Lesson 5: ERC721 & Crypto-Collectibles

Whew! Things are starting to heat up in here...

In this lesson, we're going to get a bit more advanced.

We're going to talk about **tokens**, the **ERC721** standard, and **crypto-collectible assets**.

In other words, we're going to **make it so you can trade your zombies with your friends.**

Are you ready to get started?

Zombieownership.sol

```
// Start here
```

Chapter 2: ERC721 Standard, Multiple Inheritance

Let's take a look at the ERC721 standard:

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

  function balanceOf(address _owner) public view returns (uint256 _balance);
  function ownerOf(uint256 _tokenId) public view returns (address _owner);
  function transfer(address _to, uint256 _tokenId) public;
  function approve(address _to, uint256 _tokenId) public;
  function takeOwnership(uint256 _tokenId) public;
}
```

This is the list of methods we'll need to implement, which we'll be doing over the coming chapters in pieces.

It looks like a lot, but don't get overwhelmed! We're here to walk you through it.

Note: The ERC721 standard is currently a draft, and there is no officially agreed-upon implementation yet. For this tutorial we're using the current version from OpenZeppelin's

*library, but it is possible it will change in the future before its official release. So consider this **one** possible implementation, but don't take it as the official standard for ERC721 tokens.*

Implementing a token contract

When implementing a token contract, the first thing we do is copy the interface to its own Solidity file and import it, `import ./erc721.sol`. Then we have our contract inherit from it, and we override each method with a function definition.

But wait — `ZombieOwnership` is already inheriting from `ZombieAttack` — how can it also inherit from `ERC721`?

Luckily in Solidity, your contract can inherit from multiple contracts as follows:

```
contract SatoshiNakamoto is NickSzabo, HalFinney {
  // Omg, the secrets of the universe revealed!
}
```

As you can see, when using multiple inheritance, you just separate the multiple contracts you're inheriting from with a comma, `,`. In this case, our contract is inheriting from `NickSzabo` and `HalFinney`.

Let's give it a try.

Putting it to the Test

We've already created `erc721.sol` with the interface above for you.

1. Import `erc721.sol` into `zombieownership.sol`
2. Declare that `ZombieOwnership` inherits from `ZombieAttack` AND `ERC721`

Zombieownership.sol

```
pragma solidity ^0.4.19;

import "./zombieattack.sol";
// Import file here

// Declare ERC721 inheritance here
contract ZombieOwnership is ZombieAttack {

}
```

Chapter 3: balanceOf & ownerOf

Great, let's dive into the ERC721 implementation!

We've gone ahead and copied the empty shell of all the functions you'll be implementing in this lesson.

In this chapter, we're going to implement the first two methods: `balanceOf` and `ownerOf`.

balanceOf

```
function balanceOf(address _owner) public view returns (uint256 _balance);
```

This function simply takes an `address`, and returns how many tokens that `address` owns.

In our case, our "tokens" are Zombies. Do you remember where in our DApp we stored how many zombies an owner has?

ownerOf

```
function ownerOf(uint256 _tokenId) public view returns (address _owner);
```

This function takes a token ID (in our case, a Zombie ID), and returns the `address` of the person who owns it.

Again, this is very straightforward for us to implement, since we already have a `mapping` in our DApp that stores this information. We can implement this function in one line, just a `return` statement.

Note: Remember, `uint256` is equivalent to `uint`. We've been using `uint` in our code up until now, but we're using `uint256` here because we copy/pasted from the spec.

Putting it to the Test

I'll leave it to you to figure out how to implement these 2 functions.

Each function should simply be 1 line of code, a `return` statement. Take a look at our code from previous lessons to see where we're storing this data. If you can't figure it out, you can hit the "show me the answer" button for some help.

1. Implement `balanceOf` to return the number of zombies `_owner` has.
2. Implement `ownerOf` to return the address of whoever owns the zombie with ID `_tokenId`.

Erc721.sol

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

  function balanceOf(address _owner) public view returns (uint256 _balance);
  function ownerOf(uint256 _tokenId) public view returns (address _owner);
  function transfer(address _to, uint256 _tokenId) public;
  function approve(address _to, uint256 _tokenId) public;
  function takeOwnership(uint256 _tokenId) public;
}
```

