | Create your own blockchain and demonstrate its use. | 06/03/25 | 87 |  |

# PRACTICAL 1

## Write a simple client class that generates the private and public keys by using the built-in Python RSA algorithm and test it.

**Code:**

import hashlib

import random

import binascii

import datetime

import collections

from Crypto.PublicKey import RSA

from Crypto import Random

from Crypto.Cipher import PKCS1\_v1\_5

class Client:

def \_\_init\_\_(self):

random = Random.new().read

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format='DER')).decode('ascii')

Person = Client()

print ("sender ",Person.identity)

**#alternate**

!pip install pycryptodome

import hashlib

import random

import binascii

import datetime

import collections

import Cryptodome

from Cryptodome.PublicKey import RSA

from Cryptodome import Random

from Cryptodome.Cipher import PKCS1\_v1\_5

class Client:

def \_\_init\_\_(self):

random = Random.new().read

self.\_private\_key = RSA.generate(1024, random)

self.\_public\_key = self.\_private\_key.publickey()

self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

@property

def identity(self):

return binascii.hexlify(self.\_public\_key.exportKey(format='DER')).decode('ascii')

Person = Client()

print ("sender ",Person.identity)

**Output:**



**PRACTICAL 2**

**Aim: Setting up Ethereum network by using Geth command line interface.(INSTALLATION)**

**Code:**

Install on Ubuntu via PPAs

The easiest way to install go-ethereum on Ubuntu-based distributions is with the built-in launchpad PPAs (Personal Package Archives). We provide a single PPA repository that contains both our stable and development releases for Ubuntu versions trusty, xenial, zesty and artful.

linux:

To enable our launchpad repository run:

Step 1: open new terminal

Step 2: on terminal type this command

sudo add-apt-repository -y ppa:ethereum/ethereum

#if above command gives error then run

#sudo apt-get install --reinstall ca-certificates

Step 3: install the stable version of go-ethereum:

sudo apt-get update

sudo apt-get install ethereum

linux:

To enable our launchpad repository run:

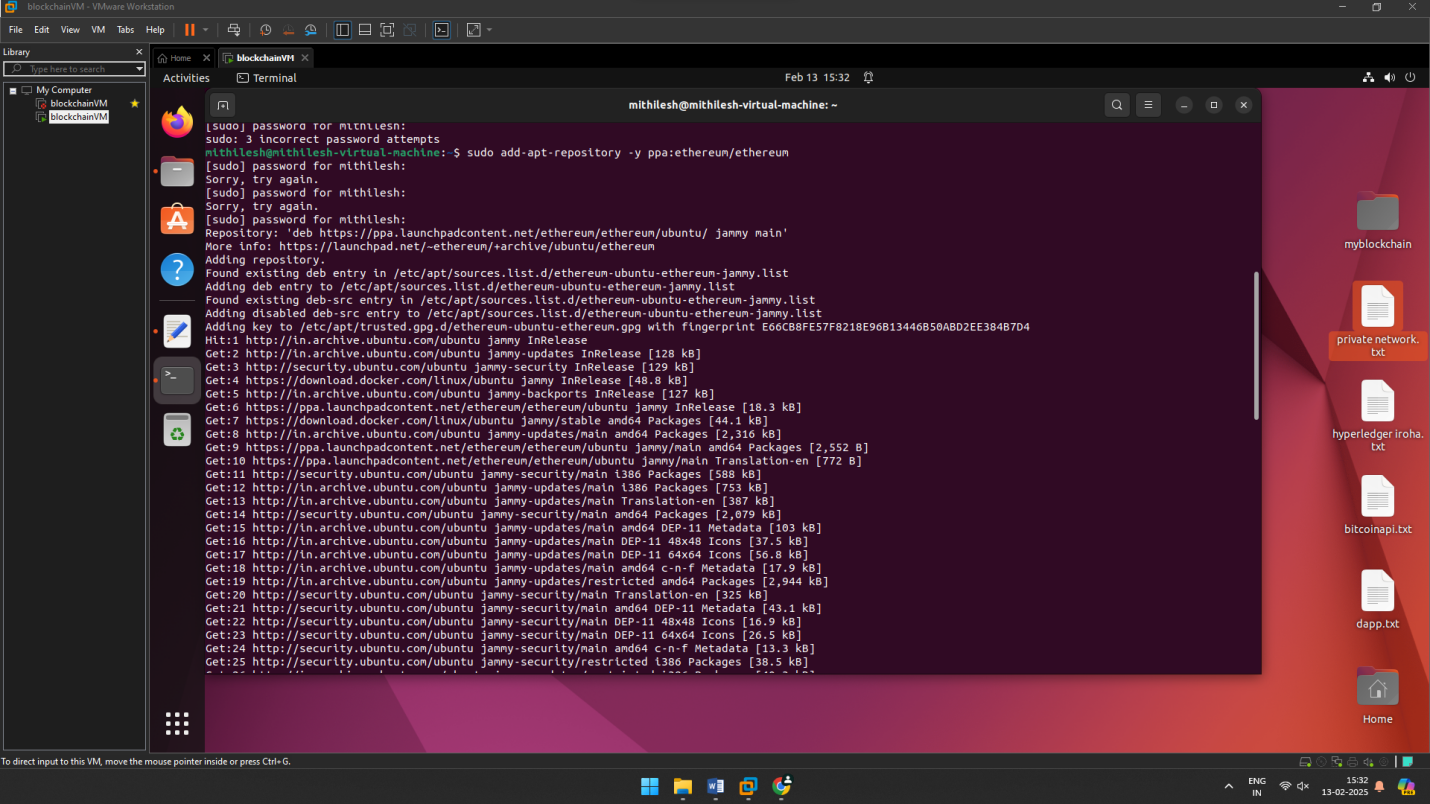
Step 1: open new terminal

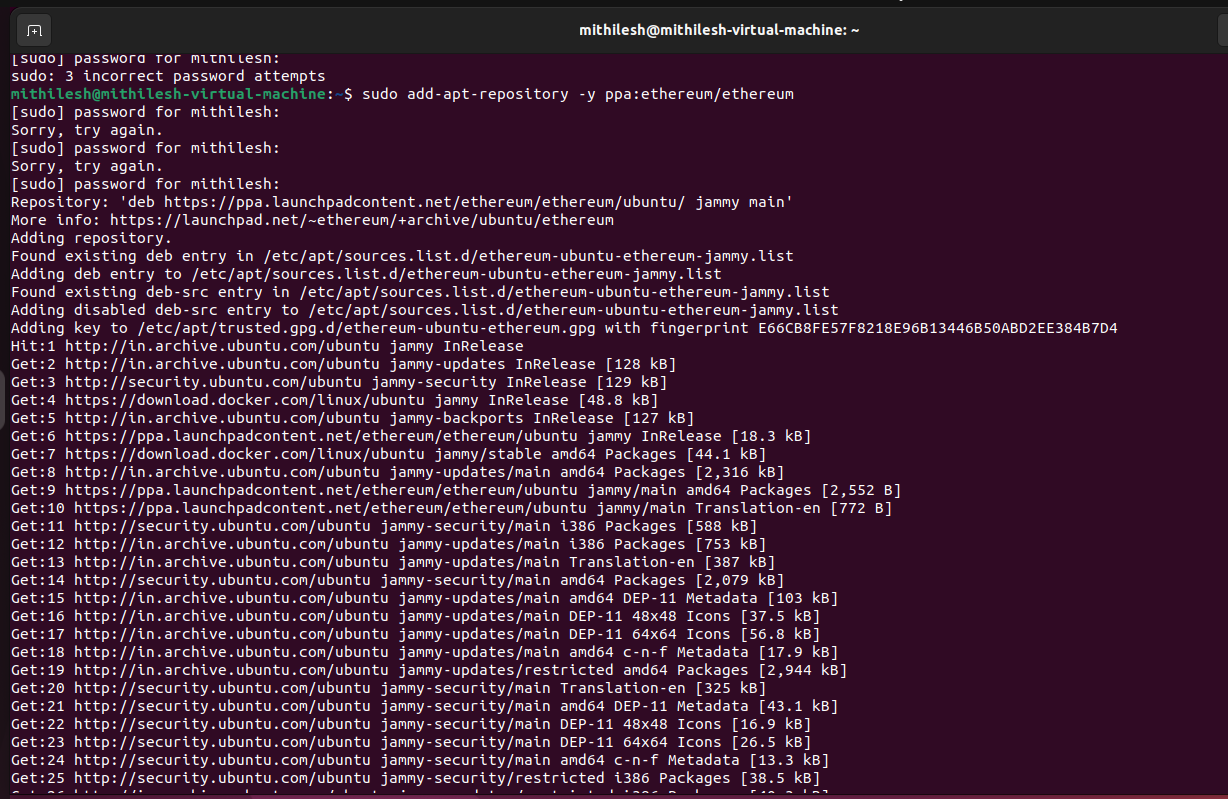
Step 2: on terminal type this command

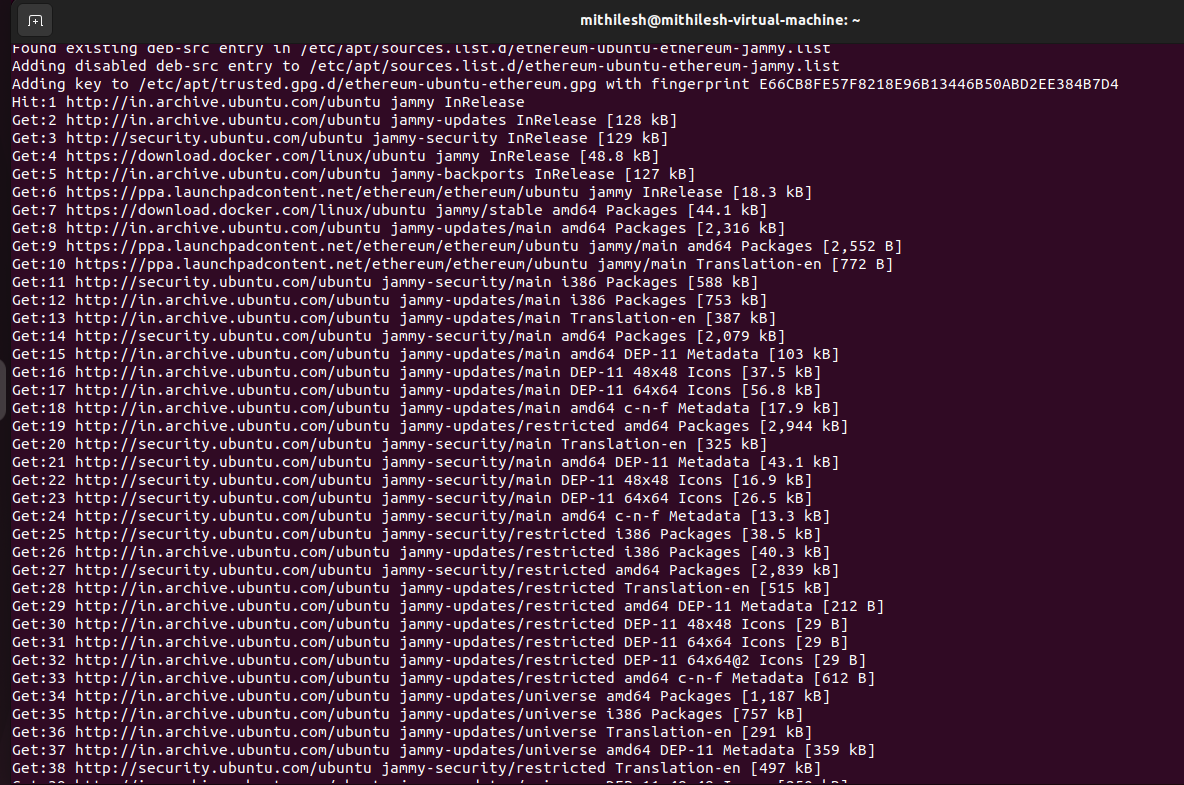
sudo add-apt-repository -y ppa:ethereum/ethereum

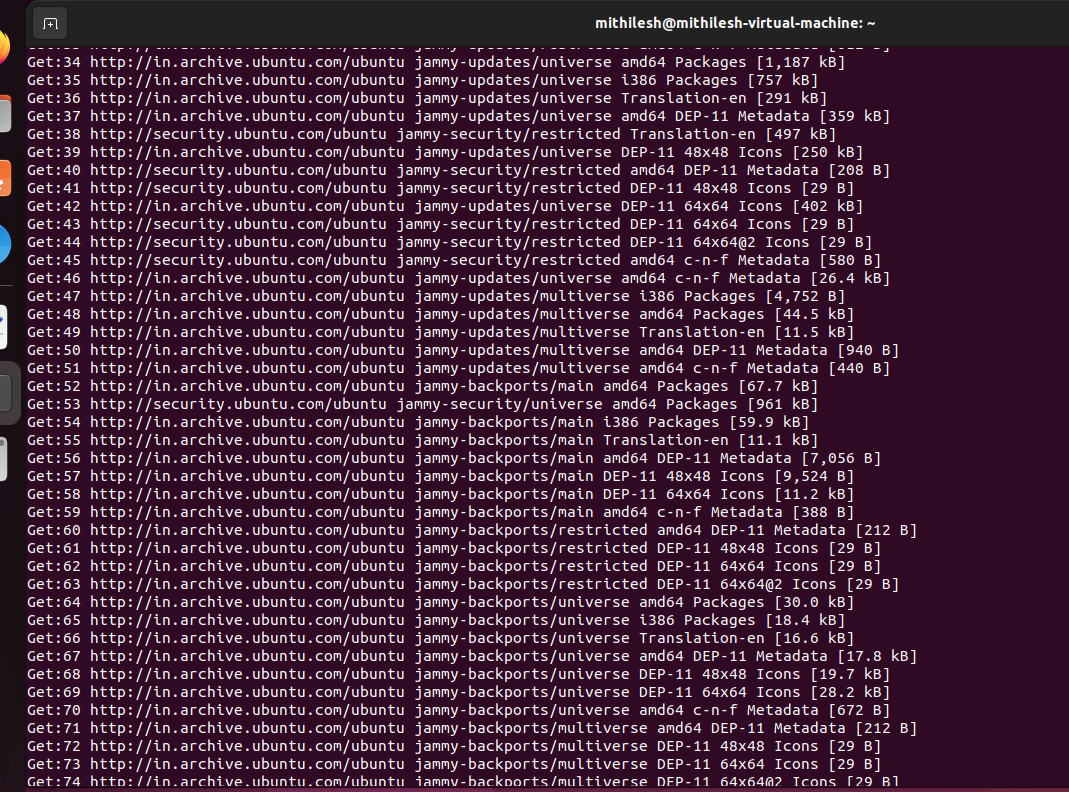
#if above command gives error then run

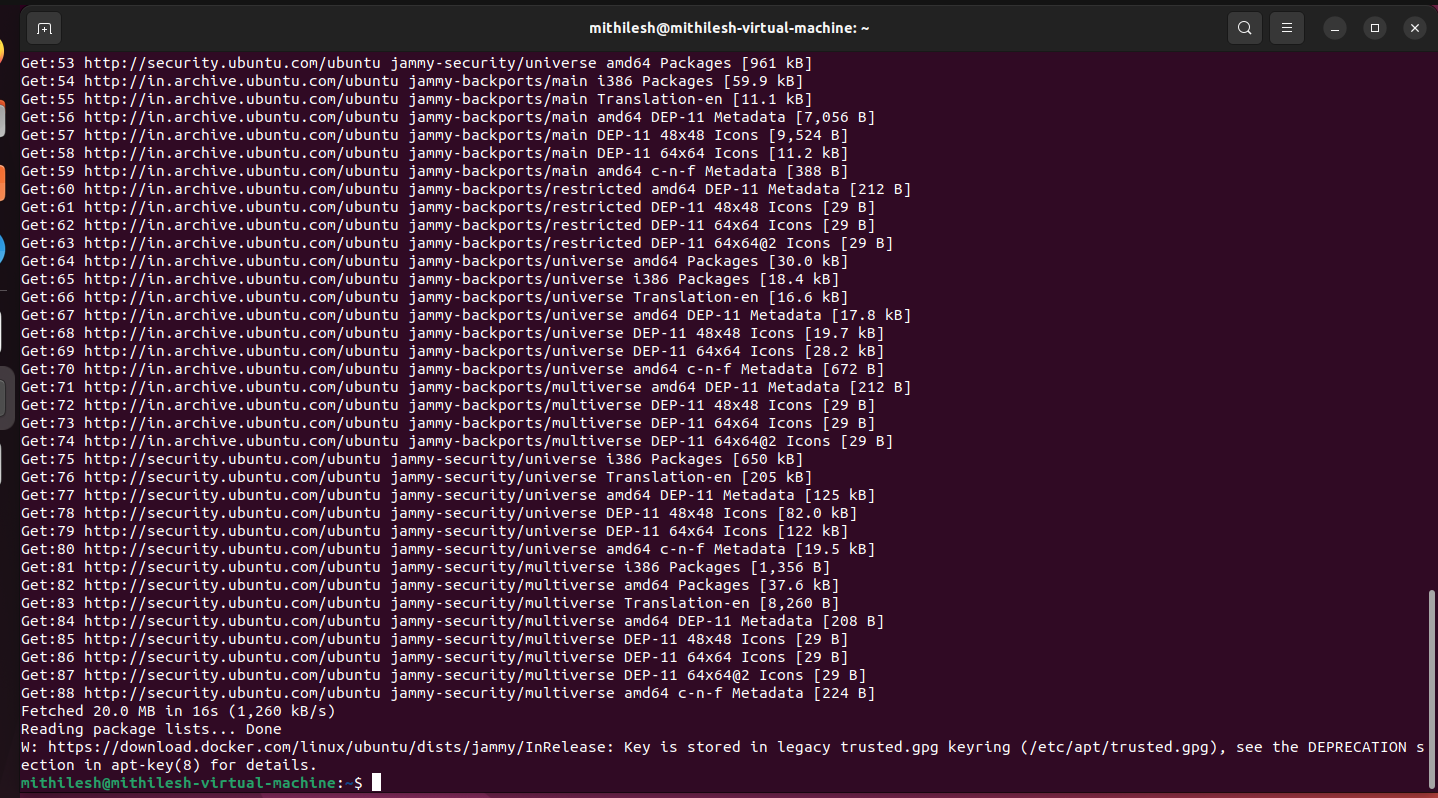
#sudo apt-get install --reinstall ca-certificates









