Lecture 25 - Chain Interoperability - Patterns in languages

Object oriented

Yes! Multiple inheritance.

```
contract SignData is Ownable {
```

Go is not object oriented.

This means constructors.

```
        constructor() {
        }
```

Or with values that it sets

```
        constructor(uint256 _pct) {
        owner_address = payable(msg.sender);
                pct  = _pct; // 1000000 times the yearly percentage rate
                ...
```

These values come from the “migration” JavaScript code.

```
const FixedTermDesposite = artifacts.require("FixedTermDesposite");

module.exports = function (deployer) {
        deployer.deploy(FixedTermDesposite, 20000); // 2% Per Year * 1000000 = 20000
};
```

Data Declaration

```
    address payable owner_address;
    uint256 pct;                                // Payment for Deposite
    uint256 numberOfdays;              // Payment can be withdrawn after X days.

        mapping(address => uint256) nOfDeposites; // Your Deposite ID
```

Data that is not a “constant” is saved from call to call over time. Data is expensive.

Dictionary

Solidity dictionaries - are multi-level maps.

```
        mapping(address => uint256) nOfDeposites; // Your Deposite ID
```

It is important to note that it will return a “default” value for all possible inputs - so a non-existent address will return a 0 in this case.

You can have a map to a map to a value.

```
        mapping(uint256 => mapping(address => uint256)) depositeAmount;
```

The access to these is the same as an array.

```
        ...
        id = nOfDeposites[msg.sender];
        ...
        theOwner = depositeOwner[_id][msg.sender];
        ...
```

Output

Nope.

No output.

Use “events” instead.

Declare an event

```
        event ReceivedFunds(address sender, uint256 value);
```

and generate the event to the log

```
        emit ReceivedFunds(msg.sender, msg.value);
```

In the test code these can be dumped out with

```
var tx = await sd.depositCertificate ( amount, {"value":amount} );
console.log ( "tx=", tx );
console.log ( "tx.logs = ", tx.logs );
for ( var i = 0, mx = tx.logs.length; i < mx; i++ ) {
        if ( tx.logs[i].event == 'DepositeMade' ) {
                console.log ( "For DepositeMoade event tx.logs["+i+"].args = ", tx.logs[i].args );
                var r = tx.logs[i].args;
                console.log ( "    .who = ", r.who );
                console.log ( "    .id = ", r.id.toString() );
                console.log ( "    .amount = ", r.amount.toString() );
                assert.equal(r.id.toString(),"1","Should have an ID of 1");
                assert.equal(r.amount.toString(),"1000000","Should have a depoiste of 1000000");
        }
}
```

Functions/Methods

Functions that “change” the data require “gas”:

```
    ...
function depositCertificate(uint256 _amount) public payable returns ( uint256 ) {
    ...
function withdrawCertificate(uint256 _id) public {
    ...
```

If a function is a “view” then it is local and “free”:

```
    function amountOnDeposite(uint256 _id) public view returns ( uint256 ) {
```

Remember to have functions to do standard things like ‘withdraw’

```
    function withdraw( uint256 amount ) public onlyOwner returns(bool) {
```

