Lesson 3: Advanced Solidity Concepts

Grr... I just can't slow you down, can I? Your Solidity skills are formidable, human...

Now that you've got some experience coding Solidity under your belt, we're going to dive into some of the more technical aspects of Ethereum development.

This lesson will be a bit less flashy (sorry, no plot twists!). But you’ll learn some really important concepts that will take you closer to building real DApps — things like **contract ownership, gas costs, code optimization, and security**.

You've been warned — no kitties and rainbows in Lesson 3!

But a lot of densely-packed Solidity knowledge. We strongly recommended that you complete Lesson 2 before starting this one.

Are you ready to get started?

Zombiefactory.sol

```
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) internal {
        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);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }

}
```

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 {

  // 1. Remove this:
  address ckAddress = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;
  // 2. Change this to just a declaration:
  KittyInterface kittyContract = KittyInterface(ckAddress);

  // 3. Add setKittyContractAddress method here

  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");
  }

}
```

Chapter 1: Immutability of Contracts

Up until now, Solidity has looked quite similar to other languages like JavaScript. But there are a number of ways that Ethereum DApps are actually quite different from normal applications.

To start with, after you deploy a contract to Ethereum, it’s **immutable**, which means that it can never be modified or updated again.

The initial code you deploy to a contract is there to stay, permanently, on the blockchain. This is one reason security is such a huge concern in Solidity. If there's a flaw in your contract code, there's no way for you to patch it later. You would have to tell your users to start using a different smart contract address that has the fix.

But this is also a feature of smart contracts. The code is law. If you read the code of a smart contract and verify it, you can be sure that every time you call a function it's going to do exactly what the code says it will do. No one can later change that function and give you unexpected results.

External dependencies

In Lesson 2, we hard-coded the CryptoKitties contract address into our DApp. But what would happen if the CryptoKitties contract had a bug and someone destroyed all the kitties?

It's unlikely, but if this did happen it would render our DApp completely useless — our DApp would point to a hardcoded address that no longer returned any kitties. Our zombies would be unable to feed on kitties, and we'd be unable to modify our contract to fix it.

For this reason, it often makes sense to have functions that will allow you to update key portions of the DApp.

For example, instead of hard coding the CryptoKitties contract address into our DApp, we should probably have a `setKittyContractAddress` function that lets us change this address in the future in case something happens to the CryptoKitties contract.

Put it to the test

Let's update our code from Lesson 2 to be able to change the CryptoKitties contract address.

1. Delete the line of code where we hard-coded `ckAddress`.
2. Change the line where we created `kittyContract` to just declare the variable — i.e. don't set it equal to anything.
3. Create a function called `setKittyContractAddress`. It will take one argument, `_address` (an `address`), and it should be an `external`function.
4. Inside the function, add one line of code that sets `kittyContract` equal to `KittyInterface(_address)`.

Note: If you notice a security hole with this function, don't worry — we'll fix it in the next chapter ;)

```
pragma solidity ^0.4.19;

// 1. Import here

// 2. Inherit here:
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) internal {
        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);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }

}
```

Chapter 2: Ownable Contracts

Did you spot the security hole in the previous chapter?

`setKittyContractAddress` is `external`, so anyone can call it! That means anyone who called the function could change the address of the CryptoKitties contract, and break our app for all its users.

We do want the ability to update this address in our contract, but we don't want everyone to be able to update it.

To handle cases like this, one common practice that has emerged is to make contracts `Ownable` — meaning they have an owner (you) who has special privileges.

OpenZeppelin's `Ownable` contract

Below is the `Ownable` contract taken from the **OpenZeppelin** Solidity library. OpenZeppelin is a library of secure and community-vetted smart contracts that you can use in your own DApps. After this lesson, we highly recommend you check out their site to further your learning!

Give the contract below a read-through. You're going to see a few things we haven't learned yet, but don't worry, we'll talk about them afterward.