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

  function balanceOf(address _owner) public view returns (uint256 _balance) {
    // 1. Return the number of zombies `_owner` has here
  }

  function ownerOf(uint256 _tokenId) public view returns (address _owner) {
    // 2. Return the owner of `_tokenId` here
  }

  function transfer(address _to, uint256 _tokenId) public {

  }

  function approve(address _to, uint256 _tokenId) public {

  }

  function takeOwnership(uint256 _tokenId) public {

  }
}
```

Chapter 4: Refactoring

Uh oh! We've just introduced an error in our code that will make it not compile. Did you notice it?

In the previous chapter we defined a function called `ownerOf`. But if you recall from Lesson 4, we also created a `modifier` with the same name, `ownerOf`, in `zombiefeeding.sol`.

If you tried compiling this code, the compiler would give you an error saying you can't have a modifier and a function with the same name.

So should we just change the function name in `ZombieOwnership` to something else?

No, we can't do that!!! Remember, we're using the ERC721 token standard, which means other contracts will expect our contract to have functions with these exact names defined. That's what makes these standards useful — if another contract knows our contract is ERC721-compliant, it can simply talk to us without needing to know anything about our internal implementation decisions.

So that means we'll have to refactor our code from Lesson 4 to change the name of the `modifier` to something else.

Putting it to the Test

We're back in `zombiefeeding.sol`. We're going to change the name of our `modifier` from `ownerOf` to `onlyOwnerOf`.

1. Change the name of the modifier definition to `onlyOwnerOf`
2. Scroll down to the function `feedAndMultiply`, which uses this modifier. We'll need to change the name here as well.

Note: We also use this modifier in `zombiehelper.sol` and `zombieattack.sol`, but so we don't spend too much of this lesson refactoring, we've gone ahead and changed the modifier names in those files for you.

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

```
contract ZombieOwnership is ZombieAttack, ERC721 {

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

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

  function transfer(address _to, uint256 _tokenId) public {

  }

  function approve(address _to, uint256 _tokenId) public {

  }

  function takeOwnership(uint256 _tokenId) public {

  }
}
```

Zombiefeeding.sol

```
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 {

  KittyInterface kittyContract;

  // 1. Change modifier name to `onlyOwnerOf`
  modifier ownerOf(uint _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    _;
  }

  function setKittyContractAddress(address _address) external onlyOwner {
    kittyContract = KittyInterface(_address);
  }

  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  function _isReady(Zombie storage _zombie) internal view returns (bool) {
      return (_zombie.readyTime <= now);
  }

  // 2. Change modifier name here as well
  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species)
internal ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    require(_isReady(myZombie));
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }
}
```

Chapter 5: ERC721: Transfer Logic

Great, we've fixed the conflict!

Now we're going to continue our ERC721 implementation by looking at transfering ownership from one person to another.

Note that the ERC721 spec has 2 different ways to transfer tokens:

