Chapter 1: Payable

Up until now, we've covered quite a few **function modifiers**. It can be difficult to try to remember everything, so let's run through a quick review:

1. We have visibility modifiers that control when and where the function can be called from: private means it's only callable from other functions inside the contract; internal is like private but can also be called by contracts that inherit from this one; external can only be called outside the contract; and finally public can be called anywhere, both internally and externally.
2. We also have state modifiers, which tell us how the function interacts with the BlockChain: view tells us that by running the function, no data will be saved/changed. pure tells us that not only does the function not save any data to the blockchain, but it also doesn't read any data from the blockchain. Both of these don't cost any gas to call if they're called externally from outside the contract (but they do cost gas if called internally by another function).
3. Then we have custom modifiers, which we learned about in Lesson 3: onlyOwner and aboveLevel, for example. For these we can define custom logic to determine how they affect a function.

These modifiers can all be stacked together on a function definition as follows:

function test() external view onlyOwner anotherModifier { */\* ... \*/* }

In this chapter, we're going to introduce one more function modifier: payable.

The payable Modifier

payable functions are part of what makes Solidity and Ethereum so cool — they are a special type of function that can receive Ether.

Let that sink in for a minute. When you call an API function on a normal web server, you can't send US dollars along with your function call — nor can you send Bitcoin.

But in Ethereum, because both the money (*Ether*), the data (*transaction payload*), and the contract code itself all live on Ethereum, it's possible for you to call a function **and** pay money to the contract at the same time.

This allows for some really interesting logic, like requiring a certain payment to the contract in order to execute a function.

Let's look at an example

contract OnlineStore {

function buySomething() external payable {

*// Check to make sure 0.001 ether was sent to the function call:*

require(msg.value == 0.001 ether);

*// If so, some logic to transfer the digital item to the caller of the function:*

transferThing(msg.sender);

}

}

Here, msg.value is a way to see how much Ether was sent to the contract, and ether is a built-in unit.

What happens here is that someone would call the function from web3.js (from the DApp's JavaScript front-end) as follows:

*// Assuming `OnlineStore` points to your contract on Ethereum:*

OnlineStore.buySomething({from: web3.eth.defaultAccount, value: web3.utils.toWei(0.001)})

Notice the value field, where the javascript function call specifies how much ether to send (0.001). If you think of the transaction like an envelope, and the parameters you send to the function call are the contents of the letter you put inside, then adding a value is like putting cash inside the envelope — the letter and the money get delivered together to the recipient.

*Note: If a function is not marked payable and you try to send Ether to it as above, the function will reject your transaction.*

# Chapter 2: Withdraws

In the previous chapter, we learned how to send Ether to a contract. So what happens after you send it?

After you send Ether to a contract, it gets stored in the contract's Ethereum account, and it will be trapped there — unless you add a function to withdraw the Ether from the contract.

You can write a function to withdraw Ether from the contract as follows:

contract GetPaid is Ownable {

function withdraw() external onlyOwner {

owner.transfer(this.balance);

}

}

Note that we're using owner and onlyOwner from the Ownable contract, assuming that was imported.

You can transfer Ether to an address using the transfer function, and this.balance will return the total balance stored on the contract. So if 100 users had paid 1 Ether to our contract, this.balance would equal 100 Ether.

You can use transfer to send funds to any Ethereum address. For example, you could have a function that transfers Ether back to the msg.sender if they overpaid for an item:

uint itemFee = 0.001 ether;

msg.sender.transfer(msg.value - itemFee);

Or in a contract with a buyer and a seller, you could save the seller's address in storage, then when someone purchases his item, transfer him the fee paid by the buyer: seller.transfer(msg.value).

These are some examples of what makes Ethereum programming really cool — you can have decentralized marketplaces like this that aren't controlled by anyone.

Let's talk about **tokens**.

If you've been in the Ethereum space for any amount of time, you've probably heard people talking about tokens — specifically **ERC20 tokens**.

A **token** on Ethereum is basically just a smart contract that follows some common rules — namely it implements a standard set of functions that all other token contracts share, such as transfer(address \_to, uint256 \_value) and balanceOf(address \_owner).

