Intro & State Variables

A video discussing all of this material is available here:
https://youtu.be/4XQJHBJSzEk

variables declared outside a function are state variables, they are stored on the blockchain when they are **public**

variables declared inside a function are only available during a call of that function, and such variables are not stored on the blockchain

Smart contracts can be written and deployed to a test environment here:
https://remix.ethereum.org The site cookies save your work

STEP 1: Write a smart contract in this file explorer

STEP 2: Compile the smart contract with a compiler version compatable with this

STEP 3: Deploy the smart contract here to see how it can be accessed on the blockchain with these options here

NOTE: If you hold down both Left-Alt and left-Shift at the same time, you can make the terminal font larger or smaller with the mouse wheel

state variable of type string

Taking the **public** statement out of the state variable will cause you to loose the **text** button on the blockchain

Variables in Solidity are stored in one of these three locations
https://docs.soliditylang.org/en/v0.5.1/types.html

- **storage** this is where data can be permanently stored, these are state variables
- **memory** variables in memory are only available when the function is executing
- **calldata** special data location that contains the function arguments, only available for external function call parameters

Types of Functions in Solidity

- ones that create transactions
- ones that do not create transactions

functions that create transactions write data on the blockchain by changing the value of a state variable, which in turn either changes the state of the smart contract, or sends Ether (ETH) to another account, which in turn changes the balance of accounts recorded on the blockchain
Functions that do not create any transaction on the blockchain are free to call (require no gas) and they do not change the state of the blockchain

This function has a single input of type **string**, for certain data types, you have to specifically declare the data location (there it must be **memory** because we (the user) are assigning the **text** value and we are outside the smart contract
https://docs.soliditylang.org/en/v0.5.1/types.html#data-location

We need to declare the data location for our string type variable **text**; our **text** variable is stored in a contract storage, but need the actual value stored in the **text** variable, not the reference to the variable, so we will declare it as **memory**

this function will update the state variable

The convention in Solidity is to prefix a function input variable with an underscore **_** so as to avoid using the same names as state variables

We need to tell solidity that this function does not modify any state variables; we do this by using the keyword **view** or **pure**

view ensures that your function does not change the state of the blockchain, **pure** declares that your function neither changes the state of the blockchain nor does it read any state variables

If **+** function returns a value, it need a **returns** keyword in the function definition

view

returns

pure

Two ways to get state variables
1.) write your own function
2.) let Solidity compiler write it

When you want to return multiple values from a single function call you will use this option

Otherwise use this option

This example shows a state variable (of type **string**) called **text** as well as a get function that returns the value of the same **text** variable; You would actually choose one of these two options but you do not need both!

Use this one or this one in the smart contract, but don't use both of them

Ether and Wei

A video discussing all of this material is available here:
https://youtu.be/bPQzjsyNw

The currency used within Ethereum is Ether, it can be used to:
- pay block reward
- pay transaction fee
- transfer between accounts

1 Ether = 10^18 Wei

The smallest unit of Ether is Wei
1 Ether is equal to 10^{18} Wei

= x wei

uint is an unsigned integer which means it cannot have a sign in front of it like "+" or "-" this means that the integer cannot be negative

giga-wei

units can only be added after literal numbers; for example, 1 **ether** is valid whereas **x wei**, is not

Gwei stands for Giga-Wei and is equal to one billion wei; the most common situation where you would see the word Gwei is when you submit a transaction to the blockchain because for the transaction, Gwei is used to set the Gas price

If you hand over to etherscan.io and look at any of the Latest Transactions, the Gas Price is listed in Gwei

Etherscan

Gas and Gas Price

A video discussing all of this material is available here:
https://youtu.be/o7SHuUGcAM

Transaction (Tx) fee = gas used * gas price (Gwei)

Each thing you do costs gas; the amount of gas required for each computation depends on what amount miners accept

Transaction (Tx)
Gas limit: 3,000
Gas Price: 2 gwei
Ether:
3,000 gas x 2 gwei / gas)
= 6,000 gwei in weight

