

## ERROR AND EXCEPTIONS

Nội dung bài học sẽ tập trung giải quyết các mục tiêu: Phân biệt được Throw, Require, Assert và Revert. Hiểu được, giải thích được và cho các ví dụ minh họa các kiểu Error & Exceptions trên. Hiểu được, giải thích được các dạng Functions. Hiểu được, giải thích được Inheritance and Modifier functions.

Kết thúc nội dung này, sinh viên trả lời được các câu hỏi sau:

Câu hỏi 1: Phân biệt Throw, Require, Assert và Revert?

Câu hỏi 2: Các loại Functions và thuộc tính, cho ví dụ và giải thích?

Câu hỏi 3: Giải thích và cho ví dụ về Inheritance and Modifier?

### 1. Phân biệt Throw, Require, Assert và Revert

#### 1.1. Throw :

Throw là phần phụ thuộc trả về của Require, Assert và Revert -> Nó giúp cho việc báo lỗi và phục hồi nguyên trạng các tác vụ đã diễn ra trong cuộc bầu cử.

#### 1.2. Require :

Hàm require dùng để đảm bảo tính hợp lệ của các điều kiện không thể được phát hiện trước khi thực thi. Nó kiểm tra inputs, contract state và return values từ việc call các external contract. Hàm này sẽ tạo ra lỗi kiểu string.

#### 1.3. Assert :

Hàm assert sẽ tạo ra lỗi kiểu uint256.

#### 1.4. Revert :

Hàm revert là một cách khác để kích hoạt các exception từ bên trong các code khác để gắn cờ lỗi và revert current call.

### 2. Require :

```

1  pragma solidity ^0.6.4;
2
3  contract Hello {
4      function sendHalf() public payable returns (uint balance) {
5          require(msg.value % 2 == 0, "Even value required.");
6          return address(this).balance;
7      }
8  }

```

Nếu hàm require thực hiện thành công, tiếp tục.. Ngược lại, trả về error, các trường hợp sử dụng Require bao gồm :

Khi bạn gọi require với một argument được đánh giá là false.

Khi bạn thực hiện một lệnh gọi đến một external function mà nó không chứa code.

Khi create contract với new keyword nhưng không kết thúc đúng cách

Khi contract của bạn nhận Ether trong hàm public không có payable bao gồm cả fallback function

Khi contract của bạn nhận Ether trong hàm public getter

Khi .transfer() lỗi

3. Assert :

```

1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract Hello {
4      function sendHalf(address payable addr) public payable returns (uint balance) {
5          require(msg.value % 2 == 0, "Even value required.");
6          uint balanceBeforeTransfer = address(this).balance;
7          addr.transfer(msg.value / 2);
8          assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
9          return address(this).balance;
10     }
11 }

```

Tại đây assert lưu trữ thông tin bên trong nhưng với phép chia yêu cầu ta phải dùng assert thay vì require, các trường hợp sử dụng Assert bao gồm :

Nếu bạn gọi assert với một argument được đánh giá là false.

Khi bạn gọi một biến chưa được khởi tạo

Khi bạn convert một giá trị lớn hoặc âm thành kiểu enum

Khi chia cho 0

Khi truy cập vào array vs index âm hoặc quá lớn

4. Revert :

```
1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract Hello {
4      function buy(uint amount) public payable {
5          if (amount > msg.value / 2 ether)
6              revert("Not enough Ether provided.");
7          require(
8              amount <= msg.value / 2 ether,
9              "Not enough Ether provided."
10         );
11     }
12 }
```

Vì ta kiểm tra điều kiện trước đó nên phải dùng revert

## 2. Functions

### 2.1. Setter và Getter

- Trong solidity, hàm getter trả về một giá trị.

- Trong Solidity, setter là một hàm điều chỉnh giá trị của một biến (sửa đổi trạng thái của hợp đồng). Để tạo một setter, bạn phải chỉ định các tham số khi bạn khai báo hàm của mình.

Ta có ví dụ về get, set như hình dưới đây:

```
contract Score {
    uint score = 5;

    function getScore() public returns (uint) {
        return score;
    }

    function setScore(uint new_score) public {
        score = new_score;
    }
}
```

Như ví dụ trên, ta define một biến trạng thái score với data type là uint với default value là 5. Sau đó viết hai hàm hiển thị, hoặc set để thay đổi điểm mới. Tức là khi giá trị mặc định là 5, và khi ta gọi hàm set thì có thể thay đổi giá trị 5 đó.

## 2.2. Constructor

Một phương thức khởi tạo là một phương thức đặc biệt trong bất kỳ ngôn ngữ lập trình hướng đối tượng nào được gọi bất cứ khi nào một đối tượng của một lớp được khởi tạo. Nhưng, nó hoàn toàn khác trong trường hợp Solidity, Solidity cung cấp một khai báo phương thức khởi tạo bên trong hợp đồng thông minh và nó chỉ gọi một lần khi hợp đồng được triển khai và được sử dụng để khởi tạo trạng thái hợp đồng. Một phương thức khởi tạo mặc định được tạo bởi trình biên dịch nếu không có phương thức khởi tạo nào được xác định rõ ràng.

Ta có ví dụ về một constructor sau đây:

```
contract constructorExample {  
    string str;  
    constructor() public {  
        str = "Hello, this discussion composed by Hiếu, hi!";  
    }  
  
    function getValue() public view returns (string memory) {  
        return str;  
    }  
}
```

Như code trên, ta đầu ta sẽ define một constructor với string, sau đó sẽ viết hàm get để trả về giá trị được định nghĩa trong constructor.

## 2.3. Fallback

Fallback là một chức năng đặc biệt có sẵn cho một hợp đồng. Nó có các tính năng sau:

Nó được gọi khi một hàm không tồn tại được gọi trên hợp đồng.

Nó được yêu cầu phải được đánh dấu bên ngoài.

Nó không có tên.

Nó không có đối số

Nó không thể trả lại bất kỳ điều gì.

Nó có thể được xác định một trong mỗi hợp đồng.

Nếu không được đánh dấu phải trả, nó sẽ ném ra ngoại lệ nếu hợp đồng nhận được ether thuần túy mà không có dữ liệu.

Ví dụ sau đây cho thấy khái niệm về fallback cho mỗi hợp đồng.

```
contract DelegateProxy {
    address internal implementation;

    fallback() external payable {
        address addr = implementation;

        assembly {
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), addr, 0, calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

## 2.4. View/ Pure function

Các hàm **view** là hàm chỉ đọc, đảm bảo rằng các biến trạng thái không thể được sửa đổi sau khi gọi chúng. Nếu các câu lệnh sửa đổi các biến trạng thái, tạo ra các sự kiện, tạo các hợp đồng khác, sử dụng phương pháp tự cấu trúc, chuyển các ete qua các cuộc gọi, Gọi một hàm không phải là 'view hoặc pure', sử dụng các lệnh cấp thấp, v.v. có trong các hàm view thì trình biên dịch đưa ra một cảnh báo trong những trường hợp như vậy. Theo mặc định, một phương thức get là hàm view. .

Ta có ví dụ sau về hàm view:

```
contract Test {
    uint num1 = 2;
    uint num2 = 4;

    function getResult() public view returns(
        uint product, uint sum){
        uint num1 = 10;
        uint num2 = 16;
        product = num1 * num2;
        sum = num1 + num2;
    }
}
```

Như contract trên, ban đầu ta định nghĩa hai biến trạng thái, sau viết define hàm view để tính tổng hai số và tính tổng sản phẩm.

Các hàm **pure** không đọc hoặc sửa đổi các biến trạng thái, các biến này chỉ trả về các giá trị bằng cách sử dụng các tham số được truyền cho hàm hoặc các biến cục bộ có trong nó. Nếu các câu lệnh đọc các biến trạng thái, truy cập địa chỉ hoặc số dư, truy cập bất kỳ khối biến toàn cục hoặc thư, gọi một hàm không thuần túy, v.v.

có trong các hàm pure thì trình biên dịch sẽ đưa ra cảnh báo trong các trường hợp như vậy.

Ta có ví dụ sau đây:

```
contract Test {  
    function getResult() public pure returns(uint product, uint sum) {  
        uint num1 = 2;  
        uint num2 = 4;  
        product = num1 * num2;  
        sum = num1 + num2;  
    }  
}
```

Như ví trên, ta có thể thấy cái biến trạng thái sẽ được định nghĩa ngay bên trong function, chứ không để ngoài như hàm view. Và nó cũng sẽ lý giải được được điểm khác nhau giữa hai hàm này.

## 2.5. Public/Private/External/Internal Functions

### 2.5.1. External

External function là một phần của giao diện hợp đồng, có nghĩa là chúng có thể được gọi từ các hợp đồng khác và thông qua các giao dịch. Một hàm bên ngoài f không thể được gọi trong nội bộ. Các hàm bên ngoài đôi khi hiệu quả hơn khi chúng nhận các mảng dữ liệu lớn, vì dữ liệu không được sao chép từ calldata vào bộ nhớ.

### 2.5.2. Public

Public function là một phần của giao diện hợp đồng và có thể được gọi nội bộ hoặc thông qua tin nhắn. Đối với các biến trạng thái công khai, một hàm getter tự động (xem bên dưới) được tạo.

### 2.5.3. Internal

Các hàm và biến trạng thái đó chỉ có thể được truy cập nội bộ (tức là từ trong hợp đồng hiện tại hoặc các hợp đồng bắt nguồn từ nó) mà không cần sử dụng nó.

### 2.5.4. Private

Các hàm riêng và các biến trạng thái chỉ hiển thị cho hợp đồng mà chúng được định nghĩa và không hiển thị trong các hợp đồng dẫn xuất.

Và ta có ví dụ sau:

