Course #1 - Exercises

1 - Create your (semi) decentralized credit organism

Traditionally, banks and credit organizations have always been central actors of finance for people. They are in charge of managing your bank accounts, granting you loans, or transferring money from your account to another. This system works, but you have to trust the bank to act in a good way. In this first exercise, we are going to implement an example of a (semi) decentralized credit organism.

Why "semi"? Because as the owner, you'll be in charge of the smart-contract and by extension, the bank. The system is decentralized by itself and fully transparent, but you still centralize the management of the bank. Many systems have been developed to propose autonomous financial services on-chain (this is called DeFi), but far too complex for this course. Let's keep it to the basics!

We will build a smart-contract that lends Ether to individuals, named LenderContract. People are allowed to request loans from the contract, and you're in charge of accepting or rejecting loan requests. For each loan, you'll take a lending fee of a variable amount, defined in the smart-contract. You can modify it at any time, but it doesn't affect already subscribed loans and it is visible publicly, thus totally transparent.

1) Create your first contract in Remix. Go into file explorer, and create a new file called *lenderContract.sol*. In this file, create your contract Lender, and don't forget to indicate to the compiler what version you're using! (*pragma solidity >=0.7.0 <0.8.0;*)

2) You have a contract, great! Now you need to fund your contract with your own money, to lend it. Fortunately, all accounts used in Remix are credited with 100 (fake) ethers. We are going to use this token as our currency. Create a function, only callable by you (the bank manager), to deposit money from your wallet into the smart-contract. Implement a public getter function that returns the current balance of the contract.

3) We are ready to implement the loan feature. First, we need to specify in the contract what a loan is. Create a structure of a *Loan*, that contains an ID, the public address of the borrower, the amount borrowed, the left-to-be-paid amount (includes the fee), the loan rate, and the loan state (4 possible status: requested, granted, denied, paid back). You can use an enum to list all those status.

4) We need to find a way to store loans whenever they are created using the new *Loan* type. Create a mapping named *idToLoan* between loan IDs (*int*) and a Loan. For practical reasons, let's also create two more variables: the first one, *clientToLoansIds* will map a client's public address to an array that contains IDs of his loans (repaid, or

not), plus another array *requestIds* that stores IDs of all loans that have the *requested* state.
Why? Because it will be far more efficient to use those two variables than just iterate into the loans mapping to find the one we're looking for.

5) You're not lending for free: create a contract state variable named rate, that defines the fee you'll take for lending. (eg. a rate of 1% means that the borrower will have to repay you 100% of the amount borrowed + 1% of the amount borrowed as a fee). When a loan will be created, it will use this variable as the current loan rate.

6) Implement a function to allow someone to request lending to the contract. The function will create a loan with the status *requested* and other relevant information, and store it into the *idToLoan*. Its ID will be stored into the *requestIds* array (created in Q4), and also be bonded to the client into the *clientToLoansIds*.

7) As the contract manager, you're in charge of granting loans or not. Create a function that returns all requested loans, using the *requestIds* array. Create another function to accept or deny a specific loan, by providing the loan ID and the decision (accepted/rejected) as parameters. Change the Loan variable status accordingly, and remove the loan ID from the *requestIds* (as the request is closed). If the request has been accepted, transfer Ether from the contract to the wallet of the client.
Be careful, you have to check many conditions to perform this operation: the person using this function must be the contract owner, the bank must have sufficient balance, ...

8) Let borrowers repay their loan, by implementing a function that takes in parameter a loan ID and expects the user to attach Ether when executing this function through a smart-contract transaction. If the loan is fully repaid after the transaction, change its status accordingly.
Don't forget to handle the case where the user gives a wrong loan ID, or refund more than expected.

9) (Bonus) The contract should work as intended, but you can only borrow Ether from an Ethereum wallet. What if your bank wants to borrow money from another? Try to implement such a system, where your contract can instantiate another bank contract on-chain using its address, and requests funds with a method.