**Visibility Modifiers – 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;

**public** can be called anywhere, both internally and externally.

**msg.sender**

In Solidity, there are certain global variables that are available to all functions. One of these is msg.sender, which refers to the address of the person (or smart contract) who called the current function.

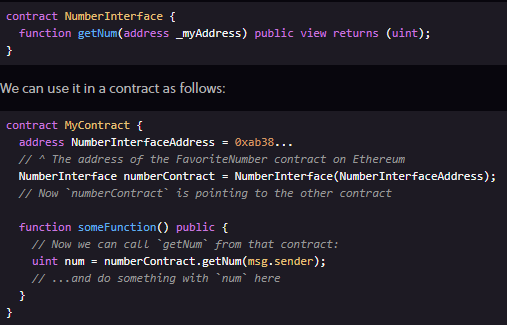
Using msg.sender gives you the security of the Ethereum blockchain — the only way someone can modify someone else's data would be to steal the private key associated with their Ethereum address.

**Interfaces**

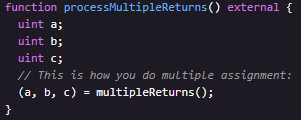
Notice that this looks like defining a contract, with a few differences. For one, we're only declaring the functions we want to interact with — in this case getNum — and we don't mention any of the other functions or state variables.

Secondly, we're not defining the function bodies. Instead of curly braces ({ and }), we're simply ending the function declaration with a semi-colon (;).

So it kind of looks like a contract skeleton. This is how the compiler knows it's an interface.



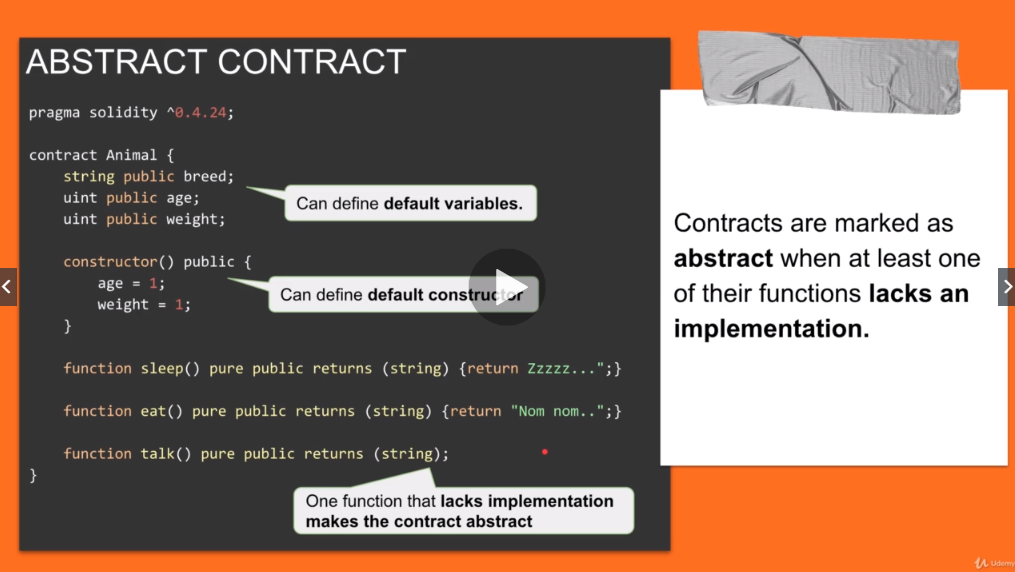
**Process multiple return syntax**

****

**uint == uint256 (same thing)**

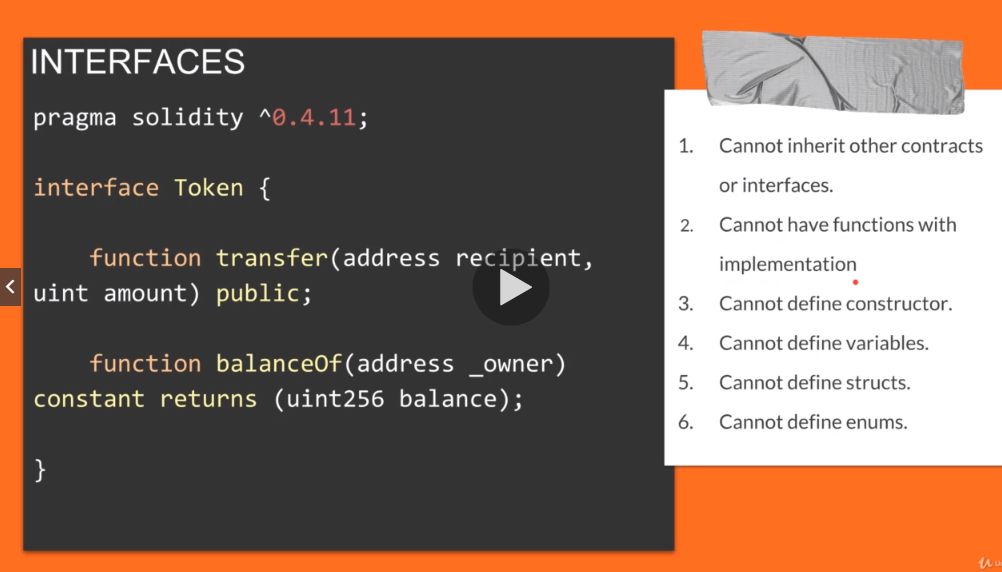
**Abstract contracts**

are base **contracts** in which at least one of their functions lacks an implementation. Other **contracts** can inherit from **abstract contracts**.