```
function transfer(address _to, uint256 _tokenId) public;
function approve(address _to, uint256 _tokenId) public;
function takeOwnership(uint256 _tokenId) public;
```

1. The first way is the token's owner calls `transfer` with the `address` he wants to transfer to, and the `_tokenId` of the token he wants to transfer.
2. The second way is the token's owner first calls `approve`, and sends it the same info as above. The contract then stores who is approved to take a token, usually in a `mapping (uint256 => address)`. Then when someone calls `takeOwnership`, the contract checks if that `msg.sender`is approved by the owner to take the token, and if so it transfers the token to him.

If you'll notice, both `transfer` and `takeOwnership` will contain the same transfer logic, just in reverse order. (In one case the sender of the token calls the function; in the other the receiver of the token calls it).

So it makes sense for us to abstract this logic into its own private function, `_transfer`, which is then called by both functions. That way we don't repeat the same code twice.

Putting it to the Test

Let's define the logic for `_transfer`.

1. Define a function named `_transfer`. It will take 3 arguments, `address _from`, `address _to`, and `uint256 _tokenId`. It should be a `private`function.
2. We have 2 mappings that will change when ownership changes: `ownerZombieCount` (which keeps track of how many zombies an owner has) and `zombieToOwner` (which keeps track of who owns what).

    The first thing our function should do is increment `ownerZombieCount` for the person **receiving** the zombie (`address _to`). Use `++` to increment.

3. Next, we'll need to **decrease** the `ownerZombieCount` for the person **sending** the zombie (`address _from`). Use `--` to decrement.
4. Lastly, we'll want to change `zombieToOwner` mapping for this `_tokenId`so it now points to `_to`.

5. I lied, that wasn't the last step. There's one more thing we should do.

   The ERC721 spec includes a `Transfer` event. The last line of this function should fire `Transfer` with the correct information — check `erc721.sol` to see what arguments it's expecting to be called with and implement it here.

Zombiefeeding.sol

```
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 {

  KittyInterface kittyContract;

  modifier onlyOwnerOf(uint _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    _;
  }

  function setKittyContractAddress(address _address) external onlyOwner {
    kittyContract = KittyInterface(_address);
  }

  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  function _isReady(Zombie storage _zombie) internal view returns (bool) {
      return (_zombie.readyTime <= now);
```

```
  }

  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species)
internal onlyOwnerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    require(_isReady(myZombie));
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }
}
```

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

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

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

  // Define _transfer() here

  function transfer(address _to, uint256 _tokenId) public {

  }
```

```
  function approve(address _to, uint256 _tokenId) public {

  }

  function takeOwnership(uint256 _tokenId) public {

  }
}
```

Chapter 6: ERC721: Transfer Cont'd

Great! That was the difficult part — now implementing the public `transfer`function will be easy, since our `_transfer` function already does almost all the heavy lifting.

Putting it to the Test

1. We want to make sure only the owner of a token/zombie can transfer it. Remember how we can limit access to a function to only its owner?

    Yep, that's right, we already have a modifier that does this. So add the `onlyOwnerOf` modifier to this function.

2. Now the body of the function really only needs to be one line... It just needs to call `_transfer`.

    Make sure to pass in `msg.sender` for the `address _from` argument.

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

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

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

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

  // 1. Add modifier here
  function transfer(address _to, uint256 _tokenId) public {
    // 2. Define function here
  }

  function approve(address _to, uint256 _tokenId) public {

  }

  function takeOwnership(uint256 _tokenId) public {

  }
}
```

Chapter 7: ERC721: Approve

Now, let's implement `approve`.

Remember, with `approve` / `takeOwnership`, the transfer happens in 2 steps:

1. You, the owner, call `approve` and give it the `address` of the new owner, and the `_tokenId` you want him to take
2. The new owner calls `takeOwnership` with the `_tokenId`, the contract checks to make sure he's already been approved, and then transfers him the token.

Because this happens in 2 function calls, we need a data structure to store who's been approved for what in between function calls.

Putting it to the Test

1. First, let's define a mapping `zombieApprovals`. It should map a `uint` to an `address`.

   This way, when someone calls `takeOwnership` with a `_tokenId`, we can use this mapping to quickly look up who is approved to take that token.

2. On the `approve` function, we want to make sure only the owner of the token can give someone approval to take it. So we need to add the `onlyOwnerOf` modifier to `approve`
3. For the body of the function, set `zombieApprovals` for `_tokenId` equal to the `_to` address.
4. Finally, there's an `Approval` event in the ERC721 spec. So we should fire this event at the end of this function. Check `erc721.sol` for the arguments, and be sure to use `msg.sender` as `_owner`.

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

  // 1. Define mapping here

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

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

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

  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }
```

```
  // 2. Add function modifier here
  function approve(address _to, uint256 _tokenId) public {
    // 3. Define function here
  }

  function takeOwnership(uint256 _tokenId) public {

  }
}
```

Chapter 8: ERC721: takeOwnership

Great, now let's finish up our ERC721 implementation with the last function! (Don't worry, there's still more to cover in Lesson 5 after this 😉)

The final function, `takeOwnership`, should simply check to make sure the `msg.sender` has been approved to take this token / zombie, and call `_transfer` if so.