```
contract Test {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    function getData() public returns(uint) { return data; }
}
```

### 3. Inheritance and Modifier

#### 3.1. "Contract Inheritance" trong Solidity

Trong Solidity, phạm vi kế thừa bị giới hạn với các công cụ sửa đổi truy cập nội bộ và công khai, có nghĩa là chỉ các thuộc tính công khai và nội bộ của hợp đồng cơ sở mới có sẵn cho hợp đồng dẫn xuất.

Trình biên dịch Solidity biên dịch tất cả mã hợp đồng cơ sở thành một hợp đồng duy nhất và hợp đồng duy nhất đó sẽ chuyển đến blockchain.

Các kiểu kế thừa trong solidity bao gồm:

Single Inheritance

Multi-level Inheritance

Hierarchical Inheritance

Multiple Inheritance

Để kế thừa trong solidity ta dùng từ khóa `is` để thể hiện kế thừa.

Ta có ví dụ sau đây:

```
contract baseContract {
    //parent contract
}

contract derivedContract is baseContract {
    //child contract
}
```

Ta có hai contract với contract `baseContract()` là contract cha và `derivedContract()` là contract con, và như trên code, ta dùng từ khóa `is` để biểu diễn sự kế thừa và cụ thể là lớp con được kế thừa từ lớp cha.

#### 3.2. "Modifier" trong Solidity

Modifier được sử dụng trong một phương thức trong Smart Contract để kiểm tra các điều kiện trước khi các đoạn mã code trong phương thức đó được thực thi (điều này là vô cùng quan trọng trong việc đảm bảo được tính an toàn trong Smart Contract)

Modifier khai báo như thuộc tính trong một Smart Contract, có thể được kế thừa và sử dụng bởi các Smart Contract con. Ví dụ:

```
contract Ownable {
    address public owner;
    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        if (msg.sender != owner) {
            revert();
        }
        _;
    }
}

// ta sẽ viết contract Killable để kế thừa modifier "onlyOwner" từ "Ownable"
contract Killable is Ownable {
    function kill() onlyOwner public {
        selfdestruct(msg.sender);
    }
}

// function kill() sẽ kế thừa modifier onlyOwner từ contract Ownable và dùng bình thường
```

### 3.3. Cách thức import file vào Solidity

Cách đơn giản nhất là ta chỉ cần dùng import, cụ thể qua ví dụ dưới đây:

```
pragma solidity ^0.4.22;

import "hehehehiuhehe.sol";
```

## LAB 11

Đầu tiên ta tạo 1 folder và trong folder đó tạo 1 file mới.

Trước khi thực hiện 1 contract, ta cần phải khai báo phiên bản đang sử dụng của Solidity đang ở phiên bản bao nhiêu.

Ta dùng “pragma”:

```
pragma solidity ^0.5.13;
```

Tiếp theo ta tạo 1 Smart Contract với tên Exception{}

Sử dụng mapping để ánh xạ với khóa là address và giá trị là uint gán là biến balanceReceived.

```
contract Exception{

    mapping(address => uint) public balanceReceived;
```



Ta tạo 1 function `receiveMoney()`: có chức năng nhận số ether từ địa chỉ đó gửi đến. Lúc này, giá trị được gán là `balanceReceived` sẽ nhận được số dư tương ứng với giá trị `value` đã gửi.

```
function receiveMoney() public payable{
    balanceReceived[msg.sender] += msg.value;
}
```

Tiếp theo, ta tạo 1 hàm `withdrawMoney()`:

```
function withdrawMoney(address payable _to, uint _amount) public{
    if(_amount <= balanceReceived[msg.sender]){
        balanceReceived[msg.sender] -= _amount;
        _to.transfer(_amount);
    }
}
```

Ta dùng `if` để đặt điều kiện số dư nhận được của địa chỉ đó phải lớn hơn hoặc bằng số tiền muốn chuyển đi.

Khi số tiền đã chuyển đi thành công thì số dư nhận được sẽ bị trừ tương ứng với số tiền chuyển đi.

Khi đó, địa chỉ `_to` sẽ được chuyển đến số tiền tương ứng.

Bây giờ, ta tiến hành Deploy và giao dịch thành công.

Giờ ta thực hiện gửi 1 ether vào smart contract.

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT

JavaScript VM

ACCOUNT

0x5B3...eddC4 (99.999999999)

GAS LIMIT

3000000

VALUE

1

ether

Sau đó, ta kéo xuống và nhấn vào receiveMoney để thực hiện.

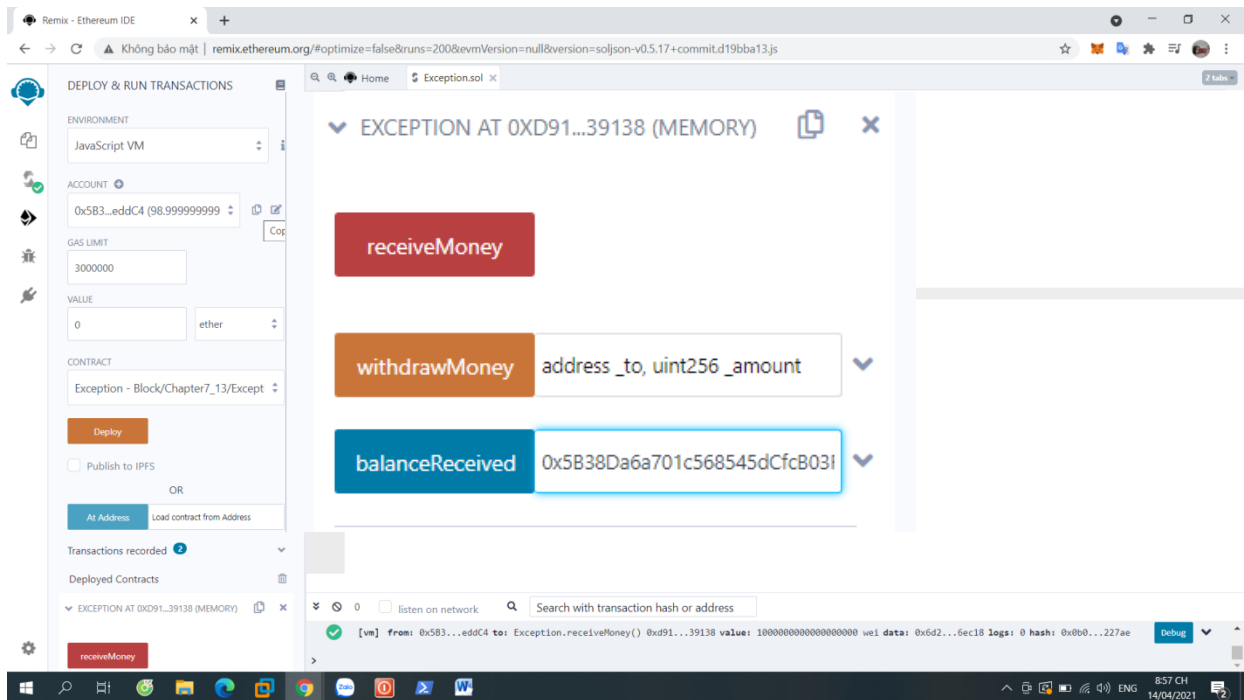
▼ EXCEPTION AT 0XD91...39138 (MEMORY)

receiveMoney

withdrawMoney address\_to, uint256 \_amount

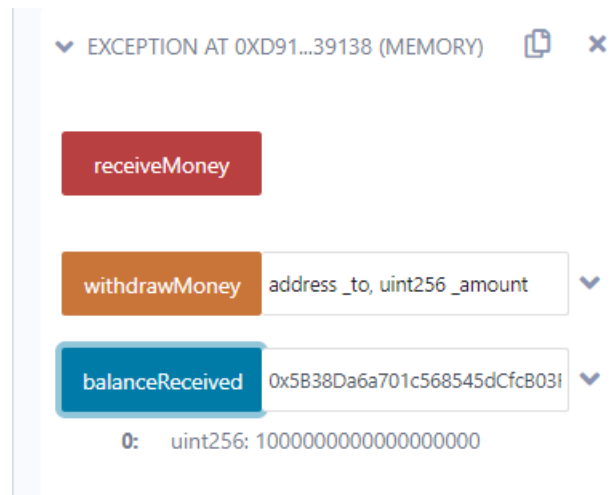
balanceReceived address

Lúc này, ta muốn xem số dư hiện tại của địa chỉ này, ta copy địa chỉ đó.

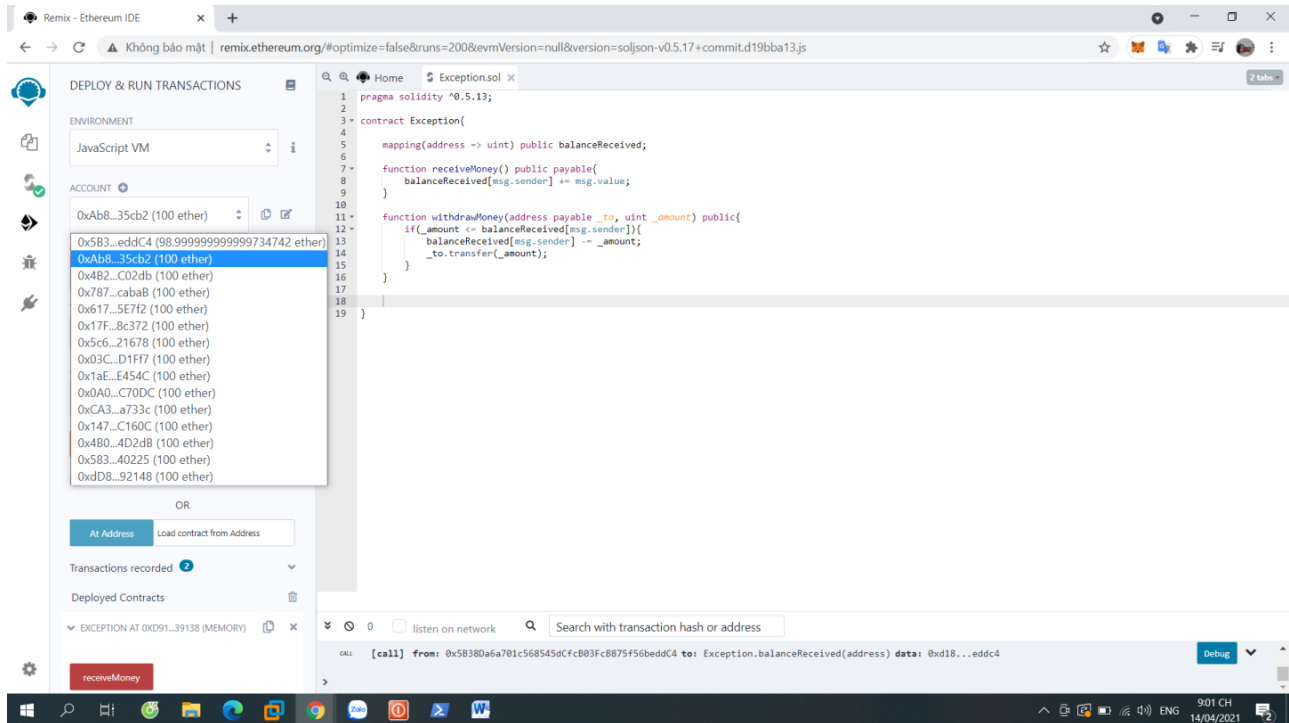


Sau đó, ta dán vào trường balanceReceived.

Và nhấn nút balanceReceived để xem số dư. Và kết quả:



Bây giờ, ta thực hiện chuyển tiền đến một tài khoản bất kì.



Ta chọn địa chỉ thứ 2 và copy địa chỉ đó lại và thực hiện dán vào trường `withdrawMoney`.

Tiếp theo, ta nhập số ether cần chuyển là bao nhiêu. Ở đây, ta chuyển đến tài khoản thứ 2 hết số ether hiện có trong số dư.

The image shows a close-up of the 'withdrawMoney' function call in the Remix IDE. The '\_to' field is filled with the address '0xAb8483F64d9C6d1EcF9b849Ae6'. The '\_amount' field is filled with '1000000000000000000'. A 'transact' button is visible at the bottom right.

Sau đó, ta nhấn **transact** để thực hiện giao dịch. Và giao dịch thực thi thành công.



