FT5004 Lab 1

Solidity Basics

Solidity is like Java/Javascript in terms of its structure and style. In this lab, we will try to familiarise ourselves with the following aspects that are important:

1. Introduction to the Remix IDE
2. Contract Fundamentals
3. Storing Data
   a. Basic Data Types
   b. Complex Data Types
4. Using Functions
   a. Storage Scope
   b. Visibility
   c. Function Modifier
   d. Special Functions
5. Events
6. Library

After you have worked your way through the example, please do the exercise questions in section 2.
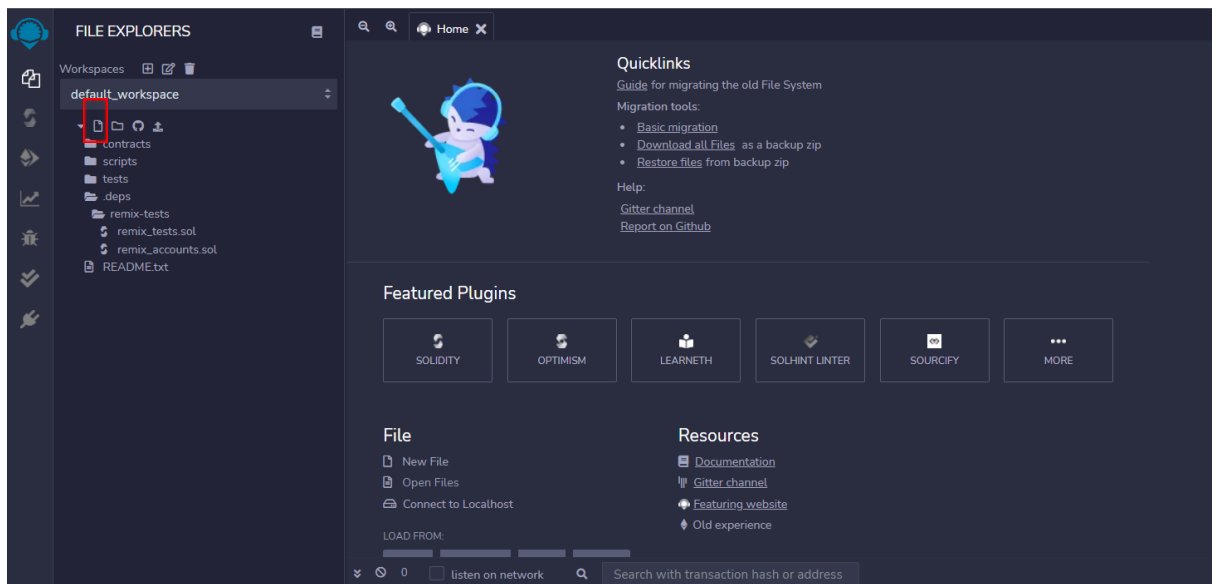

Section 1: Example

Remix IDE

Remix IDE is a web-based IDE that we can use to write and test our Ethereum smart contracts. We will use Remix as a starting point to learn more about solidity.

