City University of Hong Kong
CS4389 Decentralized Applications Development
2019/20 Semester B

(Deadline: 23rd March 2020)

Submit to Canvas

Instructions:

1. Use Solidity compiler version 0.5.12 in answering all the questions of this assignment.
2. Put the answers of Questions 1 and 2 into the file **<StudentID>AddressConversion.sol** .
3. Put the answers of Questions 3, 4 and 5 into the file **<StudentID>Dependency.sol** .
4. Put the answers of Questions 6 and 7 into the file **<StudentID>BusinessScheme.sol** .
5. **Put all the files in a single .rar file and submit it to canvas.**

Mark-Grade Mapping

Mark	Grade
80 or above	A+
70 to 79	A/A-
55 to 69	B-/B/B+
40 to 54	C-/C/C+
35 to 39	D
Below 35	F

Debugging and Development Strategy:

- Emit events to serve as debugging messages to assist you to know the execution process and program state in the program execution.
- Reverting program state could be difficult to avoid in developing smart contracts. Do not use require(), assert() or revert() before you have developed your working code.
- **Beware of the gas/ether usage** in your test run.
    - Ensure a contract instance A to have ethers before making ether transfers.
    - Ethereum requires each invocation of fallback function to incur very small gas limit. Updating state variable or emitting an event or calling a function will normally make your transaction fail due to out-of-gas error. Simply leave fallback function empty.
    - A block has a gas limit. Do not invoke a function that will execute many code and many functions through a Transactions. By doing so, it will generate an out-of-gas error.
- A contract instance once created (or deployed via remix or truffle) has its unique address. If another contract needs to refer to the former instance, the deployment actually refers to that contract address rather than the name of the former contract. So, if you update the former contract (or in Remix, you delete the former contract), the latter contract will refer to an old contract code or a non-existing code. These situations will make debugging very difficult. A better strategy is to always clean all smart contracts and re-deploy the whole set of contracts.

Question 1 [10%]. Write a smart contract called **C1** to have the following features.

C1 contains a function **address2String(address) external pure returns (string memory)** that accepts an address as a parameter and returns a string that shows the address in ASCII code format in full but without extra characters.

decoded input	{ "address x": "0xB87213121FB89CbD8B877Cb1Bb3FF84dD2869cfA" }
decoded output	{ "0": "string: 0xb87213121fb89cbd8b877cb1bb3ff84dd2869cfa" }

Question 2 [10%]. Write a smart contract **C2** that contains a function **getAddress() public returns (string memory)** that creates an instance **x** of contract C1, and invokes **x.address2String(address(this)),** and further format the returned string so that the string content is shown in the checksum address format and then **emit** the output of *x.address2String(address(this))* and the checksum address into the transaction log as an event **Addresses(string given_address, string checksum_address)**. The checksum address is also returned by **getAddress()** as a string.

- [Hint: You should get the keccak256() hash-value of the address (note that you need to handle the prefix "0x", which is not a part of the address value), and then represent the obtained hash-value into a format that takes two bytes to represent a hexadecimal number), and finally capitalize the alphabets if needed.]

Question 3. [10%] Copy C1 as contract C3 and copy C2 as contract C4.

Choose to implement one of the following two sets:

Case A:

1. Modify **getAddress()** of C4 so that it creates an instance of C3 and returns the address of this C3 instance in the string format instead of returning the address of "this" in the string format.
2. Add a function **pay(uint x, address y)** to contract C3. This function will *transfer* **x wei** to the address returned by **y.getAddress()** where **y** is an instance of contract C4.

Case B:

1. Modify getAddress() of C4 into **getAddress(address given_address)** so that it returns the given address in the string format.
2. Add a function **pay(uint x, address y)** to contract C3. This function will *transfer* **x wei** to the address returned by **y.getAddress(address(this))** where **y** is an instance of contract C4.

Hints:

- Hint 1: In your code, if C4 creates an instance of C3 in its contract body, C3 should be deployed before deploying C4; otherwise, C4 cannot be deployed successfully.
- Hint 2: You may use **Interface** to help you to resolve the cyclic dependencies between C3 and C4.]

