Subject: Laboratory Practice III **Experiment No:** 15						
Title / Aim of Experiment: Write a smart contract on a test network, for Bank account of a customer for following operations: • Deposit money • Withdraw Money • Show balance						
Name of the student:						
Department: **Class:**	**Computer Engineering** **B. E**					
Div.:		**Batch:**				
Date of Performance:						
Date of Evaluation:						
Rubrics & Marks	**Completeness** (20%) **2**	**Quality of Work** (40%) **4**	**Organization** (20%) **2**	**Presentation** (10%) **1**	**Creativity & Engagement** (10%) **1**	**Total** **10**
Marks obtained						
CO Mapped	**CO6:** Interpret the basic concepts in Blockchain technology and its applications.					
Signature of Subject Teacher						

Group C

Assignment No: 15

Title of the Assignment: Write a smart contract on a test network, for Bank account of a customer for following operations:

- Deposit money
- Withdraw Money
- Show balance

Objective of the Assignment: Students should be able to learn about smart contract for banking application and able to perform some operation like Deposit money Withdraw Money Show balance.

Prerequisite:

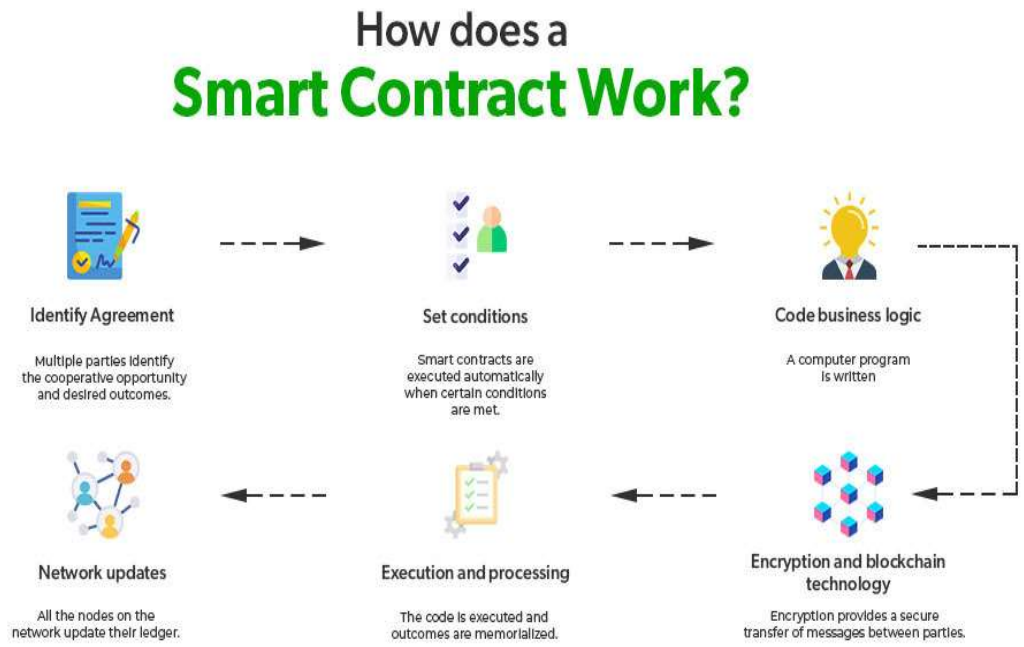
1. Basic knowledge of smart contract
2. Remix IDE
3. Basic knowledge of solidity

Contents for Theory:

1. **Smart contract**
2. **Remix IDE**
3. **Solidity Basics**
4. **Ether transaction**

Introduction to Smart Contract

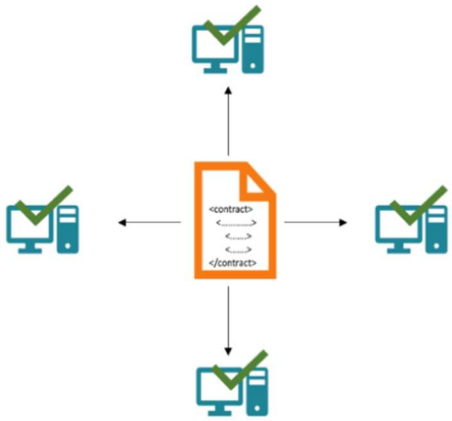
- Smart contracts are self-executing lines of code with the terms of an agreement between buyer and seller automatically verified and executed via a computer network.
- Nick Szabo, an American computer scientist who invented a virtual currency called "Bit Gold" in 1998,1 defined smart contracts as computerized transaction protocols that execute terms of a contract.2
- Smart contracts deployed to blockchains render transactions traceable, transparent, and irreversible.



How does Smart Contract work?

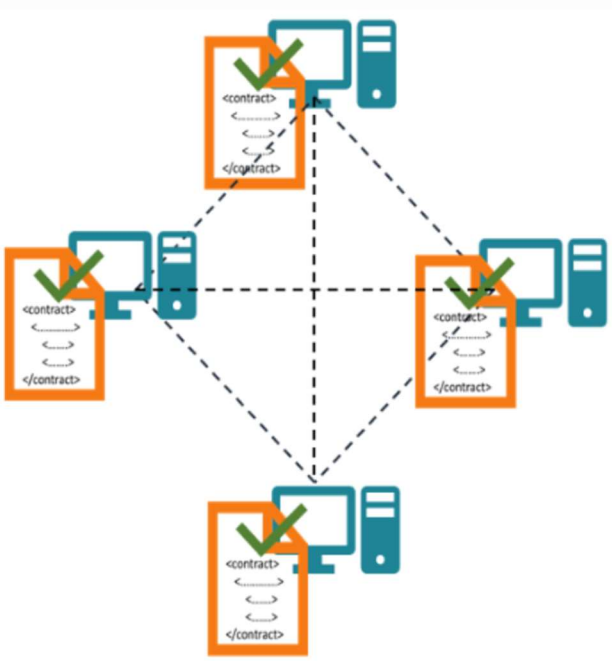
- First, the contractual parties should determine the terms of the contract. After the contractual terms are finalized, they are translated into programming code. Basically, the code represents a number of different conditional statements that describe the possible scenarios of a future transaction.

