**Lesson: Smart Contracts in Solidity**

The world spends billions of dollars annually on blockchain every year! A developer who can write smart contracts in Solidity is a huge asset to financial companies who are some of the top spenders on blockchain solutions. A smart contract is a contract written on blockchain.

Contracts are written by people in the real world every day, like, for example when you rent or buy a home, choose a cell phone service, or set up your internet service. Now, let’s say I owe my friend (we’ll name her Jenny) and her roommate (Tim) each ten dollars. I show up at their house and give twenty dollars to Tim because Jenny is not home when I come by to pay them. Now I am trusting that Tim won’t steal Jenny’s ten dollars because Jenny knows where Tim lives and that would be quite awkward. I may feel less comfortable giving Tim all the money in cash, if I owe both Tim and Jenny 10,000 dollars. What if there was some way, I could ensure that Tim would not get the $10,000 I owed him unless he paid Jenny? That’s where a contract could come in handy because it would ensure that Tim only got the money he is entitled to if Jenny got her money from him. A smart contract can ensure no is paid until all required conditions are met.

The code in a smart contract won’t execute unless all conditions of the contract are met, and no one can change them once written - not even the contract’s original author. A developer would need to write a new contract if they want to specify different conditions. The code you will learn to write in this course dictate financial transactions. It will be helpful for defining the rules of how businesses and people who pay each other and distribute large amounts of money work. It gives businesses the ability to transfer money without banks and to draw up contracts without a lawyer. Your position as the developer is vitally important because the code written automates financial and democratizes financial processes. As you can imagine, this is great for businesses. The automatic execution of these contracts simplifies business processes and eliminates distrust between parties. It also saves time and is cost effective, fast, and eliminates potential for errors.

In this lesson, you are a developer who needs to write a smart contact that purchases stocks for a company from a target source. In other words, the buyer would like to receive the item and the seller would like to get payment. They both need to fulfill their side of the agreement and we must ensure both sides of the contract’s conditions are met. Your company does not want to purchase any stock that the seller does not have and would like prompt details involving price and quantity. Let’s learn some blockchain basics, how to code in Solidity, and write your first contact!

**What are Smart Contracts in Solidity?**

In order to understand smart contracts, you first need to know what the Ethereum blockchain is and how it works. It does not have a centralized source like a cloud platform such as Google Cloud or Amazon Web Services would. Developers run the Ethereum Blockchain on their machines and are paid in cryptocurrencies called **ether**. A cryptocurrency is essentially a digital asset. The ledger that tracks these assets is the **Ethereum blockchain** (there are other types of blockchain that also serve as ledgers for cryptocurrencies like Bitcoin). The Ethereum blockchain is the database that tracks what is owed, stored, and runs smart contracts.

Developers who run the Ethereum blockchain on their machine for ether are called **miners.** **Gas** is the cost necessary to perform transactions on the network based on the amount of memory your contract uses. Miners who use their computer to run the code on the Ethereum Blockchain create a network called the Ethereum Virtual Machine and set these costs of ether based on supply and demand. It is in your best interest to ensure that your program uses the least amount of memory and data to save money, or gas.

Your role as a blockchain developer is to put financial rules in place that you write with your code. Your code serves as business logic on the Ethereum Blockchain as a smart contract. Your code will execute on the blockchain and automatically execute when the predetermined conditions of the contract are met.

**Let’s talk about Solidity**

Now, let’s talk about the programming language that is used to write smart contracts - Solidity! It is a contract orientated language that is a well-known tool used to meet money related needs auctions, crowdfunding, or wallets with a multi-signature feature. Solidity is an object-orientated programming language. It has similar syntax to JavaScript, Python, or C++. In summary, Solidity is the programming language which runs on the Ethereum virtual machine (developers’ computers) on the Ethereum blockchain.

Before we write code in Solidity let’s talk about how it works. It’s important to understand the fundamentals of any programming language so that you can reference your knowledge as we walk through coding your first contract.

· Syntax

· Variables Types

· Functions

· Events

**Syntax:**

**Pragma**: At the top of a Solidity file, you will see the word “pragma” - this defines the version of the compiler that the code is compatible with.

**Style Tips:** The style guide for Solidity is based on Python’s Pep8 guide. Below I have added some tips for writing your program. These are tips that will help your code be read by other developers, technical mangers, and other people who may be interested in the contract you are writing.

It is good practice to use spaces for indentation. Four spaces represent one indentation level. In addition, two blank lines should surround all top-level declarations and one line for function declarations, unless of course you have a lot of one-line commands. An indent will work for wrapped lines, keep each argument on a separate line and the opening parenthesis shouldn’t go together. The terminating element should be on the final line. Avoid adding extra spaces inside braces or parentheses, before a semicolon or comma, or on a fallback function. The image below contains some code that demonstrates some of these rules.