Example Fixed Term Deposit

This is like a Certificate of Deposit (CD) at a bank.

```
 1: // SPDX-License-Identifier: MIT
 2: pragma solidity >=0.4.22 <0.9.0;
 3:
 4: import "@openzeppelin/contracts/access/Ownable.sol";
 5:
 6: contract FixedTermDesposite is Ownable {
 7:     address payable owner_address;
 8:     uint256 pct;                // Payment for Deposite
 9:     uint256 numberOfdays;        // Payment can be withdrawn after X days.
10:
11:     mapping(address => uint32) nOfDeposites; // Your Deposite ID
12:     mapping(uint256 => mapping(address => uint256)) depositeAmount;
13:     mapping(uint256 => mapping(address => address)) depositeOwner;
14:     mapping(uint256 => mapping(address => uint256)) depositeDeadline;
15:
16:     event DepositeMade(address indexed who, uint256 amount, uint32 id);
17:     event FundsRemoved(address indexed who, uint256 amount, uint32 id);
18:     event ReceivedFunds(address sender, uint256 value);
19:     event Withdrawn(address to, uint256 amount);
20:
21:     constructor(uint256 _pct) {
22:         owner_address = payable(msg.sender);
23:         pct  = _pct; // 1000000 times the yearly percentage rate
24:         numberOfdays = 365;
25:     }
26:
27:     /**
28:      * @dev Create a new deposite for 1 year.
29:      */
30:     function depositCertificate(uint256 _amount) public payable returns ( uint32 ) {
```

```
30:     function depositCertificate(uint256 _amount) public payable returns ( uint32 ) {
31:         require(msg.value == _amount);
32:         uint32 id = nOfDeposites[msg.sender];
33:         id = id + 1;
34:         nOfDeposites[msg.sender] = id;
35:         depositeAmount[id][msg.sender] = _amount;
36:          depositeOwner[id][msg.sender] = msg.sender;
37:         depositeDeadline[id][msg.sender] = block.timestamp + ( numberOfdays * 1 days);
38:         emit DepositeMade( msg.sender, _amount, id);
39:         return id;
40:     }
41:
42:     /**
43:      * @dev Allow funds to be withdrawn at end of term.
44:      */
45:     function withdrawCertificate(uint32 _id) public {
46:         uint32 id;
47:         id = nOfDeposites[msg.sender];
48:         require(id >= _id && _id > 0);    // check that _id is in range.
49:
50:         address theOwner;
51:         theOwner = depositeOwner[_id][msg.sender];
52:         require(theOwner == msg.sender);                    // You are the owner.
53:         require(block.timestamp >= depositeDeadline[_id][msg.sender]);     // You'r deposite has reached term d
54:
55:         uint256 amount;
56:         amount = depositeAmount[_id][msg.sender];
57:         amount = amount + ( ( amount * pct ) / 1000000 );    // Pay the interest
58:         depositeAmount[_id][msg.sender] = 0;                // 0 left after withdrawl
59:
60:         address payable to;
61:         to = payable(theOwner);                               // convert type to payable
62:
63:         to.transfer(amount);                                // send them the $ plus interest
64:         emit DepositeMade( msg.sender, amount, _id);
65:     }
66:
67:     /**
68:      * @dev Allow funds to be withdrawn at end of term.
69:      */
70:     function amountOnDeposite(uint32 _id) public view returns ( uint256 ) {
71:         uint32 id;
72:         id = nOfDeposites[msg.sender];
73:         if ( id > _id || id <= 0 ) {
74:             return ( 0 );
75:         }
76:
77:         address theOwner;
78:         theOwner = depositeOwner[_id][msg.sender];
79:         if (theOwner != msg.sender) {
80:             return ( 0 );
81:         }
82:
83:         uint256 amount;
84:         amount = depositeAmount[_id][msg.sender];
85:
86:         return ( amount );
87:     }
88:
89:     // ------------------------------------------------------------------------------------------------------
90:
91:     /**
92:      * @dev payable fallback.  The fallback function is called when no other function
93:      * matches (if the receive ether function does not exist then this includes calls
94:      * with empty call data). You can make this function payable or not.  If it is not
95:      * payable then transactions not matching any other function which send value will
96:      * revert
```

```
 96:      * revert.
 97:      */
 98:     fallback() external payable {
 99:         emit ReceivedFunds(msg.sender, msg.value);
100:     }
101:
102:     /**
103:      * @dev payable receive.  The receive ether function is called whenever the call data
104:      *      is empty (whether or not ether is received). This function is implicitly payable.
105:      */
106:     receive() external payable {
107:         emit ReceivedFunds(msg.sender, msg.value);
108:     }
109:
110:     /**
111:      * @dev genReceiveFunds - generate a receive funds event.
112:      */
113:     function genReceivedFunds () public payable {
114:         emit ReceivedFunds(msg.sender, msg.value);
115:     }
116:
117:     /**
118:      * @dev Withdraw contract value amount.
119:      */
120:     function withdraw( uint256 amount ) public onlyOwner returns(bool) {
121:         payable(owner_address).transfer(amount);
122:         // owner_address.send(amount);
123:         emit Withdrawn(owner_address, amount);
124:         return true;
125:     }
126:
127:     /**
128:      * @dev How much do I got?
129:      */
130:     function getBalanceContract() public view onlyOwner returns(uint256){
131:         return address(this).balance;
132:     }
133:
134:     /**
135:      * @dev For futute to end the contract, take the value.
136:      */
137:     function kill() public onlyOwner {
138:         emit Withdrawn(owner_address, address(this).balance);
139:         selfdestruct(owner_address);
140:     }
141: }
```