Practical session for blockchain

(7) Advanced Token Contract

Jungyoon Song[1] Giyeong Lee[1]

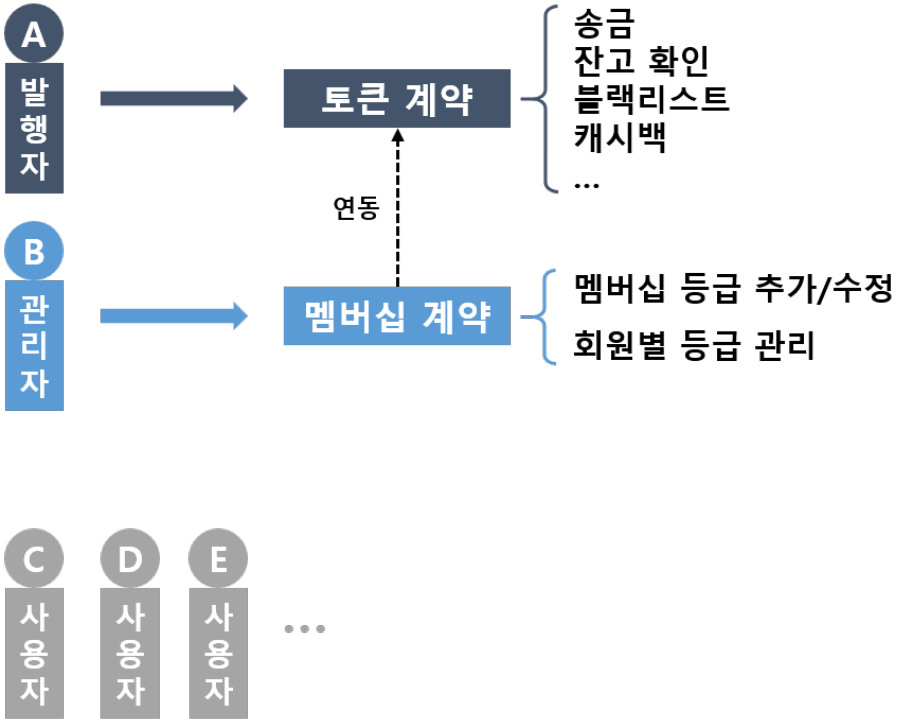
[1]Financial Risk Engineering Lab.
Seoul National University

May 11, 2022

Structure and Procedures of Advanced Token Code

- **Previous lecture** All members are the **same** membership
- **In this lecture,**
  1. Membership differential application based on criteria
  2. Use **inheritance** to increase the re-usability of the existing code
  3. **Interworking** between smart contracts

Structure and Procedures of Advanced Token Code



멤버십 계약 구조 (예시)

멤버십 등급

등급	최소 거래 횟수	최소 결제 토큰	캐시백 비율
Gold	10	1500	10
Silver	5	500	5
Bronze	0	0	0

멤버십 명부

사용자	거래횟수	거래금액	회원등급
A	21	1600	Gold
C	4	200	Bronze
D	10	1000	Silver

Membership Contract

Inheritance

- Solidity supports inheritance between smart contracts, where multiple contracts can be inherited into a single contract.

[Ex. 1] Membership Contract

```
pragma solidity >= 0.7.0 < 0.8.0;
contract Owned {
    address public owner;
    event OwnershipTransfer(address _oldAddr, address _newAddr);
    modifier onlyOwner() { require(msg.sender == owner, "Only owner can run this function"); _; }

    // Designate contract distributor as contract owner
    constructor() {
        owner = msg.sender;
    }

    // Transfer contract ownership
    function transferOwnership(address _newAddr) public onlyOwner {
        address oldAddr = owner;
        owner = _newAddr;
    emit OwnershipTransfer(oldAddr, owner);
    }
}
```

