# Exercises: Solidity Advanced

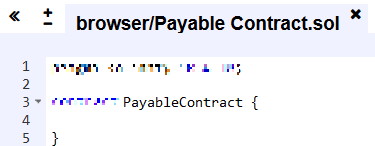
This document describes the **exercise assignments** for the ["Blockchain Academy" course @ Software University](https://softuni.bg/courses/programming-fundamentals). In this lesson, we learned about functions, modifiers, events, contracts, interactions, error handling and libraries in **Solidity** programming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

## Payable Contract

Write a contract that has a function through which anyone can send it some ether. In addition, only the owner can check the current balance 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 file and begin the contract as usual:



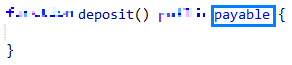
Next, we will need a variable to which the **address** of the owner of the contract is to be assigned and should **not be accessible** from outside the contract:



What we now need is the **constructor** which we will use to assign the owner`s address to the variable **owner**. The function must be accessible from outside the contract:

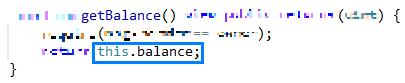


After we have defined our **constructor,** we next need to write a function through which **everyone** will be able to **send ether** to the contract:



Now, through this function the contract can receive ether. Since we do not need any additional functionality at the moment we leave the function`s body empty.

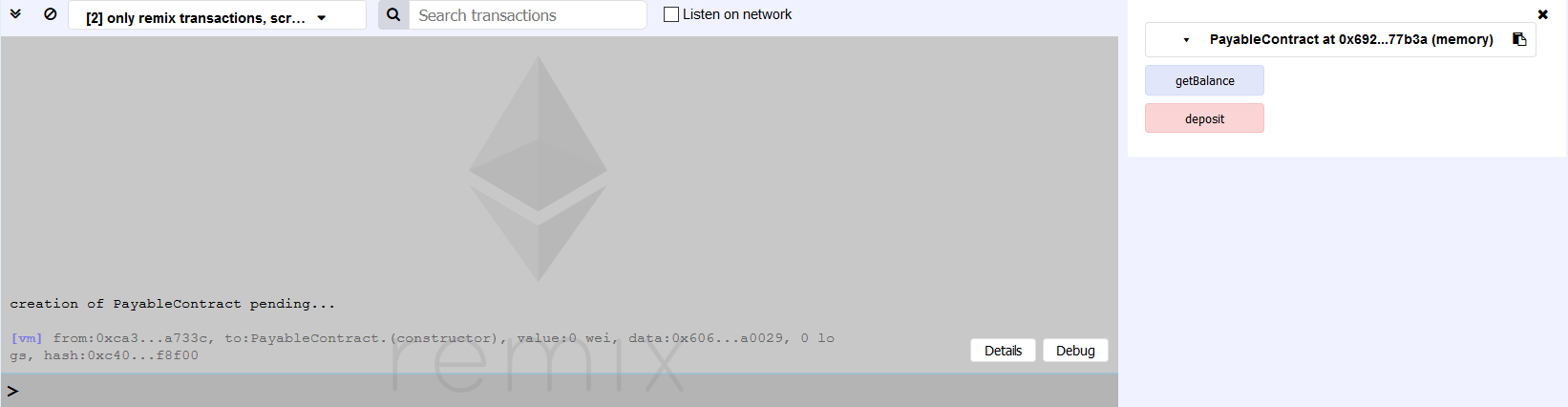
Next, we need a function that promises not to change the contracts state and allows **only** **to the owner** to check the contract`s current ether balance. The owner should be able to access this function outside the contract. The following screenshot should help you with writing the code:



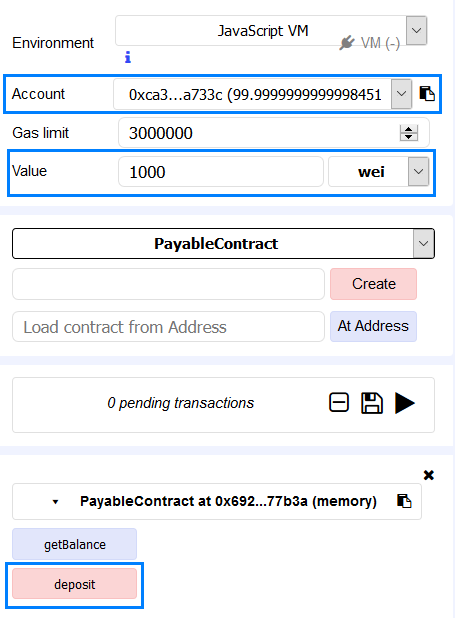
First, we state a requirement that the address calling the function is the owner`s. If that is not the case, the code execution stops. To access the contract`s ether balance we use **this.balance** which will be a positive integer.

We are almost done. What is left is to test our code!

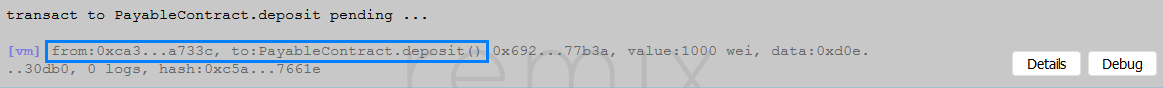
Create the contract and note the address through which you are doing so. Now, the bottom of your screens should look like this:



Now, still using the creator`s address, we should deposit **1000 Wei[[1]](#footnote-1)** to the contract`s address. To achieve this, we need to write **1000** in the field with label **Value** and click on the **[deposit] button**:



After clicking on the button, you should receive some feedback in the gray box, resembling to the one in the next screenshot:



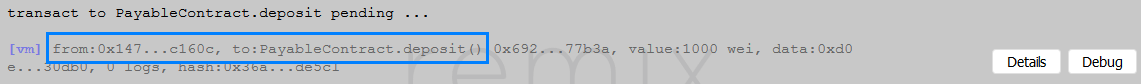
Still using the same address, click on the **[getBalance] button**. You should get the following result:



Everything seems to be working as expected.

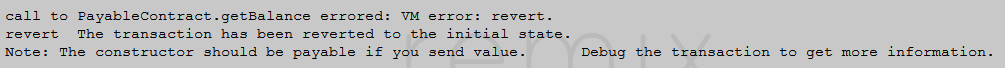
Now, change the address by choosing a different one from the drop-down menu, next to the **Account** label.