```
/**
 * @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;
  }
}
```

A few new things here we haven't seen before:

- Constructors: `function Ownable()` is a **constructor**, which is an optional special function that has the same name as the contract. It will get executed only one time, when the contract is first created.
- Function Modifiers: `modifier onlyOwner()`. Modifiers are kind of half-functions that are used to modify other functions, usually to check some requirements prior to execution. In this case, `onlyOwner` can be used to limit access so **only** the **owner** of the contract can run this function. We'll talk more about function modifiers in the next chapter, and what that weird `_;` does.
- `indexed` keyword: don't worry about this one, we don't need it yet.

So the `Ownable` contract basically does the following:

1. When a contract is created, its constructor sets the `owner` to `msg.sender` (the person who deployed it)
2. It adds an `onlyOwner` modifier, which can restrict access to certain functions to only the `owner`
3. It allows you to transfer the contract to a new `owner`

`onlyOwner` is such a common requirement for contracts that most Solidity DApps start with a copy/paste of this `Ownable` contract, and then their first contract inherits from it.

Since we want to limit `setKittyContractAddress` to `onlyOwner`, we're going to do the same for our contract.

Put it to the test

We've gone ahead and copied the code of the `Ownable` contract into a new file, `ownable.sol`. Let's go ahead and make `ZombieFactory` inherit from it.

1. Modify our code to `import` the contents of `ownable.sol`. If you don't remember how to do this take a look at `zombiefeeding.sol`.
2. Modify the `ZombieFactory` contract to inherit from `Ownable`. Again, you can take a look at `zombiefeeding.sol` if you don't remember how this is done.

```
/**
 * @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;
  }

}

```

Chapter 3: onlyOwner Function Modifier

Now that our base contract `ZombieFactory` inherits from `Ownable`, we can use the `onlyOwner` function modifier in `ZombieFeeding` as well.

This is because of how contract inheritance works. Remember:

```
ZombieFeeding is ZombieFactory
ZombieFactory is Ownable
```

Thus `ZombieFeeding` is also `Ownable`, and can access the functions / events / modifiers from the `Ownable` contract. This applies to any contracts that inherit from `ZombieFeeding` in the future as well.

Function Modifiers

A function modifier looks just like a function, but uses the keyword `modifier`instead of the keyword `function`. And it can't be called directly like a function can — instead we can attach the modifier's name at the end of a function definition to change that function's behavior.

Let's take a closer look by examining `onlyOwner`:

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

We would use this modifier as follows:

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

  // Note the usage of `onlyOwner` below:
  function likeABoss() external onlyOwner {
    LaughManiacally("Muahahahaha");
  }
}
```

Notice the `onlyOwner` modifier on the `likeABoss` function. When you call `likeABoss`, the code inside `onlyOwner` executes **first**. Then when it hits the `_;` statement in `onlyOwner`, it goes back and executes the code inside `likeABoss`.

So while there are other ways you can use modifiers, one of the most common use-cases is to add quick `require` check before a function executes.

In the case of `onlyOwner`, adding this modifier to a function makes it so **only**the **owner** of the contract (you, if you deployed it) can call that function.

Note: Giving the owner special powers over the contract like this is often necessary, but it could also be used maliciously. For example, the owner could add a backdoor function that would allow him to transfer anyone's zombies to himself!

So it's important to remember that just because a DApp is on Ethereum does not automatically mean it's decentralized — you have to actually read the full source code to make sure it's free of special controls by the owner that you need to potentially worry about. There's a careful balance as a developer between maintaining control over a DApp such that you can fix potential bugs, and building an owner-less platform that your users can trust to secure their data.

Put it to the test

Now we can restrict access to `setKittyContractAddress` so that no one but us can modify it in the future.

1. Add the `onlyOwner` modifier to `setKittyContractAddress`.

Chapter 4: Gas

Great! Now we know how to update key portions of the DApp while preventing other users from messing with our contracts.

Let's look at another way Solidity is quite different from other programming languages:

Gas — the fuel Ethereum DApps run on

In Solidity, your users have to pay every time they execute a function on your DApp using a currency called **gas**. Users buy gas with Ether (the currency on Ethereum), so your users have to spend ETH in order to execute functions on your DApp.