1. **Go to remix.ethereum.org**
2. **Click "Sure" or "Decline" depending on if you want to help improve Remix**
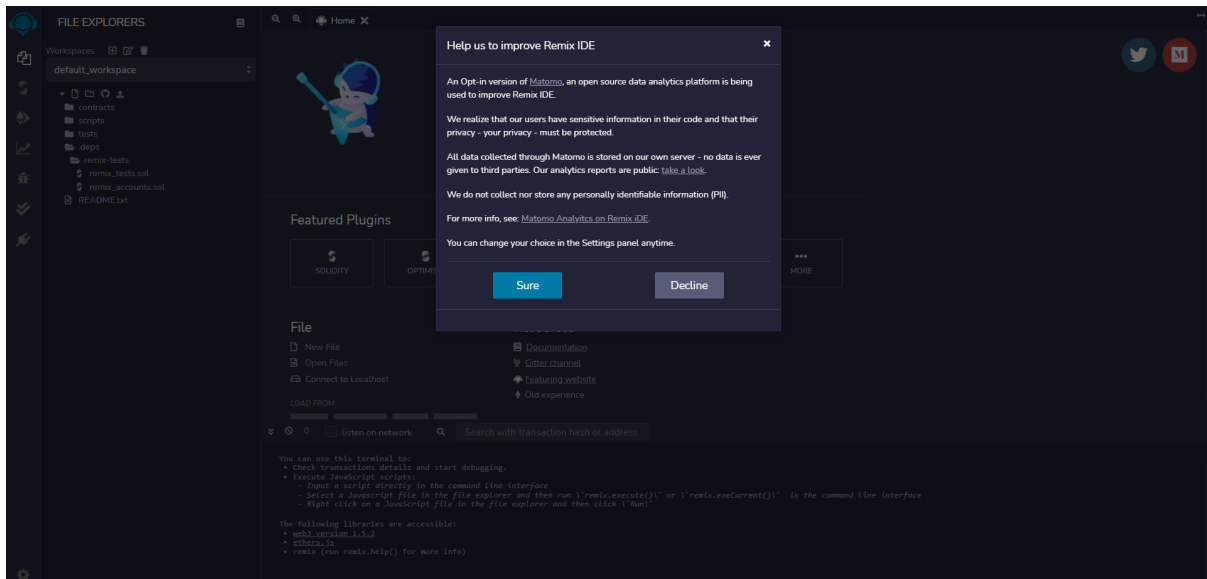
3. Look through the Remix Homepage

4. **Click the icon shown in the red box to make a new file. When prompted, name this file 'Lab2.sol'**

What we've just done is to create a new file where we can code out our smart contract. The '.sol' is the file extension, telling Remix that you are coding using the Solidity programming language.

You don't need to worry about any of the folders or file structures for now. We'll only be working on the 'Lab2.sol' file for this lab. If you have created your file, it should look something like this:
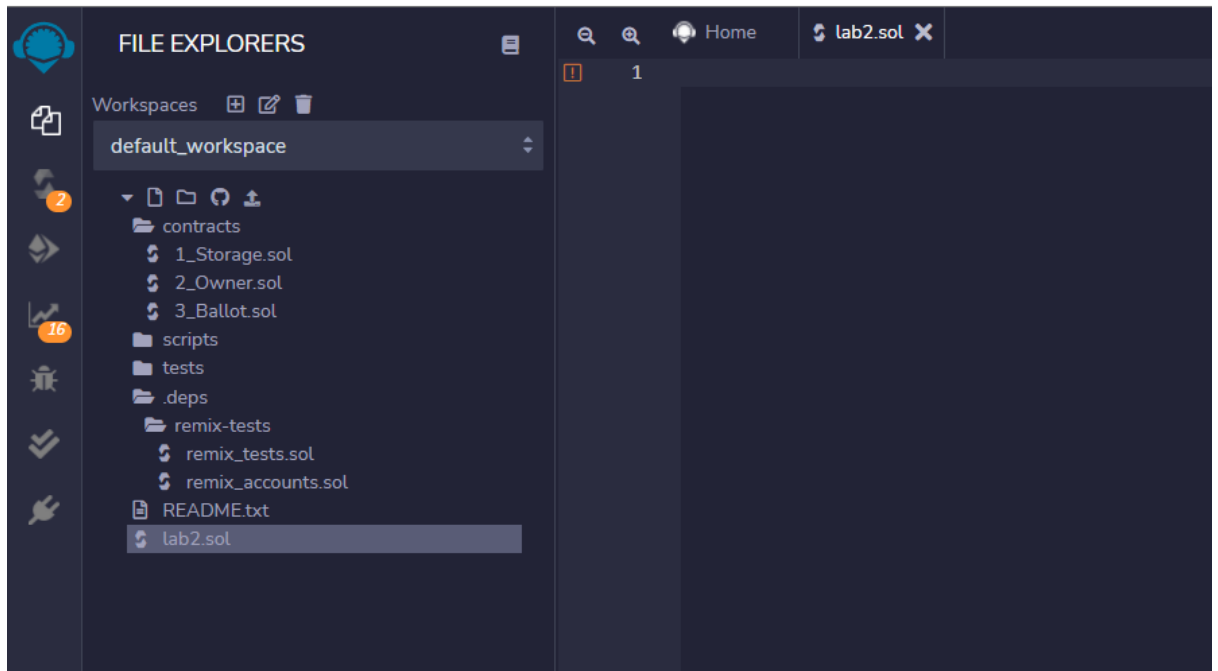
5. **You are now ready to become a Blockchain Programmer!**