Having changed the address, we will deposit another **1000 Wei**. This we will do again by writing 1000 in the field next to the label **Value** and then clicking the **[deposit]** **button**. In the gray box, you should have received a feedback similar to the one in the next screenshot:



After **from:** you should be seeing the new address that deposited another 1000 Wei.

Now, still using this address, try to check the current balance of the contract by clicking the **[getBalance] button**. The new feedback should look like this:



This error is thrown because we tried to access a function that checks whether the address that calls it is the right one.

Select the address with which you have created the contract and click the **[getBalance] button**. Now you should see the increased balance of the contract:



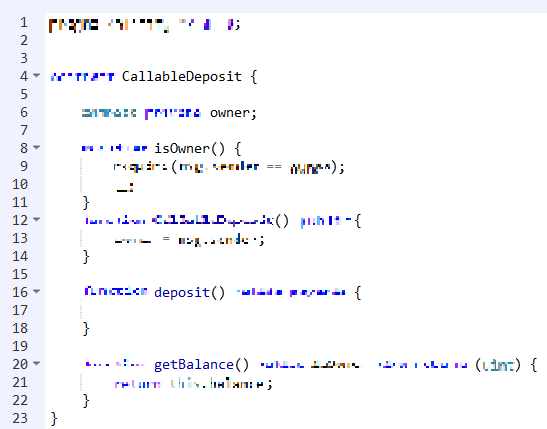
Great! You have successfully created you first contract that can receive ether and you have seen how to check the contract`s current balance.

## Receiving Funds Without Payable Function

For this exercise, we will need two contracts. The first one will have function that can be called by anyone and through which ether is sent to the contract. Only the owner will be able to call the second and third functions – the one that returns the contract`s balance and one that destroys 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 **.SOL** file and name it. Then start our contract as usual.[[2]](#footnote-2) We will be needing a variable that will **not be accessible** from outside the contract and at creation time will be assigned the **owner`s address**. This time the check will be done by using a modifier. Next, we will need a function that **promises not to alter** the contract`s state and returns its current balance if called by the owner. We will also need a **function through which ether is send** to the contract and which is **accessible to anyone** willing to do it. The next screenshot should help you to write the necessary code:



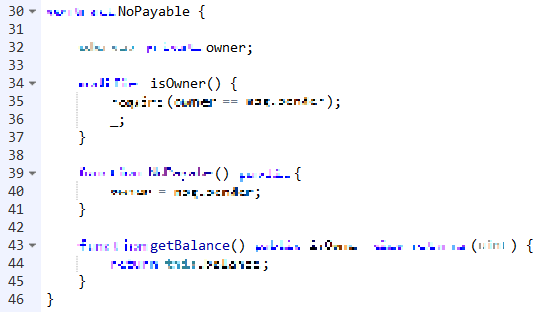
All that is left is a function that sends the contract`s balance in ether to an address specified by the owner. The function must be **accessible** from outside the contract and the successful call depends on whether it is made by the appropriate address. The next screenshot should help you in writing the code:



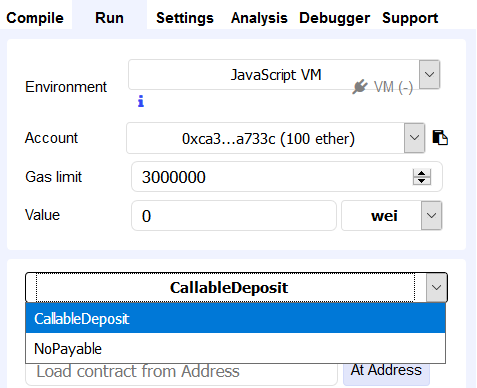
The keyword **selfdestruct** must be passed an address to which the balance of the contract is to be sent. Afterwards, this contract is destroyed.[[3]](#footnote-3)

Our first contract is ready. Now, to write the second one.

Right beneath the contract that we just wrote create another one. As for the first one we will need to make sure that only the owner will be able to see the contract`s current balance by using a **modifier**. For it to function properly we will need a variable that will store the owner`s address. This variable will be assigned the address at creation time. The next screenshot should aide you in writing the code:

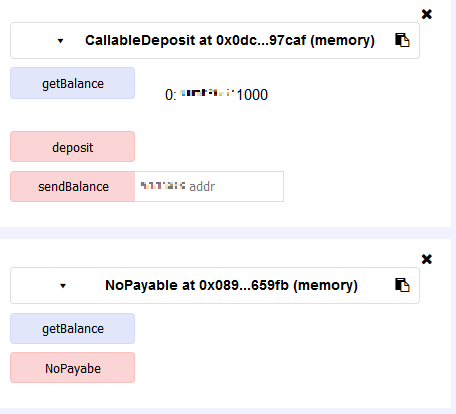


Notice that now you can choose which contract to create:



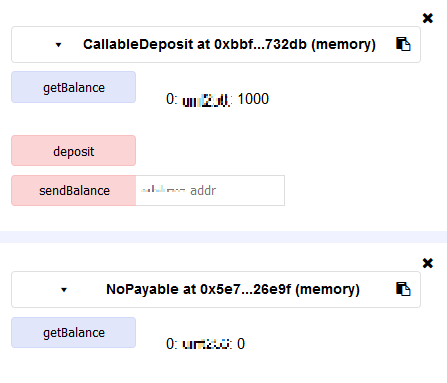
Let`s create the contract **CallableDeposit** and sent it **1000 Wei**. Next, get the balance of the contract. You should see that it is equal to 1000.

Now, from the drop-down menu chose the contract **NoPayable** and create it. In the bottom-right corner, you should be seeing this:

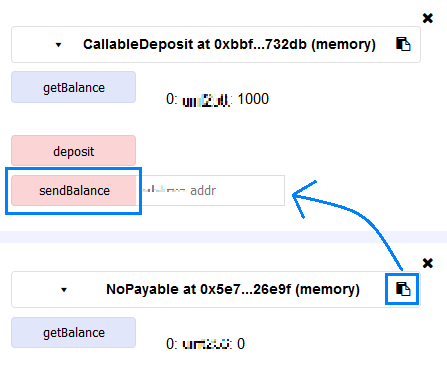


The names of the contracts are visible so you can easily distinguish between the two.