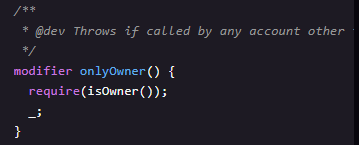
**Interfaces**

are similar to **abstract contracts** in that they help foster extensibility, however, they cannot include any implemented functions. Other contracts can inherit multiple interfaces

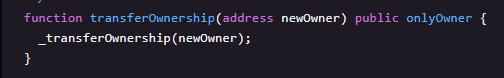
****

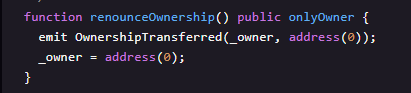
**Function/Custom Modifiers – we can define custom logic to determine how they affect a function**

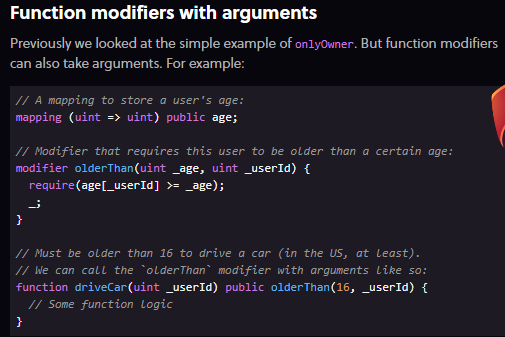
modifier onlyOwner(). Modifiers are kind of half-functions that are used to modify other functions, usually to check some requirements prior to execution. In this case, onlyOwner can be used to limit access so only the owner of the contract can run this function.



\_; means to proceed with the code





**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.

**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.

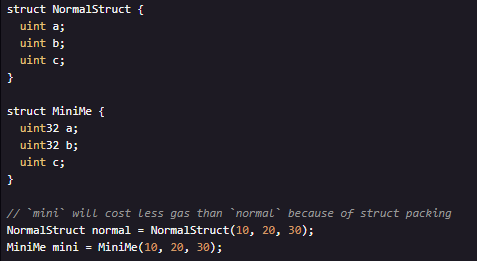
The creators of Ethereum wanted to make sure someone couldn't clog up the network with an infinite loop, or hog all the network resources with really intensive computations. So they made it so transactions aren't free, and users have to pay for computation time as well as storage.

**Struct packing to save gas**

In Lesson 1, we mentioned that there are other types of uints: 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 structs**.

If you have multiple uints 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:

**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.

**now** – returns a uint256 by default

***Passing structs as arguments***

*You can pass a storage pointer to a struct as an argument to a****private****or****internal function****. This way we can pass a reference to our zombie into a function instead of passing in a zombie ID and looking it up.*

***calldata***

*Note: calldata is somehow similar to memory, but it's only available to external functions.*

**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).

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.

**For Loops (VERY IMPORTANT TO OPTIMISE FOR GAS)**

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:



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.

**State Modifiers – 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)

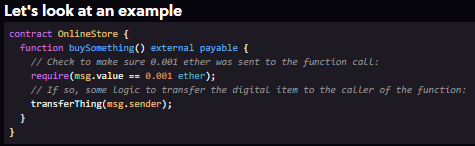
**Payable Modifie**r

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.



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:

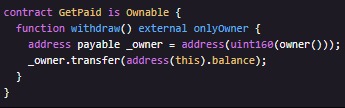


Notice the value field, where the javascript function call specifies how much ether to send (0.001). If you think of the transaction like an envelope, and the parameters you send to the function call are the contents of the letter you put inside, then adding a value is like putting cash inside the envelope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

**Withdraws**

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



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

And most important for \_owner variable that it's have to be an address payable type for doing a sending and transferring ether instruction.

But our owner() isn't a type address payable so we have to explicitly cast to address payable. Casting any integer type like uint160 to address produces an address payable.

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

**OpenZeppelin**

* 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.
* SafeMath
* 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)*;*

**Address 0**

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

**Assert vs Require**

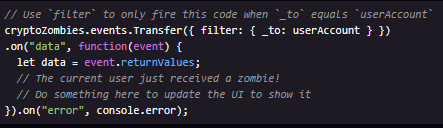
assert is similar to require, where it will throw an error if false. The difference between assert and require is that require will refund the user the rest of their gas when a function fails, whereas assert will not. So most of the time you want to use require in your code; assert is typically used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

**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:

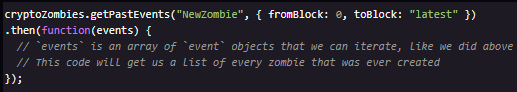
event Transfer(address indexed \_from, address indexed \_to, uint256 \_tokenId);

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



**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):



**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.