// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract MultiLoanContract {

// State variables

address public lender;

uint256 private borrowerCount;

address[] private activeBorrowers;

address[] private requestedBorrowers;

struct Loan {

uint256 loanAmount;

uint256 interestRate; // Interest rate in percentage

uint256 totalRepayment;

uint256 repaymentDueDate;

bool isLoanFunded;

bool isLoanRepaid;

}

mapping(address => Loan) public loans;

// Events

event LoanFunded(address indexed lender, uint256 amount);

event LoanRequested(

address indexed borrower,

uint256 amount,

uint256 interestRate

);

event LoanRepaid(address indexed borrower, uint256 amount);

event ContractFunded(address indexed lender, uint256 amount);

event BorrowerRemoved(address indexed borrower);

event LoanRequestRejected(address indexed borrower);

// Modifiers

modifier onlyLender() {

require(msg.sender == lender, "Only lender can call this function");

\_;

}

modifier onlyBorrower() {

require(

msg.sender != lender,

"Only the borrower can call this function"

);

\_;

}

// Constructor

constructor() {

lender = msg.sender; // Deployer is the lender

borrowerCount = 0;

}

// Function: Change lender

function changeLender(address \_address) external onlyLender {

lender = \_address;

}

// Function: Request a loan

function requestLoan(

uint256 \_loanAmount,

uint256 \_interestRate,

uint256 \_repaymentDuration

) external {

require(msg.sender != lender, "Lender cannot request a loan");

require(

loans[msg.sender].loanAmount == 0,

"Loan already exists for this borrower"

);

uint256 totalRepayment = \_loanAmount +

(\_loanAmount \* \_interestRate) /

100;

loans[msg.sender] = Loan({

loanAmount: \_loanAmount,

interestRate: \_interestRate,

totalRepayment: totalRepayment,

repaymentDueDate: block.timestamp + \_repaymentDuration,

isLoanFunded: false,

isLoanRepaid: false

});

requestedBorrowers.push(msg.sender);

emit LoanRequested(msg.sender, \_loanAmount, \_interestRate);

}

// Function: Fund the contract

function fundContract() external payable onlyLender {

emit ContractFunded(lender, msg.value);

}

// Function: Approve and fund the loan for a borrower

function approveLoan(address \_borrower) external onlyLender {

Loan storage loan = loans[\_borrower];

require(loan.loanAmount > 0, "No loan requested by this borrower");

require(!loan.isLoanFunded, "Loan is already funded");

require(

address(this).balance >= loan.loanAmount,

"Contract does not have sufficient funds"

);

loan.isLoanFunded = true;

borrowerCount++;

payable(\_borrower).transfer(loan.loanAmount);

// Move borrower from requested to active list

for (uint256 i = 0; i < requestedBorrowers.length; i++) {

if (requestedBorrowers[i] == \_borrower) {

requestedBorrowers[i] = requestedBorrowers[

requestedBorrowers.length - 1

];

requestedBorrowers.pop();

break;

}

}

activeBorrowers.push(\_borrower);

emit LoanFunded(\_borrower, loan.loanAmount);

}

// Function: Repay the loan

function repayLoan() external payable onlyBorrower() {

Loan storage loan = loans[msg.sender];

require(loan.isLoanFunded, "Loan is not yet funded");

require(!loan.isLoanRepaid, "Loan is already repaid");

require(

block.timestamp <= loan.repaymentDueDate,

"Repayment is overdue"

);

require(msg.value == loan.totalRepayment, "Incorrect repayment amount");

uint256 repaymentAmount = loan.totalRepayment;

loan.isLoanRepaid = true;

// Remove borrower from the active borrowers list

for (uint256 i = 0; i < activeBorrowers.length; i++) {

if (activeBorrowers[i] == msg.sender) {

activeBorrowers[i] = activeBorrowers[

activeBorrowers.length - 1

];

activeBorrowers.pop();

borrowerCount--;

emit BorrowerRemoved(msg.sender);

break;

}

}

delete loans[msg.sender]; // Clear loan details before transferring

// Transfer repayment amount to the lender

payable(lender).transfer(repaymentAmount);

emit LoanRepaid(msg.sender, repaymentAmount);

}

// Function: Check if repayment is overdue for a borrower

function isRepaymentOverdue(address \_borrower) public view returns (bool) {

Loan storage loan = loans[\_borrower];

return block.timestamp > loan.repaymentDueDate && !loan.isLoanRepaid;

}

// Function: Take out all contract funds

function takeOutContractFunds() public onlyLender {

require(address(this).balance > 0, "No funds are available");

payable(lender).transfer(address(this).balance);

}

// Function: Check the contract balance

function checkBalanceOfSmartContract()

public

view

onlyLender

returns (uint256)

{

return address(this).balance;

}

// Function: Get the count of all borrowers

function getBorrowerCount() public view returns (uint256) {

return borrowerCount;

}

// Function: Get the list of active borrowers

function getActiveBorrowers()

public

view

onlyLender

returns (address[] memory)

{

return activeBorrowers;

}

// Function: Get the list of requested borrowers

function getRequestedBorrowers()

public

view

onlyLender

returns (address[] memory)

{

return requestedBorrowers;

}

// Function: Check the balance of a specific borrower

function getBorrowerBalance(address \_borrower)

public

view

returns (uint256)

{

return \_borrower.balance;

}

// Function: Check the balance of the lender

function getLenderBalance() public view onlyLender returns (uint256) {

return lender.balance;

}

// Function to reject the loan request

function rejectLoanReq(address \_borrower) external onlyLender {

Loan storage loan = loans[\_borrower];

require(loan.loanAmount > 0, "No loan requested by this borrower");

require(!loan.isLoanFunded, "Loan is already funded");

// Remove borrower from requested borrowers list

for (uint256 i = 0; i < requestedBorrowers.length; i++) {

if (requestedBorrowers[i] == \_borrower) {

requestedBorrowers[i] = requestedBorrowers[requestedBorrowers.length - 1];

requestedBorrowers.pop();

break;

}

}

delete loans[\_borrower]; // Remove loan data for the borrower

emit LoanRequestRejected(\_borrower);

}

}