# Exercises: First Steps in Solidity

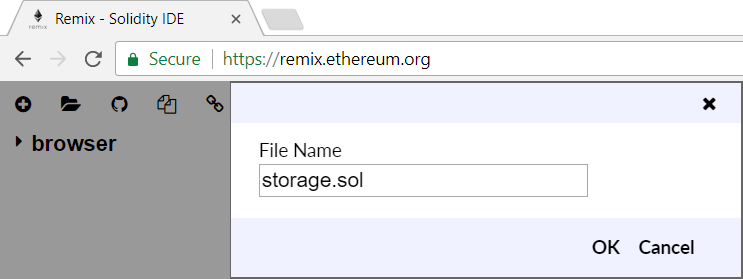
This document describes the **exercise assignments** for the ["Blockchain Academy" course @ Software University](https://softuni.bg/courses/programming-fundamentals). In this lesson we learned the **basics of Solidity** programming language. The goal of this exercise is to get practical skills in writing simple smart contracts in Solidity, publishing and testing contracts in the Remix IDE.

## Simple Storage Contract

Write a simple contract in Solidity that keeps in the blockchain an **integer variable** and provides public functions to **read** it and to **change** it. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

### Hints

Open the **Remix IDE** (<https://remix.ethereum.org>) and create a new empty Solidity source code file:



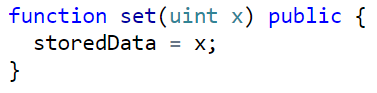
Write the source code of the contract, step by step. First, write the **contract definition**:



Next, in the contract body define the **integer data storage field** named storedData. Use an integer type of your choice, e.g. uint (256-bit unsigned integer). Your data field is just like a member field in a class, but it is **b** on the blockchain. While this contract stays alive on the blockchain, this field value also will stay with it. You may choose public or private visibility. Public fields can be read by anyone, while private fields can be read by the contract code only. Your code might look like this:

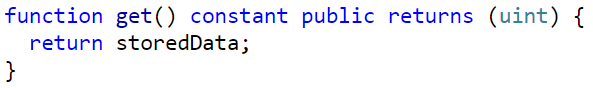


Next, write a function to **assign a value** in the data storage field:



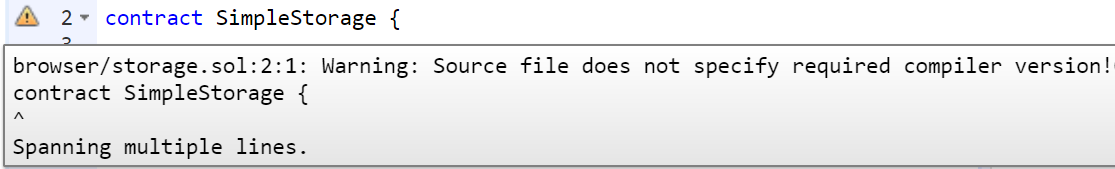
The function takes a value x as input and stores it in the data field storedData. The function visibility is declared public, which means that the function can be called by anyone. In Solidity you don’t write this.storedData like in other object-oriented languages.

Next, write a function to **read the current value** from the data storage field on the blockchain:

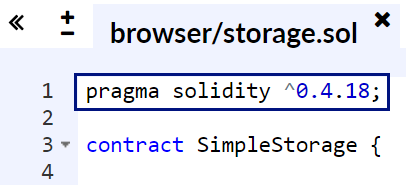


This function takes no parameters and returns a result of type uint. Its visibility is declared public. The function is also declared constant, which means that it does not change the contract’s internal state. If you don’t declare the function as constant, the compiler will issue a warning.

Now your code is almost ready, but the Solidity compiler still issues a **warning**:

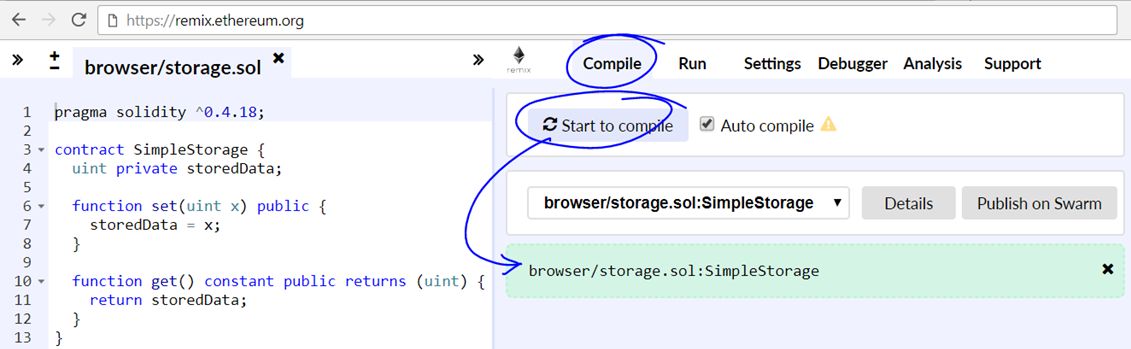


To fix this, you may **add a compiler version** definition at the start of the contract:

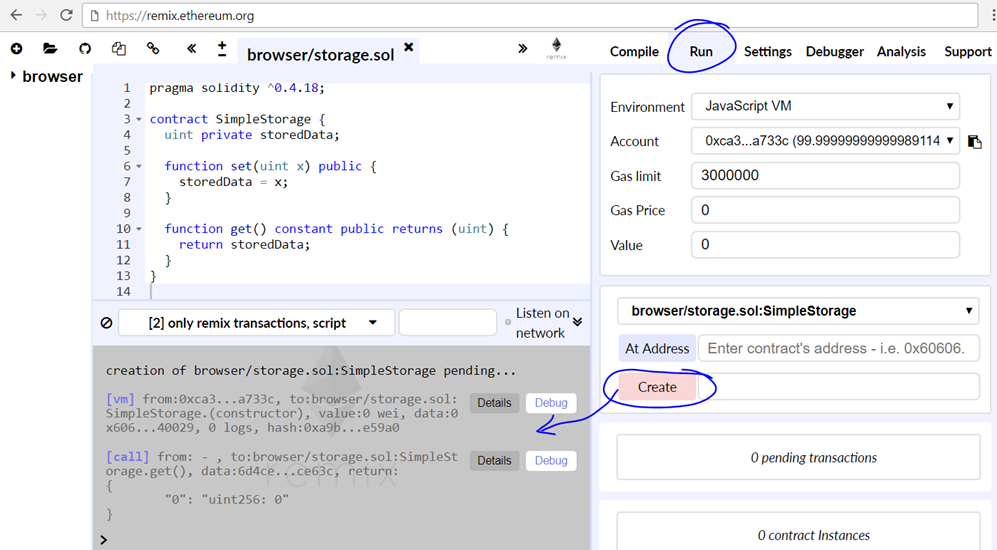


The above pragma definition says that this contract should be compiled by Solidity **compiler version 0.4.\*** (later than 0.4.18 and earlier than 0.5).

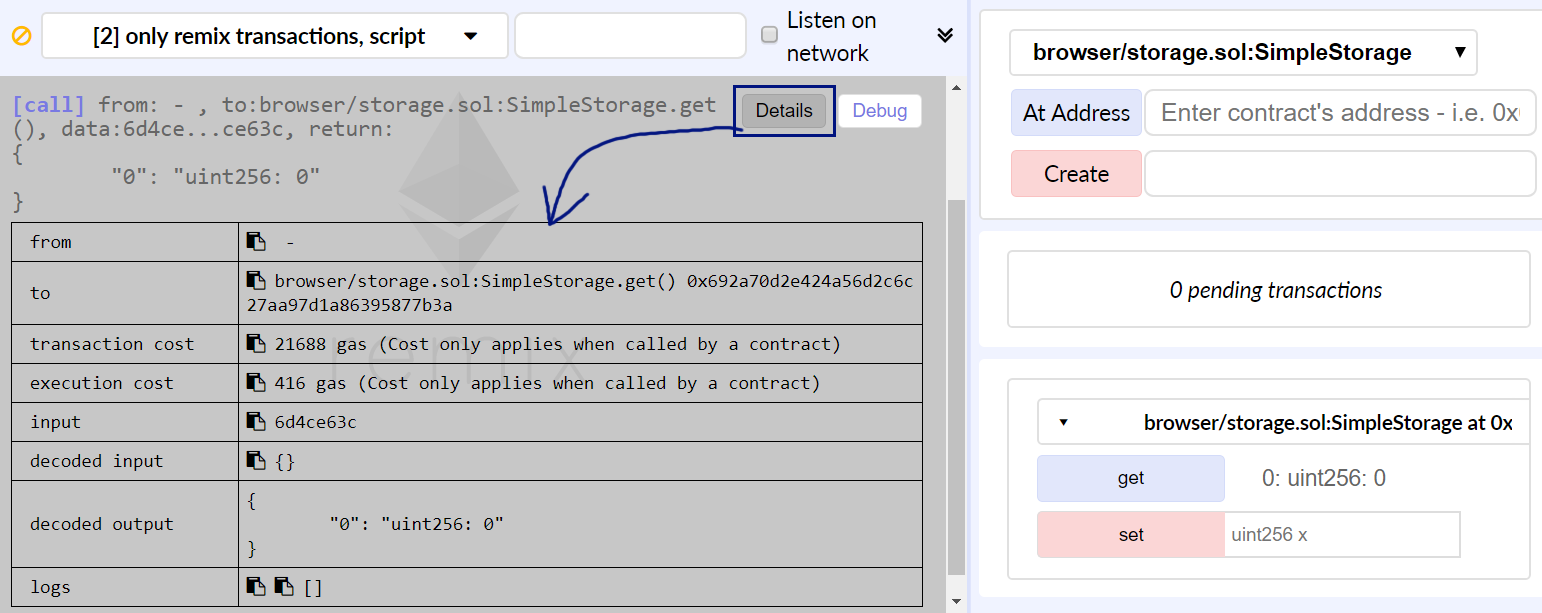
Now you are ready to **compile your code**. The Remix IDE has built-in compiler, which is by default in “auto-compile” mode. Activate the **[Compile] tab** and see the compilation results:



The next step is to **publish the contract** on the local in-browser blockchain environment. Open the **[Run] tab** in the Remix IDE. Click on the **[Create] button**. It will deploy the contract in the local JavaScript in-browser blockchain testing Ethereum network. As a result of the publish operation, the contract will be created, and its address will be returned from the blockchain network.

