Introduction to Blockchain: Technology and Applications Programming Assignment # 2

Out: March 19, 2023. Due: April 02, 2023 at 11:55 pm via LMS

Course Rules

The objective of this assignment is to test your understanding of smart contracts by developing a smart-contract based banking system. You are welcome to discuss any issues and confusions with the course staff, however **DO NOT ask other students for guidance.**

Introduction

Imagine you have recently been hired by Oracle as a back-end software developer. Considering the upsurge in the use of blockchain technology, your project manager proposes developing a blockchain based banking system and asks you to take the lead.

You remember you had studied developing smart contracts in your university course CS/EE 3812 and, thus, you confidently accept the offer. Of the many design choices, you decide to use the Ethereum blockchain to develop a *smart contract* that depicts the banking system as desired. Your project manager gives you a list of banking operations which users should be able to perform as follows:

- Opening an account (the bank account).
- Checking the details of the account.
- Depositing money to their account.
- Transferring money to other accounts.
- Withdrawing money from their account.
- Taking a loan from the bank.
- Returning the loan.

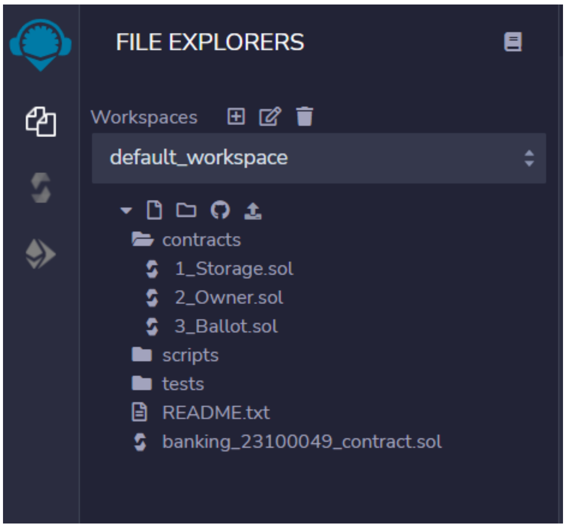
You also remember that Ethereum has its own ERC-20 standard, ether tokens, which will be regarded as the “money” in this case. To give your best shot at your first task in the company you sit at your desk and set up the *remix IDE*, all set to code your first real smart contract.

General Instructions:

You have been provided with a skeleton code file “**banking_rollnumber_contract.sol**” (available on LMS). Download and rename this file. For a student with the roll number 23100049 the file should be renamed to biddingContract_23100049.sol

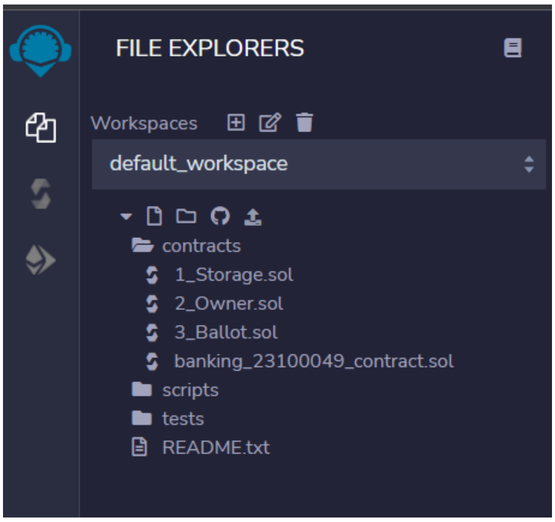
Navigate to `https://remix.ethereum.org/`. On the home page, click on open files and upload the skeleton file (the .sol file you renamed after downloading from LMS).

Your navigator would show the file as follows:



By default, remix does not upload files in the contracts directory. Move the file into the contracts directory (use right-click). This is the file in which you will be writing your code, before compiling.

In some browser versions remix **does not show** the move option when you right click on the file. **In that case**, just make another copy of the file and paste it in the contracts folder.
Once done, the navigator would look as follows:



You have also been provided with three test files named **banking_test.sol, owner_test.sol** and **exceptions_test.sol**. Download these files from LMS and open/upload them in the tests folder in remix, using the same procedure described above.