```
[vm] from: 0x583...eddC4 to: Exception.withdrawMoney(address,uint256) 0xd91...39138 value: 0 wei data: 0xf27...40000 logs: 0 hash: 0xacf...ce537
```

status: true Transaction mined and execution succeed

transaction hash: 0xacf5d98a6c632f21cf358a23114a1f5e92e63daf3cb9dd2799bb0a482cdce537

from: 0x5838Da6a701c568545dcFcB03FcB875f56beddC4

to: Exception.withdrawMoney(address,uint256) 0xd9145CCE52D386f254917e481e844e9943f39138

gas: 3000000 gas

transaction cost: 22815 gas

execution cost: 14623 gas

hash: 0xacf5d98a6c632f21cf358a23114a1f5e92e63daf3cb9dd2799bb0a482cdce537

input: 0xf27...40000

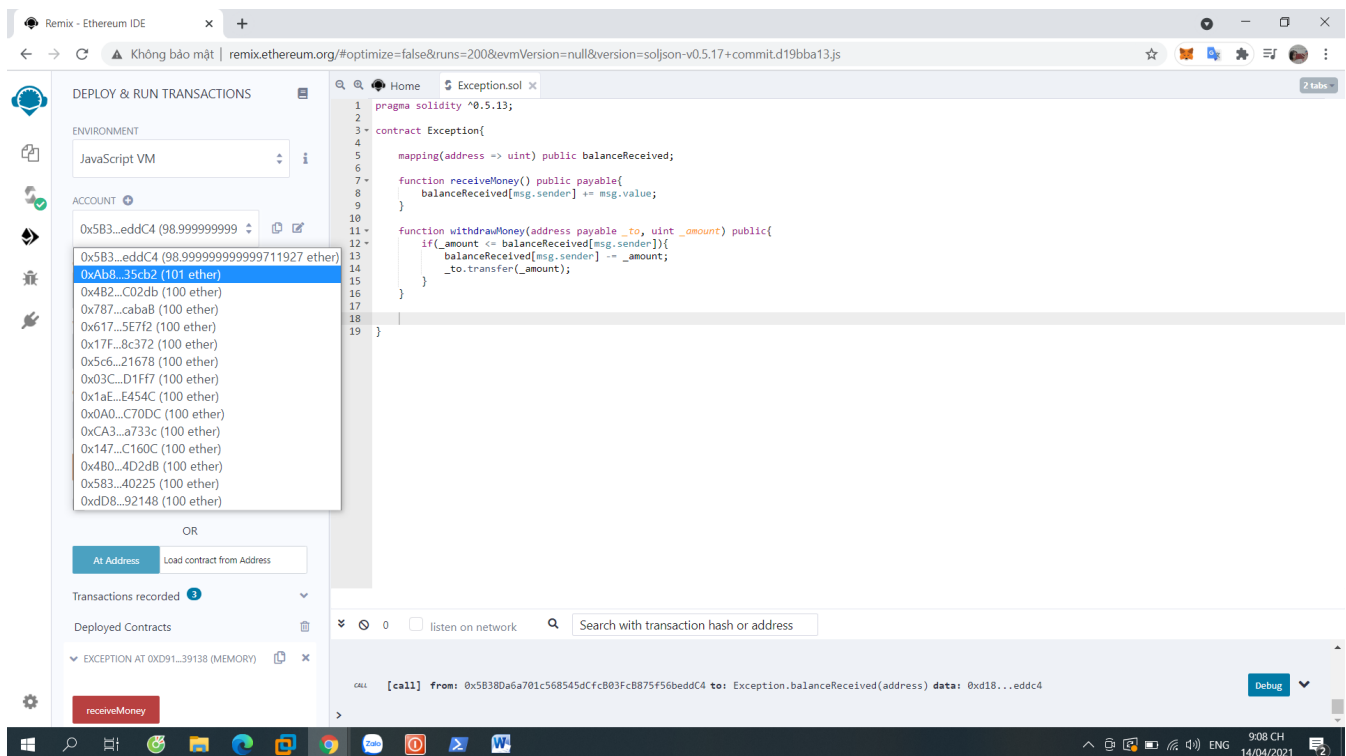
decoded input: { "address \_to": "0xab8483f64d9c6d1ecf9b849ae677d03315835cb2", "uint256 \_amount": { "type": "BigNumber", "hex": "0x0de0b3a7640000" } }

decoded output: {}

logs: []

value: 0 wei

Khi đó, số dư của địa chỉ thứ 2 sẽ tăng lên 1 ether.



Remix - Ethereum IDE

remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

ENVIRONMENT: JavaScript VM

ACCOUNT: 0x583...eddC4 (98.999999999 ether)

0xab8...35cb2 (101 ether)

0x4B2...C02db (100 ether)

0x787...cabaB (100 ether)

0x617...5E7f2 (100 ether)

0x17F...8c372 (100 ether)

0x5c6...21678 (100 ether)

0x03C...D1Ff7 (100 ether)

0x1aE...E454C (100 ether)

0x0A0...C70DC (100 ether)

0xCA3...a733c (100 ether)

0x147...C160C (100 ether)

0x4B0...4D2dB (100 ether)

0x583...40225 (100 ether)

0xdD8...92148 (100 ether)

OR

At Address: Load contract from Address

Transactions recorded: 1

Deployed Contracts: EXCEPTION AT 0XD91...39138 (MEMORY)

receiveMoney

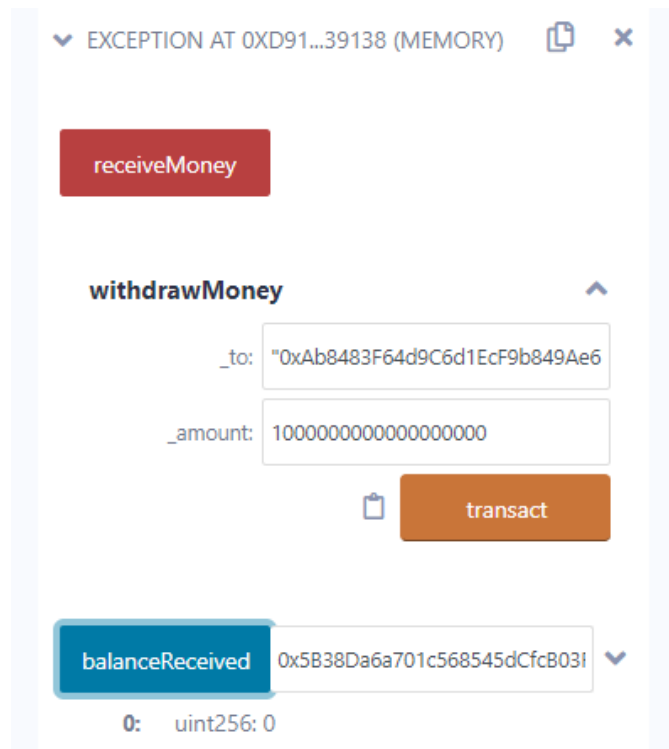
```
1 pragma solidity ^0.5.13;
2
3 contract Exception{
4
5     mapping(address => uint) public balanceReceived;
6
7     function receiveMoney() public payable{
8         balanceReceived[msg.sender] += msg.value;
9     }
10
11     function withdrawMoney(address payable _to, uint _amount) public{
12         if(_amount <= balanceReceived[msg.sender]){
13             balanceReceived[msg.sender] -= _amount;
14             _to.transfer(_amount);
15         }
16     }
17
18 }
19 }
```

CALL [call] from: 0x5838Da6a701c568545dcFcB03FcB875f56beddC4 to: Exception.balanceReceived(address) data: 0xd18...eddC4

Debug

9:08 CH 14/04/2021

Khi đã chuyển thành công, lúc này, số dư nhận được hiện tại sẽ là:



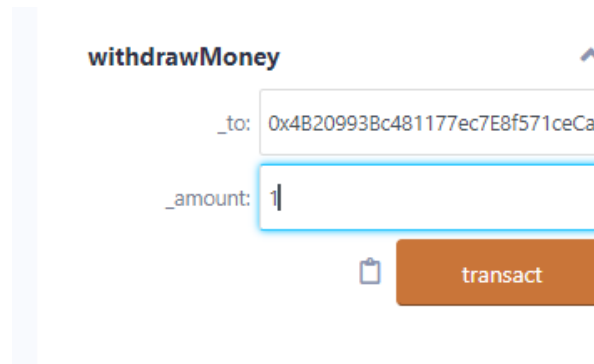
Các bạn chú ý số tiền muốn gửi phải nhỏ hơn hoặc bằng với số dư hiện tại nhé!

Thay vì, ta sử dụng if thì ta sẽ sử dụng require để đưa ra yêu cầu và có thể kiểm tra lỗi. Vì khi ta dùng require khi xảy ra lỗi, lỗi sẽ hiển thị và dừng contract nhưng vẫn hoàn nguyên lại ban đầu. Như lúc ban đầu, khi ta sử dụng if, dù xảy ra lỗi contract vẫn thực hiện.