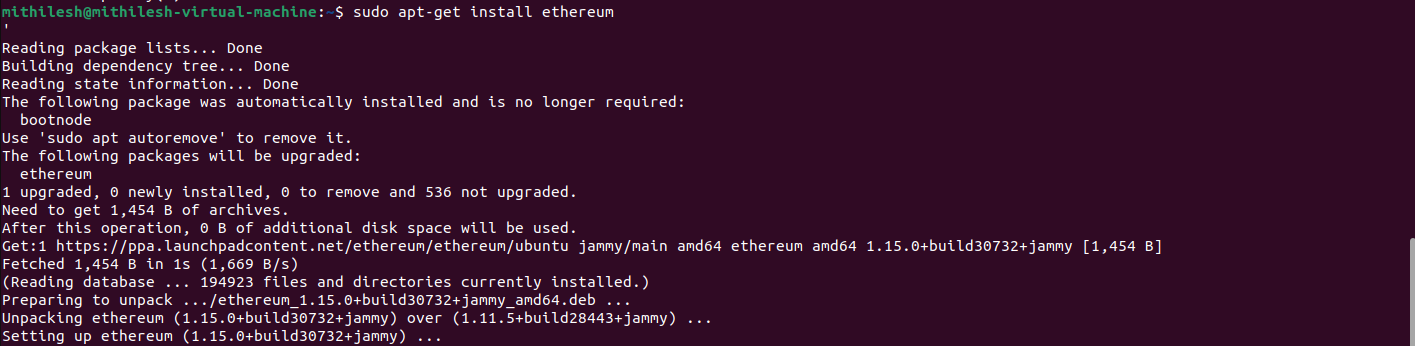


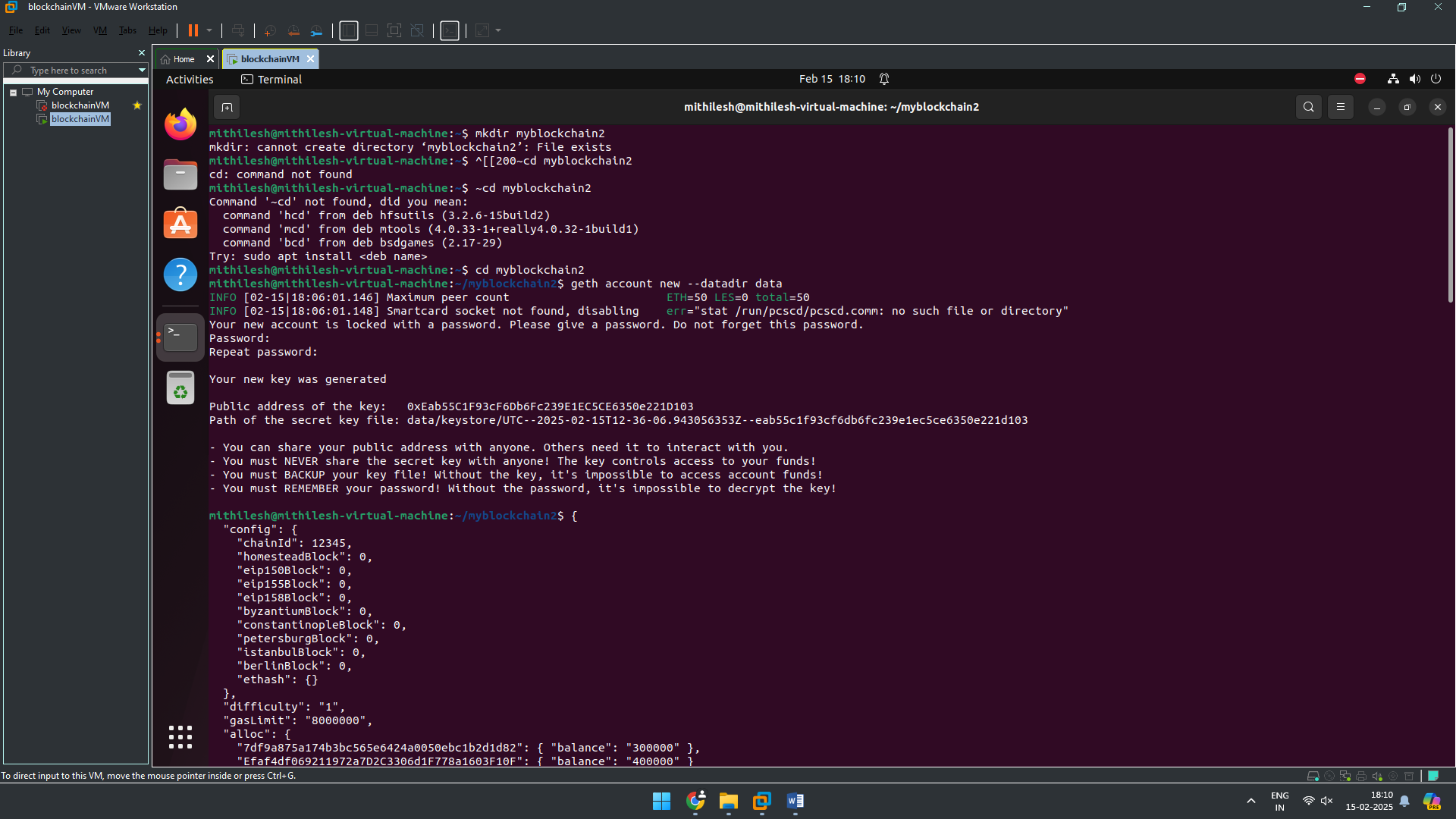
Step 3: install the stable version of go-ethereum:

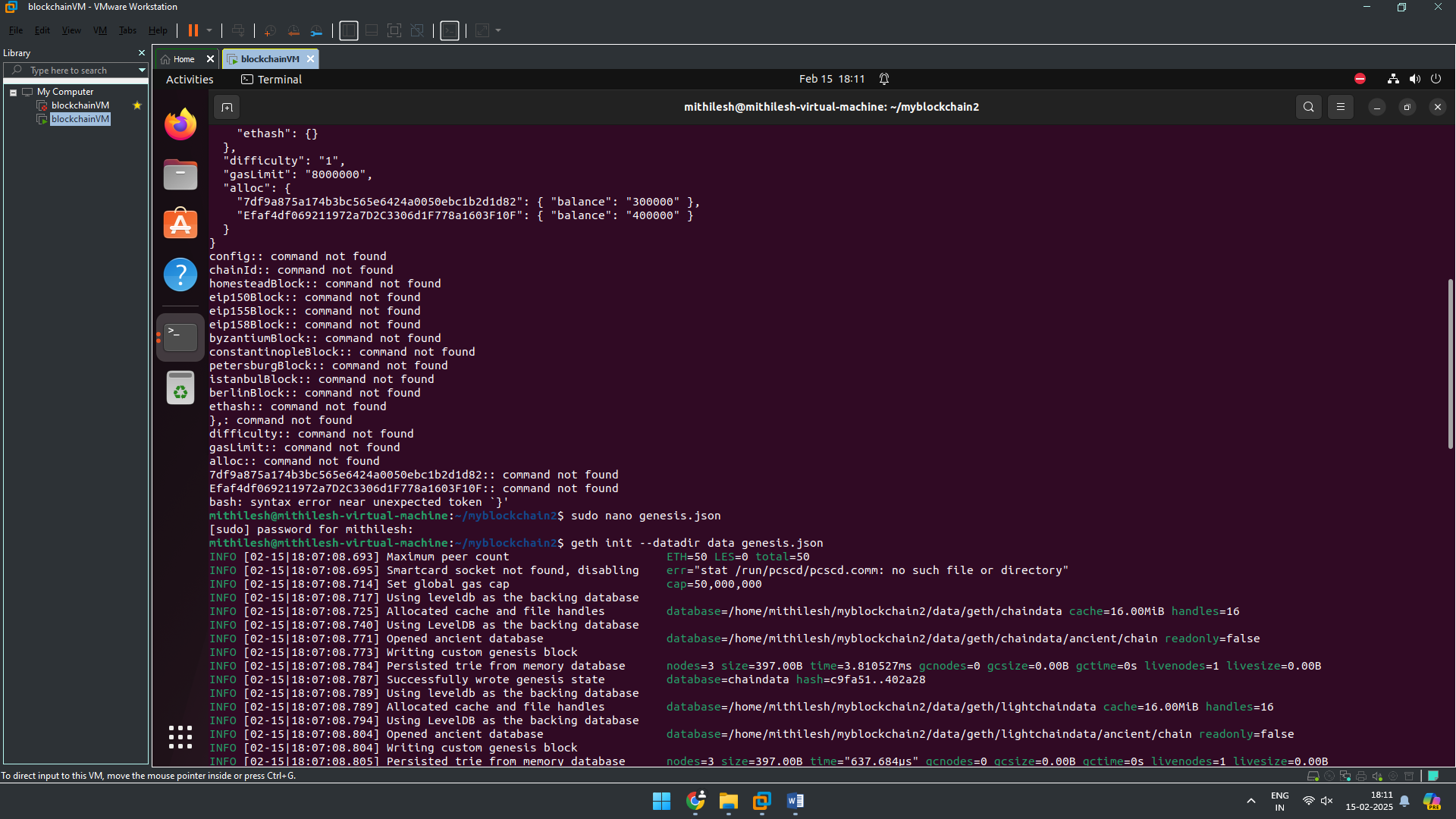
sudo apt-get update



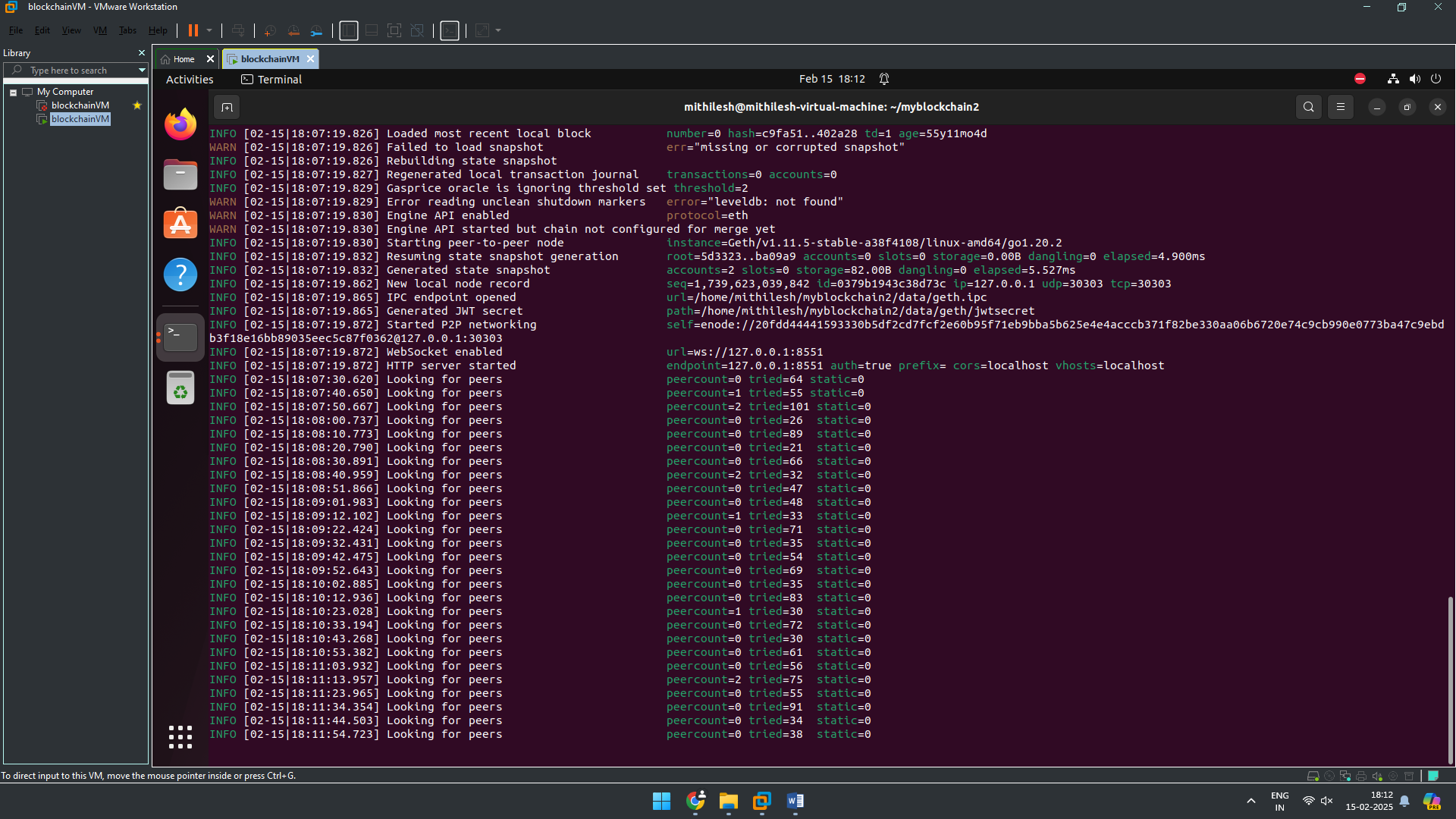
sudo apt-get install ethereum





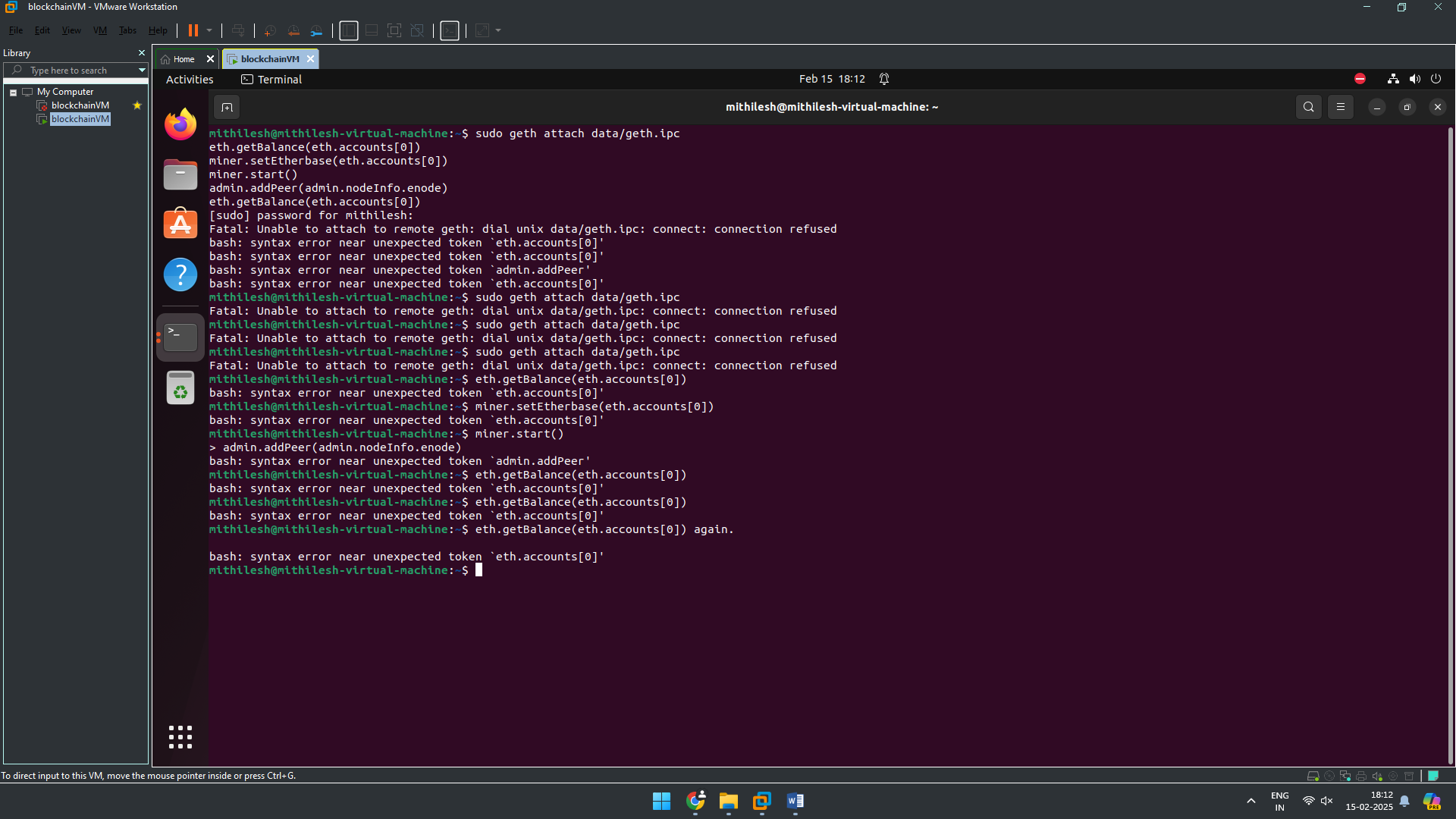






Terminal 2 :

