# Group C: Assignment No 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

**Objectives:**

1. Concept of Smart Contract

2. for Bank account of a customer

**Theory: -**

**Writing a Banking Contract**

This article will demonstrate how to write a simple, but complete, smart contract in Solidity that acts like a bank that stores ether on behalf of its clients. The contract will allow deposits from any account, and can be trusted to allow withdrawals only by accounts that have sufficient funds to cover the requested withdrawal. This post assumes that you are comfortable with the ether-handling concepts introduced in our post, writing a Contract That Handles Ether.

That post demonstrated how to restrict ether withdrawals to an —owner ‘s account. It did this by persistently storing the owner account‘s address, and then comparing it to the msg.sender value for any withdrawal attempt. Herer‘s a slightly simplified version of that smart contract, which allows anybody to deposit money, but only allows the owner to make withdrawals:

```
pragma solidity ^0.4.19;

contract TipJar {

address owner; // current owner of the contract

function TipJar() public {
```

```
owner = msg.sender;
}
function withdraw() public {
require(owner == msg.sender);
msg.sender.transfer(address(this).balance);
}
function deposit(uint256 amount) public payable {
require(msg.value == amount);
}
function getBalance() public view returns (uint256) {
return address(this).balance;
}
}
```

**Maintaining Individual Account Balances**

I am going to generalize this contract to keep track of ether deposits based on the account address of the depositor, and then only allow that same account to make withdrawals of that ether. To do this, we need a way keep track of account balances for each depositing account—a mapping from accounts to balances. Fortunately, Solidity provides a ready-made mapping data type that can map account addresses to integers, which will make this bookkeeping job quite simple. (This mapping structure is much more general key/value mapping than just addresses to integers, but that's all we need here.)

Here 's the code to accept deposits and track account balances:

```
pragma solidity ^0.4.19;
contract Bank {
mapping(address => uint256) public balanceOf; // balances, indexed by addresses
function deposit(uint256 amount) public payable {
```

```
require(msg.value == amount);

balanceOf[msg.sender] += amount; // adjust the account's balance

}

}
```

**Here are the new concepts in the code above:**

mapping(address => uint256) public balanceOf; declares a persistent public variable, balanceOf, that is a mapping from account addresses to 256-bit unsigned integers. Those integers will represent the current balance of ether stored by the contract on behalf of the corresponding address.

Mappings can be indexed just like arrays/lists/dictionaries/tables in most modern programming languages.

The value of a missing mapping value is 0. Therefore, we can trust that the beginning balance for all account addresses will effectively be zero prior to the first deposit.

It‘s important to note that balanceOf keeps track of the ether balances assigned to each account, but it does not actually move any ether anywhere. The bank contract‘s ether balance is the sum of all the balances of all accounts—only balanceOf tracks how much of that is assigned to each account. Note also that this contract doesn‘t need a constructor. There is no persistent state to initialize other than the balanceOf mapping, which already provides default values of 0.

**Withdrawals and Account Balances**

Given the balanceOf mapping from account addresses to ether amounts, the remaining code for a fully-functional bank contract is pretty small. I‘ll simply add a withdrawal function:

**bank.sol**

```
pragma solidity ^0.4.19;

function withdraw(uint256 amount) public {

require(amount <= balanceOf[msg.sender]);

balanceOf[msg.sender] -= amount;

msg.sender.transfer(amount);

}
```

}

The code above demonstrates the following:

The require(amount <= balances[msg.sender]) checks to make sure the sender has sufficient funds to cover the requested withdrawal. If not, then the transaction aborts without making any state changes or ether transfers.

The balanceOf mapping must be updated to reflect the lowered residual amount after the withdrawal.

The funds must be sent to the sender requesting the withdrawal.

**Important: Avoiding the Reentrancy Vulnerability**

In the withdraw() function above, it is very important to adjust balanceOf[msg.sender] **before** transferring ether to avoid an exploitable vulnerability. The reason is specific to smart contracts and the fact that a transfer to a smart contract executes code in that smart contract. (The essentials of Ethereum transactions are discussed in How Ethereum Transactions Work.)

