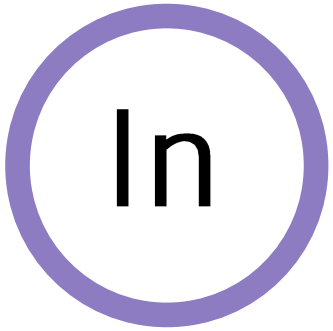
이더리움 DApp

Solidity 언어 실습



Solidity

04 Crypto Zombie(3)

Crypto Zombie Lesson3

Lesson1에선..

ZombieFactory 생성
- 모든 좀비를 기록하도록 state설정
- 좀비 생성 함수
- 각 좀비는 DNA를 통해 랜덤한 외모 설정

Lesson2에선..

- **게임기능**
  - 새로운 좀비 생성 기능
  - 좀비가 다른 좀비를 공격하고 먹을 수 있도록

Lesson3에선..

- **컨트랙트 생성 테크닉**
- **의존성 관리하기**
- **컨트랙트 보안**
- **가스 최적화 및 modifier**
- **external view와 반복문**

자세한 사항은 링크 참조
https://cryptozombies.io/ko/lesson/3/chapter/1

04 Crypto Zombie(3)

Chapter1

목표: 외부의존성 이해 및 EVM 컨트랙트의 특성 이해하기

- 현재 우리의 ZombieContract는 kittyInterface 네트웨크를 이용하고 있음
→ 의존성을 가지고 있음

** 만약?
- kitty Contract의 버그가 있다면?
- kitty가 파괴된다면?...

→ 우리의 앱도 그대로 버그가 존재할 것이다.

실습

크립토키티의 컨트랙트 주소를 업데이트 가능하도록

→ 상태 변수를 변경가능하도록 컨트랙트를 만들자!

1. 직접 주소를 써넣었던 ckAddress 부분을 지운다.
2. kittyInterface를 생성했던 줄을 변수 선언만 하도록 변경(값 대입 X)
3. setKittyContractAddress (address _address) 함수 생성 외부에서 호출받아야 하므로 (OO으로 선언(private vs external vs public vs internal)
4. 함수내에서 kittyInterface에 직접 값을 대입!

```
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
);
}
```

04 Crypto Zombie(3)

Chapter2

목표: 보안을 고려한 개발 // external함수의 이해 // 생성자 // modifier
추가내용: 다음 슬라이드

- 방금 작성한 setKittyContractAddress 함수는 external로 작성되어 있다
  → 누구나 다 해당 함수를 호출할 수 있다.
  → 누구나 다 kittcyContractAddress를 변경할 수 있다?!
- → 문제 발생

- 컨트랙트를 소유 가능하게 만들자!

OpenZeppelin Ownable Library
→ https://openzeppelin.com/contracts/

truffle**에서 해당 라이브러리를 사용하려면?**

```
npm install @openzeppelin/contracts@2.5
```

실습

Ownable 컨트랙트를 사용할 수 있도록 하자!
→ ZombieFactory 컨트랙트가 Ownable 함수를 사용하도록 하자 ➔ 상속
→ ownable함수의 위치는 “@openzeppelin/contracts/ownership/Ownable.sol”

```
/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;
  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  function Ownable() public {
    owner = msg.sender;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }
}
```

이전 버전 Ownable 라이브러리
→ 현재는 업데이트 되었음.
→ 소스코드 확인 (node_modules)

04 Crypto Zombie(3)

```sol
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Returns true if the caller is the current owner.
     */
    function isOwner() public view returns (bool) {
        return _msgSender() == _owner;
    }
```

생성자

modifier

Ownable.sol

```sol
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions anymore. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby removing any functionality that is only available to the owner.
 */
function renounceOwnership() public onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public onlyOwner {
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 */
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Ownable.sol

04 Crypto Zombie(3)

생성자

```
constructor () internal {
    address msgSender = _msgSender();
    _owner = msgSender;
    emit OwnershipTransferred(address(0), msgSender);
}
```

- constructor(생성자)
- 컨트랙트가 생성될 때 한 번 실행되는 함수
- 생략할 수 있음

04 Crypto Zombie(3)

Function Modifier(함수 제어자)