Putting it to the Test

1. First, we want to use a `require` statement to check that `zombieApprovals` for `_tokenId` is equal to `msg.sender`.

   That way if `msg.sender` hasn't been approved to take this token, it will throw an error.

2. In order to call `_transfer`, we need to know the address of the person who owns the token (it requires `_from` as an argument). Luckily we can look this up with our function `ownerOf`.

   So declare an `address` variable named `owner`, and set it equal to `ownerOf(_tokenId)`.

3. Finally, call `_transfer`, and pass it all the required information. (Here you can use `msg.sender` for `_to`, since the person calling this function is the one the token should be sent to).

Note: We could have done steps 2 and 3 in one line of code, but splitting it up makes things a bit more readable. Personal preference.

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

  mapping (uint => address) zombieApprovals;

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

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

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

  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }

  function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    zombieApprovals[_tokenId] = _to;
    Approval(msg.sender, _to, _tokenId);
  }

  function takeOwnership(uint256 _tokenId) public {
    // Start here
  }
}
```

Chapter 9: Preventing Overflows

Congratulations, that completes our ERC721 implementation!

That wasn't so tough, was it? A lot of this Ethereum stuff sounds really complicated when you hear people talking about it, so the best way to understand it is to actually go through an implementation of it yourself.

Keep in mind that this is only a minimal implementation. There are extra features we may want to add to our implementation, such as some extra checks to make sure users don't accidentally transfer their zombies to address `0` (which is called "burning" a token — basically it's sent to an address that no one has the private key of, essentially making it unrecoverable). Or to put some basic auction logic in the DApp itself. (Can you think of some ways we could implement that?)

But we wanted to keep this lesson manageable, so we went with the most basic implementation. If you want to see an example of a more in-depth implementation, you can take a look at the OpenZeppelin ERC721 contract after this tutorial.

Contract security enhancements: Overflows and Underflows

We're going to look at one major security feature you should be aware of when writing smart contracts: Preventing overflows and underflows.

What's an ***overflow***?

Let's say we have a `uint8`, which can only have 8 bits. That means the largest number we can store is binary `11111111` (or in decimal, 2^8 - 1 = 255).

Take a look at the following code. What is `number` equal to at the end?

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

In this case, we've caused it to overflow — so `number` is counterintuitively now equal to `0` even though we increased it. (If you add 1 to binary `11111111`, it resets back to `00000000`, like a clock going from `23:59` to `00:00`).

An underflow is similar, where if you subtract `1` from a `uint8` that equals `0`, it will now equal `255` (because `uint`s are unsigned, and cannot be negative).

While we're not using `uint8` here, and it seems unlikely that a `uint256` will overflow when incrementing by `1` each time (2^256 is a really big number), it's still good to put protections in our contract so that our DApp never has unexpected behavior in the future.

Using SafeMath

To prevent this, OpenZeppelin has created a ***library*** called SafeMath that prevents these issues by default.

But before we get into that... What's a library?

A ***library*** is a special type of contract in Solidity. One of the things it is useful for is to attach functions to native data types.

For example, with the SafeMath library, we'll use the syntax `using SafeMath for uint256`. The SafeMath library has 4 functions — `add`, `sub`, `mul`, and `div`. And now we can access these functions from `uint256` as follows:

```
using SafeMath for uint256;

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

We'll look at what these functions do in the next chapter, but for now let's add the SafeMath library to our contract.

