Practical 3

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

- Deposit money
- Withdraw Money
- Show balance

Objective:

- Get familiar with solidity language with the help of few simple programs

Requirement:

- Remix IDE

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).
- We can use Solidity to create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

The Ethereum Virtual Machine (EVM):

- The Ethereum Virtual Machine, also known as EVM, is the runtime environment for smart contracts in Ethereum.
- In Java, code is compiled into bytecode which runs on the JVM, independent of the underlying hardware.
- In Ethereum, smart contracts are compiled into EVM bytecode, which runs identically on every node's EVM, regardless of hardware or OS.

Smart Contracts: Self-executing contracts with the terms of the agreement directly written into code. They automatically execute transactions or operations when certain conditions are met.

Remix IDE: Remix IDE is a web-based Integrated Development Environment (IDE) designed specifically for writing, deploying, and testing Solidity smart contracts on the Ethereum blockchain. It's an easy-to-use tool that provides developers with all the essential features to work with Ethereum smart contracts without needing to install software locally.

Accessing Remix IDE: You can access Remix IDE through this URL: https://remix.ethereum.org.

Key Concepts in Solidity:

1. Contract

- **In Solidity:**
  - A **contract** is like a **class** in object-oriented languages.
  - It contains variables (data), functions (logic), events, and modifiers.
  - Everything in Ethereum (smart contracts) is built inside contracts.
- **In other languages:**
  - Like a **class in Java or C++**, where you define attributes and methods.
  - Difference: In Solidity, contracts are **deployed on blockchain** and have their own address.

2. State Variable

- **In Solidity:**
  - A **state variable** is a variable stored on the **blockchain permanently**.
  - Example: uint public count; (This will always live on blockchain until contract is destroyed).
  - Changing a state variable costs **gas (money)**.
- **In other languages:**
  - Like a **global or class-level variable** in Java/C++ that stays in memory.
  - Difference: In Solidity, state variables are stored on the **blockchain**, not in temporary RAM.

3. Function

- **In Solidity:**
  - A **function** is a piece of code that does something (calculations, updates, transfers).
  - Functions can read/write state variables or just return values.
  - Some functions are free (read-only/view), while others cost gas (if they change blockchain state).
- **In other languages:**
  - Like a **method/function** in Python, Java, or C++.
  - Difference: In Solidity, functions that **change data** on blockchain need a **transaction**.

4. Event

- **In Solidity:**
  - An **event** is like a **log message** stored on the blockchain.
  - Used to notify external applications (like front-end dApps) when something happens.
  - Example: event Transfer(address from, address to, uint amount);
- **In other languages:**
  - Similar to **print/log statements**, or **signals in frameworks**.
  - Difference: Normal logs disappear when the program ends, but Solidity events are **permanently recorded** on the blockchain.

5. Modifier

- **In Solidity:**
  - A **modifier** is like a **function decorator** that adds extra rules to functions.
  - Example: Restricting a function so only the owner can call it.
- **In other languages:**
  - Similar to **annotations in Java (@Override)** or **decorators in Python (@login_required)**.
  - Difference: In Solidity, modifiers are often used for **access control and validation**.

So, Solidity feels like **Java/C++ classes**, but instead of running on a normal computer, everything is **stored and executed on the blockchain** with costs involved.

Basic Syntax in Solidity:

1. Version Declaration

Every Solidity file begins with a version declaration to ensure compatibility:

```
pragma solidity ^0.8.0;
```

This ensures the Solidity compiler only compiles the code if it's version 0.8.0 or higher.

2. Contract Declaration

A contract is declared using the `contract` keyword:

```
contract HelloWorld {
    // Code goes here
}
```

3. State Variables

Variables(integer, string, boolean etc.) that store data persistently on the blockchain:

```
string  greeting;
```

Here, `greeting` is a state variable, and `public` means it can be accessed from outside the contract.

4. Constructor

A constructor is a special function executed once during contract deployment:

```
constructor() {
    greeting = "Hello, World!";
}
```

This constructor initializes the `greeting` variable with "Hello, World!" during contract deployment.

5. Functions

Functions are used to execute logic and can modify or retrieve contract data:

```
function sayHello() public view returns (string memory) {
    return greeting;
}
```

Here, the `sayHello` function is public, and it reads (`view`) the `greeting` variable without modifying the blockchain.