```
modifier onlyOwner() {
    require(isOwner(), "Ownable: caller is not the owner");
    _;
}
```

- 함수 제어자(Function Modifier)
- 제어자는 다른 함수들에 대한 접근을 제어하기 위해 사용되는 일종의 유사 함수.
- 보통 함수 실행 전의 요구사항 충족 여부를 확인하는 데에 사용하지.
- onlyOwner의 경우에는 접근을 제한해서 오직 컨트랙트의 소유자만 해당 함수를 실행할 수 있도록 하기 위해 사용
- _;
  modifier가 통과되면 다음 함수부를 실행하여라(함수로 돌아가서 해당 코드 실행)

04 Crypto Zombie(3)

Function Modifier(함수 제어자)

```
/**
 * @dev Throws if called by any account other than the owner.
 */
modifier onlyOwner() {
  require(msg.sender == owner);
  _;
}
```

```
contract MyContract is Ownable {
  event LaughManiacally(string laughter);

  // 아래 `onlyOwner`의 사용 방법을 잘 보게:
  function likeABoss() external onlyOwner {
    LaughManiacally("Muahahahaha");
  }
}
```

- onlyOwner가 통과되어야 다음 코드 실행 가능

04 Crypto Zombie(3)

Chapter3

목표: 함수제어자와 onlyOwner이해하기

- ZombieFactory가 Ownable을 상속 → ZombieFeeding에서도 이용 가능

실습

1. setKittyContractAddress를 Owner만 수정할 수 있도록 변경

Chapter4

목표: 구조체에 데이터 추가

- 복습: struct를 사용하면 데이터를 압축하여 저장할 수 있다? (True vs False)??
- 데이터가 압축되면 가스 절약이 가능하다

실습

크립토키티의 컨트랙트 주소를 업데이트 가능하도록

→ 상태 변수를 변경가능하도록 컨트랙트를 만들자!

1. ZombieFactory에서 Zombie구조체에 두가지 속성 추가
   level(uint32), readyTime(uint32)
- level(좀비의 레벨), readyTime(다른 좀비를 공격할 수 있는 대기시간)

04 Crypto Zombie(3)

Chapter5

목표: 시간 단위 이해하기

- 솔리디티는 시간을 다룰 수 있는 단위계 존재

- now: 유닉스 타임스탬프 값 얻을 수 있음
- seconds: 초
- minutes: 분
- hours, days, weeks, years, 같은 단위계 포함 → 해당 단위계는 해당하는 길이만큼 uint 숫자로 반환
  1 minutes → 60, 1 hours → 3600(60*60)

```
uint lastUpdated;

// `lastUpdated`를 `now`로 설정
function updateTimestamp() public {
  lastUpdated = now;
}

// 마지막으로 `updateTimestamp`가 호출된 뒤 5분이 지났으면 `true`를, 5분이 아직 지나지 않았으면 `false`를 반환
function fiveMinutesHavePassed() public view returns (bool) {
  return (now >= (lastUpdated + 5 minutes));
}
```

실습

- 재사용 대기시간(1일) 추가

1. uint coolDownTime 선언 및 1 days라고 할당.
2. Zombie 구조체 업데이트→ _createZombie 함수 업데이트
3. level에는 1 할당, readyTime은 (현재시간에 cooldownTime 더해서 할당)
   ** 주의: readyTime의 DataType 확인 ** now는 uint256 타입임.

Chapter6

목표: 구조체를 인수로 전달하기

- 우리의 좀비는 이제 readyTime 존재!
- zombieFeeding.sol 수정 → 재사용 대기시간 구현하기