- When the code is created, it is stored in the blockchain network and is replicated among the participants in the blockchain.



Step 2. The code is stored in a blockchain and replicated between participants

- Then, the code is run and executed by all computers in the network. If a term of the contract is satisfied and it is verified by all participants of the blockchain network, then the relevant transaction is executed.



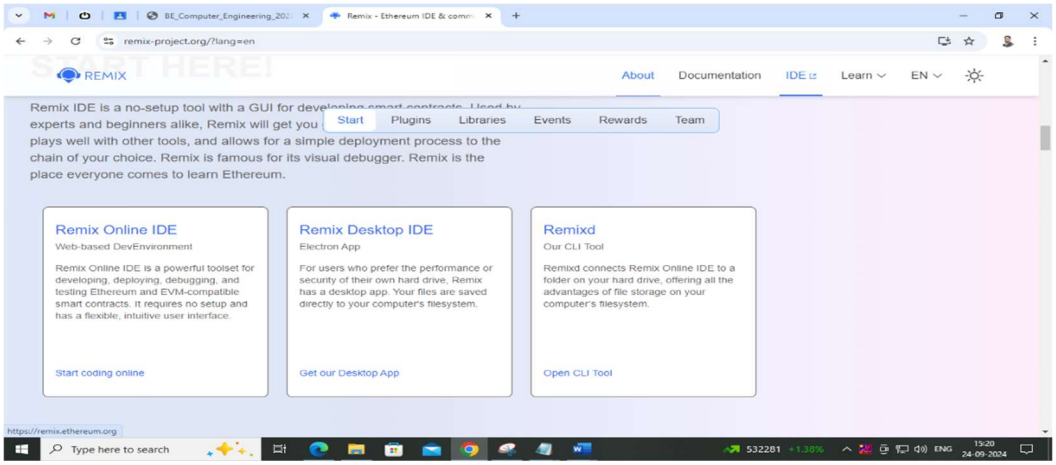
Step 3. When a term is satisfied, computers in the network verify its correctness.

Remix IDE: Installation steps

- First and foremost, head on over to the release page of the official Remix Desktop repository and grab the binary suited for your host system. This will be the .exe file for Windows users, the .dmg for macOS and the .deb for Debian-derivative GNU/Linux systems. For Ubuntu and other AppImage

setups, the .AppImage will be what you are looking for.

- Go to https://remix-project.org/
- Go to IDE option & select Desktop IDE



- Install Executable file,by double clicking on file(Applicable to .exe file Windows OS) or By clicking on IDE tab you will get online Remix Ide editor.

What is Solidity?

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM)

Solidity is statically typed programming language

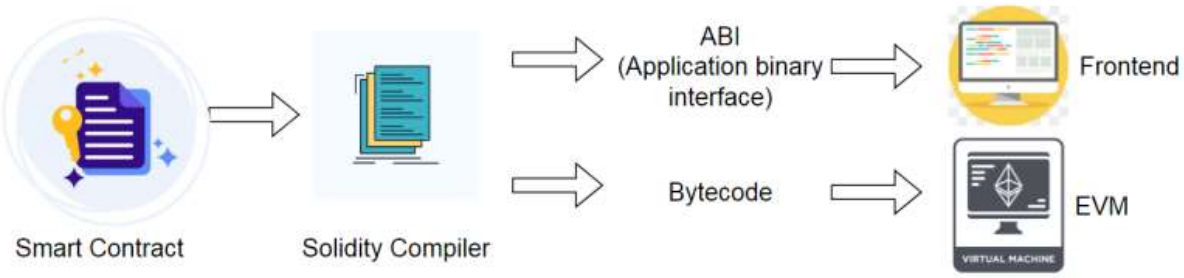
SMART CONTRACTS

Smart contracts are simply **programs stored on a blockchain that run when predetermined conditions are met**. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss.

ETHEREUM VIRTUAL MACHINE(EVM)

The Ethereum Virtual Machine is **the software platform that developers can use to create decentralized applications (DApps) on Ethereum**. This virtual machine is where all Ethereum accounts and smart contracts live.

SOLIDITY COMPILATION PROCESS



Before deploy smart contract, we need to compile solidity code to Bytecode for EVM(Ethereum Virtual Machine). You got two stuff after compiling solidity code:

- Bytecode/EVM code
- ABI(Application Binary Interface)

Bytecode is code EVM execute in blockchain network. ABI defines the actions you interact with smart contract. The actions in ABI mean the functions of smart contract.

PREREQUISITE

- Previous experience with any programming language like C, Python, or JavaScript.