In this transaction, we are setting the gas limit to 3,000 and the gas price to 2 gwei, and then, what this transaction is processed it will cost 6,000 gwei (3,000 * 2)

In abstract terms, let's assume 2 gwei is lower than the current average and we're specifying 2 gwei because we're not in a hurry to have our transaction added to the blockchain

Assuming you had 7,000 gwei in your Ether wallet

Your Account
Previous balance: 7,000 gwei
Transaction 'cost': 6,000 gwei
Current balance: 4,000 gwei

Assuming the transaction computations were such that they cost 1000 gas to process (per this) - then the cost would be 1000 gas * 2 gwei which would equal 2000 gwei, and that would mean that your account would be refunded 9,000 gwei since you effectively overpaid for this transaction 'cost' isn't the actual amount of gas that this whole thing actually costs you in the end

happens when you overallocate gas

happens when you underallocate gas

An infinite loop would cost infinity gas to process; let's assume we allocate 6,000 gwei in weight (like we did here) and that each iteration happens to cost 1,800 gwei per supply and demand; here's what would happen:

after first iteration (- 1,800 gwei, 4,200 gas left)

after second iteration (- 1,800 gas, 2,400 gas left)

after third iteration (- 1,800 gas, 600 gas left)

after forth iteration (- 1,800 gas, 0 gas left)

At every step of function execution gas is deducted until either the function finished execution or all of the gas is used up at which point the execution is avoided

not enough gas

On the fourth iteration it uses up all remaining 600 gas midway through the iteration and then the function is forcefully stopped; **any changes that were made to a state variable will be undone, but you still have to pay for the gas spent!**

computations

high gas limit = many computations

The higher you set the gas limit, the more computation your transaction can process

The lower you set the gas limit the less computation your transaction can process

fewer computations — more computations

low gas limit = few computations

tradeoffs

If you set the gas price low you'll have to pay less for your transaction, but you will have to wait longer for your transaction to be included in a block

high gas price = short waiting time

The higher gas price you set, the more Ether you will have to spend, but your transaction will be processed faster

After your transaction is sent, and included in a block, your account will be refunded for any unspent gas

shorter Tx waiting time — longer Tx waiting time

time

low gas price = long waiting time

There are two upper-bounds to limit the gas you can spend:
- Gas Limit - set by you
- Block Gas Limit - set by the network

When you send a transaction to the real Ethereum network you set the gas price, but gas price in remix is fixed at 1 wei, and we can verify that by checking the output of this function

we can see our balance here

we can click here to check the transaction cost

we can change the gas limit here

Invalid Functions

A video discussing all of this material is available here:
https://youtu.be/1TcmPoD_AnQ

Invalid Inputs & Outputs for Public Functions
- f() map
- f() [] multi-dimensional arrays (unfixed size)

In Solidity, there are certain data types that cannot be input variables in public functions (e.g "f()") and there are also data types that are not recommended (e.g "f()") as well. This is also true for function outputs: "f()" is not allowed and "f()" is not recommended

inputs

If you get rid of this statement and replace it with this statement, then this function will compile; otherwise it will not compile; the other two functions will compile just fine without using this statement. It is not recommended that you use any array as an input variable because different array sizes require different amounts of gas to process, and so sometimes the function will process just fine and other times it will run out of gas when using an array as an input variable

One way to make an array more reliable (as an input variable) and avoid the problem with gas (discussed above) is to put an upper-bound to it; this limit the amount of gas consumed

outputs

If we try using mapping and/or multi-dimensional arrays at an unfixed size as function outputs, we will also get compile errors

this function will not compile

this function will not compile

Using a one dimensional array with an unfixed size is not recommended because your contract might get called by a second contract and so your contract's function output is another contract's function input; one way to solve these issues is to write functions that have a bounded consumption of gas

multiple un-named values

multiple named values

In Solidity, you can return multiple values, and those values can also be named; the options on the left are all valid ones

function that returns multiple functions

We can call **firstFunc** and **secondFunc** instead separately, or we can just call **thisFunc** instead

destructuring assignments

Destructuring assignments can be used to assign variables to the output of a function which returns multiple values