**Output:-**



# PRACTICAL 3

## Transfer ethers from one **contract** to another on an Ethereum testnet.

**Code:**

pragma solidity ^0.8.0;

contract sendEther{

function getBalance() external view returns(uint)

{

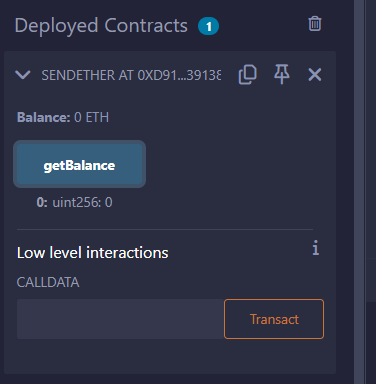
return address(this).balance;

}

receive() external payable { }

}

**Output:**



# PRACTICAL 4

## Transfer ethers from one **account** to another on an Ethereum testnet.

**Code:**

//https://dev.to/sparklesix/solidity-tutorial-how-to-build-and-deploy-a-smart-contract-to-send-ether-from-one-account-to-another-n54

pragma solidity ^0.8.11;

**1)REMIX**

contract Sender {

uint public amount;

address payable owner;

constructor (){

owner = payable(msg.sender); // set the deployer of contract as the owner

}

function sendEth(address payable receiver) payable public{

require(owner == msg.sender, "Only the owner can send funds");

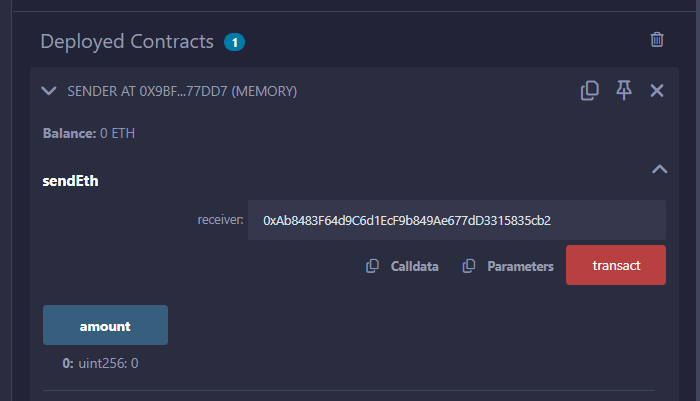
amount = msg.value;

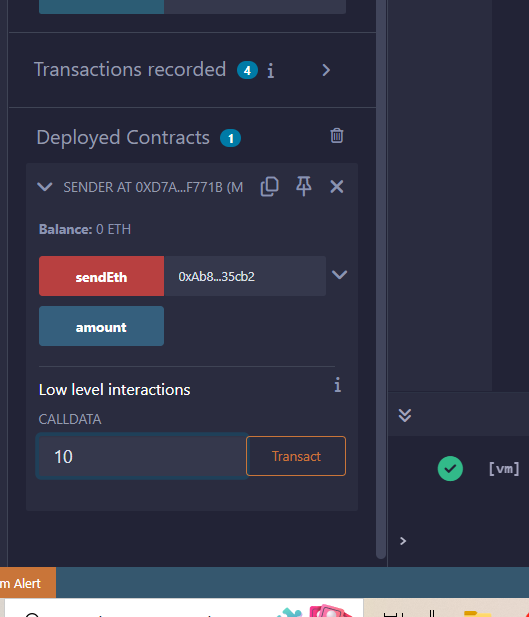
receiver.transfer(amount);

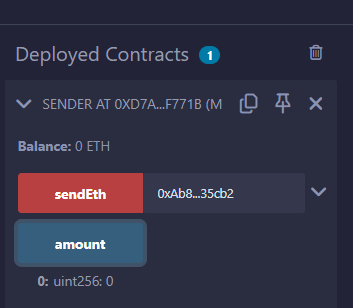
}

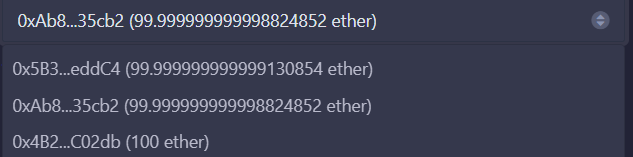
}

**Output:**









* + 1. Transfer ethers from one **account** to another on an Ethereum testnet.

pragma solidity ^0.8.11;

contract Sender {

uint public amount;

address payable owner;

constructor (){

owner = payable(msg.sender); // set the deployer of contract as the owner

}

function sendEth(address payable receiver) payable public{

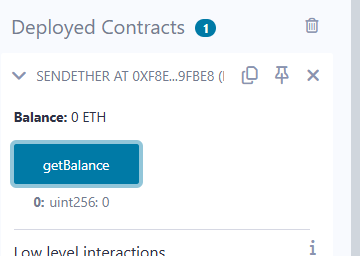
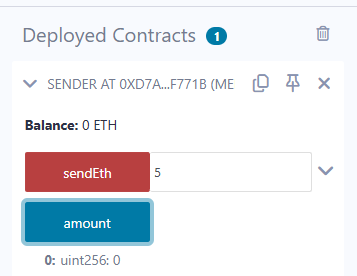
require(owner == msg.sender, "Only the owner can send funds");

amount = msg.value;

receiver.transfer(amount);

}

}



**2)Ganach**

https://abhibvp003.medium.com/how-to-install-and-execute-truffle-on-an-ubuntu-16-04-7d0ff6458c9b

https://ethereum.stackexchange.com/questions/93533/call-an-existing-contract-function-from-truffle-console

sudo apt-get -y install curl git vim build-essential

sudo apt-get install curl software-properties-common

sudo apt install npm

sudo npm install -g web3

sudo apt-get install nodejs

sudo apt install python3.9

curl -sL https://deb.nodesource.com/setup\_10.x | sudo bash -

sudo npm install --global node-sass@latest

sudo npm install -g truffle@latest

sudo npm install -g ganache-cli

export NODE\_OPTIONS=--openssl-legacy-provider

////to update npm//

sudo npm cache clean -f

sudo npm install -g n

sudo n latest

//////////////////////////////////////

Start from here!!!

mkdir upg1

cd upg1

truffle init

////////// create contract

nano contracts/HelloWorld.sol

pragma solidity ^0.5.0;

contract HelloWorld {

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

return("hello world");

}

}

////////////////////create configuration

nano migrations/1\_initial\_migration.js

const Migrations = artifacts.require("HelloWorld");

module.exports = function (deployer) {

deployer.deploy(Migrations,"hello");

};

////////////////network configuration

nano truffle-config.js

module.exports = {

networks: {

development: {

host: "127.0.0.1",

port: 8545,

network\_id: "\*",

}

}

}

////////////////////start ganache-cli

ganache-cli

////////////////

**3)truffle migrate**

truffle console

#replace contact address

contract = await HelloWorld.at('0x37354B83aadd35516c56f24b724228f29300be77')

a = await contract.sayHello()

a

* + 1. Transfer ethers from one **contract** to another on an Ethereum testnet.

pragma solidity ^0.8.11;

contract sendEther{

function getBalance() external view returns(uint)

{

    return address(this).balance;

}

receive() external payable { }

}

**PRACTICAL 5**

5)Implement and demonstrate the use of the following in Solidity:

# PRACTICAL 5a

## Variable, Operators, Loops, Decision Making, Strings, Arrays, Enums, Structs, Mappings, Conversions, Ether Units, Special Variables

**Code:-**

**A)Variables:**

supports three types of variables.

**State Variables** − Variables whose values are permanently stored in a contract storage.

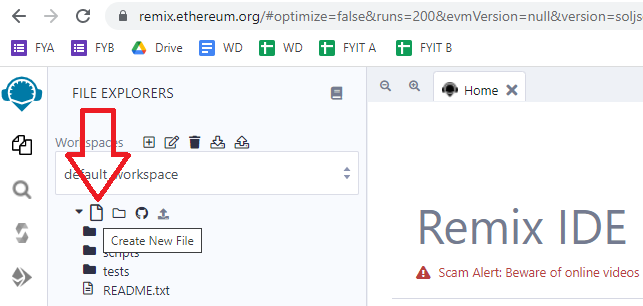
**Local Variables** − Variables whose values are present till function is executing.

**Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.i.e. blockhash(uint blockNumber) returns (bytes32), block.coinbase (address payable), block.difficulty (uint)…..and many more

Step 1: Open this website

<https://remix.ethereum.org/>

Step 2: Create new file – practical.sol



Step 3:Write the below program in new file

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;

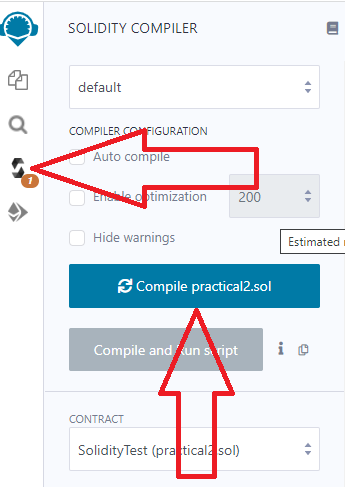
uint result = a + b;

return result; //access the state variable

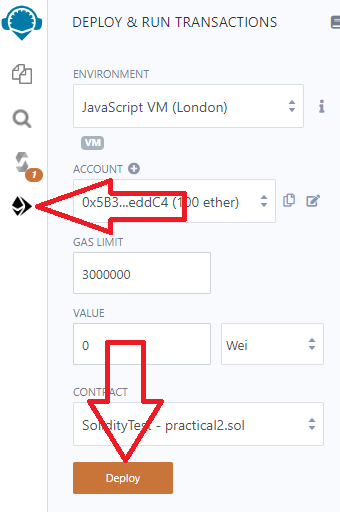
}

}

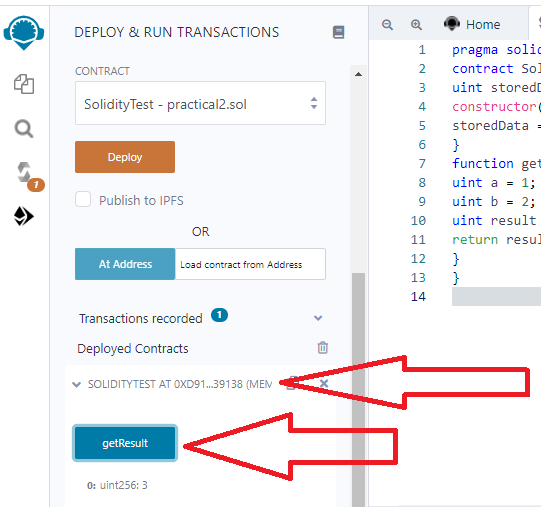
Step 4: Compile contract

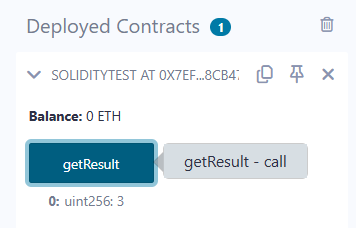


Step 5: Deploy contract



Step 6: Select the contract and click button





**1.State Variable:**

// Solidity program to

// demonstrate state

// variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

// Declaring a state variable

uint8 public state\_var;

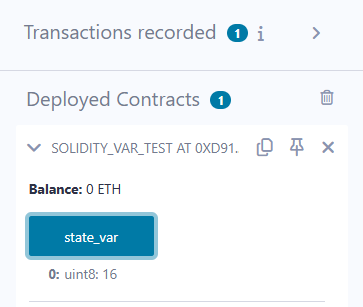
// Defining a constructor

constructor() public {

state\_var = 16;

}

}



**2.Local Variable:**

// Solidity program to demonstrate

// local variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

// Defining function to show the declaration and

// scope of local variables

function getResult() public view returns(uint){

// Initializing local variables

uint local\_var1 = 1;

uint local\_var2 = 2;

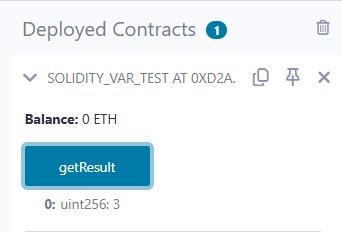
uint result = local\_var1 + local\_var2;

// Access the local variable

return result;

}

}



1. **Global variable:**

// Solidity program to

// show Global variables

pragma solidity ^0.5.0;

// Creating a contract

contract Test {

// Defining a variable

address public admin;

// Creating a constructor to

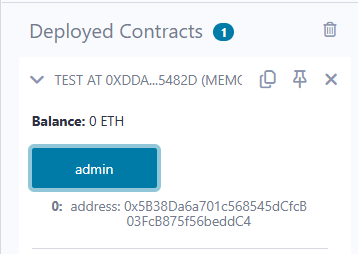
// use Global variable

constructor() public {

admin = msg.sender;

}

}



Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes.

**Public** − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.

**Internal** − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.

**Private** − Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

**B)Operators**

Solidity supports the following types of operators.

Arithmetic Operators

Comparison Operators

Logical (or Relational) Operators

Assignment Operators

Conditional (or ternary) Operators

1. **Arithematic Operator**

// Solidity contract to demonstrate

// Arithematic Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Initializing variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// with sum

uint public sum = a + b;

// Initializing a variable

// with the difference

uint public diff = a - b;

// Initializing a variable

// with product

uint public mul = a \* b;

// Initializing a variable

// with quotient

uint public div = a / b;

// Initializing a variable

// with modulus

uint public mod = a % b;

// Initializing a variable

// decrement value

uint public dec = --b;

// Initializing a variable

// with increment value

uint public inc = ++a;

}



**2.Relational Operator**

// Solidity program to demonstrate

// Relational Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// with bool equal result

bool public eq = a == b;

// Initializing a variable

// with bool not equal result

bool public noteq = a != b;

// Initializing a variable

// with bool greater than result

bool public gtr = a > b;

// Initializing a variable

// with bool less than result

bool public les = a < b;

// Initializing a variable

// with bool greater than equal to result

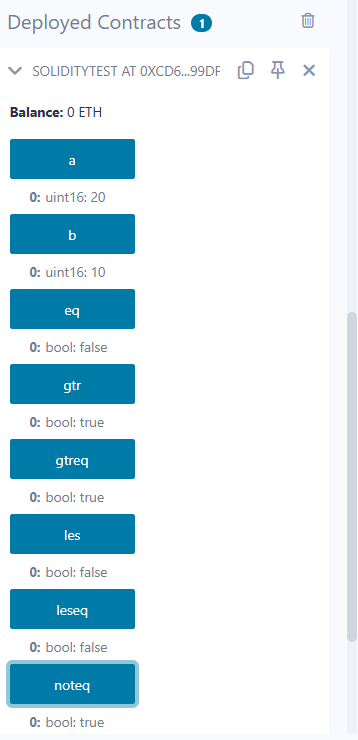
bool public gtreq = a >= b;

// Initializing a variable

// bool less than equal to result

bool public leseq = a <= b;

}



**3.Logical Operators**

// Solidity program to demonstrate

// Logical Operators

pragma solidity ^0.5.0;

// Creating a contract

contract logicalOperator{

// Defining function to demonstrate

// Logical operator

function Logic(

bool a, bool b) public view returns(

bool, bool, bool){

// Logical AND operator

bool and = a&&b;

// Logical OR operator

bool or = a||b;

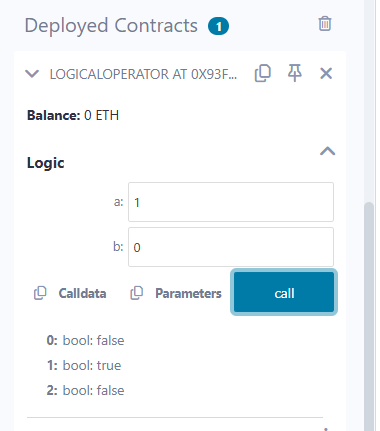
// Logical NOT operator

bool not = !a;

return (and, or, not);

}

}



**4.Bitwise Operators**

// Solidity program to demonstrate

// Bitwise Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// to '&' value

uint16 public and = a & b;

// Initializing a variable

// to '|' value

uint16 public or = a | b;

// Initializing a variable

// to '^' value

uint16 public xor = a ^ b;

// Initializing a variable

// to '<<' value

uint16 public leftshift = a << b;

// Initializing a variable

// to '>>' value

uint16 public rightshift = a >> b;

// Initializing a variable

// to '~' value

uint16 public not = ~a ;

}



**5.Assignment Operator**

// Solidity program to demonstrate

// Assignment Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public assignment = 20;

uint public assignment\_add = 50;

uint public assign\_sub = 50;

uint public assign\_mul = 10;

uint public assign\_div = 50;

uint public assign\_mod = 32;

// Defining function to

// demonstrate Assignment Operator

function getResult() public{

assignment\_add += 10;

assign\_sub -= 20;

assign\_mul \*= 10;

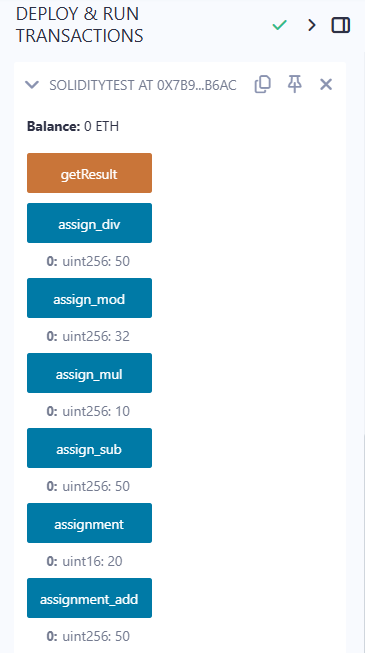
assign\_div /= 10;

assign\_mod %= 20;

return ;

}

}



6**.Conditional Operators**

// Solidity program to demonstrate

// Conditional Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest{

// Defining function to demonstrate

// conditional operator

function sub(

uint a, uint b) public view returns(

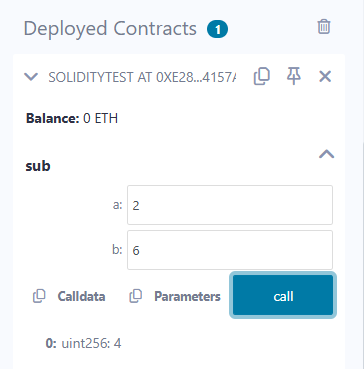
uint){

uint result = (a > b? a-b : b-a);

return result;

}

}



**C)Loops:**

1.While loop: The most basic loop in Solidity is the **while** loop which would be discussed in this chapter. The purpose of a **while** loop is to execute a statement or code block repeatedly as long as an **expression** is true. Once the expression becomes **false,** the loop terminates.