Complete "Hello World" Example 1:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HelloWorld {
    string public MyString = "hello world";
}
```

Complete "Hello World" Example 2:

```
pragma solidity ^0.8.0;

contract HelloWorld {
```

```
    string public greeting;

    constructor() {
        greeting = "Hello, World!";
    }

    function sayHello() public view returns (string memory) {
        return greeting;
    }
}
```

Explanation:

1. **SPDX License Identifier**: Optional but recommended, specifying the licensing for the code.
2. **Version Declaration**: The contract uses Solidity version 0.8.0 or higher.
3. **State Variable**: `greeting` holds the message.
4. **Constructor**: Initializes the `greeting` variable.
5. **Function `sayHello`**: Returns the message stored in `greeting`.
6. **Return:** The `returns` keyword specifies the data type that the function will return after execution. In this case, the function is expected to return a value of type `string`.
7. **String:** `string` is a Solidity data type that stores text data. In the case of `sayHello()`, the function will return a `string` that holds the value of the variable `greeting`.
8. **Memory**: Used for temporary data during function execution. Variables stored in memory are erased once the function call ends.
9. `public`: The function can be called externally and internally.
10. `view`: Indicates that this function does not modify the blockchain state (it only reads from it).
11. `returns (string memory)`: The function returns a `string` that is temporarily stored in memory, not on the blockchain.
12. `return greeting;`: The actual string stored in the `greeting` variable is returned when the function is called.

This is a basic structure to understand how Solidity works.

Example 2: Add 2 Numbers

```
pragma solidity ^0.8.0;
```

```
contract AddTwoNumbers {

    // Function to add two numbers
    function add(int num1, int num2) public pure returns (int) {
        return num1 + num2;
    }
}
```

Explanation:

- **Parameters (`int num1, int num2`)**: The function takes two signed integers (`int`) as inputs.
- **public**: The function can be called by external contracts or users.
- **pure**: Indicates that the function neither reads nor modifies the blockchain state (it just performs an operation on the input values).
- **returns (int)**: The function returns a signed integer (`int`), which is the sum of `num1` and `num2`.
- **`return num1 + num2;`**: This adds the two numbers and returns the result.

Example 3: Write a simple smart contract for Bank account of a customer for following
operations:

- Deposit Money
- Withdraw Money
- Show Balance.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BankAccount {

    // State variable to store the account balance
    uint public balance;

    // Constructor to initialize the account with an initial balance (optional)
    constructor() {
        balance = 0; // Initial balance is set to 0
    }

    // Function to deposit money into the account
    function deposit(uint amount) public {
        require(amount > 0, "Deposit amount must be greater than zero.");
        balance += amount; // Increase the balance by the deposit amount
```

```
    }

    // Function to withdraw money from the account
    function withdraw(uint amount) public {
        require(amount > 0, "Withdraw amount must be greater than zero.");
        require(balance >= amount, "Insufficient balance.");
        balance -= amount; // Decrease the balance by the withdrawal amount
    }

    // Function to check the current balance
    function getBalance() public view returns (uint) {
        return balance;
    }
}
```

Explanation:

- **pragma**: Used to define the version of the Solidity compiler.
- **contract**: This keyword is used to define a smart contract in Solidity.

- **uint**: An Unsigned integer variable (Can store only zero or positive numbers).
- **public**: This allows anyone (users or other contracts) to access and view the variable.
- **balance**: This is the state variable that stores the account's current balance.

- **constructor()**: A special function that runs only once, when the contract is first deployed. Used to initialize state variables.

- **function deposit**: A function to allow users to add money to the account.
- **(uint amount)**: Takes one unsigned integer as an input, representing the amount to deposit.
- **public**: Can be called by external users or contracts.

- **require**: A condition that must be true; otherwise, the function fails.
- **amount > 0**: Ensures that the deposit amount is positive.
- **"..."**: The error message shown if the condition fails.

- **function withdraw**: A function to allow users to withdraw money from the account.

- **function getBalance**: A function to check the current balance.
- **public**: Anyone can call this function.
- **view**: Means the function only **reads** the blockchain data, does not change it.
- **returns (uint)**: This function will return an integer, the current balance.

- **return**: Gives the balance value back to the caller.

Conclusion:

We have learned how to write, compile, deploy and run a smart contract in solidity language using remix IDE.