Putting it to the Test

We've already included OpenZeppelin's `SafeMath` library for you in `safemath.sol`. You can take a quick peek at the code now if you want to, but we'll be looking at it in depth in the next chapter.

First let's tell our contract to use SafeMath. We'll do this in ZombieFactory, our very base contract — that way we can use it in any of the sub-contracts that inherit from this one.

1. Import `safemath.sol` into `zombiefactory.sol`.
2. Add the declaration `using SafeMath for uint256;`.

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {
```

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

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

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

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

  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }

  function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    zombieApprovals[_tokenId] = _to;
    Approval(msg.sender, _to, _tokenId);
  }

  function takeOwnership(uint256 _tokenId) public {
    require(zombieApprovals[_tokenId] == msg.sender);
    address owner = ownerOf(_tokenId);
    _transfer(owner, msg.sender, _tokenId);
  }
}
```

Zombiefactory.sol

```
pragma solidity ^0.4.19;

import "./ownable.sol";
// 1. Import here

contract ZombieFactory is Ownable {

  // 2. Declare using safemath here
```

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

  uint dnaDigits = 16;
  uint dnaModulus = 10 ** dnaDigits;
  uint cooldownTime = 1 days;

  struct Zombie {
    string name;
    uint dna;
    uint32 level;
    uint32 readyTime;
    uint16 winCount;
    uint16 lossCount;
  }

  Zombie[] public zombies;

  mapping (uint => address) public zombieToOwner;
  mapping (address => uint) ownerZombieCount;

  function _createZombie(string _name, uint _dna) internal {
    uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime), 0,
0)) - 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);
    randDna = randDna - randDna % 100;
    _createZombie(_name, randDna);
  }

}
```

Openzeeplin library: safeMath to prevent overflow and underflow.

```
pragma solidity ^0.4.18;

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the quotient.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  /**
  * @dev Substracts two numbers, throws on overflow (i.e. if subtrahend is
greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

Safemath.sol

```
pragma solidity ^0.4.18;

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the quotient.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  /**
  * @dev Substracts two numbers, throws on overflow (i.e. if subtrahend is
greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
```

```
    return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

Chapter 10: SafeMath Part 2

Let's take a look at the code behind SafeMath:

```
library SafeMath {

  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

First we have the `library` keyword — libraries are similar to `contract`s but with a few differences. For our purposes, libraries allow us to use the `using`keyword, which automatically tacks on all of the library's methods to another data type:

```
using SafeMath for uint;
// now we can use these methods on any uint
uint test = 2;
test = test.mul(3); // test now equals 6
test = test.add(5); // test now equals 11
```

Note that the `mul` and `add` functions each require 2 arguments, but when we declare `using SafeMath for uint`, the `uint` we call the function on (`test`) is automatically passed in as the first argument.

Let's look at the code behind `add` to see what SafeMath does:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  assert(c >= a);
  return c;
}
```

Basically `add` just adds 2 `uint`s like `+`, but it also contains an `assert`statement to make sure the sum is greater than `a`. This protects us from overflows.

`assert` is similar to `require`, where it will throw an error if false. The difference between `assert` and `require` is that `require` will refund the user the rest of their gas when a function fails, whereas `assert` will not. So most of the time you want to use `require` in your code; `assert` is typically used when something has gone horribly wrong with the code (like a `uint` overflow).

So, simply put, SafeMath's `add`, `sub`, `mul`, and `div` are functions that do the basic 4 math operations, but throw an error if an overflow or underflow occurs.

Using SafeMath in our code.

To prevent overflows and underflows, we can look for places in our code where we use `+`, `-`, `*`, or `/`, and replace them with `add`, `sub`, `mul`, `div`.

Ex. Instead of doing:

```
myUint++;
```

We would do:

```
myUint = myUint.add(1);
```

Putting it to the Test

We have 2 places in `ZombieOwnership` where we used math operations. Let's swap them out with SafeMath methods.

1. Replace `++` with a SafeMath method.
2. Replace `--` with a SafeMath method.

Zombiefactory.sol

