Notes on Solidity

* Solidity's code is encapsulated in contracts. A contract is the fundamental building block of Ethereum applications — all variables and functions belong to a contract, and this will be the starting point of all your projects.

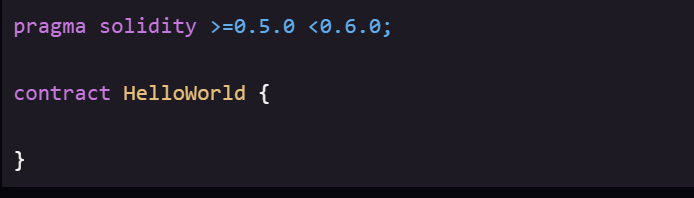
contract HelloWorld {

}

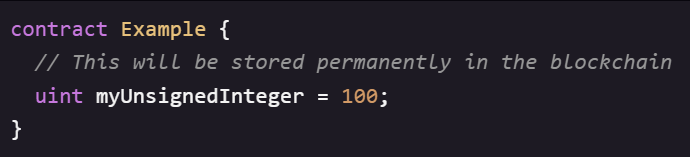
An empty contract HelloWorld would look like this.

* The solidity code should start with a “version pragma” which is a declaration of the version of the Solidity compiler to be used by a code. This is done so that the code never breaks in case we have another version of compiler in the future.

Here we have used pragma solidity >=0. 5. 0 <0. 6. 0

****

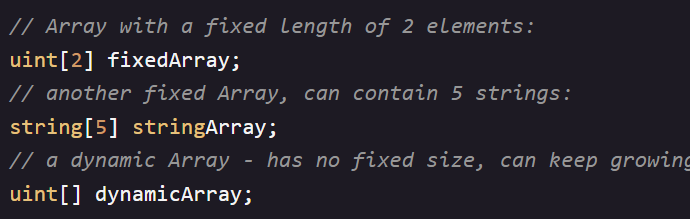
* The state variables are permanently stored in contract storage which means they are written to Ethereum blockchain.
* The uint data type is an unsigned integer that is its value must be non -negative.
* As for data types in solidity we have:
* For value types we have:
* Boolean: This data type as all other languages accepts only two values which are True or False.
* Integer: This data type is used to store integer values. In Solidity we avoid using negative values because blockchain does not deal with negative values neither with fractional values.
* Fixed Point Numbers: They are not entirely supported but can be declared as fixed and unfixed for signed and unsigned fixed-point numbers of varying sizes respectively.
* Address: It holds a 20-byte value which represents the size of an Ethereum address. An address can be used to get balance or to transfer a balance by balance and transfer method respectively.
* Bytes and Strings: Bytes are used to store a fixed-sized character set while the string is used for storing the character set equal to or more than a byte.
* Enums: It is used to create user-defined data types, used to assign a name to an integral constant which makes the contract more readable, maintainable, and less prone to errors. Options of enums can be represented by unsigned integer values starting from 0.



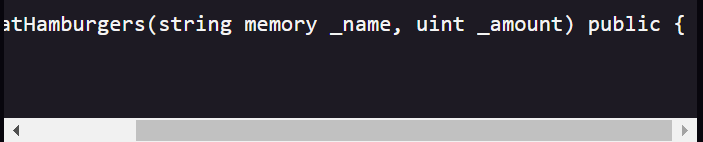
* In Solidity, uint is actually an alias for uint256, a 256-bit unsigned integer. You can declare uints with less bits — uint8, uint16, uint32, etc. But in general, you want to simply use uint except in specific cases.
* This is how structs look like:



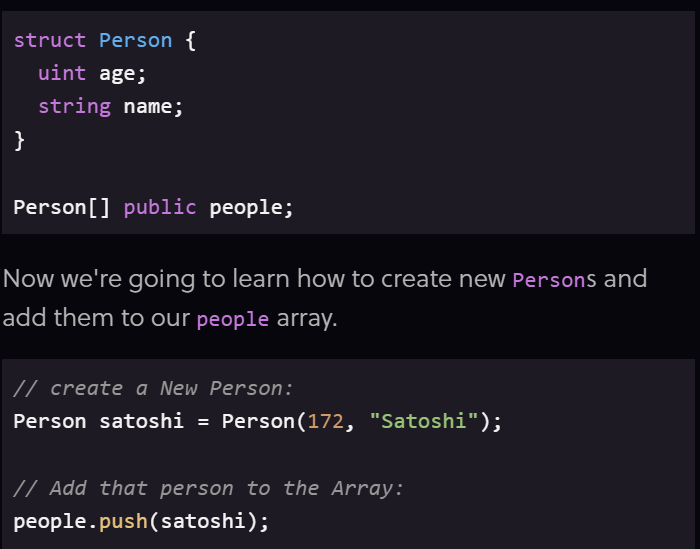
* There are two types of arrays in solidity that is fixed arrays and dynamic arrays.



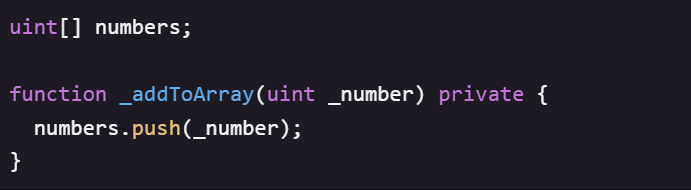
* Remember that state variables are stored permanently in the blockchain? So creating a dynamic array of structs like this can be useful for storing structured data in your contract, kind of like a database.
* Creating a public array is useful for storing public data in the contract so that other contracts would be able to read from but not write to this array.



* Here we have specified that where the variable name should be stored.
* In solidity we can pass a value in the function by value or by reference.
* We can use underscore to differentiate function variable names with the global variables.

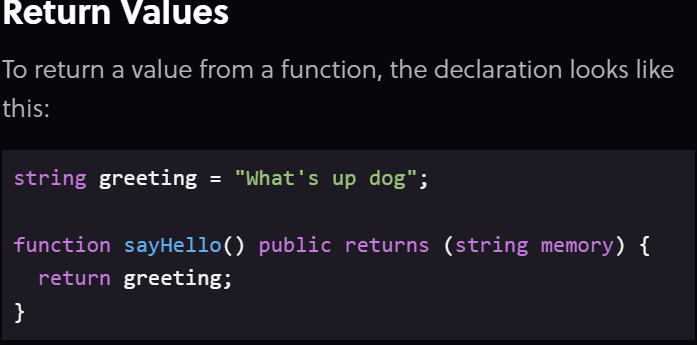


* In solidity the functions are by default public.
* This might not be good because if the function is public, the contracts might be open to attacks.

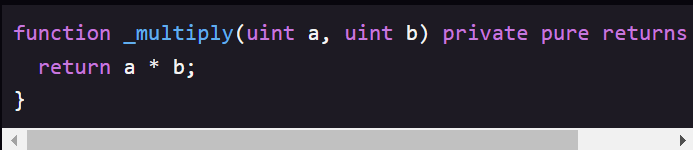


* It is a convention to start a private function using ( \_ ) before the function name.

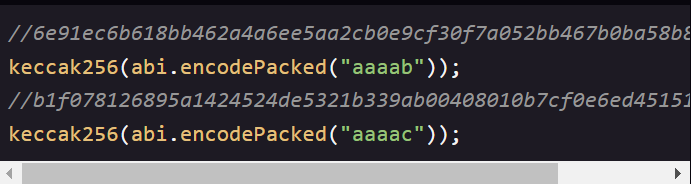
Function returning values.



* The function declaration contains the type of return value of the function.
* A function with view keyword can be only viewed and cannot be modified.

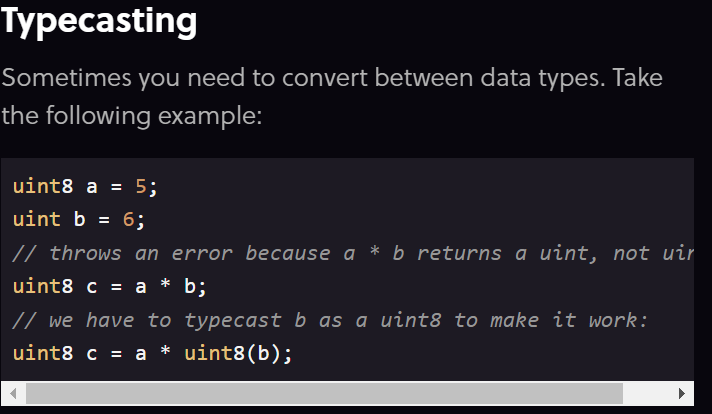


* A pure function means that we are not accessing any data in the app, its return value depends on the function parameter.
* The solidity compiler automatically tells us when to use view or public or any other modifier.
* Ethereum has the hash function keccack256 built in, which is a version of SHA3.
* A hash function basically maps an input into a random 256 – bit hexadecimal number. A slight change in the input will cause a large change in the hash.
* Also important, keccak256 expects a single parameter of type bytes. This means that we have to "pack" any parameters before calling keccak256.

