



### Experiment No. 3

#### Aim:

Creating Transactions using Solidity and Remix IDE

#### Theory:

- Consider a smart contract for basic banking operations. This contract includes all of the functionalities and capabilities that Solidity presents. Also, it demonstrates about how to send ETH between any account and the contract developed (from an account to a contractor from a contract to an account) and how to restrict the people who can use the relevant function of the smart contract.
- Create a client object to keep the client's information, which will join the contract by using the struct element. It keeps the client's ID, address, and balance in the contract. Then we create an array in the client\_account type to keep the information of all of our clients.
- Assign an ID to each client whenever they join the contract, so we define an `int` counter and set it to 0 in the constructor of the contract.
- Define an address variable for the manager and a mapping to keep the last interest date of each client. Since we want to restrict the time required to send interest again to any account, it'll be used to check whether enough time has elapsed.
- In a smart contract, in order to restrict the people that can call the relevant method or to allow the execution of the method only specific circumstances. In these kinds of circumstances, the modifier checks the condition you've implemented, and it determines whether the relevant method should be executed.
- Before implementing all of the methods we need to organize the smart contract, we have to implement two modifiers. Both methods will check the people who call the relevant method and which of the modifiers is used. One of them determines whether the sender is the manager, and the other one determines whether the sender is a client.
- The fallback function is essential to making the contract receive ether from any address. The `receive` keyword is new in Solidity 0.6.x, and it's used as a fallback function to receive ether. Since we'll receive ether from the clients as a deposit, we need to implement the fallback function.



- The setManager method will be used to set the manager address to variables we've defined. The managerAddress is consumed as a parameter and cast as payable to provide sending ether. The joinAsClient method will be used to make sure the client joins the contract. Whenever a client joins the contract, their interest date will be set, and the client information will be added to the client array.
- The deposit method will be used to send ETH from the client account to the contract. We want this method to be callable only by clients who've joined the contract, so the onlyClient modifier is used for this restriction. The transfer methods belong to the contract, and it's dedicated to sending an indicated amount of ETH between addresses. The payable keyword makes receipt of the ETH transfer possible, so the amount of ETH indicated in the msg.value will be transferred to the contract address.
- The withdraw method will be used to send ETH from the contract to the client account. It sends the unit of ETH indicated in the amount parameter, from the contract to the client who sent the transaction. We want this method to be callable only by clients who've joined the contract either, so the onlyClient modifier is used for this restriction. The address of the sender is held in the msg.sender variable.
- The sendInterest method will be used to send ETH as interest from the contract to all clients. We want this method to be callable only by the manager, so the onlyManager modifier is used for this restriction. Here, the last date when the relevant client takes the interest will be checked for all of the clients, and the interest will be sent if the specific timeperiod has elapsed. Finally, the new interest date is reset for the relevant client into the interestDate array if the new interest is sent. The getContractBalance method will be used to get the balance of the contract we deployed.

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.6;

contract BankContract {

struct client\_account{

int client\_id;

address client\_address;

uint client\_balance\_in\_ether;

}

client\_account[] clients;

int clientCounter;

address payable manager;

mapping(address => uint) public interestDate;

modifier onlyManager() {

require(msg.sender == manager, "Only manager can call this!");

\_;

}



---

```
modifier onlyClients() {
    bool isclient = false;
    for(uint i=0;i<clients.length;i++){
        if(clients[i].client_address == msg.sender){
            isclient = true;
            break;
        }
    }
}

require(isclient, "Only clients can call this!");

_;
```

```
}

constructor() public{
    clientCounter = 0;
}

receive() external payable { }
```

```
function setManager(address managerAddress) public returns(string memory){
    manager = payable(managerAddress);
    return "";
}

function joinAsClient() public payable returns(string memory){
    interestDate[msg.sender] = now;
    clients.push(client_account(clientCounter++, msg.sender, address(msg.sender).balance));
    return "";
}

function deposit() public payable onlyClients{
    payable(address(this)).transfer(msg.value);
}

function withdraw(uint amount) public payable onlyClients{
    msg.sender.transfer(amount * 1 ether);
}

function sendInterest() public payable onlyManager{
    for(uint i=0;i<clients.length;i++){
        address initialAddress = clients[i].client_address;
        uint lastInterestDate = interestDate[initialAddress];
        if(now < lastInterestDate + 10 seconds){
            revert("It's just been less than 10 seconds!");
        }
    }
}
```



```
}

payable(initialAddress).transfer(1 ether);
interestDate[initialAddress] = now;
}}

function getContractBalance() public view returns(uint){
    return address(this).balance;
}

}
```