Solidity Types

Value Types

Value type variables store their own data. These are the basic data types provided by solidity. These types of variables are always passed by value. The variables are copied wherever they are used in function arguments or assignment. Value type data types in solidity are listed below:

- **Boolean:** This data type accepts only two values True or False.
- **Integer:** This data type is used to store integer values, *int* and *uint* are used to declare *signed* and *unsigned integers* respectively.
- **Address:** Address hold a 20-byte value which represents the size of an Ethereum address. An address can be used to get balance or to transfer a balance by *balance* and *transfer* method respectively.
- **Bytes and Strings:** Bytes are used to store a fixed-sized character set while the string is used to store the character set equal to or more than a byte. The length of bytes is from 1 to 32, while the string has a dynamic length. Byte has an advantage that it uses less gas, so better to use when we know the length of data.
- **Enums:** It is used to create user-defined data types, used to assign a name to an integral constant which makes the contract more readable, maintainable, and less prone to errors. Options of enums can be represented by unsigned integer values starting from 0.

Type & Variables.sol

```
1  //SPDX-License-Identifier: Unlicensed
2
3  pragma solidity >=0.7.0;
4
5  contract Variable {
6      uint videos = 30;
7      int playlist = 3;
8      bool active = true;
9      bytes4 symbol = "web3";
10     string name = "web3Mantra";
11 }
```

Int Range Formula:- positive 2^(n-1) - 1 negative 2^(n-1)

Uint Range Formula:- (2^n) -1

Solidity Variables

Solidity 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.

Solidity is a statically typed language, which means that the state or local variable type needs to be specified during declaration. Each declared variable always have a default value based on its type. There is no concept of "undefined" or "null".

Block and Transaction Properties

- `blockhash(uint blockNumber) returns (bytes32)` : hash of the given block when `blocknumber` is one of the 256 most recent blocks; otherwise returns zero
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)` : remaining gas
- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Members of Address Types

`<address>.balance` (`uint256`)

balance of the Address in Wei

`<address>.code` (`bytes memory`)

code at the Address (can be empty)

`<address>.codehash` (`bytes32`)

the codehash of the Address

`<address payable>.transfer(uint256 amount)`

send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable

Solidity Variable 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.

Solidity Functions

A function is a block of code that performs a task. it can be called and reused multiple times. You can pass information to a function and it can send information back

Functions.sol | Type & Variables.sol

```
// SPDX-License-Identifier: Unlicensed

pragma solidity >=0.7.0;

contract functions {
    uint public val = 4;

    function add() public pure returns(uint) {
        // code
        return 3 + 5;
    }
}

// Pure -> Pure function do not view and change state variables
// view -> view function can only view state varibale cannot change them
// public -> generate a getter function to state variables
// returns -> use to specify the return data type of function
```

Storage v/s Memory

[Storage vs Memory in Solidity - GeeksforGeeks]

Storage and Memory keywords in Solidity are analogous to Computer's hard drive and Computer's RAM. Much like RAM, Memory in Solidity is a temporary place to store data whereas Storage holds data between function calls. The Solidity Smart Contract can use any amount of memory during the execution but once the execution stops, the Memory is completely wiped off for the next execution. Whereas Storage on the other hand is persistent, each execution of the Smart contract has access to the data previously stored on the storage area.

Every transaction on Ethereum Virtual Machine costs us some amount of Gas. The lower the Gas consumption the better is your Solidity code. The Gas consumption of Memory is not very significant as compared to the gas consumption of Storage. Therefore, it is always better to use Memory for intermediate calculations and store the final result in Storage.

1. State variables and Local Variables of structs, array are always stored in storage by default.
2. Function arguments are in memory.
3. Whenever a new instance of an array is created using the keyword 'memory', a new copy of that variable is created. Changing the array value of the new instance does not affect the original array.

Solidity Operators

In any programming language, operators play a vital role i.e. they create a foundation for the programming. Similarly, the functionality of Solidity is also incomplete without the use of operators. Operators allow users to perform different operations on operands. Solidity supports the following types of operators based upon their functionality.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment operators
6. Conditional Operator

Solidity 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.

Declaring Arrays

To declare an array of fixed size in Solidity, the programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type[arraySize] arrayName;
```

Fixed-size Arrays

The size of the array should be predefined. The total number of elements should not exceed the size of the array. If the size of the array is not specified then the array of enough size is created which is enough to hold the initialization.

Dynamic Array:

The size of the array is not predefined when it is declared. As the elements are added the size of array changes and at the runtime, the size of the array will be determined.

Array Operations

1. Accessing Array Elements: The elements of the array are accessed by using the index.

2. Length of Array: Length of the array is used to check the number of elements present in an array. The size of the memory array is fixed when they are declared, while in case the dynamic array is defined at runtime so for manipulation length is required.

3. Push: Push is used when a new element is to be added in a dynamic array. The new element is always added at the last position of the array.

4. Pop: Pop is used when the last element of the array is to be removed in any dynamic array.

Solidity Struct

Struct types are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Struct

To define a Struct, you must use the **struct** keyword. The struct keyword defines a new data type, with more than one member. The format of the struct statement is as follows −

```
struct struct_name {
   type1 type_name_1;
   type2 type_name_2;
   type3 type_name_3;
}
```

Example

```
struct Book {
   string title;
   string author;
   uint book_id;
}
```

Example

Try the following code to understand how the structs works in Solidity.

