**LAB 3: Solidity and Remix IDE**

**Overview**

The purpose of this lab is to help you to use an online code editor called Remix which is made specifically for creating and testing solidity contracts. It will also help you to learn a programming language called Solidity to write the actual code that gets inserted into a smart contract. This lab will also guide you on creating you first little smart contract called Inbox and to use the remix tool to test Inbox contract.

**Prerequisites:**

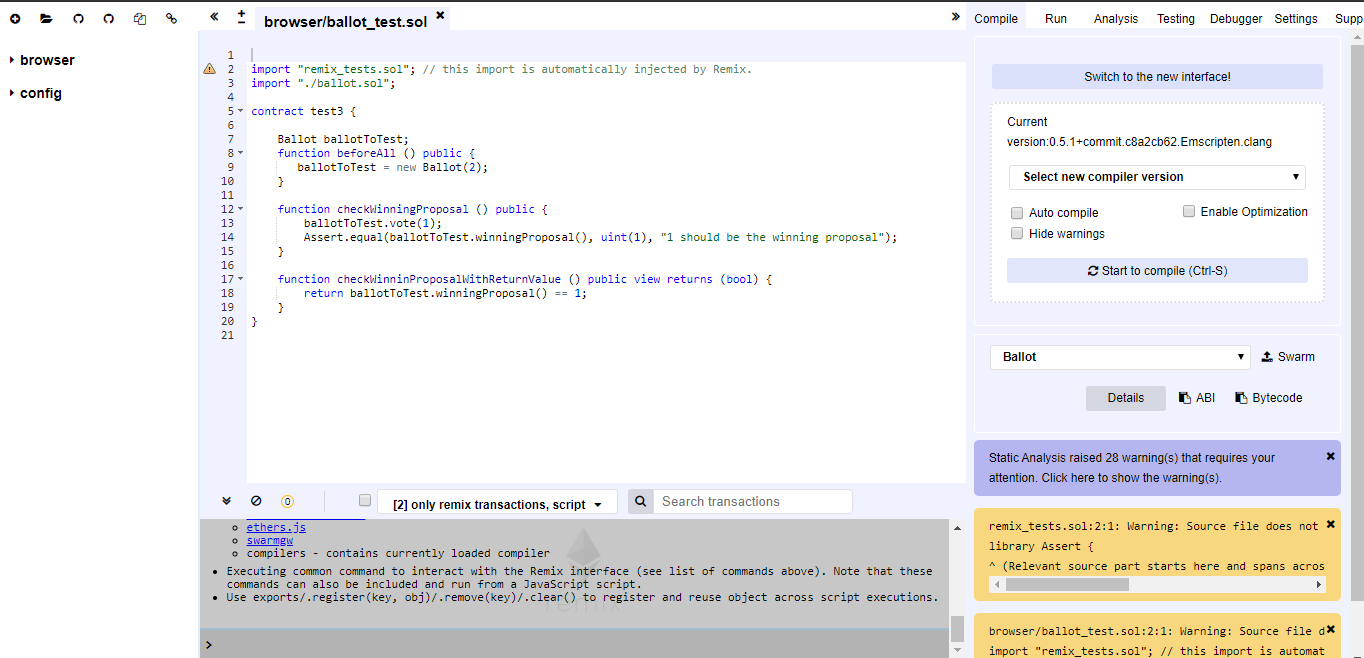
You should have completed Lab 1 – Metamask Setup, created an account in Metamask wallet.

**Remix Solidity IDE**

We're going to use an online code editor called Remix. Remix is made specifically for creating and testing solidity contracts.

* Open the Remix Solidity IDE link: <https://remix.ethereum.org/> , you will see the layout of Remix IDE, as shown below.

**Note: If you are warned about using current version of remix or previous version. Please select “Use previous version”. You need to select “Use previous version” every time you close the browser and come back again. You may also need to enter the password for your metamask account every time you close the browser and come back again.**



* On the page that pops up you're going to immediately see a bunch of code inside of a code editor. You're going to select the code inside and delete all of it. We're going to start from scratch. You can copy the following code and inside the editor:

pragma solidity ^0.4.17;

contract Inbox

{

string public message;

function Inbox(string initialMessage) public

{

message = initialMessage;

}

function setMessage(string newMessage) public

{

message = newMessage;

}

function getMessage() public view returns(string)

{

return message;