Text

Description automatically generated

There are some instances where we can use an extra single space in Solidity to surround operators, separate control from the conditional block structures for example: (if, while, and for) and to separate the conditional block from the opening brace.

Comments: To add comments to your code use // for a single line and for a multiple lines /\* \*/

Now that you know how to structure your code like a blockchain developer let’s talk about how to write it!

**Variables:**

Variables are used to store information to be referenced and manipulated in a computer program. There are few things to note about Solidity variables. First, we will talk about **variable types.**  In Solidity are two types: **value**, which holds the actual value, and **reference,** which references the point in memory, storage, or calldata, where data is stored. **Value types** include booleans, integers, fixed point numbers, addresses, contract types, fixed-size byte arrays, rational and integer literals, and enums. There are **local variables** and **state variables**.

There are two types of variable scope: **local** and **state**. **Local variables** are written inside functions - they do not save in blockchain and disappear when the function they are written in concludes.

Text

Description automatically generated

**Functions:**

A function is a group of reusable code that can be called anywhere in your program. A developer uses functions to divide a big program into a number of manageable parts.

**Example:**

function function-name(parameter-list) visability returns() {

//statements

}

**Declaration**: We are letting the program know that this code is a function by using the function keyword. Then, we are naming it something unique so we can reference it later in our code by its name.

**Parameter**: a variable’s way into the function. We can **return** a variable value using the **return keyword.**

**Visibility:** This tells us what can be called from the functions and where it can be called to.

**Example:**

The “buy” function below has a payable scope because it is changing the amount of ether available in the network. It checks to see that the msg.value (a member of msg/message object) contains the amount of ether that we allow to be sent. This program only allows 1 ether. It then sends the balance to the msg.sender (the person connected with the contract).

function buy() payable{

if (msg.value == 1 ether) {

balances[msg.sender] += 1;

etherBalance -= 1;

}

}

**Events:**

**Events** - an event stores what is passed into it as arguments into blockchain. The members (variables/functions) of the event are passed into the event from the contract. They are stored in transaction logs when emitted. They are declared with the event keyword then given a name while parameters are passed into the parentheses.