2.do-while loop: The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is **false**.

3.for loop: The **for** loop is the most compact form of looping. It includes the following three important parts −

The **loop initialization** where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.

The **test statement** which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.

The **iteration statement** where you can increase or decrease your counter.

4.loop control: Solidity provides full control to handle loops and switch statements. There may be a situation when you need to come out of a loop without reaching its bottom. There may also be a situation when you want to skip a part of your code block and start the next iteration of the loop.To handle all such situations, Solidity provides **break** and **continue** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

**1.While Loop**

pragma solidity ^0.5.0;

contract Pract3{

function test(int s, int e) public view returns(int)

{

int i;

int sum=0;

i=s;

while(i<=e)

{

sum+=i;  //sum=sum+i;

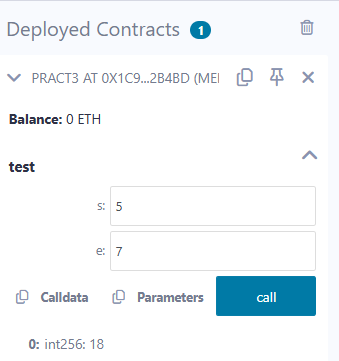
i++;

}

return sum;

}

}



**2.Do-while loop:**

pragma solidity ^0.5.0;

contract Pract3{

function test(int s, int e) public view returns(int)

{

int i;

int sum=0;

i=s;

do

{

sum+=i;  //sum=sum+i;

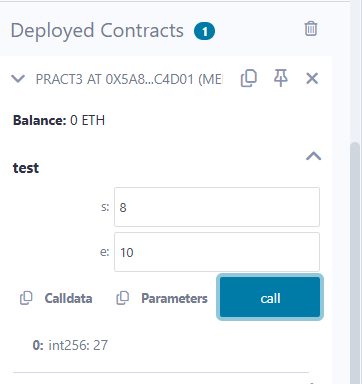
i++;

}while(i<=e);

return sum;

}

}



**3.For Loop:**

contract Pract3{

function test(int s, int e) public view returns(int)

{

int i;

int sum=0;

for(i=s;i<=e;i++)

{

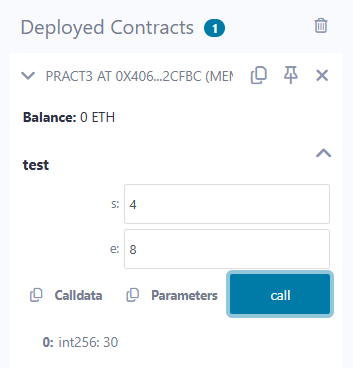
sum+=i;  //sum=sum+i;

}

return sum;

}

}



**4.loop Control: (Break statement)**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (true) {

len++;

j /= 10;

if(j==0){

break;   //using break statement

}

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);

}

}

**(continue statement)**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

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

uint n = 1;

uint sum = 0;

while( n < 10){

n++;

if(n == 5){

continue; // skip n in sum when it is 5.

}

sum = sum + n;

}

return integerToString(sum);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (true) {

len++;

j /= 10;

if(j==0){

break;   //using break statement

}

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);

}

}



**D) Decision Making:**

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.Solidity supports conditional statements which are used to perform different actions based on different conditions. Here we will explain the **if..else** statement.

1. if statement: The **if** statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally.

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public {

storedData = 10;

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {   // if statement

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

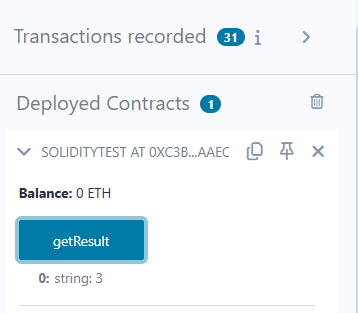
bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}}



**2.if-else statement:** The **'if...else'** statement is the next form of control statement that allows Solidity to execute statements in a more controlled way.

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 10;

bool even;

// Defining function to

// demonstrate the use of

// 'if...else statement'

function decision\_making(

) public payable returns(bool){

if (i%2 == 0){

even = true;

}

else{

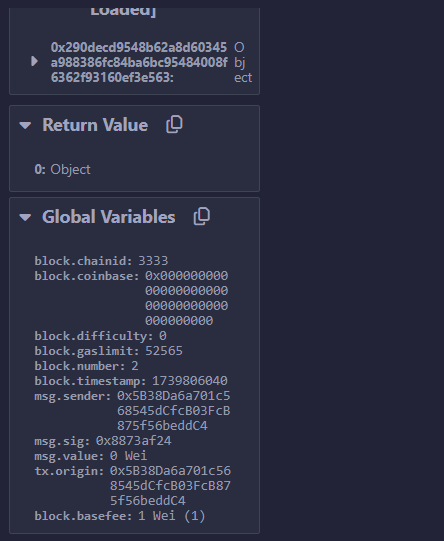
even = false;

}

return even;

}

}



**3.if-else..if statement**: The **if...else if...** statement is an advanced form of **if...else** that allows Solidity to make a correct decision out of several conditions.

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 12;

string result;

// Defining function to

// demonstrate the use

// of 'if...else if...else

// statement'

function decision\_making (

) public returns(string memory){

if(i<10){

result = "less than 10";

}

else if(i == 10){

result = "equal to 10";

}

else{

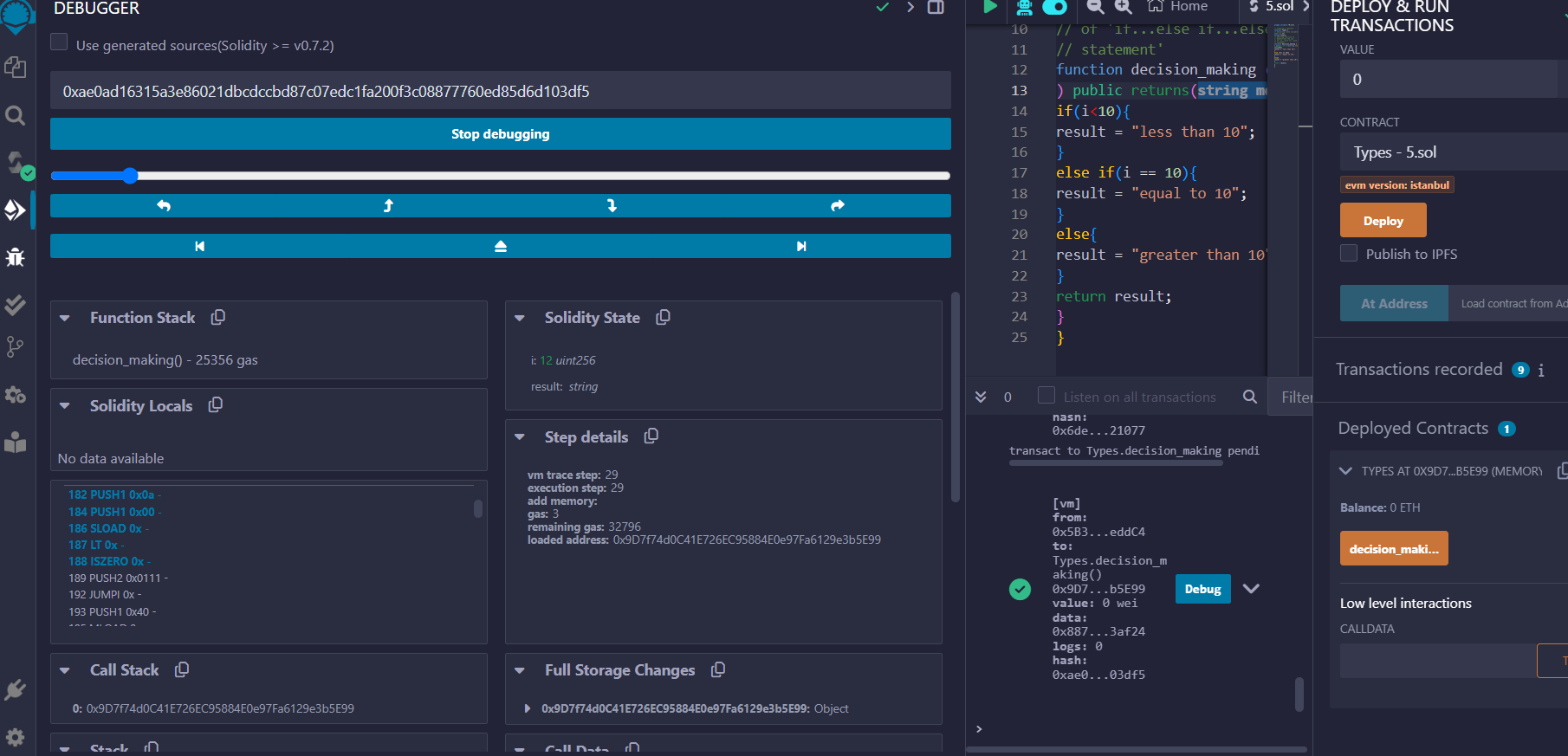
result = "greater than 10";

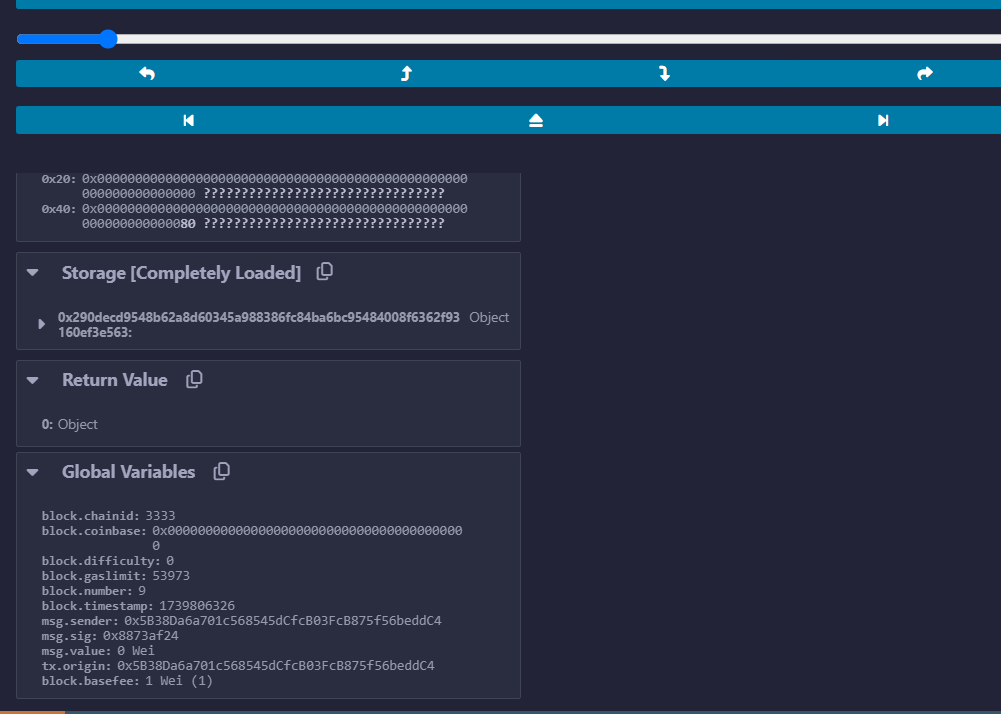
}

return result;

}

}





**B) String :PUBLIC FUNCTION**

Solidity supports String literal using both double quote (") and single quote ('). It provides string as a data type to declare a variable of type String.(Int to str)

pragma solidity ^0.5.0;

contract SolidityTest {

constructor() public{

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

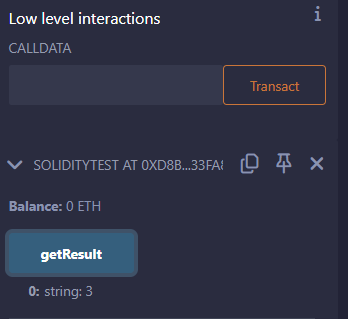
\_i /= 10;

}

return string(bstr);

}

}



**B)Array:**

Array is a data structure, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

// Solidity program to demonstrate

// accessing elements of an array

pragma solidity ^0.5.0;

function

// Creating a contract

contract Types {

// Declaring an array

uint[6] data;

uint x;

// Defining function to

// assign values to array

function array\_example() public returns (uint[6] memory)

{

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

}

function result() public view returns(uint[6] memory){

return data;

}

// Defining function to access

// values from the array

// from a specific index

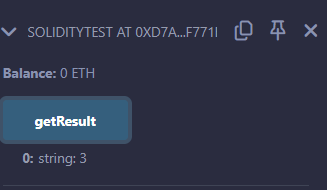
function array\_element() public view returns (uint){

uint x = data[2];

return x;

}

}



**C)Enums:**

Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums. With the use of enums it is possible to reduce the number of bugs in your code.

// Solidity program to demonstrate

// how to use 'enumerator'

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Creating an enumerator

enum week\_days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

// Declaring variables of

// type enumerator

week\_days week;

week\_days choice;

// Setting a default value

week\_days constant default\_value

= week\_days.Sunday;

// Defining a function to

// set value of choice

function set\_value() public {

choice = week\_days.Thursday;

}

// Defining a function to

// return value of choice

function get\_choice(

) public view returns (week\_days) {

return choice;

}

// Defining function to

// return default value

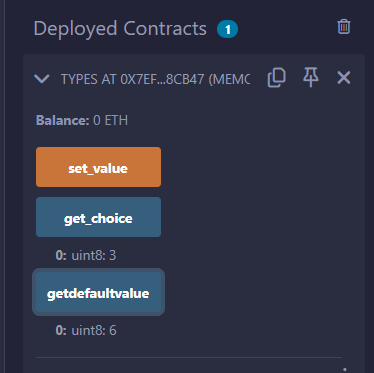
function getdefaultvalue(

) public pure returns(week\_days) {

return default\_value;

}

}



**D)Structure:**

Struct types are used to represent a record.