Now, suppose that the code in withdraw() did not adjust balanceOf[msg.sender] before making the transfer *and* suppose that msg.sender was a malicious smart contract. Upon receiving the transfer—handled by msg.sender‘s fallback function—that malicious contract could initiate *another* withdrawal from the banking contract. When the banking contract handles this second withdrawal request, it would have already transferred ether for the original withdrawal, but it would not have an updated balance, so it would allow this second withdrawal!

This vulnerability is called a —reentrancy  bug because it happens when a smart contract invokes code in a different smart contract that then calls back into the original, thereby reentering the exploitable contract. For this reason, it’s essential to always make sure a contract ‘s internal state is fully updated before it potentially invokes code in another smart contract. (And, it’s essential to remember that every transfer to a smart contract executes that contract ‘s code.)

**Conclusion: We have studied a smart contract on a test network for Bank account of customer operations.**

# Group C: Assignment No 4

**Aim:** Write a program in solidity to create Student data. Use the following constructs:

Structures

Arrays

Fallback

Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.

**Objectives:**

1. Concept of solidity

2. smart contract on Ethereum and Observe the transaction fee and Gas values

**Theory:**

**Introduction**

Blockchain is a decentralized, distributed public ledger that lets us collaborate and coordinate the members that do not trust each other to make a secure transaction. Many of you understand blockchain as a bitcoin, but bitcoin is a cryptocurrency that takes the help of blockchain technology to operate.

**Blockchain Technology**

First, when Windows was launched to create reports and work on any project, we used MS word, where only one person could edit at a time and then send to another, and the process goes on, one after the other method. After some technological evolution, we have seen Google docs, google sheets in the market where online multiple people can work on a single document simultaneously. But the problem here is that it works on centralized architecture where a single server maintains google docs and various nodes are connected. Still, if the significant server crashes or gets corrupt, all the nodes get disconnected, and all the work gets destroyed. So the solution to this is decentralized blockchain technology. Decentralized means that no single

server or single node is managing the network. Data is replicated to multiple nodes so that if any node goes down, other nodes operate as it is, and original data can quickly be recovered.

**What is Ethereum?**

Ethereum is a decentralized blockchain designed to be highly secure, fault-tolerant, and programmable. Ethereum blockchain is a choice for many developers and businesses. As said programmable, the main task of Ethereum is to securely execute and verify the application code known as smart contracts. Ethereum helps to build native scripting language(solidity) and EVM. Ethereum consensus mechanism is proof of work to operate to verify the new transaction. Now we will learn about smart contracts and how it runs on the Ethereum platform.

**Overview of Smart Contracts**

A smart contract is a small program that runs on an Ethereum blockchain. Once the smart contract is deployed on the Ethereum blockchain, it cannot be changed. To deploy the smart contract to Ethereum, you must pay the ether (ETH) cost. Understand it as a digital agreement that builds trust and allows both parties to agree on a particular set of conditions that cannot be tampered with.


Learn | Write | Earn

To understand the need for a smart contract, suppose there was one grocery shop, and ram went to buy some groceries. He purchased the groceries for 500 rupees and kept on debt that would pay the money next month when he returned, so the shopkeeper jotted down his purchase in his ledger. In between the period somehow shopkeeper changed 500 to 600 and when next month ram went to pay the money, the shopkeeper has demanded 600 INR and ram has no proof to show that he has only bought 500 INR so in this case, smart contracts play an essential role which prevents both the parties to tamper the agreement and only gets terminate when all the conditions satisfy after the deal. There are a couple of languages to write smart contracts, but the most popular is solidity.