Internally the smart contract usually has a mapping, mapping(address => uint256) balances, that keeps track of how much balance each address has.

So basically a token is just a contract that keeps track of who owns how much of that token, and some functions so those users can transfer their tokens to other addresses.

### **Why does it matter?**

Since all ERC20 tokens share the same set of functions with the same names, they can all be interacted with in the same ways.

This means if you build an application that is capable of interacting with one ERC20 token, it's also capable of interacting with any ERC20 token. That way more tokens can easily be added to your app in the future without needing to be custom coded. You could simply plug in the new token contract address, and boom, your app has another token it can use.

One example of this would be an exchange. When an exchange adds a new ERC20 token, really it just needs to add another smart contract it talks to. Users can tell that contract to send tokens to the exchange's wallet address, and the exchange can tell the contract to send the tokens back out to users when they request a withdraw.

The exchange only needs to implement this transfer logic once, then when it wants to add a new ERC20 token, it's simply a matter of adding the new contract address to its database.

### **Other token standards**

ERC20 tokens are really cool for tokens that act like currencies. But they're not particularly useful for representing zombies in our zombie game.

For one, zombies aren't divisible like currencies — I can send you 0.237 ETH, but transfering you 0.237 of a zombie doesn't really make sense.

Secondly, all zombies are not created equal. Your Level 2 zombie "**Steve**" is totally not equal to my Level 732 zombie "**H4XF13LD MORRIS 💯💯😎💯💯**". (Not even close, Steve).

There's another token standard that's a much better fit for crypto-collectibles like CryptoZombies — and they're called **ERC721 tokens.**

**ERC721 tokens** are **not** interchangeable since each one is assumed to be unique, and are not divisible. You can only trade them in whole units, and each one has a unique ID. So these are a perfect fit for making our zombies tradeable.

*Note that using a standard like ERC721 has the benefit that we don't have to implement the auction or escrow logic within our contract that determines how players can trade / sell our zombies. If we conform to the spec, someone else could build an exchange platform for crypto-tradable ERC721 assets, and our ERC721 zombies would be usable on that platform. So there are clear benefits to using a token standard instead of rolling your own trading logic.*

# 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.

# 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.

# 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 uints 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.

# 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 contracts 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 uints like +, but it also contains an assertstatement 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)*;*

# 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 uint16s, 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 uint16s and uint32s. We can call them SafeMath16 and SafeMath32.

The code will be exactly the same as SafeMath, except all instances of uint256will 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.

Great, now we can implement SafeMath on all the types of uints 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 😉).

# 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.

Chapter 1: Intro to Web3.js

By completing Lesson 5, our zombie DApp is now complete. Now we're going to create a basic web page where your users can interact with it.

To do this, we're going to use a JavaScript library from the Ethereum Foundation called **Web3.js**.

What is Web3.js?

Remember, the Ethereum network is made up of nodes, which each contain a copy of the blockchain. When you want to call a function on a smart contract, you need to query one of these nodes and tell it:

1. The address of the smart contract
2. The function you want to call, and
3. The variables you want to pass to that function.

Ethereum nodes only speak a language called **JSON-RPC**, which isn't very human-readable. A query to tell the node you want to call a function on a contract looks something like this:

// Yeah... Good luck writing all your function calls this way!

// Scroll right ==>

{"jsonrpc":"2.0","method":"eth\_sendTransaction","params":[{"from":"0xb60e8dd61c5d32be8058bb8eb970870f07233155","to":"0xd46e8dd67c5d32be8058bb8eb970870f07244567","gas":"0x76c0","gasPrice":"0x9184e72a000","value":"0x9184e72a","data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"}],"id":1}

Luckily, Web3.js hides these nasty queries below the surface, so you only need to interact with a convenient and easily readable JavaScript interface.

Instead of needing to construct the above query, calling a function in your code will look something like this:

CryptoZombies.methods.createRandomZombie("Vitalik Nakamoto 🤔")

.send({ from: "0xb60e8dd61c5d32be8058bb8eb970870f07233155", gas: "3000000" })

We'll explain the syntax in detail over the next few chapters, but first let's get your project set up with Web3.js.