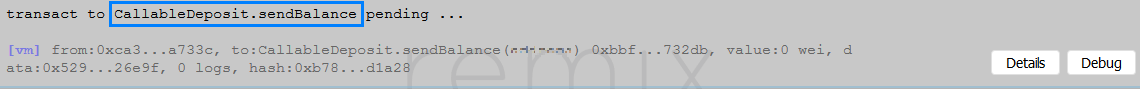
Click the **[getBalance] button** for the **NoPayable** contract. You should see something similar to the following screenshot:



So, the first contract has a balance of 1000 Wei and the balance of the second one is equal to zero. Now copy the **address** of the **second contract**, paste it in the correct format in the field, corresponding to the [**sendBalance**] button, and then click it:



The feedback in the gray box should look similar to the one in the next screenshot:



Now, try to get the balance of the **CallableDeposit** contract. You should see the following text in the gray box:



Try to get the balance of **NoPayable** contract. You should see that it has increased to 1000:



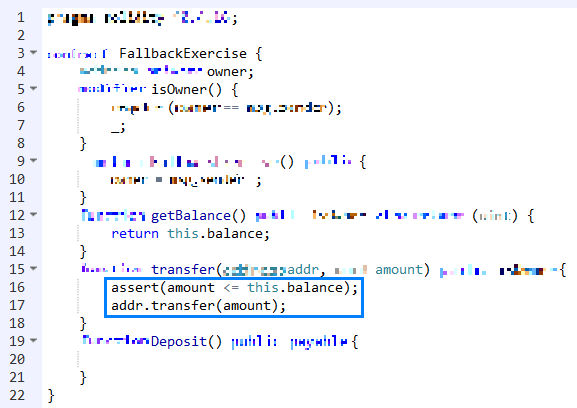
In addition, this is how a balance of a contract can increase without it having a payable function!

## Fallback Functionality

Write two contracts. The first one must have three functions – one that receives ether, one that checks the current balance of the contract, if called by the owner, and one that transfers a specified amount of ether t a specified address. The second contract must have two functions – one returning the current balance of the contract and the other must be payable so that the contract can receive ether. 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

We commence by creating a new file and naming it. In it, we will define out two contracts. The first one will need to have a variable that will be **assigned the creator`s address when the contract is deployed**. The variable should **not be accessible** from outside the contract. The contract will also need a **modifier** to check the requirement that the user invoking a function has the necessary authorization to do so. Next, we define our three functions. The first one **promises** not to change the contract`s state and **returns** the contract`s current balance if invoked by the owner. The second one **transfers** a certain amount of ether to a given address if invoked by the owner. The last one is the function **through which the contract receives ether**. All functions must be **accessible** from outside the contract. The next screenshot should help you in writing the code:

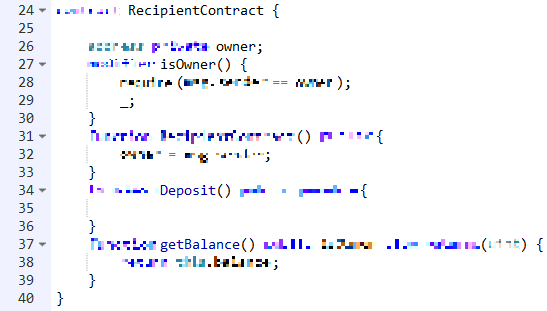


The new functionality we introduce here concerns the code on lines 16 and 17.

The code on line 16 checks whether the amount we have specified is available, i.e. the line **asserts** that the amount is equal or less than what is available at the moment in the contract`s balance. If this is not the case, the code execution is interrupted.

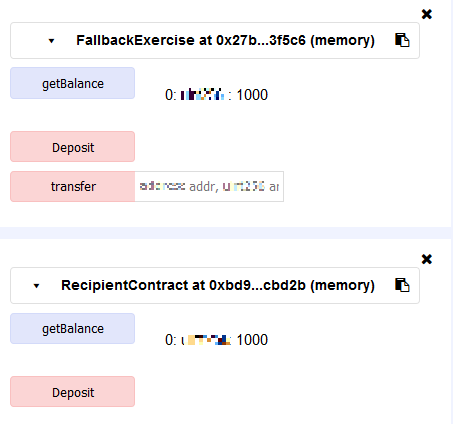
The code on line 17 says that to the specified address a specified amount of the contract`s balance will be transferred.

Right underneath this contract, we will write our second contract. Again, it must have a variable that holds the owner`s address, which is assigned during creation time. We will also need a modifier to check that the proper user is calling the **getBalance** function. We will need a payable function to deposit funds. The following screenshot should help you with writing the code:

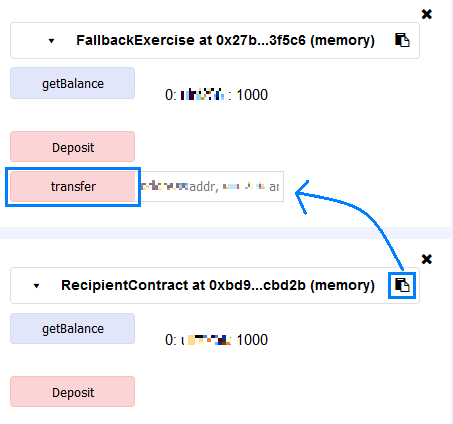


Now, let`s create our contracts and do some test!

First, we create the **FallbackExercise** contract and then the **RecipientContract**. Next, we deposit the amount of 1000 Wei in both contracts and check their respective balance. The result should look like the next screenshot:



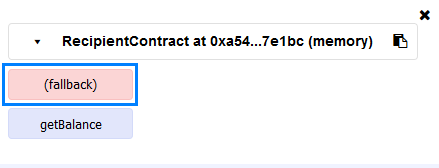
Now, let`s send some ether from the first contract to the second. Copy and paste the address of the second contract in the appropriate format and write the required amount in the field next to the **[transfer] button** and click the button (make sure that the amount is **less** than 1000).