To run these tests you will need to activate the **solidity unit testing** plugin. You can click on the plugin manager icon on the **bottom left** to activate it.

Once the plugin is activated, a double-ticked testing icon will appear on the top left side of the screen. Click on that icon and select whichever tests you want to run on your code file (the skeleton file you uploaded in the contracts folder). You would need to change the file path and filename of your code file, imported within the test files.

Replace "rollnumber" in the import statement, with **your own** numeric roll number. For example, for a student with roll number 23100049, the import statement will be as follows:
"…/contracts/banking_23100049_contract.sol".

Leave **"import remix_accounts.sol"** and **"import remix_tests.sol"** statements as they are.

You are required to submit only the contract file (banking_rollnumber_contract.sol) on LMS in the designated folder before the deadline.

Remix Overview:

The remix IDE provides you 15 test accounts (important: these are depicted by user addresses and are different from a bank account). Any of these accounts can be selected (from the account drop down list in the **"Deploy & Run Transactions"** tab on the left) to deploy and test your smart contracts. By default, each test account has a balance of 100 ethers which will be displayed next to the test account address & will be updated instantly as you make transactions.

Select the **JavaScript VM** environment and any one of the 15 test accounts (called the **contract deployer**) to deploy the contract.

For testing functions (which may include making transactions), simply select the intended test account address from the drop down list. Functions that make a transaction can be provided with the amount of ether token in the **value** field within the IDE. Keep in mind that this is not the same as the traditional way of passing arguments to functions.

1 Opening An Account

In this part, you have to set up a structure that stores the account details of each user that comes to your bank. The user details would be limited to the first name, last name, loan amount and the balance of their account.

The corresponding function in the smart contract is **openAccount(string memory firstName, string memory lastName)**. The function sets up the user details in the structure and initialises the necessary variables (loan and balance amounts are initialized to zero).

Note:

- You have to keep in mind that each user is represented by a unique address on the blockchain, the contract sets up the user account by mapping the user details to their unique address.
- If the contract deployer tries to create a bank account, the transaction should be reverted with an error message, **"Error, Owner Prohibited"** (prohibiting contract deployer from opening a bank account).
- If an existing user tries to create a bank account again, the transaction should be reverted with an error message, **"Account already exists"** (prohibiting a user to have multiple banks accounts)

2 Checking Account Information

In this part you need to create a getter function to access the bank account details of the callee and return them accordingly.

The corresponding function in the smart contract is **getDetails() returns (uint balance, string memory first_name, string memory last_name, uint loanAmount).**

Note:

- If a person who does not have a bank account tries to inquire details, the transaction should be reverted with an error message **"No Account"**.
- **Most of your test cases will fail if this function is not properly implemented.**

3 Depositing Ether

In this section, the users deposit ether from their wallet to their bank account.

The function to be used for the transaction is **depositAmount() public payable**. The function deposits the **value** sent to it, from the caller's wallet to their bank account. The bank also has a minimum deposit requirement of 1 ether tokens into any bank account.
Note:

- If the deposit amount is less then the minimum deposit requirement the transaction should be reverted with an error message **"Low Deposit"**.
- Keep in mind that in our banking system, the smart contract acts as the central authority and stores the deposited amount **in itself**. This deposited amount has to be deducted from the user's wallet and be added to his/her account balance.
- If the user does not have a bank account and tries to deposit an amount, the transaction should be reverted with an error message **"No Account"**

4 Withdrawing Ether

In this part, the user calling this function withdraws the desired amount of ether tokens from the bank to his/her wallet.

The corresponding function in the smart contract is **withDraw(uint withdrawalAmount)**. Once a withdrawal is accepted, the smart-contract **(our bank)** transfers money to the user's wallet from its reserve.