Getting started

Depending on your project's workflow, you can add Web3.js to your project using most package tools:

*// Using NPM*

npm install web3

*// Using Yarn*

yarn add web3

*// Using Bower*

bower install web3

*// ...etc.*

Or you can simply download the minified .js file from [github](https://github.com/ethereum/web3.js/blob/1.0/dist/web3.min.js" \t "_blank) and include it in your project:

<script language="javascript" type="text/javascript" src="web3.min.js"></script>

Since we don't want to make too many assumptions about your development environment and what package manager you use, for this tutorial we're going to simply include Web3 in our project using a script tag as above.

[bies](https://cryptozombies.io/course)

 Autosave Enabled

[EN](https://cryptozombies.io/en/lesson/6/chapter/2)[中文](https://cryptozombies.io/zh/lesson/6/chapter/2)[POR](https://cryptozombies.io/pt/lesson/6/chapter/2)[日本](https://cryptozombies.io/jp/lesson/6/chapter/2)[ไทย](https://cryptozombies.io/th/lesson/6/chapter/2)[한국어](https://cryptozombies.io/ko/lesson/6/chapter/2)[FR](https://cryptozombies.io/fr/lesson/6/chapter/2)[RU](https://cryptozombies.io/ru/lesson/6/chapter/2)[ES](https://cryptozombies.io/es/lesson/6/chapter/2)

Sign Out

Chapter 2: Web3 Providers

Great! Now that we have Web3.js in our project, let's get it initialized and talking to the blockchain.

The first thing we need is a **Web3 Provider**.

Remember, Ethereum is made up of **nodes** that all share a copy of the same data. Setting a Web3 Provider in Web3.js tells our code **which node** we should be talking to handle our reads and writes. It's kind of like setting the URL of the remote web server for your API calls in a traditional web app.

You could host your own Ethereum node as a provider. However, there's a third-party service that makes your life easier so you don't need to maintain your own Ethereum node in order to provide a DApp for your users — **Infura**.

Infura

[Infura](https://infura.io/) is a service that maintains a set of Ethereum nodes with a caching layer for fast reads, which you can access for free through their API. Using Infura as a provider, you can reliably send and receive messages to/from the Ethereum blockchain without needing to set up and maintain your own node.

You can set up Web3 to use Infura as your web3 provider as follows:

var web3 = new Web3(new Web3.providers.WebsocketProvider("wss://mainnet.infura.io/ws"));

However, since our DApp is going to be used by many users — and these users are going to WRITE to the blockchain and not just read from it — we'll need a way for these users to sign transactions with their private key.

*Note: Ethereum (and blockchains in general) use a public / private key pair to digitally sign transactions. Think of it like an extremely secure password for a digital signature. That way if I change some data on the blockchain, I can****prove****via my public key that I was the one who signed it — but since no one knows my private key, no one can forge a transaction for me.*

Cryptography is complicated, so unless you're a security expert and you really know what you're doing, it's probably not a good idea to try to manage users' private keys yourself in our app's front-end.

But luckily you don't need to — there are already services that handle this for you. The most popular of these is **Metamask**.

Metamask

[Metamask](https://metamask.io/) is a browser extension for Chrome and Firefox that lets users securely manage their Ethereum accounts and private keys, and use these accounts to interact with websites that are using Web3.js. (If you haven't used it before, you'll definitely want to go and install it — then your browser is Web3 enabled, and you can now interact with any website that communicates with the Ethereum blockchain!).

And as a developer, if you want users to interact with your DApp through a website in their web browser (like we're doing with our CryptoZombies game), you'll definitely want to make it Metamask-compatible.

***Note****: Metamask uses Infura's servers under the hood as a web3 provider, just like we did above — but it also gives the user the option to choose their own web3 provider. So by using Metamask's web3 provider, you're giving the user a choice, and it's one less thing you have to worry about in your app.*

Using Metamask's web3 provider

Metamask injects their web3 provider into the browser in the global JavaScript object web3. So your app can check to see if web3 exists, and if it does use web3.currentProvider as its provider.