```
function withdrawMoney(address payable _to, uint _amount) public{
    require(_amount <= balanceReceived[msg.sender], "You don't have enough ether!");
    balanceReceived[msg.sender] -= _amount;
    _to.transfer(_amount);
}
```

Lúc này, ta thực hiện Deploy, và cùng xem khi chúng ta chưa có số dư thì chúng ta có thể thực hiện chuyển tiền được hay không nha.

Ta sao chép 1 địa chỉ và dán vào trường withdrawMoney cùng với số ether muốn chuyển.



Sau đó, ta thực hiện giao dịch nhấn vào transact:

 [vm] from: 0x4B2...C02db to: Exception.withdrawMoney(address,uint256) 0xD7A...F771B value: 0 wei data: 0xf27...00001 logs: 0 hash: 0x494...b630a Debug 

transact to Exception.withdrawMoney errored: VM error: revert. revert The transaction has been reverted to the initial state. Reason provided by the contract: "You don't have enough ether!". Debug the transaction to get more information.

>

Ta sẽ thấy thông báo trả về lỗi.

Bây giờ, ta thay đổi một tí:

```
mapping(address => uint64) public balanceReceived;

function receiveMoney() public payable{
    balanceReceived[msg.sender] += uint64(msg.value);
}

function withdrawMoney(address payable _to, uint64 _amount) public{
    require(_amount <= balanceReceived[msg.sender], "You don't have enough ether!");
    balanceReceived[msg.sender] -= _amount;
    _to.transfer(_amount);
}
```

Ta thay thế kiểu giá trị uint thành uint64 và thực hiện Deploy và cũng tiến hành gửi tiền chuyển tiền như lúc này.



ENVIRONMENT

JavaScript VM

ACCOUNT +

0x5B3...eddC4 (98.999999999)

GAS LIMIT

3000000

VALUE

10 ether

Ta gửi đi 10ether và sao chép địa chỉ dán vào trường balanceReceived để xem số dư.

EXCEPTION AT 0X7EF...8CB47 (MEMORY)

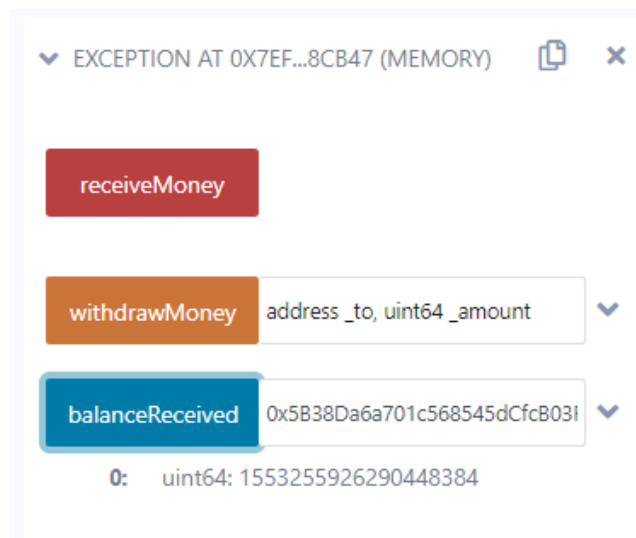
receiveMoney

withdrawMoney address\_to, uint64\_amount

balanceReceived 0x5B38Da6a701c568545dCfcB03I

0: uint64: 1000000000000000000

Sau đó, ta thực hiện gửi thêm 10 ether như vậy nữa và xem số dư lúc này sẽ như thế nào.



Lúc này, số dư sẽ trả về giá trị kiểu uint64 chứ không phải uint.

## LAB 12:

Ta tạo 1 folder và 1 file mới nằm trong folder đó.

Sau đó, ta khai báo phiên bản của Solidity.

Ta copy file của Lab 11 bỏ vào file của Lab 12 và tiếp tục thực hiện.

Lúc này, ta thực hiện Deploy.

```
pragma solidity ^0.5.13;

contract FunctionSimple{

    mapping(address => uint) public balanceReceived;

    function receiveMoney() public payable{
        assert(balanceReceived[msg.sender] + uint64(msg.value) >= balanceReceived[msg.sender]);
        balanceReceived[msg.sender] += uint64(msg.value);
    }

    function withdrawMoney(address payable _to, uint64 _amount) public{
        require(_amount <= balanceReceived[msg.sender], "You don't have enough ether!");
        assert(balanceReceived[msg.sender] >= balanceReceived[msg.sender] - _amount);
        balanceReceived[msg.sender] -= _amount;
        _to.transfer(_amount);
    }
}
```

Ta thực hành gửi và nhận 1 ether.

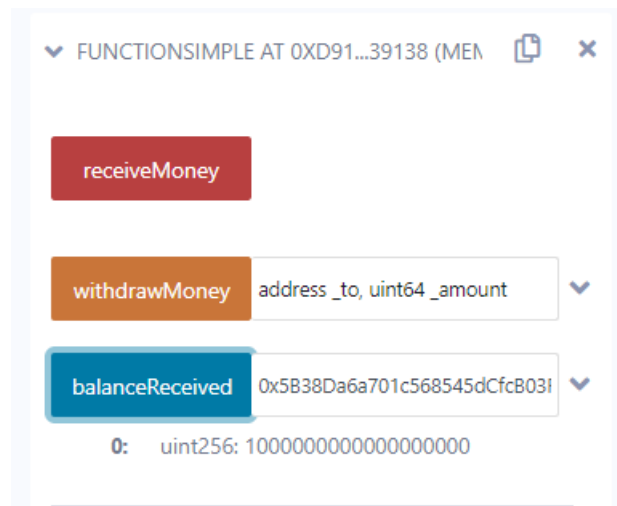
[vm] from: 0x5B3...eddC4 to: FunctionSimple.receiveMoney() 0xd91...39138 value: 10000000000000000 wei data: 0x6d2...6ec18 logs: 0

hash: 0x0b0...227ae

Debug ^

status	true Transaction mined and execution succeed
transaction hash	0x0b0f8eca4cd7aa56867a9aac4010ee65eaf80cbccad5dd99a18e4304575227ae
from	0x5B38Da6a701c568545dCfcB03FcB875F56beddC4
to	FunctionSimple.receiveMoney() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	3000000 gas
transaction cost	44134 gas
execution cost	22862 gas
hash	0x0b0f8eca4cd7aa56867a9aac4010ee65eaf80cbccad5dd99a18e4304575227ae
input	0x6d2...6ec18
decoded input	{ }
decoded output	{ }
logs	[ ]
value	10000000000000000 wei

Muốn kiểm tra số dư của địa chỉ, ta copy địa chỉ đó và dán vào trường balanceReceived.

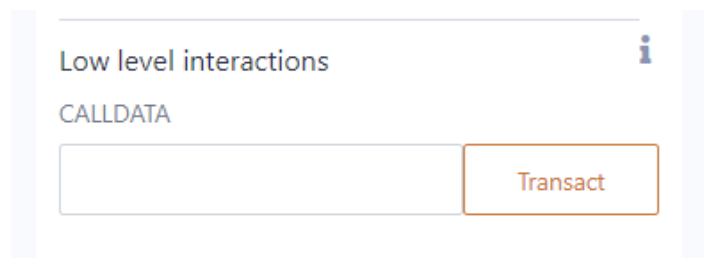


Ta tạo 1 function:

```
function() external payable{
    receiveMoney();
}
```

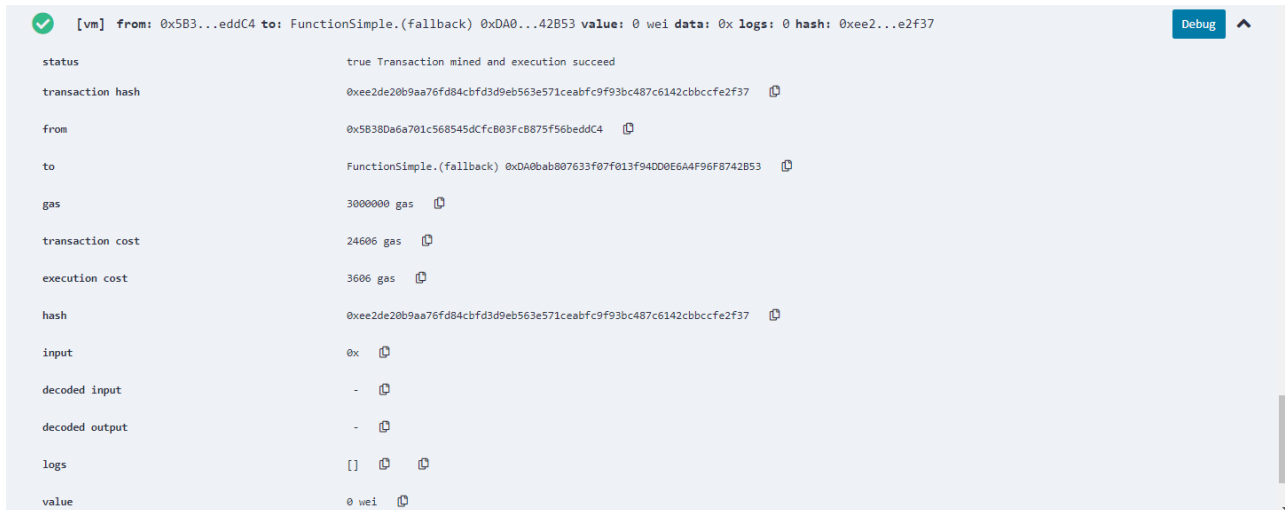
Ta thấy, có điều gì đặc biệt từ hàm này? Đó là hàm này không có tên hàm và được sử dụng từ khóa external có nghĩa là chúng có thể được gọi từ các hợp đồng khác.

Bây giờ, ta thực thi hàm và gọi 1 từ hợp đồng khác xem kết quả ra sao nha:



Ta thực hiện bấm vào Transact để gọi 1 hợp đồng từ bên ngoài.

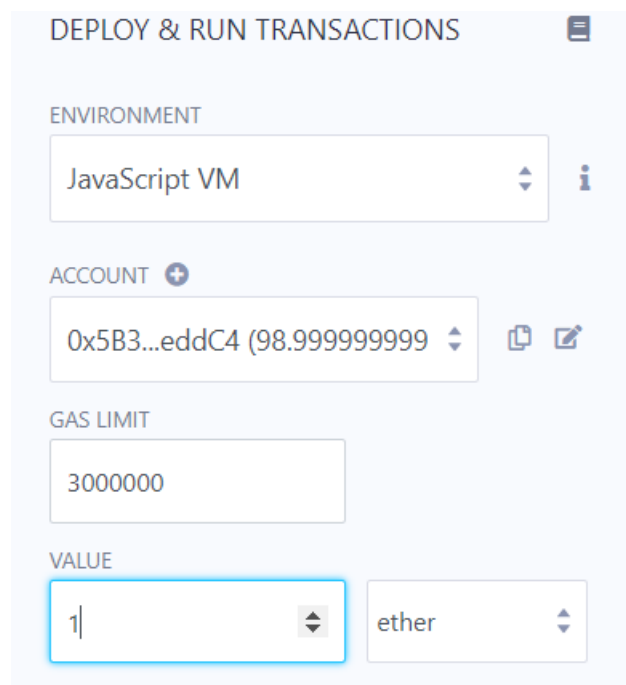
Lúc này, ta chú ý thông báo giao dịch hiển thị.



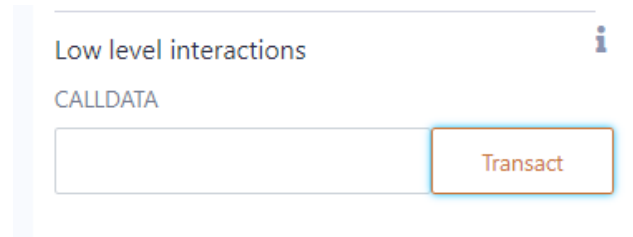
Các bạn chú ý từ khóa fallback: Hàm fallback để nhận Ether và thêm nó vào tổng số dư của hợp đồng, chức năng này phải được đánh dấu payable.

Nó được thực thi khi có ai đó gọi hàm không có trong contract hoặc tham số không đúng.

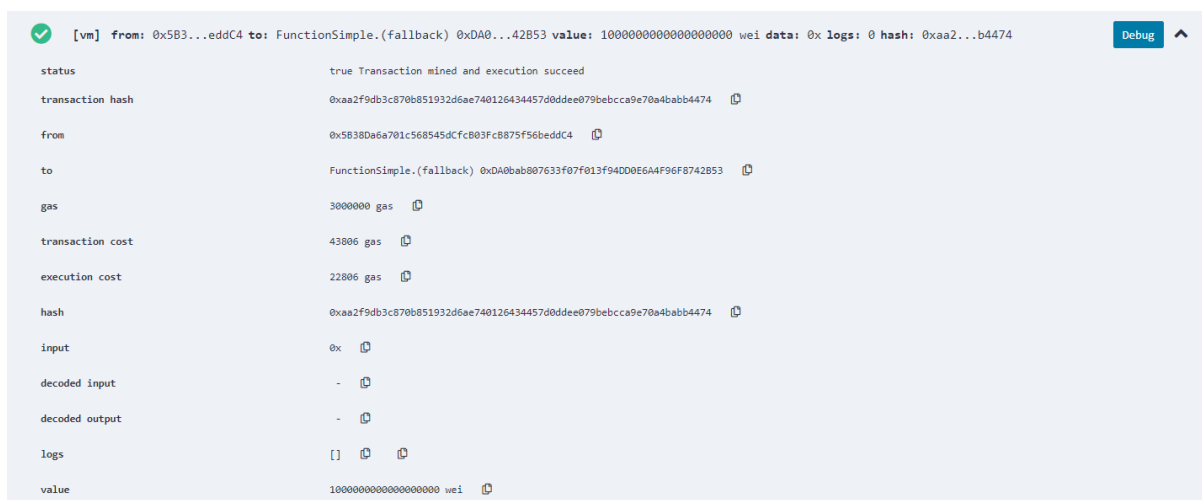
Bây giờ, ta thực hiện gửi ether và gọi giao dịch từ bên ngoài:



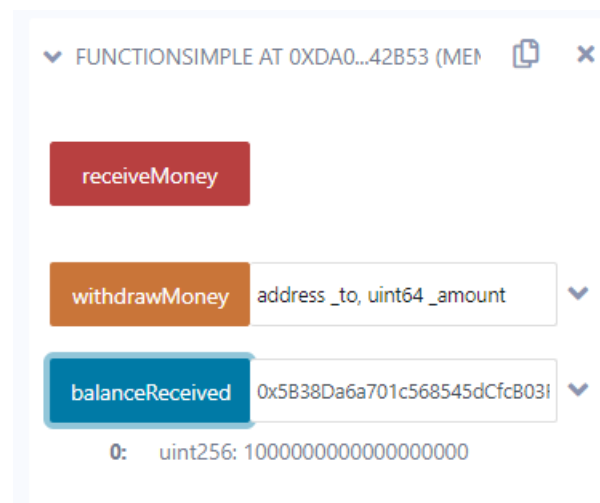
Tiếp theo, ta nhấn gọi Transact:



Như vậy, giao dịch thành công.



Giờ ta kiểm tra số dư : nhận thành công.



Bây giờ, ta update code. Ta thêm biến owner để tạo địa chỉ cho chủ sở hữu.

```
address payable owner;

constructor() public{
    owner = msg.sender;
}
```

Hàm constructor nhận đầu vào phải là địa chỉ của chủ sở hữu. Và chỉ chủ sở hữu mới có quyền nhận ether.

Ta tạo 1 function getOwner để lấy và trả về địa chỉ của chủ sở hữu.