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', 1);

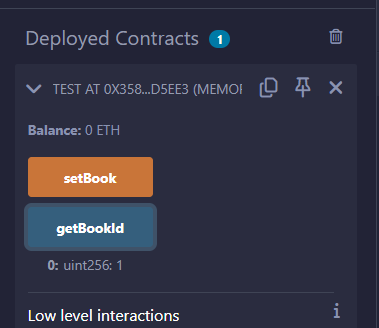
}

function getBookId() public view returns (uint) {

return book.book\_id;

}

}



**E)Mappings:**

Mapping is a reference type as arrays and structs. Following is the syntax to declare a mapping type.

mapping(\_KeyType => \_ValueType) where ,

**\_KeyType** − can be any built-in types plus bytes and string. No reference type or complex objects are allowed.

**\_ValueType** − can be any type.

pragma solidity ^0.5.0;

contract LedgerBalance {

mapping(address => uint)  balance;

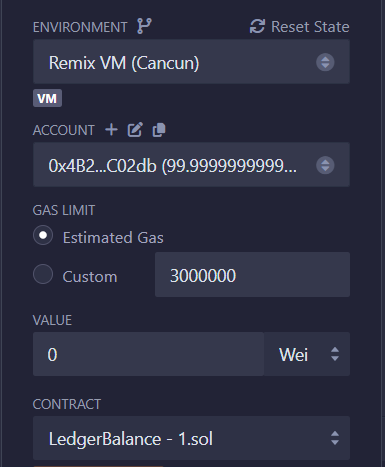
function updateBalance() public  returns(uint) {

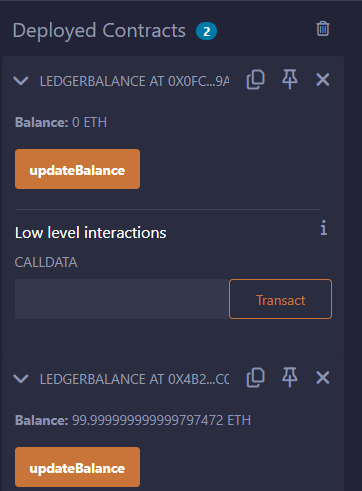
balance[msg.sender]=30;

return balance[msg.sender];

}

}





**Mapping program for String.**

pragma solidity ^0.5.0;

contract LedgerBalance {

mapping(address => string)  name;

function updateBalance() public returns(string memory){

name[msg.sender] = "Mrunali";

return name[msg.sender];

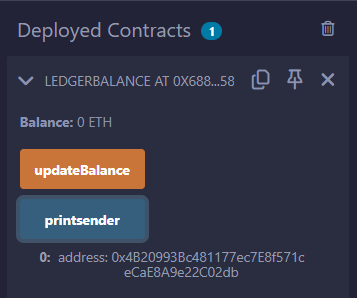
}

function printsender() public view returns(address) {

return msg.sender;

}

}



# PRACTICAL 5B

## WRITE A SOLIDITY PROGRAM FOR FUNCTION OVERLOADING, MATHEMATICAL FUNCTION & CRYPTOGRAPHIC FUNCTIONS VIEW FUNCTION, PURE FUNCTION & FALLBACK FUNCTION.

Functions, Function Modifiers, View functions, Pure Functions, Fallback Function, Function Overloading, Mathematical functions, Cryptographic functions.

**A)Function Overloading:**

The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

pragma solidity ^0.5.0;

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(2,2);

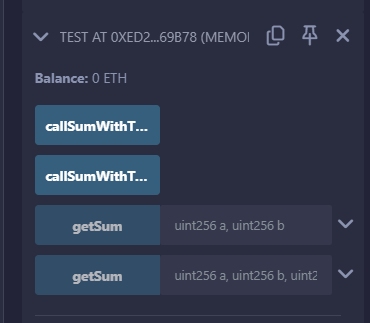
}

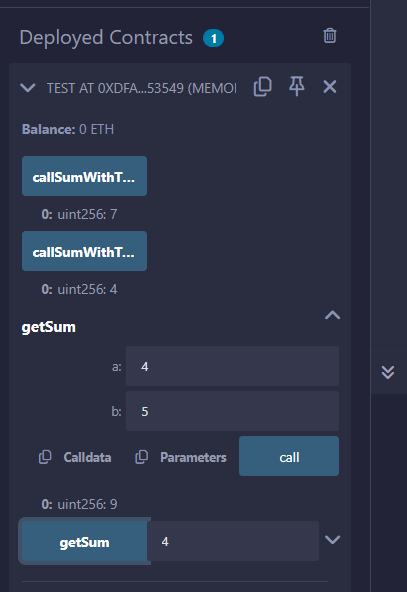
function callSumWithThreeArguments() public pure returns(uint){

return getSum(1,2,4);

}

}





**B)Mathematical Function:**

Solidity provides inbuilt mathematical functions as well.

pragma solidity ^0.5.0;

contract Test {

function callAddMod() public pure returns(uint){

return addmod(4, 5, 3);

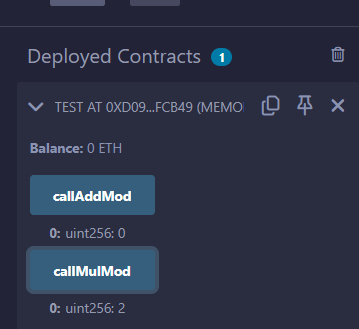
}

function callMulMod() public pure returns(uint){

return mulmod(4, 5, 3);

}

}



**Cryptographic Function:**

Solidity provides inbuilt cryptographic functions as well.

pragma solidity ^0.5.0;

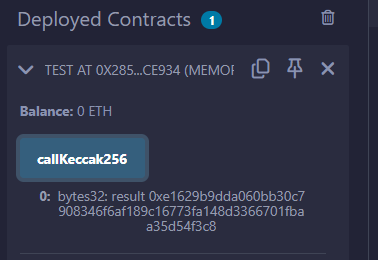
contract Test {

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

return keccak256("ABC");

}

}



**C)Function:**

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

pragma solidity ^0.5.0;

contract SolidityTest {

constructor() public{

}

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

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}

}



**D)View Function:**

View functions ensure that they will not modify the state. A function can be declared as **view**. Getter method are by default view functions.

// Solidity program to demonstrate

// how to create a contract

pragma solidity ^0.4.23;

// Creating a contract

contract Test {

// Declaring variable

string  str;

// Defining a constructor

constructor(string str\_in){

str = str\_in;

}

// Defining a function to

// return value of variable 'str'

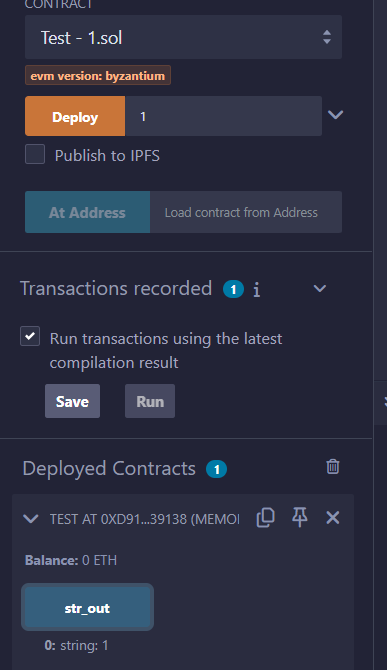
function str\_out() public view returns(string memory){

return str;

}

}

Note:  after deploy it asked u to enter string then enter string over there and then see the output after clicking on str\_out button



**E)Pure Function:**

Pure functions ensure that they not read or modify the state. A function can be declared as **pure**. Pure functions can use the revert() and require() functions to revert potential state changes if an error occurs.

pragma solidity ^0.5.0;

contract Test {

int public x=10; //global

int y=90;//state

function f1() public returns(int){

    //read and update is allowed

    x=100;

return x;

}

function f2() public view returns(int){

  //  x=100; //erro beacuse x is global/state

  //we can access but we cannot update state or global variable int view function

return x;

}

function f3() public pure returns(int){

    //we cannot access or update state or global variable in pure function

    int z=80;

return z;

}

}



**F)Fallback Function:**

Fallback function is a special function available to a contract.

pragma solidity ^0.5.0;

contract Test {

uint public x ;

function() external { x = 1; }

}

contract Sink {

function() external payable { }

}

contract Caller {

function callTest(Test test) public returns (bool) {

(bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));

require(success);

// test.x is now 1

address payable testPayable = address(uint160(address(test)));

// Sending ether to Test contract,

// the transfer will fail, i.e. this returns false here.

return (testPayable.send(2 ether));

}

function callSink(Sink sink) public returns (bool) {

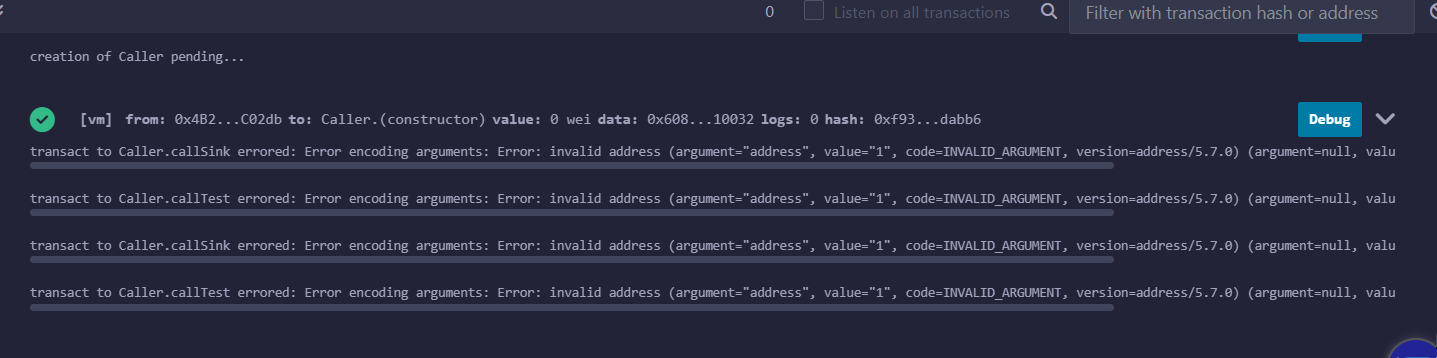
address payable sinkPayable = address(sink);

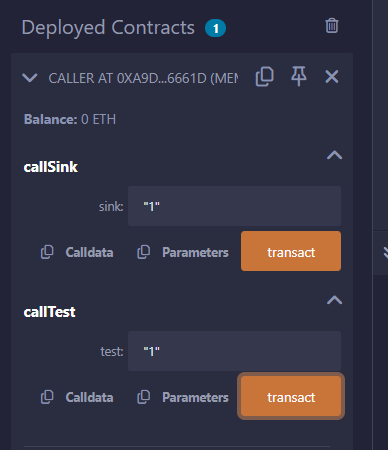
return (sinkPayable.send(2 ether));

}

}







**PRACTICAL 6**

**Aim**:-Implement and demonstrate the use of the following in Solidity.

# PRACTICAL 6a

## Withdrawal Pattern, Restricted Access.

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData; // State variable

uint public a=10;

constructor() public {

storedData = 10;

}

function getResult(uint c) public view returns(uint){

uint a = 1; // local variable

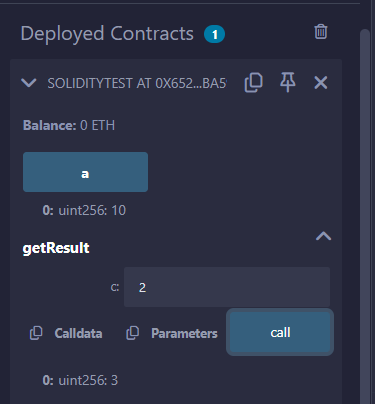
uint b = 2;

uint result = a + b;

return result; //access the state variable

}

}



Withdraw Pattern:-

The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending Ether, as a result of an effect, is a direct transfer call, this is not recommended as it introduces a potential security risk. You may read more about this on the [Security Considerations](https://docs.soliditylang.org/en/latest/security-considerations.html#security-considerations) page.

The following is an example of the withdrawal pattern in practice in a contract where the goal is to send the most of some compensation, e.g. Ether, to the contract in order to become the “richest”, inspired by [King of the Ether](https://www.kingoftheether.com/).

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.4;

contract SendContract {

    address payable public richest;

    uint public mostSent;

    /// The amount of Ether sent was not higher than

    /// the currently highest amount.

    error NotEnoughEther();

    constructor() payable {

        richest = payable(msg.sender);

        mostSent = msg.value;

    }

    function becomeRichest() public payable {

        if (msg.value <= mostSent) revert NotEnoughEther();

        // This line can cause problems (explained below).

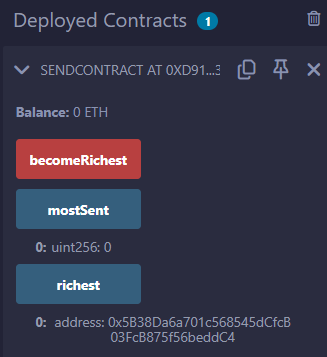
        richest.transfer(msg.value);

        richest = payable(msg.sender);

        mostSent = msg.value;

    }

}



Restricted Access:-

Restricting access is a common pattern for contracts. Note that you can never restrict any human or computer from reading the content of your transactions or your contract’s state. You can make it a bit harder by using encryption, but if your contract is supposed to read the data, so will everyone else.

You can restrict read access to your contract’s state by other contracts. That is actually the default unless you declare your state variables public.

Furthermore, you can restrict who can make modifications to your contract’s state or call your contract’s functions and this is what this section is about.

The use of function modifiers makes these restrictions highly readable.

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.4;

contract AccessRestriction {

    // These will be assigned at the construction

    // phase, where `msg.sender` is the account

    // creating this contract.

    address public owner = msg.sender;

    uint public creationTime = block.timestamp;

    // Now follows a list of errors that

    // this contract can generate together

    // with a textual explanation in special

    // comments.

    /// Sender not authorized for this

    /// operation.

    error Unauthorized();

    /// Function called too early.

    error TooEarly();

    /// Not enough Ether sent with function call.

    error NotEnoughEther();

    // Modifiers can be used to change

    // the body of a function.

    // If this modifier is used, it will

    // prepend a check that only passes

    // if the function is called from

    // a certain address.

    modifier onlyBy(address account)

    {

        if (msg.sender != account)

            revert Unauthorized();

        // Do not forget the "\_;"! It will

        // be replaced by the actual function

        // body when the modifier is used.

        \_;

    }

    /// Make `newOwner` the new owner of this

    /// contract.

    function changeOwner(address newOwner)

        public

        onlyBy(owner)

    {

        owner = newOwner;

    }

    modifier onlyAfter(uint time) {

        if (block.timestamp < time)

            revert TooEarly();

        \_;

    }

    /// Erase ownership information.

    /// May only be called 6 weeks after

    /// the contract has been created.

    function disown()

        public

        onlyBy(owner)

        onlyAfter(creationTime + 6 weeks)

    {

        delete owner;

    }

    // This modifier requires a certain

    // fee being associated with a function call.

    // If the caller sent too much, he or she is

    // refunded, but only after the function body.

    // This was dangerous before Solidity version 0.4.0,

    // where it was possible to skip the part after `\_;`.

    modifier costs(uint amount) {

        if (msg.value < amount)

            revert NotEnoughEther();

        \_;

        if (msg.value > amount)

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

    }

    function forceOwnerChange(address newOwner)

        public

        payable

        costs(200 ether)

    {

        owner = newOwner;

        // just some example condition

        if (uint160(owner) & 0 == 1)

            // This did not refund for Solidity

            // before version 0.4.0.

            return;

        // refund overpaid fees

    }

}



# PRACTICAL 6B

## WRITE A SOLIDITY PROGRAM FOR CONTRACT, INHERITANCE, CONSTRUCTORS, ABSTRACT CONTRACTS, INTERFACES, LIBRARIES, ASSEMBLY, EVENTS, ERROR HANDLING.

**A)Contract:**

Contract in Solidity is similar to a Class in C++. A Contract have following properties.

**Constructor** − A special function declared with constructor keyword which will be executed once per contract and is invoked when a contract is created.

**State Variables** − Variables per Contract to store the state of the contract.

**Functions** − Functions per Contract which can modify the state variables to alter the state of a contract.

// Calling function from external 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 {

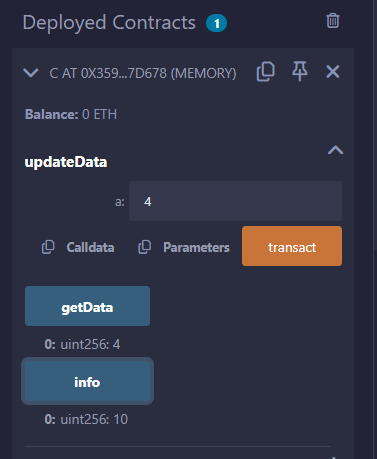
result = compute(3, 5);

}

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

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

}



**B)Inheritance:**

Inheritance is a way to extend functionality of a contract. Solidity supports both single as well as multiple inheritance.

// Solidity program to

// demonstrate

// Single Inheritance

pragma solidity >=0.4.22 <0.6.0;

// Defining contract

contract parent{

// Declaring internal

// state variable

uint internal sum;

// Defining external function

// to set value of internal

// state variable sum

function setValue() external {

uint a = 20;

uint b = 20;

sum = a + b;

}

}

// Defining child contract

contract child is parent{

// Defining external function

// to return value of

// internal state variable sum

function getValue() external view returns(uint) {

return sum;

}

}

// Defining calling contract

contract caller {

// Creating child contract object

child cc = new child();

// Defining function to call

// setValue and getValue functions

function testInheritance() public {

cc.setValue();

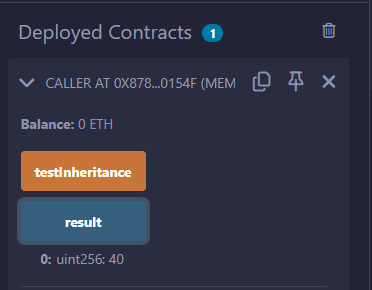
}

function result() public view returns(uint ){

return cc.getValue();

}

}



**C)Constructors:**

Constructor is a special function declared using constructor keyword. It is an optional function and is used to initialize state variables of a contract. Following are the key characteristics of a constructor.

A contract can have only one constructor.

A constructor code is executed once when a contract is created and it is used to initialize contract state.

A constructor can be either public or internal.

An internal constructor marks the contract as abstract.