### Step 1: Create BankContract.sol and Compile the Smart Contract

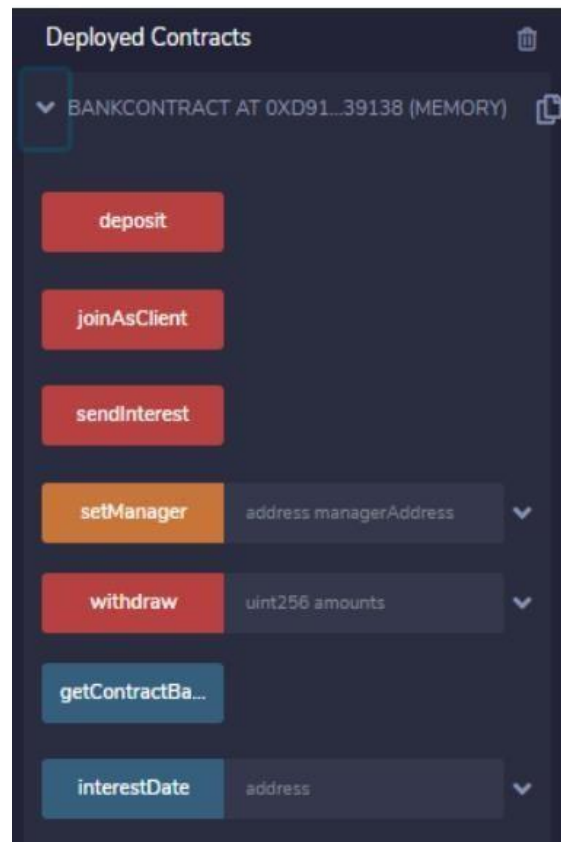
The screenshot displays the Solidity Compiler web interface. On the left, the 'SOLIDITY COMPILER' sidebar shows the compiler version as '0.6.12+commit.27d51765' and includes options for 'Auto compile' and 'Hide warnings'. Below this, there are buttons for 'Compile BankContract.sol', 'Compile and Run script', and 'Publish on Ipfs'. The main editor area on the right shows the source code for 'BankContract.sol'. The code includes a pragma statement for Solidity 0.6.6, a contract definition for 'BankContract' with a 'client\_account' struct, a 'clientCounter' variable, a 'payable' manager address, a 'public interestDate' mapping, and two modifiers: 'onlyManager' and 'onlyClients'. The 'onlyClients' modifier contains a loop that checks if the sender is a client and marks them as such. At the bottom, a transaction log shows a successful compilation event: '[vm] from: 0x5B3...eddC4 to: BankContract.(constructor) value: 0 wei data: 0x698...c0033 logs: 0 hash: 0x050...6e9'.



## Step 2: Deploy the Smart Contract.

The screenshot displays the Remix IDE interface during the deployment of a smart contract. The left sidebar, titled "DEPLOY & RUN TRANSACTIONS", shows the "BankContract" contract selected. The "Deploy" button is highlighted, and the "Publish to IPFS" checkbox is unchecked. Below the deployment options, the "Transactions recorded" section shows two transactions. The "Deployed Contracts" section lists the "BANKCONTRACT AT 0xD91...3913" with a list of functions: deposit, joinAsClient, sendInterest, setManager, withdraw, getContractBa..., and interestDate. The main editor displays the Solidity code for the "BankContract" contract, which includes a struct for client accounts, a client counter, a payable manager, and two modifiers: "onlyManager" and "onlyClients". The bottom status bar shows a successful deployment message: "[vm] from: 0x583...edc4 to: BankContract.(constructor) value: 0 wei data: 0x688...c0033 logs: 0 hash: 0x3d3...40710".

```
contract BankContract {  
    struct client account{  
        int client_id;  
        address client_address;  
        uint client_balance_in_ether;  
    }  
  
    client_account[] clients;  
  
    int clientCounter;  
    address payable manager;  
    mapping(address => uint) public interestDate;  
  
    modifier onlyManager() {  
        require(msg.sender == manager, "Only manager can call this!");  
        _;  
    }  
  
    modifier onlyClients() {  
        bool isclient = false;  
        for(uint i=0;i<clients.length;i++){  
            if(clients[i].client_address == msg.sender){  
                isclient = true;  
                break;  
            }  
        }  
        require(isclient, "Only clients can call this!");  
    }  
}
```