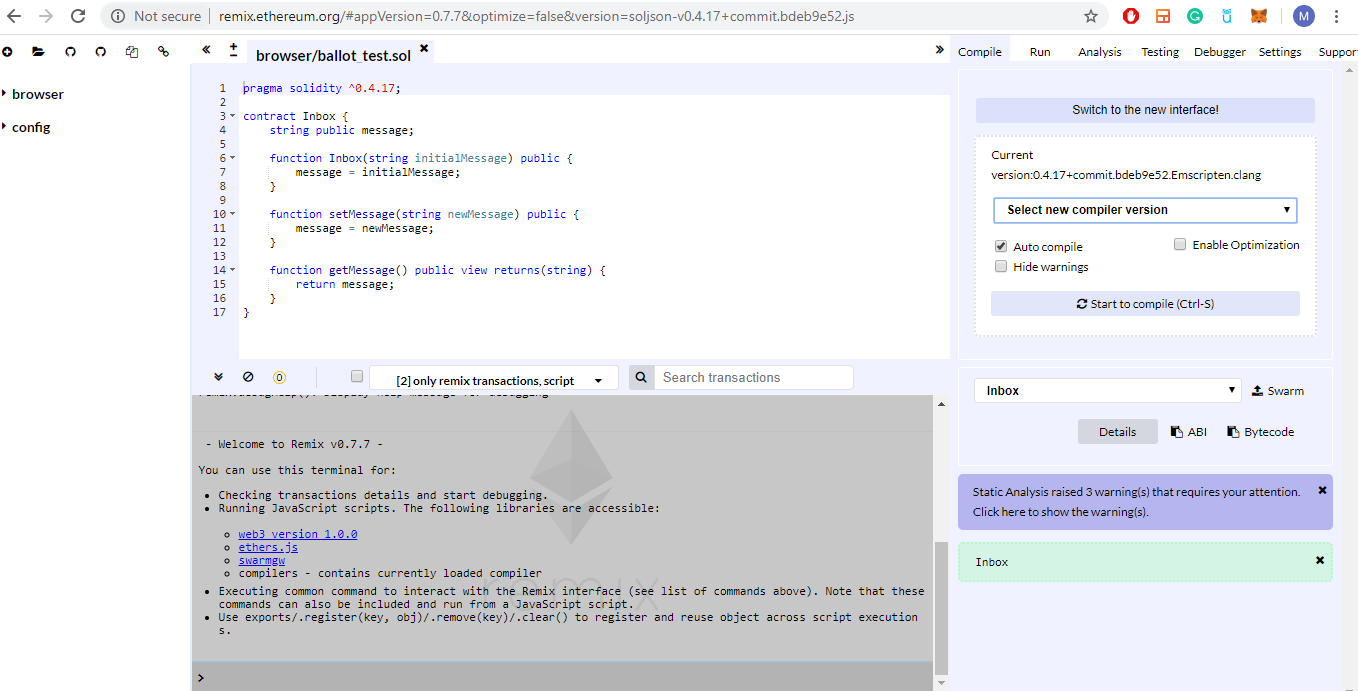
}

}

You may have noticed some deprecation or syntax warnings. Make sure you tell Remix which compiler version it should use by doing the following:

* Click the drop-down menu named "Select new compiler version" and scroll down to find a version less than 0.4.19, such as 0.4.17+commit.bdeb9e52 and select it.

This will tell Remix to re-compile with that version and your errors should go away and you Remix screen should look like this:

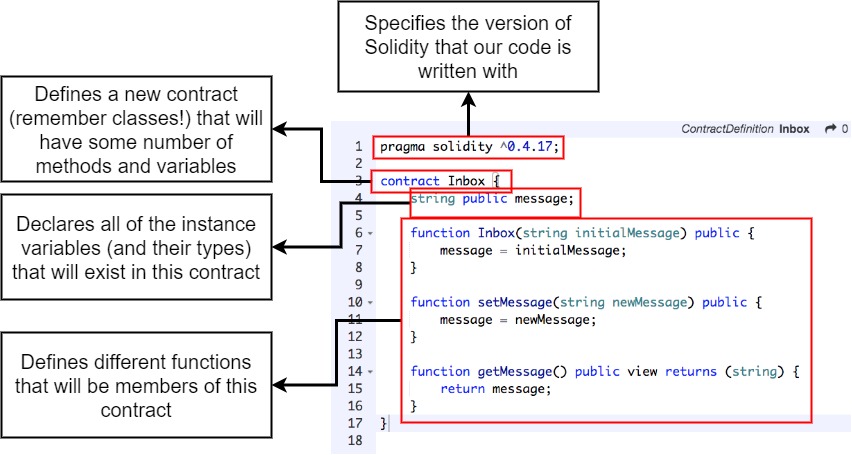


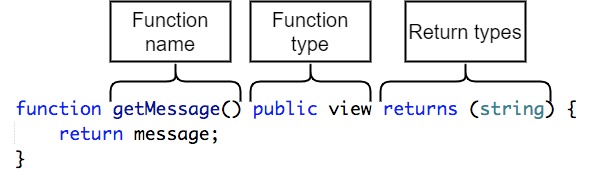
So that's our first little smart contract. But of course, we just wrote a lot of code without having any idea of what's going on. In the next section, we will tear contract apart and understand every last line of code inside of here.