Note:

- Keep in mind that, the user can only withdraw the amount less than or equal to the balance of the user in the bank. Once a withdrawal is accepted, the smart-contract **(our bank)** transfers money to the user's wallet from its reserve.
- If the contract deployer tries to withdraw any amount, the transaction should be reverted with an error message, **"Error, Owner Prohibited"**
- If the user does not have a bank account and tries to withdraw an amount, the transaction should be reverted with an error message **"No Account"**
- If a user tries to withdraw an amount more than their bank balance, the transaction should be reverted with an error message **"Insufficent Funds"**.

5 Transferring Ether

In this part, the user is provided with the functionality to transfer desired amount of ether tokens from his/her bank account to another bank account.

The corresponding function in the smart contract is **TransferEth(address payable recipient, uint transferAmount)**. The function transfers the transferAmount passed to it, from the callees' account to the recipient's account.

Note:

- If the contract deployer tries to transfer any amount, the transaction should be reverted with an error message, **"Error, Owner Prohibited"**.
- If the user does not have an account and tries to transfer an amount, the transaction should be reverted with an error message **"No Account"**.
- Keep in mind that, the user can only transfer an amount less than or equal to the balance of the user in the bank.
- If the user tries to transfer funds more than his balance. The transaction should be reverted with error message, **"Insufficient Funds"**.

6 Loan Management

In this section you will setup a system which gives out loans to the users from the bank's reserve. You will also need to store a record of loans given to each user.

You'll need to setup the following functions for this section:

- **depositTopUp() public payable**: This function is to be used by the contract deployer to deposit ether tokens in the bank's reserve for giving out loans. In our system, **only the funds deposited by this function can be used to give out loans**. If any other user tries to call this function, the transaction should be reverted with an error message, **"Only Owner can call this function"**.

- **TakeLoan(uint loanAmount)**: This function takes the loan amount as an argument & performs the following tasks:
  1. If the contract deployer tries to take a loan, the transaction is reverted with an error message, **"Error, Owner Prohibited"**.
  2. If a user does not have an account & requests a loan, the transaction is to be reverted with an error message, **"No Account"**.
  3. If the loan amount is greater than the funds available for loans, the transaction is reverted with an error message, **"Insufficient Loan Funds"**.
  4. If the requested loan amount is more than twice the user's bank balance, the transaction is reverted with an error message, **"Loan Limit Exceeded"**.
- **InquireLoan() returns (uint loanValue)**: This function should return the amount of loan the caller owes. If a user with no account calls the function, the transaction should be reverted with an error message, **"No Account"**.

Example scenario: Consider the case that Tom deploys the banking smart contract. Tom does not call the **depositTopUp()** function. Afterwards, uncle Bob makes a deposit of 2 Ether using the **depositAmount()**. These 2 Ethers are present in the bank. Afterwards, if uncle Bob tries to use **TakeLoan(1.0)** then it should revert with the error **"Insufficient Loan Funds"**. The key-point is: *If the owner does not deposit Ether using the **depositTopUp()** function, any other Ether in the bank **can not** be used to give out loans.*

7 Returning a Loan

In this part the user returns the loan amount to the bank.

The function used for this task is **returnLoan() public payable**. This function returns the value sent to it to the bank. The tokens will be taken from the wallet of the user and added to the banks' reserve.

Note:

- If a user with no bank account calls the function, the transaction should be reverted with an error message, **"No Account"**.
- If a user does not owe any amount & still calls the function, the transaction should be reverted with an error, **"No Loan"**.
- If the user mistakenly enters an amount more than what he owed, the transaction should be reverted with an error, **"Owed Amount Exceeded"**.

Things to keep in Mind:

- You may use different data types/constants/variables to keep track of how much ether is stored for what purpose, or from whom etc., keep in mind that, in the smart contract, **all the ether will be collectively stored**. The variables are just placeholders and numbers to help ease the way we code.
- You are allowed to create as many helper functions as you need.
- *You are not allowed to rename/change any function definitions or return variables.*
- You have to submit only **banking_2XXXXXXX_contract.sol** on lms in the relevant assignment tab. *No email submissions will be accepted.*

Good Luck!!