Exercise: Create a Simple Smart Contract

February 5, 2020

Abstract

The following is an introduction to the Remix software development environment followed by a task to code your own smart contract in Solidity that stores data and has multiple types of functions.

1 Software Development Environment Introduction

For the next few weeks, well be using an online tool, Remix[1], to write and test our smart contracts. It lets us develop Solidity code and compile it with only a browser interface. Smart contract execution in Remix is performed in one of three environments:

- *Javascript VM:* makes the website act like it is connected to a blockchain node.
- *Injected Web3:* allows you to connect to the DLT node defined in the browser, possibly via a plugin (e.g. MetaMask[2]).
- *Web3 Provider:* allows you to explicitly give the address of a DLT node to connect to. This can be either a node of a live permissionless DLT network, a node of a permissioned DLT network, or even a test node that simulates a DLT network (e.g. Ganache[3]).

Note that the Javascript VM environment is certainly the easiest to get started with as it requires no additional configuration, it is free to use, and it is faster than interacting with a DLT node. Therefore we will be using that environment for this week's activity.

[1] https://remix.ethereum.org
[2] https://metamask.io/
[3] https://www.trufflesuite.com/ganache

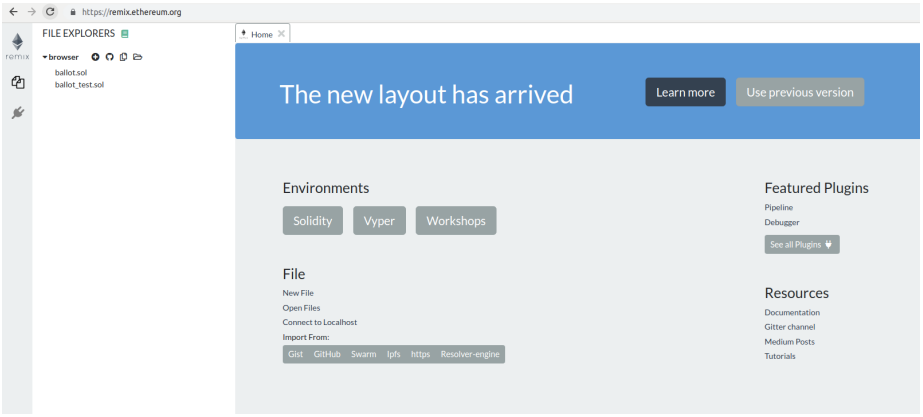

Figure 1: Opening Remix for the first time

Starting with Remix: The new Remix layout expands via plugins. The first time Remix is opened, you should only have a limited number of Plugins, such as the File Explorer and Debugger (see Figure 1's lefthand bar for an example). In the File Explorer you will see some example contracts. In the figure, you will see the example simple ballot smart contract and it's testing version contract. Feel free to explore the example contracts in your own time.

Preparing Remix: Now for our activity, you will certainly need the 'Solidity Compiler' plugin and the 'Deploy and Run Transactions' plugin. To add it (or to make sure they have been added), click on the Plugin Manager icon on the left hand column, browse to the previously named plugins and activate them. You will see their icons appear on the left hand column.

2 Coding a Smart Contract

Background: Now to introduce you to developing smart contracts and using the Remix environment, we will run through a task which uses different function types (e.g. public, private, internal, external - recall that the first three can be applied to globally defined contract variables as well), different storage types (e.g. storage, memory, call data), different contract state options (e.g. pure, view) and different smart contract interaction methods[4] (e.g. contract creation, contract state query, contract state change and contract event listenings).

Activity Introduction:

Next we can start coding our new contract. Select the File Explorer plugin and click on the 'create a new file' icon in the explorer (shaped like a cross).

[4]See lecture notes for a discussion on the different interaction methods. In this task we are focusing on interacting with a single smart contract.

Name this solidity file Storage.sol.

Smart Contract Task: Your task for this activity is to take the smart contract template code and complete it by filling in the sections marked with square brackets, i.e.: [...]. This storage smart contract has been designed to: (a) track a number (storedInteger); (b) track the last address (whoChangedTheStoredInteger) who changed this number; (c) record the address who created the contract (contractCreator); (d) allow any address to check the previous 3 variables; (d) provide any address with a few different functions to change the tracked number.

To complete the functions correctly, follow these instructions:

1. Code the constructor function to initially set all 3 global (storage) variables. The contractCreator variable should be set to the address who created this smart contract.

2. Code a simple contract state query function, named getStoredAddresses, which will allow anyone or any function to read both of the stored addresses at the same time (i.e. contractCreator and whoChangedTheStoredInteger). Note for this example contract, this should be the **only** way that an externally owned address can access the two stored addresses.