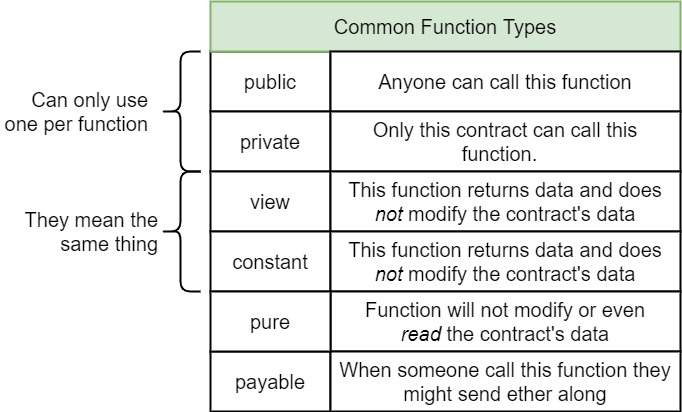
I took all this code I put it into a diagram, and I labeled a lot of the different lines with some different notes.

**Solidity Programming Language**

We're going to use a programming language called Solidity to write the actual code that gets inserted into a smart contract. In the above section, we wrote our first little contract without any idea. In the following diagram, I took all this code and labeled a lot of the different lines with some different notes.







This completes a first overview of our inbox contract but we still need to use the remix tool quite a bit to figure out how we can run deploy and test out this contract which we will discuss on next section.

**Testing With Remix:**

Now that we have a better idea of the code that's running inside of our contract. We're going to use the remix tool to test our contract.

**Understanding:**

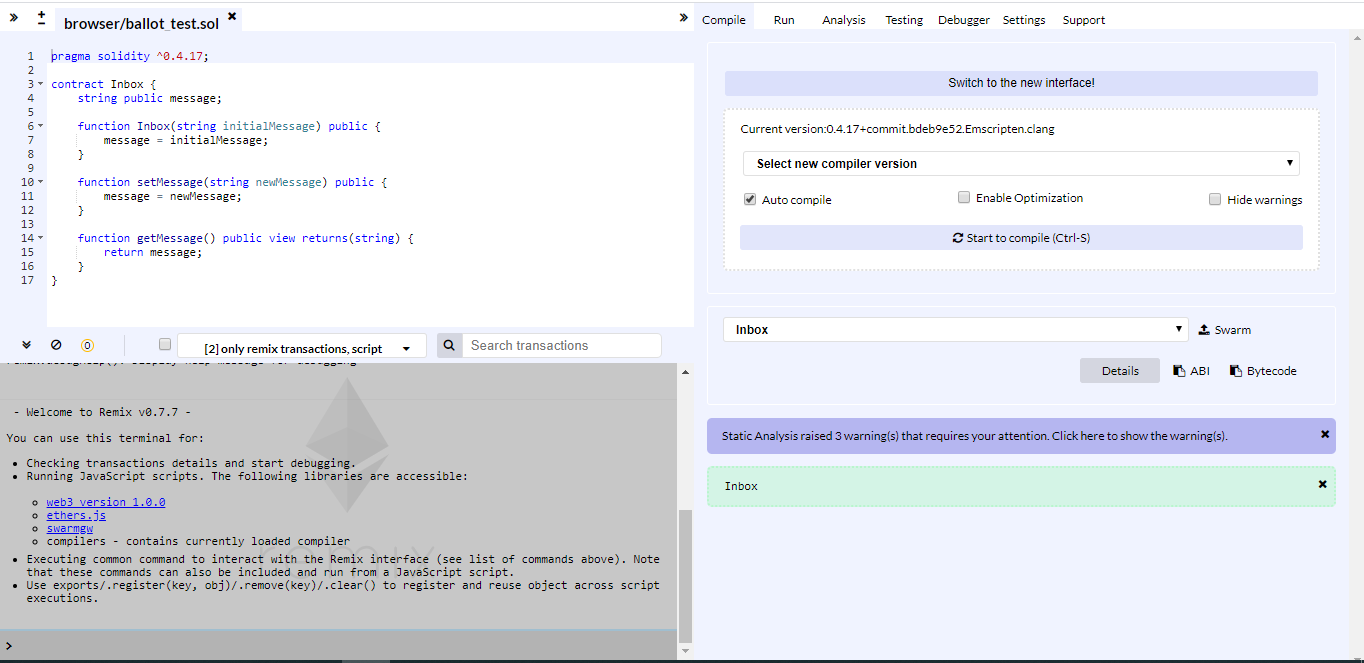
We'll first start off by taking a look at a diagram that will give you a better sense of what remix is going to do for us.