Here's some template code provided by Metamask for how we can detect to see if the user has Metamask installed, and if not tell them they'll need to install it to use our app:

window.addEventListener('load', function() {

*// Checking if Web3 has been injected by the browser (Mist/MetaMask)*

if (typeof web3 !== 'undefined') {

*// Use Mist/MetaMask's provider*

web3js = new Web3(web3.currentProvider);

} else {

*// Handle the case where the user doesn't have web3. Probably*

*// show them a message telling them to install Metamask in*

*// order to use our app.*

}

*// Now you can start your app & access web3js freely:*

startApp()

})

You can use this boilerplate code in all the apps you create in order to require users to have Metamask to use your DApp.

*Note: There are other private key management programs your users might be using besides MetaMask, such as the web browser****Mist****. However, they all implement a common pattern of injecting the variable web3, so the method we describe here for detecting the user's web3 provider will work for these as well.*

# Chapter 4: Calling Contract Functions

Our contract is all set up! Now we can use Web3.js to talk to it.

Web3.js has two methods we will use to call functions on our contract: call and send.

### **Call**

call is used for view and pure functions. It only runs on the local node, and won't create a transaction on the blockchain.

***Review:****view and pure functions are read-only and don't change state on the blockchain. They also don't cost any gas, and the user won't be prompted to sign a transaction with MetaMask.*

Using Web3.js, you would call a function named myMethod with the parameter 123 as follows:

myContract.methods.myMethod(123).call()

### **Send**

send will create a transaction and change data on the blockchain. You'll need to use send for any functions that aren't view or pure.

***Note:****sending a transaction will require the user to pay gas, and will pop up their Metamask to prompt them to sign a transaction. When we use Metamask as our web3 provider, this all happens automatically when we call send(), and we don't need to do anything special in our code. Pretty cool!*

Using Web3.js, you would send a transaction calling a function named myMethod with the parameter 123 as follows:

myContract.methods.myMethod(123).send()

The syntax is almost identical to call().

## Getting Zombie Data

Now let's look at a real example of using call to access data on our contract.

Recall that we made our array of zombies public:

Zombie[] public zombies*;*

In Solidity, when you declare a variable public, it automatically creates a public "getter" function with the same name. So if you wanted to look up the zombie with id 15, you would call it as if it were a function: zombies(15).

Here's how we would write a JavaScript function in our front-end that would take a zombie id, query our contract for that zombie, and return the result:

*Note: All the code examples we're using in this lesson are using****version 1.0****of Web3.js, which uses promises instead of callbacks. Many other tutorials you'll see online are using an older version of Web3.js. The syntax changed a lot with version 1.0, so if you're copying code from other tutorials, make sure they're using the same version as you!*

function getZombieDetails(id) {

return cryptoZombies.methods.zombies(id).call()

}

*// Call the function and do something with the result:*

getZombieDetails(15)

.then(function(result) {

console.log("Zombie 15: " + JSON.stringify(result));

});

Let's walk through what's happening here.

cryptoZombies.methods.zombies(id).call() will communicate with the Web3 provider node and tell it to return the zombie with index id from Zombie[] public zombies on our contract.

Note that this is **asynchronous**, like an API call to an external server. So Web3 returns a promise here. (If you're not familiar with JavaScript promises... Time to do some additional homework before continuing!)

Once the promise resolves (which means we got an answer back from the web3 provider), our example code continues with the then statement, which logs result to the console.

result will be a javascript object that looks like this:

{

"name": "H4XF13LD MORRIS'S COOLER OLDER BROTHER",

"dna": "1337133713371337",

"level": "9999",

"readyTime": "1522498671",

"winCount": "999999999",

"lossCount": "0" // Obviously.

}

We could then have some front-end logic to parse this object and display it in a meaningful way on the front-end.

# Chapter 5: Metamask & Accounts

Awesome! You've successfully written front-end code to interact with your first smart contract.