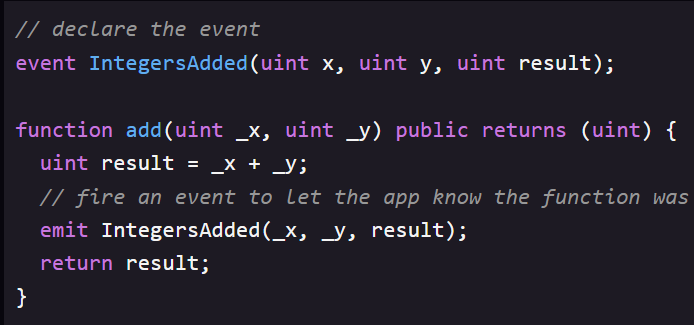


Only 1 character change in the input can return totally different values.

* Secure random number generation in blockchain is a very difficult problem.



* Events are a way for our contract to communicate that something happened on the blockchain to your app front end, which can be ‘listening’ for certain events and take action when they happen.



* Ethereum has a JavaScript library called Web3.js .
* We need JavaScript frontend that can interact with our contracts.

Here 's how we would access our contract:

var abi = /\* abi generated by the compiler \*/

    var ZombieFactoryContract = web3.eth.contract(abi)

var contractAddress = /\* our contract address on Ethereum after deploying \*/

    var ZombieFactory = ZombieFactoryContract.at(contractAddress)

        // `ZombieFactory` has access to our contract's public functions and events

// some sort of event listener to take the text input:

$("#ourButton").click(function(e) {

    var name = $("#nameInput").val()

        // Call our contract's `createRandomZombie` function:

    ZombieFactory.createRandomZombie(name)

})

// Listen for the `NewZombie` event, and update the UI

var event = ZombieFactory.NewZombie(function(error, result) {

    if (error) return

    generateZombie(result.zombieId, result.name, result.dna)

})

// take the Zombie dna, and update our image

function generateZombie(id, name, dna) {

    let dnaStr = String(dna)

        // pad DNA with leading zeroes if it's less than 16 characters

    while (dnaStr.length < 16)

        dnaStr = "0" + dnaStr

    let zombieDetails = {

        // first 2 digits make up the head. We have 7 possible heads, so % 7

        // to get a number 0 - 6, then add 1 to make it 1 - 7. Then we have 7

        // image files named "head1.png" through "head7.png" we load based on

        // this number:

        headChoice: dnaStr.substring(0, 2) % 7 + 1,

        // 2nd 2 digits make up the eyes, 11 variations:

        eyeChoice: dnaStr.substring(2, 4) % 11 + 1,

        // 6 variations of shirts:

        shirtChoice: dnaStr.substring(4, 6) % 6 + 1,

        // last 6 digits control color. Updated using CSS filter: hue-rotate

        // which has 360 degrees:

        skinColorChoice: parseInt(dnaStr.substring(6, 8) / 100 \* 360),

        eyeColorChoice: parseInt(dnaStr.substring(8, 10) / 100 \* 360),

        clothesColorChoice: parseInt(dnaStr.substring(10, 12) / 100 \* 360),

        zombieName: name,

        zombieDescription: "A Level 1 CryptoZombie",

    }

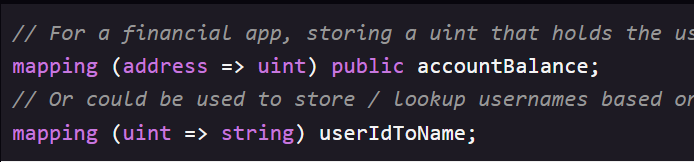
    return zombieDetails

}

What JavaScript does is that it takes the values generated in zombieDetails above and use some browser based JavaScript magic like Vue.js to swap out the images and apply CSS filters.

* The Ethereum blockchain is made up of accounts, just like bank accounts. An account has a balance of Ether (the currency used on the Ethereum blockchain), and we can send and receive Ether payments to other accounts, just like the bank account money transfer.
* Each account has an address, which we can think of like a bank account number. It is a unique identifier that points to that account, and it looks like this.
* It is a unique identifier that points to that account, and it looks like this:

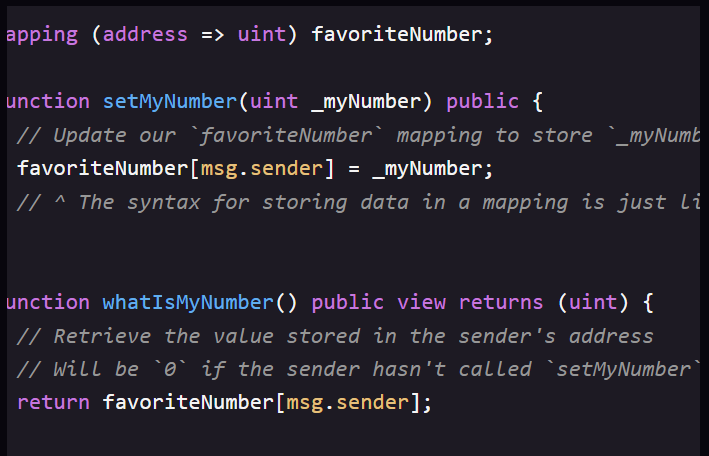
0x0cE446255506E92DF41614C46F1d6df9Cc969183



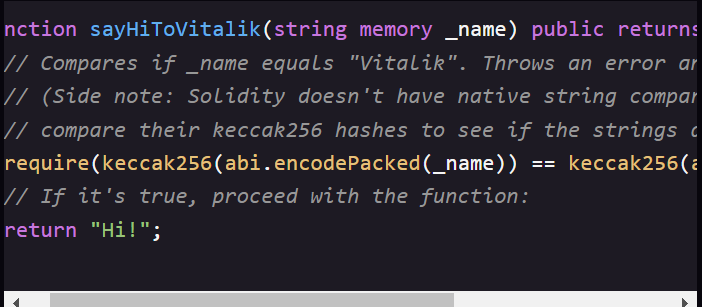
Syntax of mapping

Mapping is essentially a key value store for storing and looking up data. Here, in first example address is a key and uint is a value and in the second example uint is a key and string is a value.

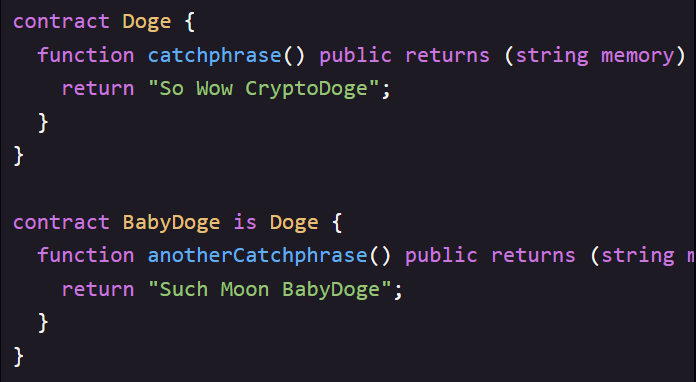
* In solidity, there are certain global variables that are available to all functions. One of these is msg. sender, which refers to the address of the person or smart contracts who called the current function.
* In solidity, function execution always needs to start with an external caller. A contract will just sit on the blockchain doing nothing until someone calls one of its functions. So, there will always be a msg. sender.



* In the above example, anyone could call setMyNumber and store a uint in our contract, which would be tied to their address. And when they called whatIsMyNumber, they would be returned the uint that they stored.
* Using msg.sender gives us the security of the Ethereum blockchain - the only way to modify someone else’s data would be to steal the private key associated with their Ethereum address.
* For checking a condition in solidity, we use require. It throws an error and stops executing if some condition is not true.

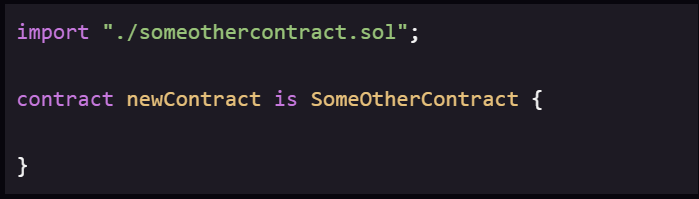


* If we call this function with sayHiToVitalik (“ Vitalik “). It will return “Hi” and for any other input, it will throw an error and not execute.

