블록체인 프로젝트 코드

-블록체인 시스템보안-



이름:박지원

Pragma solidity ^0.4.13; -> 솔리디티 버전

Contract 컨트랙트 명 {

}

* 컨트랙트 초기 뼈대

**상태 변수** -> 컨트랙트 저장소에 영구적 저장 : 즉 이더리움 블록체인에 기록됨 (데이터베이스에 데이터를 쓰는 것과 동일)

Contract Example {

Unit myUnsignedInterger = 100;

}

* myUnsignedInteger 라는 uint 생성 후 100 값 배정

부호 없는 정수 자료형 uint : 값이 음수가 아니어야함.

부호 있는 정수의 경우 자료형 int

Uint의 경우 256, 8, 16, 32등 비트를 선언할 수도 있음

**수학 연산의 경우**

덧셈 +

뺄셈 –

곱셈 \*

나눗셈 /

나머지 %

제곱(지수) \*\*

**구조체**

Struct Person{

Uint age;

String name;

}

* 구조를 통해 보다 복잡한 자료형 생성 가능
* String은 임의의 길이를 가진 UTF-8 데이터

**배열**

정적 배열과 동적 배열로 나뉨

uint[2] fixedArray; : 고정길이배열

string[5] stringArray; : 스트링배열

uint[] dynamicArray; : 동적배열 -> 원소 계속 추가 가능

public 으로 선언 가능 -> person[] public people;

**함수선언**

function eatHamburgers(string \_name, uint \_amount) {

}

String 인자와 uint 인자 (함수 인자명을 언더스코어로 시작해서 전역변수와 구별)

eatHamburgers("vitalik", 100);

: 함수 호출

Pragma solidity ^버전;

Contract HealthMembership{

Struct Member{

Address addr;

Uint money;

Bool status;

}

**구조체 생성**

Struct Person {

Uint age;

String name;

}

Person[] public people;

이 배열에 새로운 people을 추가할 경우

Person satoshi = Person(172, “Satoshi”); 🡪 새로운 사람 생성

People.push(satoshi); 🡪 이사람 배열에 추가

로 할 수 있는데 이 코드를 한줄로 people.push(Person(16, “Vitalik”)); 으로 표현 가능

Push의 경우 배열 끝에 추가해서 순서가 유지됨.

**Private / Public 함수**

함수는 기본적으로 public 선언 🡪 누구나 함수 호출, 코드 실행 가능

키워드는 함수명에 적음. -> private 함수명의 경우 \_ 로 시작함

**반환값**

함수에서 반환값을 얻으려면

string greeting = "What's up dog";

function sayHello() public returns (string) {

return greeting;

}

이와 같이 선언해야함 -> 이 함수의 경우 반환값은 string임

**함수 제어자**

View를 통해 함수 데이터를 보기만 하고 변경하지 않음을 의미

Pure를 통해 어떤 데이터도 접근하지 않는다는 것을 의미

Internal를 통해 상속하는 컨트랙트에서도 접근 가능한 private

Extrenal 를 통해 컨트랙트 바깥에서만 호출될 수 있는 public

**랜덤값과 형변환**

함수의 반환값이 랜덤인 uint가 되길 원한다면 keccak256를 통해 사용할 수 있음( 의사 난수 발생기)

keccak256("aaaab"); -> 이와 같은 형식으로 사용함

형변환

uint8 a = 5;

uint b = 6;

이의 경우 에러가 발생할 수 있음.

Uint8 c = a \* uint8(b); 와 같이 형변환을 해주어야함.

**이벤트**

블록체인 상에서 액션이 발생할 경우 의사소통하는 방식으로 이벤트 발생시 행동 취함

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

function add(uint \_x, uint \_y) public {

uint result = \_x + \_y;

// 이벤트를 실행하여 앱에게 add 함수가 실행되었음을 알린다:

IntegersAdded(\_x, \_y, result);

return result;

}

**Web3.js**

컨트랙트 작성 완료시 컨트랙트와 상호작용하는 자바 스크립트 코드를 작성해야함.

