CMSC 398F
Week #13
Solidity Continuation

•••

Announcements

- Quiz 5 should actually be released today.
    - Covers all topics from Lesson 3 → Lesson 5
        - Specifically information on cryptozombies.io
- **IMPORTANT:** NO Final Project. Final Exam will be released next Friday 12/9 Noon and will be due by Sunday 12/11 night 11:59 PM.

From Last Time

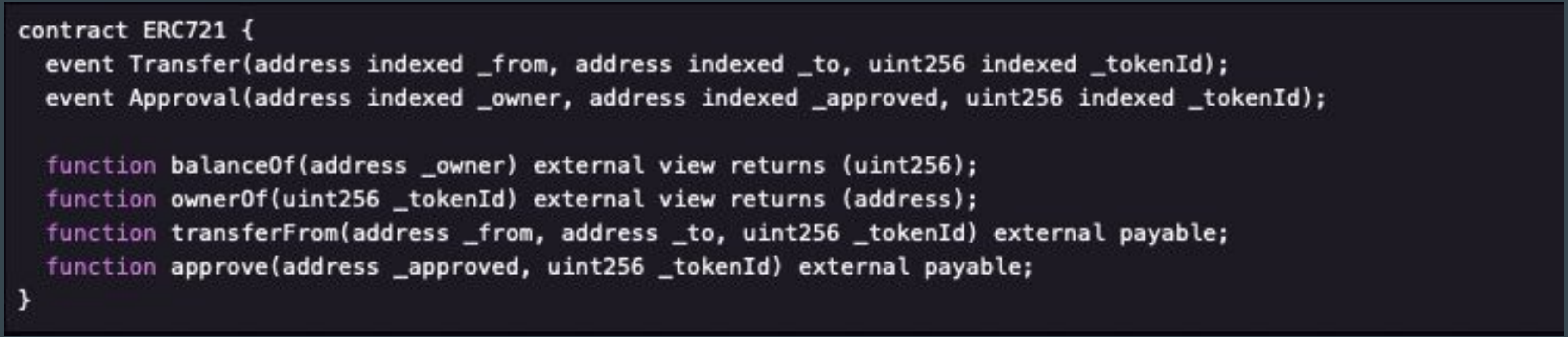
- Storage vs. Memory
- Require Keyword
- Interfaces and Inheritance
- Ownership
  - OnlyZepplin onlyOwner modifier
- Payable
- View and Pure
- Gas Fees
- ERC721 (NFT)

Tokens on Ethereum

- A token on Ethereum is just a smart contract that follows some common rules
  - transferFrom(address _from, address _to, uint256 _tokenId)
  - balanceOf(address _owner)
- Internally the contract also has a mapping of addresses to amounts, to keep track of how many tokens each user has
- ERC20 is the technical standard for fungible tokens created on the Ethereum blockchain
- Today we will be implementing ERC721
  - ERC721 is the standard for non-fungible tokens
  - Unlike ERC20, ERC721 tokens are unique, so not all of them are interchangeable
  - Also, they are not divisible, so you can only trade them in whole units.

ERC721 Standard, Multiple Inheritance

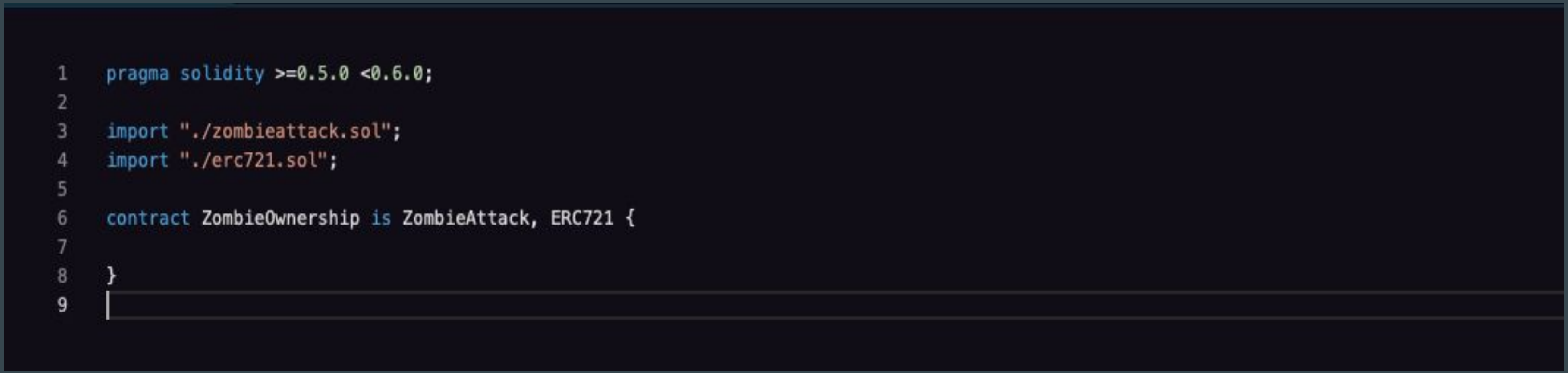
- Below is the ERC721 standard, so we need to implement all of these methods in our contract



```
contract ERC721 {
  event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);
  event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId);

  function balanceOf(address _owner) external view returns (uint256);
  function ownerOf(uint256 _tokenId) external view returns (address);
  function transferFrom(address _from, address _to, uint256 _tokenId) external payable;
  function approve(address _approved, uint256 _tokenId) external payable;
}
```