```
function _doStuff(Zombie storage _zombie) internal {
  // _zombie로 할 수 있는 것들을 처리
}
```

- 구조체 변수를 인자로 전달 할 수 있음(storage는 포인터-실제 값 변경! memory는 값 복사 –실제 값 변경 X)

실습

- 재사용 대기시간(1일) 추가

1. _triggerCooldown 함수 선언! (인자는 Zombie storage _zombie) internal 함수
2. 함수 내용은 현재시간 + 쿨다운 시간으로 zombie의 _readyTime를 업데이트 하여라

3. _isReady함수 선언! 인자는 Zombie storage _zombie) internal view 함수
4. 함수 내용은 현재시간이 zombie의 readyTime보다 큰지를 return
   ** 주의: 함수의 return값?.

04 Crypto Zombie(3)

Chapter7

목표: 이더리움 보안 이해하기

- public, external은 누구나 접근하고 호출할 수 있다는 점에 주의!
- 이를 방지하려면 onlyOwner등 함수 제어자 필요!
- 현재 feedAndMultiply → 현재 public 함수→ 누구나 접근 가능
  → 사용자들 누구나 해당 컨트랙트를 직접 호출하여
  → species와 targetDNA를 호출 할 수 있다.

실습

- feedAndMultiply 수정
- 함수 내부에서 (feedOnKitty 등)에서만 호출 되도록!

1. 함수 feedAndMultiply를 internal로 변경하여라
2. feedAndMultiply 함수가 cooldownTime을 고려하도록 만들어보자.
   **힌트: myZombie를 찾은 후에, _isReady()를 확인하는 require 문장을 추가 → 거기에 myZombie를 전달. 재사용 대기 시간이 끝난 다음에만 이 함수를 실행 가능.
3. 함수의 끝에서 _triggerCooldown(myZombie) 함수를 호출 → 좀비의 재사용 대기 시간 설정

Chapter8

목표: 함수 제어자 이해하기

- 좀비들이 특정 level에 도달하면 할 수 있는 일들을 명시하고 싶다.
- 이를 위해서 if문을 함수들 내에서 자주 사용하면?
  → 모든 함수마다 if문을 작성해서 가독성이 떨어짐.
  → 함수제어자에 인수를 전달하자

```
// 사용자의 나이를 저장하기 위한 매핑
mapping (uint => uint) public age;

// 사용자가 특정 나이 이상인지 확인하는 제어자
modifier olderThan(uint _age, uint _userId) {
  require (age[_userId] >= _age);
  _;
}

// 차를 운전하기 위하서는 16살 이상이어야 하네(적어도 미국에서는).
// `olderThan` 제어자를 인수와 함께 호출하려면 이렇게 하면 되네:
function driveCar(uint _userId) public olderThan(16, _userId) {
  // 필요한 함수 내용들
}
```

- olderThan 함수 제어자가 함수와 같이 인자 받을 수 있음
- 제어자도 재사용이 가능하도록 인자를 받을 수 있음!

실습

- 재사용 대기시간(1일) 추가

1. ZombieHelper.sol 생성 후 ZombieFeeding 상속
2. ZombieHelper에서, aboveLevel이라는 이름의 modifier 생성.
   이 제어자는 _level(uint), _zombieId(uint) 두 개의 인수.
3. 함수 내용에서는 zombies[_zombieId].level이 _level 이상인지 확인.
   주의: **_; 의 역할 확인!**

04 Crypto Zombie(3)

Chapter9

목표: 함수 제어자 사용하기

- 앞에서 만든 aboveLevel을 사용하여 함수를 정의하자!

사용예시코드

```
// 사용자의 나이를 저장하기 위한 매핑
mapping (uint => uint) public age;

// 사용자가 특정 나이 이상인지 확인하는 제어자
modifier olderThan(uint _age, uint _userId) {
  require (age[_userId] >= _age);
  _;
}