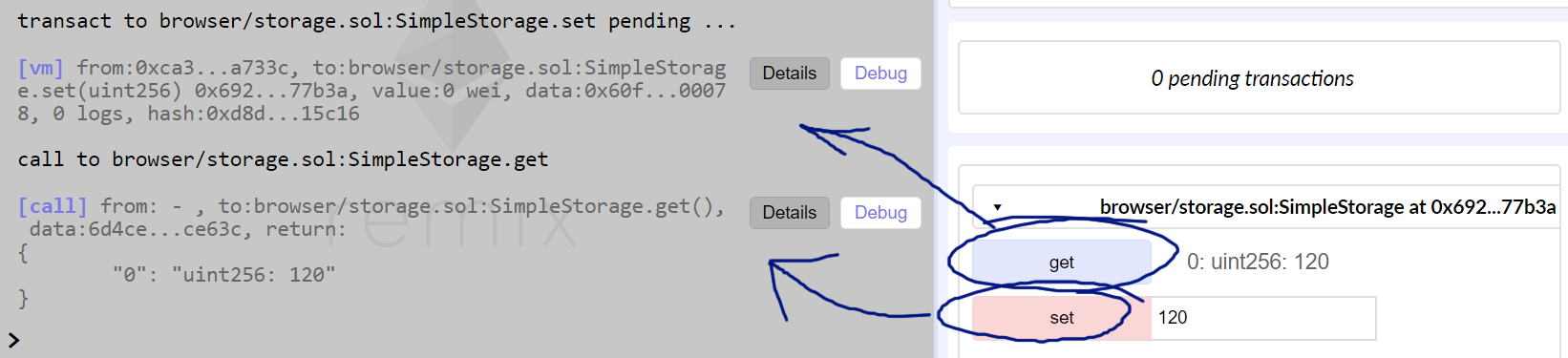


The **[Details] button** in the execution logs provides additional technical information about the transaction that caused the contract to become published:



Now the contract is published. The next step is to **test its behavior**. In Remix you can invoke published contract’s public members (call methods / read public fields) form the UI at the right-down side of the screen.

Try to change the contract data through the **[set] button**. It will invoke contract’s set(x) operation. Also test the get() operation the same way. Assign some value in the contract, then read the value back.



Voila! You successfully developed, published and tested your first smart contract.

## Incrementor Contract

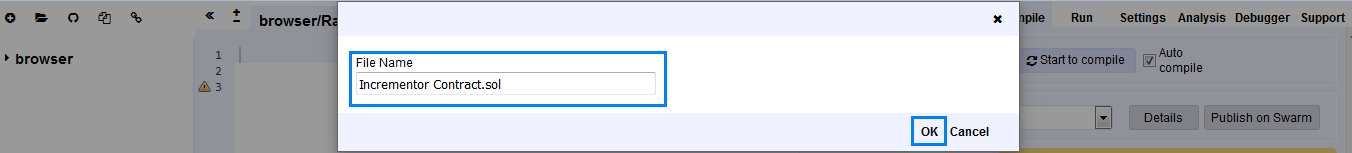
Write a simple contract in Solidity that keeps in the blockchain an **integer variable** and a public function that **incrementing** it. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

### Hints

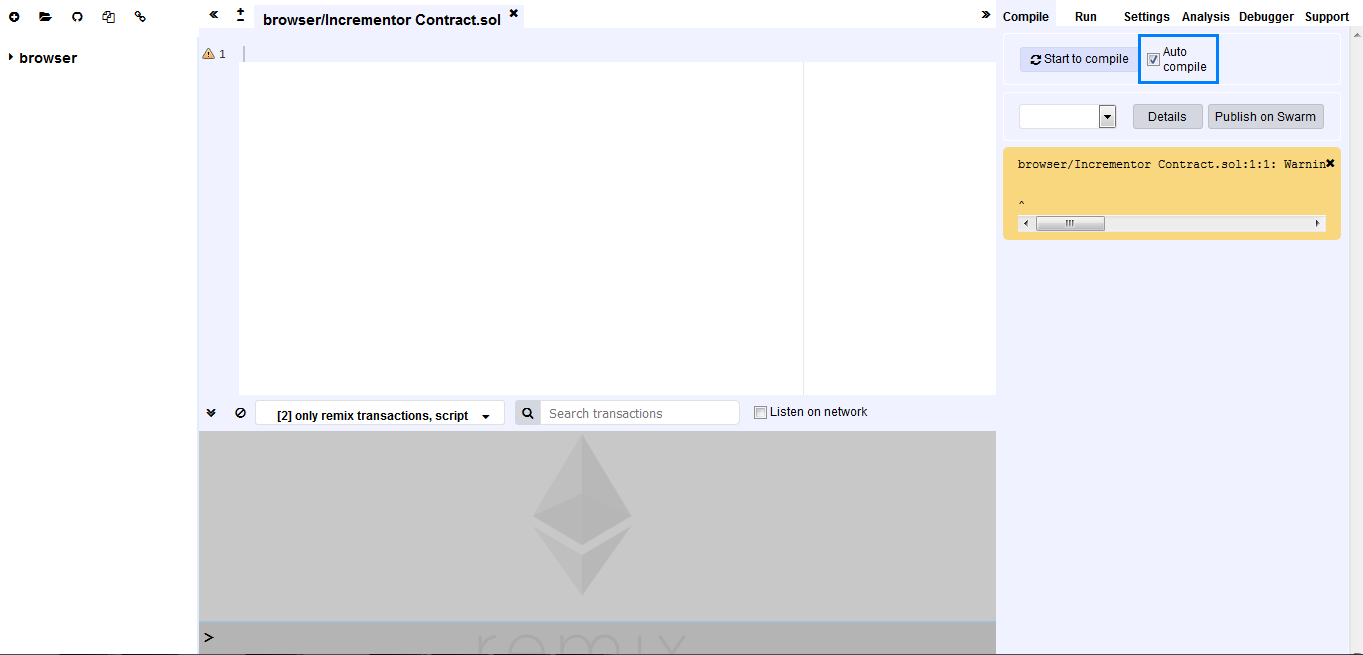
Open the **Remix IDE** (<https://remix.ethereum.org>) and create a new empty Solidity source code file (the icon for this is in the upper-left corner):

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\New Solidity File 1.png

A new window should open. In it you must write a **name** for your file (in this exercise we will be using the name “Incrementor Contract”). After you have chosen a name click **[OK].**

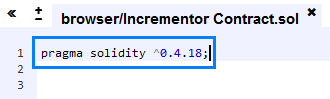


Your browser should look very similar to the following screenshot:

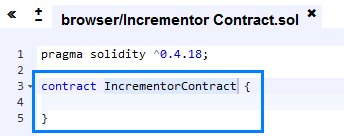


Make sure that “Auto compile” is checked so as to make testing seamless. Now we are set!

First, we select which version of the complier we will be using:



For the next step we use the key word “**contract”** to give a name to it (you are free to choose whichever name you prefer. For this exercise we chose the name “IncrementorContract” ).



Like in the previous exercise, everything we write from now on will be between these two curly brackets.

Next, we initialize an unassigned integer variable with name **“valueToBeIncremented”** setting its visibility to **private** (the variable will not be accessible from outside the contract).



We will be **getting** and **incrementing** the value of this variable. Now is the time to ask the question might happen if we **get()** the value of this variable, since we have not assigned anything to it. So, let’s test it by writing the **public** function **get()** that returns the value of the variable.

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Function Get().PNG

We define a function with the key word **“function”** and then give it a name. In this instance the name is “**get()”**.

Next, we chose whether the function will be accessible from outside the contract. In our case we want this to be so and achieve this by the key word **“public”**. Afterwards, we must promise that this function will not change the state of the contract (since this function only gets a value). This we do with the key word **“view”** – this means we just want to look at something. Finally, we define what type of variable this function will be returning – we need it to be of type unassigned integer, hence we finish our function declaration with **“returns (uint)”** (note that the key word is **returns**, not a simple **return­­**).

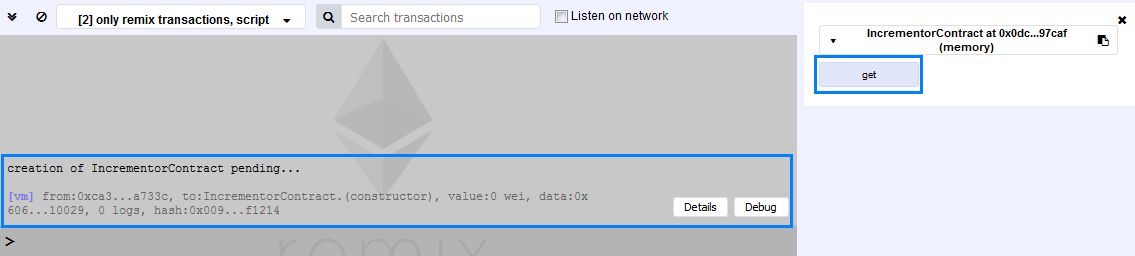
Since we only want to see the current value of our variable we write the following line of code between the curly brackets of our **get()** function:

**C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Return statement.PNG**

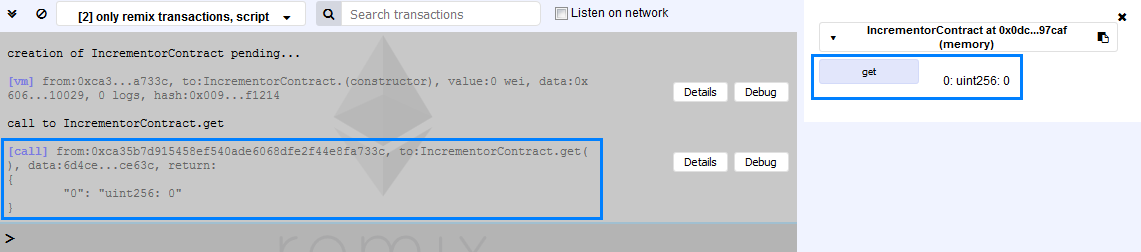
Let’s test what we have done until now. In order to do that we must select the **run** tab and deploy out contract through the **[Create] button**. Your code should be as shown in the screenshot. In addition, check that the selected **environment** is **JavaScript VM**.