ERC721 Standard, Multiple Inheritance

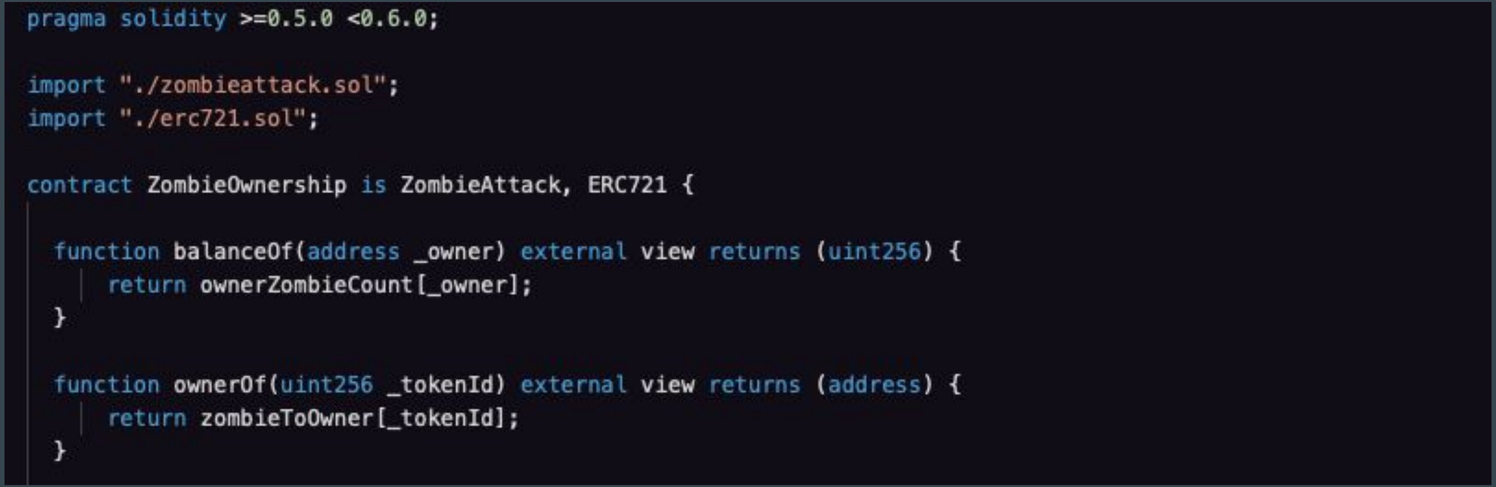
- First, we need to copy ERC 721 interface to its own file and then import it into our contract



```
pragma solidity >=0.5.0 <0.6.0;

import "./zombieattack.sol";
import "./erc721.sol";

contract ZombieOwnership is ZombieAttack, ERC721 {

}

```

balance & ownerOf

- Let’s start by implementing balance and ownerOf
  - Balance takes in an address as a parameter, and returns the number of tokens (zombies) owned by this address
  - ownerOf takes in a tokenId, and returns an address (the owner of the token)