Contract Fundamentals

A Smart Contract is a piece of code that tells the Blockchain what to do and how to do it. When writing a smart contract on Solidity, we need a few key ingredients, as shown below:
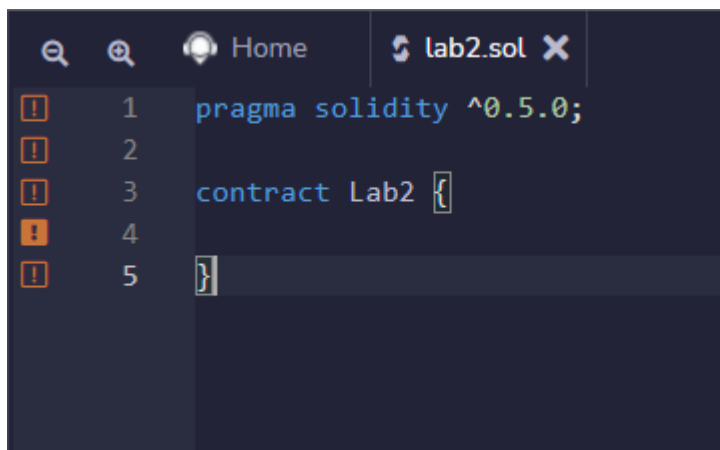
```solidity
pragma solidity 0.5.0;

contract ReadWriter {
 uint data;

 function set(uint x) public {
  data = x;
 }

 function get() public view
  returns (uint) {
   return data;
 }
}
```

Tell Remix that we want to use this version of the compiler.

Tell the compiler what your contract is called.

Populate your contract.

Pro Tip: If you are not too familiar with Java/Javascript style coding, curly braces, i.e. { } are used to compartmentalise code. Make sure that for each open bracket, there is a corresponding closed bracket. This will help you avoid a lot of debugging!

Now that we're a bit more familiar with how contracts should look, let's start creating our own smart contract. We're going to call this contract "Lab2".

Pro Tip: It is convention for the Contract name to follow the filename. This will be relevant in future labs. In our case, our file is called 'Lab2.sol', so it follows that our smart contract will be called Lab2.

1. **Establish your Solidity compiler version as 0.5.0 as so:**

Notice that we end the sentence with a ";" semicolon. All lines must end with a semicolon unless the line includes some brackets.

You might be wondering what the '^' symbol means. Solidity, like all other languages has many versions and is constantly being updated. As these updates roll in, some things might be deprecated to make way for newer features. By using the '^' symbol followed by the version of Solidity that we want the compiler to use, we are telling the compiler that this Smart Contract won't break when using version 0.5.0 and above.

2. **Create the contract Lab2 as shown in the picture above.**

You now have a basic Smart Contract. But this is a very useless contract for now. Let's work on populating it with some code.

Storing Data

Like Java, Solidity is a **strongly typed language**. This means that we must assign a type to a variable and stick to that type throughout the code.

