/\*

Joint Savings Account

---------------------

To automate the creation of joint savings accounts, you will create a solidity smart contract that accepts two user addresses that are then able to control a joint savings account. Your smart contract will use ether management functions to implement various requirements from the financial institution to provide the features of the joint savings account.

The Starting file provided for this challenge contains a `pragma` for solidity version `5.0.0`.

You will do the following:

1. Create and work within a local blockchain development environment using the JavaScript VM provided by the Remix IDE.

2. Script and deploy a \*\*JointSavings\*\* smart contract.

3. Interact with your deployed smart contract to transfer and withdraw funds.

\*/

pragma solidity ^0.5.0;

// Define a new contract named `JointSavings`

contract JointSavings {

/\*

Inside the new contract define the following variables:

- Two variables of type `address payable` named `accountOne` and `accountTwo`

- A variable of type `address public` named `lastToWithdraw`

- Two variables of type `uint public` named `lastWithdrawAmount` and `contractBalance`.

\*/

address payable accountOne;

address payable accountTwo;

address public lastToWithdraw;

uint public lastWithdrawAmount;

uint public contractBalance;

/\*

Define a function named \*\*withdraw\*\* that will accept two arguments.

- A `uint` variable named `amount`

- A `payable address` named `recipient`

\*/

function withdraw(uint amount, address payable recipient) public {

/\*

Define a `require` statement that checks if the `recipient` is equal to either `accountOne` or `accountTwo`. The `requiere` statement returns the text `"You don't own this account!"` if it does not.

\*/

require(recipient==accountOne || recipient==accountTwo, "You don't own this account!");

/\*

Define a `require` statement that checks if the `balance` is sufficient to accomplish the withdraw operation. If there are insufficient funds, the text `Insufficient funds!` is returned.

\*/

require(amount<=contractBalance, "Insufficient funds!");

/\*

Add and `if` statement to check if the `lastToWithdraw` is not equal to (`!=`) to `recipient` If `lastToWithdraw` is not equal, then set it to the current value of `recipient`.

\*/

if (lastToWithdraw!=recipient) {

lastToWithdraw=recipient;

}

// Call the `transfer` function of the `recipient` and pass it the `amount` to transfer as an argument.

recipient.transfer(amount);

// Set `lastWithdrawAmount` equal to `amount`

lastWithdrawAmount=amount;

// Call the `contractBalance` variable and set it equal to the balance of the contract by using `address(this).balance` to reflect the new balance of the contract.

contractBalance=address(this).balance;

}

// Define a `public payable` function named `deposit`.

function deposit() public payable {

/\*

Call the `contractBalance` variable and set it equal to the balance of the contract by using `address(this).balance`.

\*/

contractBalance=address(this).balance;

}

/\*

Define a `public` function named `setAccounts` that receive two `address payable` arguments named `account1` and `account2`.

\*/

function setAccounts(address payable account1, address payable account2) public{

// Set the values of `accountOne` and `accountTwo` to `account1` and `account2` respectively.

accountOne=account1;

accountTwo=account2;

}

/\*

Finally, add the \*\*default fallback function\*\* so that your contract can store Ether sent from outside the deposit function.

\*/

function() external payable{}

}