How much gas is required to execute a function depends on how complex that function's logic is. Each individual operation has a **gas cost** based roughly on how much computing resources will be required to perform that operation (e.g. writing to storage is much more expensive than adding two integers). The total **gas cost** of your function is the sum of the gas costs of all its individual operations.

Because running functions costs real money for your users, code optimization is much more important in Ethereum than in other programming languages. If your code is sloppy, your users are going to have to pay a premium to execute your functions — and this could add up to millions of dollars in unnecessary fees across thousands of users.

Why is gas necessary?

Ethereum is like a big, slow, but extremely secure computer. When you execute a function, every single node on the network needs to run that same function to verify its output — thousands of nodes verifying every function execution is what makes Ethereum decentralized, and its data immutable and censorship-resistant.

The creators of Ethereum wanted to make sure someone couldn't clog up the network with an infinite loop, or hog all the network resources with really intensive computations. So they made it so transactions aren't free, and users have to pay for computation time as well as storage.

Note: This isn't necessarily true for sidechains, like the ones the CryptoZombies authors are building at Loom Network. It probably won't ever make sense to run a game like World of Warcraft directly on the Ethereum mainnet — the gas costs would be prohibitively expensive. But it could run on a sidechain with a different consensus algorithm. We'll talk more about what types of DApps you would want to deploy on sidechains vs the Ethereum mainnet in a future lesson.

Struct packing to save gas

In Lesson 1, we mentioned that there are other types of `uint`s: `uint8`, `uint16`, `uint32`, etc.

Normally there's no benefit to using these sub-types because Solidity reserves 256 bits of storage regardless of the `uint` size. For example, using `uint8`instead of `uint` (`uint256`) won't save you any gas.

But there's an exception to this: inside `struct`s.

If you have multiple `uint`s inside a struct, using a smaller-sized `uint` when possible will allow Solidity to pack these variables together to take up less storage. For example:

```
struct NormalStruct {
  uint a;
  uint b;
  uint c;
}

struct MiniMe {
  uint32 a;
  uint32 b;
  uint c;
}

// `mini` will cost less gas than `normal` because of struct packing
NormalStruct normal = NormalStruct(10, 20, 30);
MiniMe mini = MiniMe(10, 20, 30);
```

For this reason, inside a struct you'll want to use the smallest integer sub-types you can get away with.

You'll also want to cluster identical data types together (i.e. put them next to each other in the struct) so that Solidity can minimize the required storage space. For example, a struct with fields `uint c; uint32 a; uint32 b;` will cost less gas than a struct with fields `uint32 a; uint c; uint32 b;` because the `uint32` fields are clustered together.

Put it to the test

In this lesson, we're going to add 2 new features to our zombies: `level` and `readyTime` — the latter will be used to implement a cooldown timer to limit how often a zombie can feed.

So let's jump back to `zombiefactory.sol`.

1. Add two more properties to our `Zombie` struct: `level` (a `uint32`), and `readyTime` (also a `uint32`). We want to pack these data types together, so let's put them at the end of the struct.

32 bits is more than enough to hold the zombie's level and timestamp, so this will save us some gas costs by packing the data more tightly than using a regular `uint` (256-bits).

Chapter 5: Time Units

The `level` property is pretty self-explanatory. Later on, when we create a battle system, zombies who win more battles will level up over time and get access to more abilities.

The `readyTime` property requires a bit more explanation. The goal is to add a "cooldown period", an amount of time a zombie has to wait after feeding or attacking before it's allowed to feed / attack again. Without this, the zombie could attack and multiply 1,000 times per day, which would make the game way too easy.

In order to keep track of how much time a zombie has to wait until it can attack again, we can use Solidity's time units.

Time units

Solidity provides some native units for dealing with time.

The variable `now` will return the current unix timestamp (the number of seconds that have passed since January 1st 1970). The unix time as I write this is `1515527488`.

Note: Unix time is traditionally stored in a 32-bit number. This will lead to the "Year 2038" problem, when 32-bit unix timestamps will overflow and break a lot of legacy systems. So if we wanted our DApp to keep running 20 years from now, we could use a 64-bit number instead — but our users would have to spend more gas to use our DApp in the meantime. Design decisions!