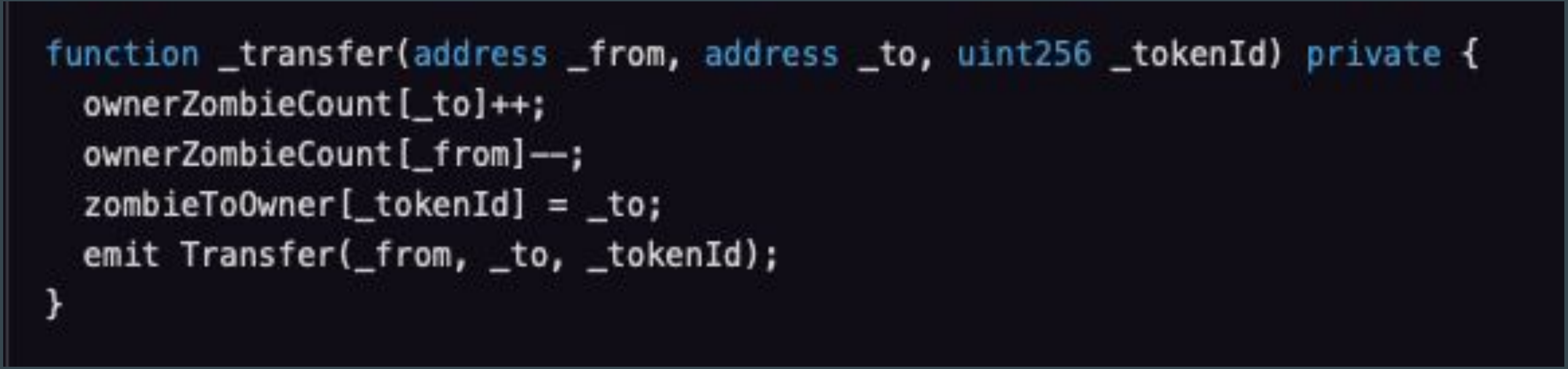
```
pragma solidity >=0.5.0 <0.6.0;

import "./zombieattack.sol";
import "./erc721.sol";

contract ZombieOwnership is ZombieAttack, ERC721 {

  function balanceOf(address _owner) external view returns (uint256) {
    return ownerZombieCount[_owner];
  }

  function ownerOf(uint256 _tokenId) external view returns (address) {
    return zombieToOwner[_tokenId];
  }
```

ERC721 Transfer Logic

- ERC721 has 2 different ways to transfer tokens
  - `function transferFrom(address _from, address _to, uint256 _tokenId) external payable;`
  - `function approve(address _approved, uint256 _tokenId) external payable;`
- The first way is the tokens owner calls transferFrom with the specified parameters
- In the second way, the token owner first calls approve with the address they want to transfer to, and the tokenID. The contract stores who is approved to take the token. Then, the owner or the approved address calls transferFrom, and the contract checks that the `msg.sender` is the owner or approved address.
- Let's abstract this logic into its own private function _transfer, which is then called by transferFrom

ERC721 Transfer Logic

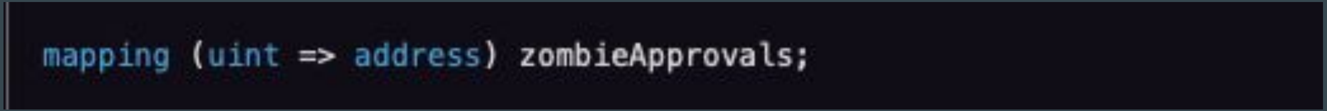
- Let’s define logic for _transfer
  - It will take 3 arguments: _address from, _address to, uint256 _tokenId, and should be private
  - When ownership changes, 2 mappings should change: ownerZombieCount (keeps track of how many zombies someone has) and zombieToOwner (keeps track of who owns what)
  - ERC721 specifies that this function should trigger a Transfer event, so we will do that as well



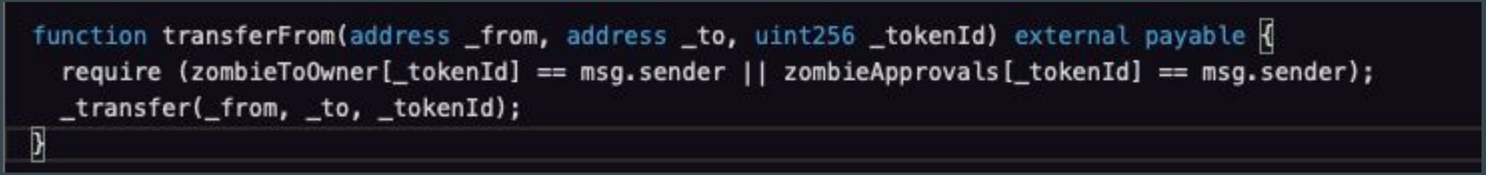
```
function _transfer(address _from, address _to, uint256 _tokenId) private {
  ownerZombieCount[_to]++;
  ownerZombieCount[_from]--;
  zombieToOwner[_tokenId] = _to;
  emit Transfer(_from, _to, _tokenId);
}
```