Now let's put some pieces together — let's say we want our app's homepage to display a user's entire zombie army.

Obviously we'd first need to use our function getZombiesByOwner(owner) to look up all the IDs of zombies the current user owners.

But our Solidity contract is expecting owner to be a Solidity address. How can we know the address of the user using our app?

## Getting the user's account in MetaMask

MetaMask allows the user to manage multiple accounts in their extension.

We can see which account is currently active on the injected web3 variable via:

var userAccount = web3.eth.accounts[0]

Because the user can switch the active account at any time in MetaMask, our app needs to monitor this variable to see if it has changed and update the UI accordingly. For example, if the user's homepage displays their zombie army, when they change their account in MetaMask, we'll want to update the page to show the zombie army for the new account they've selected.

We can do that with a setInterval loop as follows:

var accountInterval = setInterval(function() {

*// Check if account has changed*

if (web3.eth.accounts[0] !== userAccount) {

userAccount = web3.eth.accounts[0];

*// Call some function to update the UI with the new account*

updateInterface();

}

}, 100);

What this does is check every 100 milliseconds to see if userAccount is still equal web3.eth.accounts[0] (i.e. does the user still have that account active). If not, it reassigns userAccount to the currently active account, and calls a function to update the display.

## Put it to the Test

Let's make it so our app will display the user's zombie army when the page first loads, and monitor the active account in MetaMask to refresh the display if it changes.

1. Declare a var named userAccount, but don't assign it to anything.
2. At the end of startApp(), copy/paste the boilerplate accountIntervalcode from above
3. Replace the line updateInterface(); with a call to getZombiesByOwner, and pass it userAccount
4. Chain a then statement after getZombiesByOwner and pass the result to a function named displayZombies. (The syntax is: .then(displayZombies);).

We don't have a function called displayZombies yet, but we'll implement it in the next chapter.

# Chapter 6: Displaying our Zombie Army

This tutorial wouldn't be complete if we didn't show you how to actually display the data you get back from the contract.

However, realistically, you'll want to use a front-end framework like React or Vue.js in your app, since they make your life a lot easier as a front-end developer. But covering React or Vue.js is way outside the scope of this tutorial — that would be an entire tutorial of multiple lessons in itself.

So in order to keep CryptoZombies.io focused on Ethereum and smart contracts, we're just going to show a quick example in JQuery to demonstrate how you could parse and display the data you get back from your smart contract.

## Displaying zombie data — a rough example

We've added an empty <div id="zombies"></div> to the body of our document, as well as an empty displayZombies function.

Recall that in the previous chapter we called displayZombies from inside startApp() with the result of a call to getZombiesByOwner. It will be passed an array of zombie IDs that looks something like:

[0, 13, 47]

Thus we'll want our displayZombies function to:

1. First clear the contents of the #zombies div, if there's anything already inside it. (This way if the user changes their active MetaMask account, it will clear their old zombie army before loading the new one).
2. Loop through each id, and for each one call getZombieDetails(id) to look up all the information for that zombie from our smart contract, then
3. Put the information about that zombie into an HTML template to format it for display, and append that template to the #zombies div.

Again, we're just using JQuery here, which doesn't have a templating engine by default, so this is going to be ugly. But here's a simple example of how we could output this data for each zombie:

*// Look up zombie details from our contract. Returns a `zombie` object*

getZombieDetails(id)

.then(function(zombie) {

*// Using ES6's "template literals" to inject variables into the HTML.*

*// Append each one to our #zombies div*

$("#zombies").append(`<div class="zombie">

<ul>

<li>Name: ${zombie.name}</li>

<li>DNA: ${zombie.dna}</li>

<li>Level: ${zombie.level}</li>

<li>Wins: ${zombie.winCount}</li>

<li>Losses: ${zombie.lossCount}</li>

<li>Ready Time: ${zombie.readyTime}</li>

</ul>

</div>`);

});

## What about displaying the zombie sprites?

In the above example, we're simply displaying the DNA as a string. But in your DApp, you would want to convert this to images to display your zombie.