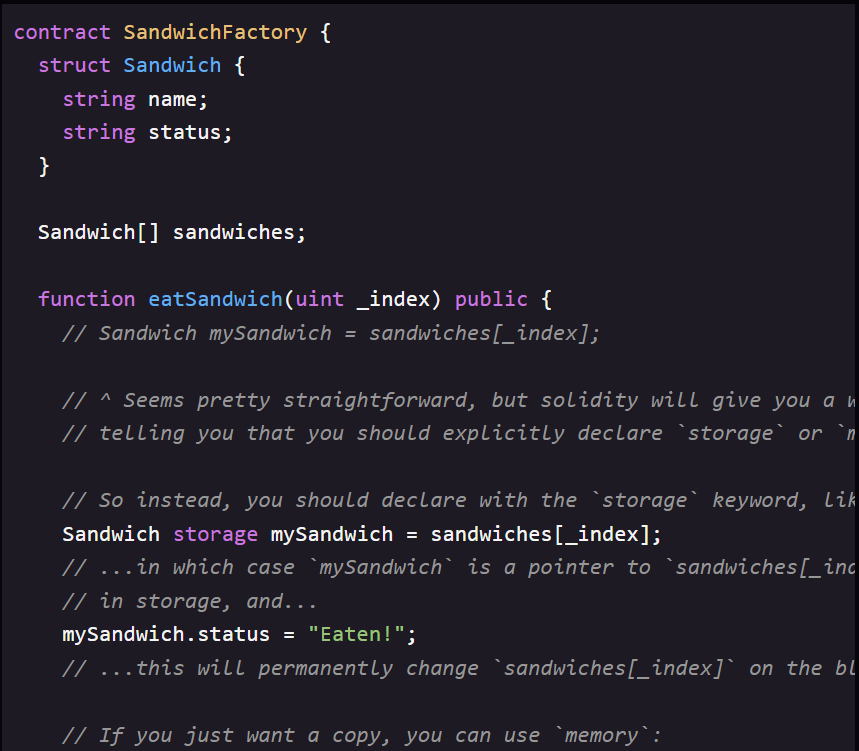


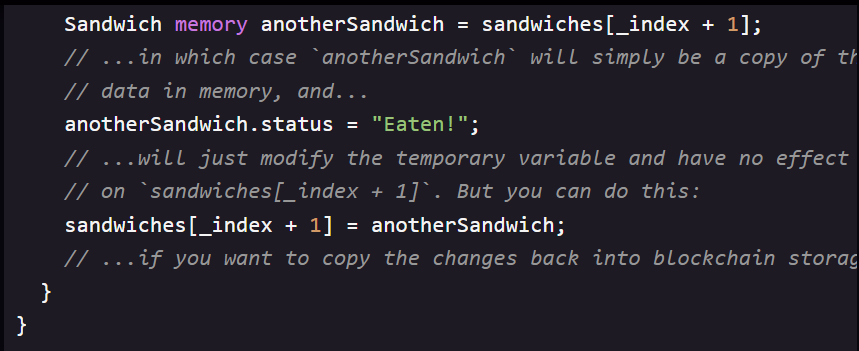
* Example of inheritance in solidity.
* BabyDoge inherits from Doge and it will have access to both catchphrase( )

and anothercatchphrase( ).

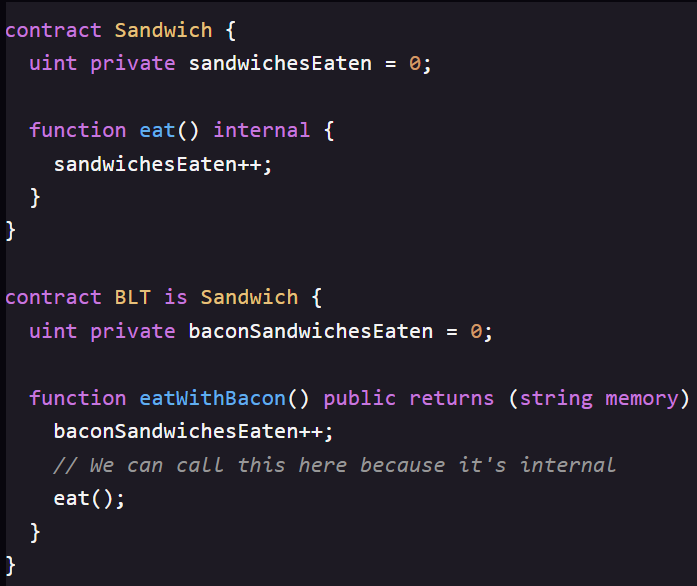


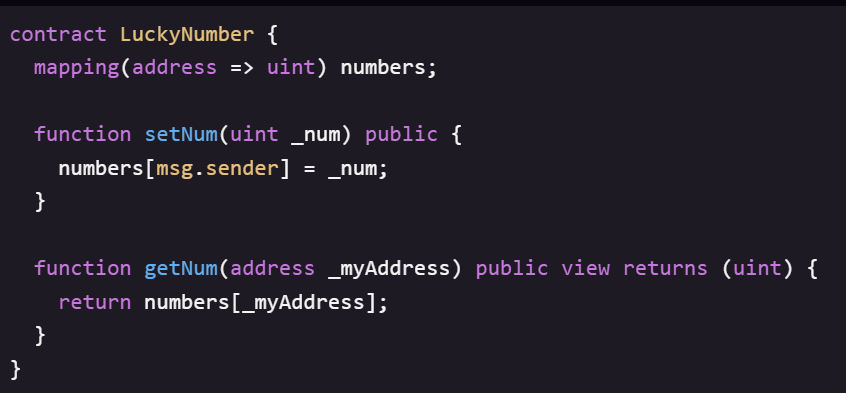
* Meaning of ( . /) is that the file someothercontract.sol is in same directory as the current contract.
* Storage refers to variables stored permanently on the blockchain and memory variables are temporary, and are erased between external function calls to our contract. We can think of it like our computer’s hard disk VS RAM.
* State variables are by default storage and written permanently to the blockchain, while variables declared inside functions are memory and will disappear when the function call ends.
* Most of the time, we do not need to use these keywords because solidity handles them by default but in some situations like dealing with structs and arrays within functions we might need to specify.



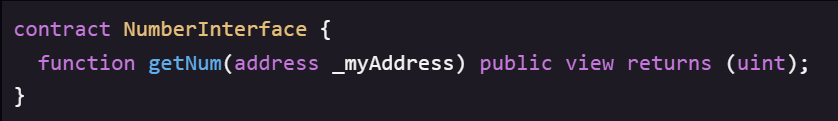


* In addition to public and private, solidity has two more types of visibility for functions: internal and external.
* Internal is same as private, except that it is also accessible to contracts that inherit from this contract.
* External is similar to public, except that these functions can only be called outside the contract - they cannot be called by other functions inside the contract.



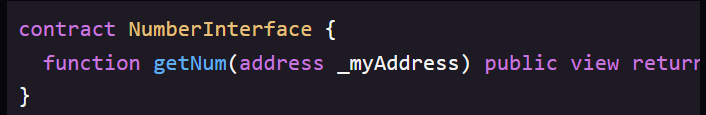


This is a contract where anyone could store their lucky number, and it will be associated with their Ethereum address. Then anyone else could look up that person’s lucky number using their address.

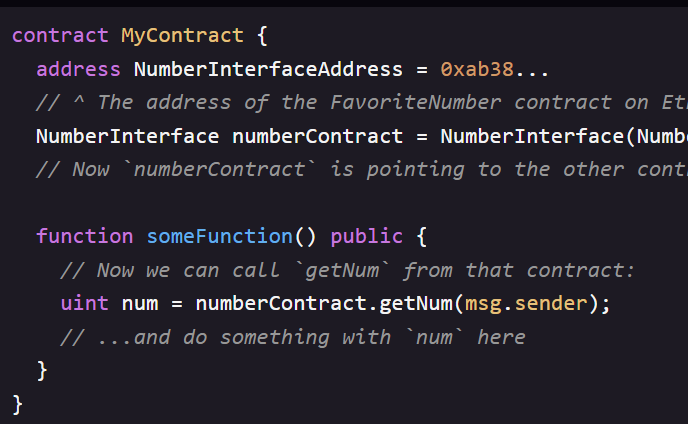


Declaring the interface for accessing the function getNum. We can see that the function ends with a semi colon instead of any brackets and this is how the compiler recognizes that this is an interface.

* In solidity, we can declare multiple values from a function.

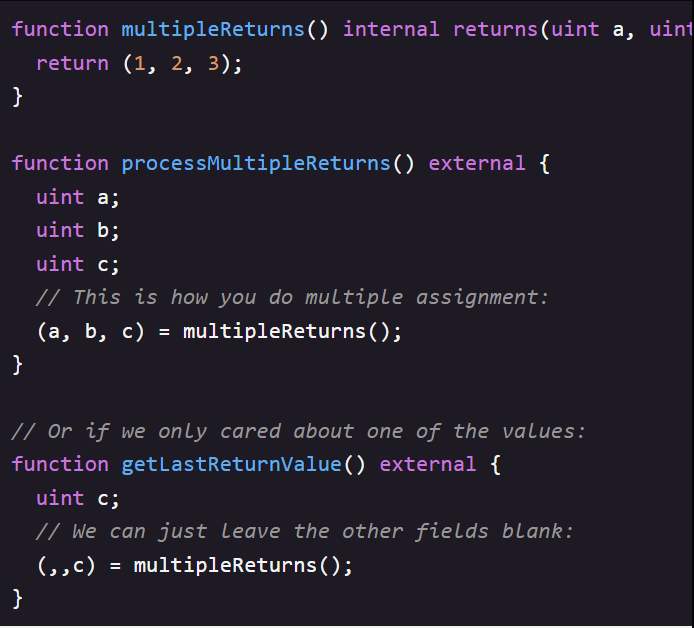


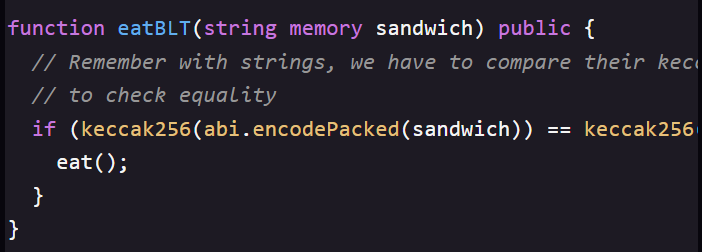
* We can use this contract in our interface as follows:-



In this way, our contract can interact with any other contract on the Ethereum blockchain, as long as they expose those functions as public or external.