After clicking the **[Create] button** some information will be displayed in the gray box in the bottom of your screens. Furthermore, in the bottom-right corner a **[get] button** should have appeared.



When you click the **[get] button** the value that is return should be equal to **zero**.



From this you should remember that when an unassigned integer variable is initialized without assigning any value to it, its default value is **zero**.

Now that we have implemented the **return** function successfully it is high time to write the function that will **increment** the value of our variable.

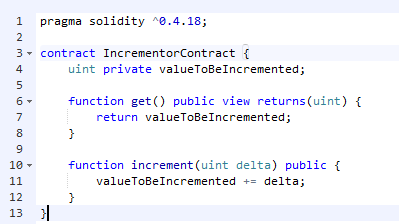
We will call our function **increment**. It must be passed a value of type **uint** with which we want to increment the value of our variable. Finally, the function should be accessible from the world that is outside our contract. All this leads to the following line of code:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Function increment(uint value).PNG

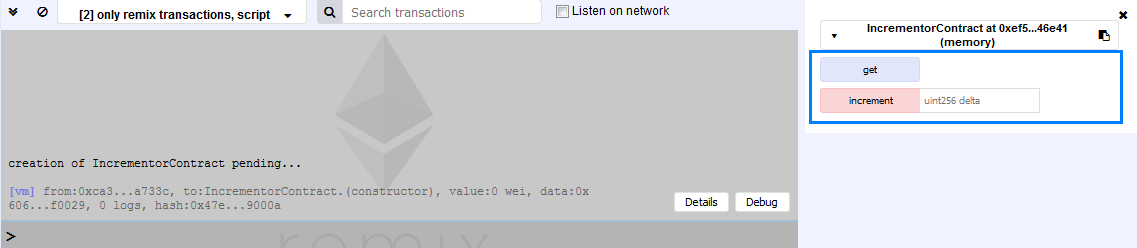
For our last step we simply need to add the value that is passed to the **increment** function to the value of the variable **valueToBeIncremented**. This is achieved through:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Add values.PNG

After going through all the above steps your code should look like the following screenshot:

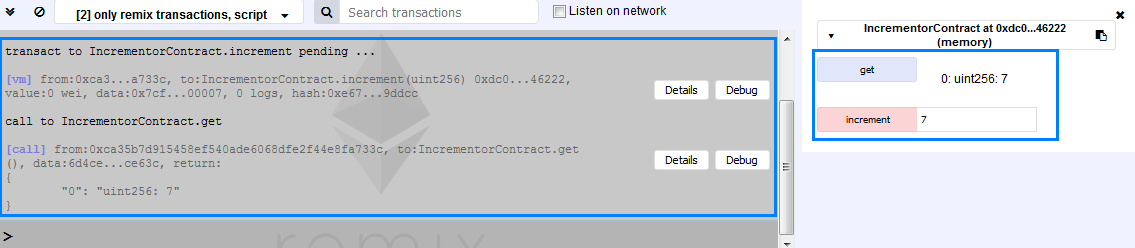


Now, let’s test it! So, by following the steps that were outlined a few pages back we create our contract. Now you must see **two** buttons in the bottom-right corner of your screens, very similar to the following screenshot:

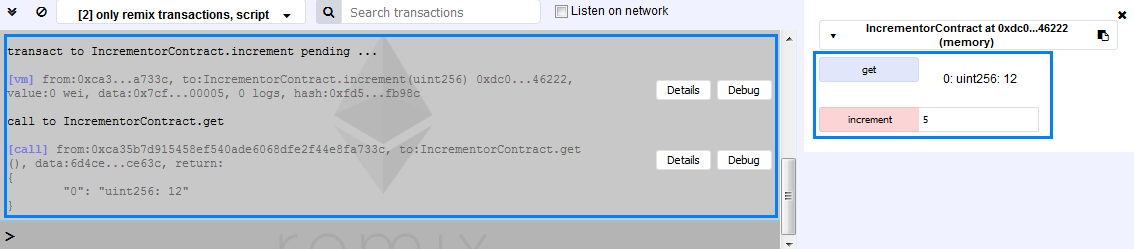


Next to the **[increment] button** you should be seeing a field with a placeholder that describes the type of value that is expected (uint) and by what name the code will be using it (delta) – all this information is part of the function **increment**.

Previously we have seen what happens if we click on the **[get] button** – the function returns zero. Now let’s test the increment function by passing it the value **7,** clicking the **[increment] button** and then clicking the **[get] button**:



From the left-hand side of this screenshot it is visible that first the **increment** function is called and afterwards the **get** function. Basically, you should have gotten the value that you have passed to the **increment** function. Let pass to this function another positive integer value (remember, because the type of the expected variable is uint a negative number will throw and error. In addition, if you pass to the function a **positive** floating-point number it will be truncated and only the integer part will be used). In our example we shall use **5**. We change the value in the field from 7 to 5, click the **[increment] button** and then click the **[get] button**. As a result, we get is the following:



It is again visible from the information in the gray box on the left that the first call is to the function **increment** and the second call is to the function **get**. The returned value is equal to 12 (the first passed value was 7 so 7 + 5 = 12).

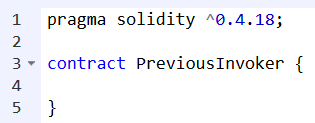
Congratulations! You have successfully completed this part of the exercises.

## Previous Invoker

Write a simple contract in Solidity that keeps the address of the previous invoker of the contract. Use the [Remix IDE](https://remix.ethereum.org/) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

### Hints

Create a new empty Solidity file with the name **Previous Invoker**. In the new file write the following code:



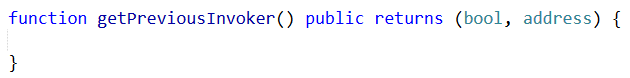
We have created out contract. Now we need to add the necessary functionality to it.

What we need first is to have a variable that keeps the **address** of the previous invoker in the persistent storage. Also, the variable must not be accessible outside the contract. Finally, the data type of the variable must be of type **address**. The code that achieves all this is the following:



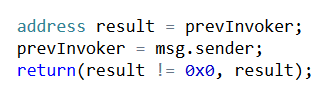
First is the **data type** - **address**. Next, we state that the variable will not be accessible from outside the contract - **private**. Finally, we write the name of the variable - in this case we have chosen **previousInvoker**.

Next, we need to write a **function** that **returns** the required information. The function **must be accessible** from outside the contract. To achieve this, we need the following code:



We start the keyword **function** followed by the name we chose for it. In this case this is **getPreviousInvoker**. Next, in parentheses we define what data types, if any, are to be passed to the function. If we are not going to pass any data types, as in this exercise, we only write the parentheses. Afterwards, we specify that the function is going to be accessible from outside the contract with the keyword **public**. Lastly, since the function returns a result, we must use the keyword **returns** (notice that there is an **"s"** in the word) and specify in parentheses what data types the function is going to be returning.

We have defined out function. Now we must write its code:



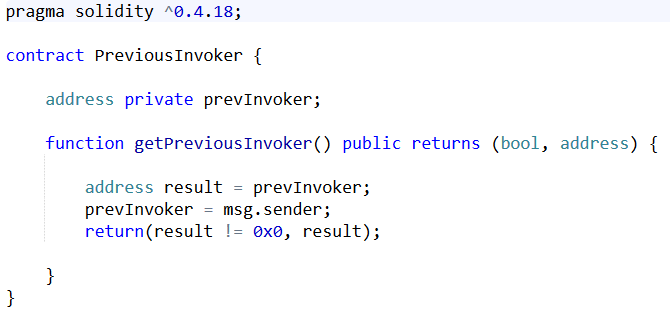
Let's go through the code line by line.

First, to achieve what we intent we need an additional variable with data type **address**. This variable is going to keep the address of the previous invoker. We assign to it the value of the variable **prevInvoker**. Now, for the time being, the value of **prevInvoker** is zero, since we have not assigned any address to it yet. Hence, the value of the variable result will also be zero for the moment.

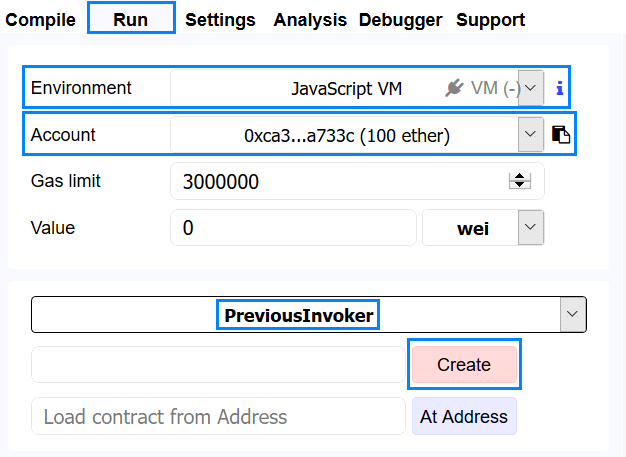
Next, to get the address that is going to invoke the function we use **msg.sender**. We assign this address to the variable **prevInvoker**. Now, the value of this variable is no longer zero but is the address that has invoked the function. Now we have all the information that we need. What is left is to **return** it.