But what is this? The feedback from the gray box show that there is something wrong…

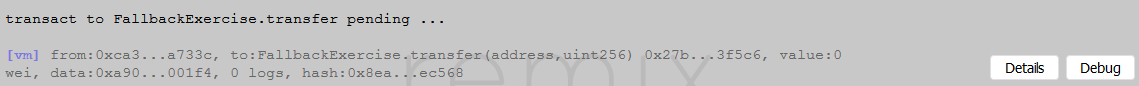


The problem is that if we want to achieve our goal of sending ether between contracts we need to have a **fallback** function that is **payable**. However, in our second contract we have named our **payable** function **Deposit**. Therefore, to remedy this we simply need to remove the name. Delete it and create the contract anew. Now it must look like this:

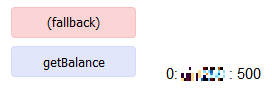


Notice that by removing the name of the function it is now displayed as **(fallback)**.

Since we have just created the contract, its balance is zero. Time to change that. Try again sending some ether to the contract`s address. For this example, we chose to send 500 Wei. Now the feedback looks promising:



But let`s check the contract`s balance:



Congratulations! Now you know how to send ether between contracts![[4]](#footnote-4)

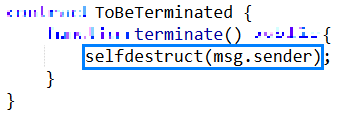
## Terminable Contract

Write two contracts of which one must contain a function that destroys the contract. All other functionality must be inherited. Only the owner can terminate the contract and receive its balance. Before it is terminated, some ether must be send to its balance. 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

If the contract that is to be terminated must contain only one function, then all other functionality must be **inherited**.

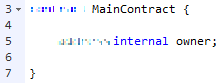
First, we will write the contract that is going to be terminated. It contains only one function that must be accessible outside the contract. The next screenshot will help you in writing the function:



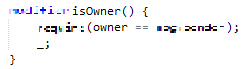
What the single line of code does is that it destroys the contract and sends its ether balance to the address specified in the brackets.

This does not do what we stated. This contract cannot receive ether nor is there any conditional concerning which address can terminate the contract. However, we cannot add any additional functionality in this contract due to the conditions that we have imposed. Hence, we need to write another contract with the functionality that **ToBeTerminated** is to **inherit**.

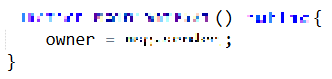
**Above** the contract that we have written, we add another one. First, we will initialize an **internal** variable that will keep the address that will create the contract. The variable is **internal** because it needs to be accessed from a contract that **inherits** the contract in which the variable is initialized (and at the same time **not accessible from outside** the contract).



The next thing we will write is going to be a **modifier**. It will check whether an address is equal to the owner’s address. If it is the code execution continues, otherwise it is interrupted. The next screenshot will help you to write the code:



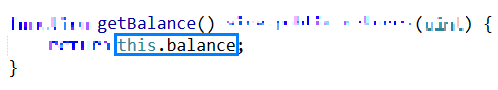
For this modifier to make any sense we need to create a function that **is accessible outside** the contract and **is called only when the contract is deployed**. In it we assign the address that creates the contract to the variable **owner**. The next screenshot will give you an idea how to write the code:



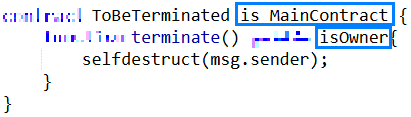
What we now need is a function that will receiver **ether**. This we will achieve by defining the function **payable** and making it accessible from outside the contract. The code is shown in the following screenshot:



To check that the code acts as intended we will add another function - one that checks the balance of the contract. It needs to be **accessible from outside** it, must **promise that it will not change the contract's state** and must **return a positive integer**. The next screenshot will aide you in writing the necessary code:

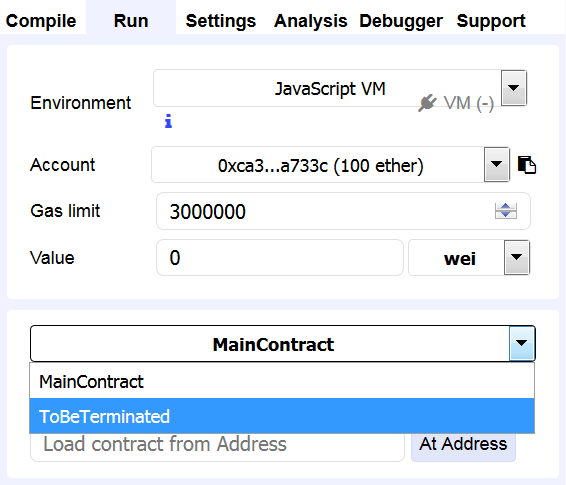


There two pieces missing if everything is to work properly. First, in order for the **ToBeTerminated** contract to inherit the **MainContract** we need to add **is MainContract** to the name of the **ToBeTerminated** contract. Secondly, we must add the modifier to the function (so that only the owner can successfully call it) that is in the **ToBeTerminated** contract:



We are done! Now, let's test our code!

The first thing to do is to deploy the **correct** contract - ToBeTerminated. To do this we must select it from the drop-down menu, shown in the following screenshot:



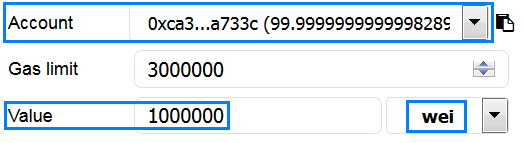
After you select it click the **[Create] button**.

What you will next notice is that although the **ToBeTerminated** contract has only one function in it, you have three buttons for three different functions - one that shows the balance of the address of the **ToBeTerminated** contract, another one is used to **deposit** ether and finally we have the function that terminates the contract.

Let's first check the address' balance. It must be equal to zero (simply click the **[getBalance] button**):



Next, through the field with label **Value** we will send some Wei to the **contract's address**:



We have chosen an amount equal to 1 000 000 Wei and we click the **[deposit] button**. The following feedback should be seen in the gray box:



Also, notice that the amount of available ether has decreased:



Now, let's again get the balance:



Good! Everything seems to be working.

Now notice how the amount of available ether in the current account increases as we **terminate** the contract[[5]](#footnote-5):



This is so, because **selfdestruct** sends all of the address` ether to the **address** that is placed in the brackets.[[6]](#footnote-6)

Congratulations! Now you have a notion how inheritance works in Solidity!