자바 스크립트 코드 완성시 컨트랙트를 불러와 웹 브라우저 기반 js를 통해 이미지 변경 후 css 필터를 적용함

매핑과 주소

주소 -> 주소는 특정 유저가 소유한다 : 고유 ID임

매핑 -> 구조화된 데이터를 저장하는 또 다른 방법

키-값 저장소로 데이터를 저장하고 검색하는데 이용됨

mapping (address => uint) public accountBalance;

예시의 키는 address이고 값은 uint 임

**Msg.sender**

현재 함수를 호출한 사람의 주소를 가리키는 것

이를 사용해 블록체인의 보안성을 이용할 수 있음

**Require**

특정 조건이 참이 아닐 때 함수가 에러 메시지를 발생하고 실행을 멈춤.

require(조건)

**상속**

컨트랙트 상속은 여러 컨트랙트에 코드 로직을 나눌 때 유용

Cotract Doge {

~~~~

}

Contract BabyDoge is Doge{

~~~~

}

와 같은 방식으로 상속받을 수 있음

**Import**

어떤 파일을 다른 파일로 불러오고 싶을 때 이용

Import “./someothercontract.sol”;

와 같은 방식으로 불러오고 상속할 수 있음.

**Storage vs Memory**

변수를 저장할 수 있는 공간

Storage 는 블록체인 상에 영구적으로 저장됨

Memory는 임시적으로 저장되는 변수 -> 컨트랙트 함수에 대한 외부 호출들이 일어나는 사이 지워짐

보통은 솔리디티가 알아서 처리해주기 때문에 입력할 필요가 없으나 함수 내 구조체와 배열을 처리할 때는 사용함

**다른 컨트랙트와 상호작용**

getNum의 경우 다른 컨트랙트와 상호작용 하고자 하는 함수만 선언

함수 몸체를 정의하지 않고 {} 를 사용하지 않으며 함수 선언을 ; 로 함

**다수의 반환값 처리**

(a,b,c) = multipleReturns(); 를 통해 다수의 값을 할 수 있다. 만약 하나의 값만 하고 싶은 경우 빈 자리는 공백으로 둔다.

pragma solidity ^0.4.19;

contract ZombieFactory {

event NewZombie(uint zombieId, string name, uint dna);

uint dnaDigits = 16;

uint dnaModulus = 10 \*\* dnaDigits;

struct Zombie {

string name;

uint dna;

}

Zombie[] public zombies;

mapping (uint => address) public zombieToOwner;

mapping (address => uint) ownerZombieCount;

function \_createZombie(string \_name, uint \_dna) private {

uint id = zombies.push(Zombie(\_name, \_dna)) - 1;

zombieToOwner[id] = msg.sender;

ownerZombieCount[msg.sender]++;

NewZombie(id, \_name, \_dna);

}

function \_generateRandomDna(string \_str) private view returns (uint) {

uint rand = uint(keccak256(\_str));

return rand % dnaModulus;

}

function createRandomZombie(string \_name) public {

require(ownerZombieCount[msg.sender] == 0);

uint randDna = \_generateRandomDna(\_name);

\_createZombie(\_name, randDna);

}

}

pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract KittyInterface {

function getKitty(uint256 \_id) external view returns (

bool isGestating,

bool isReady,

uint256 cooldownIndex,

uint256 nextActionAt,

uint256 siringWithId,

uint256 birthTime,

uint256 matronId,

uint256 sireId,

uint256 generation,

uint256 genes

);

}

contract ZombieFeeding is ZombieFactory {

address ckAddress = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;

KittyInterface kittyContract = KittyInterface(ckAddress);

function feedAndMultiply(uint \_zombieId, uint \_targetDna, string \_species) public {

require(msg.sender == zombieToOwner[\_zombieId]);

Zombie storage myZombie = zombies[\_zombieId];

\_targetDna = \_targetDna % dnaModulus;

uint newDna = (myZombie.dna + \_targetDna) / 2;

if (keccak256(\_species) == keccak256(“kitty”)){

newDna = newDna - newDna % 100 + 99;

}

\_createZombie("NoName", newDna);

}

function feedOnKitty(uint \_zombieId, uint \_kittyId) public {

uint kittyDna;

(,,,,,,,,,kittyDna) = kittyContract.getKitty(\_kittyId);

feedAndMultiply(\_zombieId, kittyDna, “kitty”);

}

}