If a function is to return any information we need to use the keyword **return** (notice that there is not an **"s"** at the end). After this keyword, in parentheses, we specify the variables whose data type must correspond to the one specified when defining the function. In this case, first we check whether result is different from zero. If it is not, then **false** is returned, otherwise - **true** (false is used to say that a previous invoker does not exist, whereas true means that the address that is returned belongs to the previous invoker). Hence, the data type is **boolean** which corresponds correctly with what we have specified. Next, we return the variable **result**, whose data type is **address**, which also corresponds to what we have specified.

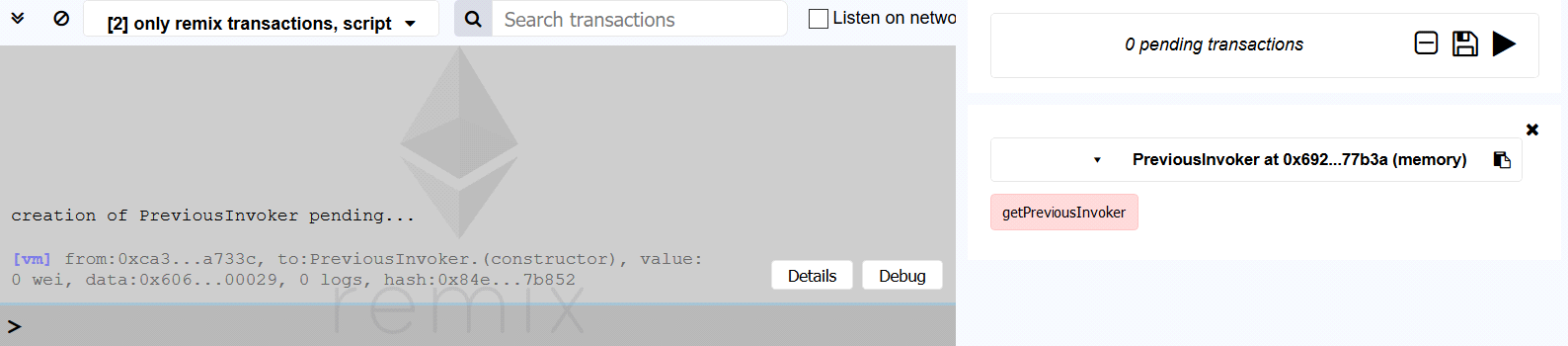
By now your code must look like the one in the following screenshot:



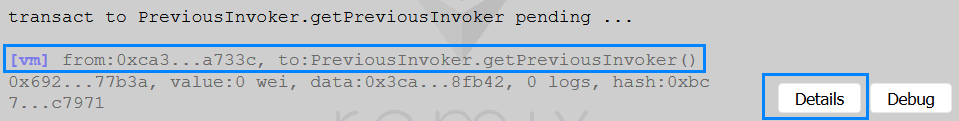
Let's test it!



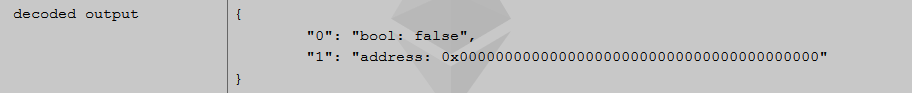
In the upper-right corner of your screens you should notice some tabs. Chose **Run**. Next, make sure that the **environment** is **JavaScript VM**. Also, take a note of the address which you are using at the moment, you will need it for the testing of the contract you have just written. The name of the contract is displayed and right underneath it you will notice the **[Create] button**. Click it. Now the bottom half of your screens should be looking like the next screenshot:



In the gray box there is some feedback about the creation of the contract. This does not interest us for the moment. On the right side of the gray box you should be seeing a **[getPreviousInvoker] button**. Through it we call our function. Click it. Some additional feedback is now added in the gray box:

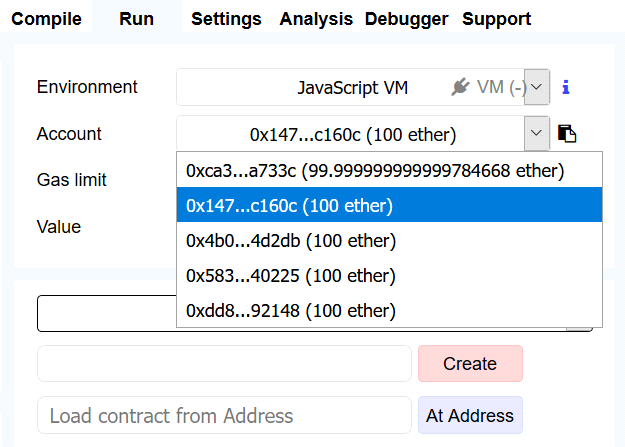


Here you can see which address (**from)** has called which function **(to)**. Next, click the **[Details] button**. Look for the row **decoded output** (to reach it you must scroll down). You should be seeing the following:

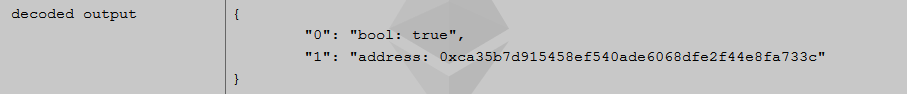


Since this is the first time that the function was invoked there is not an address that has invoked it previously and the returned value of the **bool** data type is **false** and the value of the returned data ty **address** is zero.

Now let's change the address through which will call the function again. To achieve this, we must choose another address from the drop-down menu next to the  **Account** label:

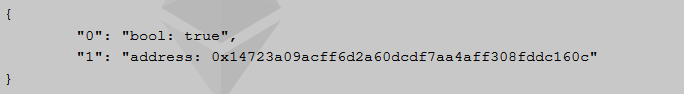


After you select a different account click the **[getPreviousInvoker] button**. Again, another feedback should appear in the gray box. Click on the **[Details] button** and scroll down until you reach the **decoded output** row:



Now, the value of the boolean should be **true** and the **address** should be different from zero. In fact, the address must be the same as the addressed that you used to invoke the function the previous time.

Now, let's choose the address that we used to deploy the contract and call the function one last time. From the drop-down menu next to the **Account** label chose the address that you used to create the contract with and click the **[getPreviousInvoker] button**. In the gray box click the **[Details] button** of the new feedback and scroll down until you reach the **decoded output** row:



The value of the boolean variable is again **true**. The address has changed as expected. It is now equal to the previously used address.

Congratulations! You successfully completed this part of the exercises!

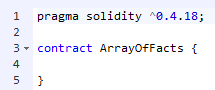
## Array of Facts

Write a simple contract in Solidity that keeps in the blockchain a **string array of facts**. There must be a function that checks whether the user that tries to add new facts is authorized to do so. Furthermore, the contract must have two additional functions – one to return the **length** of the array and the other to return a **fact** that occupies a certain place in the array. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

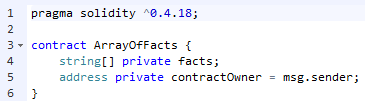
### Hints

Create a new file by following outlined steps in the previous exercises. For this part we named our file **“Array of Facts”**.

Begin the code as usual:

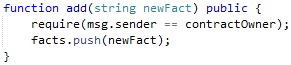


In order for the contract to have the expected functionality two variables must be initiated right from the start. The first value will be the array where the facts will be kept. The second will be used to check whether the user that is trying to add new facts has indeed the right to do so:



We have initialized the variables to be accessible only in the contract and not from the outside. What is interesting here is the code on line 5. With it we simply say that the owner of the contract is the address that has created it. In other words, **msg.sender** holds the address that initializes the creation of the contract.[[1]](#footnote-1)

What we need next is the function that will allow the **contractOwner** to add new facts. This functionality is achieved through the following code:



We create a function with the name **“add”** and make it accessible outside the contract. The next line of code is crucial – we **require** that the address that wants to add a new fact is the address that has created this contract. If this condition is not fulfilled then the next line of code is not executed and an error is thrown.

The next function we will implement is the one that returns the number of facts that are in the array. This we can do this with the following code:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 3\Function Count.PNG

The name of our function is **“count”**, it promises not to change the state of the contract (**view**), it is accessible from outside the contract **(public)** and returns an unassigned integer data type **(returns (uint))**. The value that is returned is simply the **length** property of the variable facts.

The last function that we are about to implement has to accept an integer and return the fact that occupies that place in the array. This we do through the following code:

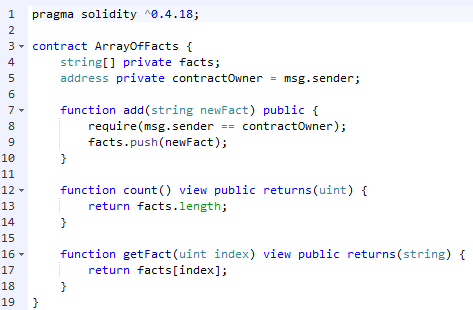
C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 3\Function getFact[index].PNG

This function that we have called **getFact** looks similar to the previous one – there only two differences:

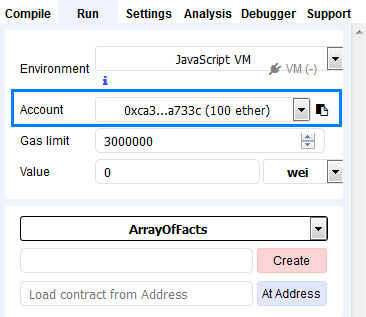
1. An **integer** must be passed;
2. The function returns a **string.**

The time has come to test our code!