### Step 3: Run the Transactions

Now, we're ready to call the functions that compound the smart contract developed. When we expand the relevant contract in the Deployed Contract subsection, the methods developed appear.

#### The setManager method

We're starting to simulate a small process by calling these methods. First, we're supposed to set a manager. Therefore, we type an address that we select from the account combo and click the yellow setManager button. The following output happens in the terminal. The decoded output shows the message that returned from the method, which is an empty string message — as we expected.



The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active, displaying a list of deployed contracts and a set of buttons for interacting with the 'BankContract' at address 0xd91...3913. The buttons include 'deposit', 'joinAsClient', 'sendInterest', 'setManager', 'withdraw', and 'getContractBa...'. The right panel shows the 'BankContract.sol' file with a transaction log. The log indicates a successful transaction with the following details: status: true Transaction mined and execution succeed, transaction hash: 0x3ba5c7d1ec261710de3ef96880d707b8bbaa49bf8618660c9e9d0bcd495e38, from: 0x58380a6a701c568545dcfc883fc887f56bedd4, to: BankContract.setManager(address) 0xd9145CCE52D386f254917e481e844e9943f39138, gas: 27873, transaction cost: 24237, execution cost: 24237, input: 0xd0e...39138, decoded input: { "address managerAddress": "0xd9145CCE52D386f254917e481e844e9943f39138" }, decoded output: { "0": "string: " }, and logs: [ ].

## The joinAsClient method

We'll continue to join as a client for three clients that we determined from the account combo and call the joinAsClient method for each one. At this time — and this is different from the previous one. we should call the method while selecting related accounts because we take the msg.sender value from here. Select three addresses one by one and press joinAsClient button. We get following output.

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active, displaying a list of deployed contracts and a set of buttons for interacting with the 'BankContract' at address 0xd91...3913. The buttons include 'deposit', 'joinAsClient', 'sendInterest', 'setManager', 'withdraw', and 'getContractBa...'. The right panel shows the 'BankContract.sol' file with a transaction log. The log indicates a successful transaction with the following details: status: true Transaction mined and execution succeed, transaction hash: 0x3ba5c7d1ec261710de3ef96880d707b8bbaa49bf8618660c9e9d0bcd495e38, from: 0x58380a6a701c568545dcfc883fc887f56bedd4, to: BankContract.setManager(address) 0xd9145CCE52D386f254917e481e844e9943f39138, gas: 27873, transaction cost: 24237, execution cost: 24237, input: 0xd0e...39138, decoded input: { "address managerAddress": "0xd9145CCE52D386f254917e481e844e9943f39138" }, decoded output: { "0": "string: " }, and logs: [ ].

## The deposit method

Now, we'll send 10 ETH from the clients' accounts to the contract by using the deposit method. In the deposit method, we take the amount declared in the msg.value from the sender that's represented in the





