

EVENT, ABI, AND LIBRARY

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

Câu hỏi 1: Khái niệm "Event" trong Solidity được hiểu như thế nào?

Câu hỏi 2: Trình bày các trường hợp sử dụng khác nhau của Events, cho ví dụ minh họa và giải thích:

- Return Value;
- Trigger;
- Datastorage.

Câu hỏi 3: "ABI Array" trong Solidity được dùng để làm gì? cho ví dụ minh họa và giải thích?

Câu hỏi 4: "Gas" là khái niệm được hiểu như thế nào trong Solidity? giải thích?

Câu hỏi 5: Nêu các đặc tính cơ bản của Library?

1. Event

1.1. Khái niệm Event:

Event là một thành viên có thể kế thừa của một hợp đồng. Một event được phát ra, nó lưu trữ các đối số được truyền trong nhật ký giao dịch. Các nhật ký này được lưu trữ trên blockchain và có thể truy cập bằng địa chỉ của hợp đồng cho đến khi hợp đồng có mặt trên blockchain. Một event được tạo ra không thể truy cập được từ bên trong các hợp đồng, thậm chí không phải event đã tạo và phát ra chúng.

Event được gửi đi các tín hiệu mà các hợp đồng thông minh có thể kích hoạt. DApps hoặc bất kỳ thứ gì được kết nối với Ethereum JSON-RPC API, có thể lắng nghe các event này và hành động theo đó. Event có thể được lập chỉ mục, để có thể tìm kiếm lịch sử các event sau này.

Các sự kiện solidity đưa ra một sự trừu tượng về chức năng ghi nhật ký của EVM. Ứng dụng có thể đăng ký và nghe các sự kiện này thông qua giao diện RPC của máy khách Ethereum. Sự kiện là thành viên có thể kế thừa của hợp đồng. Khi bạn gọi chúng, chúng khiến các đối số được lưu trữ trong giao dịch của nhật ký - một cấu trúc dữ liệu đặc biệt trong chuỗi khối. Các nhật ký này được liên kết với địa chỉ của hợp đồng, được tích hợp vào blockchain và ở đó miễn là có thể truy cập được

một khối (mãi mãi cho đến nay, nhưng điều này có thể thay đổi với Serenity). Nhật ký và dữ liệu sự kiện của nó không thể truy cập được từ trong các hợp đồng (thậm chí không phải từ hợp đồng đã tạo ra chúng). Kiểm tra xem nhật ký có thực sự tồn tại bên trong blockchain hay không. Bạn phải cung cấp tiêu đề khối vì hợp đồng có thể chỉ xem 256 block cuối cùng.

Bạn có thể thêm thuộc tính được lập chỉ mục cho tối đa ba tham số để thêm chúng vào cấu trúc dữ liệu đặc biệt được gọi là "Chủ đề" thay vì phần dữ liệu của nhật ký. Nếu bạn sử dụng mảng (bao gồm chuỗi và byte) làm đối số được lập chỉ mục, thay vào đó, hàm băm Keccak-256 của nó được lưu trữ dưới dạng một chủ đề, điều này là do một chủ đề chỉ có thể chứa một từ duy nhất (32 byte).

Tất cả các tham số không có thuộc tính được indexed đều được mã hóa ABI vào phần dữ liệu của nhật ký.

Các chủ đề cho phép bạn tìm kiếm các sự kiện, chẳng hạn như khi lọc một chuỗi khối cho các sự kiện nhất định. Bạn cũng có thể lọc các sự kiện theo địa chỉ của hợp đồng tạo ra sự kiện.

Ví dụ: mã bên dưới sử dụng web3.js subscribe("logs") method để lọc các nhật ký phù hợp với chủ đề có giá trị địa chỉ nhất định:

```
var options = {
  fromBlock: 0,
  address: web3.eth.defaultAccount,
  topics:
["0x0000000000000000000000000000000000000000000000000000000000000000",
0",
,→null, null]
};
web3.eth.subscribe('logs', options, function (error, result) {
  if (!error)
    console.log(result);
})
.on("data", function (log) {
  console.log(log);
})
```

```
.on("changed", function (log) {
});
```

Hàm băm chữ ký của sự kiện là một trong những chủ đề, ngoại trừ trường hợp bạn đã khai báo sự kiện với anonymous. Điều này có nghĩa là không thể lọc các sự kiện ẩn danh cụ thể theo tên, bạn chỉ có thể lọc theo địa chỉ hợp đồng. Ưu điểm của các sự kiện ẩn danh là chúng rẻ hơn để triển khai và gọi.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;
contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
    function deposit(bytes32 _id) public payable {

        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

Việc sử dụng trong ABI JavaScript như sau:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);
var event = clientReceipt.Deposit();
event.watch(function(error, result){
    if (!error)
        console.log(result);
});
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

Kết quả của phần trên trông giống như sau:

```
{
  "returnValues": {
```

```

        "_from": "0x1111...FFFFCCCC",
        "_id": "0x50...sd5adb20",
        "_value": "0x420042"
    },
    "raw": {
        "data": "0x7f...91385",
        "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
    }
}

```

1.2. Các trường hợp của Events:

Return Value:

Cách sử dụng đơn giản nhất của event là chuyển các giá trị trả về từ các hợp đồng đến giao diện người dùng của một ứng dụng

```

contract ExampleContract {
    event ReturnValue(address indexed _from, int256 _value);
    function foo(int256 _value) returns (int256) {
        ReturnValue(msg.sender, _value);
        return _value;
    }
}

// A frontend can then obtain the return value:
var exampleEvent = exampleContract.ReturnValue({_from: web3.eth.coinbase});
exampleEvent.watch(function(err, result) {
    if (err) {
        console.log(err);
        return;
    }
    console.log(result.args._value)
    // check that result.args._from is web3.eth.coinbase then
    // display result.args._value in the UI and call
    // exampleEvent.stopWatching()
})
exampleContract.foo.sendTransaction(2, {from: web3.eth.coinbase})

```

Khi giao dịch gọi foo được khai thác, lệnh gọi lại bên trong watch sẽ được kích hoạt. Điều này cho phép giao diện người dùng nhận các giá trị trả về từ foo một cách hiệu quả.

Trigger: Một emit trong Solidity kích hoạt một sự kiện.

```
event newPurchaseContract(  
    address contractAddress  
);
```

Ta có event newPurchaseContract

Khi ta triển khai, sự kiện trên sẽ được kích hoạt.

```
// deploy a new purchase contract  
function newPurchase()  
    public  
    payable  
    returns(address newContract)  
{  
    Purchase c = (new Purchase).value(msg.value)  
(address(msg.sender));  
    contracts.push(c);  
    lastContractAddress = address(c);  
    emit newPurchaseContract(c);  
    return c;  
}
```

Ta chú ý dòng emit newPurchaseContract (c) ;. Đây là nơi mà event newPurchaseContract () được kích hoạt để xảy ra với địa chỉ hợp đồng c được chuyển vào event newPurchaseContract ()

Thực chất, Return value dùng để trả request về cho website sử dụng và gọi tương tác ra web

```

1 pragma solidity ^0.4.25;
2 contract SimpleStorage {
3
4     uint public storedData;
5     event numberSaved(uint number);
6
7     constructor(uint initVal) public {
8         storedData = initVal;
9     }
10
11     function set(uint x) public{
12         storedData = x;
13         emit numberSaved(storedData);
14     }
15
16     function get() view public returns (uint retVal) {
17         return storedData;
18     }
19
20 }

```

Tại đây, ta sử dụng gọi hàm

```
mySmartContract.events.numberSaved();
```

Hàm này sẽ gọi giá trị emit chúng ta đã tạo ra tại hàm set cũng như là giá trị thực của storedData

Trigger :

Trigger thực chất là 1 tương tác event từ emit của Contract

```

1 pragma solidity ^0.4.25;
2 contract Coin {
3     event Sent(address from, address to, uint amount);
4
5
6     function send(address receiver, uint amount) public {
7         emit Sent(msg.sender, receiver, amount);
8     }
9 }

```

Tương tác emit qua lại gửi 1 hoặc nhiều các event, đương nhiên thông số của event cũng phải được đảm bảo theo yêu cầu của function và emit của event đó phải thực sự tương tác với nhau qua các function trong Smart Contract

Datastorage:

Datastorage: sử dụng sự kiện để viết nhật ký trong blockchain như một hình thức lưu trữ.

Dùng để lưu trữ dữ liệu vĩnh viễn, còn được gọi là lưu trữ thứ cấp. Lưu lại, đây là bộ nhớ chung có sẵn cho tất cả các chức năng trong hợp đồng. Bộ nhớ này là bộ nhớ vĩnh viễn mà Ethereum lưu trữ trên mọi nút trong môi trường của nó.

Datastorage sẽ lưu trữ emit của chúng ta vào bộ nhớ Storage để tương tác qua lại với cấp độ storage với các biến giá trị như của String hoặc Uint bất cao

Khai báo nó tương tự như khai báo 1 struct hoặc 1 biến thực thể kể cả là mapping hoặc mảng

```

var options = {
  fromBlock: 0,
  address: web3.eth.defaultAccount,
  topics: ["0x0000000000000000000000000000000000000000000000000000000000000000",
  null, null]
};
web3.eth.subscribe('logs', options, function (error, result) {
  if (!error)
    console.log(result);
})
.on("data", function (log) {
  console.log(log);
})
.on("changed", function (log) {
});

```

Datastorage sẽ lưu trữ emit của chúng ta vào bộ nhớ Storage để tương tác qua lại với cấp độ storage với các biến giá trị như của String hoặc Uint bất

```
contract Coin {  
    struct Event{  
        bytes32 name;  
        uint time;  
    }  
  
    event[] public events;  
  
    mapping(uint=>event) public events;  
}
```

cao

Khai báo nó tương tự như khai báo 1 struct hoặc 1 biến thực thể kể cả là mapping hoặc mảng (Cả 2 đều tương tự nhau).

