PROGRAM THE BLOCKCHAIN

Archive About Subscribe

Writing a Banking Contract

JANUARY 5, 2018 BY TODD PROEBSTING

This article will demonstrate how to write a simple, but complete, smart contract in Solidity that acts like a bank that stores ether on behalf of its clients. The contract will allow deposits from any account, and can be trusted to allow withdrawals only by accounts that have sufficient funds to cover the requested withdrawal.

This post assumes that you are comfortable with the ether-handling concepts introduced in our post, [Writing a Contract That Handles Ether].

That post demonstrated how to restrict ether withdrawals to an "owner's" account. It did this by persistently storing the owner account's address, and then comparing it to the `msg.sender` value for any withdrawal attempt. Here's a slightly simplified version of that smart contract, which allows anybody to `deposit` money, but only allows the `owner` to make withdrawals:

```
pragma solidity ^0.4.19;

contract TipJar {

    address owner;    // current owner of the contract

    function TipJar() public {
        owner = msg.sender;
    }

    function withdraw() public {
        require(owner == msg.sender);
        msg.sender.transfer(address(this).balance);
    }
    function deposit(uint256 amount) public payable {
        require(msg.value == amount);
    }

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Maintaining Individual Account Balances

I am going to generalize this contract to keep track of ether deposits based on the account address of the depositor, and then only allow that same account to make withdrawals of that ether. To do this, we need a way keep track of account balances for each depositing account—a mapping from accounts to balances. Fortunately, Solidity provides a ready-made `mapping` data type that can map account addresses to integers, which will make this bookkeeping job quite simple. (This mapping structure is much more general key/value mapping than just addresses to integers, but that's all we need here.)

Here's the code to accept deposits and track account balances:

```
pragma solidity ^0.4.19;

contract Bank {

    mapping(address => uint256) public balanceOf;   // balances, indexed by addresses

    function deposit(uint256 amount) public payable {
        require(msg.value == amount);

        balanceOf[msg.sender] += amount;     // adjust the account's balance
    }
}
```

Here are the new concepts in the code above:

- `mapping(address => uint256) public balanceOf;` declares a persistent public variable, `balanceOf`, that is a mapping from account addresses to 256-bit unsigned integers. Those integers will represent the current balance of ether stored by the contract on behalf of the corresponding address.

- Mappings can be indexed just like arrays/lists/dictionaries/tables in most modern programming languages.
- The value of a missing mapping value is 0. Therefore, we can trust that the beginning balance for all account addresses will effectively be zero prior to the first deposit.

It's important to note that `balanceOf` keeps track of the ether balances assigned to each account, but it does not actually move any ether anywhere. The bank contract's ether balance is the sum of all the balances of all accounts—only `balanceOf` tracks how much of that is assigned to each account.

Note also that this contract doesn't need a constructor. There is no persistent state to initialize other than the `balanceOf` mapping, which already provides default values of 0.

Withdrawals and Account Balances

Given the `balanceOf` mapping from account addresses to ether amounts, the remaining code for a fully-functional bank contract is pretty small. I'll simply add a withdrawal function:

bank.sol

```
pragma solidity ^0.4.19;

contract Bank {

    mapping(address => uint256) public balanceOf;   // balances, indexed by addresses

    function deposit(uint256 amount) public payable {
        require(msg.value == amount);
        balanceOf[msg.sender] += amount;     // adjust the account's balance
    }

    function withdraw(uint256 amount) public {
        require(amount <= balanceOf[msg.sender]);
        balanceOf[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```