```
pragma solidity ^0.4.19;

import "./ownable.sol";
import "./safemath.sol";

contract ZombieFactory is Ownable {

  using SafeMath for uint256;

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

  uint dnaDigits = 16;
  uint dnaModulus = 10 ** dnaDigits;
  uint cooldownTime = 1 days;

  struct Zombie {
    string name;
    uint dna;
    uint32 level;
    uint32 readyTime;
    uint16 winCount;
    uint16 lossCount;
  }

  Zombie[] public zombies;

  mapping (uint => address) public zombieToOwner;
  mapping (address => uint) ownerZombieCount;

  function _createZombie(string _name, uint _dna) internal {
    uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime), 0,
0)) - 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);
    randDna = randDna - randDna % 100;
    _createZombie(_name, randDna);
  }

}
```

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

  using SafeMath for uint256;

  mapping (uint => address) zombieApprovals;

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

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

  function _transfer(address _from, address _to, uint256 _tokenId) private {
    // 1. Replace with SafeMath's `add`
    ownerZombieCount[_to]++;
    // 2. Replace with SafeMath's `sub`
    ownerZombieCount[_from]--;
    zombieToOwner[_tokenId] = _to;
```

```
    Transfer(_from, _to, _tokenId);
  }

  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }

  function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    zombieApprovals[_tokenId] = _to;
    Approval(msg.sender, _to, _tokenId);
  }

  function takeOwnership(uint256 _tokenId) public {
    require(zombieApprovals[_tokenId] == msg.sender);
    address owner = ownerOf(_tokenId);
    _transfer(owner, msg.sender, _tokenId);
  }
}
```

Chapter 11: SafeMath Part 3

Great, now our ERC721 implementation is safe from overflows & underflows!

Going back through the code we wrote in previous lessons, there's a few other places in our code that could be vulnerable to overflows or underflows.

For example, in ZombieAttack we have:

```
myZombie.winCount++;
myZombie.level++;
enemyZombie.lossCount++;
```

We should prevent overflows here as well just to be safe. (It's a good idea in general to just use SafeMath instead of the basic math operations. Maybe in a future version of Solidity these will be implemented by default, but for now we have to take extra security precautions in our code).

However we have a slight problem — `winCount` and `lossCount` are `uint16`s, and `level` is a `uint32`. So if we use SafeMath's `add` method with these as arguments, it won't actually protect us from overflow since it will convert these types to `uint256`:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
  uint256 c = a + b;
  assert(c >= a);
  return c;
}

// If we call `.add` on a `uint8`, it gets converted to a `uint256`.
// So then it won't overflow at 2^8, since 256 is a valid `uint256`.
```

This means we're going to need to implement 2 more libraries to prevent overflow/underflows with our `uint16`s and `uint32`s. We can call them `SafeMath16` and `SafeMath32`.

The code will be exactly the same as SafeMath, except all instances of `uint256` will be replaced with `uint32` or `uint16`.

We've gone ahead and implemented that code for you — go ahead and look at `safemath.sol` to see the code.

Now we need to implement it in ZombieFactory.

Putting it to the Test

Assignment:

1. Declare that we're using `SafeMath32` for `uint32`.
2. Declare that we're using `SafeMath16` for `uint16`.
3. There's one more line of code in ZombieFactory where we should use a SafeMath method. We've left a comment to indicate where.

Zombieownership.sol

```
pragma solidity ^0.4.19;

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

contract ZombieOwnership is ZombieAttack, ERC721 {

  using SafeMath for uint256;

  mapping (uint => address) zombieApprovals;

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

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

  function _transfer(address _from, address _to, uint256 _tokenId) private {
    ownerZombieCount[_to] = ownerZombieCount[_to].add(1);
    ownerZombieCount[msg.sender] = ownerZombieCount[msg.sender].sub(1);
    zombieToOwner[_tokenId] = _to;
    Transfer(_from, _to, _tokenId);
  }