In case, no constructor is defined, a default constructor is present in the contract.

pragma solidity ^0.5.0;

contract Base {

uint data;

constructor(uint \_data) public {

data = \_data;

}

function getresult()public view returns(uint){

return data;

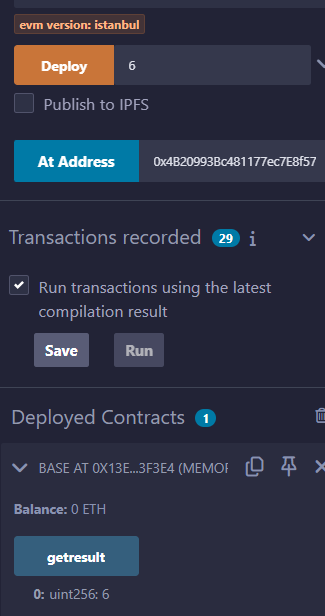
}

}

contract Derived is Base (5) {

constructor() public {}

}



**// Indirect Initialization of Base Constructor**

pragma solidity ^0.5.0;

contract Base {

uint data;

constructor(uint \_data) public {

data = \_data;

}

function getresult()public view returns(uint){

return data;

}

}

contract Derived is Base {

constructor(uint \_info) Base(\_info \* \_info) public {}

}

**D)Abstract Contracts:**

Abstract Contract is one which contains at least one function without any implementation. Such a contract is used as a base contract. Generally an abstract contract contains both implemented as well as abstract functions. Derived contract will implement the abstract function and use the existing functions as and when required.

pragma solidity ^0.5.0;

contract Calculator {

function getResult() public view returns(uint);

}

contract Test is Calculator {

function getResult() public view returns(uint) {

uint a = 4;

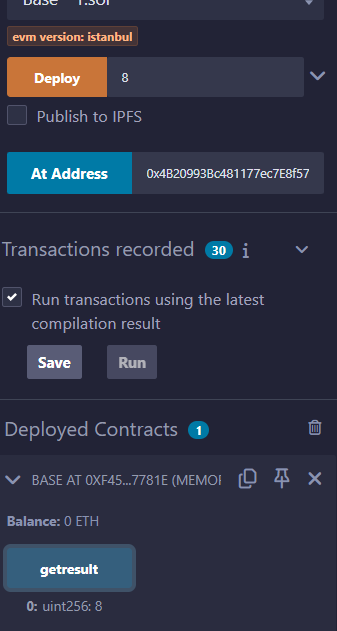
uint b = 2;

uint result = a + b;

return result;

}

}



**E)Interfaces:**

Interfaces are similar to abstract contracts and are created using interface keyword. Following are the key characteristics of an interface.

Interface can not have any function with implementation.

Functions of an interface can be only of type external.

Interface can not have constructor.

Interface can not have state variables.

pragma solidity ^0.5.0;

interface Calculator {

function getResult() external view returns(uint);

}

contract Test is Calculator {

constructor() public {}

function getResult() external view returns(uint){

uint a = 5;

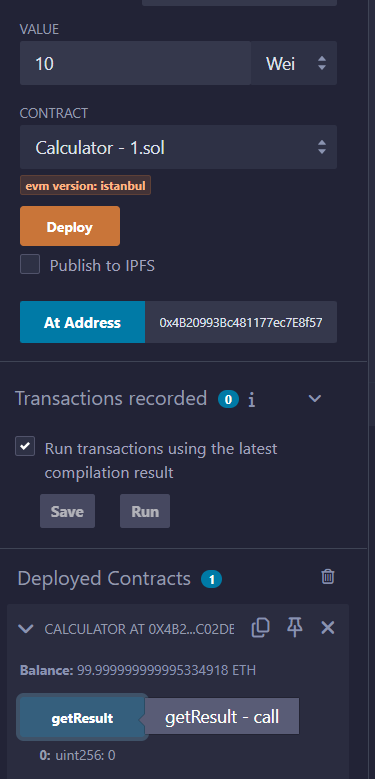
uint b = 2;

uint result = a + b;

return result;

}

}



# PRACTICAL 6c

## Libraries,Assembly, Events, Error handling.

**Libraries:**

Libraries are similar to Contracts but are mainly intended for reuse. A Library contains functions which other contracts can call. Solidity have certain restrictions on use of a Library.

pragma solidity ^0.5.0;

library Search {

function indexOf(uint[] storage self, uint value) public view returns (uint) {

for (uint i = 0; i < self.length; i++)

if (self[i] == value) return i;

return uint(-1);}

}

contract Test {

uint[] data;

uint value;

uint index;

constructor() public {

data.push(6);

data.push(7);

data.push(8);

data.push(9);

data.push(10);

}

function isValuePresent() external {

value = 9;

//search if value is present in the array using Library function

index = Search.indexOf(data, value);

}

function getresult() public view returns(uint){

return index;

}}



**Assembly:**

Solidity provides an option to use assembly language to write inline assembly within Solidity source code. We can also write a standalone assembly code which then be converted to bytecode. Standalone Assembly is an intermediate language for a Solidity compiler and it converts the Solidity code into a Standalone Assembly and then to byte code. We can used the same language used in Inline Assembly to write code in a Standalone assembly.

pragma solidity ^0.5.0;

library Sum {

function sumUsingInlineAssembly(uint[] memory \_data) public pure returns (uint o\_sum) {

for (uint i = 0; i < \_data.length; ++i) {

assembly {

o\_sum := add(o\_sum, mload(add(add(\_data, 0x20), mul(i, 0x20))))

}}

}

}

contract Test {

uint[] data;

constructor() public {

data.push(1);

data.push(2);

data.push(3);

data.push(4);

data.push(5);

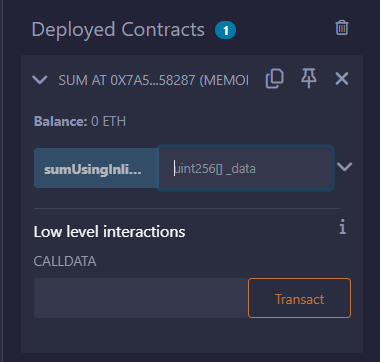
}

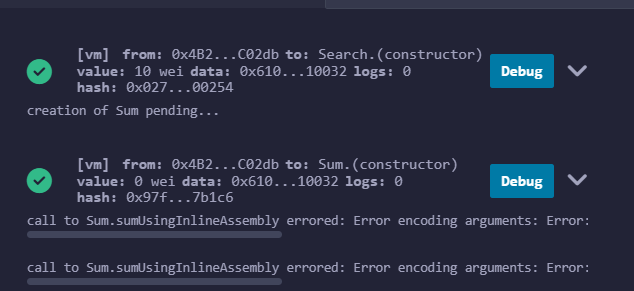
function sum() external view returns(uint){

return Sum.sumUsingInlineAssembly(data);

}

}





**Events:**

Event is an inheritable member of a contract. An event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain. An event generated is not accessible from within contracts, not even the one which have created and emitted them.

// Solidity program to demonstrate

// creating an event

pragma solidity ^0.4.21;

// Creating a contract

contract eventExample {

// Declaring state variables

uint256 public value = 0;

// Declaring an event

event Increment(address owner);

// Defining a function for logging event

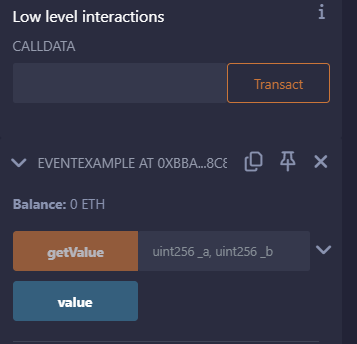
function getValue(uint \_a, uint \_b) public {

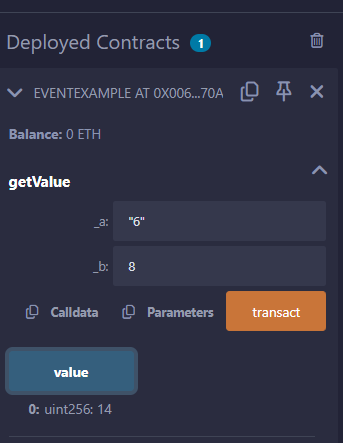
emit Increment(msg.sender);

value = \_a + \_b;

}

}





**Error Handling:**

Solidity provides various functions for error handling. Generally when an error occurs, the state is reverted back to its original state. Other checks are to prevent unauthorized code access.

**Solidity program to demonstrate require statement.**

// Solidity program to

// demonstrate require

// statement

pragma solidity ^0.5.0;

// Creating a contract

contract requireStatement {

// Defining function to

// check input

function checkInput(uint8 \_input) public view returns(string memory){

require(\_input >= 0, "invalid uint");

require(\_input <= 255, "invalid uint8");

return "Input is Uint8";

}

// Defining function to

// use require statement

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

require(\_input % 2 != 0);

return true;

}

}





**Solidity program to demonstrate assert statement.**

// Solidity program to

// demonstrate assert

// statement

pragma solidity ^0.5.0;

// Creating a contract

contract assertStatement {

// Defining a state variable

bool result;

// Defining a function

// to check condition

function checkOverflow(uint8 \_num1, uint8 \_num2) public {

uint8 sum = \_num1 + \_num2;

assert(sum<=255);

result = true;

}

// Defining a function to

// print result of assert

// statement

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

if(result == true){

return "No Overflow";

}

else{

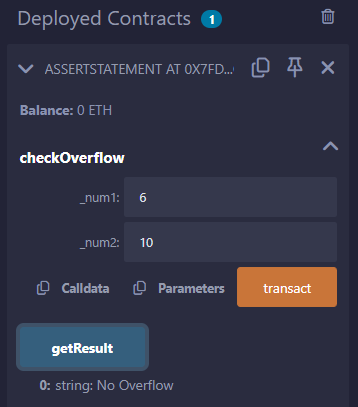
return "Overflow exist";

}

}

}





**Solidity program to demonstrate revert statement.**

// Solidity program to

// demonstrate revert

pragma solidity ^0.5.0;

// Creating a contract

contract revertStatement {

// Defining a function

// to check condition

function checkOverflow(uint \_num1, uint \_num2) public view returns(

string memory, uint) {

uint sum = \_num1 + \_num2;

if(sum < 0 || sum > 255){

revert(" Overflow Exist");

}

else{

return ("No Overflow", sum);

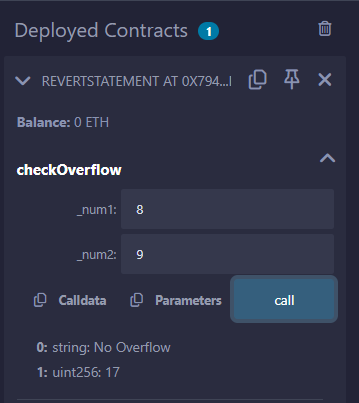
}

}

}

https://www.tutorialspoint.com/solidity/index.htm





**PRACTICAL 7**

**Aim**:-Deploying a contracts on an external blockchain by using Ganache and/or MyEtherwalllet, Metamask

https://abhibvp003.medium.com/how-to-install-and-execute-truffle-on-an-ubuntu-16-04-7d0ff6458c9b

https://ethereum.stackexchange.com/questions/93533/call-an-existing-contract-function-from-truffle-console

sudo apt-get -y install curl git vim build-essential

sudo apt-get install curl software-properties-common

sudo apt install npm

sudo npm install -g web3

sudo apt-get install nodejs

sudo apt install python3.9

curl -sL https://deb.nodesource.com/setup\_10.x | sudo bash -

sudo npm install --global node-sass@latest

sudo npm install -g truffle@latest

sudo npm install -g  ganache-cli

export NODE\_OPTIONS=--openssl-legacy-provider

////to update npm//

sudo npm cache clean -f

sudo npm install -g n

sudo n latest

//////////////////////////////////////

Start from here!!!

mkdir upg1

cd upg1

truffle init

////////// create contract

nano contracts/HelloWorld.sol

pragma solidity ^0.5.0;

contract HelloWorld {

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

        return("hello world");

    }

}

////////////////////create configuration

nano migrations/1\_initial\_migration.js

const Migrations = artifacts.require("HelloWorld");

module.exports = function (deployer) {

  deployer.deploy(Migrations,"hello");

};

////////////////network configuration

nano truffle-config.js

module.exports = {

  networks: {

    development: {

      host: "127.0.0.1",

      port: 8545,

      network\_id: "\*",

    }

  }

}

////////////////////start ganache-cli

ganache-cli

////////////////

truffle migrate

truffle console

#replace contact address

contract = await HelloWorld.at('0x37354B83aadd35516c56f24b724228f29300be77')

a = await contract.sayHello()

a

**PRACTICAL 8**

**Aim**:-Create your own blockchain and demonstrate its use.Deploy a local private blockchain over a network with Ethereum or Rust (VM )

Install on Ubuntu via PPAs

The easiest way to install go-ethereum on Ubuntu-based distributions is with the built-in launchpad PPAs (Personal Package Archives). We provide a single PPA repository that contains both our stable and development releases for Ubuntu versions trusty, xenial, zesty and artful.

linux:

To enable our launchpad repository run:

Step 1: open new terminal

Step 2: on terminal type this command

sudo add-apt-repository -y ppa:ethereum/ethereum

#if above command gives error then run

#sudo apt-get install --reinstall ca-certificates

Step 3: install the stable version of go-ethereum:

sudo apt-get update

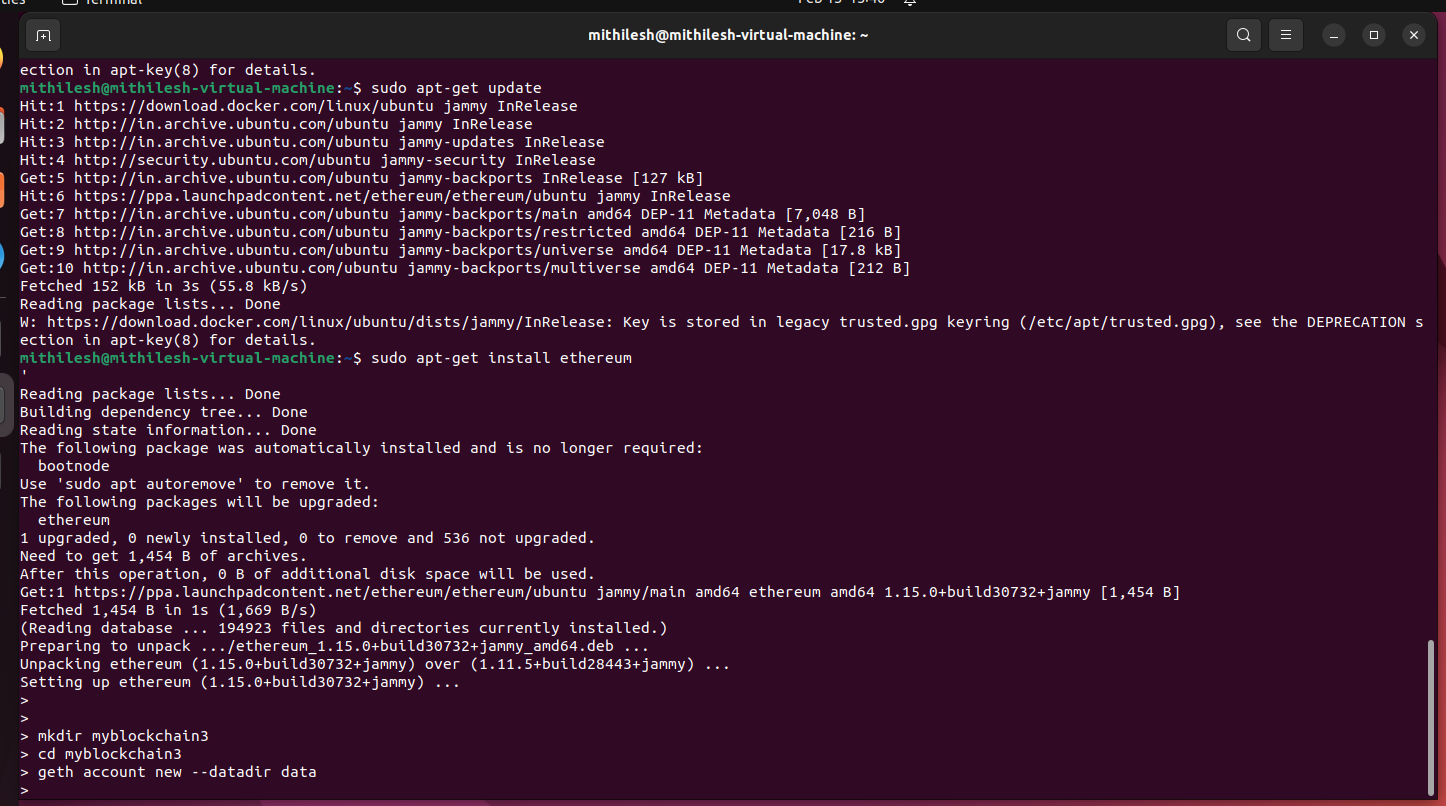
sudo apt-get install ethereum

Step 4: create new directory for storing blockchain data

mkdir myblockchain2

cd myblockchain2

geth account new --datadir data



Step 5:Create genesis.json file

sudo nano genesis.json

{

"config": {

"chainId": 12345,

"homesteadBlock": 0,

"eip150Block": 0,

"eip155Block": 0,

"eip158Block": 0,

"byzantiumBlock": 0,

"constantinopleBlock": 0,

"petersburgBlock": 0,

"istanbulBlock": 0,

"berlinBlock": 0,

"ethash": {}

},

"difficulty": "1",

"gasLimit": "8000000",

"alloc": {

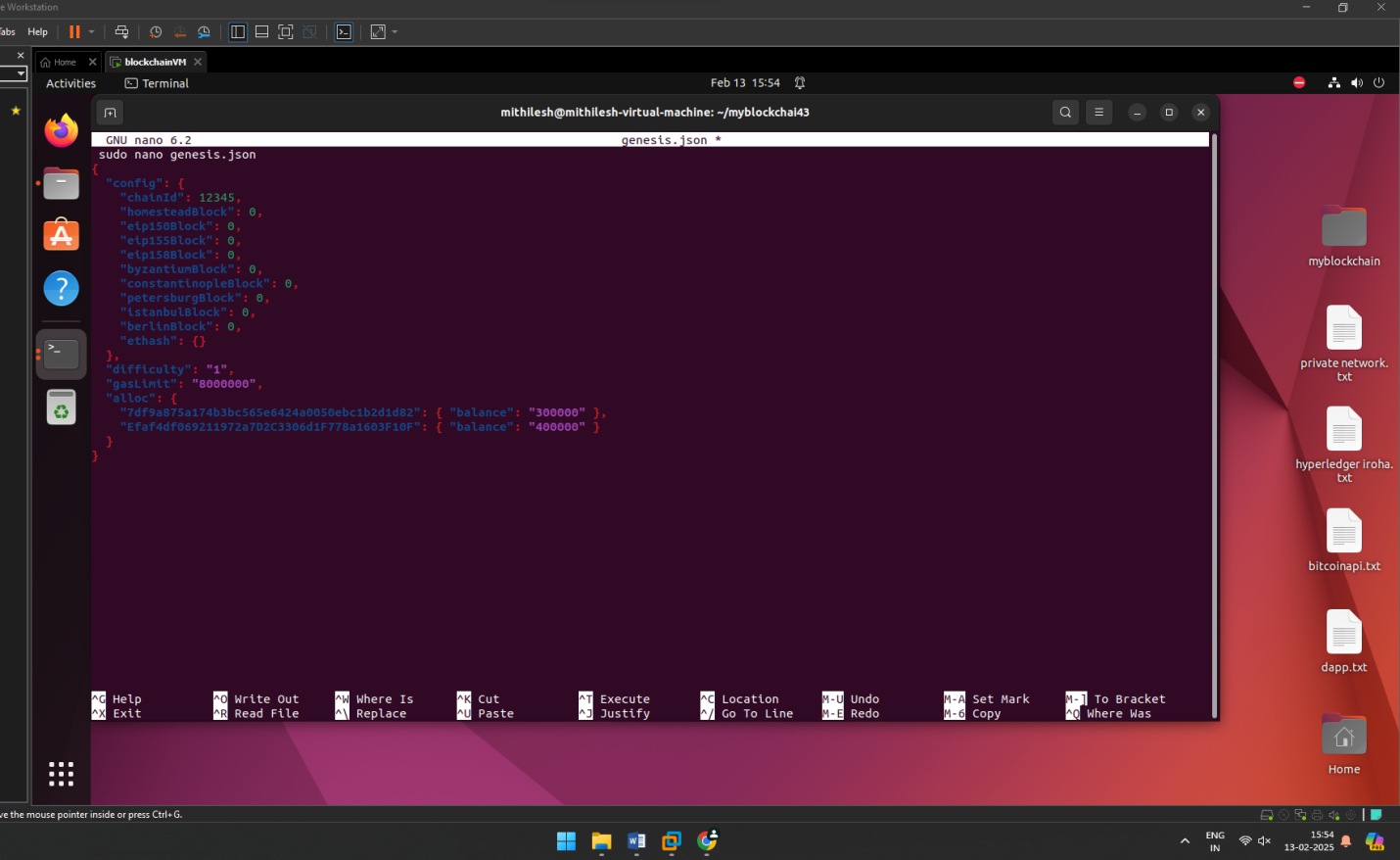
"7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance": "300000" },

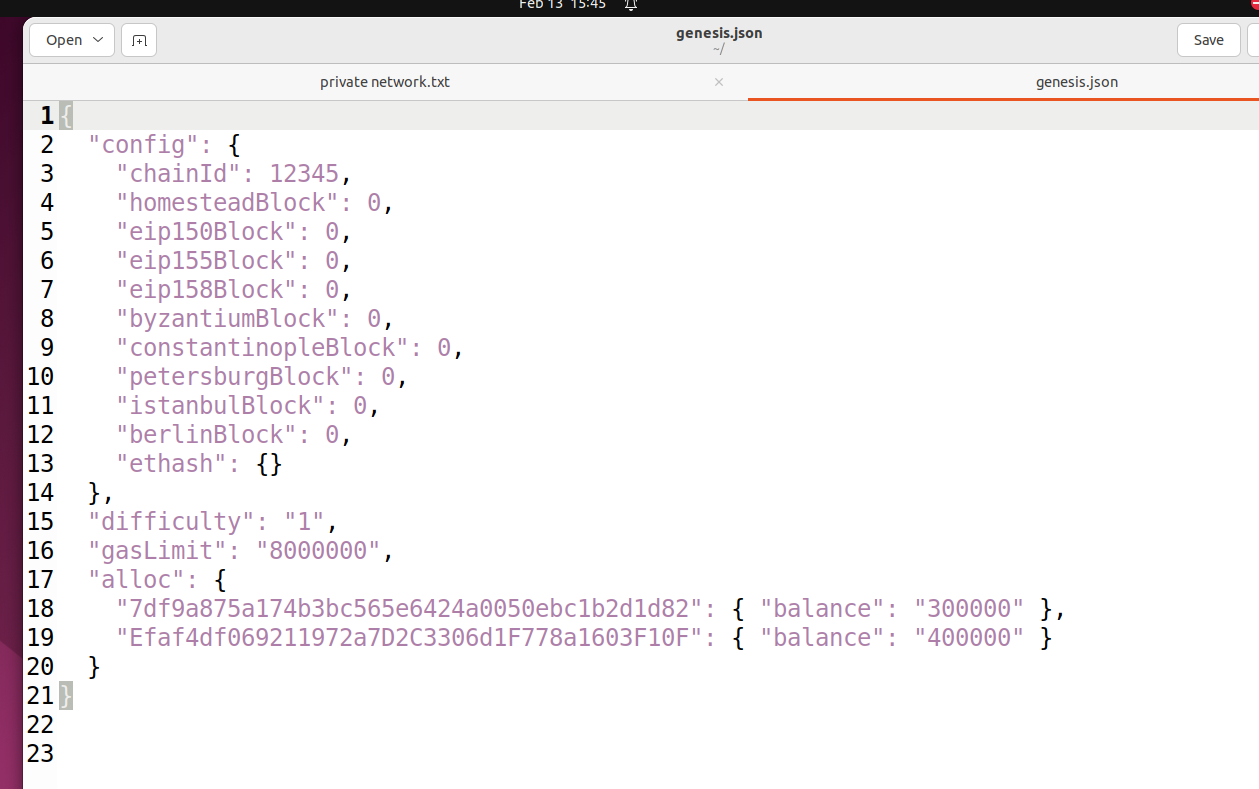
"Efaf4df069211972a7D2C3306d1F778a1603F10F": { "balance": "400000" }

}

}

save the file -> ctrl +o to write -> {enter} save -> ctrl +x exit





Step 6: initialize the block

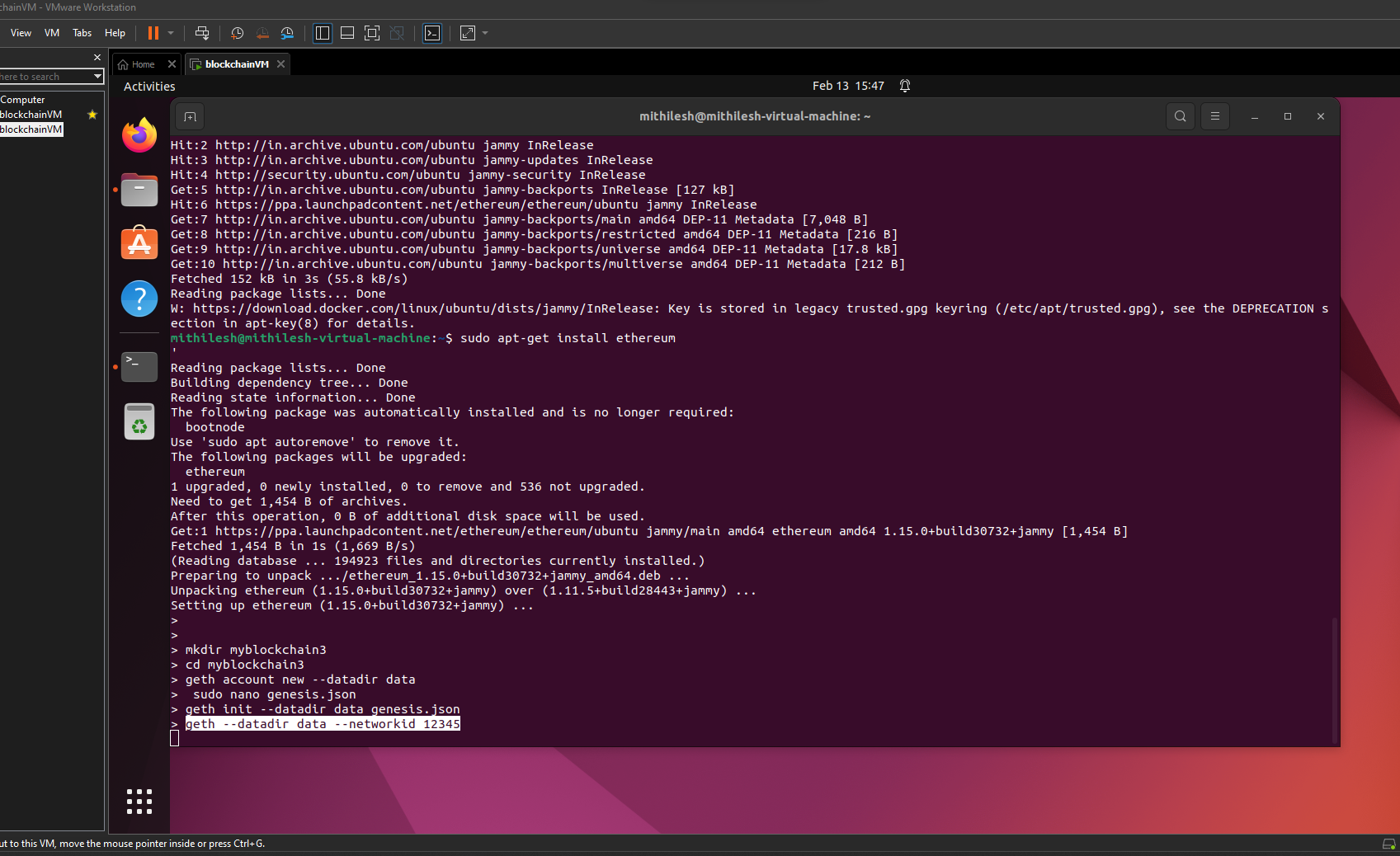
geth init --datadir data genesis.json

Step 7: create network

geth --datadir data --networkid 12345

[do not close this terminal]

//////////////////////////////////////////////////////////////////////



Step 8: open new tab/terminal 2:

sudo geth attach data/geth.ipc

eth.getBalance(eth.accounts[0])

miner.setEtherbase(eth.accounts[0])

miner.start()

admin.addPeer(admin.nodeInfo.enode)

eth.getBalance(eth.accounts[0])

Step 10: Wait for 10-20 minutes and check balance

eth.getBalance(eth.accounts[0])

if ether balance is 0 wait for 10-20minutes for mining process to get complete and run eth.getBalance(eth.accounts[0]) again.

After balance is updated you can check current block height

eth.blockNumber

# PRACTICAL 9

## Implement the mining module of Bitcoin client . The mining module, or miner, should produce blocks that solve proof-of-work puzzle

**Code:-**Open Python IDLE and create new Script.

########################

!pip install bitcoinlib

from bitcoinlib.wallets import Wallet

w = Wallet.create('Wallet1')

key1 = w.get\_key()

print('Wallet Address:',key1.address)

w.scan()

print(w.info())



Open CMD and install bitcoinlib package

**pip install bitcoinlib**

# PRACTICAL 10

## Compile and test smart contracts on a testing framework using the Ethereum Virtual Machine (EVM).

**Code:-**

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

import "remix\_tests.sol"; // this import is automatically injected by Remix.

import "hardhat/console.sol";

import "../contracts/3\_Ballot.sol";

contract BallotTest {

    bytes32[] proposalNames;

    Ballot ballotToTest;

    function beforeAll () public {

        proposalNames.push(bytes32("candidate1"));

        ballotToTest = new Ballot(proposalNames);

    }

    function checkWinningProposal () public {

        console.log("Running checkWinningProposal");

        ballotToTest.vote(0);

        Assert.equal(ballotToTest.winningProposal(), uint(0), "proposal at index 0 should be the winning proposal");

        Assert.equal(ballotToTest.winnerName(), bytes32("candidate1"), "candidate1 should be the winner name");

    }

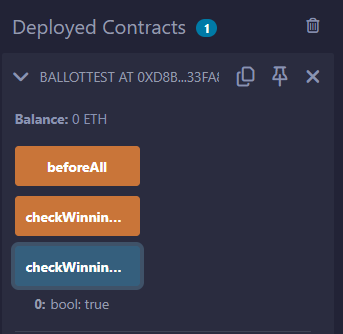
    function checkWinninProposalWithReturnValue () public view returns (bool) {

        return ballotToTest.winningProposal() == 0;

    }

}

**Output:-**

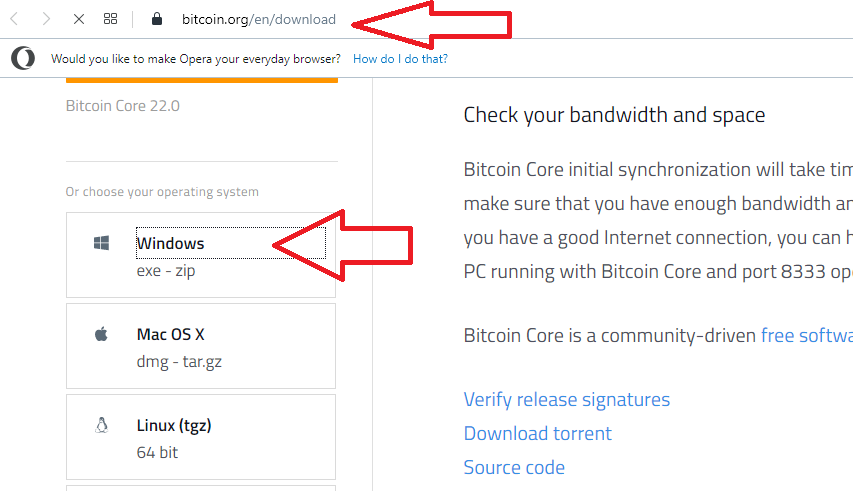


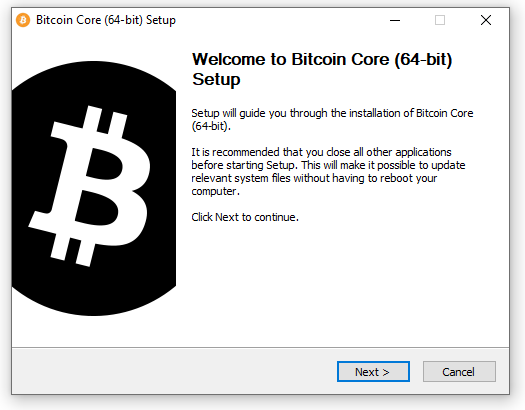
**PRACTICAL 11**

**Aim**:-**Demonstrate the use of Bitcoin Core API.**

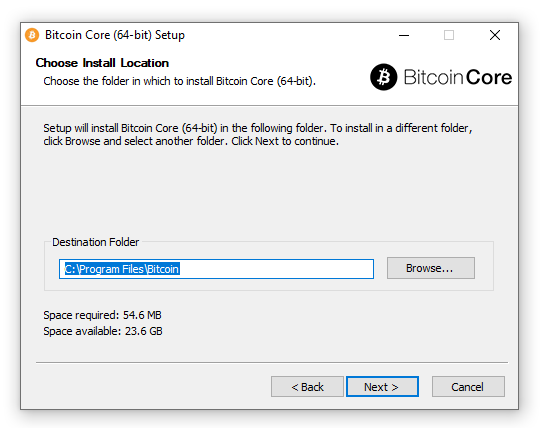
Step 1: Visit: <https://bitcoin.org/en/download>

Step 2: Download windows setup [use and try with Linux version as well]