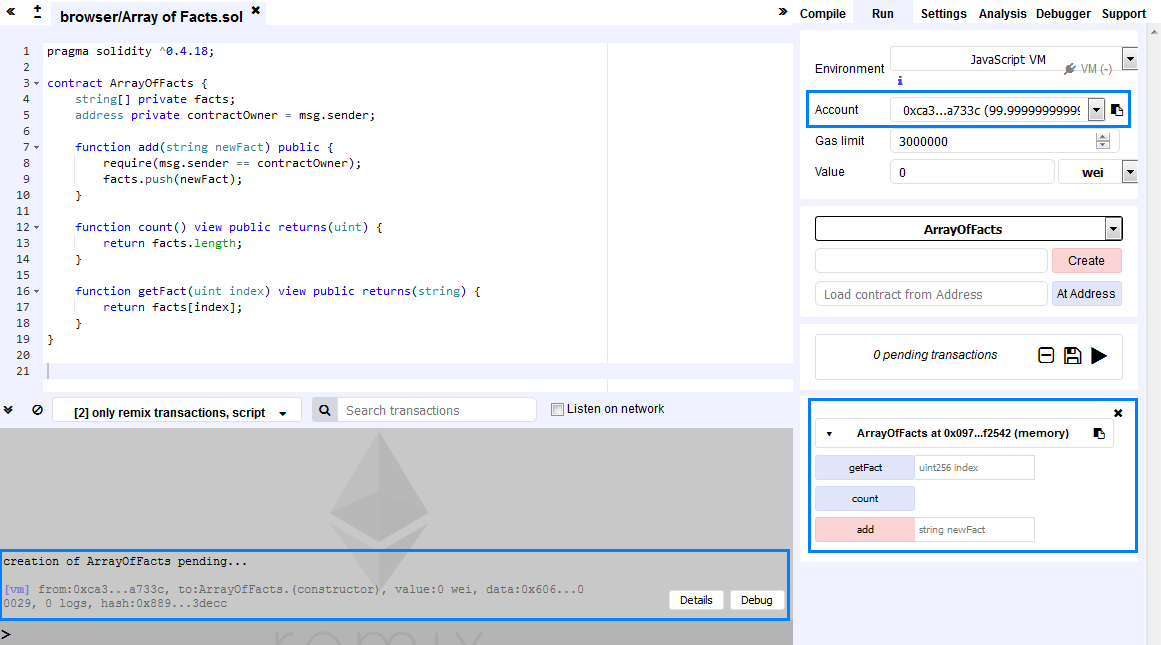
If you have followed the steps your code should look very much like the one on the next screenshot:



Now you should create the contract. While doing this notice the account which you are using at the moment. After clicking the **[Create] button** the amount of available ether should decrease:

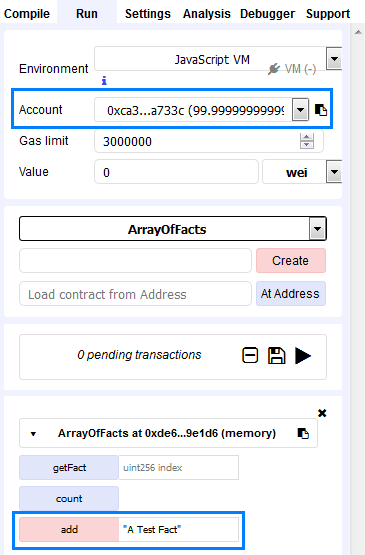


After creating the contract your window should look similar to the next screenshot:

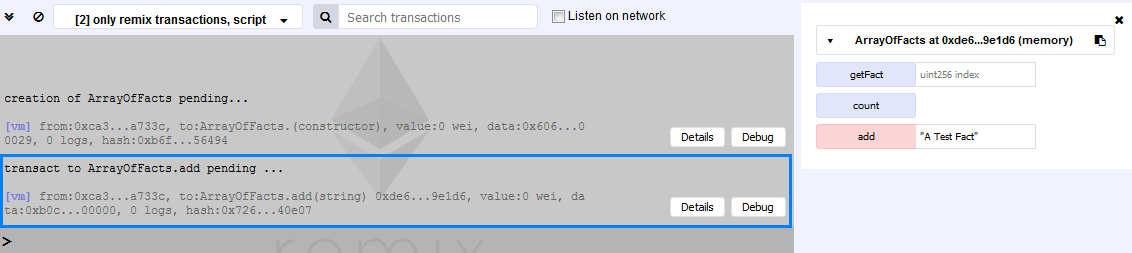


As can be noticed the amount of available ether has decreased, the contract has been successfully deployed and the buttons of the three functions are visible.

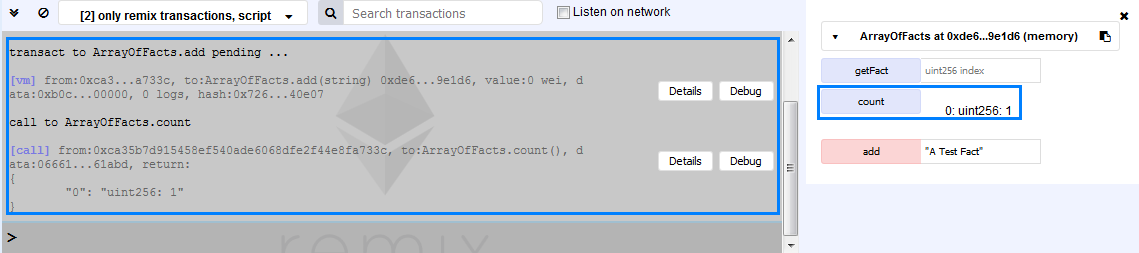
First, we should add a fact, called **“A Test Fact”**. In order to do so we **must not** have changed the account and in the field with **button** **[add]** we place the string **in quotation marks**.



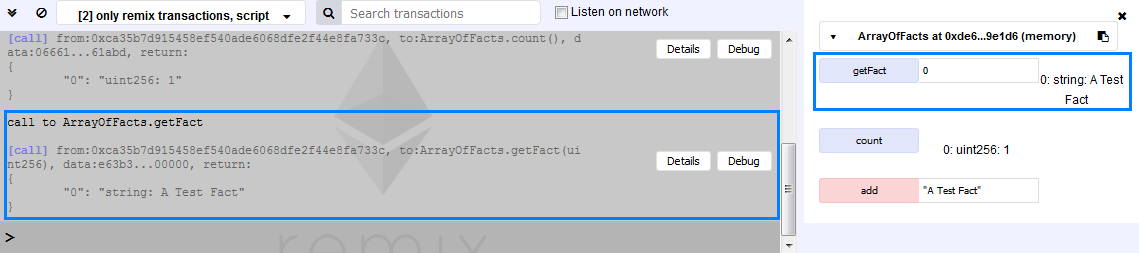
Click the **[add] button**. You should see some feedback from the gray box:



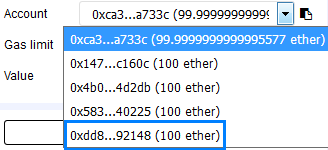
Now, let’s get the count of our facts by clicking the **[count] button**. You should see the value 1 returned by this function:



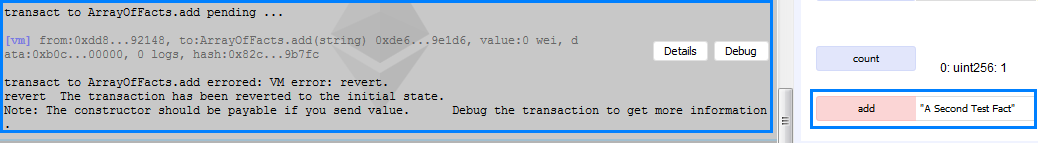
Next, we shall try and get our fact by pointing directly to it. Since we are dealing with **indexes** our only fact is at index **zero**. Hence, the value that we are going to pass to the function **getFact** is **zero**. After clicking the **[getFact] button** you should see the fact that you have **pushed** a few moments ago:



Let’s try some further testing by changing the address:

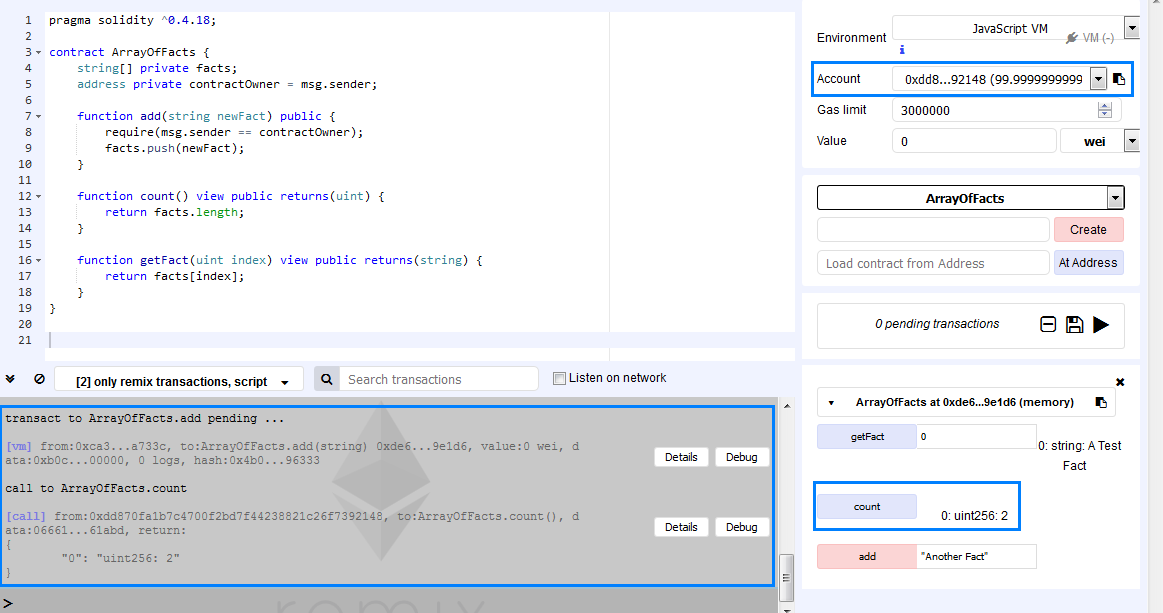


If we try to add a new fact from an address that differs from the one that is associated with the creation of the contract we get the following result:

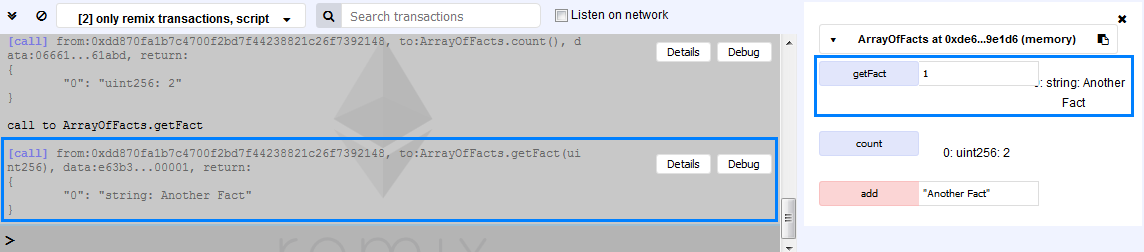


Because the address differs from the one that created the contract no addition to the **facts array** was made and the transaction was reverted.