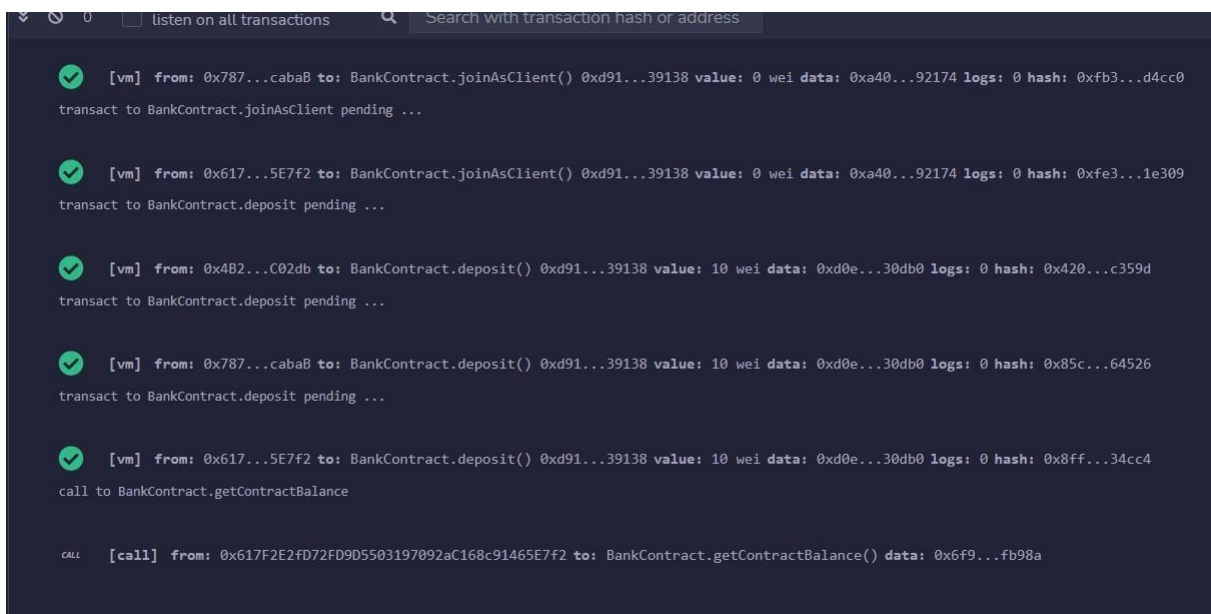
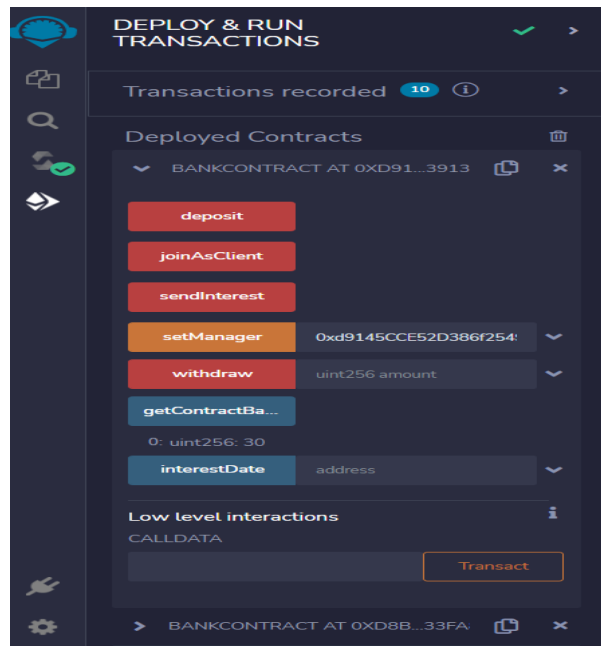
msg.sender variable. Therefore, we set 10 ETH and call the deposit method by clicking the red deposit button for each client account, like we did before for the joinAsClient method. After these operations, the following messages show in the terminal, which means those three accounts sent 10 ETH from their account to the contract address. Also, the final state of the account's balances look like this:

The screenshot displays the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel shows the 'BankContract' deployed at address 0xd91...3913. The main editor shows the Solidity code for the contract, including the 'deposit' function. The right sidebar shows the 'TRANSACTIONS' panel with a list of transactions. The first three transactions are 'deposit' calls from different addresses, each with a value of 10 ETH (1000000000000000000 wei). The fourth transaction is a 'getContractBalance' call from address 0x617f2e24d72f0905503197092ac168c91465e7f2, which returns a balance of 30 ETH (3000000000000000000 wei).