Datastorage Hợp đồng theo nghĩa Solidity là một tập hợp mã (functions) và dữ liệu (trạng thái của nó) nằm tại một địa chỉ cụ thể trên chuỗi khối Ethereum. Dòng Uint storedata; khai báo một biến trạng thái được gọi là storeData kiểu uint (số nguyên không dấu của 256 bit). Bạn có thể coi nó như một vùng duy nhất trong cơ sở dữ liệu mà bạn có thể truy vấn và thay đổi bằng cách gọi các hàm của mã quản lý cơ sở dữ liệu.

```
contract SimpleStorage {  
    uint storedData;  
  
    function set(uint x) public {  
        storedData = x;  
    }  
  
    function get() public view returns (uint) {  
        return storedData;  
    }  
}
```


Hợp đồng xác định các chức năng get và set có thể được sử dụng để sửa đổi hoặc truy xuất giá trị của biến.

1.3. Ví dụ ứng dụng Event:

Event: là cách thuận tiện để ghi lại một điều gì đó đã xảy ra trong hợp đồng. Các sự kiện được phát ra (emit) được lưu lại trong blockchain cùng với dữ liệu khác của hợp đồng. Đây là một ví dụ, chúng ta sử dụng sự kiện để ghi lại lịch sử quyên góp của Hội từ thiện.

```
contract Charity {
    // define event
    event LogDonate(uint _amount);

    mapping(address => uint) balances;

    function donate() payable public {
        balances[msg.sender] += msg.value;
        // emit event
        emit LogDonate(msg.value);
    }
}

contract Game {
    function buyCoins() payable public {
        // 5% goes to charity
        charity.donate.value(msg.value / 20)();
    }
}
```

Tất cả các giao dịch gọi hàm donate của hợp đồng Charity, dù trực tiếp hay không, sẽ hiển thị trong danh sách sự kiện của hợp đồng đó cùng với số tiền quyên góp.

Một sự kiện ví dụ từ hợp đồng wallet là:

```
event Deposit(address from, uint value);
```

Ứng dụng (dapp, ứng dụng web, ứng dụng khác) quan tâm đến việc gửi tiền vào

hợp đồng ví sẽ lắng nghe sự kiện này. Ứng dụng sẽ kết nối với nút Ethereum qua JSON-RPC và theo dõi (chờ) sự kiện xảy ra hoặc đọc tất cả các sự kiện trong quá khứ để đồng bộ hóa trạng thái nội bộ của ứng dụng với chuỗi khối Ethereum.

```
pragma solidity ^0.5.11;

contract EventSimple{
    mapping(address => uint) public tokenBalance;

    event TokensSend(address _from, address _to, uint _amount);

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

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

        emit TokensSend(msg.sender, _to, _amount);

        return true;
    }
}
```

- Tất cả các giao dịch khi gọi hàm sendToken() sẽ hiển thị trong danh sách sự kiện của hợp đồng: địa chỉ tài khoản gửi, địa chỉ tài khoản nhận, và số tiền chuyển.

2. ABI AND GAS

2.1. ABI

Giao diện nhị phân ứng dụng hợp đồng (ABI) là cách tiêu chuẩn để tương tác với các hợp đồng trong hệ sinh thái Ethereum, cả từ bên ngoài chuỗi khối và tương tác từ hợp đồng với hợp đồng. Dữ liệu được mã hóa theo loại của nó, như được mô tả trong thông số kỹ thuật này. Mã hóa không tự mô tả và do đó yêu cầu một lược đồ để giải mã.

ABI là từ viết tắt của Application Binary Interface. API dùng để xác định hàm nào trong hợp đồng thông minh được gọi và trả về dữ liệu theo định dạng mong đợi.

```
const path = require('path');
const fs = require('fs');
const solc = require('solc');

const helloPath = path.resolve(__dirname, 'contracts', 'hello.sol');
const source = fs.readFileSync(helloPath, 'utf8');
const compiledSource = solc.compile(source, 1).contracts[':Hello'];

module.exports = compiledSource;

console.log(compiledSource);
```

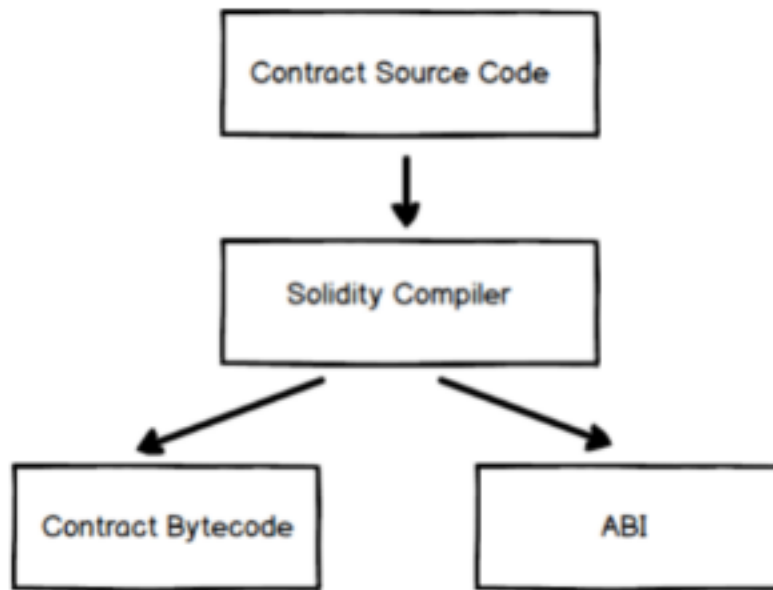
Biến **helloPath** chứa đường dẫn đến file hello.sol

Biến **source** chứa kết quả đọc của việc đọc file có đường dẫn là helloPath, tức là file hello.sol đó.

Biến **compiledSource** dùng để chứa nội dung biên dịch chứa trong biến source. Ta sử dụng **sol.compile()** để biên dịch. **[':Hello']** là tên của contract ta khai báo trong file hello.sol.

Giờ thì chúng ta có thể xóa dòng **console.log(compiledSource);** không cần thiết đi rồi đây.

Quá trình compile code sẽ tạo 2 cái mới: ABI và Bytecode:



- ABI Array bao gồm các Functions/ Parameters/ Return values của contract. Nó cũng chính là Json file.
- Application Binary Interface (ABI) là cách tiêu chuẩn để tương tác với các hợp đồng trong hệ sinh thái Ethereum, cả từ bên ngoài chuỗi khối và tương tác từ hợp đồng với hợp đồng.
- Trong EVM, mã đã biên dịch được lưu trữ dưới dạng dữ liệu nhị phân và các giao diện con người có thể đọc được sẽ biến mất và các tương tác hợp đồng thông minh phải được dịch sang định dạng nhị phân mà EVM có thể hiểu được.
- ABI xác định các phương thức và cấu trúc mà bạn có thể đơn giản sử dụng để tương tác với hợp đồng nhị phân đó (giống như API), chỉ ở cấp độ thấp hơn. ABI chỉ ra cho người gọi thông tin cần thiết (chữ ký hàm và khai báo biến) để mã hóa sao cho nó được hiểu bởi lệnh gọi Máy ảo thành mã byte (hợp đồng).
- ABI là giao diện giữa hai mô-đun chương trình, một trong số đó chủ yếu ở cấp mã máy. Giao diện là phương thức mặc định để mã hóa / giải mã dữ liệu vào hoặc ra khỏi mã máy. Trong Ethereum, về cơ bản, đó là cách bạn mã hóa một ngôn ngữ để có các lệnh gọi hợp đồng tới EVM hoặc cách đọc dữ liệu từ các giao dịch.

- Mã hóa ABI trong hầu hết các trường hợp được tự động hóa bởi các công cụ là một phần của trình biên dịch như REMIX hoặc ví có thể tương tác với blockchain. Bộ mã hóa ABI yêu cầu mô tả về giao diện của hợp đồng như tên hàm và các tham số thường được cung cấp dưới dạng JSON.

- Ví dụ về một Json output như dưới:

```
{
  "astId": 2,
  "contract": "fileA:A",
  "label": "x",
  "offset": 0,
  "slot": "0",
  "type": "t_uint256"
}
```

Ví dụ trên là cách bố trí lưu trữ của hợp đồng A { uint x; } từ tệp đơn vị nguồn A và

- astId là id của nút AST trong khai báo của biến trạng thái
- contract là tên của hợp đồng bao gồm tiền tố đường dẫn của nó
- label là tên của biến trạng thái.
- offset là độ lệch tính bằng byte trong vùng lưu trữ theo mã hóa
- slot là vùng lưu trữ nơi biến trạng thái cư trú hoặc bắt đầu. Con số này có thể rất lớn và do đó giá trị JSON của nó được biểu diễn dưới dạng một chuỗi.
- type là một mã định danh được sử dụng làm chìa khóa cho thông tin loại của biến (được mô tả trong phần sau)

2.2. GAS

Gas là một đơn vị đo lường công việc tính toán của các giao dịch hoặc hợp đồng thông minh trong mạng Ethereum. Hệ thống này tương tự như việc sử dụng kilowatt (kW) để đo điện trong nhà bạn; lượng điện bạn sử dụng không được tính bằng đô la mà thay vào đó thông qua kWh hoặc Kilowatts mỗi giờ.

Hiểu đơn giản, hệ thống này diễn ra tương tự như việc KW trở thành đơn vị để đo lường lượng điện năng tiêu thụ vậy. Nếu như lượng điện mà bạn sử dụng không trực tiếp được tính bằng đô la mà được thông qua kWh hoặc Kilowatts/giờ từ đó quy đổi ra lượng tiền cần trả. Thì đối với mạng lưới Ethereum cũng vậy, mọi thứ được thông qua trung gian là Gas để quy đổi thành coin.