// 차를 운전하기 위하서는 16살 이상이어야 하네(적어도 미국에서는).
// `olderThan` 제어자를 인수와 함께 호출하려면 이렇게 하면 되네:
function driveCar(uint _userId) public olderThan(16, _userId) {
  // 필요한 함수 내용들
}
```

실습

1. changeName(uint _zombieId, string _newName)이라는 external 함수.
   → 2레벨 이상인 좀비만 사용 가능하도록 구성
2. 함수 내용은 Tx 요청자와 zombie의 주인이 같은지 검증
   → 힌트: require과 msg.sender
3. 이름바꾸기 기능 추가. 전달받은 _newName으로 사용

4. changeDna 함수 정의 → 인자 2개 (??), 접근제어자 (??), Level이 20이상일때만 가능하도록! (힌트: changeName과 비슷)
5. 함수 내용은 Tx요청자와 zombie 주인 같은지 검증하고 dna를 변경하게 함.

Chapter10 & Chapter11

목표: view 함수 사용하기

- **우리는 사용자의 전체 좀비를 볼 수 있는 메소드가 필요!**
- **+ view 함수는 가스를 소모하지 않는다**
- view 함수를 쓰는 것은 블록체인 상에 실제로 어떤 것도 수정 X
- view 함수를 쓰는 것은 web3.js(이더리움과 통신을 도와주는 클라이언트 library)에게 이렇게 말하는 것과 같음
  "이 함수를 실행할 때, 전체 네트워크 검증이 아니라 너의 local 이더리움 노드에서 꺼내만 줘!)
- 실제로 Transaction 만들지 않으므로 검증도 필요없고, 가스를 소모하지 않음

➔ 외부에서 접근하여 조회만 하는 것은?! → external view로 선언
→ TX 로직중에 view가 호출되는 것은 여전히 가스를 소모할 것(검증이 필요하기 때문)
- 메모리는 함수가 끝날때 없어짐. (그러므로 view 함수는 당연히 가스가 들지 않음)

```
function getArray() external pure returns(uint[]) {
  // 메모리에 길이 3의 새로운 배열을 생성한다.
  uint[] memory values = new uint[](3);
  // 여기에 특정한 값들을 넣는다.
  values.push(1);
  values.push(2);
  values.push(3);
  // 해당 배열을 반환한다.
  return values;
}
```

메모리에 배열 선언 방법

실습

- 사용자의 zombie를 반환하는 함수 추가

1. getZombiesByOwner(address _owner)라는 이름의 함수 선언
   → (조회기능만 가질 예정, 가스 최적화 해야함)
2. 이 함수는 반환값이 uint 배열 형태가 될 예정 (uint[])

3. 함수 내용: result라는 uint배열 형태의 변수를 생성 ([storage vs memory] 설정 필요)
   → uint[] <memory vs storage> result = <_owner가 소유한 좀비의 수>

04 Crypto Zombie(3)

Chapter10 & Chapter11

목표: for 반복문 이해하기

우리는 사용자의 전체 좀비를 볼 수 있는 메소드가 필요!

- **+ 전체 좀비를 우리가 코드로서 일일이 검증하는 것은 무리가 있음. (좀비의 개수만큼 비교 해야 함.)**

```
function getEvens() pure external returns(uint[]) {
  uint[] memory evens = new uint[](5);
  // 새로운 배열의 인덱스를 추적하는 변수
  uint counter = 0;
  // for 반복문에서 1부터 10까지 반복함
  for (uint i = 1; i <= 10; i++) {
    // `i`가 짝수라면...
    if (i % 2 == 0) {
      // 배열에 i를 추가함
      evens[counter] = i;
      // `evens`의 다음 빈 인덱스 값으로 counter를 증가시킴
      counter++;
    }
  }
  return evens;
}
```

짝수 찾기 반복

실습

- for 반복문을 써서 getZombiesByOwner 함수 작성!
- 우리의 모든 좀비들에 접근 + 소유자 확인 + result 배열에 추가 후 반환

** hint: 배열.length(배열의 길이를 반환)

1. uint count 변수 선언(값: 0)
2. 좀비가 찾는 좀비가 맞으면 result배열에 추가

04 Crypto Zombie(3)

lesson3 마무리 실습

solidity >=0.5
** ZombieFactory의 모든 함수의 string형 인자를 memory로 받도록 명시하자!
→ 내부에서 사용될 함수의 string, array, 구조체는 (memory 또는 storage 선언!)
→ 외부에서 사용될 함수는 calldata 선언

```
function changeName(uint256 _zombieId, string calldata _newName)
    external
    aboveLevel(2, _zombieId)
{
    require(msg.sender == zombieToOwner[_zombieId]);      You, 3 mi
    zombies[_zombieId].name = _newName;
}
```