```
address payable owner;

constructor() public{
    owner = msg.sender;
}

function getOwner() public view returns(address){
    return owner;
}
```

Tiếp theo, ta tạo 1 function destroySmartContract

```
address payable owner;

constructor() public{
    owner = msg.sender;
}

function getOwner() public view returns(address){
    return owner;
}

function destroySmartContract()public{
    require(msg.sender == owner, " You are not the owner!");
    selfdestruct(owner);
}
```

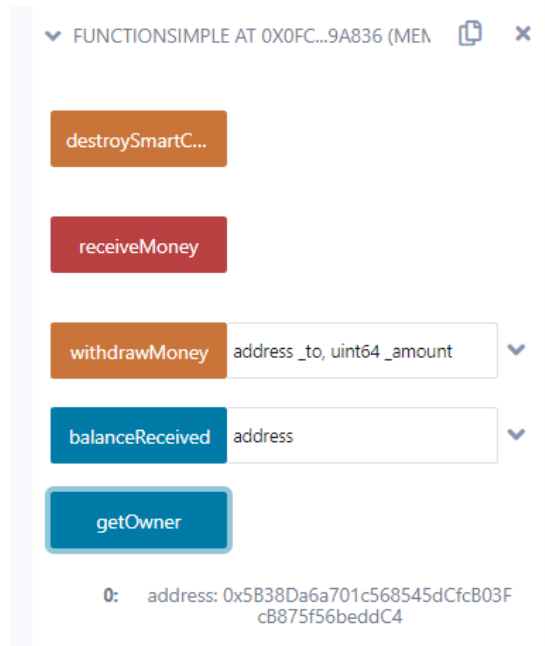
Chúng ta không thể xóa thông tin khỏi Blockchain, nhưng chúng ta có thể cập nhật trạng thái hiện tại để ta không thể tương tác với một địa chỉ nữa trong tương lai. Khi quá trình tự cấu trúc được gọi, bạn không thể tương tác với Hợp đồng thông minh nữa.

Khi “selfdestruct” được gọi, thực hiện hủy SmartContract, số ether đã được gửi sẽ được chuyển ngược lại địa chỉ của owner.

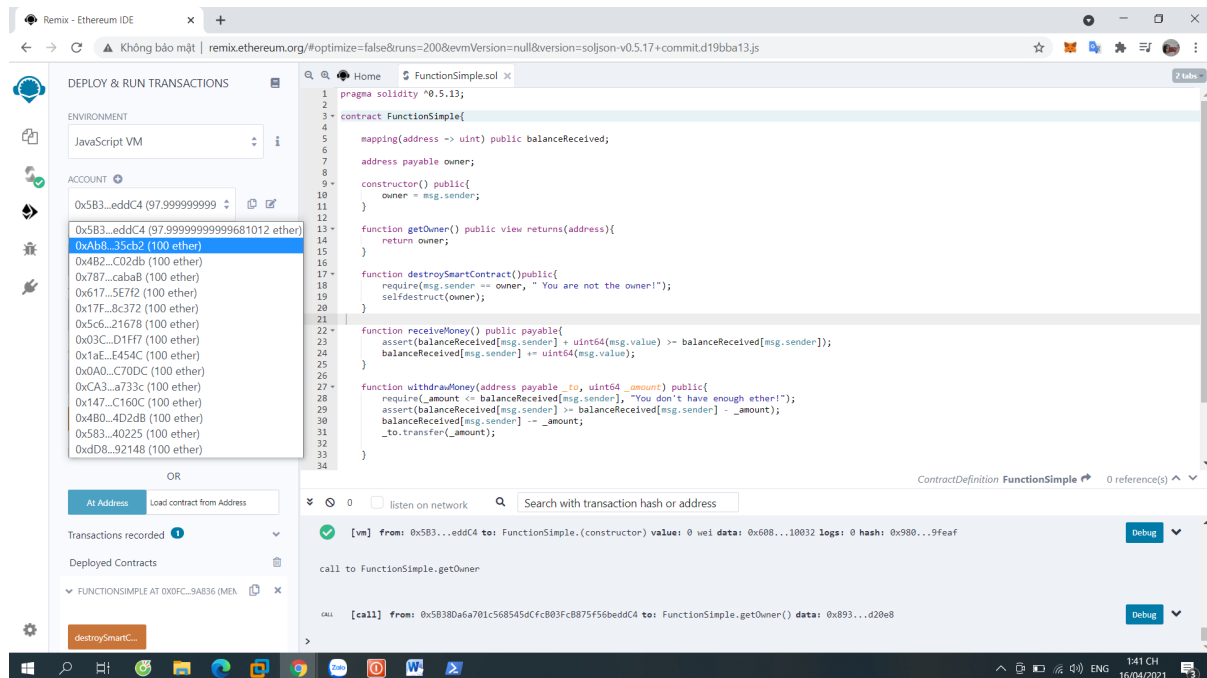


Bây giờ, ta Deploy.

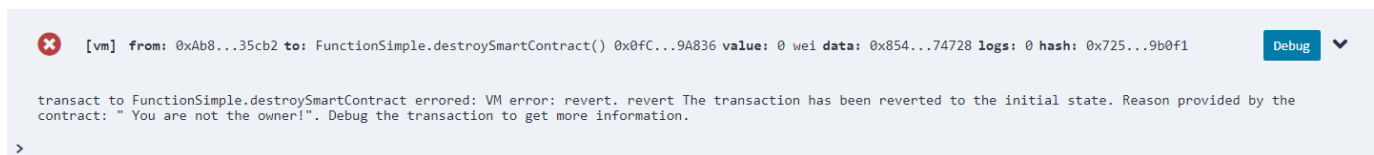
Khi Deploy thành công, để kiểm tra xem địa chỉ nào là địa chỉ của chủ sở hữu, ta nhấn vào getOwner.



Giờ ta chuyển account sang 1 địa chỉ khác và bấm vào destroySmartContract xem sao.



Khi đó thông báo hiển thị:



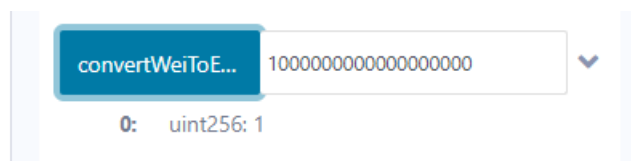
Lỗi này dễ hiểu vì hàm destroy có require yêu cầu địa chỉ được phép destroy phải là owner. Và khi ta chuyển sang 1 địa chỉ khác thì hiển nhiên lỗi sẽ xuất hiện.

Tiếp theo, ta tạo 1 hàm:

```
function convertWeiToEther(uint _amountInWei) public pure returns(uint){
    return _amountInWei / 1 ether;
}
```

Hàm này có chức năng chuyển giá trị Wei sang Ether.

Khi ta nhập giá trị có đơn vị là Wei khi call hàm convertWeiToEther thì giá trị trả về sẽ có đơn vị Ether.



## LAB 13:

Đầu tiên ta tạo 1 folder và tạo 1 file trong folder đó.

Trước khi bắt đầu tạo 1 contract, ta cần khai báo phiên bản của Solidity đang ở phiên bản bao `pragma solidity ^0.5.11;` nhiều:

Bắt đầu tạo 1 Smart Contract, mình đặt tên cho contract là InheritanceModifier{ }.

```
pragma solidity ^0.5.11;

contract InheritanceModifier{
```

Tiếp theo, ta khai báo các kiểu dữ liệu:

```
contract InheritanceModifier{

    mapping(address => uint) public tokenBalance;

    address owner;

    uint tokenPrice = 1 ether;

    constructor() public{
        owner = msg.sender;
        tokenBalance[owner] = 100;
    }
}
```

Đầu tiên, ta có kiểu mapping với kiểu khóa là address và giá trị là uint được gán biến tokenBalance.

Kiểu address với biến là owner

Kiểu dữ liệu uint có biến tokenPrice được gán giá trị là 1 ether.

Và 1 hàm constructor với yêu cầu đầu vào phải là địa chỉ của owner và lúc đó tokenBalance của địa chỉ owner sẽ được gán là 100. Và khi ta gọi biến tokenBalance thì mặc định sẽ là 100.

Tiếp theo, ta tạo các function:

```
function createNewToken() public{
    require(msg.sender == owner, "You are not allowed!");
    tokenBalance[owner]++;
}
```

Function này yêu cầu địa chỉ được thực hiện phải là địa chỉ của owner. Và chỉ có địa chỉ của owner mới có thể tạo token. Và giá trị trả về là khi ta gọi tokenBalance của owner sẽ tăng dần lên 1. Ta có biến đếm ++.