In practice, this means that you **cannot** do something like this:

Code Chunk	Allowed?
Var1 = 3 Var1 = "Hi"	No
Var1 = 3 Var1 = 2	Yes

Solidity has a number of **basic data types** as seen below:

Solidity – basic data types

- **bool – boolean**
- **enum \<name> { \<member names> … }**
  - Eg:
    enum direction { left, right, up, down }
- **int, uint, int8, uint8, int16, uint16 … uint256**
  - signed and unsigned integers (with size, defaults to 256)
  - int = int256, uint = uint256
  - int8 => 1 byte signed integer
- **address – holds a 20 byte address**
- **contract**
- **Bytes1 .. Bytes32 – fixed size byte array**
- **String**

Solidity uses these basic data types and augments them to add additional functionality. Through this Solidity is also able to offer **complex data types:**
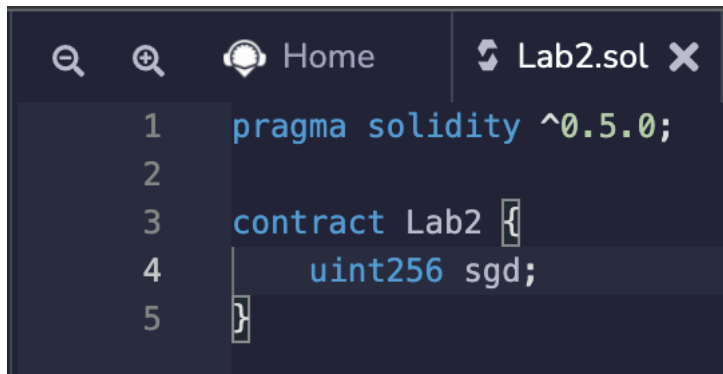
Solidity – complex data types

- **bytes – dynamic array of bytes**
- **\<type>[] – dynamic array**
  - Eg:
    uint[] numberArray;
- **mapping(\<key type> => \<value type>)
  – hash map of \<key type> to \<value type>**
  - Eg:
    mapping(address => uint) balance;
- **struct \<name> { \<member types> }**
  - Eg:
    struct record {
      uint id;
      address addr;
    }

Now that we know what types of data are available to us, let us start by storing a couple of variables.

1. **Create a variable 'sgd' inside your Smart Contract as shown:**

   This variable is of the type uint256. It will store an **unsigned integer** of size 256 bits.

2. **To this, add a complex data type: an array of uint256 numbers.**

   To create an Array, we first need to specify what type the elements of the Array will hold. Unlike Python, Solidity does not allow mixing of types in an array. This means all elements of the array MUST be of the same type.

```
uint256 sgd;
uint256[] qty;
```

   Call the array 'qty'.

3. **Awesome!** You have just added 2 variables to your contract. Next, let's see how we can instruct the compiler to manipulate these variables.

Using Functions

A function in Solidity is similar to functions in any other programming language. It takes some input (optional) and does some computation to produce an output (optional).

The structure of declaring a function in Solidity is as follows:

```
function <name> (<params>) <visibility /
   modifier>  <return type> {
      <function body> …
}
```

For every function, we must fill in the elements in the <> brackets.

1. **Let us make a simple function which accepts values for our Price**

```
function add_Price(uint256 pr) public {
    sgd = pr;
}
```

   Let us go through word by word to understand what we are asking the compiler to do
   a. **function** : tell the compiler to expect a function
   b. **add_Price**: tell it the name of the function
   c. **uint256 pr**: give it an input parameter 'pr' of the type uint256
   d. **public**: Tell the compiler who can access this function
   e. **sgd = pr**: assign the value of sgd which we had initialised to be equal to the value we just provided as a parameter.

2. **Now that we have the price of a product, let us fill up the quantity of orders received**. **We do this through another function:**

```
function add_Qty( uint256[] memory arr) public {
    qty = arr;
}
```

   This function is similar to the previous one, except that it has the keyword 'memory' in it. **Memory is to tell the compiler that it is meant for temporary storage and not long-term storage.** Understanding and managing memory is important because <u>the more memory your contract takes, the more expensive it is to deploy it to the Ethereum network.</u>

3. **Now we have a function to add the prices and a function to add the quantities. Let us say we want to know the total sales, that is sum(price * quantity). We can use a function to do this:**

```solidity
function total_Sales() public view returns (uint256) {
    uint256 ret = 0;
    for (uint i=0; i<qty.length; i++) {
        uint256 to_add = qty[i] * sgd;
        ret = ret + to_add;
    }

    return ret;
}
```