Now let’s go back to the address with which we can add facts and add a new fact – **“Another Fact”**. After that change to another address and get the **count** of the facts.



From the gray box is visible from which address which functions were called. So far, so good – everything seems to be working as expected. Let’s try to see a fact from an address that differs from the one that created the contract. Since we have only 2 facts for now, the allowed values as inputs are **zero** and **one**. From the following screenshot it seems that we have implemented everything correctly:



Congratulations! You just implemented a contract that checks whether the user is authorized to manipulate a contract’s data.[[2]](#footnote-2)

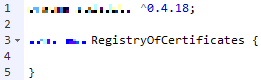
## Registry of Certificates

Write a simple contract in Solidity that represents a registry of certificates. The contract must check whether the user that tries to add a certificate is the **contract’s owner**. If this is so the certificate is added, otherwise – no. The other functionality should allow a user to check whether a certain certificate is in the registry. In this exercise the certificates will be hashed and the hashed value will be passed to the appropriate function. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected. You may use this link <https://emn178.github.io/online-tools/sha3_512.html> to create your hashes.

### Hints

In order for everything to work as expected we will be using two variables. One of them is a simple **address** but the other will be a **mapping**. When we map a certificate’s hash to a uint it is very easy to check whether certain certificate exists in the registry.

As usual we start by creating and naming a new file[[3]](#footnote-3):

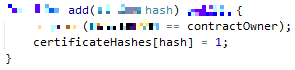


Next, we will initialize the two variables that we are going to be using:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 4\Variables Initiation 1.png

The first variable maps a **string** to a **uint**. You should specify a key word that makes this variable accessible only in the contract. The second variable specifies that the contract’s owner is the address that initiates the contract’s creations.

The next piece of code that must be written is the function that adds a **certificate’s hash** to the variable **certificateHashes**. This should be allowed only if the address calling the function is the owner’s address.



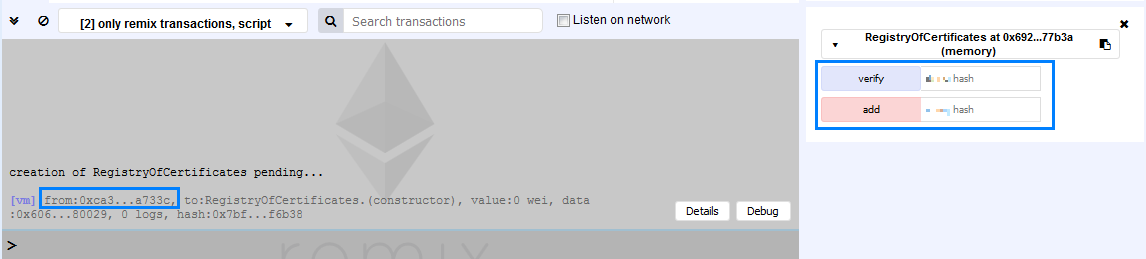
The name of our function is **“add”** and is accessible from outside the contract. Inside it resides a requirement that the address that calls the function is the same as the address that owns the contract. If this is the case we map the passed **hash** of the certificate to the **value** **1**. Why this is done will be obvious after writing the last function which will be called **“verify”**.

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 4\Function Verify 1.png

This function receives the hash of the certificate that we are interested in (since hashing the same value always produces the same hash value). The function must **promise** **not to change** the contract’s state with a certain key word. In addition, the function should be **accessible** **from outside** the contract and it must be specified that this function **produces a value** that is either true (if the certificate is in the registry) or false (if there is not such a certificate in the registry). The verification is achieved through a single line of code that checks whether the passed **hash** is mapped to the **value 1** – if it is, then such a certificate exists in the registry, otherwise there is not such a certificate.

Now we should test whether everything works as expected.

First, we deploy the contract, all the while noting the address we are using. The bottom of your screens should resemble to the following screenshot:



On the left, in the gray box, the address that has deployed the contract is visible. On the right you should be having two fields associated with **[verify]** and [**add] buttons**. Now, using the address that deployed the contract, let’s try adding a certificate with the name **“Certificate of Proficiency in Solidity”**. The hash that this name produces (without the quotation marks) is

ae2a982435afe921046aaeb3458fb3c7afa2731176f12bbd6aaec80dd3f1d90d63a1b89d749889bd739ca924970a05c96fdcd91d530e9ebcd98b760ac27e5654

Add this value to the registry.[[4]](#footnote-4) In the gray box you should see something that looks like this screenshot:

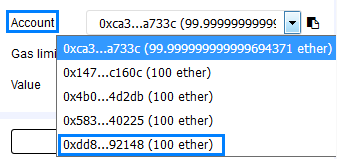
C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 4\Added Certificate 1.png

The function **add** has been called by the address in the field **from**. Using the same address, we will try to check whether the certificate has been registered. To do this we will pass the same hash into the **verify** function.[[5]](#footnote-5) The result should look as follows:



The information in the gray box shows which function has been called, from which address and what value the function has returned. In this case the function returns true, because there exists such a certificate (or rather the **hash** of the certificate exists in the registry).

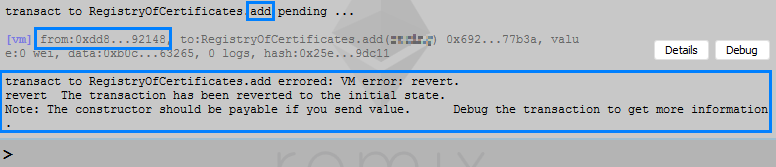
Now we should try to add a certificate’s hash from **another** address. From the Remix IDE find the label **“Account”** and from the drop-down menu choose a different address.



After having selected it try passing to the **add** function the following value:

4d0708679980aa73079996c3fc858fb24ccb15e5950178641261e8abcbfc72bb095819bc30f632480a61c4781f580ed76b0d872368dc6dee5d2739138a198f2e

This is the **hash** of the text **“Certificate of Advanced Solidity”** (without the quotation marks). You should see similar information in the gray box:

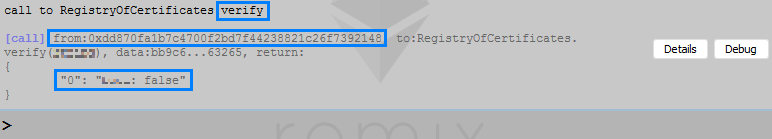


The **add** function is called by an **address** that ends in 92148 (notice that it differs from the one that deployed the contract). Because of this an error message is shown and the hash is not added to the registry.

Now, let’s try to check whether a certificate’s hash is in the registry by using this same address. The result should resemble the following:



Here we use the hash for the certificate **Certificate of Proficiency in Solidity** since we know it is in the registry. From the presented information it is visible which address calls the verify function – one that is different from the owner’s address. Finally, let’s check whether the **hash** of the **“Certificate of Advanced Solidity”** (without the quotation marks) has somehow crawled into the registry. By placing the appropriate **hash** in the field of function **verify** you should get the following result:



The information shows what function has been called and by which address. The function returns **“false”** and shows that there is not any such hash of a certificate in the registry.

Congratulations! You have successfully implemented a registry for certificates![[6]](#footnote-6)

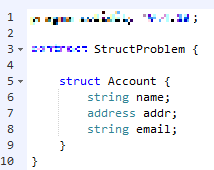
## Problem about Structs

Write a simple contract in Solidity that keeps track of users’ names, addresses and e-mails. The contract must check whether the user that tries to get users’ information is the contract’s owner – other users must not be able to consult the records. The contract should provide functionality that allows to anyone to add their address’ information. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

### Hints

For this exercise we will be using a **struct** variable. It will hold the **name** of the user as a **string**, the **address** of the user as an **address** and the e-mail of the user as a **string**. In addition, we will use two **modifiers** to check that certain conditions are fulfilled.

Let’s start as usual and define our **struct** data type:



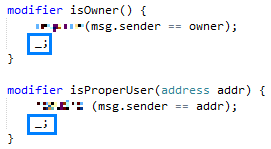
We name our **struct** data type **“Account”**. It will hold information about the **name**, **address** and **e-mail** of the user. Next, we need a variable that will store all users – we will use an **array** that uses our struct.

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 5\Array Initialisation.PNG

Additionally, we will need a variable that will keep the **address** of the owner. This variable **must not be accessible** from the outside.