  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }

  function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    zombieApprovals[_tokenId] = _to;
    Approval(msg.sender, _to, _tokenId);
  }

  function takeOwnership(uint256 _tokenId) public {
    require(zombieApprovals[_tokenId] == msg.sender);
    address owner = ownerOf(_tokenId);
    _transfer(owner, msg.sender, _tokenId);
  }
}
```

Zombiefactory.sol

```
pragma solidity ^0.4.19;

import "./ownable.sol";
import "./safemath.sol";

contract ZombieFactory is Ownable {

  using SafeMath for uint256;
  // 1. Declare using SafeMath32 for uint32
  // 2. Declare using SafeMath16 for uint16

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

  uint dnaDigits = 16;
```

```
  uint dnaModulus = 10 ** dnaDigits;
  uint cooldownTime = 1 days;

  struct Zombie {
    string name;
    uint dna;
    uint32 level;
    uint32 readyTime;
    uint16 winCount;
    uint16 lossCount;
  }

  Zombie[] public zombies;

  mapping (uint => address) public zombieToOwner;
  mapping (address => uint) ownerZombieCount;

  function _createZombie(string _name, uint _dna) internal {
    // Note: We chose not to prevent the year 2038 problem... So don't need
    // worry about overflows on readyTime. Our app is screwed in 2038 anyway ;)
    uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime), 0,
0)) - 1;
    zombieToOwner[id] = msg.sender;
    // 3. Let's use SafeMath's `add` here:
    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);
    randDna = randDna - randDna % 100;
    _createZombie(_name, randDna);
  }

}
```

Chapter 12: SafeMath Part 4

Great, now we can implement SafeMath on all the types of `uint`s we used in our DApp!

Let's fix all those potential issues in `ZombieAttack`. (There was also one `zombies[_zombieId].level++;` that needed to be fixed in `ZombieHelper`, but we've taken care of that one for you so we don't take an extra chapter to do so 😉).

Putting it to the Test

Go ahead and implement SafeMath methods on all the `++` increments in `ZombieAttack`. We've left comments in the code to make them easy to find.

Zombiefactory.sol

```
pragma solidity ^0.4.19;

import "./ownable.sol";
import "./safemath.sol";

contract ZombieFactory is Ownable {

  using SafeMath for uint256;
  using SafeMath32 for uint32;
  using SafeMath16 for uint16;

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

  uint dnaDigits = 16;
  uint dnaModulus = 10 ** dnaDigits;
  uint cooldownTime = 1 days;

  struct Zombie {
    string name;
    uint dna;
    uint32 level;
    uint32 readyTime;
    uint16 winCount;
    uint16 lossCount;
  }

  Zombie[] public zombies;

  mapping (uint => address) public zombieToOwner;
```

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

  function _createZombie(string _name, uint _dna) internal {
    uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime), 0,
0)) - 1;
    zombieToOwner[id] = msg.sender;
    ownerZombieCount[msg.sender] = ownerZombieCount[msg.sender].add(1);
    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);
    randDna = randDna - randDna % 100;
    _createZombie(_name, randDna);
  }

}
```

Zombieattack.sol

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    // Here's one!
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  function attack(uint _zombieId, uint _targetId) external onlyOwnerOf(_zombieId)
{
    Zombie storage myZombie = zombies[_zombieId];
    Zombie storage enemyZombie = zombies[_targetId];
```

```
    uint rand = randMod(100);
    if (rand <= attackVictoryProbability) {
      // Here's 3 more!
      myZombie.winCount++;
      myZombie.level++;
      enemyZombie.lossCount++;
      feedAndMultiply(_zombieId, enemyZombie.dna, "zombie");
    } else {
      // ...annnnd another 2!
      myZombie.lossCount++;
      enemyZombie.winCount++;
      _triggerCooldown(myZombie);
    }
  }
}
```

Chapter 13: Comments

The Solidity code for our zombie game is finally finished!

In the next lessons, we'll look at how to deploy the code to Ethereum, and how to interact with it with Web3.js.

But one final thing before we let you go in Lesson 5: Let's talk about **commenting your code**.

Syntax for comments

Commenting in Solidity is just like JavaScript. You've already seen some examples of single line comments throughout the CryptoZombies lessons:

```
// This is a single-line comment. It's kind of like a note to self (or to others)
```

Just add double `//` anywhere and you're commenting. It's so easy that you should do it all the time.

But I hear you — sometimes a single line is not enough. You are born a writer, after all!

Thus we also have multi-line comments:

```
contract CryptoZombies {
  /* This is a multi-lined comment. I'd like to thank all of you
    who have taken your time to try this programming course.
```

```
    I know it's free to all of you, and it will stay free
    forever, but we still put our heart and soul into making
    this as good as it can be.

    Know that this is still the beginning of Blockchain development.
    We've come very far but there are so many ways to make this
    community better. If we made a mistake somewhere, you can
    help us out and open a pull request here:
    https://github.com/loomnetwork/cryptozombie-lessons

    Or if you have some ideas, comments, or just want to say
    hi - drop by our Telegram community at https://t.me/loomnetwork
  */
}
```

In particular, it's good practice to comment your code to explain the expected behavior of every function in your contract. This way another developer (or you, after a 6 month hiatus from a project!) can quickly skim and understand at a high level what your code does without having to read the code itself.

The standard in the Solidity community is to use a format called ***natspec***, which looks like this:

```
/// @title A contract for basic math operations
/// @author H4XF13LD MORRIS 💯💯😎💯💯
/// @notice For now, this contract just adds a multiply function
contract Math {
  /// @notice Multiplies 2 numbers together
  /// @param x the first uint.
  /// @param y the second uint.
  /// @return z the product of (x * y)
  /// @dev This function does not currently check for overflows
  function multiply(uint x, uint y) returns (uint z) {
    // This is just a normal comment, and won't get picked up by natspec
    z = x * y;
  }
}
```

`@title` and `@author` are straightforward.

`@notice` explains to a **user** what the contract / function does. `@dev` is for explaining extra details to developers.

`@param` and `@return` are for describing what each parameter and return value of a function are for.

Note that you don't always have to use all of these tags for every function — all tags are optional. But at the very least, leave a `@dev` note explaining what each function does.

Put it to the test

If you haven't noticed by now, the CryptoZombies answer-checker ignores comments when it checks your answers. So we can't actually check your natspec code for this chapter ;)

However, by now you're a Solidity whiz — we're just going to assume you've got this!

Give it a try anyway, and try adding some natspec tags to `ZombieOwnership`:

1. `@title` — E.g. A contract that manages transfering zombie ownership
2. `@author` — Your name!
3. `@dev` — E.g. Compliant with OpenZeppelin's implementation of the ERC721 spec draft

Chapter 14: Wrapping It Up

Congratulations! That concludes Lesson 5.

As a reward, we've transferred you your very own Level 10 **H4XF13LD MORRIS** 💯💯😎💯💯 zombie!

(Omg, the legendary **H4XF13LD MORRIS** 💯💯😎💯💯 zombie!!!!111)

Now you have 4 zombies in your army.

Before you move on, you have the option to rename any of them if you'd like by clicking on them to the right and entering a new name. (Though I don't know why you would ever want to rename **H4XF13LD MORRIS** 💯💯😎💯💯, clearly the best name ever).

Let's recap:

In this lesson we learned about:

- Tokens, the ERC721 standard, and tradable assets/zombies
- Libraries and how to use them
- How to prevent overflows and underflows using the SafeMath library
- Commenting your code and the natspec standard

This lesson concludes our game's Solidity code! (For now — we may add even more lessons in the future).

In the next 2 lessons, we're going to look at how to deploy your contracts and interact with them using ***web3.js*** (so you can build a front-end for your DApp).

Go ahead and rename any of your zombies if you like, then proceed to the next chapter to complete the lesson.