Overview	
Comments	
Transaction Information	
TxHash:	0x08b36b754691aa6f0608cb983bd23f2eec045a40f6ea41165dd48e8046af1514
TxReceipt Status:	Success
Block Height:	5082447 (23 block confirmations)
TimeStamp:	4 mins ago (Feb-13-2018 10:58:24 AM +UTC)
From:	0xdc7693bd416f4627871c82b4fc030e42238921b3
To:	0x27bd240886d755e1d273a21d2f00d8598c1c5724
Value:	1.01682595274441134 Ether (\$846.17)
Gas Limit:	21000
Gas Used By Txn:	21000
Gas Price:	0.000000008 Ether (8 Gwei)
Actual Tx Cost/Fee:	0.000168 Ether (\$0.14)
Cumulative Gas Used:	866792
Nonce:	0

Nhìn qua hình ảnh bên trên là Etherscan, các bạn có thể thấy tất cả những thông tin liên quan đến Gas. Và mình sẽ giải thích ý nghĩa của những thông số phía trên như sau.

-+- Gas Limit: Đây là lượng Gas tối đa mà người dùng sẽ trả cho giao dịch này. Số tiền mặc định cho một lần chuyển ETH tiêu chuẩn là 21.000 Gas.

-+-Gas Used by Txn: Đây là lượng Gas thực tế được sử dụng để thực hiện giao dịch. Vì nó là một giao dịch tiêu chuẩn nên Gas sử dụng là tiêu chuẩn 21.000 Gas.

-+-Gas Price: Đây là số lượng ETH trả cho mỗi đơn vị Gas. Người dùng đã chọn trả 8 Gwei cho mỗi đơn vị Gas, đây được coi là giao dịch ưu tiên cao và nó sẽ được thực hiện rất nhanh.

-+-Actual Tx Cost Fee: Đây sẽ là phí giao dịch thực tế mà người dùng sẽ trả cho giao dịch bằng giá trị Ether. Với giao dịch này thì người dùng chỉ cần trả 14 cent (0,14 USD) để chuyển ETH chỉ trong vòng 2 phút.

Điều quan trọng là phải hiểu rằng các loại giao dịch khác nhau đòi hỏi một lượng khí khác nhau để hoàn thành. Chẳng hạn, một giao dịch đơn giản gửi ETH từ nơi này đến nơi khác tốn 21.000 Gas trong khi gửi token ICO từ ví MyEtherWallet (Links to an external site.) (MEW) của bạn tốn nhiều chi phí hơn do mức độ tính toán cao hơn đã kết thúc. Dưới đây, một hướng dẫn về cách mở ví MEW, đây là ví hỗ trợ tiền ETH và ERC-20.

Gas Limit

Gas Limit liên quan đến lượng gas tối đa mà bạn sẵn sàng chi cho một giao dịch cụ thể. Gas Limit cao hơn có nghĩa là phải thực hiện nhiều công việc tính toán hơn để thực hiện hợp đồng thông minh. Việc chuyển ETH tiêu chuẩn yêu cầu giới hạn khí là 21.000 đơn vị gas.

Các lệnh bạn muốn thực hiện càng phức tạp, bạn càng phải trả nhiều gas hơn. Bạn có thể thấy điều này khi hoạt động khi tham gia vào ICO yêu cầu bạn gửi ETH vào hợp đồng thông minh của nó hoặc khi bạn muốn rút tiền ICO của mình để trao đổi; phí chuyển nhượng cao hơn nhiều so với Gas Limit 21.000 mặc định. Điều này là do các hợp đồng thông minh của ICO sở hữu các mã phức tạp hơn nhiều và yêu cầu tính toán nhiều hơn so với chuyển khoản ETH đơn giản.

Gas Limit hoạt động như một cơ chế an toàn để bảo vệ bạn khỏi cạn kiệt tiền do mã lỗi hoặc lỗi trong hợp đồng thông minh.

Chi phí thực hiện và sử dụng tài nguyên được xác định trước trong Ethereum về đơn vị gas. Đây còn được gọi là gas cost. Ngoài ra còn có giá gas có thể được điều chỉnh ở mức giá thấp hơn khi giá Ether tăng và giá cao hơn khi giá Ether giảm.

Ví dụ: để gọi một hàm trong hợp đồng sửa đổi một chuỗi sẽ tốn gas, điều này đã được xác định trước và người dùng nên thanh toán bằng gas để đảm bảo giao dịch này được thực hiện suôn sẻ.

Sau khi tạo, mỗi giao dịch được tính một lượng gas nhất định, mục đích của nó là giới hạn số lượng công việc cần thiết để thực hiện giao dịch và thanh toán cho việc thực hiện này. Trong khi EVM thực hiện giao dịch, gas dần dần cạn kiệt theo các quy tắc cụ thể.

Giá gas là giá trị do người tạo giao dịch đặt, người này phải trả trước từ tài khoản gửi. Nếu một số khí còn lại sau khi thực hiện, nó sẽ được hoàn lại theo cách tương tự. $\text{gas_price} * \text{gas}$

Nếu gas được sử dụng hết tại bất kỳ thời điểm nào (tức là giá trị âm), một ngoại lệ hết gas sẽ được kích hoạt, điều này sẽ hoàn nguyên tất cả các sửa đổi được thực hiện về trạng thái trong khung cuộc gọi hiện tại.

Trong Solidity, người dùng của bạn phải trả tiền mỗi khi họ thực hiện một chức năng trên DApp bằng một loại tiền gọi là gas . Người dùng mua gas với Ether (tiền tệ trên Ethereum), vì vậy người dùng của bạn phải chi tiêu ETH để thực hiện các chức năng trên DApp của bạn.

Cần bao nhiêu gas để thực hiện một chức năng phụ thuộc vào sự phức tạp của logic chức năng đó. Mỗi hoạt động cá nhân có chi phí gas dựa trên khoảng bao nhiêu tài nguyên máy tính sẽ được yêu cầu để thực hiện các hoạt động đó. Tổng chi phí gas của chức năng của bạn là tổng chi phí gas của tất cả các hoạt động cá nhân.

Bởi vì chạy chức năng chi phí tiền thật cho người dùng của bạn, tối ưu hóa mã là quan trọng hơn nhiều trong Ethereum hơn trong các ngôn ngữ lập trình khác. Nếu mã của bạn là cầu thả, người dùng của bạn sẽ phải trả phí bảo hiểm để thực hiện các chức năng của bạn – và điều này có thể làm tăng hàng triệu đô la phí không cần thiết qua hàng ngàn người dùng.

Ethereum là tên của một loại Blockchain, còn Ether (ETH) là nhiên liệu cho mạng đó. Khi bạn gửi ETH, tạo hợp đồng thông minh, biên dịch lệnh... hay tất cả điều gì mà tương tác với Blockchain, bạn đều phải trả phí. Khoản phí đó được tính bằng Gas/Gas Price và Gwei.

Trong Solidity, người dùng của bạn phải trả tiền mỗi khi họ thực hiện một chức năng trên DApp bằng một loại tiền gọi là gas . Người dùng mua gas với Ether (tiền tệ trên Ethereum), vì vậy người dùng của bạn phải chi tiêu ETH để thực hiện các chức năng trên DApp của bạn.

+ Gas Limit: được gọi là giới hạn năng lượng vì đó là số tiền tối đa của đơn vị Gas mà bạn sẵn sàng chi cho giao dịch. Điều này tránh được tình huống có một lỗi ở nơi nào đó trong hợp đồng, và bạn gửi 1 ETH mà không có nơi nhận. Nếu bạn không đủ Gas Limit thì khi gửi giao dịch sẽ gặp lỗi “Out of Gas” và giao dịch không được thực hiện.

+ Gas Price: Nếu bạn muốn tiết kiệm hơn cho một giao dịch, hãy giảm số tiền bạn trả thông qua Gas Price. Con số Gas Price này quyết định tốc độ giao dịch của bạn diễn ra nhanh hay chậm.

+ Gwei là tốc độ chuyên, Gwei cao giúp giao dịch được ưu tiên xử lý trước.

3. LIBRARY

Library tương tự như hợp đồng, nhưng mục đích của chúng là chúng chỉ được triển khai một lần tại một địa chỉ cụ thể và mã được sử dụng lại bằng cách sử dụng tính năng DELEGATECALL (CALLCODE cho đến khi Homestead) của EVM. Điều này có nghĩa là nếu các hàm thư viện được gọi, mã của chúng được thực thi trong ngữ cảnh của hợp đồng gọi, tức là điều này trở đến lệnh gọi hợp đồng, và đặc biệt là có thể truy cập dung lượng lưu trữ từ calling.

Như một thư viện là một phần biệt lập của mã nguồn, nó chỉ có thể truy cập các biến trạng thái của hợp đồng gọi nếu chúng được cung cấp rõ ràng (nó sẽ không có cách đặt tên cho chúng, nếu không). Các hàm Library chỉ có thể được gọi trực tiếp (tức là không sử dụng DELEGATECALL) nếu chúng không sửa đổi trạng thái (tức là nếu chúng là dạng xem hoặc các hàm thuần túy), bởi vì các thư viện được coi là không có trạng thái.

Đặc biệt, không thể phá hủy một thư viện

So với hợp đồng, các library bị hạn chế theo những cách sau:

- Chúng không thể có các biến trạng thái.
- Chúng không thể kế thừa cũng như không được thừa kế
- Chúng không thể nhận Ether
- Chúng không thể bị phá hủy.

Ví dụ

Ta có ví dụ sau về sử dụng library trong solidity:

```
pragma solidity ^0.5.0;

library libraryExample {

    struct Constants {
        uint Pi;
        uint EulerNb;
        uint PythagoraConst;
        uint TheodorusConst;
    }

}
```

Như ví dụ trên, ta có thể biết được cách define một library, đó là library ten_library; và như các đặc tính của nó, ta không thể phá hủy hay nhận ether,..