## The getContractBalance method

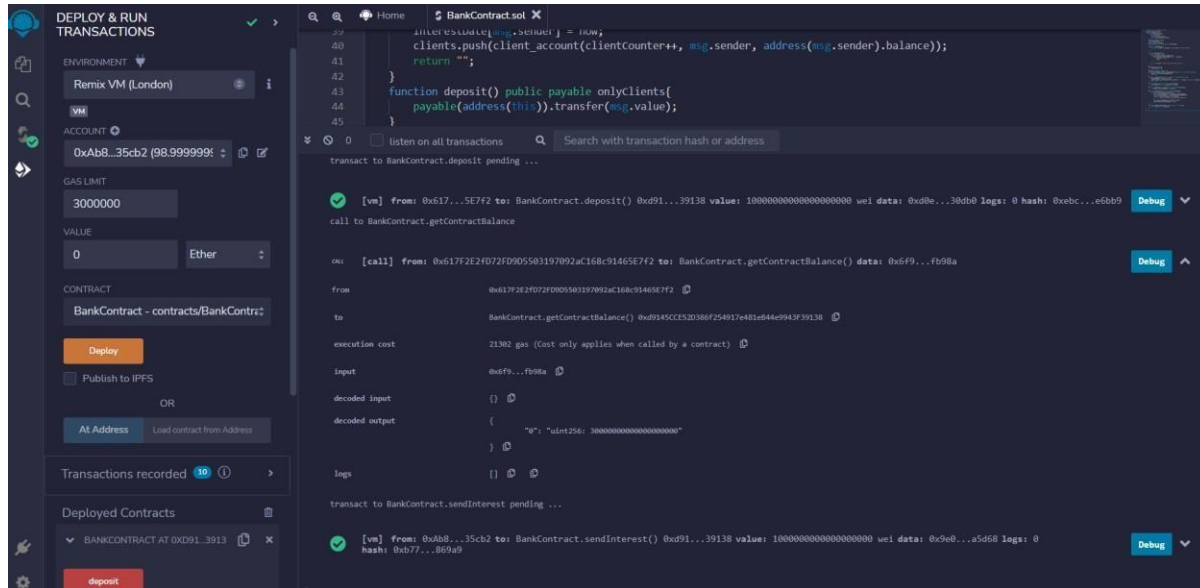
Now, we call the getContractBalance method to check whether the 30 ETH that was sent from the clients exist in the contract account. Therefore, we click the blue getContractBalance button, and it returns an amount that corresponds to 30 ETH in Wei.





## The sendInterest method

After checking that the contract isn't empty anymore, we can send interest to our clients by calling the sendInterest method. We select the account that we've set as manager account before and click the red sendInterest button. The message in the image above appears after calling the method in the terminal. This message means that 1 ETH was sent to each client's account successfully. We can see the balance of each client increased from 89 to 90 ETH after this operation.



We implemented a restriction that checks whether 10 seconds have elapsed since the `sendInterest` method was called. To check this control, we call the same method one more time in 10 seconds. The transaction went as expected, and the "It's just been less than 10 seconds!" message appeared in the terminal, as in the image below.

## The withdraw method

Now, we call the last method we've developed to withdraw an amount from the contract to the client's account. In the withdraw method, we transfer the amount declared in the `msg.value` from the account to the sender that's represented in the `msg.sender` variable. At this point, there's a problem realized, which is that the people who haven't joined the contract as a client can call this method too. We use the `onlyClient` modifier to avoid this problem. When we select an account belonging to anyone who hasn't joined the contract as a client and then call the withdraw method



# Vidyavardhini's College of Engineering and Technology, Vasai

## Department of Computer Science & Engineering (Data Science)

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is visible, showing the 'BankContract' contract and its methods: deposit, joinAsClient, sendInterest, setManager, withdraw, getContractBalance, and interestDate. The 'withdraw' method is selected, and the transaction log shows a successful transaction with the value 0 wei and data 0x2e1...0000a.

```
8 }
9 client_account[] clients;
10 int clientCounter;
11 address payable manager;
12 mapping(address => uint) public interestDate;
13
14 modifier onlyManager() {
15     require(msg.sender == manager, "Only manager can call this!");
16     _;
17 }
18
19 modifier onlyClients() {
20     bool isclient = false;
21     for(uint i=0;i<clients.length;i++){
22         if(clients[i].client_address == msg.sender){
23             isclient = true;
24             break;
25         }
26     }
27     require(isclient, "Only clients can call this!");
28     _;
29 }
30
31 constructor() public{
32     clientCounter = 0;
33 }
34 receive() external payable { }
35 function setManager(address managerAddress) public returns(string memory){
36     manager = payable(managerAddress);
37     return "";
38 }
```