3. Code a contract state change function, named setStoredInteger, which will allow anyone or any function to call it. This function will do the following:

   (a) has an integer as input

   (b) returns a boolean

   (c) if the inputted integer is a certain number (you choose), then the function will return false. Make sure you choose a number that is not a prime number.

   (d) otherwise, you should change the storedInteger and whoChangedTheStoredInteger variables and emit the storageChanged event.

4. Code another function, named letsMultiple, which will allow only this contract or inheritors of this contract to call it. This contract will accept a single integer and returns the integer multipled by another. Make sure that it is possible for this function to multiply to the number you choose for 3 part (c).

5. Code another contract state change function, named setStoredIntegerWithMultiplication, which will allow any address to call it but not functions of this contract. This function will do the following:

   (a) has an integer as input

   (b) requires a payment

   (c) returns nothing

(d) only performs the following computation if the sent amount of cryptocurrency is over a certain amount (you decide)

(e) takes the inputted variable and uses the letsMultiple function to multiple it

(f) stores the multipled integer by calling the setStoredInteger function

(g) if the setStoredFunction returns true, then the multiplicationStored event is emitted with a way to show it has been stored, otherwise the multiplicationStored event is emitted in a way to show it has **not** been stored.

Some hints regarding the task:

- For parts 2, 3, 4, 5 and 6 remember to choose the correct function type (i.e. public, private, internal or external?) and state modification option (i.e. pure, view or none?)
- In this contract, we are dealing with basic value types (i.e. integers, booleans and addresses). Therefore we do not have to explicitly detail the data location (i.e. storage, memory or call data). But implicitly the contract global variables go into storage, while variables declared in the functions are memory variables.
- There are some variables that every Solidity contract can access[5]. To access the sender of a message, use msg.sender. An alternative is tx.origin, which gives the original sender of the message (there can be a chain of message senders), but tx.origin is more likely to introduce errors[6]. Additionally, msg.value returns the amount of wei cryptocurrency sent by the transaction. Whereas wei and ether are reserved key words for Ethereum's native cryptocurrency (where 1 ether = 1,000,000,000 wei)[7].

Compiling your smart contract: After a certain amount of coding you will be ready to use the Solidity Compiler plugin, which you will have previously installed in the *Starting with Remix* section. So, select that plugin in the side menu, make sure you have selected the correct contract and click on the compile button. Now either you will see a green tick on the plugin icon indicating that the compilation has completed successfully (see Figure 2 for an example) or you will see a red box and details about the remaining errors. If you have any errors, iterate the process until the your code compiles.

Running your smart contract: Now move to the Deploy and Run Transactions plugin, where you will see something like Figure 3. Using this plugin we will simulate deploying your contract onto a blockchain (recall that in the Remix javascript environment we do not actually connect to any blockchain).

[5] See here for the full list: https://solidity.readthedocs.io/en/v0.5.11/units-and-global-variables.html

[6] See: https://consensys.github.io/smart-contract-best-practices/recommendations/

[7] Again, see the following for more information: https://solidity.readthedocs.io/en/v0.5.11/units-and-global-variables.html