A screenshot of a cell phone

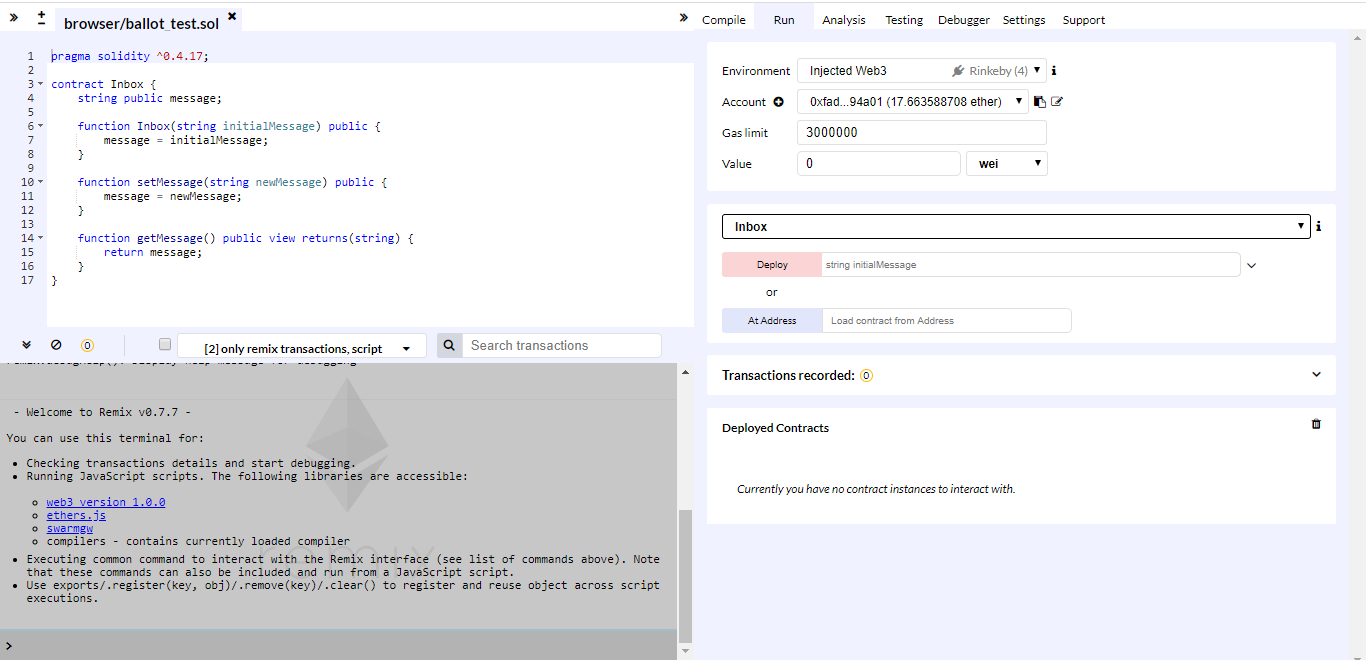
Description automatically generated

So, at this point we're sitting down inside of the remix editor and we have some amount of contract source. We're about to have this contract to be automatically compiled by the solidity compiler. And that's going to compile it to bytecode that will then be deployed to an in-browser fake theory network and that will give us an instance of our inbox contract. This part right here might sound a little surprising in browser fake network. Well behind the scenes remix is not just a code editor. It also hosts a miniature tiny little fake theory network that we can use to simulate deploying and interacting with our contract.

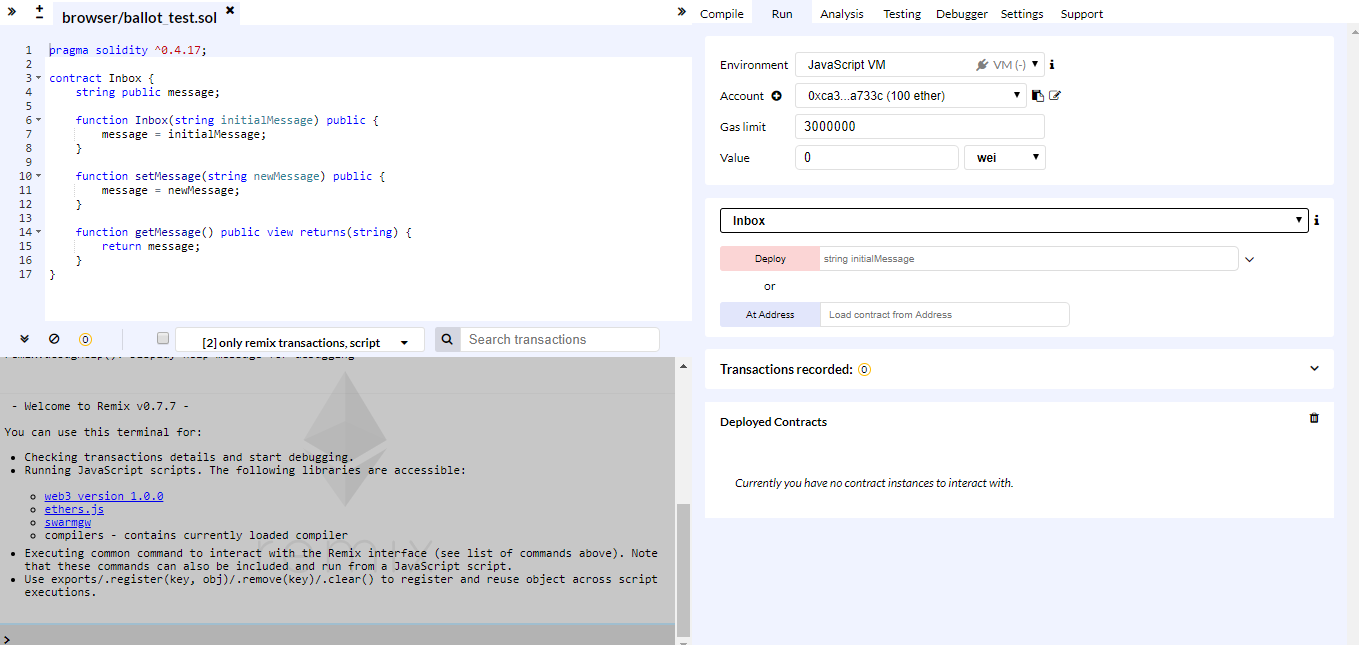
* You can setup your screen like below with collapsing the left tab and adjusting the size of other tabs which will make you easy to see changes during contract deployment:



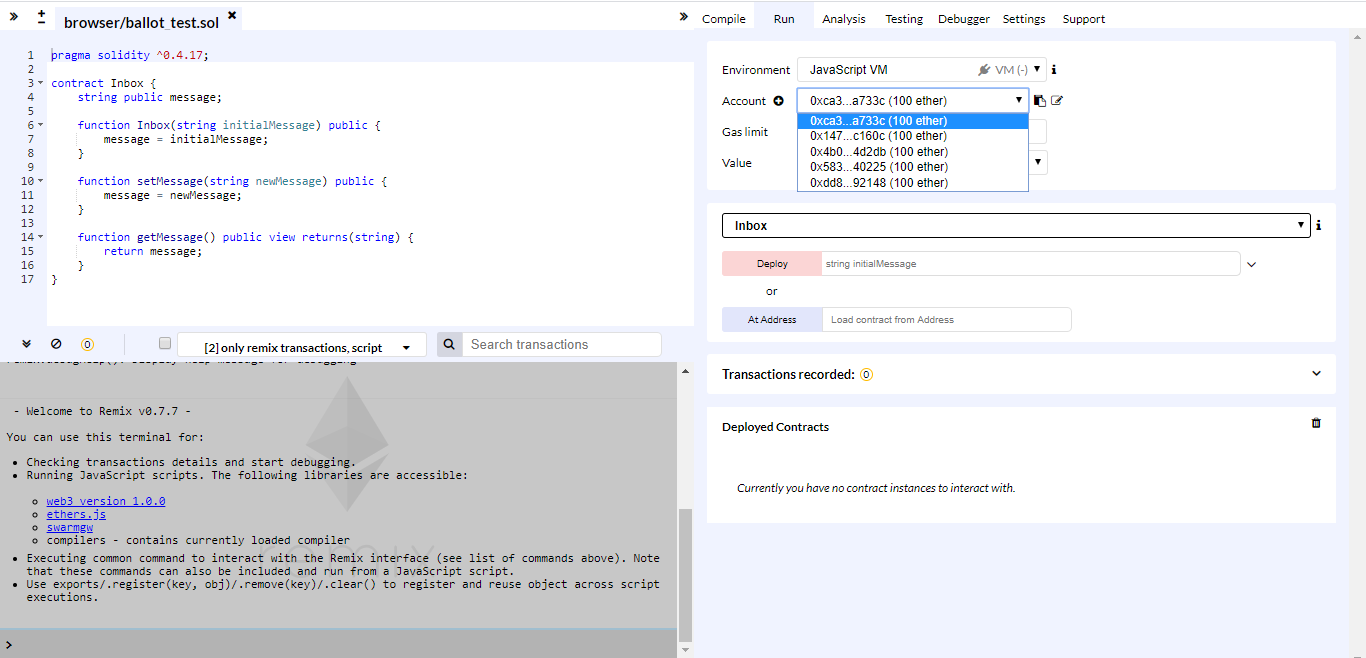
* Find the top right-hand tab marked as Run.