Library trong Solidity là một loại hợp đồng thông minh khác có chứa mã có thể sử dụng lại. Sau khi được triển khai trên blockchain (chỉ một lần), nó được gán một địa chỉ cụ thể và các thuộc tính / phương thức của nó có thể được sử dụng lại nhiều lần bởi các hợp đồng khác trong mạng Ethereum..

library: Deploy 1 lần và được sử dụng bởi các hợp đồng khác thông qua DELEGATECALL

Hạn chế: Chúng không có bất kỳ bộ nhớ nào (vì vậy không thể có các biến trạng thái không phải là hằng số)

Họ không thể giữ ether (vì vậy không thể có chức năng fallback)

Không cho phép các chức năng payable (vì chúng không thể chứa ether)

Không thể kế thừa cũng như không được kế thừa

Không thể bị phá hủy (không có chức năng selfdestruct())

Bạn xác định hợp đồng thư viện với từ khóa library thay vì contract từ khóa truyền thống được sử dụng cho các hợp đồng thông minh tiêu chuẩn. Chỉ cần khai báo library dưới pragma solidity câu lệnh (phiên bản trình biên dịch). Xem ví dụ mã của chúng tôi bên dưới.

```
pragma solidity ^0.5.0;

library libraryName {

    // struct, enum or constant variable declaration

    // function definition with body
}
```

Như chúng ta đã thấy, các hợp đồng thư viện không có dung lượng lưu trữ. Do đó, chúng không thể giữ các biến trạng thái (các biến trạng thái không phải là hằng số). Tuy nhiên, các thư viện vẫn có thể triển khai một số kiểu dữ liệu:

struct và enum: đây là các biến do người dùng xác định.

Bất kỳ biến nào khác được định nghĩa là constant(không thay đổi), vì các biến hằng số được lưu trữ trong bytecode của hợp đồng, không phải trong bộ nhớ.

Đặc tính của Library:

library trong Solidity là một loại hợp đồng thông minh khác có chứa mã có thể sử dụng lại. Sau khi được triển khai trên blockchain (chỉ một lần), nó được gán một địa chỉ cụ thể và các thuộc tính / phương thức của nó có thể được sử dụng lại nhiều lần bởi các hợp đồng khác trong mạng Ethereum.

Chúng cho phép phát triển theo cách mô-đun hơn. Đôi khi, sẽ hữu ích nếu coi thư viện như một singleton trong EVM, một đoạn mã có thể được gọi từ bất kỳ hợp đồng nào mà không cần phải triển khai lại.

Những lợi ích:

Tuy nhiên, libraries trong Solidity không chỉ giới hạn ở khả năng tái sử dụng. Dưới đây là một số ưu điểm của chúng:

Khả năng sử dụng: Các chức năng trong thư viện có thể được sử dụng bởi nhiều hợp đồng. Nếu bạn có nhiều hợp đồng có một số mã chung, thì bạn có thể triển khai mã chung đó dưới dạng thư viện.

Kinh tế: Triển khai mã chung làm thư viện sẽ tiết kiệm gas vì gas cũng phụ thuộc vào quy mô của hợp đồng. Sử dụng hợp đồng cơ sở thay vì thư viện để tách mã chung sẽ không tiết kiệm xăng vì trong Solidity, tính năng kế thừa hoạt động bằng cách sao chép mã.

Tiện ích bổ sung tốt: Các thư viện solidity có thể được sử dụng để thêm các hàm thành viên vào các kiểu dữ liệu. Ví dụ, hãy nghĩ về các thư viện như thư viện chuẩn hoặc gói mà bạn có thể sử dụng trong các ngôn ngữ lập trình khác để thực hiện các phép toán phức tạp trên các con số.

Hạn chế:

Các thư viện trong Solidity được coi là không có trạng thái và do đó có các hạn chế sau

Chúng không có bất kỳ bộ nhớ nào (vì vậy không thể có các biến trạng thái không phải là hằng số)

Họ không thể giữ ete (vì vậy không thể có chức năng dự phòng)

Không cho phép các chức năng phải trả (vì chúng không thể chứa ete)

Không thể kế thừa cũng như không được kế thừa

Không thể bị phá hủy (không có selfdestruct()chức năng nào kể từ phiên bản 0.4.20)

Library tương tự như Contract nhưng chủ yếu nhằm mục đích sử dụng lại. Library chứa các hàm mà các hợp đồng khác có thể gọi. Solidity có những hạn chế nhất định đối với việc sử dụng Library . Sau đây là các đặc điểm chính của Solidity Library .

Các chức năng Library có thể được gọi trực tiếp nếu chúng không sửa đổi trạng thái. Điều đó có nghĩa là các hàm thuần túy hoặc chỉ xem có thể được gọi từ bên ngoài Library .

Library không thể bị phá hủy vì nó được cho là không có trạng thái.

Library không thể có các biến trạng thái.

Library không thể kế thừa bất kỳ phần tử nào.

Library không thể được kế thừa.

Ví dụ về Library:

```
pragma solidity ^0.5.0;

library Search {
    function indexOf(uint[] storage self, uint value) public view returns (uint)
        for (uint i = 0; i < self.length; i++) if (self[i] == value) return i;
        return uint(-1);
}
```

Như ví dụ trên, ta biết được cách tạo 1 Library bằng cú pháp `library library_name { }`, ở đây ta tạo library thay vì tạo contract như trước đây từng làm. Phía trong Library ta hoàn toàn có thể tạo các function như trong 1 Contract.

LAB 14

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

Tiếp theo, ta tiến hành khai báo phiên bản của Solidity đang ở phiên bản bao nhiêu. Ta sử dụng từ khóa “pragma”:

```
pragma solidity ^0.5.11;
```

Bây giờ, ta tạo 1 contract với tên EventSimple.

Tiếp đến, ta tạo 1 biểu thức mapping với loại từ khóa là address và giá trị là uint. Và biến của mapping là tokenBalance.

Ta tạo 1 hàm khởi tạo constructor : với đầu vào là số tokenBalance của địa chỉ nào thực hiện Deploy sẽ có giá trị = 100.

```
contract EventSimple{
    mapping(address => uint) public tokenBalance;

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

Để thực hiện gửi token ta tạo 1 function sendToken:

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

    emit TokensSent(msg.sender, _to, _amount);

    return true;
}
```

Function này có thể hiểu như sau: khi ta thực hiện gửi số token đến một địa chỉ nào đó. Hàm này sẽ xét các điều kiện. Như ta thấy, yêu cầu require đưa ra là số tokenBalance của địa chỉ thực hiện chuyển phải \geq số tiền cần chuyển.

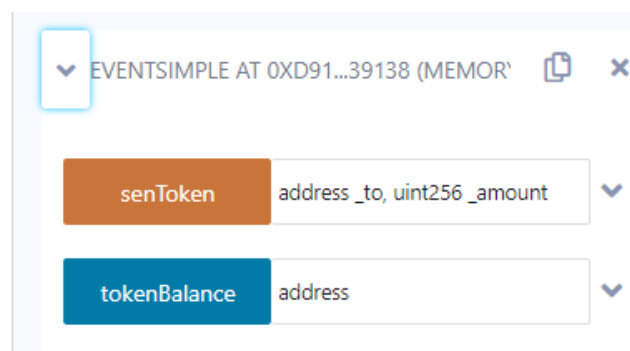
Hàm require nên được sử dụng để đảm bảo đáp ứng các điều kiện hợp lệ, chẳng hạn như đầu vào, hoặc các biến trạng thái hợp đồng hoặc để xác thực các giá trị trả về từ các lệnh gọi đến các hợp đồng bên ngoài.

Và 2 hàm assert được đưa ra để kiểm tra các điều kiện, các lỗi nội bộ và kiểm tra các bất biến.

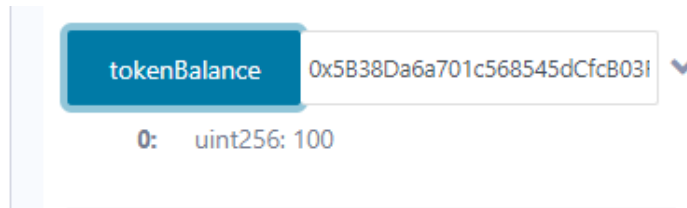
Khi các điều kiện đã được đáp ứng: số token sẽ được chuyển tới địa chỉ chúng ta muốn chuyển là _to.

Và như ta thấy, giá trị trả về của function là kiểu bool nên vì thế khi hàm thực hiện đúng hàm sẽ trả về true.

Bây giờ, ta thực hiện Deploy:



Để xem giá trị đầu vào của địa chỉ Delpoy ta copy địa chỉ và dán vào trường tokenBalace sau đó, ta nhấn vào nút tokenBalance:

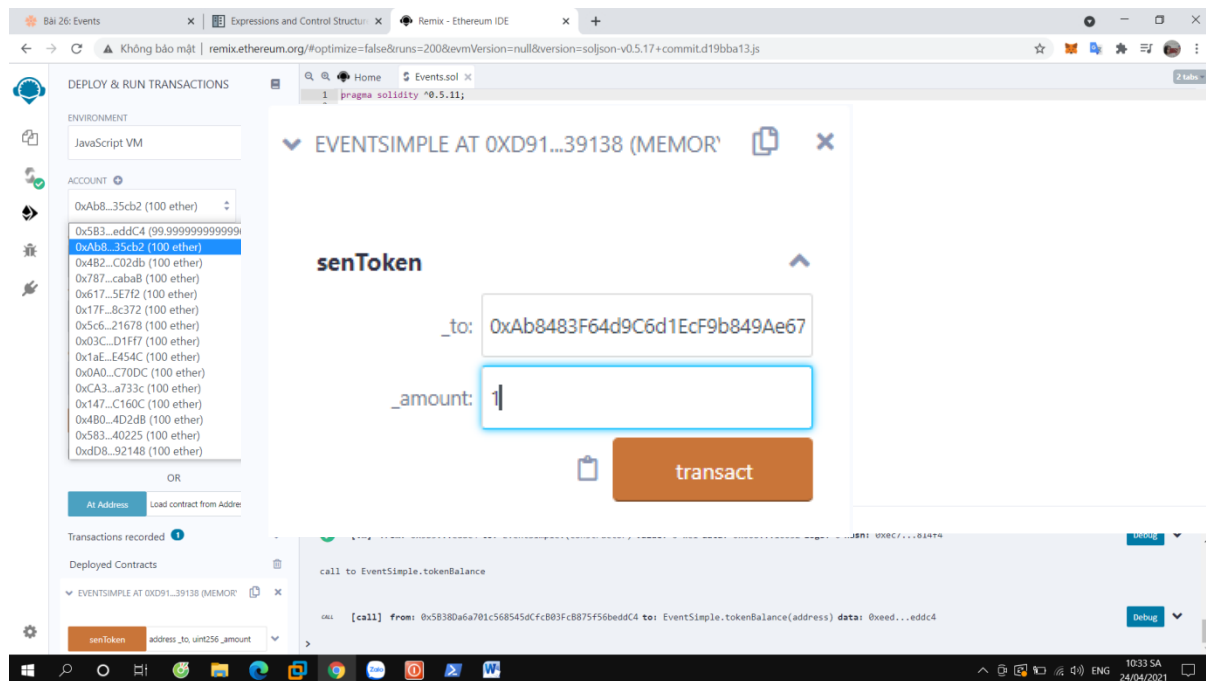


tokenBalance 0x5B38Da6a701c568545dCfc8031

0: uint256: 100

Tiếp theo, ta copy 1 địa chỉ bất kì và dán vào trường senToken cùng với số _amount cần chuyển.

