Checking the Sender in a Smart Contract

DECEMBER 26, 2017 BY TODD PROEBSTING

This article will demonstrate how to write a simple, but complete, smart contract in Solidity that accepts and distributes ether on behalf of the contract's owner. It assumes that you are comfortable with the ether-handling concepts introduced in our last Solidity example blog post.

Establishing a Smart Contract "Owner"

Let's start with the code from last time:

```
pragma solidity ^0.4.19;

contract CommunityChest {
    function withdraw() public {
        msg.sender.transfer(address(this).balance);
    }

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

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

The obvious (and unrealistic!) shortcoming of this contract is that anybody can withdraw ether from the contract, so it is of no real use. The contract needs the notion of an owner account that will be the only account that can withdraw ether.

Fortunately, every account has a unique address, and it is possible to store addresses persistently and to test for address equality. Here's the changed code to do that, renamed "TipJar" because anybody can deposit, but only the owner can withdraw:

```
pragma solidity ^0.4.19;

contract TipJar {

    address owner;    // current owner of the contract

    function TipJar() public {  // contract's constructor function
        owner = msg.sender;
    }

    function withdraw() public {
        require(owner == msg.sender);
        msg.sender.transfer(address(this).balance);
    }

    // unchanged code omitted
}
```

Here's a quick explanation of the code above:

- The TipJar contract has a global, persistent value, `owner`, that stores the current owner.
- The TipJar contract's constructor sets the initial value of `owner` to `msg.sender`, which in the case of a constructor is the address of the account that is deploying the contract.
- The `withdraw()` function now requires that the account requesting the withdraw transaction (`msg.sender`) be the current owner.

The key things to note above are that address of a transaction's sender's account is available to the smart contract (as `msg.sender`), and that addresses can be stored and compared just like other scalar values.

Changing the Owner

An important pattern to enable in smart contracts is making them transferable. In the case of the TipJar example, that would mean allowing the current owner to transfer ownership to another account:

```
pragma solidity ^0.4.19;

contract TipJar {

    function changeOwner(address newOwner) public {
        require(owner == msg.sender);
        owner = newOwner;
    }

    // unchanged code omitted
}
```

The code above repeats two patterns from the previous code samples:

- The `require` statement aborts the transaction if any account other than the current owner attempts to change the owner.
- The persistent state variable `owner` is updated with the value held in the `newOwner` parameter.

Creating New Modifiers

Solidity has a powerful mechanism for creating custom function "modifiers". Up until now, we've only used the built-in modifiers `public` and `view`. Modifiers allow programmers to summarize boilerplate prologue (and epilogue) code that may appear in many functions. In the example above, the `require(owner == msg.sender)` code is a simple example of such prologue code. The code below will eliminate that by creating a new `ownerOnly` modifier:

tipjar.sol

```
pragma solidity ^0.4.19;

contract TipJar {
    address owner;

    modifier ownerOnly {
        require(owner == msg.sender);
        _;   // <--- note the '_', which represents the modified function's body
    }

    function TipJar() public {  // contract's constructor function
        owner = msg.sender;
    }

    function changeOwner(address newOwner) public ownerOnly {
        owner = newOwner;
    }

    function withdraw() public ownerOnly {
        msg.sender.transfer(address(this).balance);
    }

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

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

The code above encompasses the following changes:

- The `ownerOnly` modifier is created, which includes the `require` statement for checking if the `owner` variable is equal to the transaction's sender.
- The `ownerOnly` modifier's definition includes the _ (underscore) that signifies where the body of the modified function should be substituted.
- The `changeOwner` and `withdraw` functions both now use the `ownerOnly` modifier in the function header rather than including the `require` statement in their body. The two alternatives are equivalent.

Modifier declarations can also include boilerplate code after the underscore, which would then become the epilogue of any functions so decorated.

Multiple modifiers may be placed on a given function declaration, and all of their boilerplate code will be applied to the function's code.

Summary

- Account addresses are type `address` in Solidity.
- The address of the account sending a transaction is accessible to the contract as `msg.sender`.
- New function modifiers can be defined, and they can be composed.

← Testing and Deploying Smart Contracts with Remix How Ethereum Transactions Work →