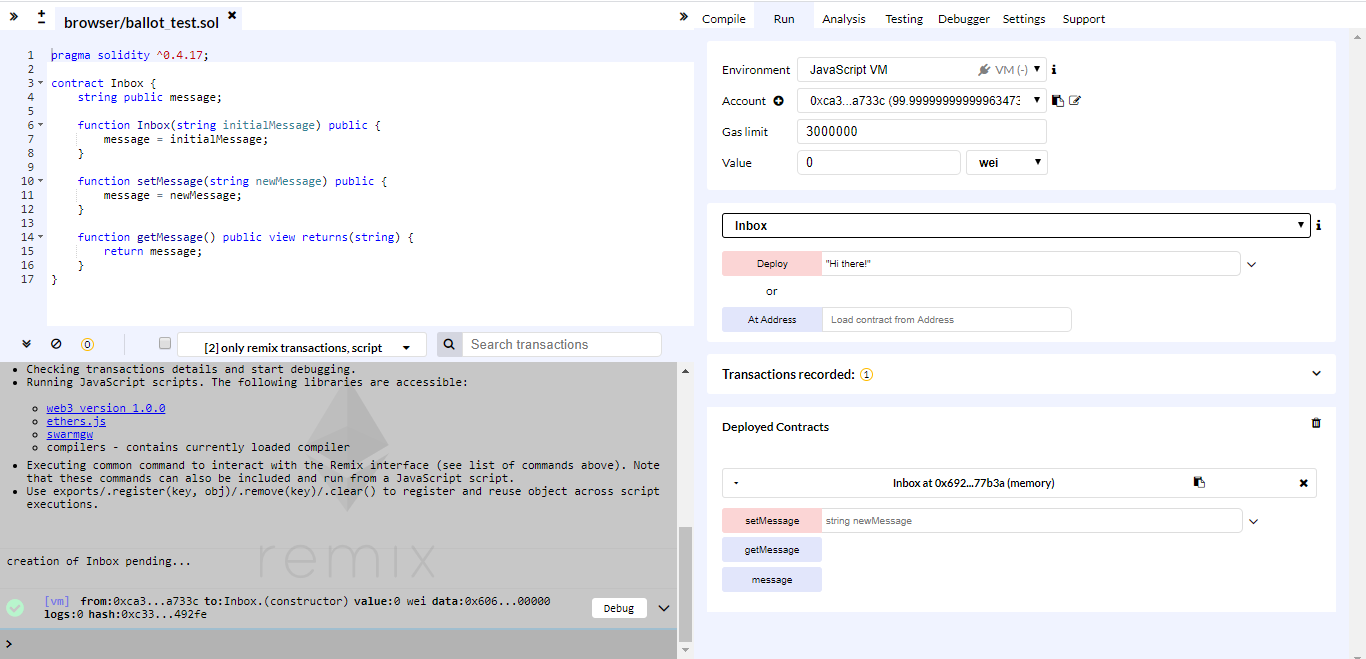
* On the Environment Setting, you will see Injected Web3 as selected. We do not want to use Injected Web3 instead use JavaScript VM. Select JavaScript VM using dropdown menu. This tells the remix tool that we want to attempt to deploy our contract to that in-browser virtual theory network.



* Next you'll see a listing that says Account. When you select the JavaScript VM that boots up the internal virtual network and then it automatically creates a handful of different accounts each of which are assigned 100 either to work with. So, these are all accounts that exist only inside of our browser right now and they have been prefunded with 100 either. You can see all accounts by clicking the dropdown menu.