Here we are assigning the outputs of the function **returnMultipleVals()** and the variable types declared here are consistant with the types of values **return** that are being returned by the function being called

If a function returns three parameters, but you don't care about the second one (in this case, the **b**) then you can use destructuring by adding in an **empty-space with a comma** to let Solidity know to skip that value

View and Pure Functions

A video discussing all of this material is available here:
https://youtu.be/kmnxALALBc

- **view functions do not modify the state of the blockchain**
- **pure functions do not modify the state of the blockchain, nor do they read the state either**

According to the Solidity docs here, the following statements are considered modifying the state:
1- Writing to state variables
2- Emitting events - also known as 'logging'
3- Creating other contracts
4- Using selfdestruct.
5- Sending Ether via calls
6- Calling any function not marked view or pure
7- Using low-level calls
8- Using inline assembly that contains certain opcodes

NOTE: this is a function to call when you want to delete your contract from the blockchain

pure makes a stronger statement than **view**

According to the Solidity docs here, the following are considered reading from the state:
1- Reading from state variables
2- Accessing address(this).balance or <address>.balance
3- Accessing any of the members of block, tx, msg (with the exception of msg.sig and msg.data)
4- Calling any function not marked view or pure
5- Using inline assembly that contains certain opcodes

Inheritance (Override-1)

A video discussing all of this material is available here:
https://youtu.be/GzHHRB-1o8

contract B is inheriting from contract A, and they both have a function with the same name; normally, the function in contract B would override the one in contract A if they have the same name, but here, contract B's function accepts an input variable whereas contract A's function not; therefore, contract B will have two functions of the same name available to it when deployed

this is a valid **view** function this function reads from the **state**, it could not be a **pure** function

A view function cannot call another function that is neither **view** or **pure**

This is an invalid **pure** function because pure functions cannot call a non-pure function, even if that function is a **view** function

this function will not compile

this function will not compile

this is a valid **pure** function since it doesn't read and/or modify any state

Function Modifiers

A video discussing all of this material is available here:
https://youtu.be/thAOMqNcKPM

Function Modifiers are used for:
- restricting write access
- input validation
- reentrancy guard

Function modifiers are reusable code that can be attached to a function; this reusable code can be executed before and or after the function itself is executed

when the contract is deployed, it sets the owner to msg.sender

the "_" is a special character that you can only use in a function modifier; it tells Solidity to execute the rest of the code inside the parent function

If the message sender is not the current owner, the transaction fails

You attach a modifier to a function by declaring it here in the function signature

reentrancy guard

code here, and next it calls itself again here but since locked is set to true, at this point, the require statement fails here and the transaction therefore fails so as to prevent a reentrancy hack

Inheritance (Constructor-1)

A video discussing all of this material is available here:
https://youtu.be/dcKpiOh7TNQ

In Solidity, constructors are always executed in a certain order: You cannot override a state variable that is declared in a parent contract by simply redeclaring it in a child contract

this is one way to call the constructors of multiple parent contracts ··· this is another way to call the parent constructor inside the constructor of the child contract; here we do not put commas between contracts when listing them

this is how you pass variables to parent contracts, the syntax is similar to this, except here, we are passing in a fixed input and here, we are passing in variables

here the constructor is accepting two different inputs to be passed to the parent constructors (X & Y) and here it passes the _name variable to contract X and here it passes text to contract Y

compile and then deploy contract D, and then upon deployment, set the string_name to "foo", and the string_text to "bar"

Calling the state variable name returns the string "foo" and calling the state variable text returns the string "bar"

Inheritance (Constructor-2)

A video discussing all of this material is available here
this section starts at time: 3:26 / 8:09