## Event Creation

Write a contract in Solidity that has a function that fires an event. It should show the address of the contract’s owner. 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

We start as usual and write the following code in the body of our contract:

Event initialisation

With this piece of code, we initialize an **event**. After the key word we write the name that we will be using for this event and in the brackets, we stipulate the **data type** that needs to be passed to this event. Since we need the owner’s address, we will be using the **address** data type.

Next, we initialize a variable with name **owner**, which will assign the address that deploys the contract and will not be accessible from the outside. The assignment should be done in the contract’s **constructor**.

Finally, we will need a function **accessible** from the outside, which when called **triggers** the event. The next screenshot will help you in writing the necessary code:

Function ShowAddress 1

An event is triggered simply by **writing its name** and passing the **correct data type** in the brackets.

Let’s test our code!

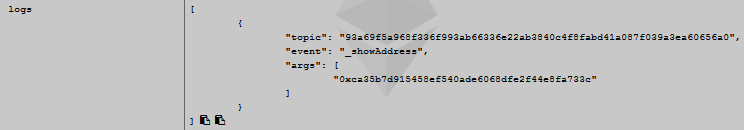
After you deploy the contract, you should see something familiar to the next screenshot:



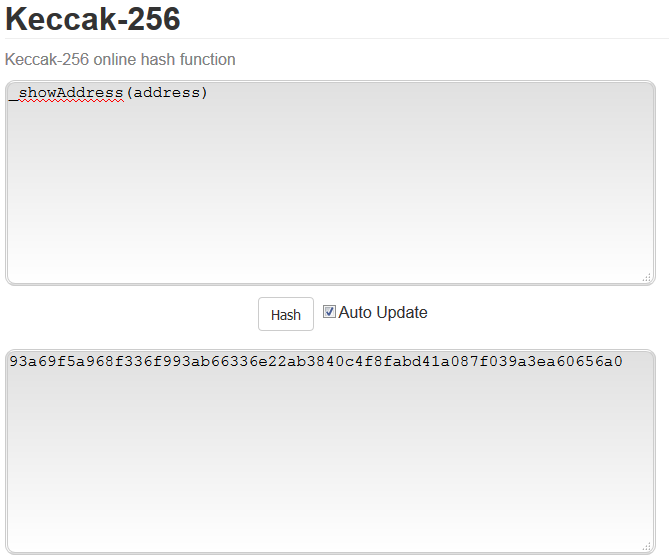
Now, click on the **[showAddress]** **button**. In the gray box, you will receive a similar feedback:

Feedback

Click on the **[Details] button**. Now you see a lot of information but what is of interest to us for the moment is the row entitled **logs** (you will need to scroll down to reach it):



In the right column, you will notice **“event”** followed by the name of our event. Right below it, you see the **“args”.** You should compare this address with the one that created the contract – the **must** be the same. If you take the **name** of the **event** with its **parameters** in brackets and has it with Keccak-256 you will receive the **topic[[7]](#footnote-7)**:



This is how a simple event is implemented!

## Event with Multiple Parameters

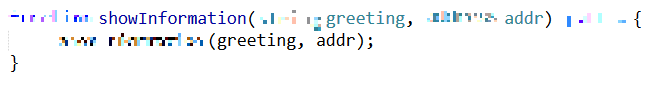
Write a contract in Solidity that contains a function that receives a string and an address. An event must be used that shows the values of these two variables. Use the [Remix IDE](https://remix.ethereum.org) to write the code, publish the contract in a testing environment and test it to ensure it works as expected.

### Hints

An **event** can receive more than one parameter. What needs to be done is simply add another data type to be passed. Therefore, we use the following code to define our **event**:

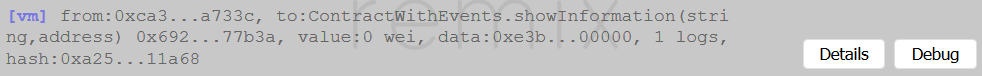
Event Initialisation

Now we need a function that will trigger this event. Since we have not initialized any other variable, the information to be passed to the event will be taken from the passed values to the function itself. The next screenshot should help you in writing the code:

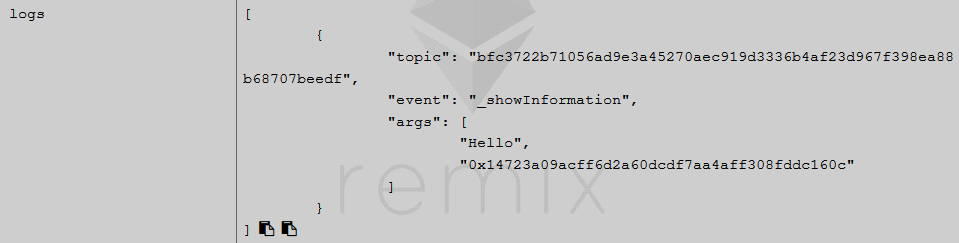


Now let's test our event!

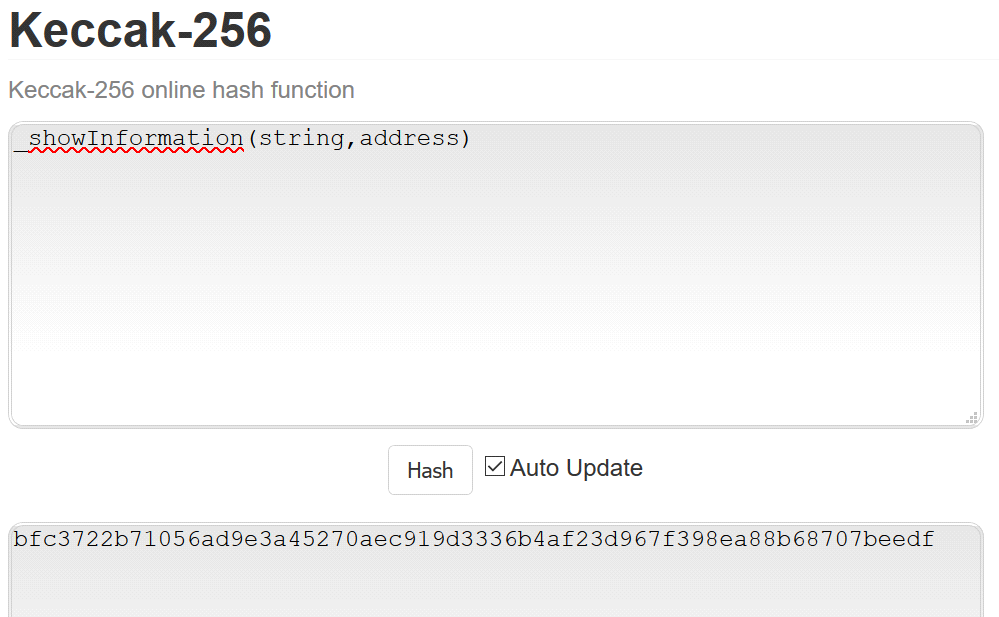
First, we **create** the contract. You should be seeing the **[showInformation] button**, in the bottom-right corner of your screens, with a field for input information next to it. In it write the **greeting** and **address.**[[8]](#footnote-8) In this example we will use the **greeting** "**Hello"** and the **address 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c**. After clicking the **[showInformation] button,** you should see a feedback in the gray box that will be very similar to the next screenshot:



Next, we will click the **[Details] button** and go through the displayed information until we find the row that says **logs**. If you have used as values the ones presented in the previous paragraph, in it you must be seeing the following screenshot:



You see our greeting and the address that we have passed as values. If you would like to use more than two parameters, you simply must specify all that types that will be passed to the **event**, separated by **comma**.[[9]](#footnote-9) In addition, notice that the **topic** is again the **Keccak-256 hash** of the **name** and **data types** of the event:[[10]](#footnote-10)



Now you know how to create and use an event with multiple parameters!

## Event with Indexed Parameters

Create an event with multiple parameters that are going to be **indexed**.[[11]](#footnote-11) 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

We start by defining our contract as usual. Then we initialize the **event** with the following code:



You add index to a parameter with the keyword **indexed**. After it, you specify the **name** of the index. Therefore, to create an indexed parameter you must:

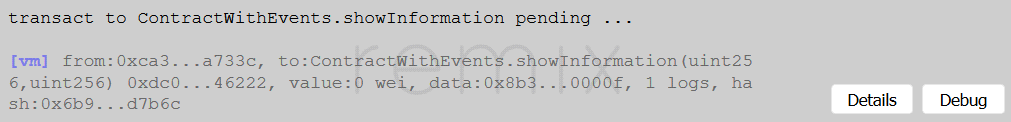
1. Specify the **data type**;
2. Write the keyword **indexed**;
3. Write the **name** of the index.

Next, we will create a function that will be accessible from outside the contract and pass to it some positive integer values. The following screenshot should help you in writing the code:

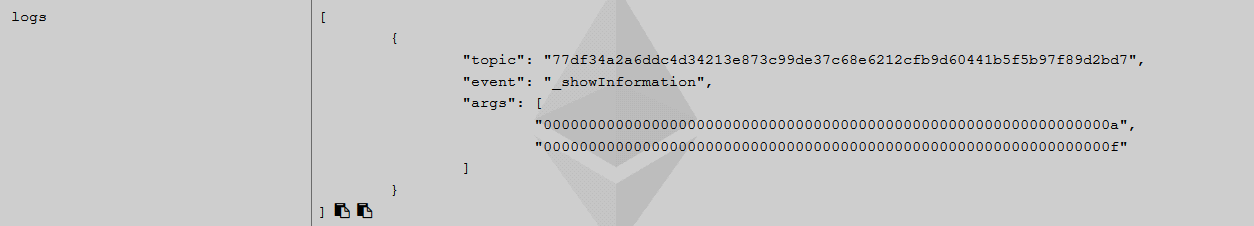


Now, to test the code!

After you create the contract, the **[showInformation] button** will appear with a filed next to it for input information in the bottom-right corner of your screens. In the field, write the values **10** and **15** in the appropriate format. Next click the **[showInformation] button**. The feedback from the gray box must be looking similar to the next screenshot:



Now, click the **[Details] button** and search for the row **logs**. You must be presented with the following information:



The first thing to notice is that if you try to verify the **topic** by hashing the **name** and the **data types** (without their respective **indexed** keywords and **names** for the indexes) you will get a wrong value as seen from the next screenshot:



To resolve this problem, you should only add **256** to the **uint**, the keyword **indexed** and the **name** for the respective index are not used when creating the **topic:**

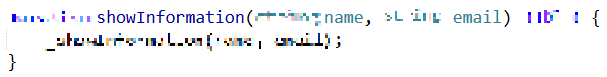
****

Again, it is important to notice that there are not any spaces, the data types are separated by a single comma and the brackets follow immediately after the name of the event.

Next, the values that we have passed are converted to their hexadecimal representation. If there were not any indexing, the values will not have gone any conversion.[[12]](#footnote-12) Now we will rewrite the event and change the data types that are to be passed to it. The next screenshot should help you when you write the code:

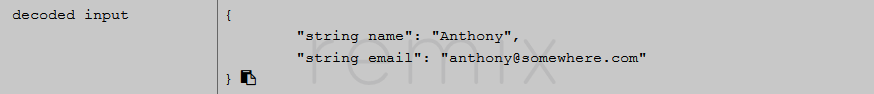


Next, we change the data types and their respective names for our **showInformation** function. it must be still accessible from outside the contract. The following screenshot will help you in writing the code:

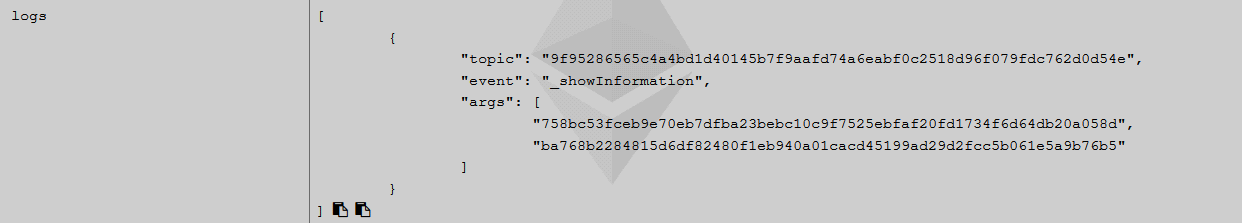


Now to create the contract, pass some values in the appropriate field in the correct format and check what has happened.

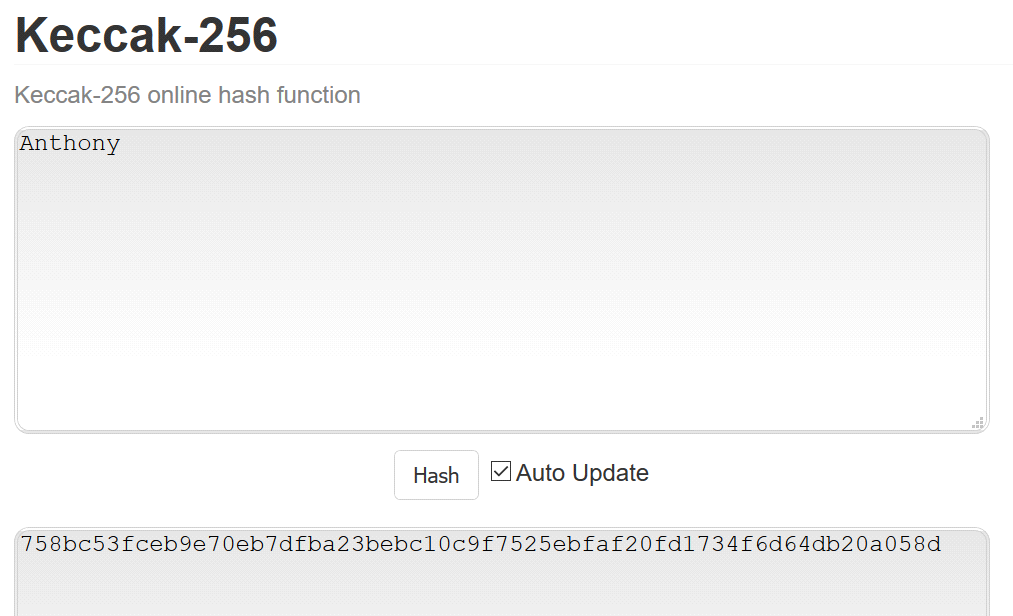
This time, let's first look at the raw **decoded input**. There you should see your input:

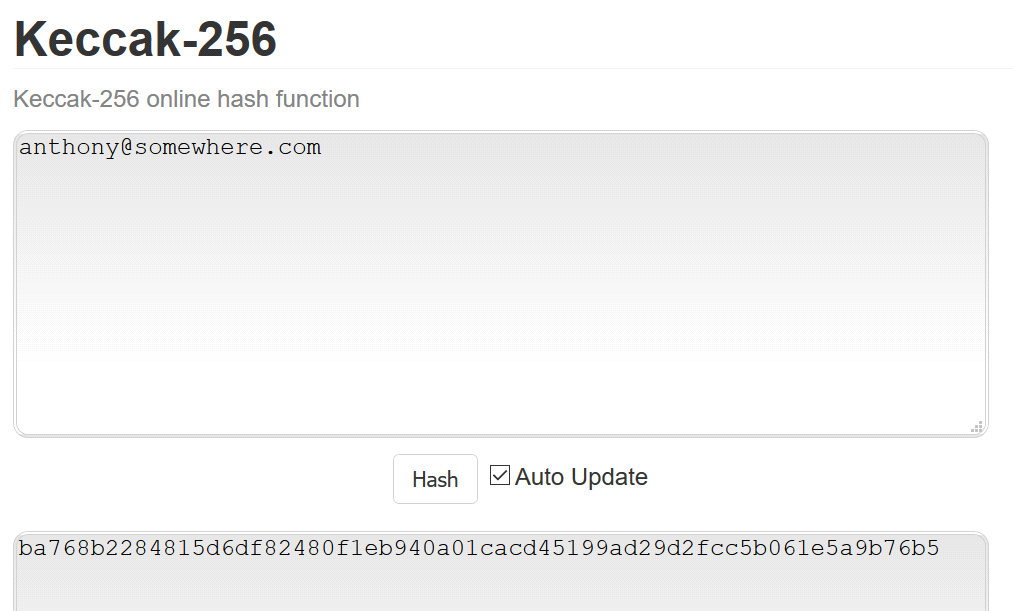


Next, we will look at the **logs**:



You are familiar with how the value for the **topic** is created. But what are these values for arguments?! When a parameter is indexed, it must be stored as 32-bytes data. Since a string may have an arbitrary length, something must be done to compress it. Hence, what we have is the **Keccak-256 hash value** of each string:





Therefore, indexing strings does not work very well now since the argument you receive when you filter by an index is a hash value and you cannot see what has produced it.[[13]](#footnote-13)

Congratulations! You have learned how to use events!

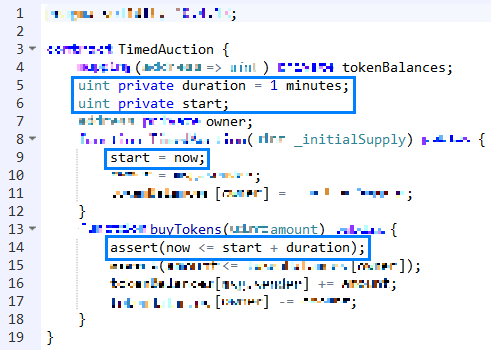
## A Simple Timed Auction

Write a contract for an auction which continues for 1 **minute** after contract`s creation. At creation time an initial supply of tokens must be allocated to the owner`s address. 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 and name it and then start writing the contract. We need to map addresses to amounts of tokens allocated to those addresses. We will need to have a variable that is assigned the time at which the contract is created and a variable that keeps the duration of the auction. We will also need a variable that is going to be assigned the address of the owner. These variables must not be accessible from outside the contract.