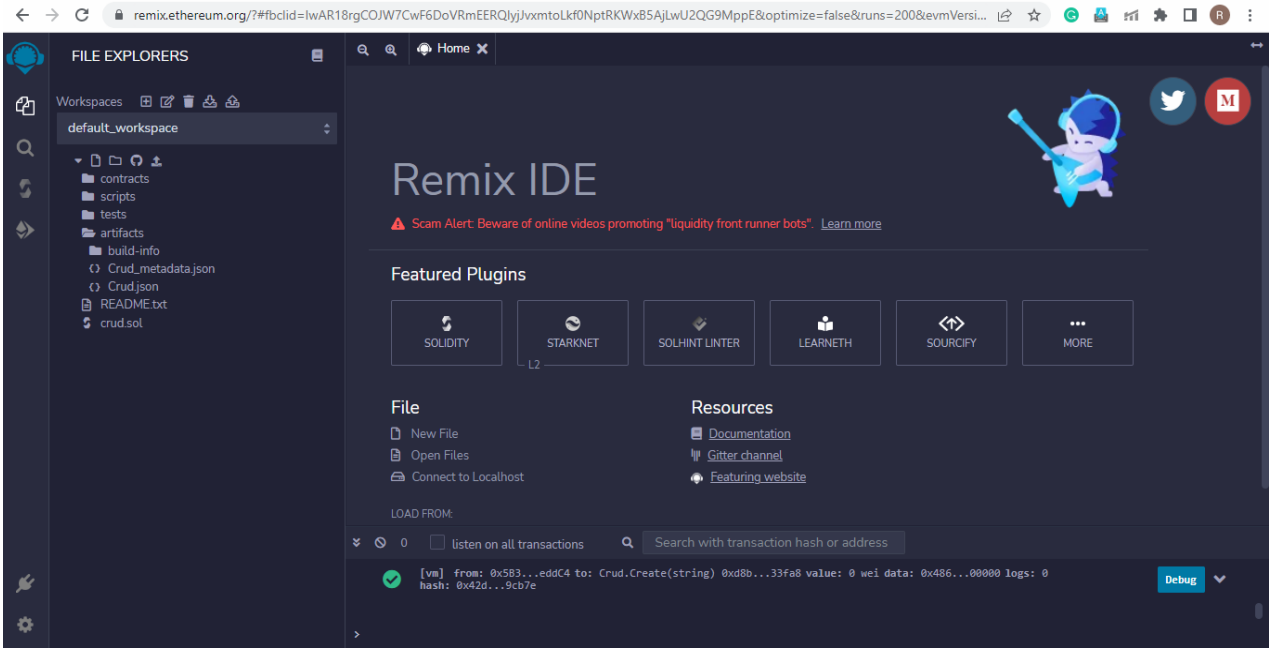
**Introduction to Solidity Programming**

Solidity is object-oriented, high-level statically-typed programing language used to create smart contracts. Solidity programming looks similar to Javascript, but there are a lot of differences

between both languages. In solidity, you need to compile the program first, while in Javascript, you can run the program directly in your browser or by using Node JS. With solidity, you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets. It is also a case-sensitive programming language. Visit the official solidity documentation to read more and be updated about any new functionality release.

**What is Remix IDE?**

It is an online IDE for creating solid, smart contracts, so you do not need to install or download anything to do any setup. You can develop, deploy, and Administer your solidity smart contract using Remix IDE. Visit this link to access the Remix IDE where you will find multiple options and a window shown below. The window is a little bit similar to VS code where on the left-hand side, you will find some icons to terminate to other options like a compiler, file explorer, search files, deploy, etc.




Using the file explorer, you can create and open any file. We can set the compiler version, run our smart contract, and observe the output using the compiler. Each compiler type provides a different amount of fake ethers used for practicing purposes. **Solidity Compilation Process** Smart contract compilation is a critical process to understand how a smart contract runs when created using solidity. We will understand the process using the below flow chart. We can see that the smart contract written in solidity with sol extension first gets the compiler version. After it goes under the compiler, It gets split into two parts where one is Byte code, and the other is ABI (Abstract Binary Interface) key. Byte code is only executed and deployed on the

Ethereum blockchain, not the complete smart contract. Whenever any smart contract wants to communicate with this smart contract, they need the ABI key to call functions and variables.

To observe how ABI and Byte code is generated on Remix IDE, visit IDE, open any contract in the contracts folder, and compile and run it. Scroll down, and you will find two options: ABI and Byte code, where you can copy and paste them into any notepad and observe how your code gets converted to Byte code.

To create a smart contract, the first thing is to define the compiler version to use using the Pragma keyword (you can also determine whether the program supports multiple versions or the version in a particular range); after this, you define the contract using the contract keyword which is same as creating a class in object-oriented programming.