Solidity also contains the time units `seconds`, `minutes`, `hours`, `days`, `weeks` and `years`. These will convert to a `uint` of the number of seconds in that length of time. So `1 minutes` is `60`, `1 hours` is `3600` (60 seconds x 60 minutes), `1 days` is `86400` (24 hours x 60 minutes x 60 seconds), etc.

Here's an example of how these time units can be useful:

```
uint lastUpdated;

// Set `lastUpdated` to `now`
function updateTimestamp() public {
  lastUpdated = now;
}

// Will return `true` if 5 minutes have passed since `updateTimestamp` was
// called, `false` if 5 minutes have not passed
function fiveMinutesHavePassed() public view returns (bool) {
  return (now >= (lastUpdated + 5 minutes));
}
```

We can use these time units for our Zombie `cooldown` feature.

Put it to the test

Let's add a cooldown time to our DApp, and make it so zombies have to wait **1 day** after attacking or feeding to attack again.

1. Declare a `uint` called `cooldownTime`, and set it equal to `1 days`. (Forgive the poor grammar — if you set it equal to "1 day", it won't compile!)
2. Since we added a `level` and `readyTime` to our `Zombie` struct in the previous chapter, we need to update `_createZombie()` to use the correct number of arguments when we create a new `Zombie` struct.

   Update the `zombies.push` line of code to add 2 more arguments: `1` (for `level`), and `uint32(now + cooldownTime)` (for `readyTime`).

Note: The `uint32(...)` is necessary because `now` returns a `uint256` by default. So we need to explicitly convert it to a `uint32`.

`now + cooldownTime` will equal the current unix timestamp (in seconds) plus the number of seconds in 1 day — which will equal the unix timestamp 1 day from now. Later we can compare to see if this zombie's `readyTime` is greater than `now` to see if enough time has passed to use the zombie again.

We'll implement the functionality to limit actions based on `readyTime` in the next chapter.

Chapter 6: Zombie Cooldowns

Now that we have a `readyTime` property on our `Zombie` struct, let's jump to `zombiefeeding.sol` and implement a cooldown timer.

We're going to modify our `feedAndMultiply` such that:

1. Feeding triggers a zombie's cooldown, and
2. Zombies can't feed on kitties until their cooldown period has passed

This will make it so zombies can't just feed on unlimited kitties and multiply all day. In the future when we add battle functionality, we'll make it so attacking other zombies also relies on the cooldown.

First, we're going to define some helper functions that let us set and check a zombie's `readyTime`.

Passing structs as arguments

You can pass a storage pointer to a struct as an argument to a `private` or `internal` function. This is useful, for example, for passing around our `Zombie` structs between functions.

The syntax looks like this:

```
function _doStuff(Zombie storage _zombie) internal {
  // do stuff with _zombie
}
```

This way we can pass a reference to our zombie into a function instead of passing in a zombie ID and looking it up.

Put it to the test

1. Start by defining a `_triggerCooldown` function. It will take 1 argument, `_zombie`, a `Zombie storage` pointer. The function should be `internal`.
2. The function body should set `_zombie.readyTime` to `uint32(now + cooldownTime)`.
3. Next, create a function called `_isReady`. This function will also take a `Zombie storage` argument named `_zombie`. It will be an `internal view` function, and return a `bool`.
4. The function body should return `(_zombie.readyTime <= now)`, which will evaluate to either `true` or `false`. This function will tell us if enough time has passed since the last time the zombie fed.

Zombifeeding.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;