We did this by splitting up the DNA string into substrings, and having every 2 digits correspond to an image. Something like:

*// Get an integer 1-7 that represents our zombie head:*

var head = parseInt(zombie.dna.substring(0, 2)) % 7 + 1

*// We have 7 head images with sequential filenames:*

var headSrc = "../assets/zombieparts/head-" + i + ".png"

Each component is positioned with CSS using absolute positioning, to overlay it over the other images.

If you want to see our exact implementation, we've open sourced the Vue.js component we use for the zombie's appearance, which you can view [here](https://github.com/loomnetwork/zombie-char-component).

However, because there's a lot of code in that file, it's outside the scope of this tutorial. For this lesson, we'll stick with the extremely simple JQuery implementation above, and leave it to you to dive into a more beautiful implementation as homework 😉

# Chapter 7: Sending Transactions

Awesome! Now our UI will detect the user's metamask account, and automatically display their zombie army on the homepage.

Now let's look at using send functions to change data on our smart contract.

There are a few major differences from call functions:

1. sending a transaction requires a from address of who's calling the function (which becomes msg.sender in your Solidity code). We'll want this to be the user of our DApp, so MetaMask will pop up to prompt them to sign the transaction.
2. sending a transaction costs gas
3. There will be a significant delay from when the user sends a transaction and when that transaction actually takes effect on the blockchain. This is because we have to wait for the transaction to be included in a block, and the block time for Ethereum is on average 15 seconds. If there are a lot of pending transactions on Ethereum or if the user sends too low of a gas price, our transaction may have to wait several blocks to get included, and this could take minutes.

Thus we'll need logic in our app to handle the asynchronous nature of this code.

## Creating zombies

Let's look at an example with the first function in our contract a new user will call: createRandomZombie.

As a review, here is the Solidity code in our contract:

function createRandomZombie(string \_name) public {

require(ownerZombieCount[msg.sender] == 0);

uint randDna = \_generateRandomDna(\_name);

randDna = randDna - randDna % 100;

\_createZombie(\_name, randDna);

}

Here's an example of how we could call this function in Web3.js using MetaMask:

function createRandomZombie(name) {

*// This is going to take a while, so update the UI to let the user know*

*// the transaction has been sent*

$("#txStatus").text("Creating new zombie on the blockchain. This may take a while...");

*// Send the tx to our contract:*

return CryptoZombies.methods.createRandomZombie(name)

.send({ from: userAccount })

.on("receipt", function(receipt) {

$("#txStatus").text("Successfully created " + name + "!");

*// Transaction was accepted into the blockchain, let's redraw the UI*

getZombiesByOwner(userAccount).then(displayZombies);

})

.on("error", function(error) {

*// Do something to alert the user their transaction has failed*

$("#txStatus").text(error);

});

}

Our function sends a transaction to our Web3 provider, and chains some event listeners:

* receipt will fire when the transaction is included into a block on Ethereum, which means our zombie has been created and saved on our contract
* error will fire if there's an issue the prevented the transaction from being included in a block, such as the user not sending enough gas. We'll want to inform the user in our UI that the transaction didn't go through so they can try again.

*Note: You can optionally specify gas and gasPrice when you call send, e.g. .send({ from: userAccount, gas: 3000000 }). If you don't specify this, MetaMask will let the user choose these values.*

# Chapter 8: Calling Payable Functions

The logic for attack, changeName, and changeDna will be extremely similar, so they're trivial to implement and we won't spend time coding them in this lesson.

*In fact, there's already a lot of repetitive logic in each of these function calls, so it would probably make sense to refactor and put the common code in its own function. (And use a templating system for the txStatus messages — already we're seeing how much cleaner things would be with a framework like Vue.js!)*

Let's look at another type of function that requires special treatment in Web3.js — payable functions.

## Level Up!

Recall in ZombieHelper, we added a payable function where the user can level up:

function levelUp(uint \_zombieId) external payable {

require(msg.value == levelUpFee);

zombies[\_zombieId].level++;

}

The way to send Ether along with a function is simple, with one caveat: we need to specify how much to send in wei, not Ether.