Membership Contract

[Ex. 1] Membership Contract

```
contract Membership is Owned {

    struct MembershipLevel {
        string name;                // Membership name
        uint256 times;              // Minimum number of transactions required to achieve the grade
        uint256 sum;                // Minimum transaction amount required to achieve the grade
        int8 rate;                // Cashback Rate
    }

    struct History {
        uint256 times;          // Cumulative number of transactions
        uint256 sum;            // Cumulative transaction amount
        uint256 levelIndex;          // Current membership grade index
    }

    address public tokenAddr;
    MembershipLevel [] public levels;
    mapping(address => History) public tradingHistory;
    modifier onlyToken() { require(msg.sender == tokenAddr, "This function is not for users"); _; }
```

Membership Contract

[Ex. 1] Membership Contract

```
// constructor omitted (inheritance)
// Interworking between smart contracts
function setToken(address _tokenAddr) public onlyOwner {
    tokenAddr = _tokenAddr;
}

function pushLevel(string memory _name, uint256 _times, uint256 _sum, int8 _rate) public onlyOwner {
    levels.push(MembershipLevel({
        name : _name,
        times : _times,
        sum : _sum,
        rate : _rate
    }));
}
function editLevel(uint256 _index, string memory _newName, uint256 _newTimes,
                uint256 _newSum, int8 _newRate) public onlyOwner {
                    require(_index < levels.length);
                    levels[_index].name = _newName;
                    levels[_index].times = _newTimes;
                    levels[_index].sum = _newSum;
                    levels[_index].rate = _newRate;
                }
```