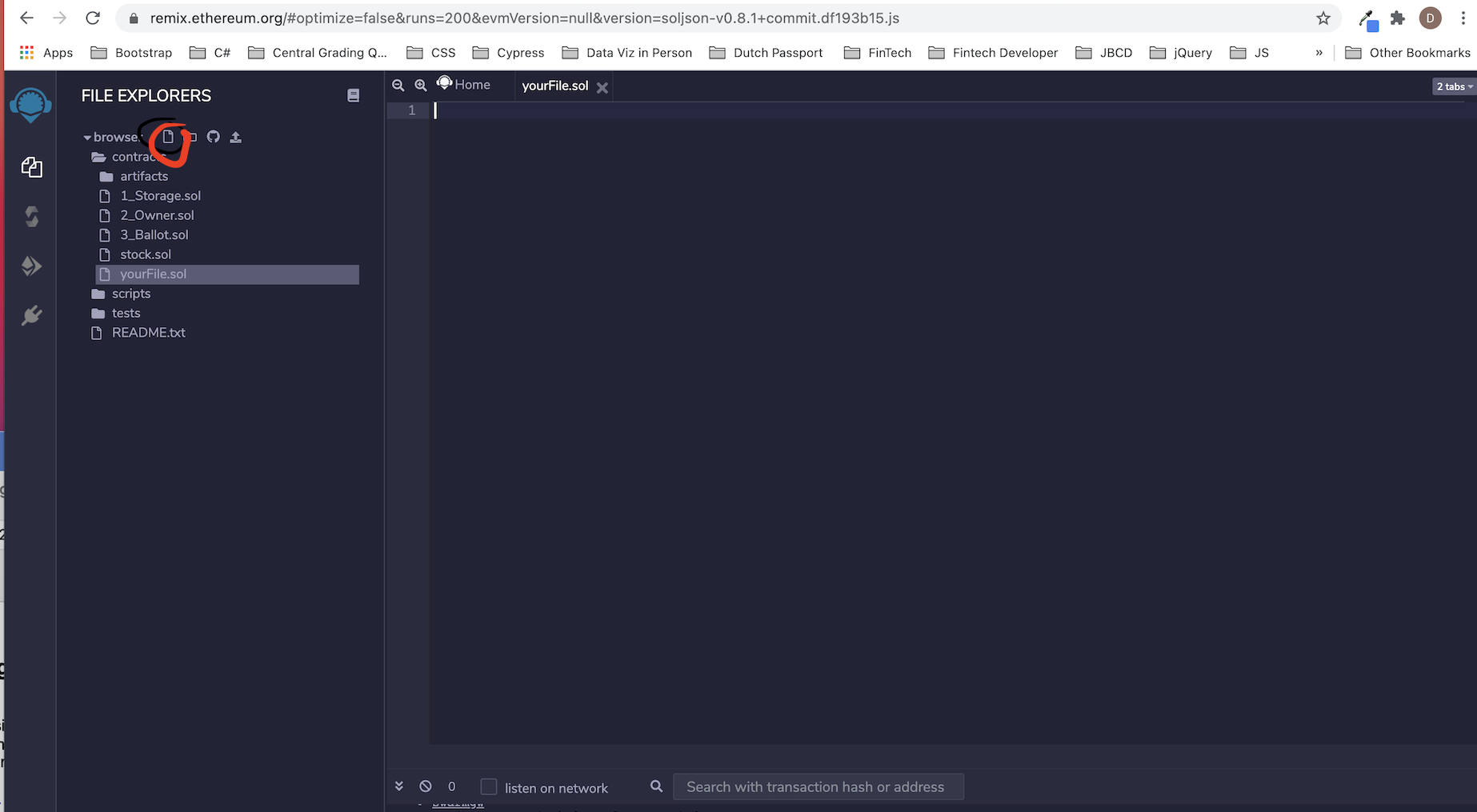
**Writing contracts:**

The website below called Remix allows you to write Solidity contracts in your browser. You are able to compile them, run, and deploy them here as well! Here is how we will set up a new file so you can start coding. You can use any code editor you would like but if you want to use Remix Please click the link:

[Remix - Ethereum IDE](https://remix.ethereum.org/)

[remix.ethereum.org](https://remix.ethereum.org/)

Let’s add a new file! Please click the contracts folder. The icon that looks like a piece by the word browser will add a new file when clicked. Files, end written in the Solidity programming language end in .sol so please name your file like this example yourFileName.sol.



**Happy pseudo coding! What is pseudo coding?**

Pseudo coding is a description in plain English about the code you are going to write. It is a developer’s blueprint to the algorithms they design, similar to how an architect sketches homes before they build them. In order to pseudo code, it is important to understand the task so let’s go over the instructions

**Instructions:**

As previously mentioned, you are a developer creating a contract that is monitoring stock exchanges for a corporate client, your code will need the following:

* A share to be exchanged.
* A transaction to occur.
* A count for each transaction and assets. Don’t lose track of money here!
* Track each new stock or share of stock added and update the blockchain network accordingly each time a transaction happens, a new type stock is added, or the quantity of the stock shares and price change. **Remember the events and emit**.
* Your boss asked that this contract ensures that a transaction won’t occur if there are not enough shares of the requested stock in the network. She also does not want to lose business if the transaction is able to partially execute.

**Logistics:**

* We will need to reference list of list of stock shares and transactions throughout this process.
* A way to check for the correct stock in our program (something to reference our shares id). A comparing function!
* A function to change the count of the transaction and assets.

**To do list:** The list of comments and the starter code below can be copied and pasted into the file you created on Remix or you can download the starter code [here](https://github.com/DevonMartens/SolidityContractLesson/blob/main/starter_code/start.sol). Each portion of the activity is broken down into parts (i.e. functions, structs, ect). There are more comments early on in the code than later. This is to avoid repetition. These comments are hints, and there are more hints in each step. Good luck!

// add pragma to specify Solidity version

pragma solidity ^0.8.1;

//Name of the contract it begins here

contract StockExchange {

/\* TO DO: create struct for the shares it is safe to say that we will need a quantity/price and address to be sent and an identifier\*/

// the transaction struct here

/\* a transaction needs a price/quantity also, we should track when it occurred, if it was successful

and the buyer and seller\*/

struct Transaction {

bytes6 buyer; // 6 bytes string

bytes6 seller; // 6 bytes string

int quantity;

int price;

uint256 timestamp;

int8 state; // 0:PENDING / 1:VALIDATED / 2:REJECTED

}

// TO DO: use int or numbers to count our transactions and assets (create variables)

// TO DO: enable mapping for each struct so we can access our transaction and asset

//TO DO: write your first function

/\*Your first function! The function is declared with the word function it passes in two strings as parameters a & b.

This function is pure which ensures it does not modify the state or the strings that are passed into it in the future

The word internal means that this function can only be accessed within the contract. We are returning a true or false or a

bool \*/

//TO DO: add a for loop to your function

/\* The for loop has 3 arguments. They are separated by a ;

1. The first is the starting value the unsigned int is = to 0.

2. the second (i < 6) is a test statement. Until this is no longer true the loop will execute

3. the final statement is adding 1 to i with ++ which starts at 0. The loop will execute 6 times.