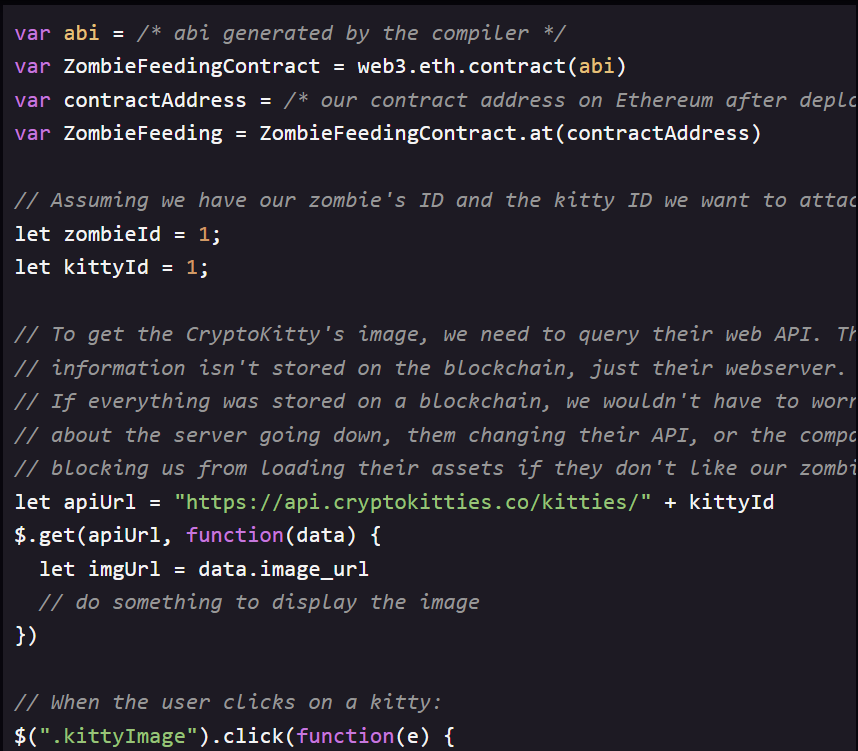
Handling multiple return values:

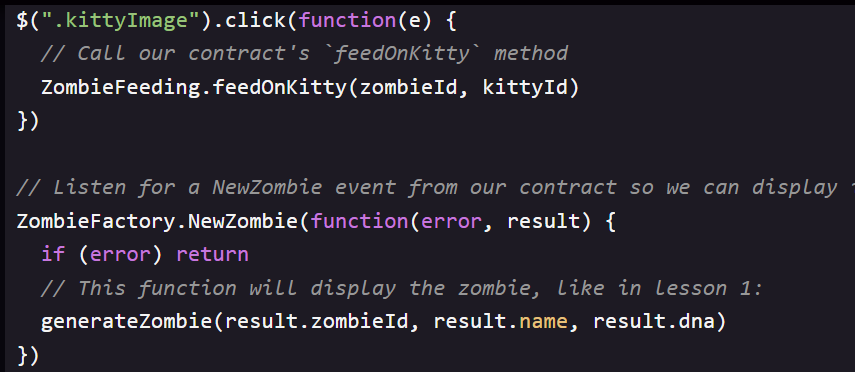




If statements in Solidity look just like JavaScript.

We can interact with our deployed contract using JavaScript and web3.js by:





* After we deploy a contract to Ethereum, it is immutable, which means that it can never be modified or updated again.
* The initial code we 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 is a flaw in our contract code, there is no way for us to patch it later. We should have to tell our users to start using a different contract address that has the fix.
* This is also a feature of smart contracts. The code is law. If we read the code of a smart contract and verify it, we can be sure that every time we call a function it is going to do exactly what the code says it will do. No one can later change the function and give us some unexpected results.
* We don’t 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, we make contracts Ownable – meaning they have an owner ( us ) who has special privileges.

This is the Ownable contract taken from the OpenZeppelin Solidity library. OpenZeppelin is a library of secure and community – vetted smart contracts that we can in our own DApps.

/\*\*

 \* @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;

  }

}

* We know what are constructors, 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.
* The Ownable contract does the following things: when a contract is created, its constructor sets the owner to msg . sender (the person who deployed it).
* It adds an onlyOwner modifier, which can restrict access to certain functions to only the owner.
* It allows us to transfer the contract to a new owner.
* The 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.

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));

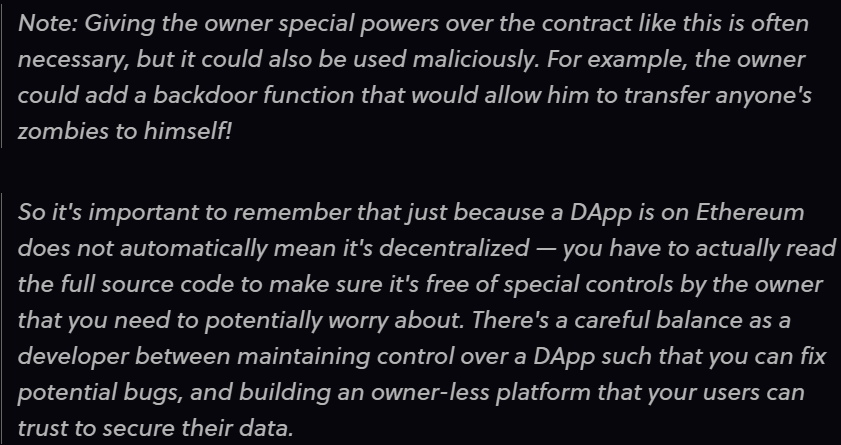
    emit OwnershipTransferred(\_owner, newOwner);

    \_owner = newOwner;

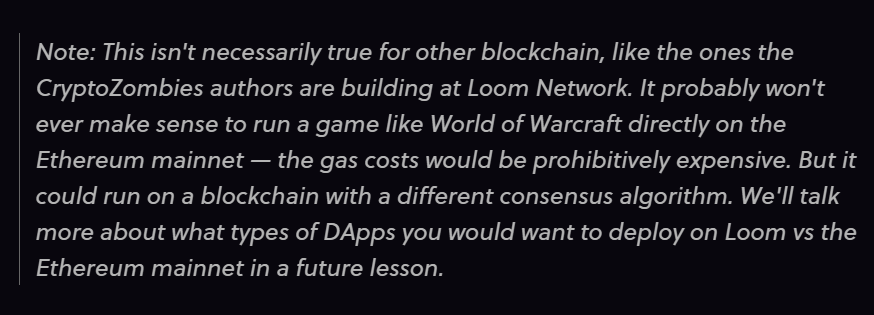
  }

}

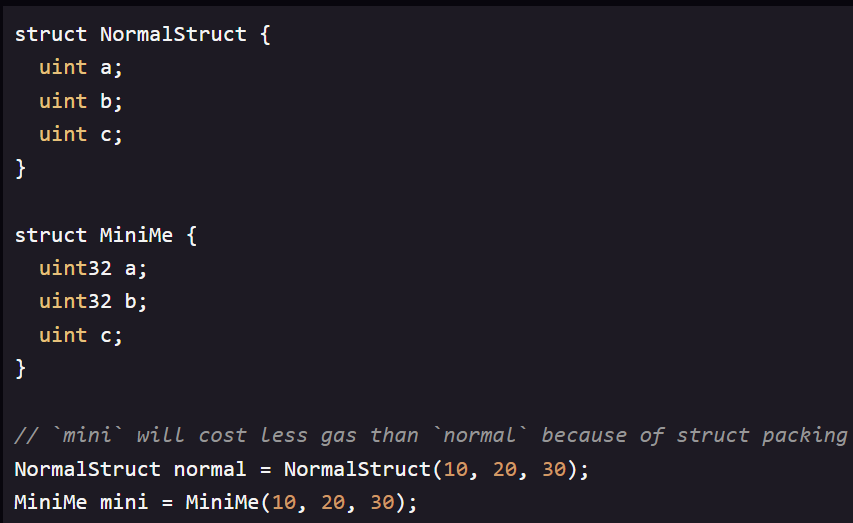
* Notice the onlyOwner modifier on the renounceOwnership function. When we call the renounceOwnership, the code inside onlyOwner executes first. When it hits \_; statement in the onlyOwner, it goes back and executes the code inside the renounceOwnership.
* While there are other ways to use modifiers, one of the most common use – cases is to add a quick require check before a function executes.
* In case of onlyOwner adding require makes it so only the owner of the contract can call that function.



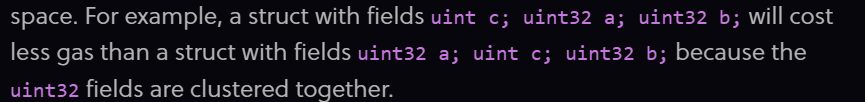
* In solidity, our users have to pay every time they execute a function on our DApp using a currency called gas. Users buy gas with Ether ( the currency on Ethereum ), so your users have to spend ETH in order to execute functions on our DApp.
* How much gas is required depends upon the complexity of the logic of that function. Each individual operation has a gas cost based roughly on how much computing resources will be required to perform that operation.
* The total gas cost of our function is the sum of the gas costs of all its individual operations.
* Running a function costs money to the users, so code optimization is much more important in Ethereum than in other programming languages.
* Ethereum is like a big, slow, but extremely secure computer. When we execute a function, every single node on the network needs to run the same function to verify its output – thousands of nodes verifying every function execution is what makes Ethereum decentralized, and its data is immutable and censorship resistant.
* The creators of Ethereum wanted to make sure that someone could not clog up the network with an infinite loop, or hog all the network resources with really intensive computations. That’s why they made it so transactions are not free, and users have to pay for computation time as well as storage.