Below we see in what order parent constructors will be called and executed; whereas on the right (Constructor-1) we detail how to pass input variables in a child constructor to a parent constructor

the order in which the parent constructors are called is determined by the inheritance order here and not by the order in which the parent constructors are called on these lines here

compile and then deploy contract E and then do contract F; both produce the same order of firing in the logs here

click here to see this dropdown

Two ways to call parent contracts
- using the keyword super
- explicitly

Inheritance (Super)

A video discussing all of this material is available here
this section starts at time: 6:56 / 12:37

This example on the left shows how the parent (more basally) functions will be called when using the keyword super; contracts F, E, & C first emit an event then call the parent contract by using the keyword super

super

in this example here, the parent contract is called directly whereas here, super is used to call all parent contracts in the order of inheritance, because the bar() function in contract A is being overridden by the bar() functions in contracts B and C

emit

here, X, Y, & Z are hypothetical

click here to see this dropdown

Constructor

A video discussing all of this material is available here:
This section starts at time: 5:05 / 8:09

a constructor is an optional function that is executed only once when the contract is initially deployed to the blockchain; if it accepts input variables, they are entered upon deployment

this will give you the unix timestamp for when the contract was deployed to the blockchain

constructor() has access to special variables like msg.sender & block.timestamp

Inheritance (Order)

A video discussing all of this material is available here:
https://youtu.be/Q5_Gmm_IZSI
The order of inheritance determines which contract is inherited; according to the docs, parent contracts must be ordered from most basalic to most derived (like reverse in Python)

Meaning that basalic + less inheritance while derived = more inheritance, that gives you this and vice-versa

In the example above, contract A and contract B are both equally basalic, so contract C inherits whichever one is listed last in the order of this order has contracts listed out of order (from most base-like to most derived) then the Solidity compiler might throw an error, otherwise the order of inheritance will go from right to left for functions, but left to right (the opposite) for contractors which inherit from left to right

depth first

Solidity will perform a depth-first search when inheriting, such that in the example on the left, if you call foo() on contract D, it will return A because it will first look in itself (D) then search from right to left (C B) until it finds the function that it is looking for (B)

compile and then deploy contract D

click here to see this dropdown

Inheritance (Events)

A video discussing all of this material is available here
this section starts at time: 8:40 / 12:37

Solidity uses events to log things on the blockchain, and applications can be notified when events have been emitted so that front-end applications don't have to constantly monitor the contract (on the blockchain) to see if such changes have occured

emit

Events are declared with the event keyword and are capitalized by convention; you can pass parameters that you want the event to log in the most basalic contract where the event is found and those parameters will get passed to the more derived contracts downstream

Unlike return statements ( ) to a function called, emitted events with the same name will get called twice, and not be overridden

click here to see this dropdown

Inheritance (Override-2)

A video discussing all of this material is available here:
This section starts at time: 3:05 / 8:09

Here is an example of how not to override a variable in a parent contract

If we deploy contract B and then call myAddress here, we get the address that is redefined here in contract B, and not the original definition found here in contract A - this is what we want to get, but if we were to then call getAddress() from contract B here, we would get the original definition for address found in contract A, instead of the redefined address in contract B this is because you cannot override a state variable that is declared in a parent contract by redeclaring it inside of a child contract

The correct way to override state variables (that are defined in a parent contract) is to redefine such variables inside a child contract's constructor() like this so when you deploy such a child contract the this in both cases this will give you the unix timestamp for when the contract was deployed to the blockchain

Visibility

A video discussing all of this material is available here:
https://youtu.be/Nb2Q16OnCq

These are the 4 key words that determine the visibility of state variables and functions:

- **private** - visible inside the contract
- **internal** - visible inside the contract and child contract(s)
- **public** - visible to everyone
- **external** - visible to other contracts and accounts

A state variable or function declared **private** is only accessable to the contract that defines it

A state variable or function declared **internal** is only accessable to the contract that defines it as well as any child contracts it has

A state variable or function declared **public** is accessable by any contract and any account