## What's a Wei?

A wei is the smallest sub-unit of Ether — there are 10^18 wei in one ether.

That's a lot of zeroes to count — but luckily Web3.js has a conversion utility that does this for us.

*// This will convert 1 ETH to Wei*

web3js.utils.toWei("1", "ether");

In our DApp, we set levelUpFee = 0.001 ether, so when we call our levelUpfunction, we can make the user send 0.001 Ether along with it using the following code:

CryptoZombies.methods.levelUp(zombieId)

.send({ from: userAccount, value: web3js.utils.toWei("0.001", "ether") })

# Chapter 9: Subscribing to Events

As you can see, interacting with your contract via Web3.js is pretty straightforward — once you have your environment set up, calling functions and sending transactions is not all that different from a normal web API.

There's one more aspect we want to cover — subscribing to events from your contract.

## Listening for New Zombies

If you recall from zombiefactory.sol, we had an event called NewZombie that we fired every time a new zombie was created:

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

In Web3.js, you can **subscribe** to an event so your web3 provider triggers some logic in your code every time it fires:

cryptoZombies.events.NewZombie()

.on("data", function(event) {

let zombie = event.returnValues;

*// We can access this event's 3 return values on the `event.returnValues` object:*

console.log("A new zombie was born!", zombie.zombieId, zombie.name, zombie.dna);

}).on("error", console.error);

Note that this would trigger an alert every time ANY zombie was created in our DApp — not just for the current user. What if we only wanted alerts for the current user?

## Using indexed

In order to filter events and only listen for changes related to the current user, our Solidity contract would have to use the indexed keyword, like we did in the Transfer event of our ERC721 implementation:

event Transfer(address indexed \_from, address indexed \_to, uint256 \_tokenId);

In this case, because \_from and \_to are indexed, that means we can filter for them in our event listener in our front end:

*// Use `filter` to only fire this code when `\_to` equals `userAccount`*

cryptoZombies.events.Transfer({ filter: { \_to: userAccount } })

.on("data", function(event) {

let data = event.returnValues;

*// The current user just received a zombie!*

*// Do something here to update the UI to show it*

}).on("error", console.error);

As you can see, using events and indexed fields can be quite a useful practice for listening to changes to your contract and reflecting them in your app's front-end.

## Querying past events

We can even query past events using getPastEvents, and use the filters fromBlock and toBlock to give Solidity a time range for the event logs ("block" in this case referring to the Ethereum block number):

cryptoZombies.getPastEvents("NewZombie", { fromBlock: 0, toBlock: "latest" })

.then(function(events) {

*// `events` is an array of `event` objects that we can iterate, like we did above*

*// This code will get us a list of every zombie that was ever created*

});

Because you can use this method to query the event logs since the beginning of time, this presents an interesting use case: **Using events as a cheaper form of storage**.

If you recall, saving data to the blockchain is one of the most expensive operations in Solidity. But using events is much much cheaper in terms of gas.

The tradeoff here is that events are not readable from inside the smart contract itself. But it's an important use-case to keep in mind if you have some data you want to be historically recorded on the blockchain so you can read it from your app's front-end.

For example, we could use this as a historical record of zombie battles — we could create an event for every time one zombie attacks another and who won. The smart contract doesn't need this data to calculate any future outcomes, but it's useful data for users to be able to browse from the app's front-end.

# Chapter 10: Wrapping It Up

Congratulations! You've successfully written your first Web3.js front-end that interacts with your smart contract.

As a reward, you get your very own The Phantom of Web3 zombie! Level 3.0 (for Web 3.0 😉), complete with fox mask. Check him out to the right.

## Next Steps

This lesson was intentionally basic. We wanted to show you the core logic you would need in order to interact with your smart contract, but didn't want to take up too much time in order to do a full implementation since the Web3.js portion of the code is quite repetitive, and we wouldn't be introducing any new concepts by making this lesson any longer.