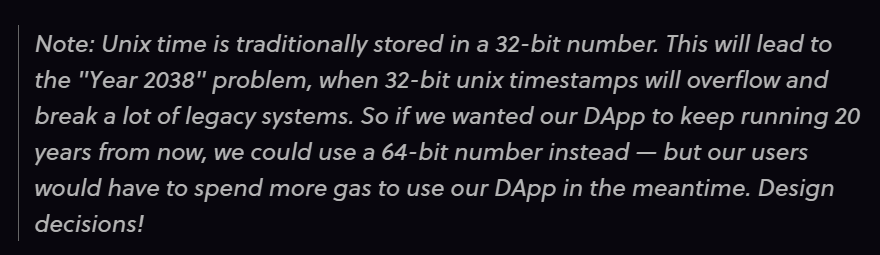


* We had learnt that there are other types of units: uint8, uint16, uint32 etc.
* Normally there is no benefit to use these sub – types because Solidity reserves 256 bits of storage regardless of the uint size. Using uint8 instead of uint ( uint256 ) won’t save us any gas.
* But there is exception that is inside structs:

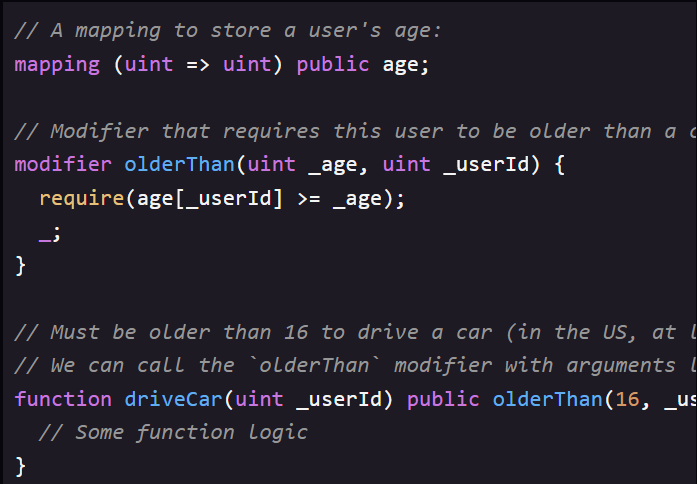


* If we have multiple uints inside a struct, using a smaller – sized uint when possible, will allow Solidity to pack these variables together to take up less storage.
* In the code above ‘mini ‘will cost less gas than ‘normal ‘because of struct packing. For this reason, inside a struct we will want to use the smallest integer sub – types we can get away with.
* Also, we would want cluster the identical data types together that is put them next to each other in the struct so that Solidity can minimize the required storage space.

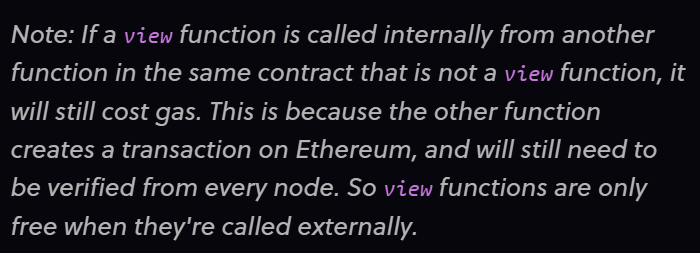




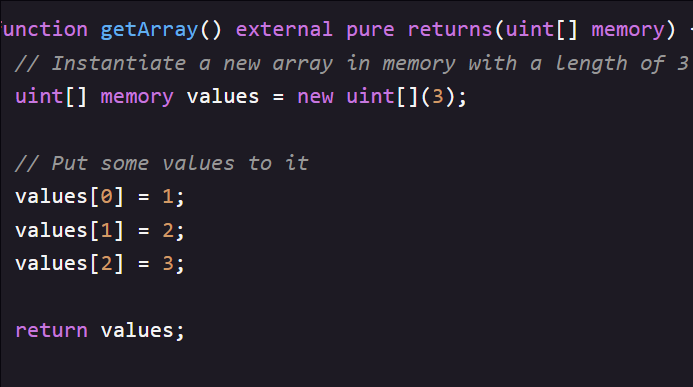
* Solidity also contains the time units - seconds, minutes, hours, days, weeks and years. These will convert to a uint of the number of seconds in that length of time. So, 1 minutes is 60, 1 hours is 3600 and 1 day is 86400.
* An important practice is to examine all our public and external functions, and try to think of ways users might abuse them. Unless these functions have a modifier like onlyOwner, any user can call them and pass them any data they want to.

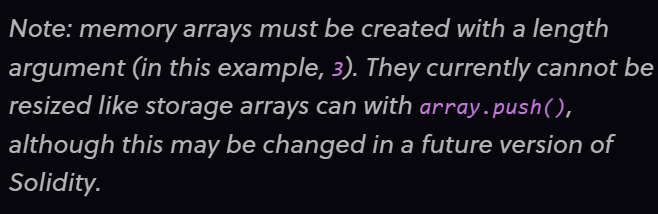


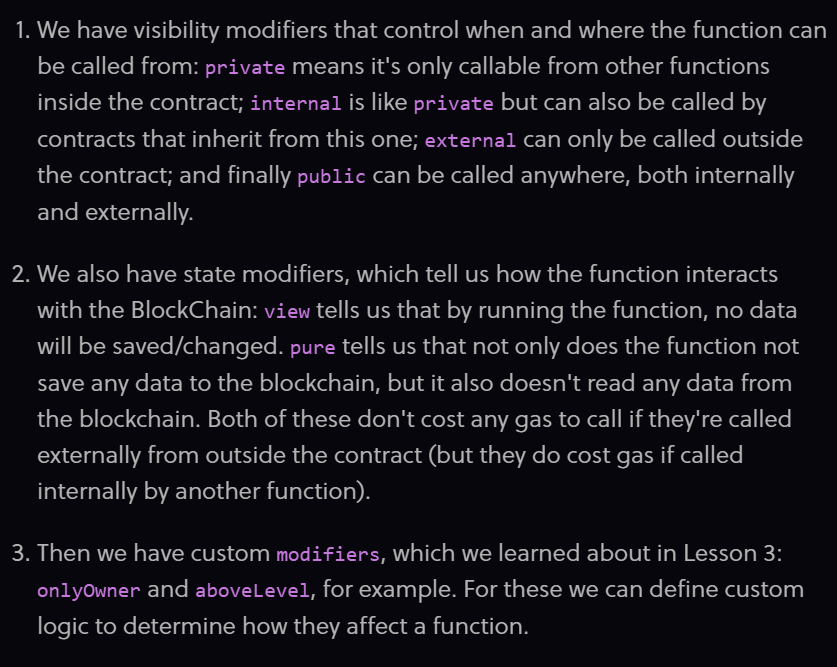
* In the above code we can see that the function modifiers can also take arguments.
* We can clearly see here that olderThan modifier takes arguments just like a function does. The driveCar function passes its arguments to the modifier.
* View functions don’t cost any gas when they are called externally by 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 our local Ethereum node to run the function, and it does not actually have to create a transaction on the blockchain ( which would need to be run on every single node, and cost gas ).
* We can optimize our DApp’s gas usage for our users by using read – only external view functions wherever possible.



* One of the more expensive operations in Solidity is using storage – particularly writes.
* This is because every time we write or change a piece of data, it is written permanently on 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.
* We need to keep costs down and want to avoid ,writing data to storage except when absolutely necessary.
* In other programming languages, looping over large data sets is expensive. But in Solidity, this is way cheaper than using storage if it is an external view function, since view functions don’t cost our users any gas.
* We 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 is a view function called externally.



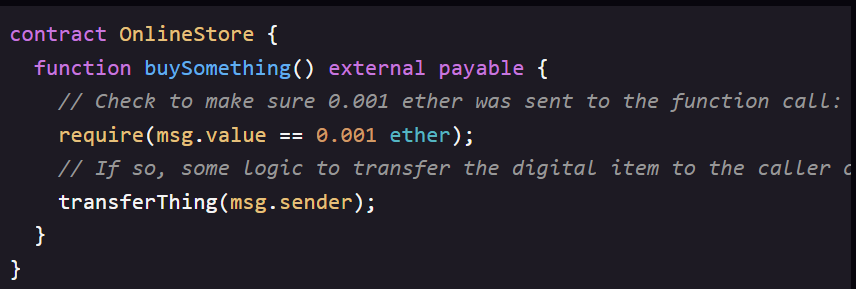




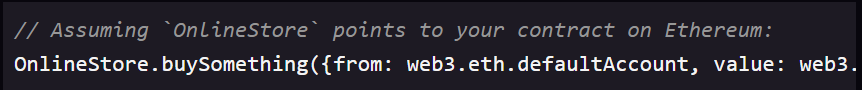
* These modifiers can all be stacked together on a function definition as follows:

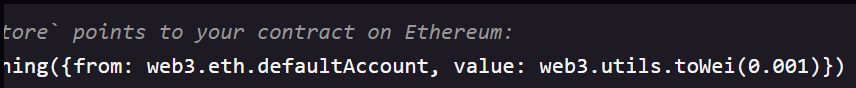


* We have one more modifier which is payable. Payable functions are part of what makes Solidity and Ethereum so cool – they are a special type of function that can receive ether.
* When we call an API function on a normal web server, we cannot send US dollars along with our function call – nor can we send bitcoin. But in Ethereum, because both the money ( Ether ), the data ( transaction payload ), and the contract code itself all live on Ethereum, it is possible for us to call a function and pay money to the contract at the same time.