```
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;
   }
}
```

CONSTRUCTOR

Constructor is a special function declared using **constructor** keyword. It is an optional funtion 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.

After a constructor code executed, the final code is deployed to blockchain. This code include public functions and code reachable through public functions. Constructor code or any internal method used only by constructor are not included in final code.

A constructor can be either public or internal.

A 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 Test {
   constructor() {}
}
```

In case, base contract have constructor with arguments, each derived contract have to pass them.

Base constructor can be initialized directly using following way –

```
pragma solidity ^0.5.0;

contract Base {
   uint data;
   constructor(uint _data) {
      data = _data;
   }
}
contract Derived is Base (5) {
   constructor() {}
}
```

Base constructor can be initialized indirectly using following way –

```
pragma solidity ^0.5.0;

contract Base {
   uint data;
   constructor(uint _data) {
      data = _data;
   }
}
contract Derived is Base {
   constructor(uint _info) Base(_info * _info) {}
}
```

Payable

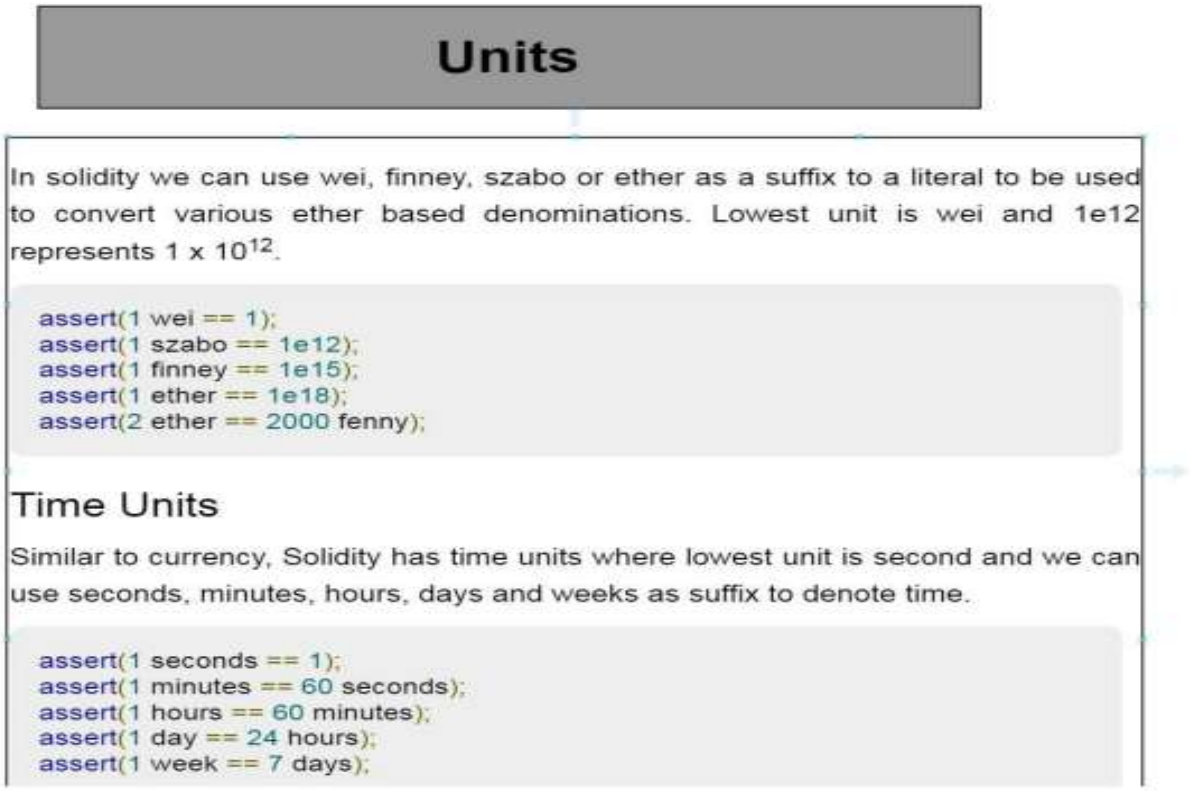
In **Solidity**, we can use the keyword `payable` to specify that an address or a function can receive Ether.

```
pragma solidity >=0.7.0;

contract payables {
    address payable public owner;

    constructor() {
        owner = payable(msg.sender);
    }

    function transferEth() payable public {
        owner.transfer(msg.value);
    }
}
```



Steps to execute program:

- **Step 1: Access Remix**

  Open your web browser and go to the Remix IDE.

- **Step 2: Create a New File**

  a. In the Remix interface, click on the File Explorers icon on the left sidebar (it looks like a folder).
  b. Click on the "+" icon to create a new file.
  c. Name the file MyBank.sol.

- **Step 3: Copy the Code**

  1) Copy the entire Solidity code you provided.
  2) Paste it into the new file (MyBank.sol) you just created in Remix.

- **Step 4: Compile the Contract**

1) Click on the Solidity Compiler icon on the left sidebar (it looks like a Solidity logo).
2) Ensure that the compiler version is set to 0.8.0 or higher (you can select it from the dropdown).
3) Click the Compile MyBank.sol button. Make sure there are no errors in the compilation process.

Step 5: Deploy the Contract

1) Click on the Deploy & Run Transactions icon on the left sidebar (it looks like a "play" button).
2) Under the Environment dropdown, select JavaScript VM (London). This creates a simulated blockchain environment.
3) Click on the Deploy button. This will deploy your contract to the simulated environment.

Step 6: Interact with the Contract

1) After deployment, you will see your contract listed under Deployed Contracts.
2) Expand the contract's interface to view its functions.