One for each of the previously declared bytes \*/

/\* the if statement will check if a is not equal to b with the != operator. The index(or total string) of

a and b is passed in with [i] and will return false\*/

//otherwise it will return true

//TO: add events here. 3 of them with parameters passed in including address index(es) price quantity timestamp

//buyer seller if applicable

/\* TO DO: Write a function that gets an asset index in the mapping by id

-the function will be public is used because we are storing some of this into block chain

-View is used here because it does not modify the state\*/

//TO DO: write a for loop that references asset count. I starts at 1 rather than the zero index

//TO: add an if statement to check the id and if it is correct return the inedec

//If it is not return -1 for false this ends the function

// TO DO: write a function that returns assets count

//TO DO: write a function that returns

//TO DO: write a function that registers an asset into the assets list

/\* This function registers an asset into the assets list(it is payable because it allows ether to be added to the network)

it is public because we need to access blockchain and modify the available assets.

if else statement will be sucheck if this a new asset or if we are updating an assets price and quantity. We are emitting the

events for either case with the emit statement.

\*/

//get the real number using asset\_count

//the memory keyword is used here because this is not being saved to blockchain

//count

//update blockchain with all in. msg.sender represents the address of the current user

//if we already have that id in place we will change the quantity and price

// TO DO: write a function that gets an asset by index in the list

//TO DO: Gets an asset by id (string)

//TO DO: write a function is declared the source, target, quantity are parameters public is the visibility

//if there is no target or source return false

//else

//add a new transaction to the transaction count

//if statement emits if there are enough stock and update the assets accordingly

// validate transaction

//set a new quantity

//tell the count this worked

//update logs with transaction and asset information

// validate partial transaction where only some of the stocks wanted were sold- still a transaction and the remainder is rejected

// create the rejected transaction

//otherwise the transaction is rejected but still executed

//return true

// Get transaction by index

function getTransactionByIndex(int i) public view

returns (bytes6 buyer, bytes6 seller, int quantity, int price, uint256 timestamp, int8 state ) {

if (i >= 1 && i <= transaction\_count)

return (transactions[i].buyer, transactions[i].seller,

transactions[i].quantity, transactions[i].price, transactions[i].timestamp, transactions[i].state);

else

return ("DEVONM", "DEVONM",-1,-1, 0, 2);

}

// Get the next transaction index involving an share

function getNextTransactionIdInvolvingShare(bytes6 id, int start) public view returns (int) {

if (start < 1 || start > transaction\_count)

return -1;

for (int i = start; i <= transaction\_count; i++) {

if (compareStrings(transactions[i].buyer, id) == true || compareStrings(transactions[i].seller, id))

return i;

}

return -1;

}

}

**Starter code:** A **struct** is given and the contract is opened which I will explain in detail. Pay attention to the last two functions in your program. They will be good for reference as you write your other functions. These functions have for **loops,** **returns,** and **if/else** statements you can view as examples.

**Happy Coding!**

**Solution:**

The link [here](https://github.com/DevonMartens/SolidityContractLesson/blob/main/starter_code/start.sol) is the to the solution file. I walk through each part of the solution in this lesson.

**Development: It is time to code!**

Remember the word **pragma** from the syntax lesson? Pragma is used to start every document in Solidity. The latest version of Solidity 0.8.1. is declared. Then the name of your contract.

pragma Solidity ^0.8.1;

contract YourContractName {

}

The share(s) you are sending for this assignment is what is being sent. A description of what is being sent is important. Each individual stock has a price, the number of shares you are sending has quantity, and the quantity belongs to a certain company. For example, Gamestop stock is represented by the GME abbreviation - we will use a string to declare this. We will reference this a lot, so I recommend using something like the word “id”. In order to send anything, even a letter, you need a destination.

Everything that is being sent needs an **address;** this is the **address variable.** Every account on the Ethereum blockchain has an address that you can send and receive from just like your mail, or an email, or a bank account number. It is not necessary to create a real address in this lesson. However, if you would like to create one for transactions in the future, here is a [video](https://btcdirect.eu/en-gb/ethereum-wallet) with more information and a [how-to guide](https://medium.com/the-ethereum-name-service/step-by-step-guide-to-registering-a-eth-name-on-the-new-ens-registrar-c07d3ab9d6a6).

The other variables for the amount and quantity will be integers declared as **int** because they are numbers. **In summary**, **in order to send shares we need a destination, a company name for the stock name, a price for the stock, and the amount of stock we are sending.**

Since this transaction on the Ethereum network will use memory on this network, the more memory you use the more money, or gas, your company spends. In order to save money, we will make “share” a struct. Your employer will be happy that you are saving your company money by having the contract use less memory. A struct has a group of elements (your variables), and, in this case, represents a record.