```
function burnToken() public{
    require(msg.sender == owner, "You are not allowed!");
    tokenBalance[owner] --;
}
```

Với function ta thấy khá tương tự như function trên. Nhưng ở đây, giá trị trả về biến sẽ là --. Tức là khi ta thực thi hàm thành công, và khi mỗi lần ta gọi biến tokenBalance thì giá trị sẽ trừ dần 1.

Ta tạo tiếp 1 function:

```
function purchaseToken()public payable{
    require((tokenBalance[owner] * tokenPrice) / msg.value > 0, "not enough tokens");
    tokenBalance[owner] -= msg.value / tokenPrice;
    tokenBalance[msg.sender] += msg.value / tokenPrice;
}
```

Function này có yêu cầu điều kiện là số tokenBalance của owner nhân với số tokenPrice( ở đây là 1ether) khi chia với giá trị value nhập vào mà lớn hơn 0 thì mới tiếp tục thực thi.

Và kết quả trả về , đối với tokenBalance của owner sẽ bị trừ đi tương ứng với số value. Và số tokenBalance của 1 địa chỉ khác sẽ được cộng vào value.

Tiếp đến, ta tạo 1 function sendToken có chức năng chuyển tiền.

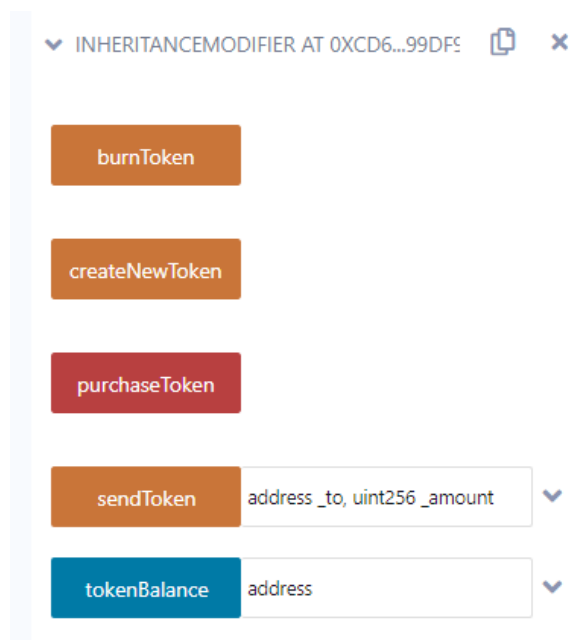
```
function sendToken(address _to, uint _amount) public{
    require(tokenBalance[msg.sender] >= _amount, "Not enough tokens");
    assert(tokenBalance[_to] + _amount >= tokenBalance[_to]);
    assert(tokenBalance[msg.sender] - _amount <= tokenBalance[msg.sender]);
    tokenBalance[msg.sender] -= _amount;
    tokenBalance[_to] += _amount;
}
```

Yêu cầu của hàm là số tokenBalance của địa chỉ muốn chuyển phải lớn hơn hoặc bằng số tiền cần chuyển.

Lúc này, khi thực thi hàm. Kết quả trả về sẽ là tokenBalance của địa chỉ chuyển sẽ bị trừ đi tương ứng số tiền đã chuyển

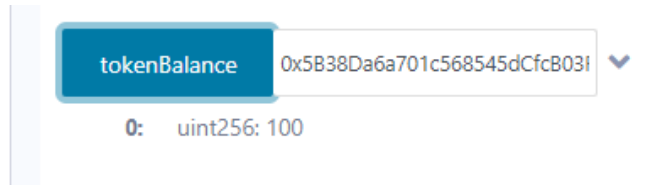
Và địa chỉ được chuyển tiền đến sẽ nhận được tương ứng số tiền đã được chuyển.

Bây giờ, ta tiến hành thực thi contract:



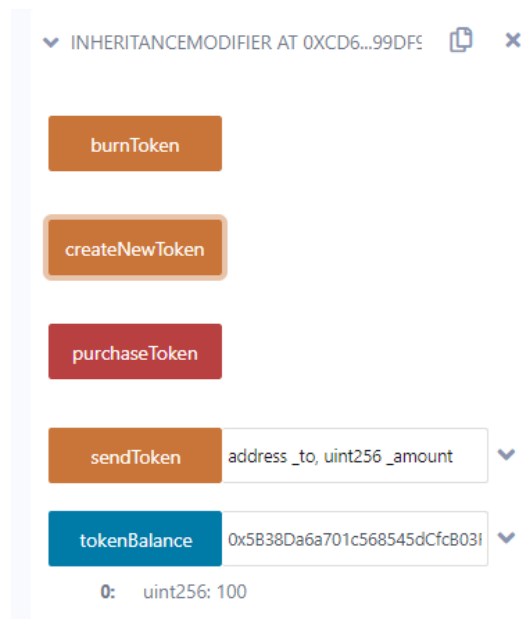
Giờ để xem giá trị ban đầu ta gán cho địa chỉ của owner có đúng là 100 hay không. Ta sao chép địa chỉ của owner (là địa chỉ đã thực hiện Deploy contract) và dán vào trường tokenBalance.

Tiếp đó, ta nhấn vào nút tokenBalance và xem kết quả:

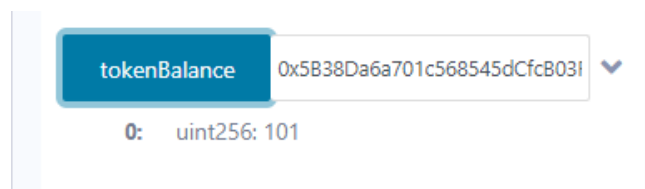


Bây giờ ta hãy thử thực thi 2 hàm `createNewToken` và `burnToken` xem sao nha:

Rất đơn giản ta chỉ cần nhấn vào nút `createNewToken`:



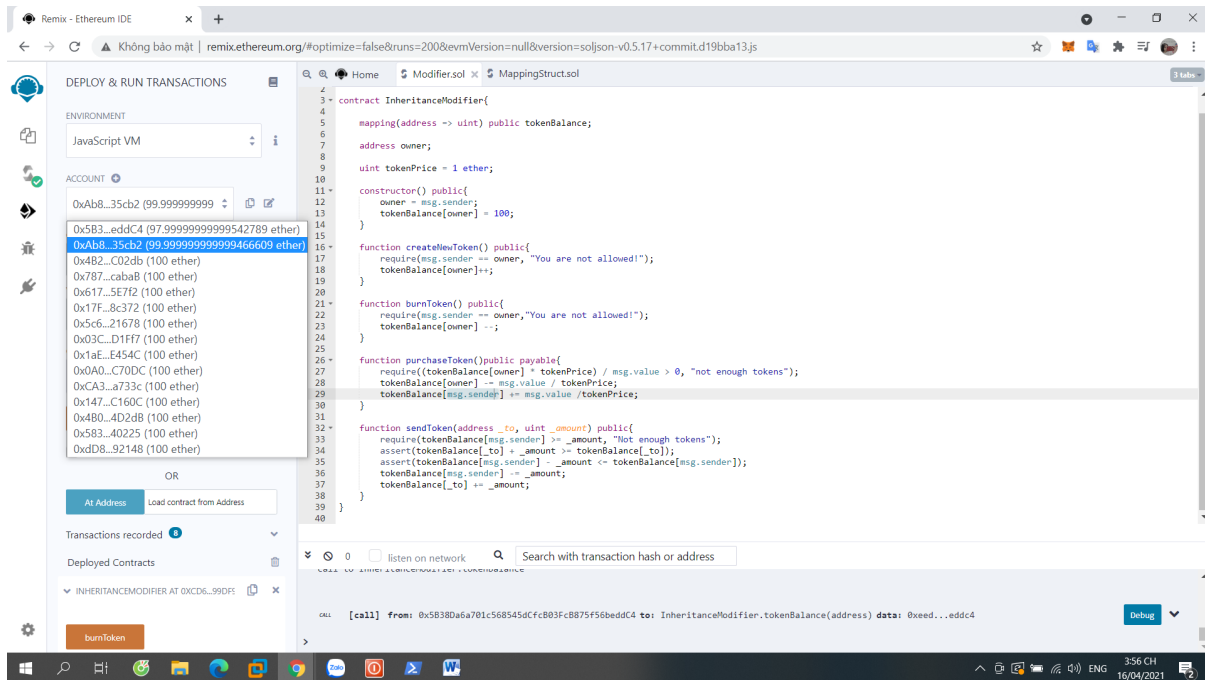
Lúc này, sẽ chưa có gì xảy ra. Để biết được nó đã thay đổi như thế nào hãy nhấn vào lại nút `tokenBalance`.



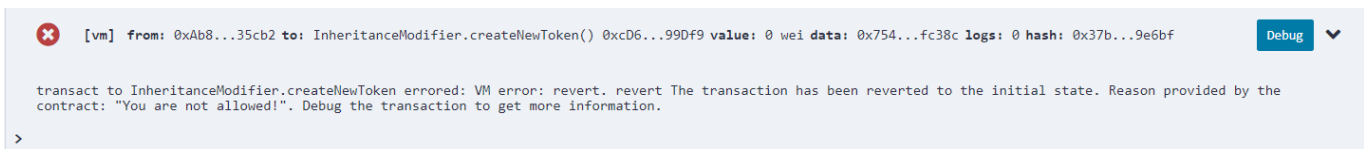
Giá trị đã tăng lên 1.

Ngược lại, ta nhấn vào `burnToken` để vớt đi 1. Cứ thực hiện như vậy ta sẽ hiểu hơn.

À! Còn 1 vấn đề nữa, này giờ mình chỉ thực hiện trên địa chỉ của owner. Bây giờ hãy thử chuyển sang 1 địa chỉ khác và thực hiện các bước như trên xem có được không nhé !

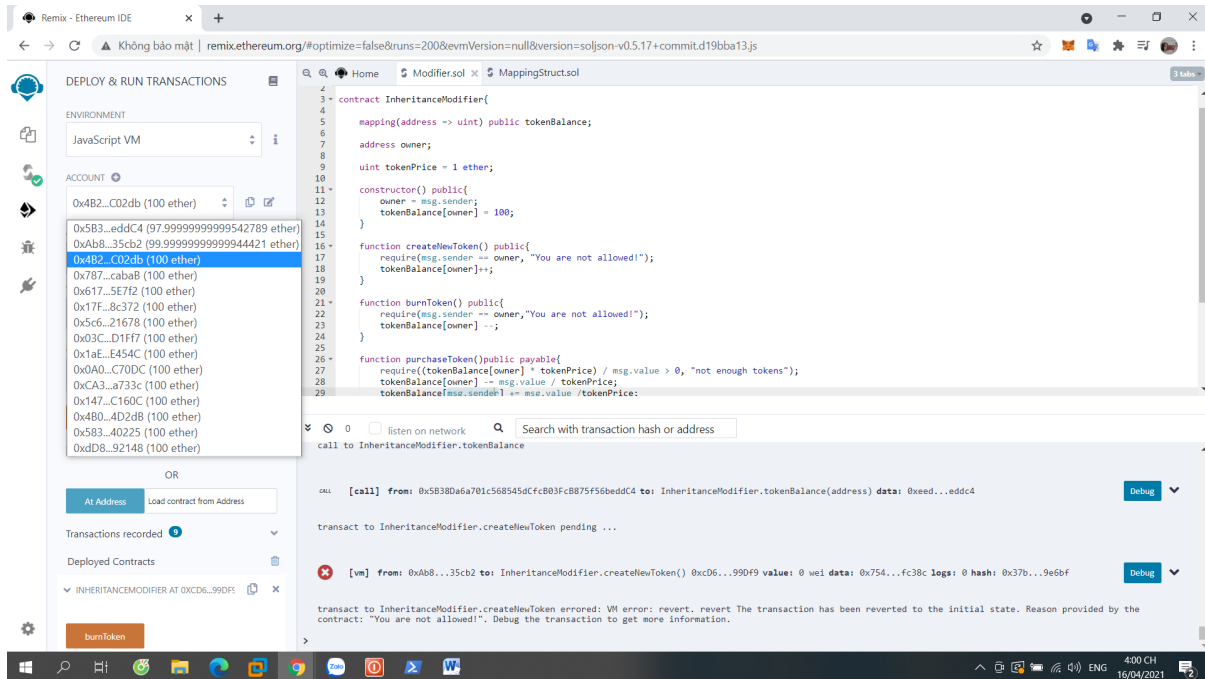


Mình chuyển sang địa chỉ thứ 2. Và thực thi createNewToken.

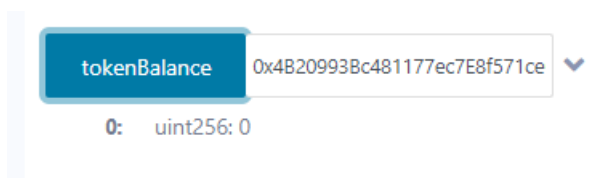


Các bạn sẽ nhận được thông báo lỗi. Hiển nhiên rồi, vì điều kiện ta đã đưa ra cho các hàm đó là địa chỉ được thực thi phải là owner.

Tiếp theo nữa, mình sẽ chọn chuyển sang tài khoản thứ 3 và sao chép địa chỉ đó dán vào trường tokenBalance.



Và đương nhiên giá trị của tài khoản này sẽ là 0.



Chúng ta đang ở tài khoản thứ 3 và bây giờ chúng ta thực hiện chức năng gửi ether qua hàm purchaseToken.

Ở đây, mình chọn gửi 1 ether.



Sau đó, ta kéo xuống và nhấn vào purchaseToken. Thông báo giao dịch thành công.

Sau khi đã gửi thành công, ta thực hiện gọi tokenBalance để xem địa chỉ đó trả về gì:

Bây giờ, ta thực hiện chuyển tiền:

Từ địa chỉ thứ 3 đó, ta chuyển sang 1 tài khoản khác( mình chọn tài khoản thứ 4) và sao chép địa chỉ đó lại. Và đừng quên sau khi sao chép xong thì quay trở lại địa chỉ thứ 3 nha.

Lúc này ta kéo xuống và dán vào trường sendToken.

### sendToken

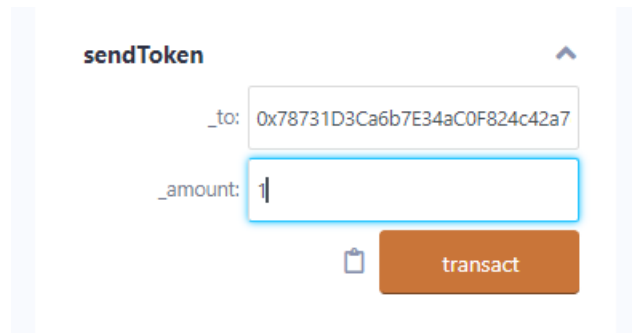
\_to: 0x78731D3Ca6b7E34aC0F824c42a7

\_amount: uint256



transact

Ở hàm sendToken, ta còn 1 giá trị nữa là uint. Ta nhập giá trị cần chuyển vào. Mình chuyển 1.



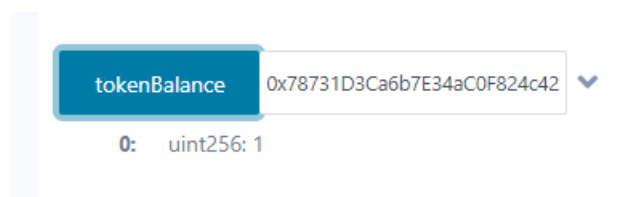
Sau đó, ta nhấn transact để bắt đầu giao dịch.

Sau khi đã giao dịch, ta nhấn vào tokenBalance để xem số token đã bị chuyển đi chưa nhé.



Như vậy, là đã thành công.

Bây giờ, ta chuyển sang địa chỉ thứ 4 để kiểm tra số tokenBalance. Ta sao chép địa chỉ thứ 4 và dán vào trường tokenBalance.



Địa chỉ thứ 4 đã được chuyển vào 1ether.

Ta thấy code ta khá dài, khi có tới 2 hàm có yêu giống nhau và được lặp lại 2 lần. Vậy có cách nào để code chúng ta có thể giản đơn hơn được không?

Câu trả lời là có. Ta sẽ sử dụng hàm Modifier. Modifier là gì ?

Modifier là hàm được sử dụng trong Smart Contract để kiểm tra các điều kiện trước khi các đoạn mã code trong phương thức đó được thực thi. Đảm bảo được tính an toàn trong Smart Contract.

Vậy giờ ta sẽ tiến hành update code tí nhé !

