Blockchain – Smart Contracts – 2
(Aug 22, 2019)

Dr. Deven Shah

TCET Mumbai

Smart Contract

- Solidity's code is encapsulated in contracts.
- A contract is the fundamental building block of Ethereum applications —
- All variables and functions belong to a contract, and this will be the starting point of all your projects.

```
contract Employee {
}
```

Version Pragma

- All solidity source code should start with a "version pragma“
- A declaration of the version of the Solidity compiler this code should use. This is to prevent issues with future compiler versions potentially introducing changes that would break your code.
- It looks like this:

```
pragma solidity ^0.4.19;
contract Employee {
}
```

State Variables and Integers

- State variables are permanently stored in contract storage. This means they're written to the Ethereum blockchain.

```
uint32 empId;
string name;
string location;
uint32 level;
```

- The uint data type is an unsigned integer, meaning its value must be non-negative. There's also an int data type for signed integers.
- Note: In Solidity, uint is actually an alias for uint256, a 256-bit unsigned integer. You can declare uints with less bits — uint8, uint16, uint32, etc.

Structs – Complex data type

```
struct Emp {
        uint32 empId;
        string name;
        string location;
        uint32 level;
        uint favNumber;
}
```

Arrays

- There are two types of arrays in Solidity: fixed arrays and dynamic arrays:

```
// Array with a fixed length of 2 elements:
uint[2] fixedArray;

// a dynamic Array - has no fixed size, can keep growing:
uint[] dynamicArray;
```

You can also create an array of structs

```
Emp[] public employees;
```

Public Arrays

You can declare an array as public, and Solidity will automatically create a getter method for it. So other contract can read from this array (not to write).

Working with Structs and Arrays

Emp[] public employees;

How to add records in Arrays of structs

Create a new employee

Emp deven = Emp(121,deven,ghatkoper,0,16)

employees.push(deven);

You can combine

employees.push(Emp(121,deven,ghatkoper,0,16);

Function Declarations

```
function _createEmployee(uint32 _empId, string
_name, string _location, uint _favNumber)
internal
{
//function body
}
```

Private / Public Functions

- Default : Public : Any other contract can call public contract's function and execute its code.
- May not be always desirable.
- Makes contract vulnerable to attacks.
- Good practice: Private : can be called by only other function within contract.
- Normally Private function starts with _.

Return Values

Return Value

```
function _generateRandomFavNumber(string _str) private view returns (uint) {
            uint fav = uint(keccak256(_str));
            return fav % favModulus;
}
```

Multiple Return

```
function multipleReturns() internal returns(uint, uint, uint) {
  return (1, 2, 3);
}
```

How to assign multiple return

```
 (a, b, c) = multipleReturns();
```

Visibility Modifiers

- Control when and where the function can be called from

VISIBILITY	ACCESSIBIITY
Private	Current contract
Internal	Current contract and derived contract
External	Other contract
Public	All contracts

Function modifiers

- How the function interacts with Blockchain

STATE MODIFIERS	DESCRIPTION
View	Only read, no write
Pure	No read, no write

Events

Events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen.

```
event IntegersAdded(uint x, uint y, uint result);

function add(uint _x, uint _y) public {
  uint result = _x + _y;
 // fire an event to let the app know the function was called:
  IntegersAdded(_x, _y, result);
  return result;
}
```

Your app front-end could then listen for the event. A javascript implementation would look something like:

```
YourContract.IntegersAdded(function(error, result) {
  // do something with result
}
```

Mapping and Addresses

- To provide owner to any data in blockchain
  - Mapping and address are two data types
  - Each account in blockchain has an address and its own by specific user and smart contract.
  - We want to give ownership to data created by user by calling a function.

```
// For a financial app, storing a uint that holds the user's account balance:
mapping (address => uint) public accountBalance;

// Or could be used to store / lookup usernames based on userId
mapping (uint => string) userIdToName;
```

A mapping is essentially a key-value store for storing and looking up data. In the first example, the key is an address and the value is a uint, and in the second example the key is a uint and the value a string.

msg.sender

- msg.sender is a global variable that are available to all functions. It refers to the address of the person or smart contract, who called the current function.
- In solidity, function execution always needs to start with an external caller.
- A contract will just sit on the blockchain doing nothing until somemone calls one of its function. So there is always a msg.sender

```
mapping (address => uint) favoriteNumber;

function setMyNumber(uint _myNumber) public {
// Update `favoriteNumber` mapping to store `_myNumber` under `msg.sender`
favoriteNumber[msg.sender] = _myNumber;
}

function whatIsMyNumber() public view returns (uint) {
  // Retrieve the value stored in the sender's address
  // Will be `0` if the sender hasn't called `setMyNumber` yet
  return favoriteNumber[msg.sender];
}
```

require

```
modifier onlyOneEmployee() {
require(ownerEmployeeCount[msg.sender] == 0);
_;
}
```

- Set a condition at starting of function and if not true it will stop executing.