Here is a sample struct (not the one you will be writing) for a pool party where we don’t want any one person to eat all the chips, and we don’t want any children to come. Please take a moment to write your struct!

contract PoolParty {

struct People {

int chipsForEachPerson;

bytes32 name; //32 because people can have long names

int age;

}

}

Your struct should look something like this now:

pragma Solidity ^0.8.1;

contract StockExchange {

struct Share {

address creator;

bytes6 id; // 6 bytes string

int price;

int quantity;

}

}

Now that we have specified the asset we are sending, let’s move on and talk about the struct for the transaction. The structure for the transaction needed the following:

1. A transaction usually has a buyer and a = seller (the Buyer buys the stock from the **seller**).

1. A buyer will buy a certain quantity of stocks at a certain price.
2. The timestamp for this will be an **unassigned integer which** can be from 6 – 256 bytes. This is the time of transaction execution or **now**/ **block.timestamp**.
3. The state of the transaction can be one the following we will use numbers to check this:
   * Pending (0): when the transaction has been created.
   * Validated (1): when the transaction is valid, this means that the target quantity is credited and the transaction has been executed.
   * Rejected (2): in this case, the desired quantity is not available

The transaction struct should look like this:

// The transaction structure

struct Transaction {

bytes6 seller; // 6 bytes string

bytes6 buyer; // 6 bytes string

int quantity;

int price;

uint256 timestamp;

int8 state; // 0:PENDING / 1:VALIDATED / 2:REJECTED

}

It is important for businesses to track the transactions that are being made and the assets that are being sent. For this reason, variables that track these should not be written inside a function. They need to be written inside the contract as a **state variable** so they will be saved to blockchain.

//counting our transactions and shares

int transaction\_count;

int shares\_count;

The keyword **mapping** in Solidity sets up key value pairs. Structs work well with mapping which will reference them by their Ethereum address on the blockchain. We are going to set up mapping for each struct so we can reference it in our data.

// mapping for each struct

mapping (int => Transaction) transactions;

mapping (int => Shares) share;

It is time to write your first function in Solidity. Feel free to do some research on the topic but you do not need to because it will all be explained. Google and other search engines are amazing resources for programmers. The function needed for this program will compare two strings, or the byte character, from our struct. This will check the id for our future transactions. We will return “false” if the strings are not the same, and “true” if they are. **Hint: I used a “for loop”, an “if statement”, and the “return”. I also used “boolean operators” to determine what this function returns.**

/\*Your first function the function is declared with the word function it passes in two strings as parameters a & b.

This function is pure which ensures it does not modify the state or the strings that are passed into it in the future

The word internal means that this function can only be accessed within the contract. We are returning a true or false or a

bool \*/

function compareStrings(bytes6 a, bytes6 b) internal pure returns (bool) {

/\* The for loop has 3 arguments. They are separated by a ;

1. The first is the starting value, the unsigned int is = to 0.

2. the second (i < 6) is a test statement. Until this is no longer true the loop will execute

3. the final statement is adding 1 to i with ++ which starts at 0. The loop will execute 6 times.

One for each of the previously declared bytes \*/

for (uint i = 0; i < 6; i ++) {

/\* the if statement will check if a is not equal to b with the != operator. The index(or total string) of

a and b is passed in with [i] and will return false\*/

if (a[i] != b[i]) {

return false;

}

}

//otherwise it will return true

return true;

}

The first piece of this program is now complete! The function you just wrote was hard, don’t worry if it didn’t turn out perfectly. The next challenge is a tricky one as well, but by the end of this lesson you will have a smart contract that you wrote to show off your development skills.

This function uses the last function you wrote and mapping to get the asset index (in other words, the struct you built that I have referenced in my examples by using (Share) by the id or what you chose in your struct.) We will use a return function again to get the **index** of the asset by a given **id (bytes6). Hint: We will use the last function we wrote. The id will be a parameter. In addition, the structure will be similar with a “for loop”, an “if statement”, and a “return” (we will return the index OR -1). Final hint: make this function public.**

/\* Gets an asset index in the mapping by id

-public is used because we are storing some of this into block chain

-View is used here because it does not modify the state

- \_id signifies a place holder/allowing space for a function

\*/

function getShareIndex(bytes6 \_id) public view returns (int index) {

//the index in this for loop references asset count. I starts at 1 rather than the zero index

for (int i = 1; i <= share\_count; i++) {

//asset count is referencing assets if the strings are the same we return the index

if(compareStrings(shares[i].id, \_id) == true)

return i;

}

//or negative one for false

return -1;

}

That was a tough function! The next two functions are very similar to one another, so, to simplify things let’s review them together. We are going to get the number of transactions and sharess share\_count and the number of available transactions transaction\_count here.