This function is more complex that the previous two. Let us see the new elements we have added:

   a. 'view' when declaring the function:

   - **payable** – <u>able to receive ether</u>. **Otherwise, the call will throw an error if ether is provided.**

   - **view** – <u>does not change state variables.</u> **Can be called without transaction fee (read-only functions, does not modify state at all)**

   - **pure**– <u>does not access state variables at all</u>. **Can be called without transaction fee. 'pure' calculation function.**
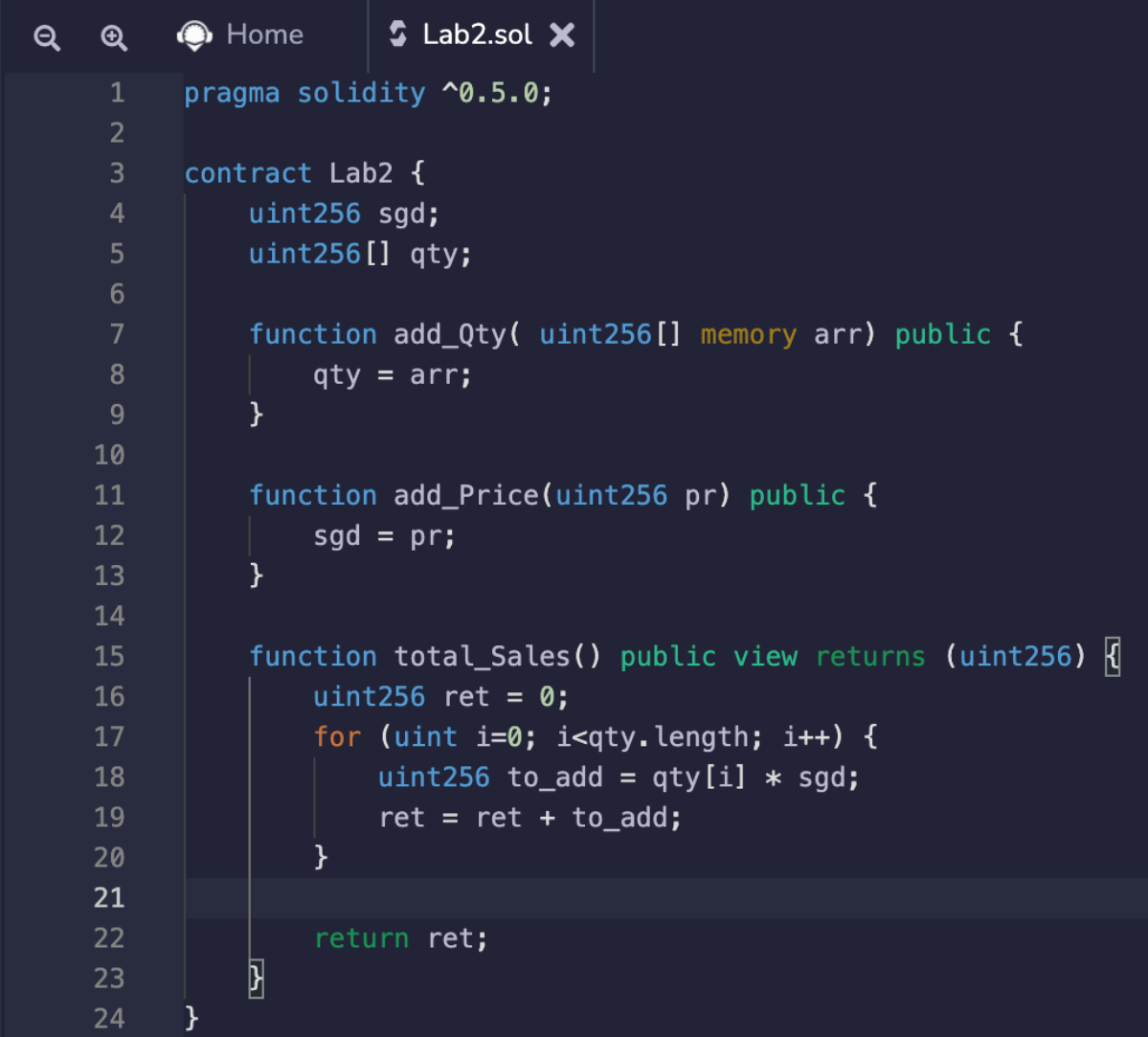
   b. for (uint i=0; i < qty.length; i++) {… :
   This is a classic **for loop**. We are iterating over the qty array. Inside the {} of this loop, we are taking the quantity * price and adding it to a temporary variable called 'ret'.

   At the end of the loop, we return 'ret' to the user.

Deploying Your Contract

Your contract should look like this:

```solidity
pragma solidity ^0.5.0;

contract Lab2 {
    uint256 sgd;
    uint256[] qty;

    function add_Qty( uint256[] memory arr) public {
        qty = arr;
    }

    function add_Price(uint256 pr) public {
        sgd = pr;
    }

    function total_Sales() public view returns (uint256) {
        uint256 ret = 0;
        for (uint i=0; i<qty.length; i++) {
            uint256 to_add = qty[i] * sgd;
            ret = ret + to_add;
        }

        return ret;
    }
}
```

If you are satisfied, go to the Compiler tab on the left:

Once there, click the **Compile Lab2.sol**. If there are any errors, you should see it below in red boxes (but you shouldn't have errors if you've done the example correctly).

Now, to see how the contract works in action, go to the third tab to deploy and run transactions:

To Deploy, click the Deploy button in Orange.

If you correctly deploy the contract, you should see the contents of the 2 red boxes below:
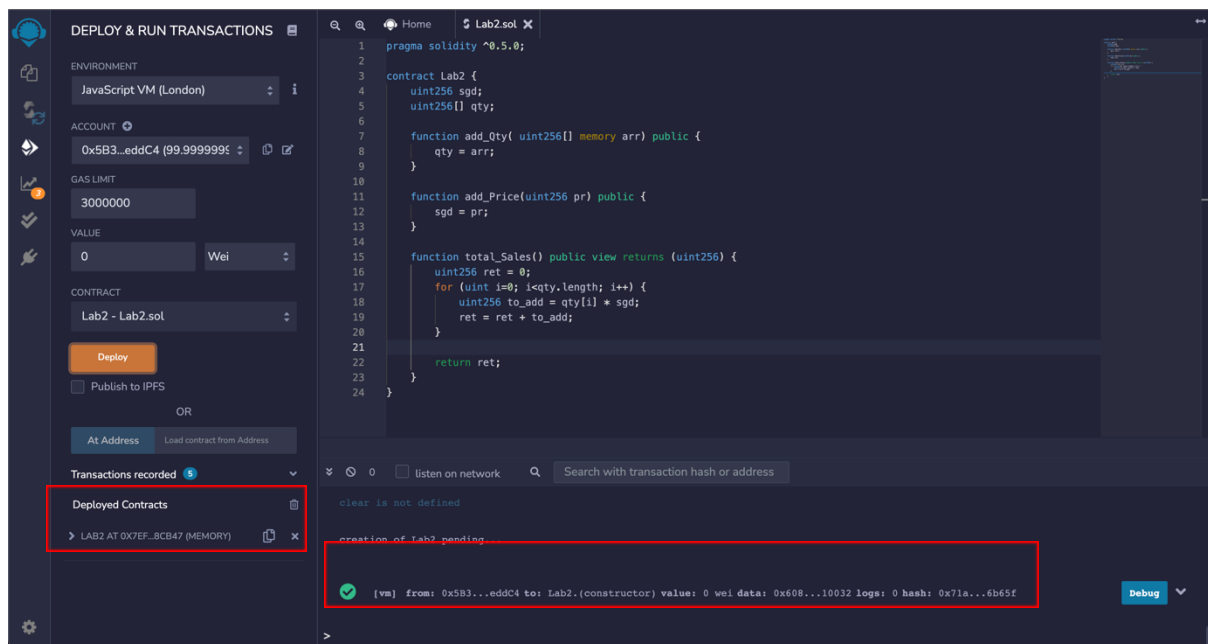

Congratulations, you have deployed your contract to a test environment!