So we've left this implementation bare-bones. Here's a checklist of ideas for things we would want to implement in order to make our front-end a full implementation for our zombie game, if you want to run with this and build it on your own:

1. Implementing functions for attack, changeName, changeDna, and the ERC721 functions transfer, ownerOf, balanceOf, etc. The implementation of these functions would be identical to all the other sendtransactions we covered.
2. Implementing an "admin page" where you can execute setKittyContractAddress, setLevelUpFee, and withdraw. Again, there's no special logic on the front-end here — these implementations would be identical to the functions we've already covered. You would just have to make sure you called them from the same Ethereum address that deployed the contract, since they have the onlyOwner modifier.
3. There are a few different views in the app we would want to implement:

a. An individual zombie page, where you can view info about a specific zombie with a permalink to it. This page would render the zombie's appearance, show its name, its owner (with a link to the user's profile page), its win/loss count, its battle history, etc.

b. A user page, where you could view a user's zombie army with a permalink. You would be ableto click on an individual zombie to view its page, and also click on a zombie to attack it if you're logged into MetaMask and have an army.

c. A homepage, which is a variation of the user page that shows the current user's zombie army. (This is the page we started implementing in index.html).

1. Some method in the UI that allows the user to feed on CryptoKitties. We could have a button by each zombie on the homepage that says "Feed Me", then a text box that prompted the user to enter a kitty's ID (or a URL to that kitty, e.g. <https://www.cryptokitties.co/kitty/578397>). This would then trigger our function feedOnKitty.
2. Some method in the UI for the user to attack another user's zombie.

One way to implement this would be when the user was browsing another user's page, there could be a button that said "Attack This Zombie". When the user clicked it, it would pop up a modal that contains the current user's zombie army and prompt them "Which zombie would you like to attack with?"

The user's homepage could also have a button by each of their zombies that said "Attack a Zombie". When they clicked it, it could pop up a modal with a search field where they could type in a zombie's ID to search for it. Or an option that said "Attack Random Zombie", which would search a random number for them.

We would also want to grey out the user's zombies whose cooldown period had not yet passed, so the UI could indicate to the user that they can't yet attack with that zombie, and how long they will have to wait.

1. The user's homepage would also have options by each zombie to change name, change DNA, and level up (for a fee). Options would be greyed out if the user wasn't yet high enough level.
2. For new users, we should display a welcome message with a prompt to create the first zombie in their army, which calls createRandomZombie().
3. We'd probably want to add an Attack event to our smart contract with the user's address as an indexed property, as discussed in the last chapter. This would allow us to build real-time notifications — we could show the user a popup alert when one of their zombies was attacked, so they could view the user/zombie who attacked them and retaliate.
4. We would probably also want to implement some sort of front-end caching layer so we aren't always slamming Infura with requests for the same data. (Our current implementation of displayZombies calls getZombieDetailsfor every single zombie every time we refresh the interface — but realistically we only need to call this for the new zombie that's been added to our army).
5. A real-time chat room so you could trash talk other players as you crush their zombie army? Yes plz.

That's just a start — I'm sure we could come up with even more features — and already it's a massive list.

Since there's a lot of front-end code that would go into creating a full interface like this (HTML, CSS, JavaScript and a framework like React or Vue.js), building out this entire front-end would probably be an entire course with 10 lessons in itself. So we'll leave the awesome implementation to you.

*Note: Even though our smart contract is decentralized, this front-end for interacting with our DApp would be totally centralized on our web-server somewhere.*

*However, with the SDK we're building at*[*Loom Network*](https://medium.com/loom-network/loom-network-is-live-scalable-ethereum-dapps-coming-soon-to-a-dappchain-near-you-29d26da00880)*, soon you'll be able to serve front-ends like this from their own DAppChain instead of a centralized web server. That way between Ethereum and the Loom DAppChain, your entire app would run 100% on the blockchain.*

## Conclusion

This concludes Lesson 6. You now have all the skills you need to code a smart contract and a front-end that allows users to interact with it!

In the next lesson, we're going to be covering the final missing piece in this puzzle — deploying your smart contracts to Ethereum.

Go ahead and click "Next Chapter" to claim your rewards!

Cryptocurrency and Decentralized Applications