// returns assets count

function getSharesCount() public view returns (int uid) {

return asset\_count;

}

// returns transactions count

function getTransactionsCount() public view returns (int uid) {

return transaction\_count;

}

Awesome - you are about halfway through writing your first stock trading smart contract! Let’s check our code to make sure we are not missing anything. Your code should look something like this:

pragma solidity ^0.8.1;

contract StockExchange {

struct Share {

address creator;

bytes6 id; // 6 bytes string

int price;

int quantity;

}

// The transaction structure

struct Transaction {

bytes6 buyer; // 6 bytes string

bytes6 seller; // 6 bytes string

int quantity;

int price;

uint256 timestamp;

int8 state; // 0:PENDING / 1:VALIDATED / 2:REJECTED

}

// shares counted/transaction count

int transaction\_count;

int share\_count;

// map to struct

mapping (int => Transaction) transactions;

mapping (int => Share) shares;

// compare 6 bytes string

function compareStrings(bytes6 a, bytes6 b) internal pure returns (bool) {

for (uint i = 0; i < 6; i ++) {

if (a[i] != b[i]) {

return false;

}

}

return true;

}

// Gets an share index in the mapping by id

function getShareIndex(bytes6 \_id) public view returns (int index) {

for (int i = 1; i <= share\_count; i++) {

if(compareStrings(shares[i].id, \_id) == true)

return i;

}

return -1;

}

// Gets shares count

function getSharesCount() public view returns (int uid) {

return share\_count;

}

// Gets transactions count

function getTransactionsCount() public view returns (int uid) {

return transaction\_count;

}

}

}

Alright, congratulations, we are halfway through this lesson! Before moving forward with more functions, we need to write events for our function to use. Let’s send some stuff to blockchain! Since we are using assets and transactions, we’ll need to keep a record of a few things: if an asset joins the network, if an asset joins the network with a new price or quantity, and if a transaction is executed.

Here is an example:

event ShareJoined(address indexed share\_address, int index, bytes6 id, int quantity, int price, uint256 timestamp);

Take that line of code and place it under your compareStrings function. That is the first of the three events you will need. It is tracking assets that have joined the network. The event includes the asset address, the index, the id, the quantity, and timestamp. The variables placed in the even probably all make sense now, minus the word **indexed**. This keyword is allowing your program to search for variables. It will be used in your next two events. Please take some try to code them.

All the events completed should look like this:

event ShareJoined(address indexed share\_address, int index, bytes6 id, int quantity, int price, uint256 timestamp);

event TransactionExecuted(address indexed buyer\_address, int buyer\_share\_index, int seller\_share\_index, bytes6 buyer,

bytes6 seller, int quantity, int price, uint256 timestamp, int8 state);

event ShareUpdated(address indexed share\_address, int index, bytes6 id, int quantity, int price, uint256 timestamp);

Please scroll down almost to the very bottom of your code. Still inside the contract (the bottom curly brace needs to be below this next function.) This function will register assets to the asset list. We will need to check the id with an if/else statement if the asset has been added to the asset list already. If the index = -1 then we will count the assets and add the needed information, and if we already have the stock, then we will need to change the quantity and perhaps the price.

// Registers a share into the shares list in the events/sends to sender

function register(bytes6 \_id, int \_quantity, int \_price) public payable returns (bool success) {

int share\_index = getShareIndex(\_id);

if(share\_index == -1){

share\_count = share\_count + 1;

Share memory \_share = Share(msg.sender, \_id, \_price, \_quantity);

shares[share\_count] = \_share;

emit ShareJoined(msg.sender,share\_count, \_id, \_quantity, \_price, block.timestamp);

} else {

shares[share\_index].quantity = \_quantity;

shares[share\_index].price = \_price;

emit ShareUpdated(msg.sender, share\_index, \_id, \_quantity, \_price, block.timestamp);

}

return true;

}

We will write something similar for the transaction to check if it worked and update accordingly. First, we need to write two functions - both will do almost the same thing. They will check for the asset and retrieve the information. The only difference is one function will check by the name and the other will check by the id. Otherwise it will return the following ("DEVONM",-1,-1);.

/ Gets an share by index in the list

function getShareByIndex(int i) public view returns (bytes6 id, int price, int quantity) {

if(i >= 1 && i <= share\_count)

return (shares[i].id, shares[i].price, shares[i].quantity);

else

return ("DEVONM",-1,-1);

}

// Gets an share by id (string)

function getShare(bytes6 \_id) public view returns (bytes6 id, int price, int quantity) {

for (int i = 1; i <= share\_count; i++) {

if(compareStrings(shares[i].id, \_id) == true)

return (shares[i].id, shares[i].price, shares[i].quantity);

}

return ("DEVONM",-1,-1);

}

Alright we are ready for transactions to occur. Before we write the function, try to determine the requirements based on the following information:

* We are adjusting ether available and number of shares in the network.
* The transaction needs a buyer and a seller (we can’t have a transaction without either one).
* The transaction should not occur if there are not enough stocks, but a partial transaction should occur if there are some of the stocks, and if there none of the requested stocks no transaction should occur.
* The logs should include updated asset information about the transactions.