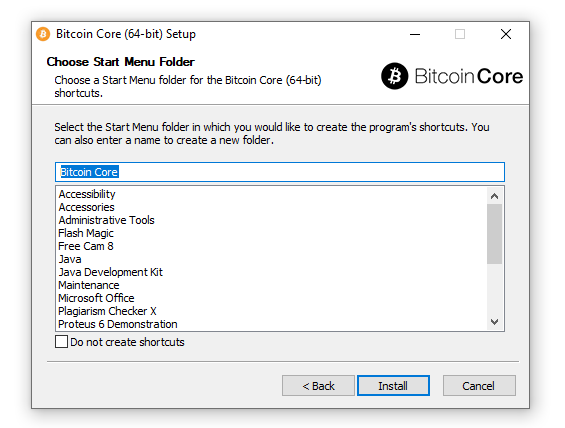


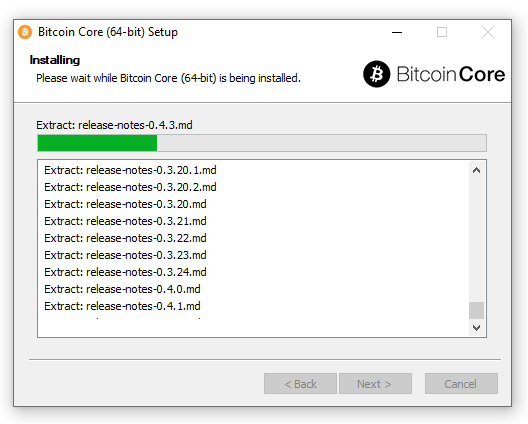
Step 3: Run the setup file-> click next

Step 4: Click Next

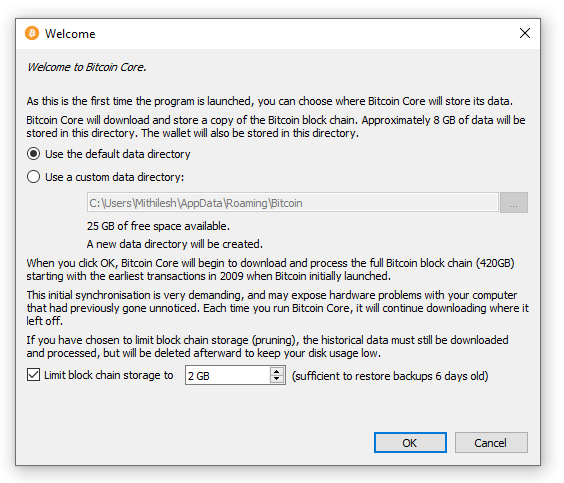


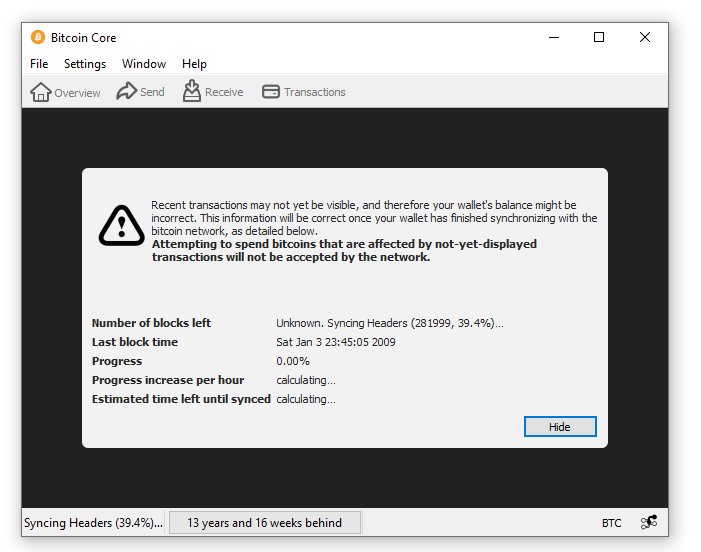
Step 5: Finally click on Install



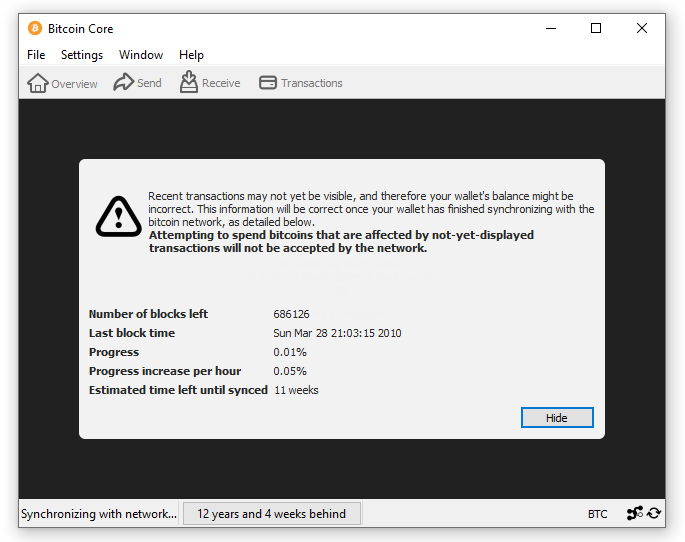


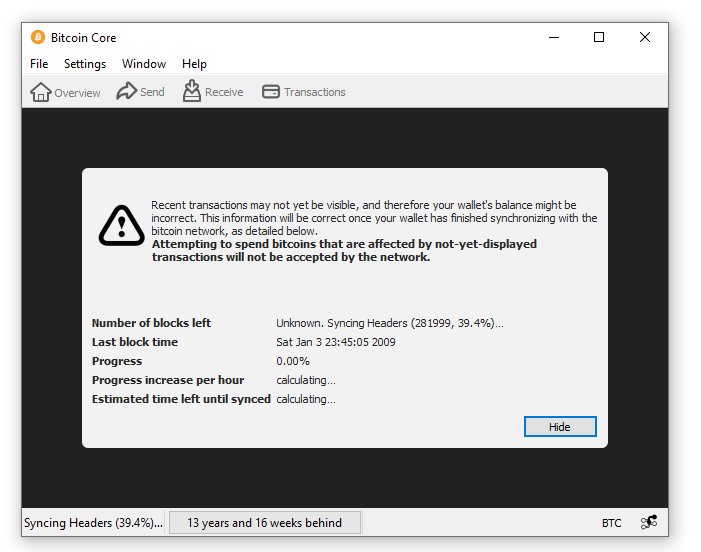
Launch Bitcoin Core-> Click OK.



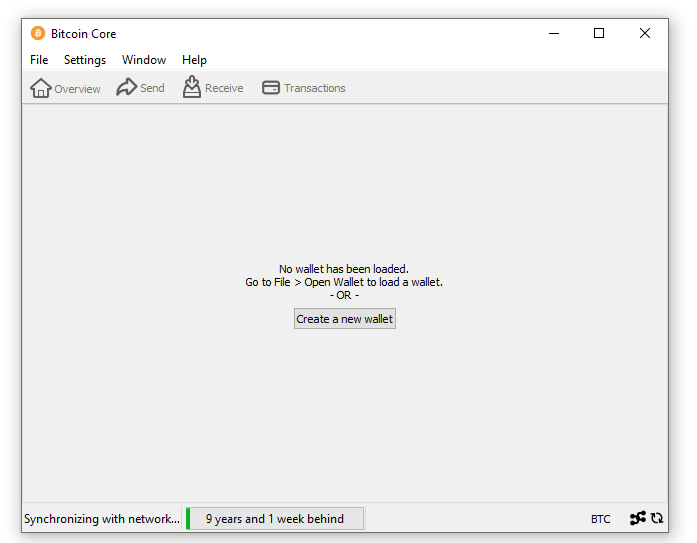


Click on Hide button [Synchronization take place in background]

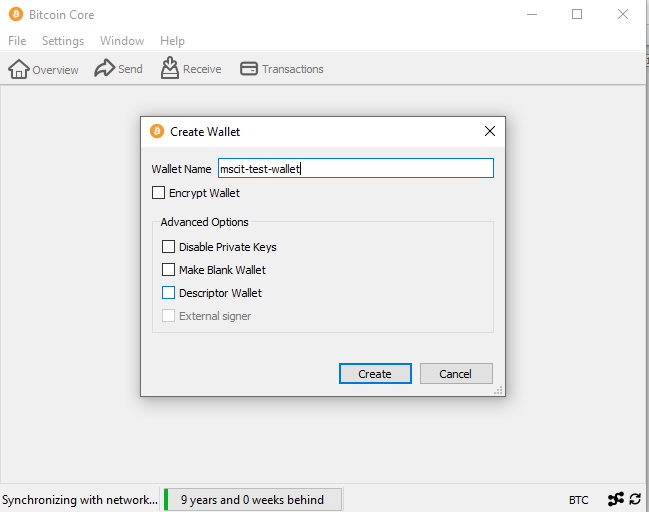




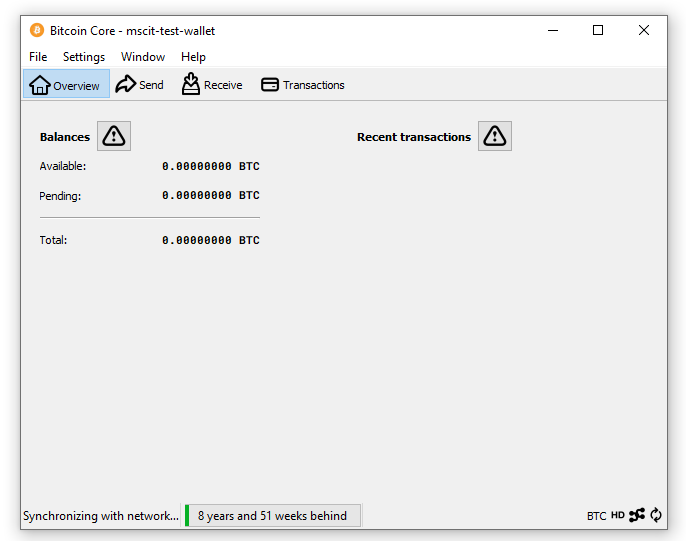
You can create a wallet -> Create a new wallet

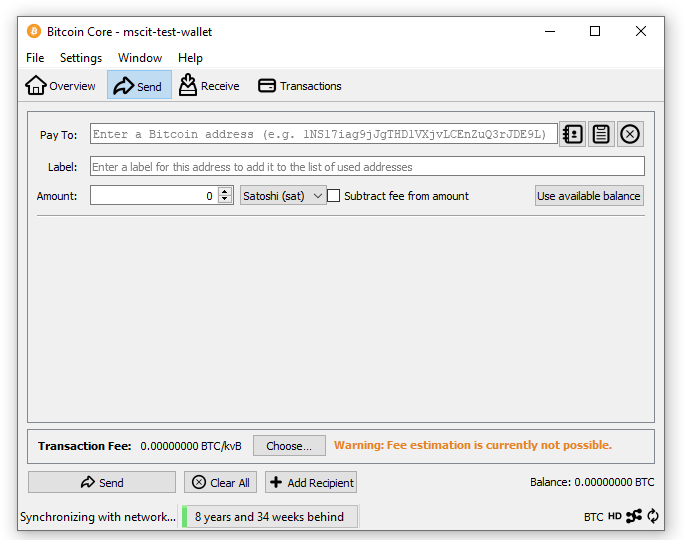


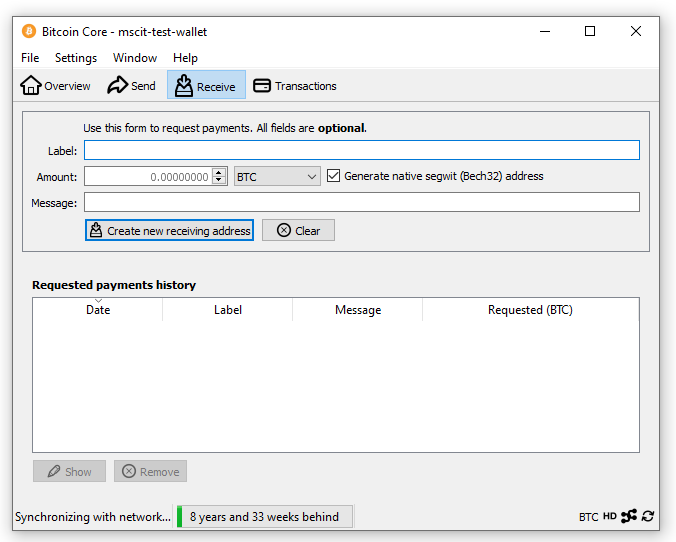
Enter Wallet name

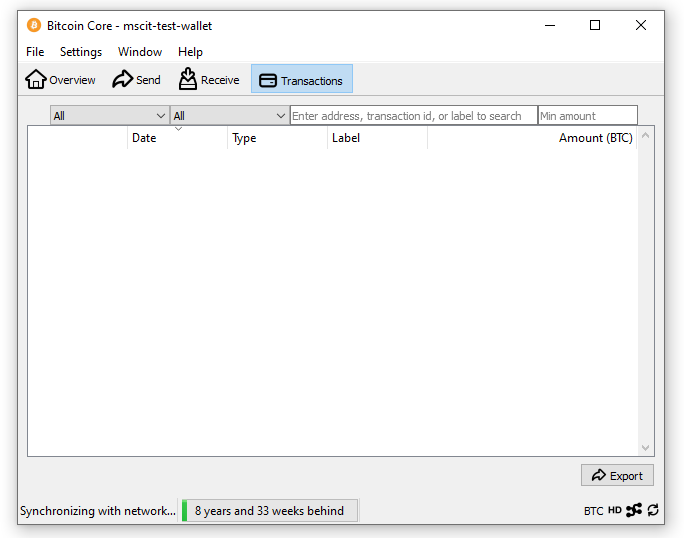


Finally Account is setup









# PRACTICAL 12

## Create your own blockchain and demonstrate its use.

**Code:-**

# following imports are required by PKI

import hashlib

import random

import binascii

import datetime

import collections

from Crypto.PublicKey import RSA

from Crypto import Random

from Crypto.Cipher import PKCS1\_v1\_5

from collections import OrderedDict

import Crypto

import Crypto.Random

from Crypto.Hash import SHA

from Crypto.Signature import PKCS1\_v1\_5

class Client:

    def \_\_init\_\_(self):

      random = Random.new().read

      self.\_private\_key = RSA.generate(1024, random)

      self.\_public\_key = self.\_private\_key.publickey()

      self.\_signer = PKCS1\_v1\_5.new(self.\_private\_key)

    @property

    def identity(self):

      return binascii.hexlify(self.\_public\_key.exportKey(format='DER')).decode('ascii')

class Transaction:

    def \_\_init\_\_(self, sender, recipient, value):

       self.sender = sender

       self.recipient = recipient

       self.value = value

       self.time = datetime.datetime.now()

    def to\_dict(self):

       if self.sender == "Genesis":

          identity = "Genesis"

       else:

          identity = self.sender.identity

       return collections.OrderedDict({

          'sender': identity,

          'recipient': self.recipient,

          'value': self.value,

          'time' : self.time})

    def sign\_transaction(self):

       private\_key = self.sender.\_private\_key

       signer = PKCS1\_v1\_5.new(private\_key)

       h = SHA.new(str(self.to\_dict()).encode('utf8'))

       return binascii.hexlify(signer.sign(h)).decode('ascii')

def display\_transaction(transaction):

       #for transaction in transactions:

       dict = transaction.to\_dict()

       print ("sender: " + dict['sender'])

       print ('-----')

       print ("recipient: " + dict['recipient'])

       print ('-----')

       print ("value: " + str(dict['value']))

       print ('-----')

       print ("time: " + str(dict['time']))

       print ('-----')

def dump\_blockchain (self):

   print ("Number of blocks in the chain: " + str(len (self)))

   for x in range (len(TPCoins)):

      block\_temp = TPCoins[x]

      print ("block # " + str(x))

      for transaction in block\_temp.verified\_transactions:

         display\_transaction (transaction)

         print ('--------------')

      print ('=====================================')

class Block:

   def \_\_init\_\_(self):

      self.verified\_transactions = []

      self.previous\_block\_hash = ""

      self.Nonce = ""

def sha256(message):

      return hashlib.sha256(message.encode('ascii')).hexdigest()

def mine(message, difficulty=1):

   assert difficulty >= 1

   #if(difficulty <1):

   #        return

   #'1'\*3=> '111'

   prefix = '1' \* difficulty

   for i in range(1000):

      digest = sha256(str(hash(message)) + str(i))

      if digest.startswith(prefix):

         return i #i= nonce value

A = Client()

B =Client()

C =Client()

t0 = Transaction (

   "Genesis",

   A.identity,

   500.0

)

t1 = Transaction (

   A,

   B.identity,

   40.0

)

t2 = Transaction (

   A,

   C.identity,

   70.0

)

t3 = Transaction (

   B,

   C.identity,

   700.0

)

#blockchain

TPCoins = []

block0 = Block()

block0.previous\_block\_hash = None

Nonce = None

block0.verified\_transactions.append (t0)

digest = hash (block0)

last\_block\_hash = digest #last\_block\_hash it is hash of block0

TPCoins.append (block0)

block1 = Block()

block1.previous\_block\_hash = last\_block\_hash

block1.verified\_transactions.append (t1)

block1.verified\_transactions.append (t2)

block1.Nonce=mine (block1, 2)

digest = hash (block1)

last\_block\_hash = digest

TPCoins.append (block1)

block2 = Block()

block2.previous\_block\_hash = last\_block\_hash

block2.verified\_transactions.append (t3)

Nonce = mine (block2, 2)

block2.Nonce=mine (block2, 2)

digest = hash (block2)

last\_block\_hash = digest

TPCoins.append (block2)

dump\_blockchain(TPCoins)

#####################################################################

save the file -> ctrl +O to write ->  {enter} save -> ctrl +x exit

**Run this file**

Output:

