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.

Ownable Contracts

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 private _owner;

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

  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  constructor() internal {
    _owner = msg.sender;
    emit OwnershipTransferred(address(0), _owner);
  }

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

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

```
  /**
   * @return true if `msg.sender` is the owner of the contract.
   */
  function isOwner() public view returns(bool) {
    return msg.sender == _owner;
  }

  /**
   * @dev Allows the current owner to relinquish control of the contract.
   * @notice Renouncing to ownership will leave the contract without an owner.
   * It will not be possible to call the functions with the `onlyOwner`
   * modifier anymore.
   */
  function renounceOwnership() public onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
  }

  /**
   * @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 {
    _transferOwnership(newOwner);
  }

  /**
   * @dev Transfers control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0));
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
  }
}
```

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

- Constructors: `constructor()` 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.

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`:

```
pragma solidity >=0.5.0 <0.6.0;

/**
 * @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 private _owner;
```

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

/**
 * @dev The Ownable constructor sets the original `owner` of the contract to the sender
 * account.
 */
constructor() internal {
  _owner = msg.sender;
  emit OwnershipTransferred(address(0), _owner);
}

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

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

/**
 * @return true if `msg.sender` is the owner of the contract.
 */
function isOwner() public view returns(bool) {
  return msg.sender == _owner;
}

/**
 * @dev Allows the current owner to relinquish control of the contract.
 * @notice Renouncing to ownership will leave the contract without an owner.
 * It will not be possible to call the functions with the `onlyOwner`
 * modifier anymore.
 */
function renounceOwnership() public onlyOwner {
  emit OwnershipTransferred(_owner, address(0));
  _owner = address(0);
}
```

```
  /**
   * @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 {
    _transferOwnership(newOwner);
  }

  /**
   * @dev Transfers control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0));
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
  }
}
```

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

So while there are other ways you can use modifiers, one of the most common use-cases is to add a 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.

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 other blockchain, 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 blockchain with a different consensus algorithm. We'll talk more about what types of DApps you would want to deploy on Loom 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.

Time units

Solidity provides some native units for dealing with time.

The variable `now` will return the current unix timestamp of the latest block (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));
}
```

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
}
```

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.

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.

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[] memory) {
  // Instantiate a new array in memory with a length of 3
  uint[] memory values = new uint[](3);

  // Put some values to it
  values[0] = 1;
  values[1] = 2;
  values[2] = 3;

  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.

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[] memory) {
  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[] memory) {
  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]`.