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 5\Owner Variable Initialisation 1.png

Now we need to create two conditions. The first one must check whether the user that tries to access the records is the contract’s owner. The next condition must check whether the user’s address is the same as the address that is to be placed in the records. This needs to be checked so that there aren’t any randomly passed addresses. This we are going to achieve through the usage of two **modifiers**. The piece of code that we are going to need is presented in the next screenshot:



Let’s look at the first modifier. We named it **isOwner**. For it to function correctly we need a **requirement** verifying that the current user is the owner. If this is true the next line of code is executed. The symbol in the blue rectangle simply says **continue**. Hence, if the requirement is not fulfilled then this line is not executed and the code execution stops.

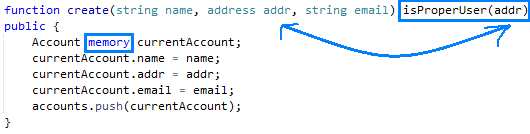
The next **modifier** is called **isProperUser**. To it we pass an address that must be the same as the address of the user. If this is not the case, the code execution is stopped, otherwise it continues.

Now we need a function that determines who owns the contract. This function must be accessible and invoked only ones, at the moment the contract is created. In this function we assign to the previously initiated variable **owner** the address that creates the contract:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 5\Constructor Initiation 1.png

This code is the contract’s **constructor**. What makes this function different from any other defined in the contract is the name of the function. Whenever a function is defined with a name that is **exactly** the same as the **contract’s name**, then this function fulfils the role of the **constructor**. Furthermore, this function is called **only** at creation time and can never be called again after the contract’s deployment.

Our next step is to define a function that creates a record. The code is presented in the next screenshot:

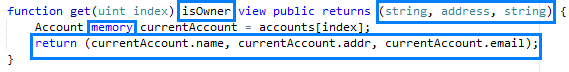


Let’s get through this function step by step.

First, we define the function with its proper **key word**. We name it **“create”** and specify what type of data types will be passed to it. After the closing bracket we place a **modifier** to which we pass the **address variable addr**. Through this we check that the address that the user passes is the same as the address that tries to create a record. If this is the case the code in the next lines is executed, otherwise the code execution is stopped.

In the function’s body we initialize a variable of data type Account and we name it **“currentAccount”**. We use the key word **memory** to specify that this variable is a temporary one. Afterwards we pass the input variables and finally we **push** our temporary variable into the **array** where it will be kept for future reference.

The code for our last function is presented in the next screenshot:



After we have specified what type of data the function will receive we place our **modifier** that checks whether the address that calls the function is the address of the owner of the contract. If this is the case, the code execution continues, otherwise it is stopped.

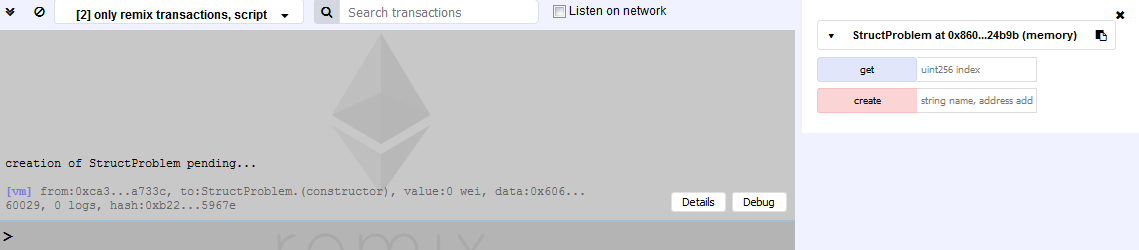
Another thing that you should notice is that the function **returns multiple variables**. To achieve this after the **key word returns** in brackets are specified the type of data that the function returns – in our example the data types are **two strings** and an **address**.

Next, we initialize a **temporary variable** (using the **key word memory**) and name it **“currentAccount”**. In it we will keep the account we are looking for.

Finally, in order to return the information, we need, after **return** in brackets we specify the values that we are interested in. In our case this is the **name (string)**, the **address (address)** and **email (string)**.

Time to test our code!

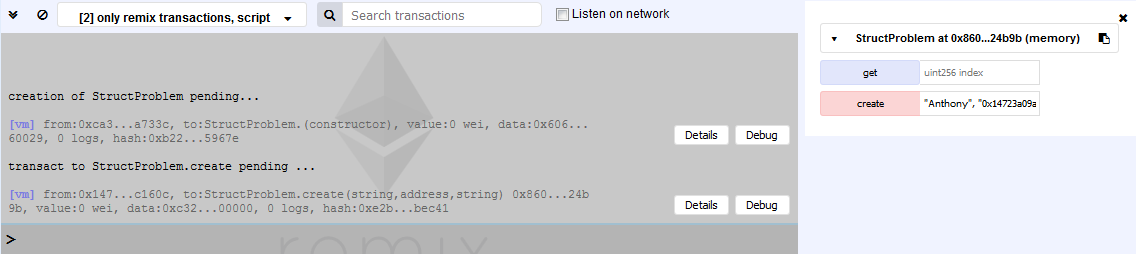
After you create the contract you should see something similar to the next screenshot:



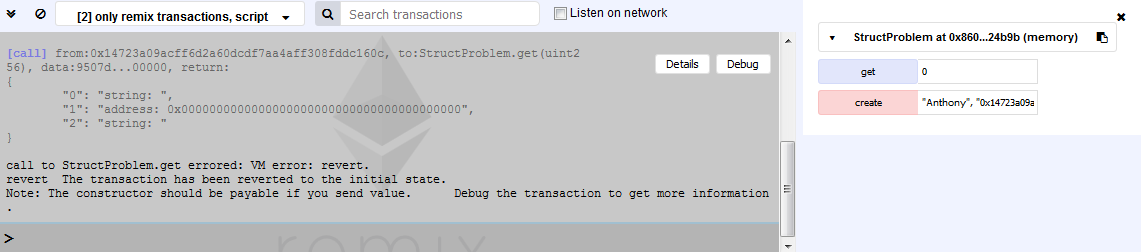
First from **Account** we change the address. Next, we will use the function **create**. To it we pass a **string** for the **name**, the **current address** and a **string** for the **e-mail**. In our example we pass the following:

"Anthony", "0x14723a09acff6d2a60dcdf7aa4aff308fddc160c", [anthony@somewhere.com](mailto:anthony@somewhere.com)

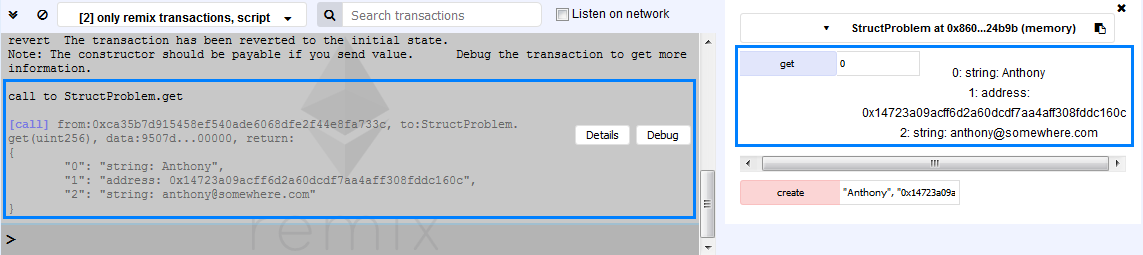
After clicking the **[create] button** you should see something similar to the next screenshot:



Now we have a **record at index 0**. Using the same address try to call the function **get**, passing to it the value 0. You should get a result similar to the one in the next screenshot:



It is obvious that no information whatsoever concerning the records was accessed. Now, change back to the address that was used to create the contract and again call the **get** function, passing it the value 0. Now you should see the following result:



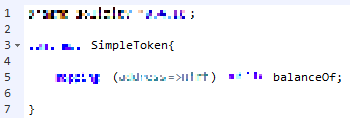
And that’s that! You have learned how to pass and return multiple variables and have successfully implemented and used the struct data type.

## Simple Token

Write a contract in Solidity that represents a simple token. The initial supply must be set at contract’s creation and this amount must be allocated to the address that creates the contract. Add a functionality that allows for transfers of tokens between the address of the contract’s creator and other addresses. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

### Hints

First and foremost, you will need a variable that keeps track of how much **tokens (positive integer value)** are owned by which **address**. This variable must be accessible from outside the contract since we want to be able to check the balances of the different addresses. Let’s start!

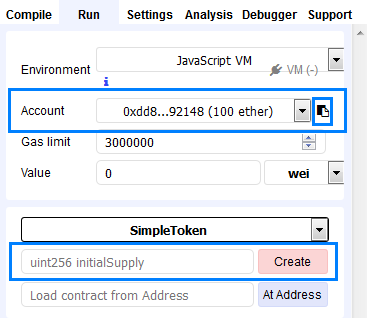


The variable **balanceOf** will keep track for the current ownership of tokens.[[7]](#footnote-7) What we shall present next is the crux of this problem – how to assign at creation time the **initial supply** of the token. In addition, one must not be able to tamper with this amount after the contract’s creation. The logic is as follows.

We need a certain functionality that will allow us to call a function exactly one time, at creation time. This function must accept a positive integer, which will represent the **initial supply** of the token that will be assigned to the contract’s owner. After the contract’s creation it must be impossible to change this amount, even for the contract’s owner. The reason for this is obvious – everyone must be absolutely certain that the initial supply will adhere to what has been agreed upon – otherwise the token will simply won’t have any value, it won’t attract any investors’ attention. The next screenshot should help you in writing the code:

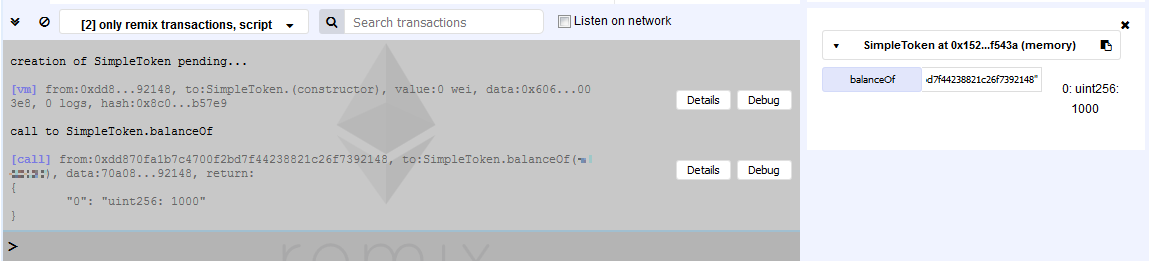
C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 7\Constructor 1.png

Let’s test what we have written so far.