ERC721 Transfer Logic

- Now we must implement the external transferFrom function, which is required by ERC721
  - We must make sure only the owner or an approved address of a token/zombie can transfer it
    - We can create a mapping of uint -> address. We can use this to quickly lookup if someone is approved to take that token
  - Lastly, we will call transfer



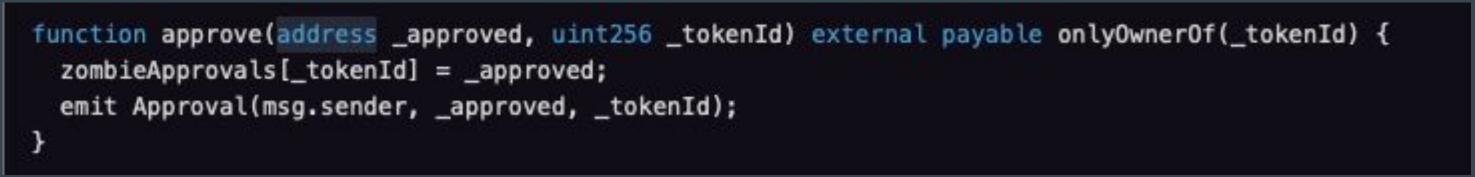
```
mapping (uint => address) zombieApprovals;
```



```
function transferFrom(address _from, address _to, uint256 _tokenId) external payable {
  require (zombieToOwner[_tokenId] == msg.sender || zombieApprovals[_tokenId] == msg.sender);
  _transfer(_from, _to, _tokenId);
}
```

Implementing Approve

- Remember that with approve the transfer happens in 2 steps
  a. The owner calls approve and gives it the approved address of the new owner, and the tokenID we want to send
  b. The new owner can then call transferFrom with the tokenId. The contract checks to make sure the new owner has been approved before the transfer occurs (hence the require statement in the last part)
- We can add to the zombieApprovals map that we created earlier
  - We must also add the onlyOwnerOf modifier to approve, so that only the current owner of the token can call approve



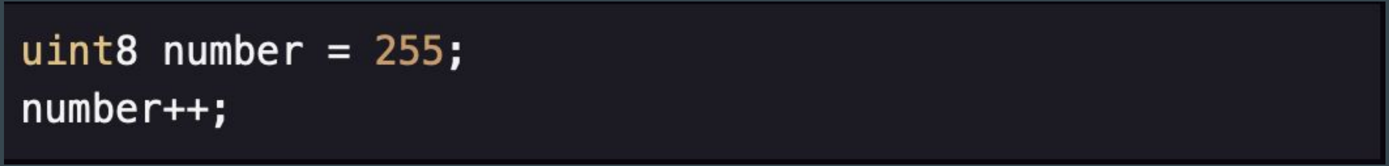
```
function approve(address _approved, uint256 _tokenId) external payable onlyOwnerOf(_tokenId) {
  zombieApprovals[_tokenId] = _approved;
  emit Approval(msg.sender, _approved, _tokenId);
}
```

Withdraw

```
contract GetPaid is Ownable {
  function withdraw() external onlyOwner {
    address payable _owner = address(uint160(owner()));
    _owner.transfer(address(this).balance);
  }
}
```

Preventing Overflows

- Security flaws?
  - Overflow and Underflow



```
uint8 number = 255;
number++;
```

- How can we prevent this?
  - SafeMath library from OpenZeppelin

Importing SafeMath

```
using SafeMath for uint256;

uint256 a = 5;
uint256 b = a.add(3); // 5 + 3 = 8
uint256 c = a.mul(2); // 5 * 2 = 10
```

dApps

- A decentralized application (dApp) is a type of distributed open source software application that runs on a peer-to-peer (P2P) blockchain network rather than on a single computer or central server.
- Decentralized apps run on multiple platforms at once. Therefore, when you make a dApp, you rarely end up with a single application.
- What are dApps made of?
- Popular dApps: OpenSea, Brave Browser, Uniswap

Tech Stack for dApps

- A smart contract or combination of smart contracts is where you'd put the decentralized logic of your dApp. What actions do you want the dApp to perform automatically, relying on the chain consensus?
- Create front-end for the application.
- Create a centralized backend: If you're building Ethereum dApps that require massive data storage or plan on running reports or envision any other functionality that's out of the scope of on-chain transactions, you will need a web solution on a private server.
- Push it to your favorite App store!

Summary

- Solidity concepts
- dApps development