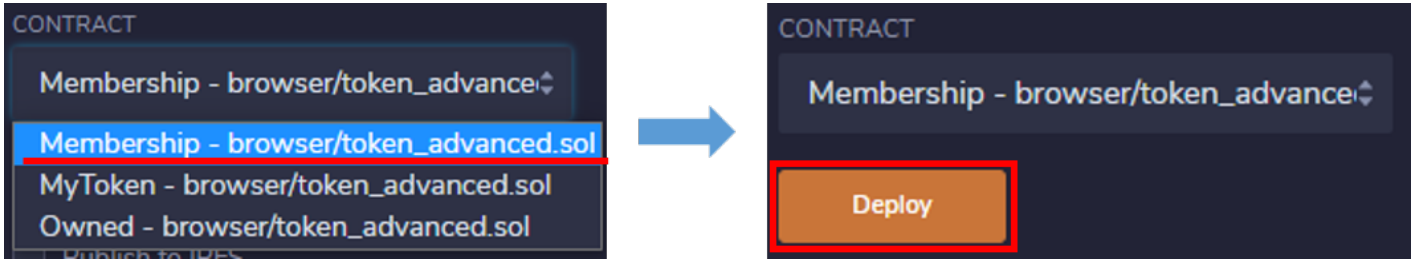
Membership Contract

[Ex. 1] Membership Contract

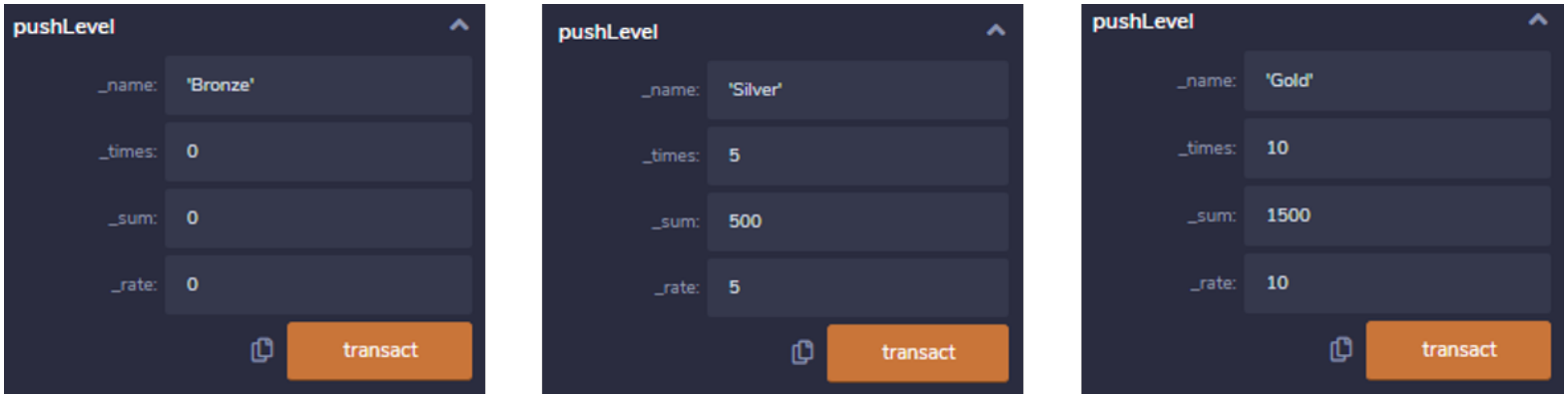
```
    function updateHistory(address _member, uint256 _value) public onlyToken {
                    tradingHistory[_member].times += 1;
                    tradingHistory[_member].sum += _value;
                    uint256 index = tradingHistory[_member].levelIndex;
                    int8 tmpRate;
        for (uint i = 0; i < levels.length; i++) {
            if (tradingHistory[_member].times >= levels[i].times
                && tradingHistory[_member].sum >= levels[i].sum
                && tmpRate < levels[i].rate) {
                    index = i;
                    tmpRate = levels[i].rate;
                }
            }
            tradingHistory[_member].levelIndex = index;
        }
        function getCashbackRate(address _member) public view returns (int8 rate) {
           rate = levels[tradingHistory[_member].levelIndex].rate;
        }
}
```