Important points related to smart contract Compilation

1. Contract Bytecode is public in readable form – It means It does not get encrypted because It will run on different nodes of Ethereum. For then, It needs to decrypt again and again not to increase computation time. It is kept in a readable form.
2. The contract doesn‘t have to be public – It does not need to keep contracts public, but most organizations keep them public to maintain the trust.
3. Bytecode is immutable

4. ABI act as a bridge between application and smart contract

5. ABI and bytecode cannot be generated without source code

State and Local Variables in Solidity

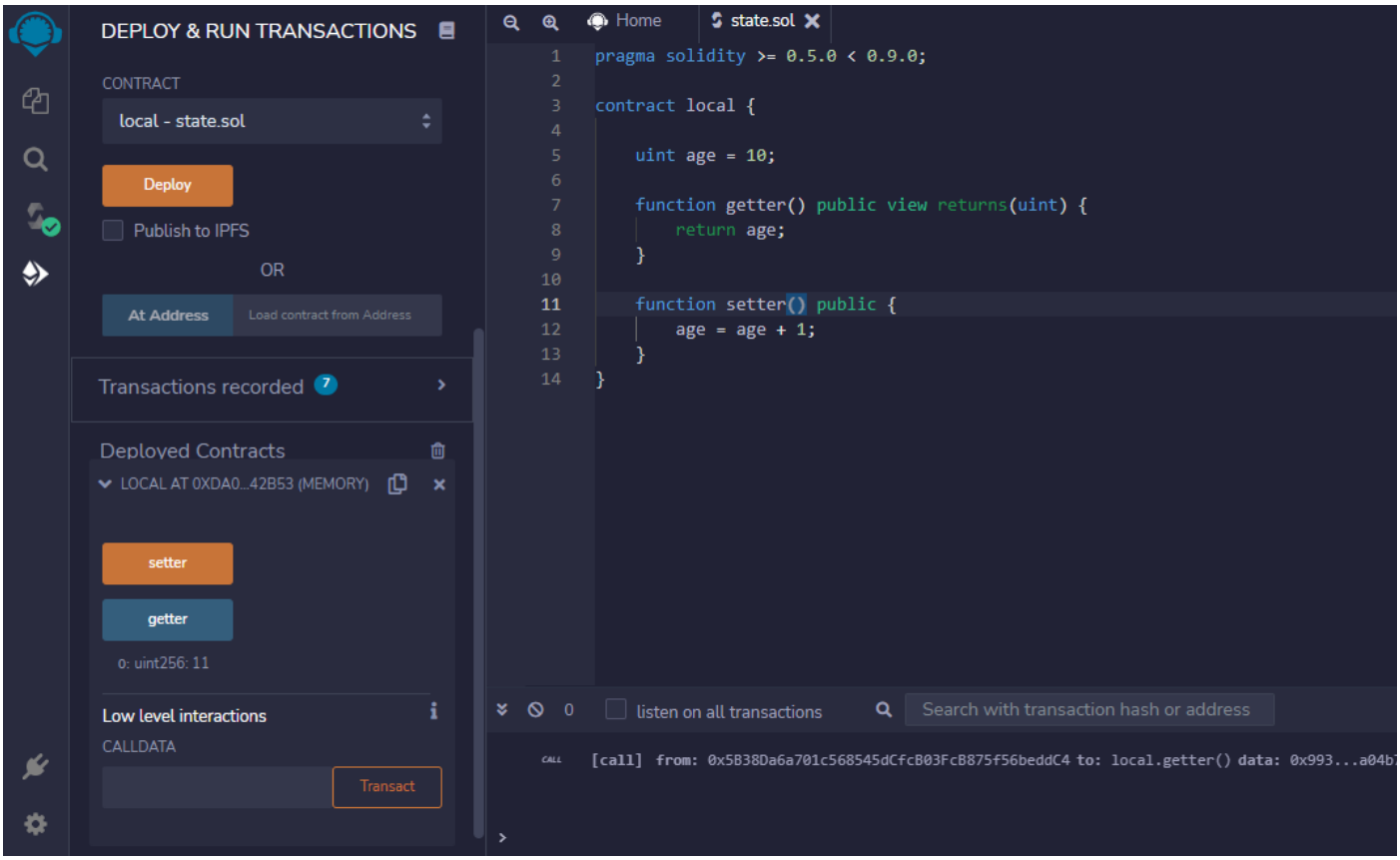
Any variable declared on the contract level is known as a state variable. The critical property of the state variable is that it is permanently stored in the blockchain, so you have to pay some amount of gas and use the state variable with care. Solidity does not have a concept of Null or None; indeed, each data type has one default value which on declaration is assigned to that variable. To define Public before any variable or function, automatically, one get function is set with that variable, and you can access its value. Storage to state variable is not dynamically allocated (To initialize state variable with the value, you need to assign a value at declaration time, use constructor, use getter and setter functions). An instance of a contract variable cannot have another state variable besides those already declared.Local variables are those variables that are declared in the function body and are stored in a stack, not in contract storage. Local variables don‘t cost gas; some types reference the storage by default. Memory keywords cannot be used at the contract level.