Transaction log: [vm] from: 0x4B2...C02db to: BankContract.withdraw(uint256) 0xd91...39138 value: 0 wei data: 0x2e1...0000a logs: 0 hash: 0xd1f...2bad8

through the red withdraw button, the "Only clients can call this!" is displayed:

The screenshot shows the transaction log with a failed transaction. The transaction was from 0x17F...8c372 to BankContract.withdraw(uint256) 0xd91...39138 with a value of 0 wei and data 0x2e1...0000a. The transaction failed with a VM error: revert. The reason provided by the contract is "Only clients can call this!".

```
[vm] from: 0x17F...8c372 to: BankContract.withdraw(uint256) 0xd91...39138 value: 0 wei data: 0x2e1...0000a logs: 0 hash: 0x203...ddc1f
transact to BankContract.withdraw errored: VM error: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "Only clients can call this!".
Debug the transaction to get more information.
```

After calling the method, we see the ETH amount increased as much as we were expecting.

The screenshot shows the account balance for the address 0x4B2...C02db. The balance is 99.999999999999810523 ether, which is an increase from the previous state.

ACCOUNT +

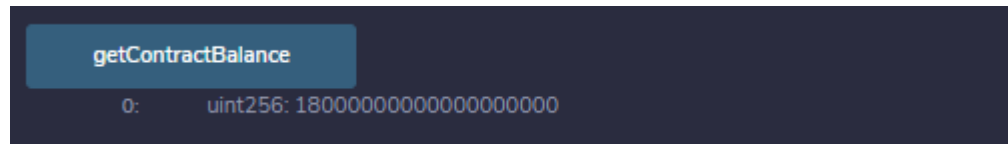
0x4B2...C02db (99.999999999999810523 ether)



---

### The balance of the sender

After these operations, only 18 ETH is supposed to be remaining in the contract address because a total of 12 ETH has been sent from the contract (3 ETH as interest and 9 ETH as a withdrawal). If we call the `getContractBalance` method, we see following result:



### Source code:

```
pragma solidity ^0.6.6;
contract BankContract {
    struct client_account{
        int client_id;
        address client_address;
        uint client_balance_in_ether;
    }

    client_account[] clients;
    int clientCounter;
    address payable manager;
    mapping(address => uint) public interestDate;
    modifier onlyManager() {
        require(msg.sender == manager, "Only manager can call this!");
        _;
    }

    modifier onlyClients() {
        bool isclient = false;
        for(uint i=0;i<clients.length;i++){
            if(clients[i].client_address == msg.sender){
                isclient = true;
                break;
            }
        }
        require(isclient, "Only clients can call this!");
        _;
    }

    constructor() public{
        clientCounter = 0;
    }
}
```



```
receive() external payable { }
```

```
function setManager(address managerAddress) public returns(string memory){  
    manager = payable(managerAddress);  
    return "";  
}
```

```
function joinAsClient() public payable returns(string memory){  
    interestDate[msg.sender] = now;  
    clients.push(client_account(clientCounter++, msg.sender, address(msg.sender).balance));  
    return "";  
}
```

```
function deposit() public payable onlyClients{  
    payable(address(this)).transfer(msg.value);  
}
```