Mình copy địa chỉ thứ 2 sau đó chuyển lại địa chỉ ban đầu nha:



ENVIRONMENT

JavaScript VM

ACCOUNT

0xAb8...35cb2 (100 ether)

0x5B3...eddc4 (99.99999999999999)

0x482...C02db (100 ether)

0x787...caba8 (100 ether)

0x617...5E7f2 (100 ether)

0x17f...8c372 (100 ether)

0x5c6...21678 (100 ether)

0x03c...D1f77 (100 ether)

0x1aE...E454C (100 ether)

0x0A0...C70DC (100 ether)

0xCA3...a733c (100 ether)

0x147...C160C (100 ether)

0x480...4D2db (100 ether)

0x583...40225 (100 ether)

0xd08...92148 (100 ether)

OR

At Address Load contract from Address

Transactions recorded

Deployed Contracts

EVENTSIMPLE AT 0xD91...39138 (MEMOR)

senToken address_to uint256 _amount

senToken

_to: 0xAb8483F64d9C6d1EcF9b849Ae67

_amount: 1

transact

call to EventSimple.tokenBalance

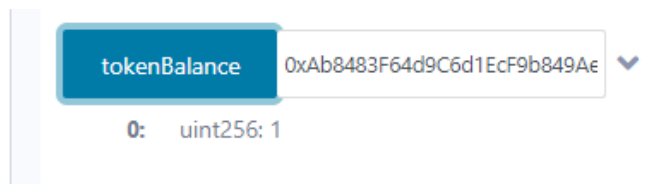
[call] from: 0x5B38Da6a701c568545dCfc8031 to: EventSimple.tokenBalance(address) data: 0x5B38Da6a701c568545dCfc8031

Bây giờ, ta nhấn vào **transact** để bắt đầu giao dịch. Khi giao dịch thành công, ta xem các thông số ở thông báo của Remix.

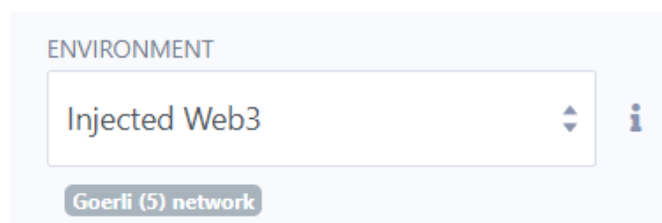


Ta chú ý thông số **decoded output**: giá trị trả về **true**.

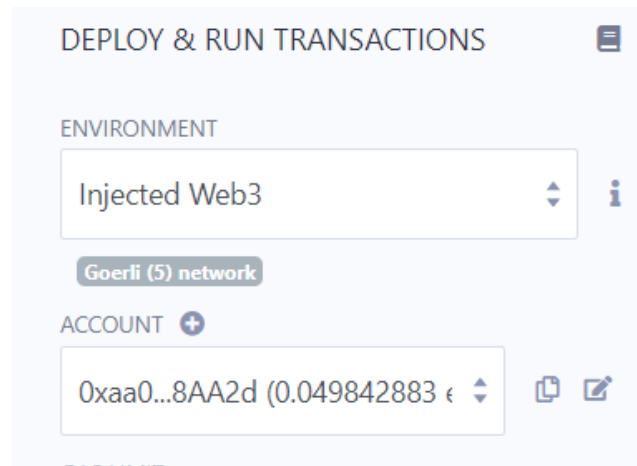
Khi đó, ta kiểm tra số **tokenBalance** của địa chỉ vừa chuyển tới.



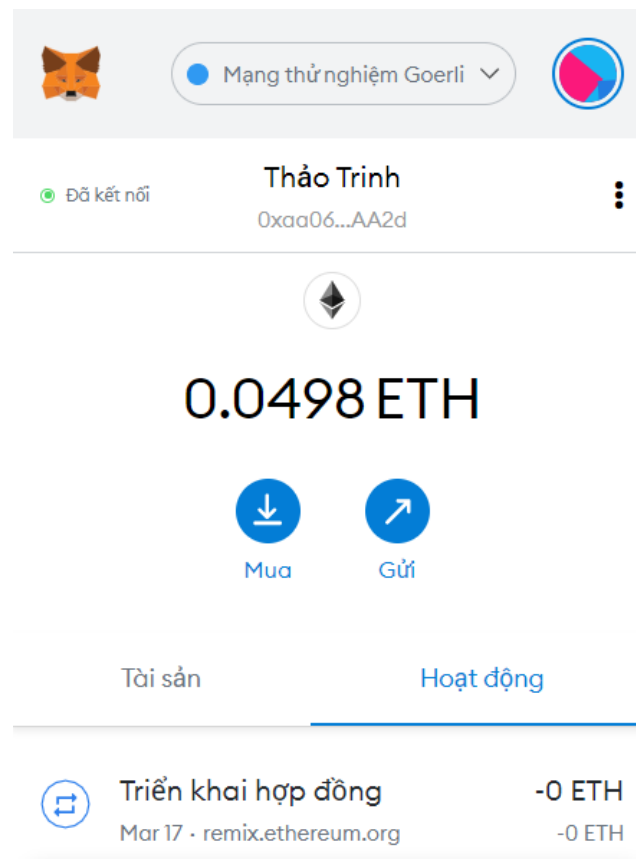
Bây giờ, ta chuyển đổi môi trường từ **JavaScript** sang **Injected Web3**. Khi đó **MetaMask** sẽ hiển thị. Ta thực hiện trên mạng thử nghiệm **Goerli** nhé:



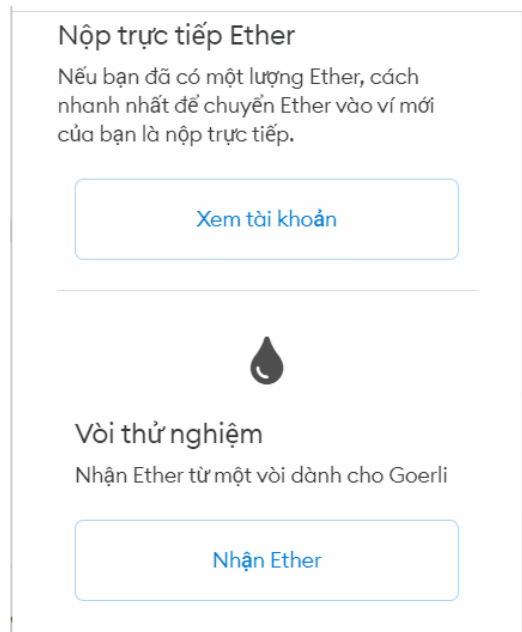
Khi đã kết nối thành công ta sẽ thấy ở ô Account sẽ là tài khoản của MetaMask:



Bây giờ, ta thử lấy giá trị ether xem sao nha:



Ta nhấn vào mua, sau đó ta nhấn vào Nhận Ether:



Tiếp theo, ta copy địa chỉ của tài khoản và dán vào ô này để nhận 0.05ether.



Chúng ta chờ giao dịch thành công và khi đó ta kiểm tra lại số ether tài khoản:



Bây giờ, ta quay trở lại Remix và thực hiện Deploy:

Mạng thử nghiệm Goerli

Thảo Trình → Hợp đồng mới

http://remix.ethereum.org

TRIỂN KHAI HỢP ĐỒNG

0

DETAILS DATA

GAS FEE 0.000914
Không có sẵn tỷ lệ quy đổi nào

Giá gas (GWEI) 3,472754404 Giới hạn gas 263071

AMOUNT + GAS FEE

TOTAL 0.000914
Không có sẵn tỷ lệ quy đổi nào

Từ chối Xác nhận

Lúc này, MetaMask hiển thị và chúng ta xác nhận.

Tiếp theo, ta chuyển ether, ta copy địa chỉ tài khoản dán vào trường senToken. Sau đó, ta bắt đầu giao dịch.

EVENTSIMPLE AT 0XF5E...EDD44 (BLOCKCH)

senToken

_to: 0xaa06008A0D3862CF5811B003905

_amount: 1

transact

Khi nhấn vào transact, MetaMask sẽ hiển thị để chúng ta xác nhận. Và thông báo giao dịch thành công hiển thị:



[block:4673307 txIndex:9] from: 0xaa0...8AA2d to: EventSimple.sendToken(address,uint256) 0xF5E...eDD44 value: 0 wei data: 0x458...00001 logs: 0 hash: 0xe33...9c50a

status: true Transaction mined and execution succeed

transaction hash: 0xe33d2bd36a61b2d22db58726f6247c3d3d6fd58f44b772309de75af6d8b9c50a

from: 0xaa06008A0D3862CF5811B0039056AA8Bb28AA2d

to: EventSimple.sendToken(address,uint256) 0xF5E706915E50FD098f94469076ff6d3d61eDD44

gas: 28045 gas

transaction cost: 28045 gas

hash: 0xe33d2bd36a61b2d22db58726f6247c3d3d6fd58f44b772309de75af6d8b9c50a

input: 0x458...00001

decoded input: { "address _to": "0xaa06008A0D3862CF5811B0039056AA8Bb28AA2d", "uint256 _amount": { "type": "BigNumber", "hex": "0x01" } }

decoded output: -

logs: []

value: 0 wei

Khi giao dịch với MetaMask cần có cách giám sát hoạt động của hợp đồng sau khi nó được triển khai. Một cách để thực hiện điều này là xem xét tất cả các giao dịch của hợp đồng, tuy nhiên điều đó là chưa đủ, vì các lời gọi giữa các hợp đồng không được ghi lại trên blockchain.