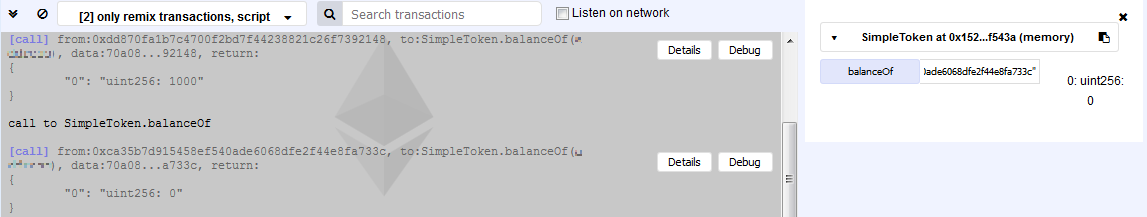


First, note the address which deploys the contract (also, do notice the icon in the small blue rectangle right next to the arrow for the drop-down menu – by clicking it you copy the currently selected address). Then, next to the **[Create] button** you will see a field which uses uint as an input – this is the field where you must set the number of tokens that are going to be available in the contract. After you have decided on the amount (for example 1000) write the value and click on the **[Create] button**.

Now a **[balanceOf]** **button** must be available in the bottom-right corner of your screens. In its field place the **address** which you have used to deploy the contract then click the associated button. What you will see should resemble to the following screenshot:



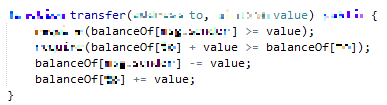
Next, as an input choose a different address and check its balance. The value must be zero.



Everything seems to be working as expected. Let’s continue!

What we need to do now is to write a function that allows the transfer of tokens between the **owner’s address** and **users’ addresses**. Even if this sounds trivial there are some conditions that must be fulfilled in order for the function to behave accordingly.

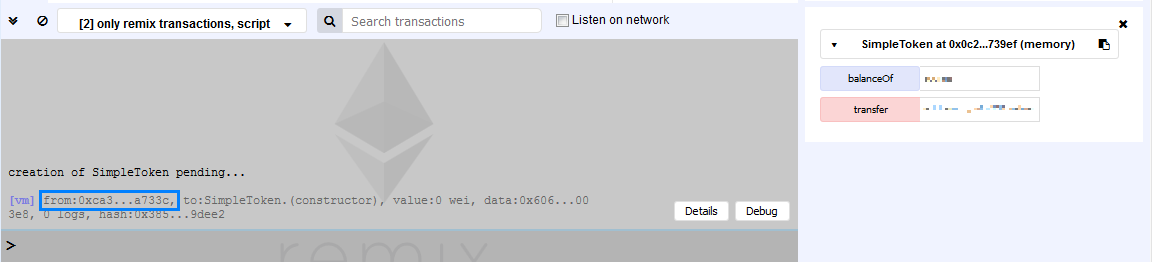
First, the function must be accessible outside the contract. Second, we need to make sure that there are enough tokens available at the owner’s address. Next, we must protect the users from **overflows[[8]](#footnote-8)**. If all the requirements are fulfilled then the holdings of the user are increased with the required amount and the holdings of the contract’s owner are decreased with the same amount. The following screenshot should help you write the necessary code:



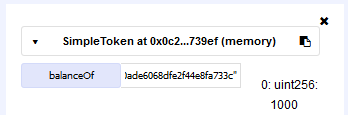
Time to test the code!

Choose the address that will deploy the contract. Notice that **only** through this address tokens can be send to other addresses. Next, choose the number of tokens that are going to be available. Write the value in the appropriate field and create the contract.

After the contract’s creation two buttons will be available in the bottom-right corner of your screens – **[balanceOf]** and **[transfer]**. The first shows the balance of an address, the latter is used by the contract’s owner to send tokens to an address. It should look similar to the following screenshot:

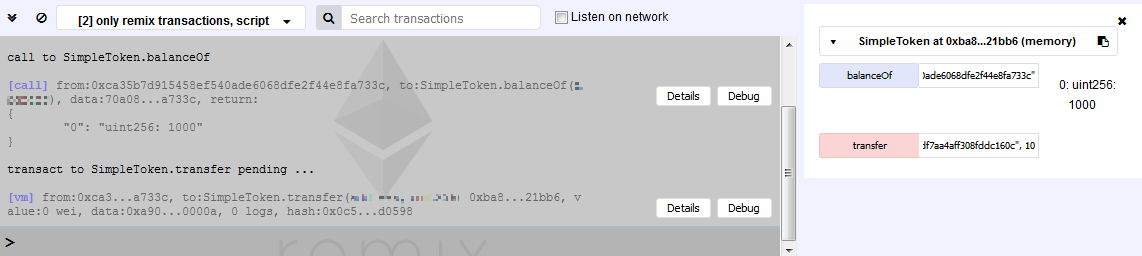


Notice the address that follows **“from:”** this is the contract’s creator. Only through this address tokens can be send to another address. If you pass this address to the field next to the **[balanceOf] button** and then click it you will see the number of tokens that the contract has been deployed with.



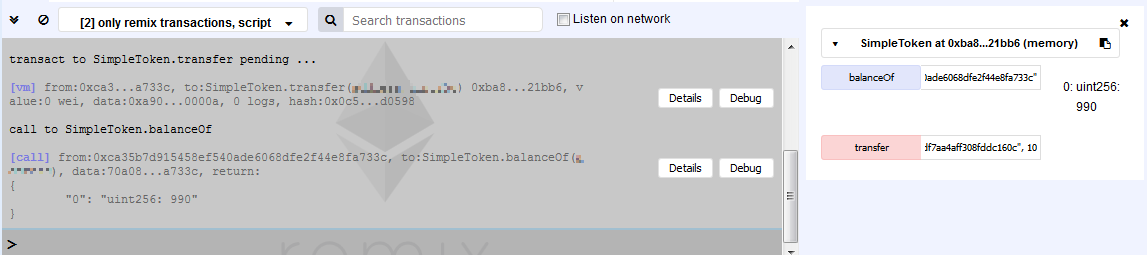
Now, to send tokens to an address!

From the drop-down menu with label **“Acount”** select an address that is different from the one that you have used to deploy the contract. Copy and paste it into the field next to the **[transfer]** **button**. Again, from the same drop-down menu chose the address which was used to create the contract. Then after the address to which you want to send tokens add a **comma** and finally specify the **amount** that you are willing to send. Click the **[transfer]** **button**. You should see something similar to the following screenshot:

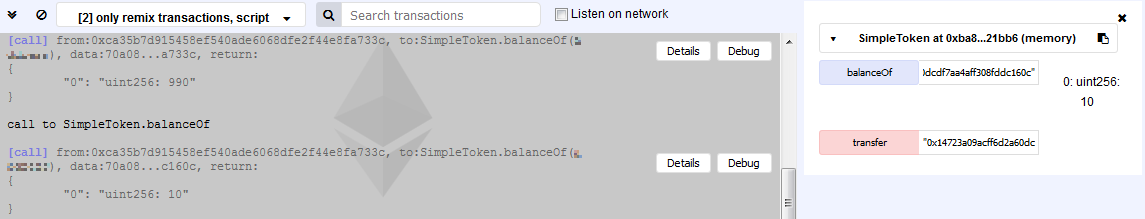


What you should notice is that the addresses after the **“from:”** are the same. The first call is to **balanceOf** which shows the balance in tokens of the address (1 000 in this case). The next call is to **transfer** which transfers a certain number of tokens (10) to another address. Now is the time to check the balances of the two addresses:

First, the balance of the address that created the contract (and from which tokens are send):



Next, the address that received the tokens:



And that’s it! You have created a contract that is deployed with a predefined number of tokens, that keeps track of how much tokens each address holds and the functionality to send tokens from the address of the contract’s creator!

# What to Submit?

Create a **zip file** (e.g. your-username-intro-solidity.zip) holding the **.sol** files from problems above.

Submit your zip file as **homework** at the course Web site.

1. Before continuing we advise you to try writing some simple code to verify this statement. Hint: in the Remix IDE look for a label “Account”. [↑](#footnote-ref-1)
2. We strongly suggest expanding this part of the exercise and adding some additional checks. For example, whether a fact already exists, whether there are any facts at all in the array, or whether the passed index is in the right appropriate range. [↑](#footnote-ref-2)
3. Starting from this problem some of the key words will be masked intentionally. It will be up to you to write the correct expression. [↑](#footnote-ref-3)
4. If you simply copy-paste it, it will NOT work – something must be added to it. Hint: look at Problem 3. [↑](#footnote-ref-4)
5. Again, be cautious how you pass the value. [↑](#footnote-ref-5)
6. We strongly advise you to try adding other functionalities and checks as an extension to this exercise. You may ponder how to check whether a certain certificate is already in the registry or whether there are any certificates registered at all. [↑](#footnote-ref-6)
7. If you create the contract with only this line of code you will notice a field with a **[balanceOf]** **button** next to it. Try it and check the token’s balance of a random address (obviously it should be zero). [↑](#footnote-ref-7)
8. <https://en.wikipedia.org/wiki/Integer_overflow> [↑](#footnote-ref-8)