A function declared **external** can only be called by other contracts and accounts, but the contract it is defined in; a state variable like this cannot be declared **external**, since it doesn't make sense to have a state variable that stores data that is only accessable to other contracts and accounts

private

internal

public

external

Events

A video discussing all of this material is available here:
https://youtu.be/8duMOmgAuKs

Smart contracts can log that something has happened on the blockchain by firing events; applications can be automatically notified when events happen so they do not have to monitor a contract on a blockchain to see what state changes have occured

Anyone on the blockchain can query a smart contract's logs with input variable(s) if such variables are declared **indexed** - this keyword allows anyone to query a log for specific event(s) that match such criteria

this query would return these

indexed

It is possible to create an event where no parameter is passed

To see the events that were fired via the emit keyword, execute the firstEvents() function and then open the transaction log and scroll down to the keyword logs where you will find the events that were fired

These logs will be stored on the blockchain but smart contracts will not be able to access information on these logs; the purpose of emitting events is to store information on the blockchain that smart contracts do not need access to, because storing such information on logs is cheaper than storing it on smart contracts

Error

A video discussing all of this material is available here:
https://youtu.be/k4mdv97upSM

Sometimes you want your function to fail if something is not right

In Solidity, there are 3 ways to throw an error which would stop the further execution of the function being called:

assert - used to check for conditions that should never be possible; should always evaluate to true
gas up all the gas

require - used to validate inputs, state conditions before execution, and to check outputs of other functions
gas does not use up all the gas

revert - will only accept one argument (the error message to be thrown) whereas require will accept two

assert

require

revert

When a function throws an error, it will undo all of the changes that were made during the current call; Solidity actually throws **bubble up** (like in Python or JavaScript) meaning that if function a calls another function b, and b throws an error, a will also fail and all of the changes that were made during the call of a and b will be undone

overflow

this is the largest number possible for the data type uint in Solidity; when you exceed it, the number "overflows" back to 0 like on a clock. Likewise, a number "underflows" when it reaches the lowest possible number for data type uint, which is 0

underflow

If a number is deposited here and then the max number is added to it, this overflow message will be generated; if the amount of money withdrawn here is greater than the balance here, then this error message will be generated

NOTE: Newer versions of Solidity will have different error messages as they auto-detect discrepancies in Ether sent from an account vs. Ether remaining in that same account

Looping

A video discussing all of this material is available here:
https://youtu.be/hxdQvcsDvU

Loops in Solidity are structured just like they are in JavaScript, but in Solidity you want to make sure that your loops have an upper bound so they do not run for more iterations than you are expecting them to because that would use more gas than you were expecting to be used; this array has 100 elements, and the loop is limited to 100 iterations here, this is how loops in Solidity should be structured as opposed to leaving them open like this where you can have unlimited iterations

Arrays

A video discussing all of this material is available here:
https://youtu.be/9xDAvK7Vouw

arrays are declared with type, visessibility, and name; they can be either dynamically sized like this or fixed sized like this; the actual size (for the latter case) is defined inside the brackets (in this case, size = 10) once the size of the array has been declared, it cannot be changed; the default value for each array element is 0 so this array will have 10 zeros in it; we can also create a dynamic array with default values stored in it like this and the number of elements in it can be changed

In Solidity, there are three built-in functionalities:
- push() is a function that adds an element to the end of an array; pop() is a function that removes the last element in the array; length is a property that stores the length of an array

Solidity does not have an unshift() or or splice() method like JavaScript, so a custom function will be needed to delete an element at a specified index, but this would set the element's value to 0, to keep arrays compact, we can prob the last element in an array and swap it to the index we want deleted, then delete the last element after that

Mapping

A video discussing all of this material is available here:
https://youtu.be/0DJvYMCOzs8