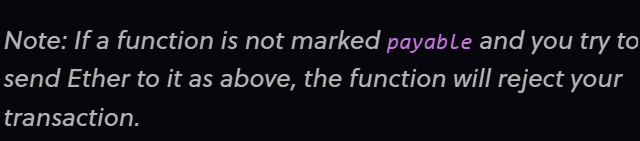
 The comment is caller of the function.

* Here msg.value is a way to see how much ether was sent to the contract, and Ether is a built in unit.
* If someone calls this function from web3.js, it will be as follows:

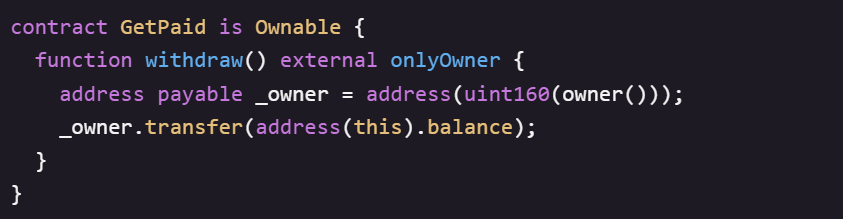




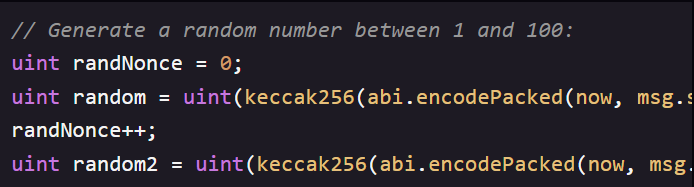
* Notice the value field, where the JavaScript function call specifies how much ether to send (0.001). If we think of the transaction like an envelope, and the parameters we sent to the function call are the contents of the letter we put inside, then adding a value is like putting cash inside the envelope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

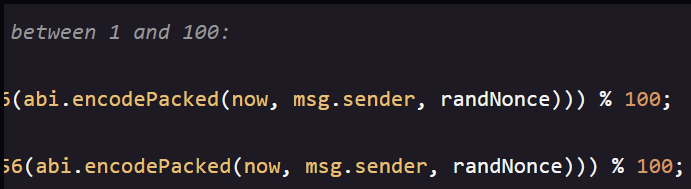


* After we send ether to a contract, it gets stored in the contract’s Ethereum account, and it will be trapped there – unless we add a function to withdraw the Ether from the contract.
* We can write a function to withdraw Ether from the contract as follows.

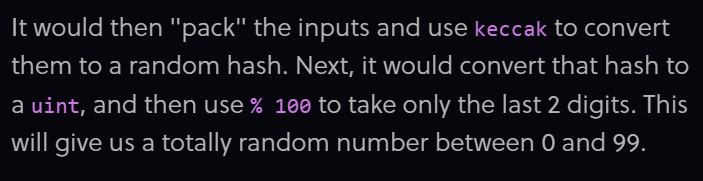


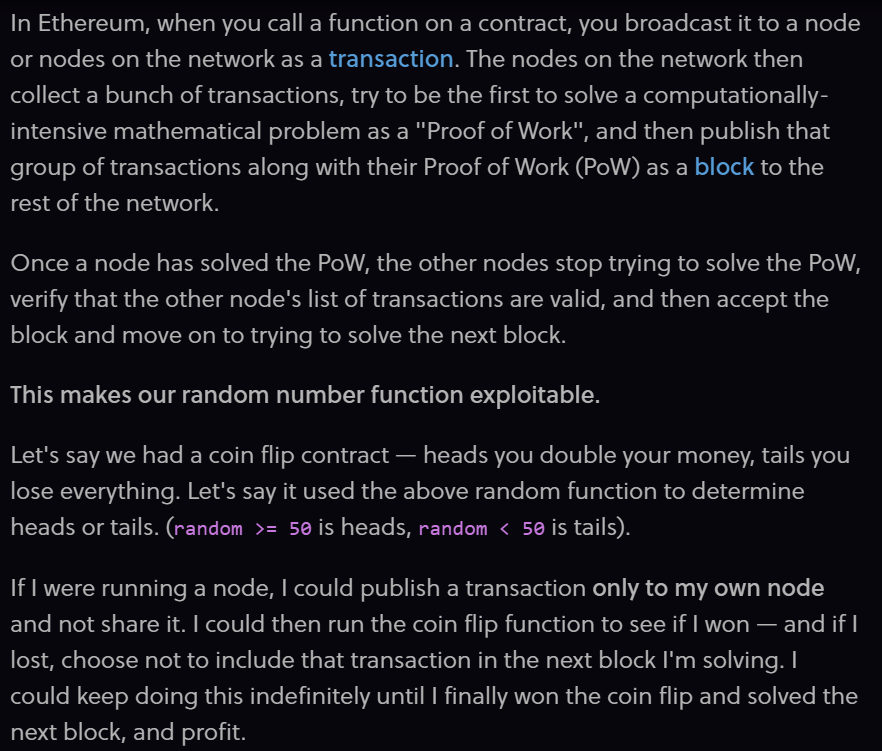
* We cannot transfer Ether to an address unless that address is of type address payable. But in the code snippet above, \_owner variable is of type uint160 meaning that we must explicitly cast it to address payable.
* Once we cast the address from uint160 to address payable, we can transfer Ether to that address using 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 be equal to 100 Ether.
* We can use transfer to send funds to any Ethereum address. For example, we could have a function that transfer Ether back to the msg.sender if they overpaid for an item.
* Or in a contract with a buyer and a seller, we can 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 ) .

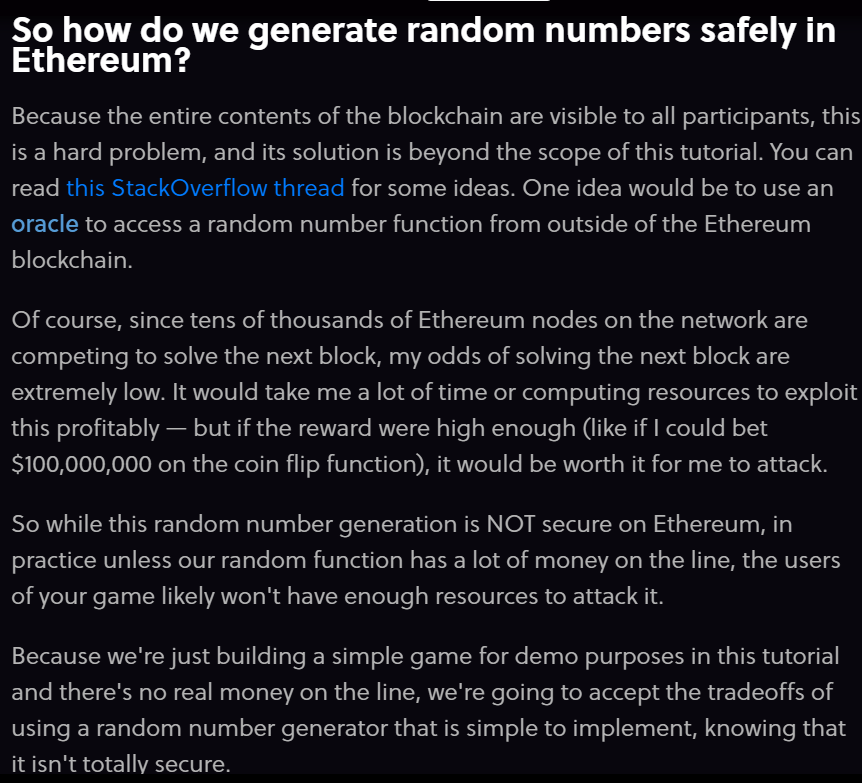




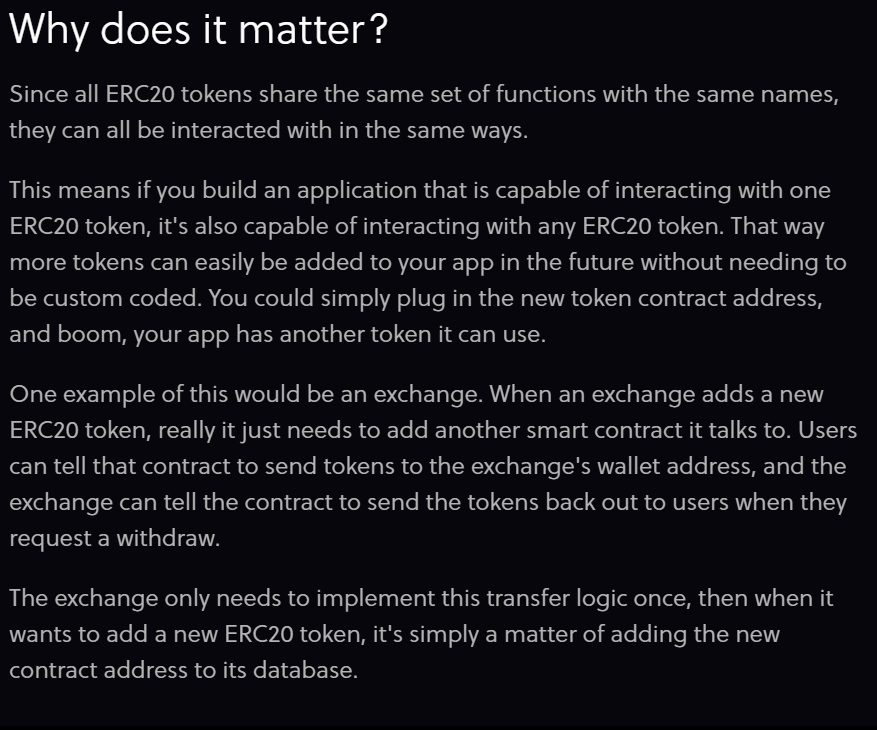
* This would take the timestamp of now, the msg.sender, and an incrementing nonce ( a number that is only ever used once, so we don’t run the same hash function with the same input parameters twice ) .



