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    uint256 numberOfdays;              // Payment can be withdrawn after X days.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:
12:
```

```
12:
13:
14:
15:     mapping(address => uint32) nOfDeposites; // Your Deposite ID
16:     mapping(uint256 => mapping(address => uint256)) depositeAmount;
17:     mapping(uint256 => mapping(address => address)) depositeOwner;
18:     mapping(uint256 => mapping(address => uint256)) depositeDeadline;
19:
20:     event DepositeMade(address indexed who, uint256 amount, uint32 id);
21:     event FundsRemoved(address indexed who, uint256 amount, uint32 id);
22:     event ReceivedFunds(address sender, uint256 value);
23:     event Withdrawn(address to, uint256 amount);
24:
25:     constructor(uint256 _pct) {
26:         owner_address = payable(msg.sender);
27:         pct  = _pct; // 1000000 times the yearly percentage rate
28:         numberOfdays = 365;
29:     }
30:
31:     /**
32:      * @dev Create a new deposite for 1 year.
33:      */
34:     function depositCertificate(uint256 _amount) public payable returns ( uint32 ) {
35:         require(msg.value == _amount);
36:         uint32 id = nOfDeposites[msg.sender];
37:         id = id + 1;
38:         nOfDeposites[msg.sender] = id;
39:         depositeAmount[id][msg.sender] = _amount;
40:          depositeOwner[id][msg.sender] = msg.sender;
41:         depositeDeadline[id][msg.sender] = block.timestamp + ( numberOfdays * 1 days);
42:         emit DepositeMade( msg.sender, _amount, id);
43:         return id;
44:     }
45:
46:     /**
47:      * @dev Allow funds to be withdrawn at end of term.
48:      */
49:     function withdrawCertificate(uint32 _id) public {
50:         uint32 id;
51:         id = nOfDeposites[msg.sender];
52:         require(id >= _id && _id > 0);    // check that _id is in range.
53:
54:         address theOwner;
55:         theOwner = depositeOwner[_id][msg.sender];
56:         require(theOwner == msg.sender);                    // You are the owner.
57:         require(block.timestamp >= depositeDeadline[_id][msg.sender]);     // You'r deposite has reached term c
58:
59:         uint256 amount;
60:         amount = depositeAmount[_id][msg.sender];
61:         amount = amount + ( ( amount * pct ) / 1000000 );    // Pay the interest
62:         depositeAmount[_id][msg.sender] = 0;                // 0 left after withdrawl
63:
64:         address payable to;
65:         to = payable(theOwner);                               // convert type to payable
66:
67:         to.transfer(amount);                                // send them the $ plus interest
68:         emit DepositeMade( msg.sender, amount, _id);
69:     }
70:
71:
72:
73:
74:
75:
76:
77:
78:
```

```
 78:
 79:     /**
 80:      * @dev Allow funds to be withdrawn at end of term.
 81:      */
 82:     function amountOnDeposite(uint32 _id) public view returns ( uint256 ) {
 83:         uint32 id;
 84:         id = nOfDeposites[msg.sender];
 85:         if ( id > _id || id <= 0 ) {
 86:             return ( 0 );
 87:         }
 88:
 89:         address theOwner;
 90:         theOwner = depositeOwner[_id][msg.sender];
 91:         if (theOwner != msg.sender) {
 92:             return ( 0 );
 93:         }
 94:
 95:         uint256 amount;
 96:         amount = depositeAmount[_id][msg.sender];
 97:
 98:         return ( amount );
 99:     }
100:
101:     // ----------------------------------------------------------------------------------------------------------
102:
103:     /**
104:      * @dev payable fallback.  The fallback function is called when no other function
105:      * matches (if the receive ether function does not exist then this includes calls
106:      * with empty call data). You can make this function payable or not.  If it is not
107:      * payable then transactions not matching any other function which send value will
108:      * revert.
109:      */
110:     fallback() external payable {
111:         emit ReceivedFunds(msg.sender, msg.value);
112:     }
113:
114:     /**
115:      * @dev payable receive.  The receive ether function is called whenever the call data
116:      *      is empty (whether or not ether is received). This function is implicitly payable.
117:      */
118:     receive() external payable {
119:         emit ReceivedFunds(msg.sender, msg.value);
120:     }
121:
122:     /**
123:      * @dev genReceiveFunds - generate a receive funds event.
124:      */
125:     function genReceivedFunds () public payable {
126:         emit ReceivedFunds(msg.sender, msg.value);
127:     }
128:
129:     /**
130:      * @dev Withdraw contract value amount.
131:      */
132:     function withdraw( uint256 amount ) public onlyOwner returns(bool) {
133:         payable(owner_address).transfer(amount);
134:         // owner_address.send(amount);
135:         emit Withdrawn(owner_address, amount);
136:         return true;
137:     }
138:
139:
140:
141:
142:
143:
144:
```

```
144:
145:
146:     /**
147:      * @dev How much do I got?
148:      */
149:     function getBalanceContract() public view onlyOwner returns(uint256){
150:         return address(this).balance;
151:     }
152:
153:     /**
154:      * @dev For fututе to end the contract, take the value.
155:      */
156:     function kill() public onlyOwner {
157:         emit Withdrawn(owner_address, address(this).balance);
158:         selfdestruct(owner_address);
159:     }
160: }
```