Để làm được việc đó ta thực hiện sử dụng event để ghi lại các sự kiện, ta update code như sau:

```
event TokensSent(address _from, address _to, uint _amount);

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

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

    emit TokensSent(msg.sender, _to, _amount);

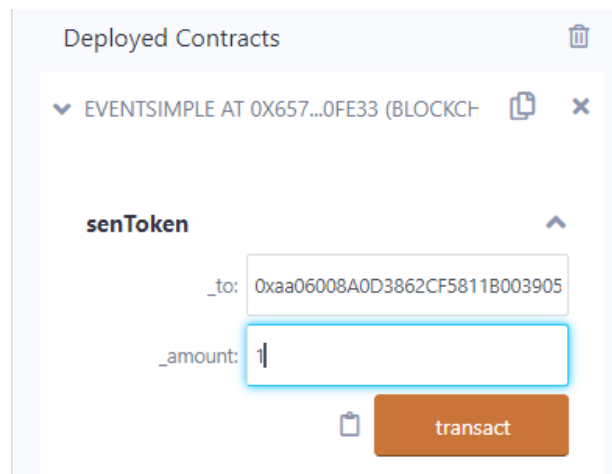
    return true;
}
```

Event là cách thuận tiện để ghi lại một điều gì đó đã xảy ra trong hợp đồng. Các sự kiện được phát ra (emit) được lưu lại trong blockchain cùng với dữ liệu khác của

hợp đồng. Như vậy, chúng ta sử dụng event để ghi lại lịch sử địa chỉ gửi và nhận cùng với số `_amount` cần chuyển.

Bây giờ, ta thực hiện Deploy và chuyển lại 1 lần nữa để xem mọi thứ đã được lưu lại hay không nha:

Ta cũng thực hiện tương tự như lúc này:



Ta nhấn transact để thực thi giao dịch. Và xác nhận từ MetaMask, tiếp đó, ta xem các thông số:



Nếu như lúc này ta thấy ở thông số logs không hiển thị bất kì thứ gì ngoài dấu “[]”.

Thì bây giờ sau khi ta thêm event vào thì mọi thứ giao dịch đã được ghi lại ở thông số nhật ký logs. Như vậy, quá dễ dàng để chúng ta có thể xem lại những điều đã giao dịch.

LAB 15:

Trong mọi giao dịch, ta đều có trình gỡ lỗi hiển thị trạng thái của hợp đồng trong khi thực hiện giao dịch.

Nó có thể được sử dụng cho các giao dịch được tạo trên Remix hoặc bằng cách cung cấp địa chỉ của giao dịch. Sau đó giả định rằng bạn có mã nguồn của hợp đồng hoặc bạn đã nhập địa chỉ của một hợp đồng đã được xác minh.

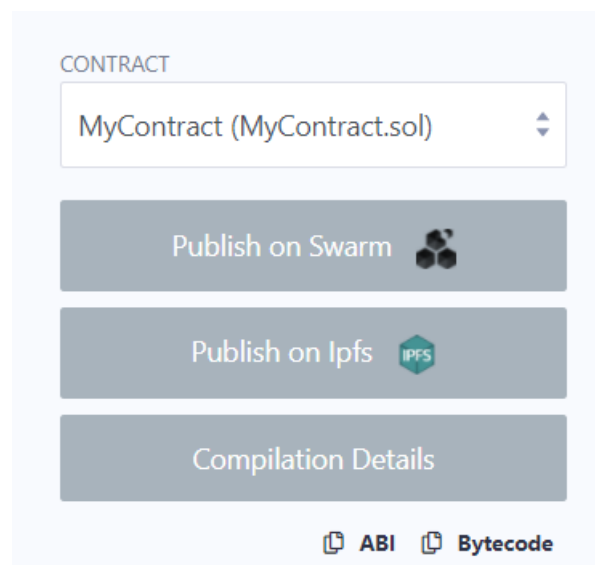
Bây giờ, ta cùng tìm hiểu kỹ về Debugger:

Ta tạo ra một contract đơn giản :

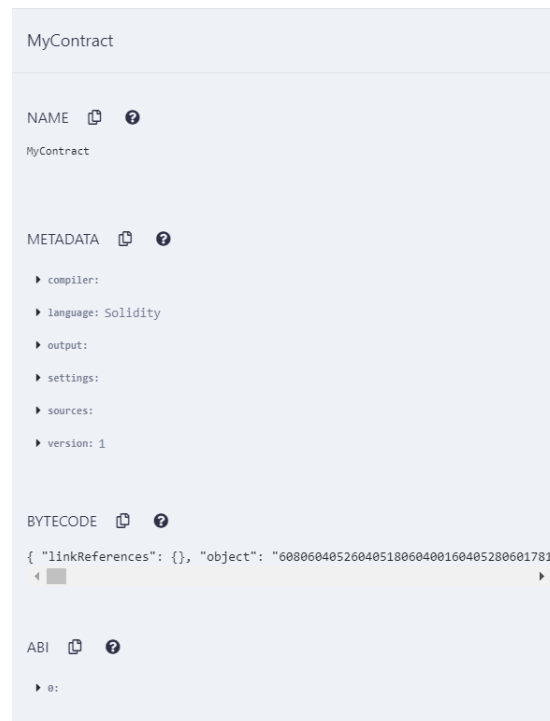
```
pragma solidity ^0.5.13;

contract MyContract{
    string public myString = "Hello the real world!!!";
}
```

Để xem chi tiết trình biên soạn của 1 hợp đồng, ta nhấn vào Compilation Details:

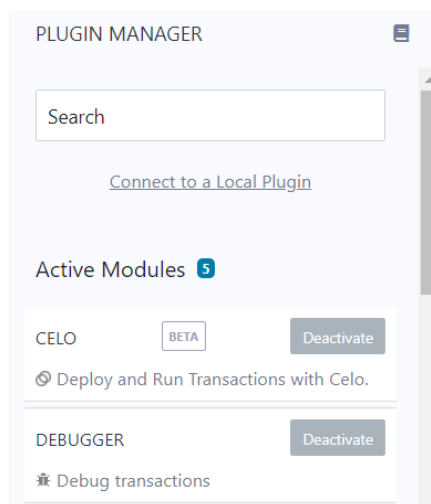


Khi đó, sẽ xuất hiện một bảng các thông số gồm :

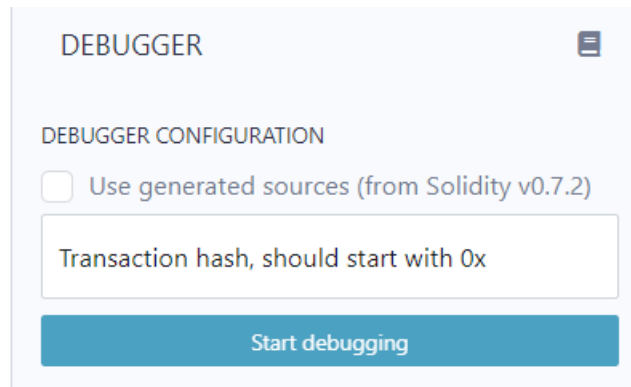


Để bắt đầu phiên gỡ lỗi:

Kích hoạt Trình gỡ lỗi trong Trình quản lý Plugin



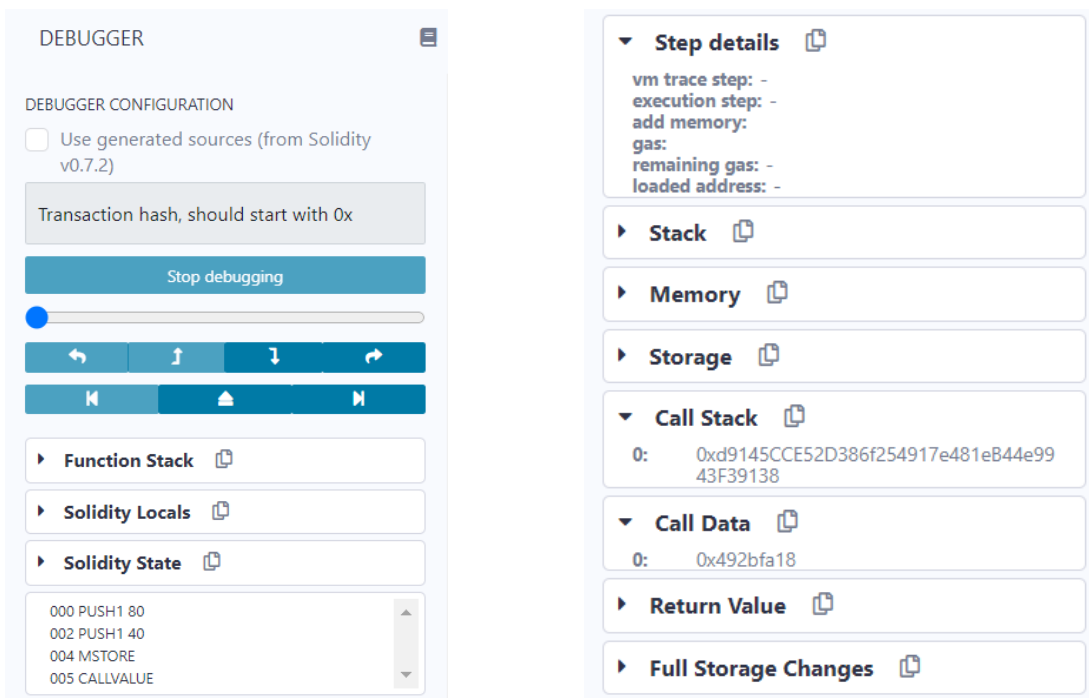
Tiếp đó, ta nhấp vào biểu tượng của Debugger và bắt đầu trình gỡ lỗi.