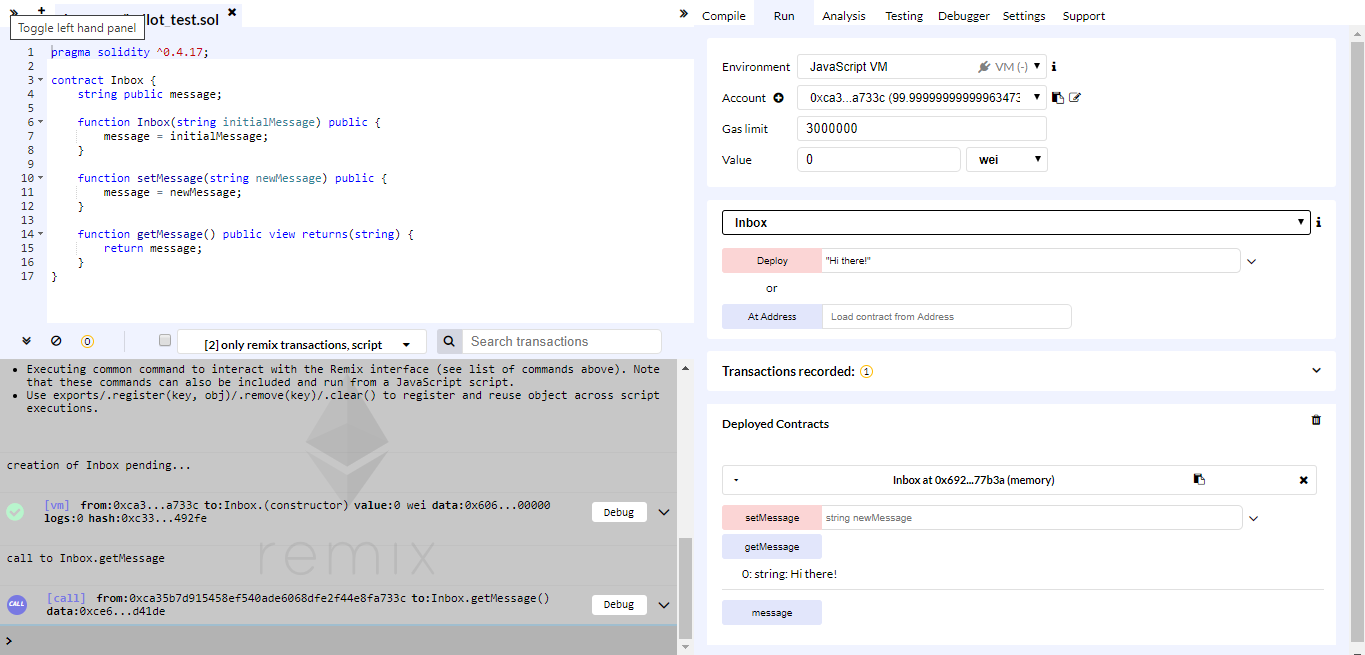


* Next you will see Gas Limit. It refers to the maximum amount of gas you are willing to spend on a particular transaction. Next you will see value. Gas value refers to the fee, or pricing value, required to successfully conduct a transaction or execute a contract on the Ethereum blockchain platform. We will discuss later on the course about these two things.
* Next you will see a drop down and has already selected Inbox. This is selecting a contract that we want to attempt to deploy to our local test network that’s running inside the browser.
* Underneath that you'll see something that says string initial message and to the left it have “Deploy” button. In our smart contract code, we have a function that has the same exact name as our contract. And because it had the same name, it refer to as a constructor function and constructor functions are automatically called whenever we first deploy or create an instance of a contract. Our constructor function says that accepts a string argument that is called initial message. So, this is how string initial message also appears over here in this input.
* So, we're going to do is enter in some string then click Create. When we do that an instance of our inbox contract will be automatically created the constructor function will be called and this constructor function will be called with an initial message of whatever we place into this input right here.
* Type “Hi there!”. Make sure you get double quotes too and Click Create. Next you will see a little bit of activity on console.



Down there at the bottom right, there is now a new entry this represents a contract that we just deployed to our local test fake network. So, this is an instance of the inbox contract which lists out all the different functions. These are essentially all the different ways that we can interact with the contract. These are buttons and if we click any of them it will automatically invoke the corresponding function

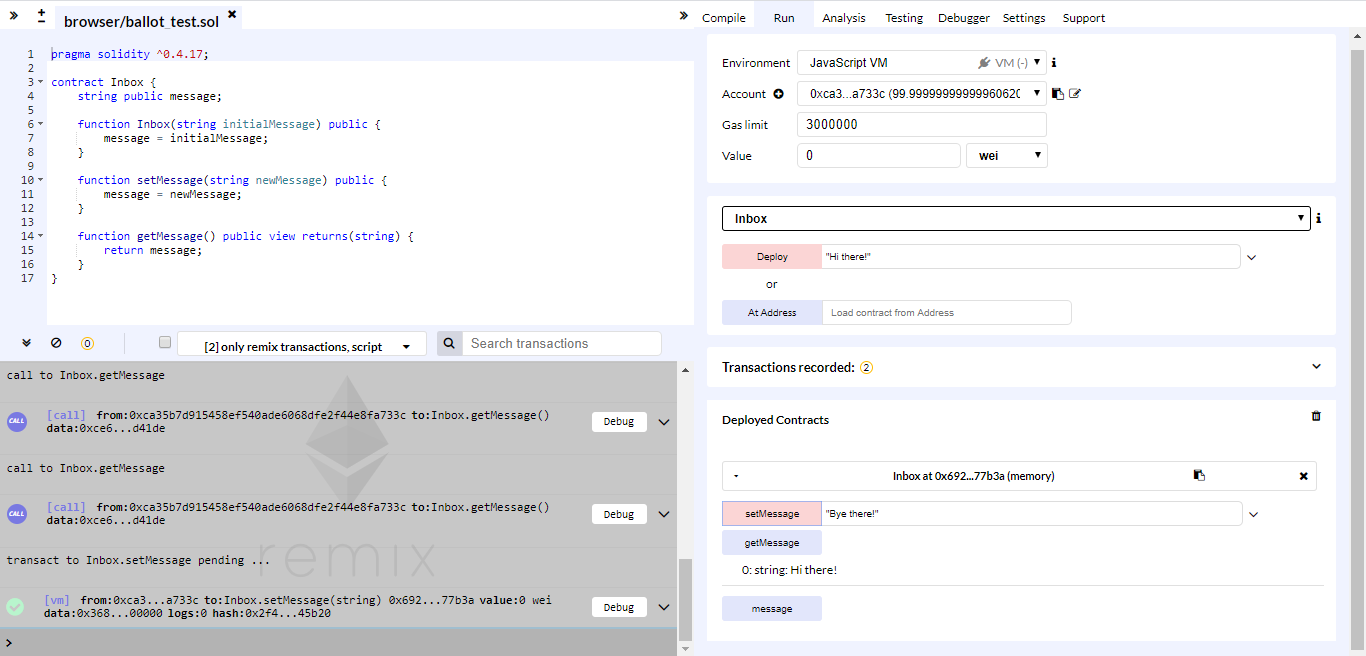
* Click the Get Message and you will see our initial message at output.