Step 7: Execute Functions

Deposit Funds:

- In the Value field, input the amount of Ether you want to deposit (e.g., 1 for 1 Ether).
- Click on the deposite function. This will execute the deposit.

View Balance:

- Click on the viewBalance function. This will return your current balance in the bank.

Withdraw Funds:

- In the input field next to the withdraw function, specify the amount you want to withdraw (e.g., 0.5 for 0.5 Ether).
- Click on the withdraw function to execute the withdrawal.

Step 8: Check Events

- After each transaction, check the Logs section in Remix to view emitted events (like LogDepositeMade).

Conclusion: - In this way we have explored Smart Contract and Implement smart contract for banking application**.**

Reference Link: -

- https://www.simplilearn.com/solidity-interview-questions-article
- https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7a.js
- https://docs.soliditylang.org/en/v0.8.27/

Assignment Question

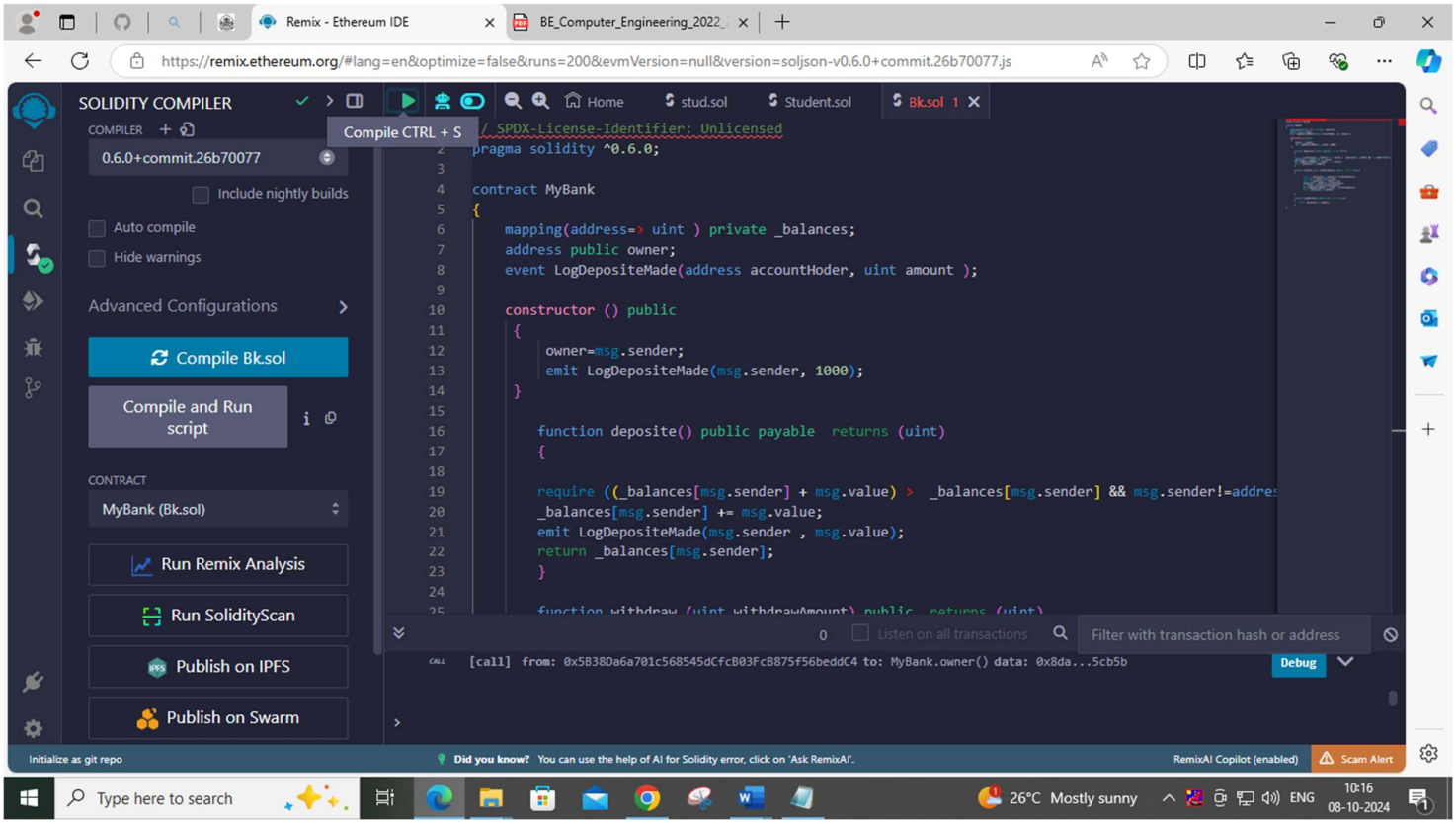
1. **What is Solidity and Explain Solidity Struct?**
2. **What is a smart contract in Blockchain?**
3. **What are the main differences between Solidity and other programming languages like Python, Java, or C++?**
4. **What is EVM bytecode?**
5. **Explain Solidity Compilation process in details?**

Code:

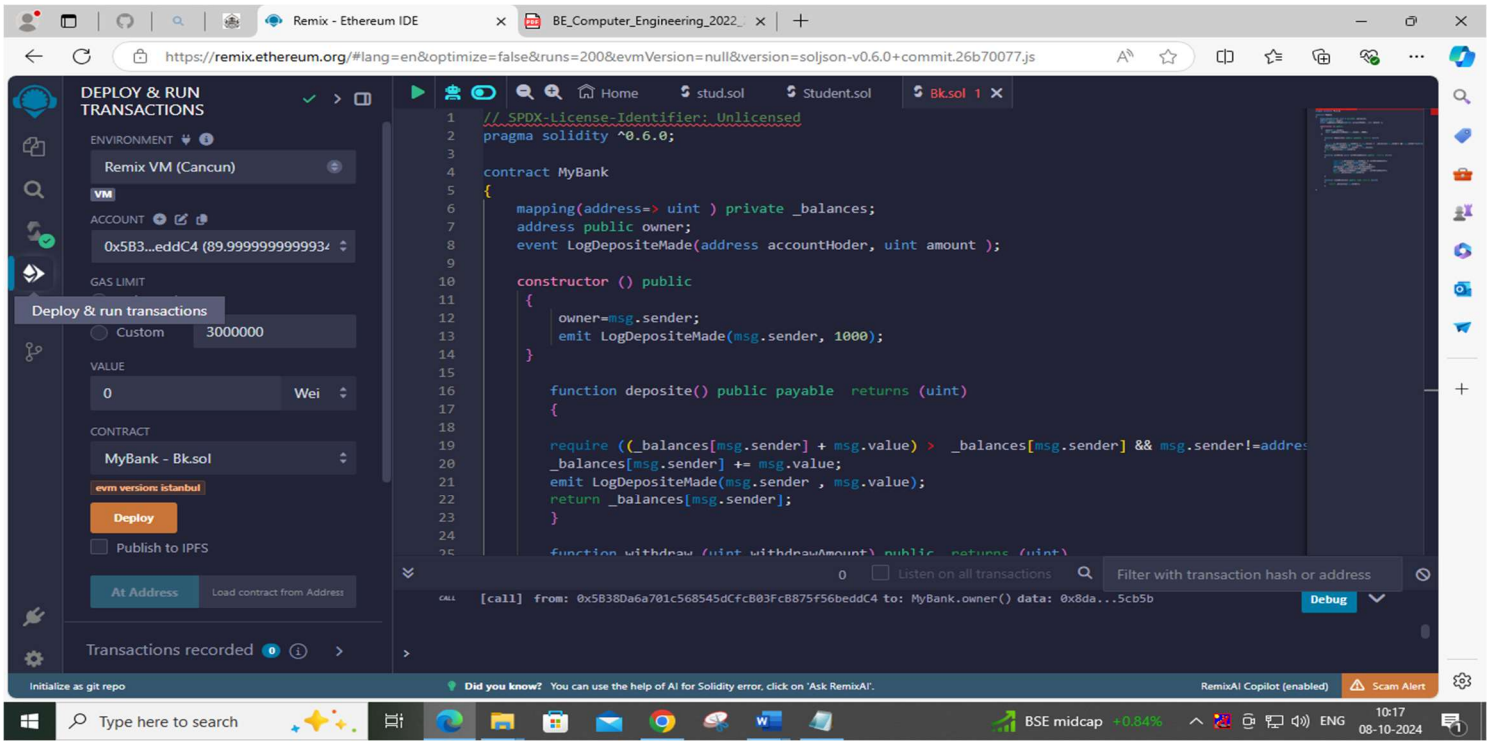
```
pragma solidity ^0.6.0;
contract MyBank
{
   mapping(address=> uint ) private _balances;
   address public owner;
   event LogDepositeMade(address accountHoder, uint amount );
   constructor () public
    {
       owner=msg.sender;
       emit LogDepositeMade(msg.sender, 1000);
    }
      function deposite() public payable  returns (uint)
      {
      require ((_balances[msg.sender] + msg.value) >  _balances[msg.sender] && msg.sender!=address(0));
      _balances[msg.sender] += msg.value;
      emit LogDepositeMade(msg.sender , msg.value);
      return _balances[msg.sender];
      }
      function withdraw (uint withdrawAmount) public  returns (uint)
      {     require (_balances[msg.sender] >= withdrawAmount);
            require(msg.sender!=address(0));
            require (_balances[msg.sender] > 0);
            _balances[msg.sender]-= withdrawAmount;
            msg.sender.transfer(withdrawAmount);
            emit LogDepositeMade(msg.sender , withdrawAmount);
            return _balances[msg.sender];
      }
      function viewBalance() public view returns (uint)
      {
         return _balances[msg.sender];
      }
  }
```