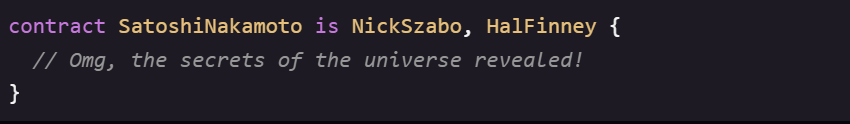




* A secure way to pull data in from outside of Ethereum is using oracles.
* A token on Ethereum is basically just a smart contract that follows some common rules – namely it implements a standard set of functions that all other token contracts share, such as transferFrom(address \_from, address \_to, uint256 \_tokenId) 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.



* When implementing a token contract, the first thing we do is copy the interface to its own Solidity file and import it, . Then we have our contract inherit from it, and we override each method with a function definition.
* Solidity allows a contract to inherit from multiple contracts as follows:

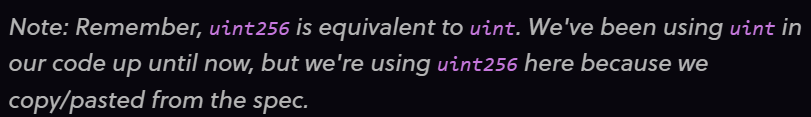




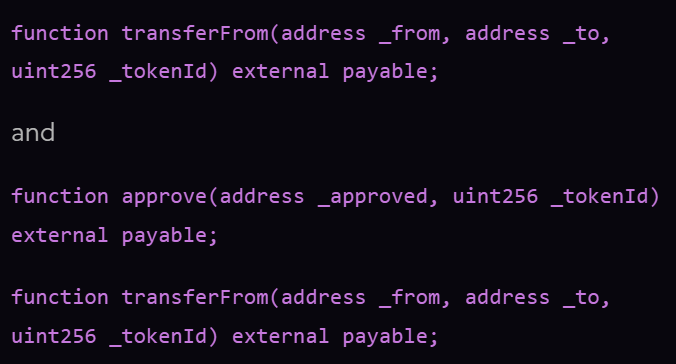
* The above function takes and address, and returns how many tokens that address owns.

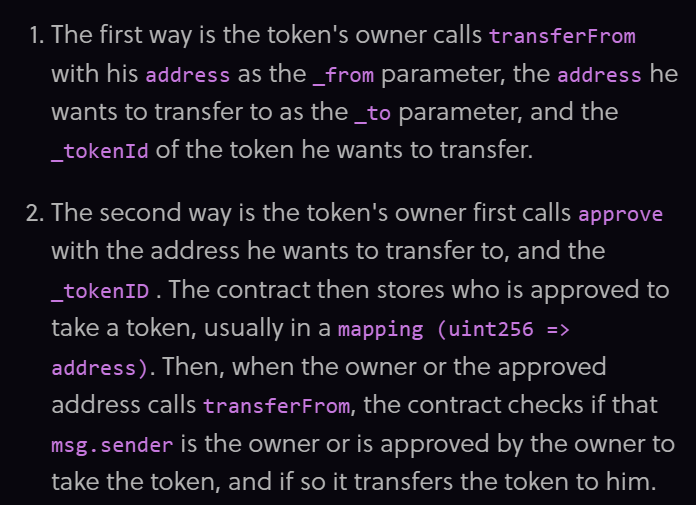


* The function above takes a token ID ( in our case, a ID ), and returns the address of the person who owns it.
* This is very straightforward for us to implement, since we already have a mapping in our DApp that stores this information. We can implement this function in one line, just a return statement.

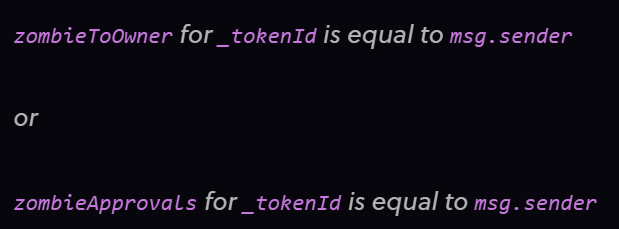


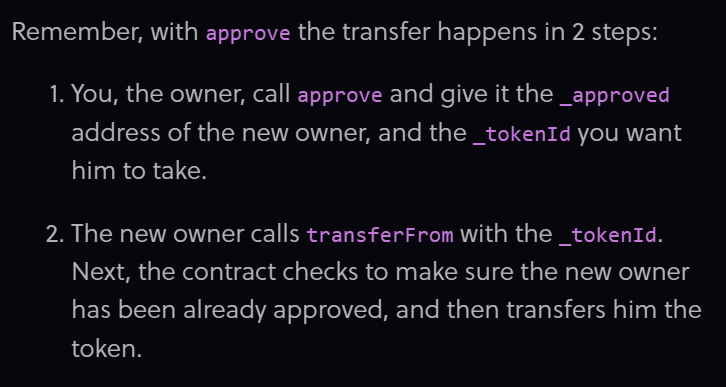
* Here we have a function and a modifier with the same name but we cannot change the function name because we are following ERC721 token standard, which means other contracts will expect our contract to have functions with these exact names defined.
* The ERC721 spec has 2 different ways to transfer tokens:





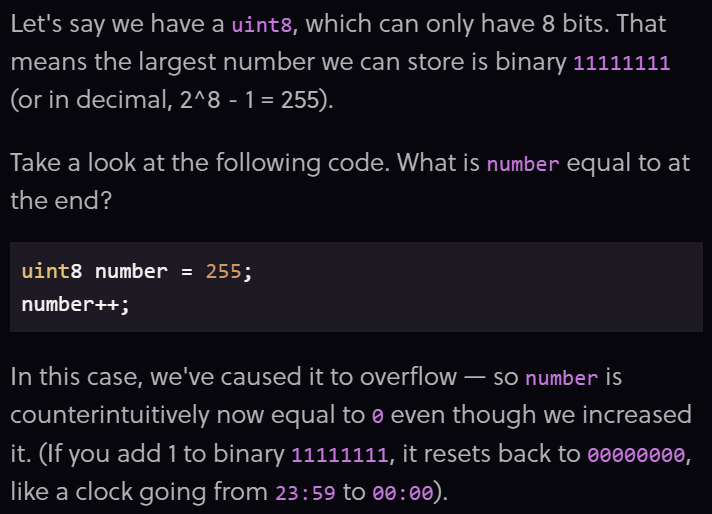
* For checking that only the owner or the approved address of a token.



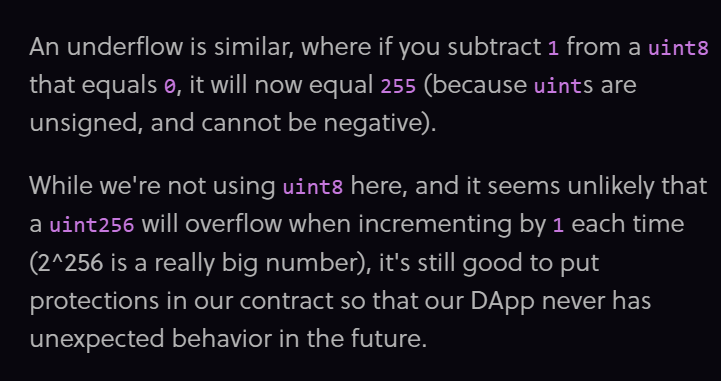


* If users accidently transfer their tokens to address 0 which is called “burning” a token – basically it is sent to an address that no one has the private key of, essentially making it unrecoverable.

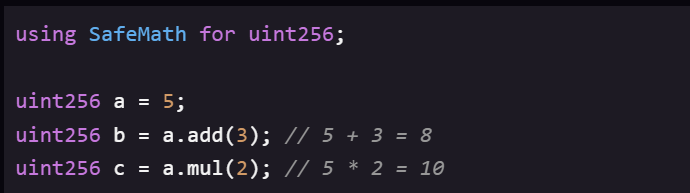
What is overflow?