Now, let's try playing with it. In the add_Price and add_Qty function boxes, add some sample variables. For example, pr = 3 and arr = [1, 2, 3]
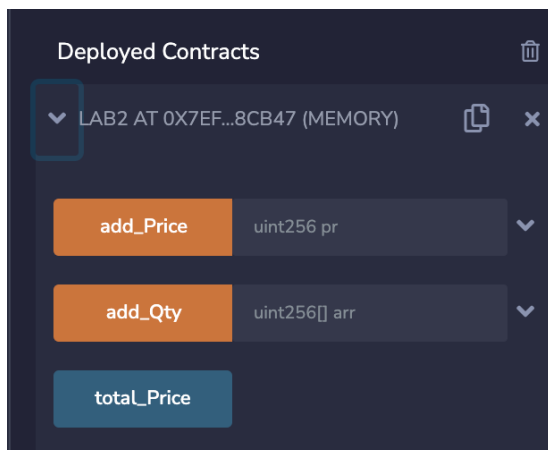

Once satisfied click the orange buttons to execute the function call. You should see green ticks in the console, indicating that your function call was successful, as so:
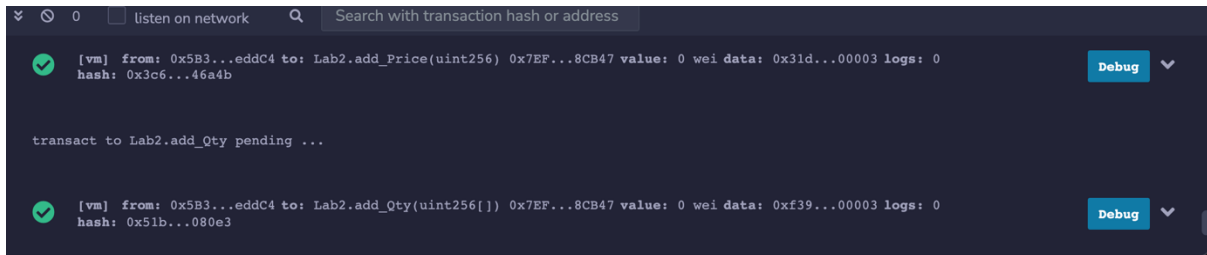
Now, let's try the last function: total_Price. This function does not take an input, but should work if you have gotten the previous two functions to work.

Using the inputs I shared earlier, I got this output:
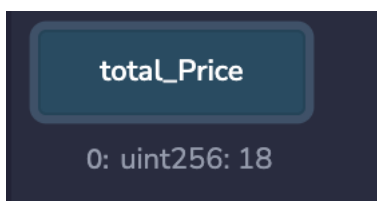
Lab 2 exercise: Build a smart contract such that the inputs are two arrays. The first array represents the price of each order and the second array represents the quantity of each order (difference from the example above is that the price could be different between orders). The output is calculating the total sales. If the length of the two inputted arrays do not match, return with an error message "Length of variables not matched." (Hint: use `require` function)

A sample output should look like this, where the output 23=2*4+3*5:
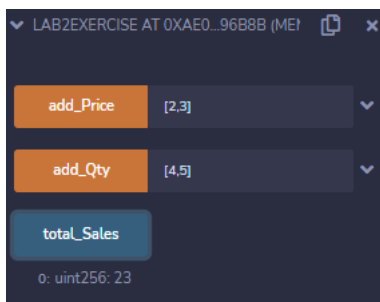
When the lengths are not matched, the return should be like this:

```
call to Lab2Exercise.total_Sales


CALL    [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Lab2Exercise.total_Sales() data: 0x2bb...4cfa0


call to Lab2Exercise.total_Sales errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "Length of variables not matched.".
Debug the transaction to get more information.
```

Submission: Please submit a zip file containing two files: Lab2.sol and Lab2Exercise.sol. The deadline of the submission is the end of next Wednesday.