Membership Contract

- Deploy membership contract

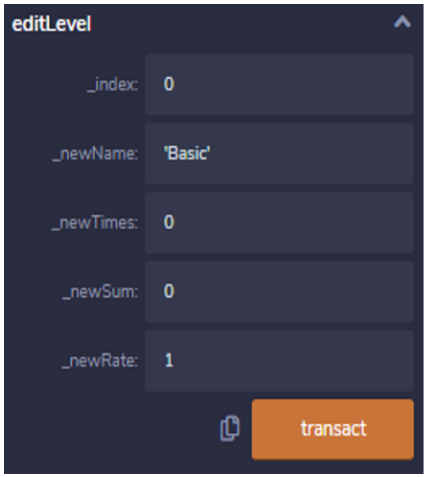


- Add membership degree

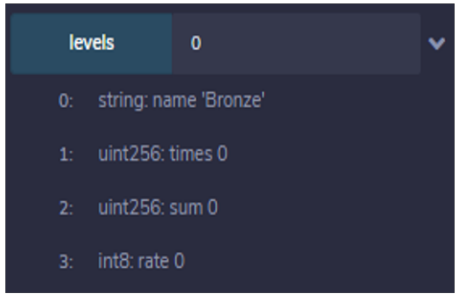


Membership Contract

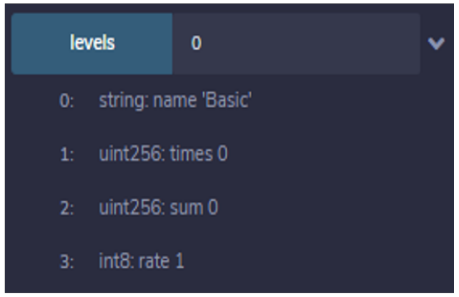
- Edit membership degree



<Before>



<After>



Advanced Token Contract

[Ex. 2] Advanced Token Contract

```
contract MyTokenWithFeatures is Owned{
    uint256 public totalSupply; // uint256 == uint
    mapping (address => uint256) public balanceOf;
    mapping (address => mapping(address => uint256)) private approved;
    mapping (address => Membership) public memberships;

    string public name;
    string public symbol;
    uint256 public decimals;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

    // Blacklist
    mapping (address => uint8) public blacklist;
    event Blacklisted(address indexed _address);
    event DeletedFromBlacklist(address indexed _address);
    event RejectedPaymentToBlacklistedAddr(address indexed _from, address indexed _to, uint256 _value);
    event RejectedPaymentFromBlacklistedAddr(address indexed _from, address indexed _to, uint256 _value);
    event Cashback(address indexed _from, address indexed _to, uint256 _cashback);
```

Advanced Token Contract

[Ex. 2] Advanced Token Contract

```
constructor (string memory _name, string memory _symbol,
             uint256 _supply, uint256 _decimals) {
    name = _name;
    symbol = _symbol;
    decimals = _decimals;

    balanceOf[msg.sender] = _supply * 10 ** _decimals;
    totalSupply = _supply * 10 ** _decimals;
}
function setMembership(Membership _membershipAddr) public {
    memberships[msg.sender] = Membership(_membershipAddr);
}
```

Advanced Token Contract

[Ex. 2] Advanced Token Contract

```
function isBlacklisted(address _address) public view returns (bool inBlacklist) {
    inBlacklist = blacklist[_address] == 1;
            return inBlacklist;
}

function pushBlacklist(address _address) public onlyOwner {
    blacklist[_address] = 1;
    emit Blacklisted(_address);
}

function deleteFromBlacklist(address _address) public onlyOwner {
    blacklist[_address] = 0;
    emit DeletedFromBlacklist(_address);
}

function isValidTransfer(address _from, address _to, uint256 _value)
internal view returns (bool isValid) {
    if (balanceOf[_from] >= _value && balanceOf[_to] + _value >= balanceOf[_to]) {
        isValid = true;
    } else {
        isValid = false;
    }

    return isValid;
}
```

Advanced Token Contract

[Ex. 2] Advanced Token Contract

```
function transfer(address _to, uint256 _value) public returns (bool success) {
    if (isBlacklisted(msg.sender)) {
        emit RejectedPaymentFromBlacklistedAddr(msg.sender, _to, _value);
        success = false;
    } else if (isBlacklisted(_to)) {
        emit RejectedPaymentToBlacklistedAddr(msg.sender, _to, _value);
        success = false;
    } else if (isValidTransfer(msg.sender, _to, _value)) {
        uint256 cashBackRate = uint256(memberships[_to].getCashbackRate(msg.sender));
        uint256 cashback = _value * cashBackRate / 100;
        memberships[_to].updateHistory(msg.sender, _value);

        _value -= cashback;
        emit Cashback(msg.sender, _to, cashback);

        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;

        emit Transfer(msg.sender, _to, _value);
        success = true;
    } else {
        success = false;
    }

    return success;
}
```