**Functions in Solidity**

Functions are an essential part of any programming language for the reusability of a particular code. We will see the getter and setter function in solidity to learn how to create a function in solidity. The getter function is a function from which we can access the value of our variables. It is a view-only function, so we can define it as a view or Pure, which states that the value of the state variable cannot be changed it returns the variable ‘s value, so we define the return type of value. On the other side, the setter function changes the value, so it is a simple public function.

```
pragma solidity >= 0.5.0 < 0.9.0;

contract local {

uint age = 10;

function getter() public view returns(uint) {

return age;

}

function setter() public {

age = age + 1;

}

}
```

After writing the above code on Remix IDE in a new file with sol extension, you can compile the code, visit the deploy section, and deploy the code to observe the deploy section output as shown below. The value will increase as you click the setter and getter function buttons.




Suppose you want to implement the setter function to set the new value of age so we can pass the parameter in the setter function and set the value of age. Thus, in the setter function, we change the matter, so we need to pay a certain amount of gas in the setter function, but the getter function is view-only and does not require any amount of gas to be paid. By default, the visibility of a function is private, so to make it public, we define a function as Public.

**Pure and View in Solidity**

We have seen that we use to view and pure where we are not updating the state variable. But pure, you cannot use where it is also reading the state variable. Pure is used where both reading and writing are not performed. Indeed, in View, reading is allowed, but writing is not permitted. When we do not define any one of the following to a function, it simply warns that we can provide one restriction of pure or View to function.

**Constructor in Solidity**

A constructor is a particular type of function which executes only once when you create your contract. Constructor is used to work with state variables and define smart contract's owners. You can create only one constructor, and it is optional to create. The compiler creates a default constructor if there is no explicitly defined constructor.

```
pragma solidity >= 0.5.0 < 0.9.0;

contract local {

uint public count;

constructor(uint new_count) {

count = new_count;

} }
```

the above code, we have created a constructor, and before clicking on deploy, you need to define the value of the constructor before it is called only once, so enter the value and click on deploy, and on scrolling, you can observe the value of count.

**Control Statements in Solidity**

All programming languages have control statements that help us check multiple conditions using loops and an if-else ladder, and solidity also supports loops and if-else statements.

Loops in Solidity

Solidity also supports three loops: a while loop, a Do while loop, and a loop. If you are familiar with any other programming language, you must know about control statements and using loops to run particular code multiple times with different values. In solidity, you cannot write the loops directly in contract storage; instead, you must declare them in any function.

While the loop runs a code snippet multiple times until the condition is proper, the loop terminates when the condition is false. In contrast, the loop runs 0 or multiple times.

Do while loop is a loop that runs even one time when the condition in the while loop is false. So it is used when you need to run a particular code at least once and if certain conditions meet, then run it multiple times.

For loop is a loop that is used when you know the start and end time of the loop and how many intervals you need to take. For loop, the initialization and iterator updating are part of loop syntax.