Output: -

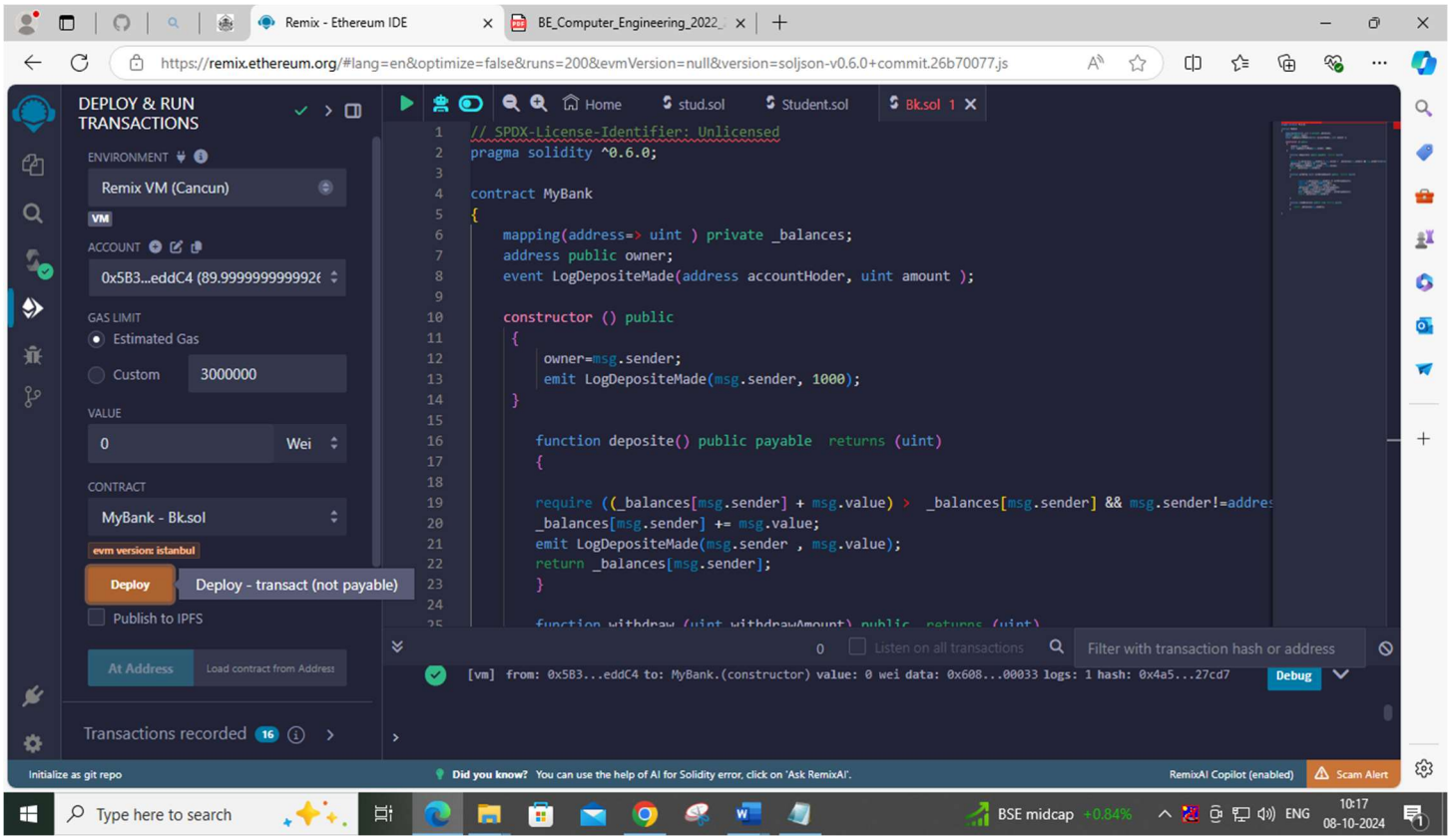
Step 1 – Compile Bank. sol



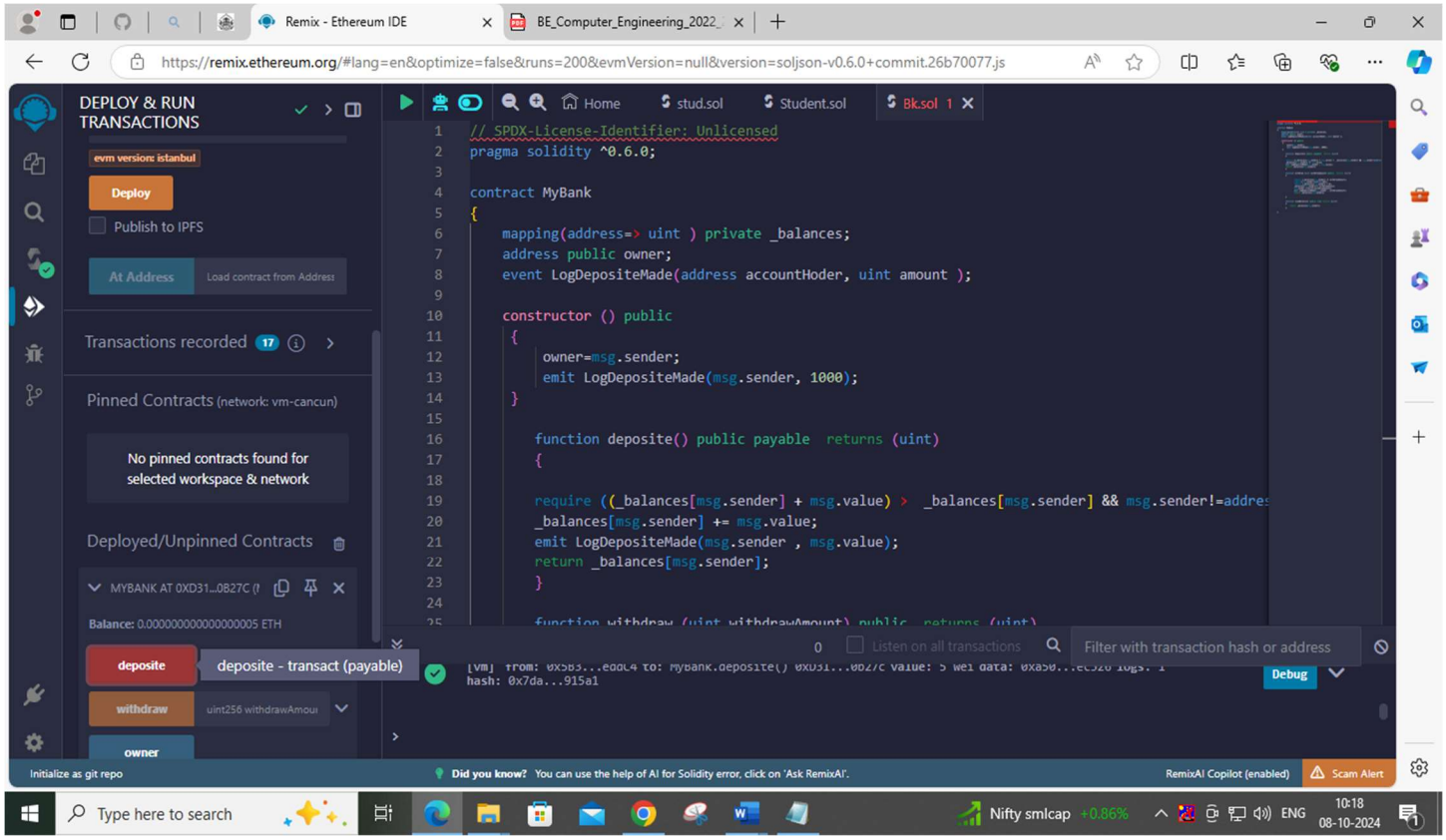
Step 2:- Deploy and run