Please see the solution to this portion of the activity below:

function transact(bytes6 buyer, bytes6 seller, int quantity) public payable returns (bool success) {

//if there is no seller or buyer retun false

int si = getShareIndex(buyer);

int ti = getShareIndex(seller);

if(si == -1 || ti == -1) {

return false;

}

else {

//add a new transaction to the transaction count

transaction\_count = transaction\_count + 1;

Transaction memory \_t = Transaction(shares[si].id, shares[ti].id, quantity, shares[ti].price, block.timestamp, 0);

transactions[transaction\_count] = \_t;

//emits if there are enough stock and update the shares accodingly

if((shares[ti].quantity - quantity) >= 0) { // validate transaction

shares[ti].quantity -= quantity;

transactions[transaction\_count].state = 1;

emit TransactionExecuted(msg.sender,si,ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

emit ShareUpdated(shares[ti].creator, ti, shares[ti].id, shares[ti].quantity, shares[ti].price, block.timestamp);

}

else if(shares[ti].quantity > 0) { // validate partial transaction

transactions[transaction\_count].state = 1;

transactions[transaction\_count].quantity = shares[ti].quantity;

emit TransactionExecuted(msg.sender, si, ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

// create the rejected transaction

transaction\_count = transaction\_count + 1;

Transaction memory \_t1 = Transaction(shares[si].id, shares[ti].id, quantity - shares[ti].quantity, shares[ti].price, block.timestamp, 0);

transactions[transaction\_count] = \_t1;

shares[ti].quantity = 0;

transactions[transaction\_count].state = 2;

emit TransactionExecuted(msg.sender, si, ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

emit ShareUpdated(shares[ti].creator, ti, shares[ti].id, shares[ti].quantity, shares[ti].price, block.timestamp);

}

else {

transactions[transaction\_count].state = 2;

emit TransactionExecuted(msg.sender, si, ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

}

}

return true;

}

**Complete solution:**

Please see the code sample below. You can check your code against it.

pragma solidity ^0.8.1;

contract StockExchange {

struct Share {

address creator;

bytes6 id; // 6 bytes string

int price;

int quantity;

}

// The transaction structure

struct Transaction {

bytes6 buyer; // 6 bytes string

bytes6 seller; // 6 bytes string

int quantity;

int price;

uint256 timestamp;

int8 state; // 0:PENDING / 1:VALIDATED / 2:REJECTED

}

// shares counted/transaction count

int transaction\_count;

int share\_count;

// map to struct

mapping (int => Transaction) transactions;

mapping (int => Share) shares;

// compare 6 bytes string

function compareStrings(bytes6 a, bytes6 b) internal pure returns (bool) {

for (uint i = 0; i < 6; i ++) {

if (a[i] != b[i]) {

return false;

}

}

return true;

}

// events stored in blockchain

event ShareJoined(address indexed share\_address, int index, bytes6 id, int quantity, int price, uint256 timestamp);

event TransactionExecuted(address indexed buyer\_address, int buyer\_share\_index, int seller\_share\_index, bytes6 buyer,

bytes6 seller, int quantity, int price, uint256 timestamp, int8 state);

event ShareUpdated(address indexed share\_address, int index, bytes6 id, int quantity, int price, uint256 timestamp);

// Gets an share index in the mapping by id

function getShareIndex(bytes6 \_id) public view returns (int index) {

for (int i = 1; i <= share\_count; i++) {

if(compareStrings(shares[i].id, \_id) == true)

return i;

}

return -1;

}

// Gets shares count

function getSharesCount() public view returns (int uid) {

return share\_count;

}

// Gets transactions count

function getTransactionsCount() public view returns (int uid) {

return transaction\_count;

}

// Registers a share into the shares list in the events/sends to sender

function register(bytes6 \_id, int \_quantity, int \_price) public payable returns (bool success) {

int share\_index = getShareIndex(\_id);

if(share\_index == -1){

share\_count = share\_count + 1;

Share memory \_share = Share(msg.sender, \_id, \_price, \_quantity);

shares[share\_count] = \_share;

emit ShareJoined(msg.sender,share\_count, \_id, \_quantity, \_price, block.timestamp);

} else {

shares[share\_index].quantity = \_quantity;

shares[share\_index].price = \_price;

emit ShareUpdated(msg.sender, share\_index, \_id, \_quantity, \_price, block.timestamp);

}

return true;

}

// Gets an share by index in the list

function getShareByIndex(int i) public view returns (bytes6 id, int price, int quantity) {

if(i >= 1 && i <= share\_count)

return (shares[i].id, shares[i].price, shares[i].quantity);

else

return ("DEVONM",-1,-1);