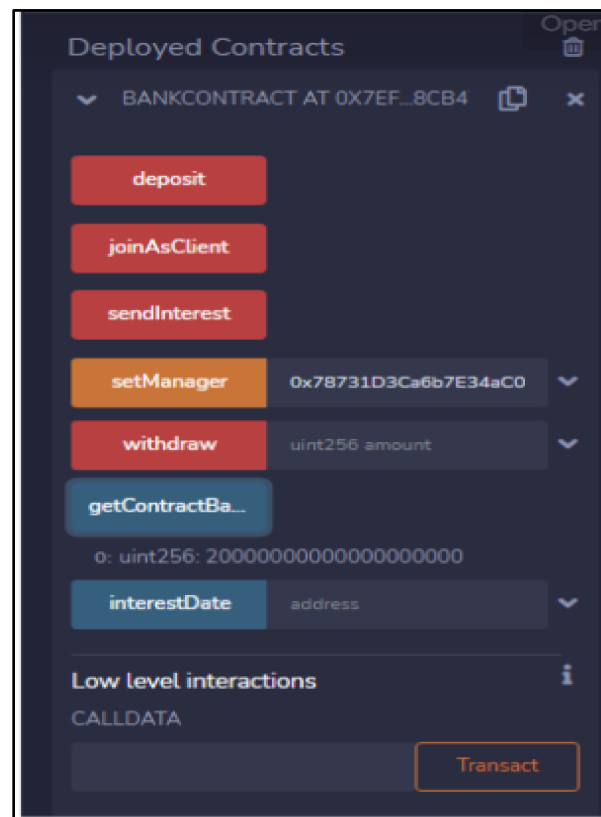
```
function withdraw(uint amount) public payable onlyClients{  
    msg.sender.transfer(amount * 1 ether);  
}
```

```
function sendInterest() public payable onlyManager{  
    for(uint i=0;i<clients.length;i++){  
        address initialAddress = clients[i].client_address;  
        uint lastInterestDate = interestDate[initialAddress];  
        if(now < lastInterestDate + 10 seconds){  
            revert("It's just been less than 10 seconds!");  
        }  
        payable(initialAddress).transfer(1 ether);  
        interestDate[initialAddress] = now;  
    }  
}
```

```
function getContractBalance() public view returns(uint){  
    return address(this).balance;  
}
```



### Initial Value





**Deposit Value:**

**DEPLOY & RUN TRANSACTIONS** ✓ >

ENVIRONMENT

Remix VM (London) ⊕ i

VM

ACCOUNT ⊕

0xAb8...35cb2 (79.999999%) ⊕ 📄 🔗

GAS LIMIT

3000000

VALUE

10

Ether ⊕

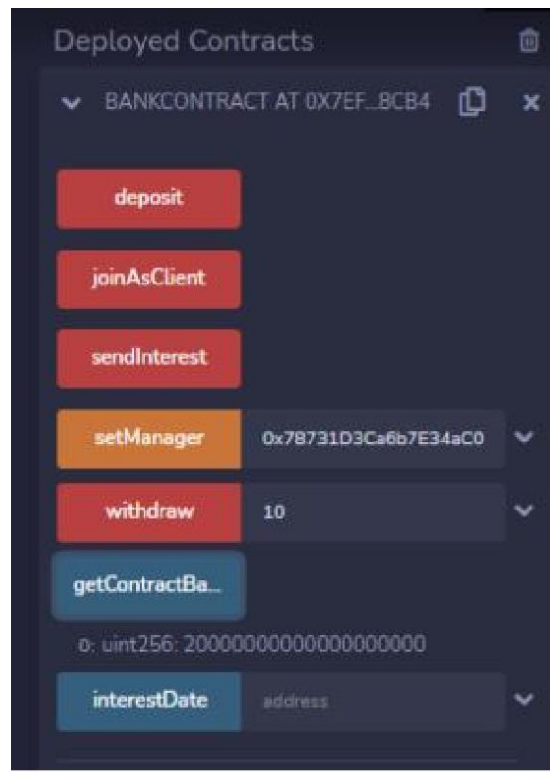
CONTRACT

BankContract - exp3dupoli.sol ⊕

Deploy

**Withdraw Amount:**





**Observations and Findings:** We have successfully understood what solidity is and how smart contracts work. We have implemented how transactions happen in a contract, along with depositing and withdrawing ethers from our account.

**Conclusion:**

**Q. How can you create and execute transactions within a Solidity smart contract using Remix IDE, and what is the role of gas in these transactions?**