A mapping in Solidity is like a Dictionary in Python or a Map in JavaScript; however, in Solidity, mappings are iterate through a mapping and in order to create a mapping that you could iterate on, you would have to use a combination of mapping and arrays wherein you store the keys in an array; you also cannot get the size of a mapping; the format is: mapping(keytype, valuetype)
- the keytype for a mapping has to be a built-in value type, a string, or bytes; the valuetype can be any type including an array or another mapping; Unlike arrays, mappings will not return errors when you try to access keys that do not exist, instead the default value of 0 will be returned; as with arrays, the delete keyword does not remove an element in a mapping, but instead sets its value to 0; To see all this, compile and deploy this contrak on Remix and then copy the contract address here and paste it here with a uint value here, then hit the get button; After that, hit the this button and you will see this uint value here, but if you copy the address and paste it here, the hit the this button after that, when you hit the this button the value will be 0 instead of 123

Enums

A video discussing all of this material is available here:
https://youtu.be/ysbd7N1SJ0

In Solidity, Enums are a data type used to create a custom-defined type; imagine we have an online bookstore in which the status of an order starts as pending, once it is pending, it is either canceled or shipped, if it is shipped the order can be either accepted or rejected after that; the order cannot be canceled after it is shipped

We can model this situation using enum

This is an enum and the name of this type is Status, we define the possible Status in a public state variable & declare a name for it to be stated inits; the default value of the enum will be the first element defined in the enum; The function on the left ensure the flows shown in the diagram above are maintained, in example, the status cannot be accepted or rejected if it is not currently shipped

If you compile and deploy the contract on Remix, then click the currentStatus, you will get a returned integer because the 0 is the first index of our first (and default) element in the enum; if you call ship and then currentStatus after that, the value returned will be 1; if you try to cancel an order after it has been shipped, the transaction will fail because an order cannot be canceled unless it is still pending per this require statement.

Structs

A video discussing all of this material is available here:
https://youtu.be/5Y8HqJYmFBc

A struct is used to define a custom data type; the Remix uses CapsCards & these are called fields, here we're creating a struct consisting of the (struct) data type and this function will create a new instance of the Todo struct then it will .push() that new instance into this array; This order must match this order, but here the order doesn't matter; here you can set only some field attributes and the rest will initialize with default values

initialization method 1

initialization method 2

the get's the fields out of the array

You don't need to make a function like this because Solidity automatically creates getter functions like this when a struct gets created; If you compile and deploy the contract on Remix, you can create a struct, then get it returned here or here by querying the index (0) where it occurs in this array; updating the text here will change the text being returned to these here while toggling the boolean here will change the return value here

Payable

A video discussing all of this material is available here:
https://youtu.be/yD9EL1QN4QQ

Contracts in Solidity can send and receive Ether by using the keyword **payable**; in order for this deposit() function to be able to receive Ether into this (Wallet) contract, it must be declared payable; the sender is stored in a global variable msg.sender and msg.value will hold the amount of Ether that was sent with this transaction, & the current balance is stored using this which is casted into an address data type, where the this refers to this contract

we only want the owner of this contract to be able to send Ether so we will use this modifier to ensure that is the case; and in order to do that we need to define the owner here as a private state variable; and then when the contract is deployed we assign the owner to the sender of the transaction here; this built-in function is only available to payable addresses like this; if you compile and deploy this contract on Remix, this button will be red because the constructor is payable; which means we can send Ether when we deploy this contract; this button is red because its function is payable; if we specify 100 wei here before hitting deploy, the balance here will be 100; if we then enter 10 here and hit deposit, then hit the getBalance button, we will have 110 wei; if we enter any value of Ether here and then click the withdraw button, the transaction will not process and an error message will occur in the log because this function is not defined as payable; only the owner of this contract can transfer Ether using transfer(); to do this select account #2 here, then copy the #2 address here, then switch back to account #1 (the owner of this contract) here; after that; next, paste in the account #2 address here and the amount you want to transfer here

YouTube Tutorial References:

Learn Solidity (0.5) by Examples
Smart Contract Programmer

github.com/Richard-Burd/solidity-sandbox
last updated @10:43am on 45/August/2021 by Richard Burd rick.a.burd@gmail.com

Solidity Illustrated