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

  // 1. Define `_triggerCooldown` function here

  // 2. Define `_isReady` function here

  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");
  }

}
```

Chapter 7: Public Functions & Security

Now let's modify `feedAndMultiply` to take our cooldown timer into account.

Looking back at this function, you can see we made it `public` in the previous lesson. An important security practice is to examine all your `public` and `external` functions, and try to think of ways users might abuse them. Remember — unless these functions have a modifier like `onlyOwner`, any user can call them and pass them any data they want to.

Re-examining this particular function, the user could call the function directly and pass in any `_targetDna` or `_species` they want to. This doesn't seem very game-like — we want them to follow our rules!

On closer inspection, this function only needs to be called by `feedOnKitty()`, so the easiest way to prevent these exploits is to make it `internal`.

Put it to the test

1. Currently `feedAndMultiply` is a `public` function. Let's make it `internal` so that the contract is more secure. We don't want users to be able to call this function with any DNA they want.
2. Let's make `feedAndMultiply` take our `cooldownTime` into account. First, after we look up `myZombie`, let's add a `require` statement that checks `_isReady()` and passes `myZombie` to it. This way the user can only execute this function if a zombie's cooldown time is over.
3. At the end of the function let's call `_triggerCooldown(myZombie)` so that feeding triggers the zombie's cooldown time.

Chapter 8: More on Function Modifiers

Great! Our zombie now has a functional cooldown timer.

Next, we're going to add some additional helper methods. We've created a new file for you called `zombiehelper.sol`, which imports `zombiefeeding.sol`. This will help to keep our code organized.

Let's make it so zombies gain special abilities after reaching a certain level. But in order to do that, first we'll need to learn a little bit more about function modifiers.

Function modifiers with arguments

Previously we looked at the simple example of `onlyOwner`. But function modifiers can also take arguments. For example:

```
// A mapping to store a user's age:
mapping (uint => uint) public age;

// Modifier that requires this user to be older than a certain age:
modifier olderThan(uint _age, uint _userId) {
  require(age[_userId] >= _age);
  _;
}

// Must be older than 16 to drive a car (in the US, at least).
// We can call the `olderThan` modifier with arguments like so:
function driveCar(uint _userId) public olderThan(16, _userId) {
  // Some function logic
}
```

You can see here that the `olderThan` modifier takes arguments just like a function does. And that the `driveCar` function passes its arguments to the modifier.

Let's try making our own `modifier` that uses the zombie `level` property to restrict access to special abilities.

Put it to the test

1. In `ZombieHelper`, create a `modifier` called `aboveLevel`. It will take 2 arguments, `_level` (a `uint`) and `_zombieId` (also a `uint`).
2. The body should check to make sure `zombies[_zombieId].level` is greater than or equal to `_level`.
3. Remember to have the last line of the modifier call the rest of the function with `_;`.

Zombiehelper.sol

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  // Start here

}
```

Chapter 9: Zombie Modifiers

Now let's use our `aboveLevel` modifier to create some functions.

Our game will have some incentives for people to level up their zombies:

- For zombies level 2 and higher, users will be able to change their name.
- For zombies level 20 and higher, users will be able to give them custom DNA.

We'll implement these functions below. Here's the example code from the previous lesson for reference:

```
// A mapping to store a user's age:
mapping (uint => uint) public age;

// Require that this user be older than a certain age:
modifier olderThan(uint _age, uint _userId) {
  require (age[_userId] >= _age);
  _;
}

// Must be older than 16 to drive a car (in the US, at least)
function driveCar(uint _userId) public olderThan(16, _userId) {
  // Some function logic
}
```

Put it to the test

1. Create a function called `changeName`. It will take 2 arguments: `_zombieId` (a `uint`), and `_newName` (a `string`), and make it `external`. It should have the `aboveLevel` modifier, and should pass in `2` for the `_level` parameter. (Don't forget to also pass the `_zombieId`).
2. In this function, first we need to verify that `msg.sender` is equal to `zombieToOwner[_zombieId]`. Use a `require` statement.
3. Then the function should set `zombies[_zombieId].name` equal to `_newName`.
4. Create another function named `changeDna` below `changeName`. Its definition and contents will be almost identical to `changeName`, except its second argument will be `_newDna` (a `uint`), and it should pass in `20`for the `_level` parameter on `aboveLevel`. And of course, it should set the zombie's `dna` to `_newDna` instead of setting the zombie's name.