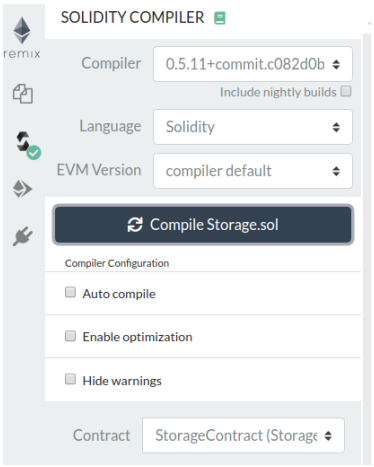

Figure 2: The Remix Solidity Compiler Plugin

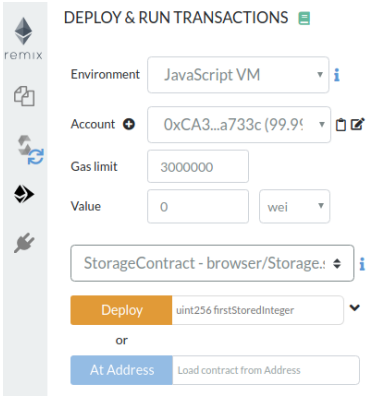

Figure 3: The Remix Run and Deploy Transactions Plugin

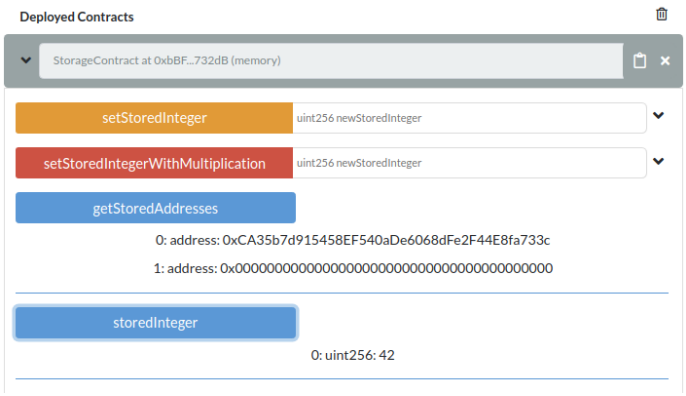

Figure 4: A deployed contract on Remix

To briefly describe Figure 3, the environment selected is the Javascript one for reasons discussed previously. The Account tab provides us with some example blockchain accounts that have been prefunded with cryptocurrency. The Gas limit tab allows us to set the computational gas limit for every block on our simulated blockchain. The value tab lets us send a transaction with an associated cryptocurrency value. Next we have our contract selection box - make sure to select the contract we have been working on!

Then we can choose to either use the Deploy button to deploy a new version of this contract or use the At Address button to connect to a previously deployed version of the contract. As this is our first deployment of the contract, we will be using the Deploy button. Presuming you coded the constructor with one variable as input, we will have to instantiate that variable in the text field next to the deploy button.

Once the contract is deployed, you will see something like Figure 4. The blue boxes are contract state query functions, the orange and red are contract state change functions, where the red function requires payment of cryptocurrency.

Now we will use another account to set a new stored integer. Select another account from the Accounts tab on the top left of the page. Then at the setStoredInteger function enter a number and click on the button. If you have coded it correctly, and not entered an integer value that would have returned false for that function (according to 3 part (c)), then you will have changed the state of the contract. Re-click on both contract state query functions to check that you have updated the contract state. You should see something like Figure 5. Take some time to inspect the transaction and call information that is appearing in the bottom right hand corner of Remix, see Figure 6 for an example. Transactions will have an associated green tick (to indicate their associated computation completed) or a red cross (to indicate that their associated computation errored and reverted). If you click on the downwards facing arrow you will find out more information, including the log section that will denote any associated events that fired.

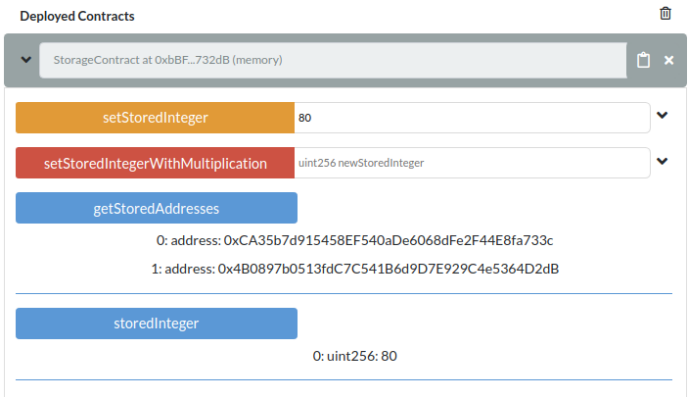

Figure 5: Changing the stored integer and recording the account who made the change

Figure 6: An example transaction log

The final part of the activity is to set a new integer using the multiplication function. To do so, first we need to enter a value of cryptocurrency over the limit you previously set in 5 part (d). Therefore, go to the value tab in the top left and enter the correct amount. Afterwards, browse to the setIntegerWithMultiplication function and enter the integer you want to be multiplied and then stored. Click on the function button and check that the integer was saved with the multiplication by clicking on the storedInteger function button. You should see something like Figure 7. Also inspect the transaction log and see that both of your events should now have been fired. You should also try and call this function when you enter a number that will lead to the setStoredInteger button returning false. Explore the differences in the transaction logs. This part of the activity shows that when a function (setIntegerWithMultiplication in this case) calls another contract state change function (setStoredInteger), it can get an immediate response, unlike when a contract state change function is called direct from a transaction.

You have now completed your first Solidity smart contract, deployed in on a virtual machine and interacted with it. Congratulations!

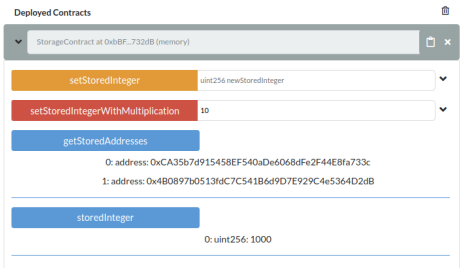

Figure 7: Changing the stored integer with a multiplication. My contract used a 100x multiplication.