}

// Gets an share by id (string)

function getShare(bytes6 \_id) public view returns (bytes6 id, int price, int quantity) {

for (int i = 1; i <= share\_count; i++) {

if(compareStrings(shares[i].id, \_id) == true)

return (shares[i].id, shares[i].price, shares[i].quantity);

}

return ("DEVONM",-1,-1);

}

// executes a transaction where buyer buys from seller a certain quantity

//Here a function is declared the buyer, seller, quantity are parameters public is the visability

function transact(bytes6 buyer, bytes6 seller, int quantity) public payable returns (bool success) {

//if there is no seller or buyer retun false

int si = getShareIndex(buyer);

int ti = getShareIndex(seller);

if(si == -1 || ti == -1) {

return false;

}

else {

//add a new transaction to the transaction count

transaction\_count = transaction\_count + 1;

Transaction memory \_t = Transaction(shares[si].id, shares[ti].id, quantity, shares[ti].price, block.timestamp, 0);

transactions[transaction\_count] = \_t;

//emits if there are enough stock and update the shares accodingly

if((shares[ti].quantity - quantity) >= 0) { // validate transaction

shares[ti].quantity -= quantity;

transactions[transaction\_count].state = 1;

emit TransactionExecuted(msg.sender,si,ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

emit ShareUpdated(shares[ti].creator, ti, shares[ti].id, shares[ti].quantity, shares[ti].price, block.timestamp);

}

else if(shares[ti].quantity > 0) { // validate partial transaction

transactions[transaction\_count].state = 1;

transactions[transaction\_count].quantity = shares[ti].quantity;

emit TransactionExecuted(msg.sender, si, ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

// create the rejected transaction

transaction\_count = transaction\_count + 1;

Transaction memory \_t1 = Transaction(shares[si].id, shares[ti].id, quantity - shares[ti].quantity, shares[ti].price, block.timestamp, 0);

transactions[transaction\_count] = \_t1;

shares[ti].quantity = 0;

transactions[transaction\_count].state = 2;

emit TransactionExecuted(msg.sender, si, ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

emit ShareUpdated(shares[ti].creator, ti, shares[ti].id, shares[ti].quantity, shares[ti].price, block.timestamp);

}

else {

transactions[transaction\_count].state = 2;

emit TransactionExecuted(msg.sender, si, ti, transactions[transaction\_count].buyer, transactions[transaction\_count].seller, transactions[transaction\_count].quantity, transactions[transaction\_count].price, transactions[transaction\_count].timestamp, transactions[transaction\_count].state);

}

}

return true;

}

// Get transaction by index

function getTransactionByIndex(int i) public view

returns (bytes6 buyer, bytes6 seller, int quantity, int price, uint256 timestamp, int8 state ) {

if (i >= 1 && i <= transaction\_count)

return (transactions[i].buyer, transactions[i].seller,

transactions[i].quantity, transactions[i].price, transactions[i].timestamp, transactions[i].state);

else

return ("DEVONM", "DEVONM",-1,-1, 0, 2);

}

// Get the next transaction index involving an share

function getNextTransactionIdInvolvingShare(bytes6 id, int start) public view returns (int) {

if (start < 1 || start > transaction\_count)

return -1;

for (int i = start; i <= transaction\_count; i++) {

if (compareStrings(transactions[i].buyer, id) == true || compareStrings(transactions[i].seller, id))

return i;

}

return -1;

}

}

**Compile**

The Remix IDE (the website that we wrote our code on) also allows you to compile your code. While in the file you just wrote click the button on the side of your browser that looks like two arrows going in a circle. Then click the button that says compile yourFileName.sol. See the photo below, my file is called stock.

Graphical user interface, application

Description automatically generated

If you notice any errors on the bottom of your screen double check your code against the solution. Since we did not use a real address, we will not deploy this contract. Here is some [documentation on deploying](https://remix-ide.readthedocs.io/en/latest/run.html) with remix if you are interested.

**Task Recap: Congratulations!**

Now that you have developed your first smart contract and your boss is impressed with your work, you have time to go over what you learned to do with Solidity

1. Why blockchain developers are so valuable

2. Ethereum blockchain fundamentals

- How the Network works

- Mining and what it is

- Gas

- Ether

3. Smart Contracts and how they work!

4. Solidity fundamentals including:

- Declare the version of Solidity for the compiler to use with pragma.

- Create a contract.

- Events (declare and emit)

- Functions

- Variables

- Returns

- If/else statements

- For loops

- Mapping

5. The remix IDE

- How to write/compile a contract on remix

**Next**

In our next lesson we will learn how to deploy contracts and create and use real addresses for contracts on the Ethereum network.