So let us look after the syntax of each type of loop using a sample program.

```
pragma solidity >= 0.5.0 < 0.9.0;

contract Loops {

uint [3] public arr;

uint public count;

function Whileloop() public {

while(count < arr.length) {

arr[count] = count;

count++;

} }

function Forloop() public {

for(uint i=count; i<arr.length; i++) {

arr[count] = count;

count++;

} }

function doWhileLoop() public {

do {

arr[count] = count;

count++;

}while(count < arr.length);

} }
```

**If-else Statements in Solidity**

If-else statements are an essential part of any programming language that helps compare two or more two types of values to make a particular decision. Below is the sample code snippet denoting the use of if-else in the solidity that you should try and deploy the contract. After deploying, check by entering the different values.

```
pragma solidity >= 0.5.0 < 0.9.0;
```

```
contract Array {
function check(int a) public pure returns(string memory) {
string memory value;
if(a > 0) {
value = "Greater Than zero";
}
else if(a == 0) {
value = "Equal to zero";
}
else {
value = "Less than zero";
}
return value;
} }
```

**Arrays in Solidity**

The array is a special data structure used to create a list of similar type values. The array can be of fixed size and dynamic-sized. With the help of index elements can be accessed easily. below is a sample code to create, and access a fixed-sized array in solidity.

```
pragma solidity >= 0.5.0 < 0.9.0;
contract Array {
uint [4] public arr = [10, 20, 30, 40];
function setter(uint index, uint value) public {
arr[index] = value;
}
function length() public view returns(uint) {
return arr.length;
} }
```

You can compile and deploy the code to try changing the array elements with an index and printing the array length.

**Creating Dynamic Array**

A dynamic array is an array where we can insert any number of elements and delete the details easily using an index. So solidity has functions like push and pops like python, making it easy to create a dynamic array. Below is a code using which you can create a dynamic array. After writing code, compiles and deploy the code by visiting the deploy section in the left-side navigation bar. After that, try inserting and deleting some elements from an array.

```
pragma solidity >= 0.5.0 < 0.9.0;

contract Array {

uint [] public arr;

function PushElement(uint item) public {

arr.push(item);

}

function Length() public view returns(uint) {

return arr.length;

}

function PopElement() public {

arr.pop();

} }
```

Structure in Solidity

The structure is a user-defined data type that stores more than one data member of different data types. As in array, we can only store elements of the same data type, but in structure, you can keep elements of different data types used to create multiple collections. The structure can be made outside and inside the contract storage, and the Structure keyword can be used to declare the form. The structure is storage type, meaning we use it in-store only, and if we want to use it in function, then we need to use the memory keyword as we do in the case of a string.

```
pragma solidity >= 0.5.0 < 0.9.0;

struct Student {
```

```
uint rollNo;

string name;

}

contract Demo {

Student public s1;

constructor(uint _rollNo, string memory _name) {

s1.rollNo = _rollNo;

s1.name = _name;

}

// to change the value we have to implement a setter function

function changeValue(uint _rollNo, string memory _name) public {

Student memory new_student = Student( {

rollNo : _rollNo,

name : _name

});

s1 = new_student;

} }
```

**Create a Smart Contract with CRUD Functionality**

We have excellent theoretical and hands-on practical knowledge about solidity, and now you can create a primary smart contract like hello world, getter, and setter contracts. So it‘s a great time to try making some functional smart contracts, and the best way to try all the things in one code is to create one program that performs all CRUD operations.

```
pragma solidity ^0.5.0;

contract Crud {

struct User {

uint id;

string name;
```

```
}
User[] public users;
uint public nextId = 0;
function Create(string memory name) public {
users.push(User(nextId, name));
nextId++;
}
function Read(uint id) view public returns(uint, string memory) {
for(uint i=0; i<users.length; i++) {
if(users[i].id == id) {
return(users[i].id, users[i].name);
} } }
function Update(uint id, string memory name) public {
for(uint i=0; i<users.length; i++) {
if(users[i].id == id) {
users[i].name =name;
} } }
function Delete(uint id) public {
delete users[id];
}
function find(uint id) view internal returns(uint) {
for(uint i=0; i< users.length; i++) {
if(users[i].id == id) {
return i;
} }
// if user does not exist then revert back
```

```
revert("User does not exist");

} }
```

**Conclusion:** We have studied program writing in solidity to create Student data by using constructs like structures, arrays, fallback and also deployed this as smart contract on Ethereum and Observe the transaction fee and Gas values.