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";
```

```
contract ZombieHelper is ZombieFeeding {

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

  // Start here

}
```

Chapter 10: Saving Gas With 'View' Functions

Awesome! Now we have some special abilities for higher-level zombies, to give our owners an incentive to level them up. We can add more of these later if we want to.

Let's add one more function: our DApp needs a method to view a user's entire zombie army — let's call it `getZombiesByOwner`.

This function will only need to read data from the blockchain, so we can make it a `view` function. Which brings us to an important topic when talking about gas optimization:

View functions don't cost gas

`view` functions don't cost any gas when they're called externally by a user.

This is because `view` functions don't actually change anything on the blockchain – they only read the data. So marking a function with `view` tells `web3.js` that it only needs to query your local Ethereum node to run the function, and it doesn't actually have to create a transaction on the blockchain (which would need to be run on every single node, and cost gas).

We'll cover setting up web3.js with your own node later. But for now the big takeaway is that you can optimize your DApp's gas usage for your users by using read-only `external view` functions wherever possible.

*Note: If a `view` function is called internally from another function in the same contract that is **not** a `view` function, it will still cost gas. This is because the other function creates a transaction on Ethereum, and will still need to be verified from every node. So `view` functions are only free when they're called externally.*

Put it to the test

We're going to implement a function that will return a user's entire zombie army. We can later call this function from `web3.js` if we want to display a user profile page with their entire army.

This function's logic is a bit complicated so it will take a few chapters to implement.

1. Create a new function named `getZombiesByOwner`. It will take one argument, an `address` named `_owner`.
2. Let's make it an `external view` function, so we can call it from `web3.js`without needing any gas.
3. The function should return a `uint[]` (an array of `uint`).

Leave the function body empty for now, we'll fill it in in the next chapter.

Chapter 11: Storage is Expensive

One of the more expensive operations in Solidity is using `storage` — particularly writes.

This is because every time you write or change a piece of data, it’s written permanently to the blockchain. Forever! Thousands of nodes across the world need to store that data on their hard drives, and this amount of data keeps growing over time as the blockchain grows. So there's a cost to doing that.

In order to keep costs down, you want to avoid writing data to storage except when absolutely necessary. Sometimes this involves seemingly inefficient programming logic — like rebuilding an array in `memory` every time a function is called instead of simply saving that array in a variable for quick lookups.

In most programming languages, looping over large data sets is expensive. But in Solidity, this is way cheaper than using `storage` if it's in an `external view`function, since `view` functions don't cost your users any gas. (And gas costs your users real money!).

We'll go over `for` loops in the next chapter, but first, let's go over how to declare arrays in memory.

Declaring arrays in memory

You can use the `memory` keyword with arrays to create a new array inside a function without needing to write anything to storage. The array will only exist until the end of the function call, and this is a lot cheaper gas-wise than updating an array in `storage` — free if it's a `view` function called externally.

Here's how to declare an array in memory:

```
function getArray() external pure returns(uint[]) {
  // Instantiate a new array in memory with a length of 3
  uint[] memory values = new uint[](3);
  // Add some values to it
  values.push(1);
  values.push(2);
  values.push(3);
  // Return the array
  return values;
}
```

This is a trivial example just to show you the syntax, but in the next chapter we'll look at combining this with `for` loops for real use-cases.

Note: memory arrays ***must*** *be created with a length argument (in this example, `3`). They currently cannot be resized like storage arrays can with `array.push()`, although this may be changed in a future version of Solidity.*

Put it to the test

In our `getZombiesByOwner` function, we want to return a `uint[]` array with all the zombies a particular user owns.

1. Declare a `uint[] memory` variable called `result`
2. Set it equal to a new `uint` array. The length of the array should be however many zombies this `_owner` owns, which we can look up from our `mapping` with: `ownerZombieCount[_owner]`.
3. At the end of the function return `result`. It's just an empty array right now, but in the next chapter we'll fill it in.

Chapter 12: For Loops

In the previous chapter, we mentioned that sometimes you'll want to use a `for` loop to build the contents of an array in a function rather than simply saving that array to storage.

Let's look at why.

For our `getZombiesByOwner` function, a naive implementation would be to store a `mapping` of owners to zombie armies in the `ZombieFactory` contract:

```
mapping (address => uint[]) public ownerToZombies
```

Then every time we create a new zombie, we would simply use `ownerToZombies[owner].push(zombieId)` to add it to that owner's zombies array. And `getZombiesByOwner` would be a very straightforward function:

```
function getZombiesByOwner(address _owner) external view returns (uint[]) {
  return ownerToZombies[_owner];
}
```

The problem with this approach

This approach is tempting for its simplicity. But let's look at what happens if we later add a function to transfer a zombie from one owner to another (which we'll definitely want to add in a later lesson!).

That transfer function would need to:

1. Push the zombie to the new owner's `ownerToZombies` array,
2. Remove the zombie from the old owner's `ownerToZombies` array,
3. Shift every zombie in the older owner's array up one place to fill the hole, and then
4. Reduce the array length by 1.

Step 3 would be extremely expensive gas-wise, since we'd have to do a write for every zombie whose position we shifted. If an owner has 20 zombies and trades away the first one, we would have to do 19 writes to maintain the order of the array.

Since writing to storage is one of the most expensive operations in Solidity, every call to this transfer function would be extremely expensive gas-wise. And worse, it would cost a different amount of gas each time it's called, depending on how many zombies the user has in their army and the index of the zombie being traded. So the user wouldn't know how much gas to send.

Note: Of course, we could just move the last zombie in the array to fill the missing slot and reduce the array length by one. But then we would change the ordering of our zombie army every time we made a trade.

Since `view` functions don't cost gas when called externally, we can simply use a for-loop in `getZombiesByOwner` to iterate the entire zombies array and build an array of the zombies that belong to this specific owner. Then our `transfer`function will be much cheaper, since we don't need to reorder any arrays in storage, and somewhat counter-intuitively this approach is cheaper overall.

Using `for` loops

The syntax of `for` loops in Solidity is similar to JavaScript.

Let's look at an example where we want to make an array of even numbers:

```
function getEvens() pure external returns(uint[]) {
  uint[] memory evens = new uint[](5);
  // Keep track of the index in the new array:
  uint counter = 0;
  // Iterate 1 through 10 with a for loop:
  for (uint i = 1; i <= 10; i++) {
    // If `i` is even...
    if (i % 2 == 0) {
      // Add it to our array
      evens[counter] = i;
      // Increment counter to the next empty index in `evens`:
      counter++;
    }
  }
  return evens;
}
```

This function will return an array with the contents `[2, 4, 6, 8, 10]`.

Put it to the test

Let's finish our `getZombiesByOwner` function by writing a `for` loop that iterates through all the zombies in our DApp, compares their owner to see if we have a match, and pushes them to our `result` array before returning it.

1. Declare a `uint` called `counter` and set it equal to `0`. We'll use this variable to keep track of the index in our `result` array.
2. Declare a `for` loop that starts from `uint i = 0` and goes up through `i < zombies.length`. This will iterate over every zombie in our array.
3. Inside the `for` loop, make an `if` statement that checks if `zombieToOwner[i]` is equal to `_owner`. This will compare the two addresses to see if we have a match.
4. Inside the `if` statement:
    1. Add the zombie's ID to our `result` array by setting `result[counter]` equal to `i`.
    2. Increment `counter` by 1 (see the `for` loop example above).

That's it — the function will now return all the zombies owned by `_owner`without spending any gas.

Zombiehelper.sol

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {
  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

  function changeName(uint _zombieId, string _newName) external aboveLevel(2,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20,
_zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].dna = _newDna;
  }

  function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    // Start here
    return result;
  }

}
```

Chapter 13: Wrapping It Up

Congratulations! That concludes Lesson 3.

Let's recap:

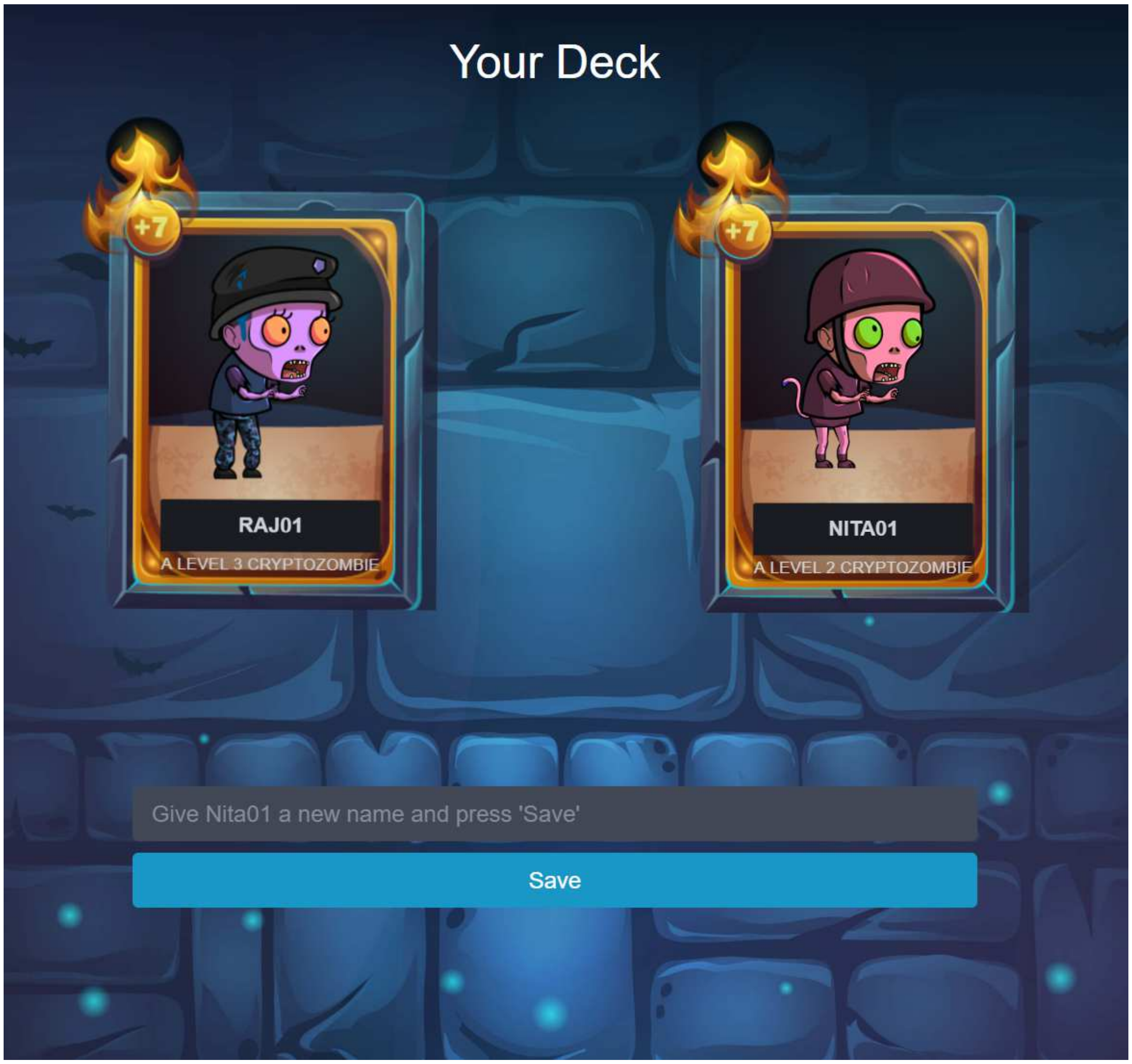
- We've added a way to update our CryptoKitties contracts
- We've learned to protect core functions with `onlyOwner`
- We've learned about gas and gas optimization
- We added levels and cooldowns to our zombies
- We now have functions to update a zombie's name and DNA once the zombie gets above a certain level
- And finally, we now have a function to return a user's zombie army

Claim your reward

As a reward for completing Lesson 3, both of your zombies have leveled up!

And now that NoName (the kitty-zombie you created in Lesson 2), is upgraded to level 2, you can call `changeName` to give him/her a name. NoName no more!

Go ahead and give NoName a name, then proceed to the next chapter to complete the lesson.


Your Deck
+7
RAJ01
A LEVEL 3 CRYPTOZOMBIE
+7
NITA01
A LEVEL 2 CRYPTOZOMBIE
Give Nita01 a new name and press 'Save'
Save