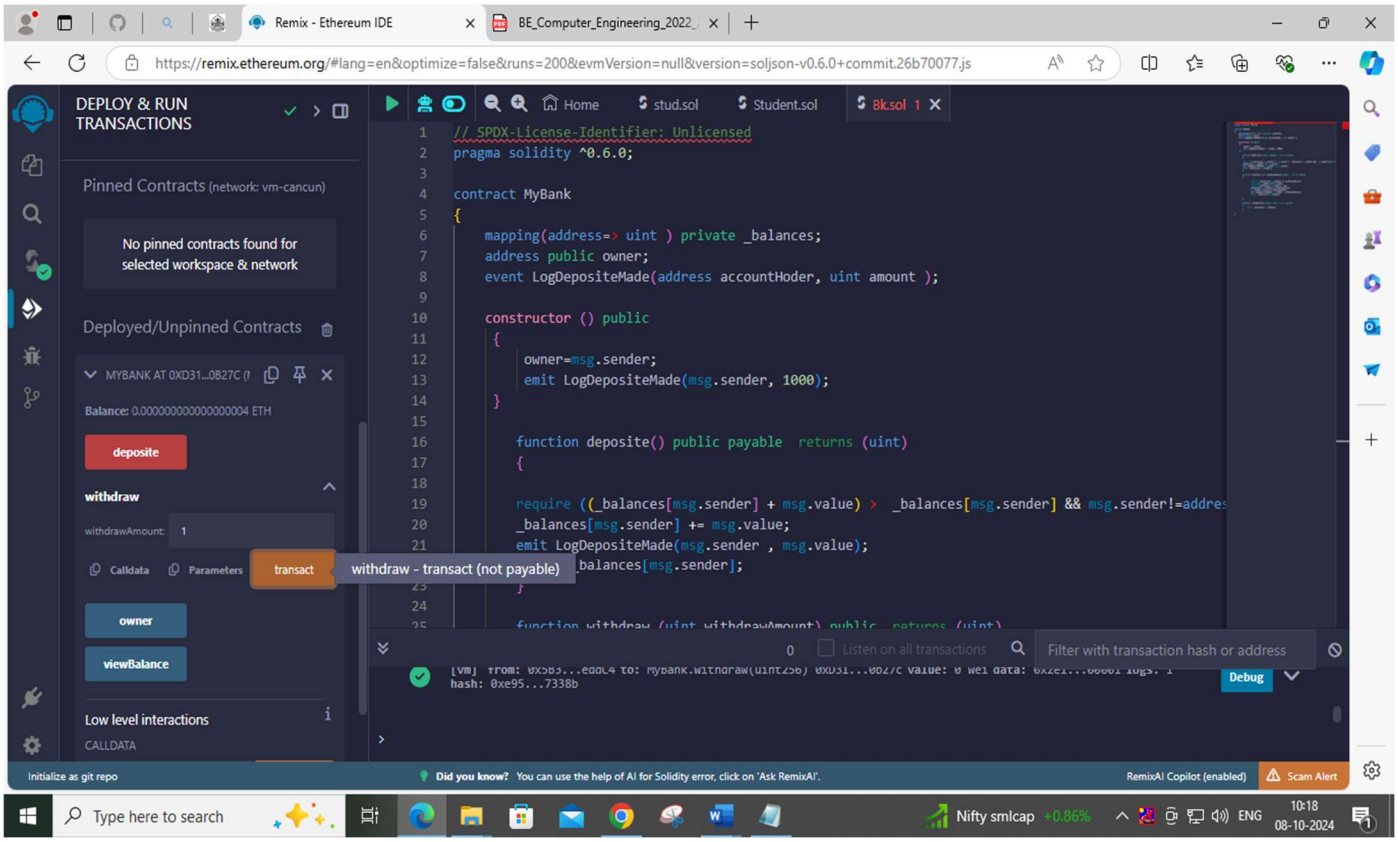
Step 03: - Deploy Contract



Step 04: - Deposit 5 Wei



Step 05: - Withdraw 5 Wei



Step 06: - Show Balance