The following code skeleton is provided to you.

```
interface IC3 {
   function address2String(address) external pure returns(string memory);
   function pay (uint , address) external;
}

interface IC4 {
   // uncomment one of the following two lines
   // function getAddress() external returns (string memory);
  // function getAddress(address given_address) external returns (string memory);
}

contract C3 is IC3 { …}
contract C4 is IC4 {…}
```

Question 4. [10%] Write a contract **Rewards** with the following properties:

State variables	Meaning
mapping (address=>uint) rewards_ledger	user address maps to its reward points balance, initially empty. **Each reward point is equivalent to one wei.**
address[] clients_list	A list of clients, initially empty
uint reward_ratio	How many reward points earned for each wei spending? e.g., if reward_ratio is 8, and spending is 8 wei, then there will be 1 reward point.

- **function earnRewards(address current_client, uint spending) returns (bool status).** This function will insert *current_client* into *clients_list* if *current_client* does not exist in *clients_list*, and update the reward points balance of *current_client* kept in *rewards_ledger*. The ratio between reward and spending is based on the state variable ***reward_ratio***. The variable *spending* keeps the spending in wei. The function returns true if current_client exists in *client_list* before the function is invoked; otherwise, it returns false.
- **redeemRewards(address current_ client, uint points) returns (bool status).** This function will deduct the amount of reward points kept in *rewards _ledger* for *current_ client* by the amount of reward points specified in the second parameter of the function. It ensures that the ledger after deduction will remain non-negative. It will also make the function call **msg.sender.transfer(points)** that transfer certain wei equivalent to the amount of points deducted above to the contract instance that calls *redeemRewards*(). The function returns a Boolean value denoted by **status.** The function returns true if the redeemable reward points are not less than points requested and a transfer operation has been performed, otherwise, it returns false.

- The value of **reward_ratio** is inputted as a parameter when the instance is created. This value will not be changed by the instance.
- **getRewardRatio()**. This function will return the conversion ratio used by the instance.

Question 5 [10%]. Copy C3 as contract C5. In contract C5, **revise** pay(uint x) so that the function will accept an instance **rw** of *Rewards* as input parameter, transfer x wei to **rw**, invoke **rw.**earnRewards(this, x) followed by **rw.**redeemRewards(this, *points*) where *points* is calculated based on the reward ratio returned by **rw.**getRewardRatio(). [Hint: You may require to create or revise some Interfaces too.]

Question 6 [50%]. The basic idea of this question is to enable a *businessOutlet* to create a *scheme* for the other *businessOutlets*, and to join the *scheme* upon receiving an invitation from another *businessOutlet* (including itself), the *businessOutlet* can vote for agreeing on the *scheme* or vote for disagreement. A *scheme* is established if there are more agreed votes than a given threshold.

Write the contract **BusinessOutlet** and the contract **Scheme** with the following properties and copy *Rewards* as contract **myRewards**. **[Note: You may add other parameters or functions or Interfaces to assist you in developing BusinessOutlet and Scheme.]**

- BusinessOutlet
    - When creating the *businessOutlet* instance, the constructor of the instance accepts a string as the businessOutlet name.
    - **createScheme(address[] businessOutlets, string scheme_name, string scheme_content, uint agree_vote_threshold, uint reward_ratio).** This function will invoke the function **createScheme(businessOutlets, scheme_name, scheme_content, agree_vote_threshold, reward_ratio)** of the instance **s** of *Scheme*. Moreover, it will pass the reward ratio to the Reward contract. [Hint: if you set the function as **external**, you should declare array and string using **calldata**, otherwise, you should declare them as **memory**.]
    - **schemeInvitation(address a)**. This function will receive an address of a *Scheme* instance that some other businessOutlet has invited this businessOutlet to vote on.
    - **vote(bool myvote).** This function will cast the vote of the businessOutlet regarding the **invited** Scheme.
    - **joinRewardsScheme(myRewards**) or **joinRewardsScheme (I_Rewards)**. IRewards is the Interface for myRewards. You only need to implement one of these two functions. The function to be implemented will accepts an instance of myRewards.
    - **payBill(uint spending).** This function will call earnRewards of the myRewards instance received by joinRewardsScheme().
    - **payBillByRewards (uint points).** This function will call redeemRewards of the myRewards instance received by joinRewardsScheme().
- Scheme
    - **createScheme(address[] businessOutlets, string schemeName, string scheme_content, uint scheme_threshold, uint ratio)**. This function will ensure each address in businessOutlets is an existing contract address by transferring 1 wei to each such businessOutlet. It will send itself to each such businessOutlet in the businessOutlet list.