Advanced Token Contract

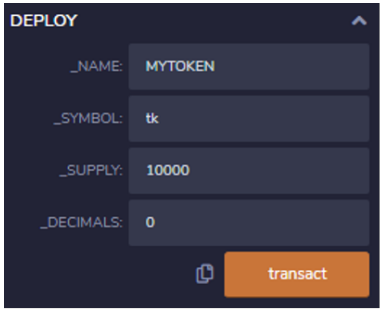
[Ex. 2] Advanced Token Contract

```
    function approve(address _spender, uint256 _value) public returns (bool success) {
        if (approved[msg.sender][_spender] + _value >= approved[msg.sender][_spender]) {
            approved[msg.sender][_spender] = approved[msg.sender][_spender] + _value;
            emit Approval(msg.sender, _spender, _value);
            success = true;
        } else {
            success = false;
        }
    }
    function allowance(address _owner, address _spender) public view returns (uint256 remaining) {
        remaining = approved[_owner][_spender];
        return remaining;
    }
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
        if (isBlacklisted(_from)) {
            emit RejectedPaymentFromBlacklistedAddr(_from, _to, _value);
            success = false;
        } else if (isBlacklisted(_to)) {
            emit RejectedPaymentToBlacklistedAddr(_from, _to, _value);
            success = false;
        } else if (allowance(_from, msg.sender) > 0 && allowance(_from, msg.sender) >= _value &&
                   isValidTransfer(_from, _to, _value)) {
            uint256 cashBackRate = uint256(memberships[_to].getCashbackRate(_from));
            uint256 cashback = _value * cashBackRate / 100;
            memberships[_to].updateHistory(_from, _value);
            _value -= cashback;
            emit Cashback(msg.sender, _to, cashback);
            balanceOf[_from] -= _value;
            balanceOf[_to] += _value;
            approved[_from][msg.sender] -= _value;
            emit Transfer(_from, _to, _value);
            success = true;
        } else {
            success = false;
        }
        return success;
    }
}
```

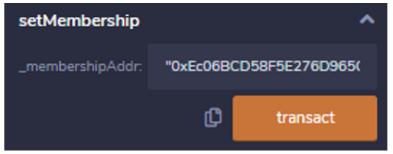
Advanced Token Contract

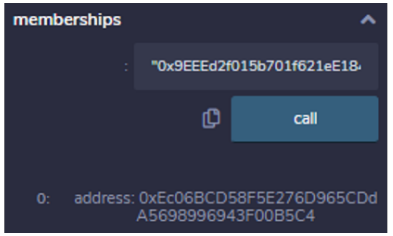
- **Interworking** between token contract and membership contract

1. Deploy Token Contract

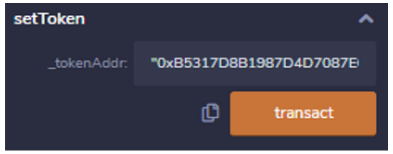


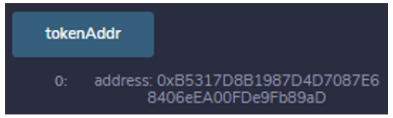
2. Interworking Membership Contract





3. Interworking Token Contract





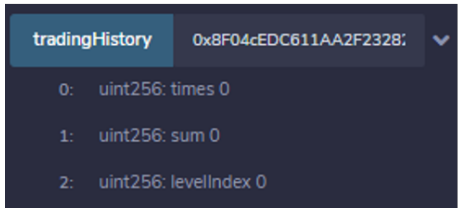
Advanced Token Contract

- Check transaction history and grade changes by transferring money to the membership operation account
  1. Established membership criteria

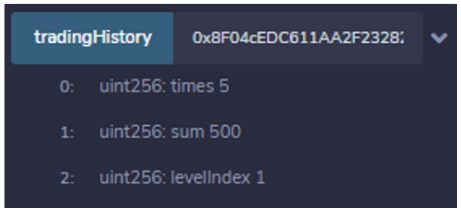
멤버십 등급			
등급	최소 거래 횟수	최소 결제 토큰	캐시백 비율
Gold	10	1500	10
Silver	5	500	5
Basic	0	0	1

  2. transfer 100 token for 5 times

0. Default



1. Grade update (levelIndex 0 → 1)

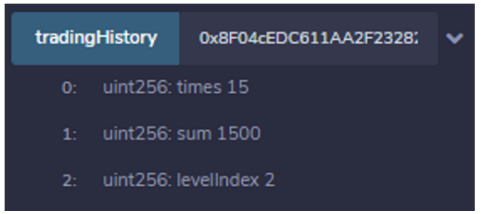


Advanced Token Contract

- Check transaction history and grade changes by transferring money to the membership operation account
  3. transfer 100 token for 10 times

2. Not meet the cumulative transaction amount

3. Meet the cumulative transaction amount



References

- Etherscan. https://etherscan.io/