Tiếp theo đó, ta tiến hành Deploy và gọi myString. Khi đó, sẽ hiển thị thông báo giao dịch ta nhận vào Debug để bắt đầu gỡ lỗi.

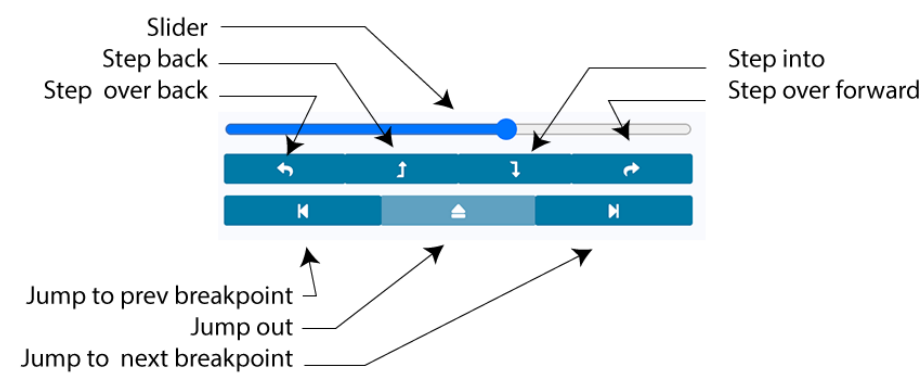


Khi đó, Debugger sẽ xuất hiện 1 trang mới:



Trang này sẽ xem xét tùy chọn Nguồn được tạo của Trình gỡ lỗi, điều hướng và bảng điều khiển của nó.

Sử dụng các nguồn đã tạo sẽ giúp bạn dễ dàng kiểm tra các hợp đồng của mình hơn. Khi tùy chọn được chọn, bạn có thể bước vào các đầu ra của trình biên dịch đó - trong khi gỡ lỗi.



Slider: Di chuyển thanh trượt sẽ làm nổi bật mã có liên quan trong Trình chỉnh sửa. Ở cấp độ chi tiết nhất, nó cuộn qua các mã opcode của một giao dịch (xem phần opcode bên dưới). Tại mỗi opcode, trạng thái của giao dịch thay đổi và những thay đổi này được phản ánh trong bảng điều khiển của Trình gỡ lỗi.

Step over back: Nút này chuyển đến opcode trước đó. Nếu bước trước đó liên quan đến một lệnh gọi hàm, hàm sẽ không được nhập.

Step back: Nút này quay lại mã opcode trước đó.

Step into: Nút này chuyển sang mã opcode tiếp theo. Nếu dòng tiếp theo chứa một lệnh gọi hàm, Bước vào sẽ đi vào hàm.

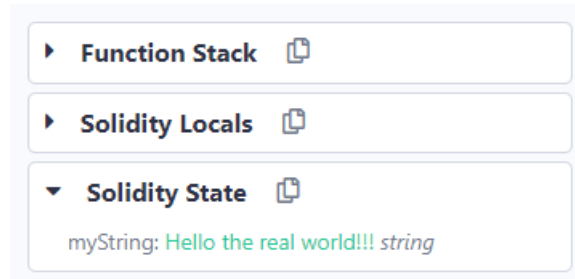
Step over forward: Nút này chuyển sang mã opcode tiếp theo. Nếu bước tiếp theo liên quan đến một lệnh gọi hàm, hàm sẽ không được nhập.

Jump to the previous breakpoint: Các điểm ngắt có thể được đặt trong rãnh của Trình chỉnh sửa. Nếu bước hiện tại trong cuộc gọi đã vượt qua điểm ngắt, nút này sẽ di chuyển thanh trượt đến điểm ngắt được thông qua gần đây nhất.

Jump out: Khi bạn đang trong cuộc gọi và nhấp vào nút này, thanh trượt sẽ được chuyển đến cuối cuộc gọi.

Jump to the next breakpoint: Nếu một điểm ngắt ở phía trước trong mã, nút này sẽ tiến tới điểm đó.

Bảng điều khiển của Debugger:

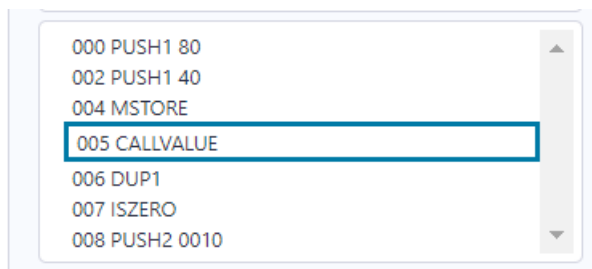


Function Stack: liệt kê các chức năng mà giao dịch đang tương tác.

Solidity Locals: là các biến cục bộ bên trong một hàm.

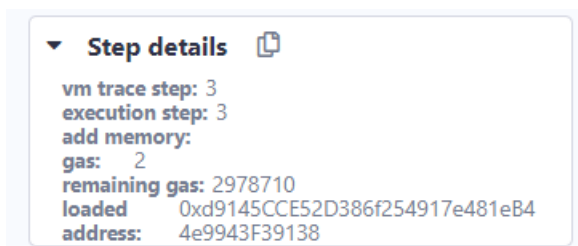
Solidity State: Đây là các biến trạng thái của hợp đồng.

Opcodes: Bảng điều khiển này hiển thị số bước và mã opcode mà trình gỡ lỗi hiện đang bật.

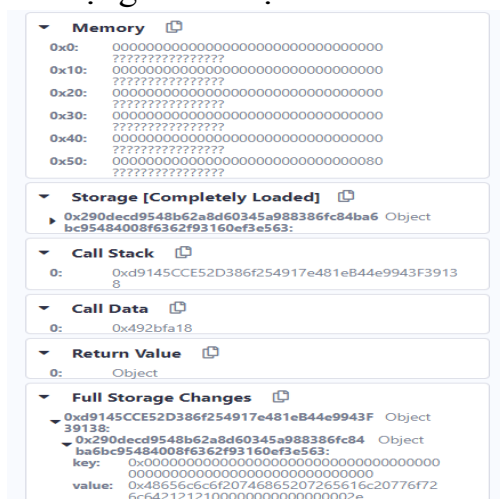


Khi bạn kéo thanh trượt (phía trên các nút điều hướng), số bước được tiêu điểm và mã chọn sẽ thay đổi.

Step details: Chi tiết bước hiển thị thêm thông tin về bước opcode.



Memory: Bộ nhớ sẽ bị xóa cho mỗi cuộc gọi tin nhắn mới. Bộ nhớ là tuyến tính và có thể được định địa chỉ ở mức byte. Đọc được giới hạn ở độ rộng 256 bit trong khi ghi có thể có chiều rộng 8 bit hoặc 256 bit.



Storage: Đây là kho lưu trữ lâu dài.

Call Stack: Tất cả các tính toán được thực hiện trên một mảng dữ liệu được gọi là call stack. Nó có kích thước tối đa là 1024 phần tử và chứa các từ 256 bit.

Call Data: Dữ liệu cuộc gọi chứa các tham số hàm.



Return Value: Tham chiếu đến giá trị mà hàm sẽ trả về.

Full Storage Changes: Điều này cho thấy lưu trữ liên tục ở cuối chức năng.

LAB 16

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

Sau đó, ta thực hiện khai báo phiên bản Solidity đang sử dụng:

```
pragma solidity ^0.5.11;
```

Tiếp theo, ta tạo 1 contract với tên LibrariesSimple

Bên trong contract ta khai báo các biến :

Ta sử dụng mapping: với tham số truyền đi là address và trả về kiểu uint , gắn với biến của mapping là tokenBalance.

```
contract LibrariesSimple{  
    mapping(address => uint)public tokenBalance;
```

Tiếp theo, ta tạo 1 hàm khởi tạo, với đầu vào là số token của địa chỉ Deploy bằng 1. Và khi ta muốn xem số token của bất kì địa chỉ nào.

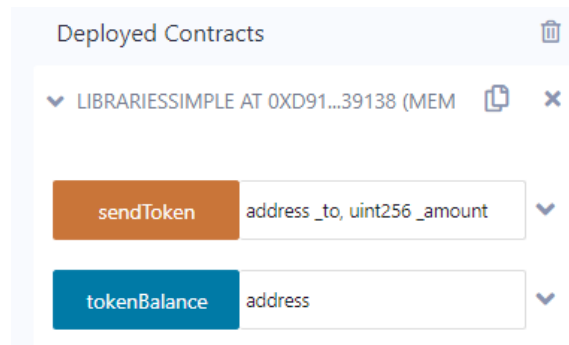
```
contract LibrariesSimple{  
    mapping(address => uint)public tokenBalance;  
  
    constructor()public{  
        tokenBalance[msg.sender] = 1;  
    }  
}
```

Ta thực hiện tạo 1 hàm sendToken :

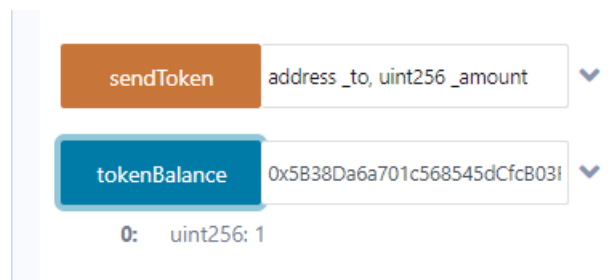
```
function sendToken(address _to, uint _amount)public returns(bool){  
    tokenBalance[msg.sender] -= _amount;  
    tokenBalance[_to] += _amount;  
  
    return true;  
}
```

Hàm này có chức năng gửi token đến 1 địa chỉ nào đó. Hàm trả về giá trị bool (true or false).

