



Basics VIII: Access Restriction

Access restriction is a common practice in smart contracts.

The access restriction design pattern restricts access to the functions of the contract based on roles, time and cost. Modifiers are the essential applicable tools here.

Let's examine the concept through a sample code. *Important lines are explained with a few lines of comments above them.*

```
contract AccessRestriction {

    /*
    The state variables owner and lastOwnerChange are
    populated with the contract creator and the current
    timestamp at contract creation time.
    */
    address public owner = msg.sender;
    uint public lastOwnerChange = now;

    /*The first modifier onlyBy(address _account) is
    attached to the changeOwner(...) function*/
    modifier onlyBy(address _account) {
        require(msg.sender == _account);
        _;
    }

    /*
    The second modifier, onlyAfter(uint _time), is
    similar to the first, except that it throws an exception
    if the function to which it is connected is called
    before the specified time.
    */
    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    /*
    Before going into the execution of the guarded function,
    the third and final modifier costs(uint _amount) accepts
    an amount of money as input and ensures that the value given
    with the calling transaction is at least as high as the stated amount.
    */
    modifier costs(uint _amount) {
        require(msg.value >= _amount);
        _;
    }

    /*
    Extra if-clause checks if more money was supplied in the
    transaction than was required and returns the excess to the sender.
    This is an excellent illustration of the different options that
```

```

        modifiers may give.
    */
    if (msg.value > _amount) {
        msg.sender.transfer(msg.value - _amount);
    }
}


/*
We ensure that the function's initiator (msg.sender) is
equivalent to the variable supplied in the modifier call, which
in this instance is owner. When this modifier is used, an exception
is thrown if the guarded function is called by someone other than
the current owner.
*/
function changeOwner(address _newOwner) public onlyBy(owner) {
    owner = _newOwner;
}

/*
It is given with the period of the last change of ownership
plus four weeks and is utilized. (Four weeks are added instead
of one month because Solidity does not support months as a unit
of time.). As a result, the function call can only be successful
if it has been at least four weeks since the last update.
*/
function buyContract() public payable onlyAfter(lastOwnerChange + 4 weeks) costs(1 ether) {
    owner = msg.sender;
    lastOwnerChange = now;
}
/*
To receive money with a transaction, the payable modification of the buyContract()
method is required.
*/
}

```

Reference:

Common Patterns - Solidity 0.4.21 documentation

 <https://docs.soliditylang.org/en/v0.4.21/common-patterns.html#restricting-access>

Bonus:

How to Test Ethereum Smart Contracts for Access Restriction

Prerequisite: This article assumes an understanding of Solidity and Ethereum smart contracts. Controlling access to smart contracts is vital to ensuring security. Common patterns, like OpenZeppelin's <https://betterprogramming.pub/how-to-test-ethereum-smart-contracts-for-access-restriction-9dff445400d0>