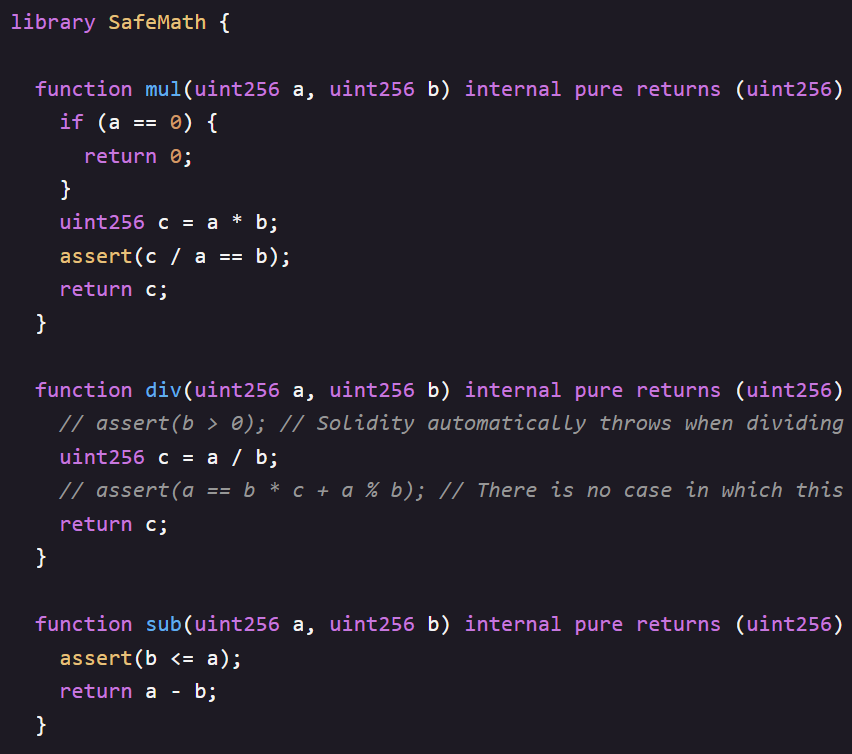


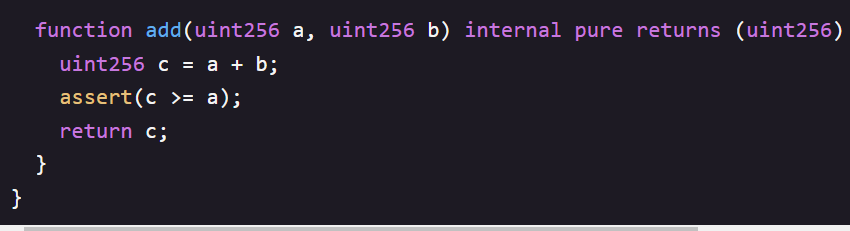
What is underflow ?



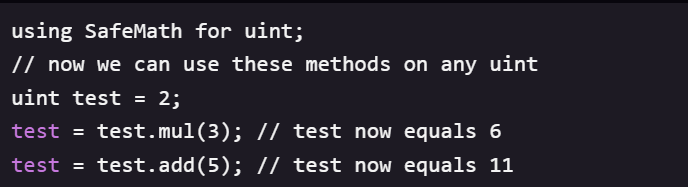
* To prevent these overflows and underflows OpenZeppelin has created a library called SafeMath that prevents these issues by default.
* A library is a special type of contract in Solidity. One of the things it is useful for is to attach functions to native data types.



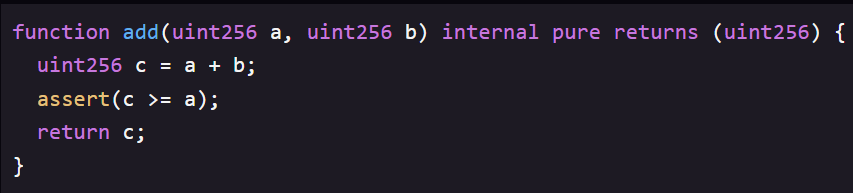




* First we have library keyword – libraries are similar to contracts but with a few differences. For our purposes, libraries allow us to use the using keyword, which automatically tacks on all of the library’s methods to another data type.

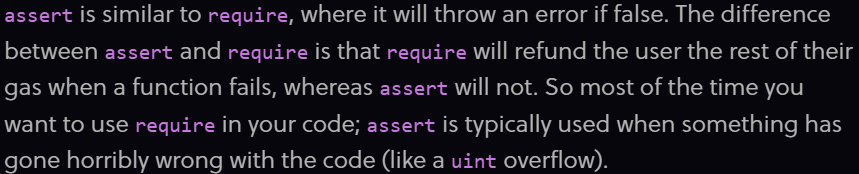


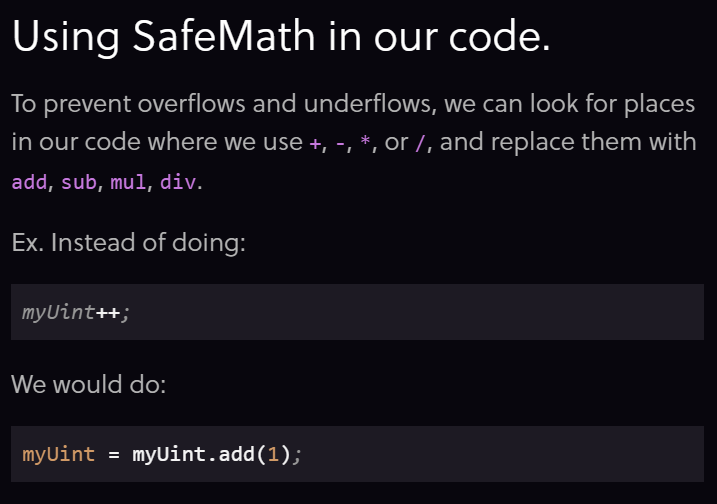
* The mul and add functions each require 2 arguments, but when we declare using SafeMath for uint, the uint we call the function on ( test ) is automatically passed in as the first argument.

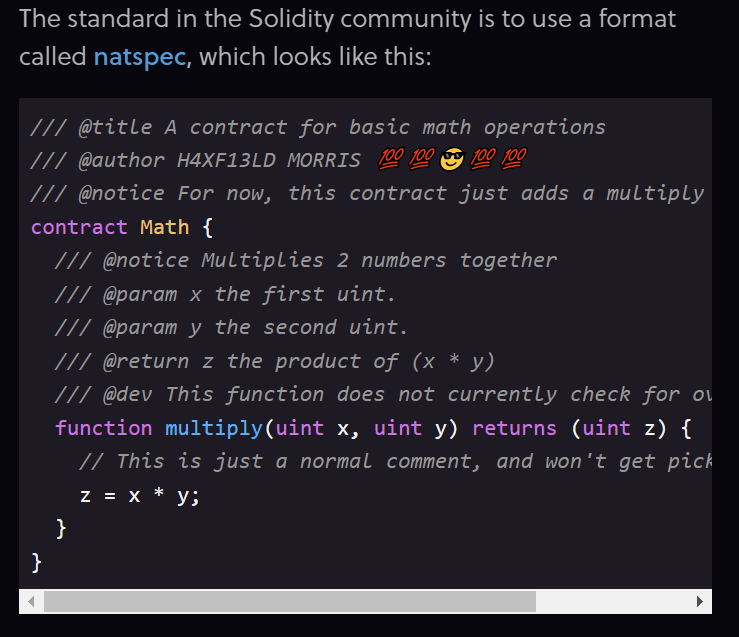


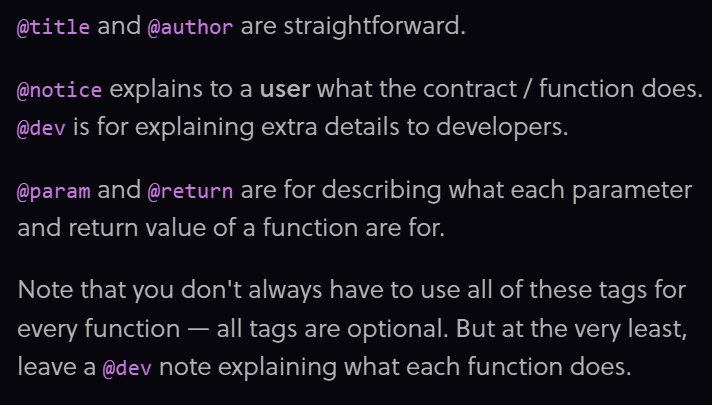
Basically add just adds 2 uints like +, but it also contains an assert statement to make sure the sum is greater than a. This protects us from overflows. The assert statement is also similar to require, where it will throw an error if false.

* Note the difference between the assert and require









Thanks Ayush for looping me in.

Hello Aravind,

Hope you are doing well!

We are grateful to Elastic for its guidance and support in our past editions as the Gold sponsor for hackCBS 3.0 and Associate sponsor for hackCBS 4.0. The response which we received, the engagement we were able to create and the hacker experience that we had, all this would not have been possible without your support. This year we are back on ground with hackCBS 5.0 being an in-person hackathon and we hope to make it even bigger than before. Hence, we look forward to a prosperous collaboration together.

This time the hackers are out of their homes and ever ready to build a hack so, we aim to target a much bigger audience with hackCBS 5.0 .

I’d love to have the opportunity to tell you more about the event and see how we can take this forward. What does your calendar look like for a brief chat or meeting this week or probably next week?