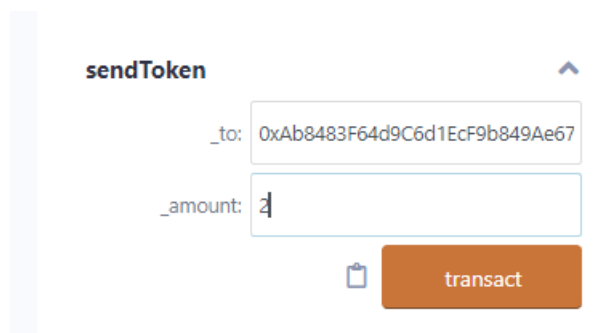
Bây giờ, ta tiến hành Deploy:



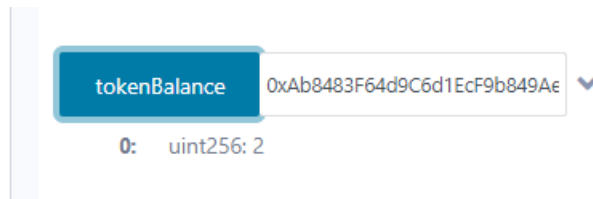
Bây giờ, để xem số token của địa chỉ thực hiện Deploy ta chỉ cần copy địa chỉ đó và dán vào trường tokenBalance rồi nhấn tokenBalance. Và chắc chắn rằng giá trị hiển thị sẽ là 1.



Để gửi token, ta copy bất kì 1 địa chỉ nào đó và dán vào trường sendToken cùng với số token cần gửi, sau đó nhấn transact để bắt đầu giao dịch:

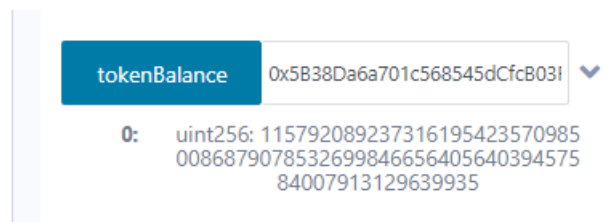


Khi đã giao dịch thành công, ta copy địa chỉ đã được gửi đó dán vào trường tokenBalance để xem kết quả:



Với địa chỉ được gửi, mặc dù vẫn nhận được giá trị là 2, nhưng bây giờ hãy cùng kiểm tra lại giá trị của địa chỉ ban đầu như nào nha:

Ta thấy, giá trị trả về sẽ là một giá trị lớn nhất của uint. Vì ta thấy, giá trị đầu vào của địa chỉ đó chỉ tối đa là 1, nhưng ta thực hiện gửi đi 2. Như thế là không hợp lệ.



Bây giờ, ta thực hiện import 1 thư viện (library):

Chúng ta vào đường dẫn sau: <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/61973af29f469e62dadf9df239699175d5bee95e/contracts/math/SafeMath.sol>

Ta copy link của đường dẫn này và thực hiện import vào file của mình.

Library tương tự như Contract nhưng chủ yếu nhằm mục đích sử dụng lại. Library chứa các hàm mà các contract khác có thể gọi.

Sau khi được triển khai trên blockchain (chỉ một lần), nó được gán một địa chỉ cụ thể và các thuộc tính / phương thức của nó có thể được sử dụng lại nhiều lần bởi các hợp đồng khác trong mạng Ethereum.

Sau khi đã import vào ta khai báo lại phiên bản của Solidity.

Như vậy, lúc này ta có:

```
pragma solidity ^0.6.0;  
  
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/61973af29f469e62dadf9df239699175d5bee95e/contracts/math/SafeMath.sol";
```

SafeMath là một thư viện toán học vững chắc được thiết kế đặc biệt để hỗ trợ các hoạt động toán học an toàn: an toàn có nghĩa là nó ngăn chặn tràn khi làm việc với uint.

Bên trong contract, Để sử dụng lại SafeMath ta sử dụng:

```
contract LibrariesSimple {  
  
    using SafeMath for uint;
```

có thể được sử dụng để đính kèm các hàm thư viện (từ thư viện) vào bất kỳ kiểu dữ liệu nào trong ngữ cảnh của một hợp đồng. Các hàm này sẽ nhận đối tượng mà chúng được gọi làm tham số đầu tiên.

Khi đó, thay vì lúc đầu ta sử dụng += hoặc -= để thể hiện phép tính thì bây giờ, ta tham chiếu đến thư viện SafeMath để sử dụng các phép tính toán.

```
function sendToken(address _to, uint _amount) public returns (bool) {  
    tokenBalance[msg.sender] = tokenBalance[msg.sender].sub(_amount);  
    tokenBalance[_to] = tokenBalance[_to].add(_amount);  
  
    return true;  
}
```

Các function của thư viện SafeMath:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {  
    uint256 c = a + b;  
    require(c >= a, "SafeMath: addition overflow");  
  
    return c;  
}
```



```

function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    return sub(a, b, "SafeMath: subtraction overflow");
}

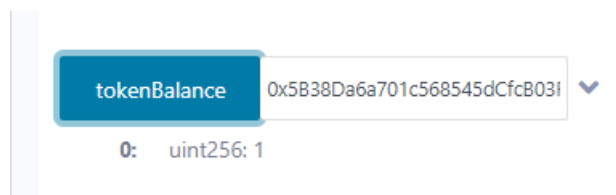
/**
 * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
 * overflow (when the result is negative).
 *
 * Counterpart to Solidity's '-' operator.
 *
 * Requirements:
 * - Subtraction cannot overflow.
 */
function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
    require(b <= a, errorMessage);
    uint256 c = a - b;

    return c;
}

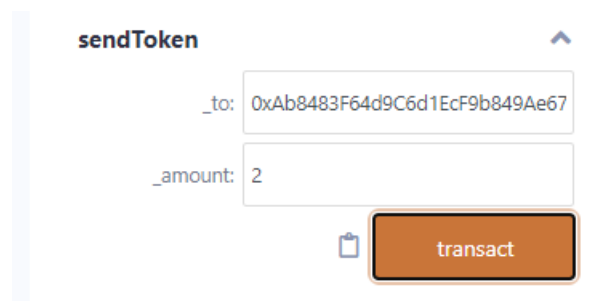
```

Bây giờ, ta thực hiện Deploy:

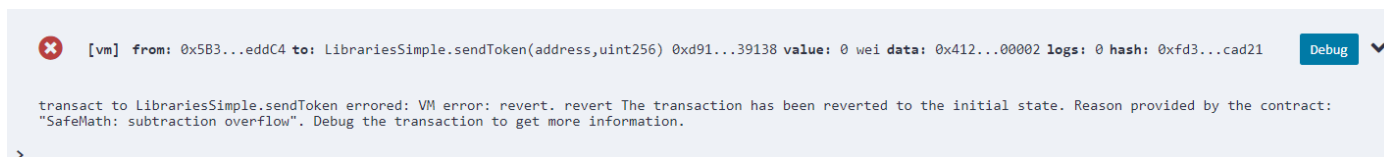
Số token của địa chỉ Deploy là 1.



Lúc này, ta thử gửi hơn 1 token cho một địa chỉ khác xem hiện tượng xảy ra như thế nào: ta gửi 2 token và thực hiện giao dịch.



Ta xem bảng biểu Remix hiển thị thông báo kết quả như nào nha:



Một lỗi đã xảy ra, ta hiểu đơn giản: Tổng số token hiện có là 1, nhưng số tiền cần gửi đi lại lớn hơn số token hiện có, như thế không hợp lệ. Hãy chú ý hàm sub mình đã để ở trên nhé!.

Như vậy, ta chỉ mới thực hiện kế thừa Libraby. Bây giờ, ta sẽ tạo 1 file mới và thực hiện về libraby nhé.

Cũng tương tự như contract ta cũng thực hiện khai báo phiên bản Solidity:

```
pragma solidity >=0.4.16 <0.7.0;
```

Tiếp theo, ta tạo một library:

```
library Search{

    function indexOf(uint[] storage self, uint value)public view returns(uint){
        for(uint i =0; i< self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}
```

Như chúng ta đã thấy, các hợp đồng thư viện không có dung lượng lưu trữ. Do đó, chúng không thể giữ các biến trạng thái (các biến trạng thái không phải là hằng số). Tuy nhiên, các thư viện vẫn có thể triển khai một số kiểu dữ liệu:

struct và enum: đây là các biến do người dùng xác định.

Bất kỳ biến nào khác được định nghĩa là constant(không thay đổi), vì các biến hằng số được lưu trữ trong bytecode của hợp đồng, không phải trong bộ nhớ.

Ta tạo 1 function indexOf gồm các biến hằng được lưu trữ trong storage, giá trị trả về kiểu uint. Và bên trong ta thực hiện 1 vòng lặp.

Tiếp theo, ta tạo 1 contract:

```
contract NotUsingForExample{
    uint[] data;

    function append(uint value)public{
        data.push(value);
    }

    function replace(uint _old, uint _new) public{
        //This performs the library function calldata
        uint index = Search.indexOf(data, _old);
        if(index == uint(-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Ta khai báo biến uint[] data

Tạo 1 function append chứa kiểu uint. Và hàm trả về sẽ là 1 giá trị được đẩy lên.

Tiếp theo, ta tạo 1 hàm replace. Hàm này có tác dụng thay thế giá trị cũ và mới.

Hàm này sẽ gọi dữ liệu từ hàm Search: Nếu index truyền vào bằng -1 thì giá trị sẽ được đẩy lên 1 giá trị mới và ngược lại index sẽ nhận giá trị.

Ta có 1 contract khác:

```
contract UsingForExample{
    using Search for uint[];
    uint[] data;

    function append(uint value) public{
        data.push(value);
    }

    function replace(uint _old, uint _new) public{
        //This performs the library call
        uint index = data.indexOf(_old);
        if(index == uint (-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Ta nhận thấy, contract này được kế thừa từ library Search với cách khai báo:

Using Search for uint[];