- **vote(bool agree)**. This function will check whether the businessOutlet calling this function has been invited to make decision on the scheme, keeps track of the votes received.
- **createConsensus(myRewards rw)**. This function will check whether the sum of votes received exceeds the *scheme_threshold* received via createScheme() of the Scheme instance. If this is the case, it passes **rw** to each businessOutlet that has agreed in joining the scheme. [Note that if a businessOutlet votes to disagree, that businessOutlet will not receive the myRewards instance.]

- **Question 7 [Bonus marks 10%]** Modify myRewards with the following features. Thus, after a scheme is established, a Rewards contract instance is created for a set of agreed businessOutlets. These businessOutlets can now use this joint Rewards contract to book-keep the balance of reward points of each customer.

  myRewards

  - Revise **earnRewards** and **redeemRewards** so that these two functions will check whether the caller of these two functions are in the businessOutlet list provided by the Scheme instance.
  - **setup(address[] businessOutlets, uint ratio)**: This function will be called by createConsensus() before passing it to each businessOutlet in the businessOutlet list.

The following contract User is given to you.

```
contract User is I_User {
    string name;
    function () external payable {}
    constructor (string memory myName) public payable { name = myName; }
    function pay(address payable businessOutlet_address, uint spending) external {
        businessOutlet_address.transfer(spending);
        I_BusinessOutlet(uint160(businessOutlet_address)).payBill(spending);
    }
    function redeem(address payable businessOutlet_address, uint points) external returns (bool status)
 {
        return I_BusinessOutlet(uint160(businessOutlet_address)).payBillByRewards(points);
    }
}
```

The following test scenario creates three businessOutlets, where businessOutlet b0 makes a schemes to all three businessOutlets, and two of them accepts the scheme. There are two users. The first user pays (and earns points) and redeem earned points, and the second user can only pay the bill without earning points as businessOutlet b1 does not join the bonus point scheme.

```
contract TestCase {
    // warning:   need lots of gas: set gas limit to 30000000 to deploy this contract.
    // you should have all the above contracts deployed before deploy TestCase
    function () external payable {}
    constructor () public payable { }

    function Test01() public payable {
        // run by a test EAO with sufficient ethers (e.g., 100 ethers)
        // make sure to trnasfer ether to this contract address before running Test01()
        I_BusinessOutlet b0 = new BusinessOutlet("BusinessOutlet 0");
        I_BusinessOutlet b1 = new BusinessOutlet("BusinessOutlet 1");
        I_BusinessOutlet b2 = new BusinessOutlet("BusinessOutlet 2");
        address(b0).transfer(0.20 ether);
        address(b1).transfer(0.10 ether);
        address(b2).transfer(0.10 ether);
        address[] memory partners = new address[](3);
        partners[0] = address(b0);
        partners[1] = address(b1);
        partners[2] = address(b2);


        I_Rewards myRewards = new myRewards();
        address(myRewards).transfer(0.10 ether);
        address _s =  b0.createScheme(partners, "Joint Promotion Plan 1",
                                "Earn Points from our Partners! Earning one point for every 8 dollars
                                spending!",2, 8);

        address payable s = address(uint256(_s));
        b1.vote(false); // not join
        b2.vote(true); // join
        b0.vote(true); // join
        I_Scheme(s).createConsensus(myRewards);  // s will populate Scheme

        I_User u0 = new User("Mary");
        I_User u1 = new User("John");
        address(u0).transfer(0.10 ether);
        address(u1).transfer(0.10 ether);
        u0.pay(address(b2), 2000);  // user u0 spends 2000 dollars (i.e., wei)
        u0.redeem(address(b2), 200);
        u1.pay(address(b1), 1000);  // b1 does not join the scheme.
    }
}
```

Sample Output of executing the test function Test01()

[**Hint for debugging**: You may (1) emit some events to the Transaction Log to help you visualize the sequence of actions, and/or (2) change some state variables to public to view the values in Remix.]

status	0x1 Transaction mined and execution succeed
transaction hash	0x7be56468ed6a4664aebe739b37b770751a2b7cb5b38a72f8774f968019a34a0e
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	TestCase.Test01() 0xef55bfac4228981e850936aaf042951f7b146e41
gas	30000000 gas
transaction cost	9151803 gas
execution cost	9130531 gas
hash	0x7be56468ed6a4664aebe739b37b770751a2b7cb5b38a72f8774f968019a34a0e
input	0x181...0e354
decoded input	{}
decoded output	{}
logs	[]
value	0 wei

-end-