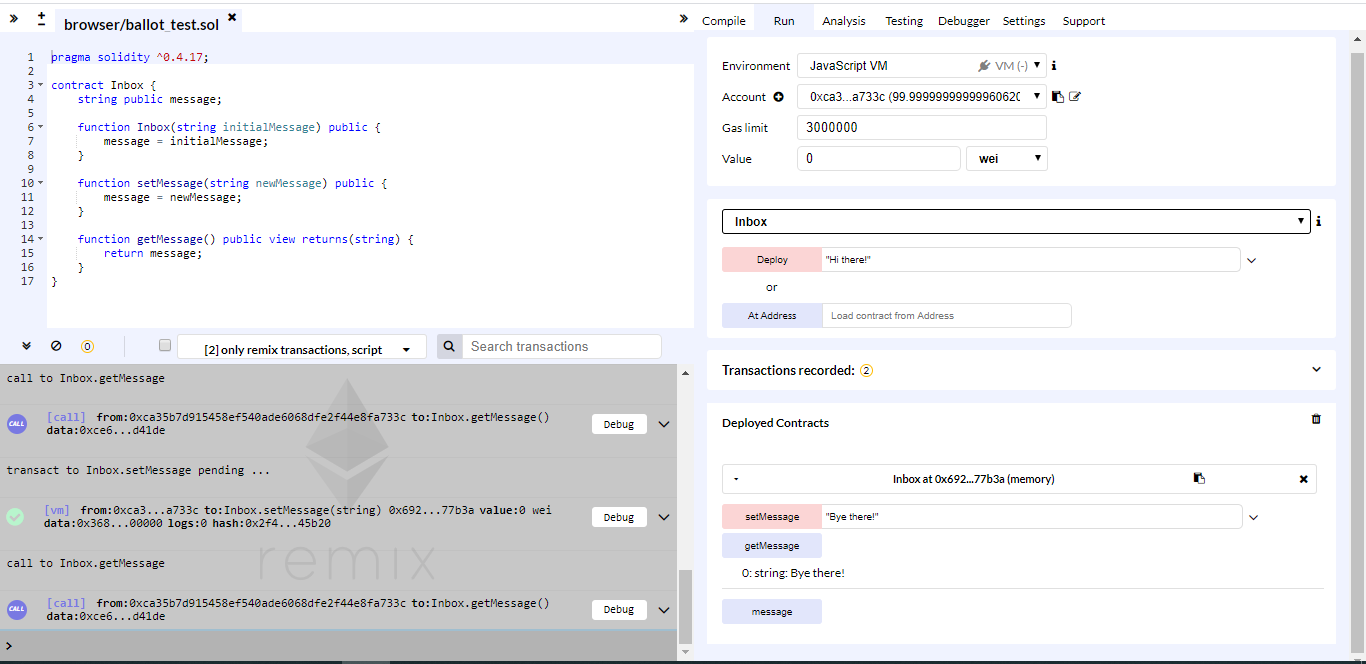
It says that we got back “0:string:Hi there!” 0 means that this is the first return value from the function. In this case are functionally returns one value and so this is the first of just the one value that gets returned. If we had returned multiple values from get message then we would see zero value and then one and the next value that got returned after the 0 we get the type of data that got returned. In this case our message is always a string. This is telling us we got back a value of type string. And then finally it tells us the actual value that was returned which in this case is high there.

Let's try using the setMessage button down here as well so we can enter in a new message as a string.

* Type “Bye there!”. Make sure you get double quotes too and Click setMessage. That's going to send a function call to our deployed contract. You can see that inside of our console over here and that invokes the setMessage function within our inbox contract.



Notice how when we called setMessage that updated the value of the internal message variable. But it did not automatically update the value of “Hi there!” that we had just gotten back from call to getMessage. So, these values will not automatically update. Instead we have to click getMessage again to attempt to retrieve that value again.



So, here is a diagram of our contract as it is deployed currently on the test network inside the contracts storage. We have that message variable and it can be modified or accessed by either the getMessage or setMessage functions in either of those can be called by anyone on the network. So, essentially our contract right now is just storing a little message that someone else can read.

A screenshot of a cell phone

Description automatically generated

To verify that it can be modified or accessed by anyone on the network, you can perform the same operations as getMessage and setMessage with different accounts listed on the account tab.