```
modifier onlyOwner(){
    require(msg.sender == owner, "You are not allowed!");
    _;
}

function createNewToken() public onlyOwner{
    tokenBalance[owner]++;
}

function burnToken() public onlyOwner{
    tokenBalance[owner] --;
}
```

Ký tự `_;` trong modifier đại diện cho đoạn code của phương thức sử dụng modifier này và nó được thực thi ngay khi vượt qua được điều kiện đã kiểm tra tại các câu lệnh của modifier. Hiểu đơn giản: `_;` xuất hiện sẽ trả lại luồng thực thi cho function gọi nó.

Như ta thấy, 2 function đã kế thừa modifier `onlyOwner` từ contract ban đầu. Và lúc này ta vẫn sử dụng bình thường.

Ta cũng tiến hành Deploy và thử chọn 1 địa chỉ không phải là owner và thực hiện các function xem modifier hoạt động hiệu quả không nhé. Và tất nhiên là chạy tốt rồi. Các bạn hãy thử để xem có hiển thị lỗi không nha.



[vm] from: 0x787...cabaB to: InheritanceModifier.burnToken() 0xe28...4157A  
value: 0 wei data: 0xfaa...0a264 logs: 0 hash: 0xf88...ade9f

Debug



transact to InheritanceModifier.burnToken errored: VM error: revert. revert The transaction has been reverted to the initial state. Reason provided by the contract: "You are not allowed!". Debug the transaction to get more information.

Trong Solidity, ta cũng có thể sử dụng kế thừa từ các contract khác.

Đối với code của chúng ta lúc này, ta sẽ tạo 1 contract khác chứa các biến và ta sẽ kế thừa lại contract đó.

```
contract Owned{
    address owner;

    constructor() public{
        owner = msg.sender;
    }

    modifier onlyOwner(){
        require(msg.sender == owner, "You are not allowed!");
        _;
    }
}

contract InheritanceModifier is Owned{

    mapping(address => uint) public tokenBalance;

    uint tokenPrice = 1 ether;

    constructor() public{
        tokenBalance[owner] = 100;
    }

    function createNewToken() public onlyOwner{
        tokenBalance[owner]++;
    }

    function burnToken() public onlyOwner{
        tokenBalance[owner] --;
    }
}
```

Từ code lúc trước, ta update code như thế này. Như vậy, ta thấy, ta đã tạo ra thêm 1 contract mới là Owned và contract InheritanceModifier đã kế thừa contract Owned.

Để kế thừa ta dùng từ khóa is.

Đối với contract đầu tiên, nó chỉ chứa các biến và điều kiện.

Đối với contract thứ hai, vì đã được kế thừa nên nó sẽ chứa các phương thức của chính nó và của cả contract Owned.

Và khi đi Deploy và thực hiện giao dịch mọi thứ vẫn sẽ diễn ra như bình thường.

Trong Solidity, khi ta có 2 file nằm trong 1 folder và ta muốn import 1 file nào đó vào 1 file khác thì ta sẽ thực hiện theo cách nào.

Đơn giản, khi ta muốn import 1 file cùng folder thì ta chỉ cần sử dụng hàm import.

Ở bài này, ta sẽ tách contract đầu tiên sang 1 file mới nằm cùng folder nha.

Ta sẽ tạo 1 file mới và copy đoạn code của contract Owned và bỏ qua file vừa tạo.

Lúc này, ta sẽ thực hiện import file Owned.sol vừa tạo vào file Modifier.sol :

```
pragma solidity ^0.5.11;  
  
import"./Owned.sol";  
  
contract InheritanceModifier is Owned{
```

Như vậy, là xong. Bây giờ, ta cũng sẽ Deploy như bình thường.