Next, we will need a constructor and a function through which tokens are bought and allocated to the correct address. The function should be accessible from outside the contract. The next screenshot should help you in writing the code:

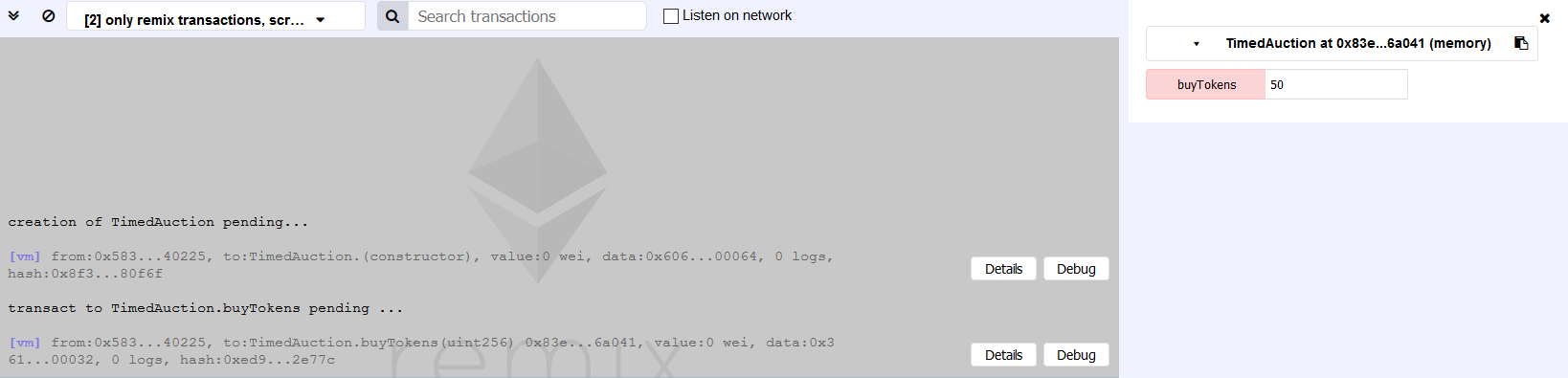


To **duration** we have assign the value **1 minutes**, but we could have chosen 60 seconds or any other time duration that was required. Next, during creation time we have used to keyword **now** to assign the time at which the auction starts.[[14]](#footnote-14)

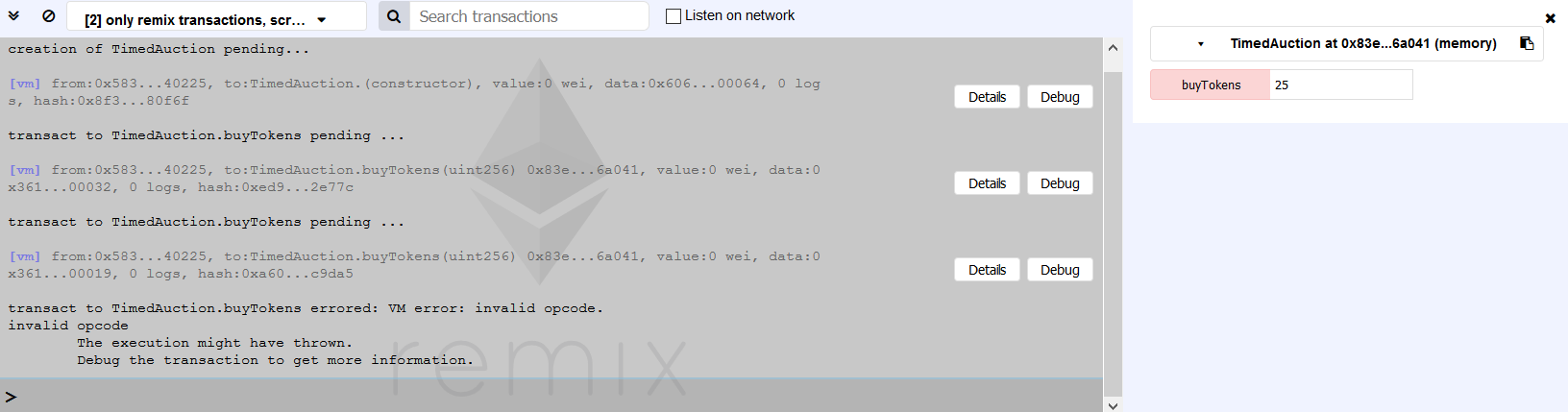
In the **buyTokens** function, we first check whether the auction is still open by comparing **now** with sum of the **duration** (1 minute) and the **start** (the moment at which the auction started). If **now** is earlier that the desired amount of tokens, if available, is assigned to the address.

Let’s test the code to see whether it works as intended.

First, we create the contract (remember to state the amount of the initial supply) and buy some tokens:



According to the feedback, everything works smoothly. Now to wait for a minute to pass and to again try to purchase some tokens:



It is obvious from the feedback that the auction is no longer available.

This concludes this part of the exercises!

## A Simple Timed Auction (2)

Write a contract for an auction, which continues for 1 **block** after contract's creation. At creation time an initial supply of tokens must be allocated to the owner`s address. 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

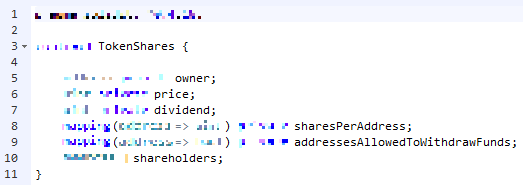
Use **block.number**.

## More Advanced Token: Company Shares

Write a contract that creates an initial supply of shares, their price and the dividend that will be paid. In addition, only the contract’s owner must be allowed to allocate additional funds to the contract. Furthermore, only the contract’s owner must be able to allow withdrawal of funds. In addition, it must be possible only for the contract’s owner to see the addresses of the shareholders. Addresses must be allowed to purchase available shares only on the predetermined price. Shareholders may withdraw funds only if allowed to. Prices and dividends must be ether.

### Hints

The contract begins with the definition of the variables that we are going to be using:



The first one will be keeping the **address** of the **owner** and will **not be accessible** from outside the contract. The second is going to keep the price of the shares as a **positive integer** and will **not be accessible** from outside the contract. Next, we need to map:

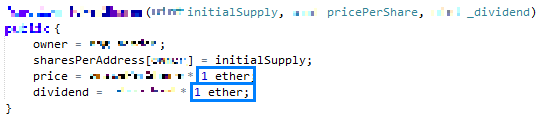
* An **address** to a **positive integer** (sharesPerAddress);
* An **address** to data type that returns **true or false** (addressesAllowedToWithdrawFunds).

Both these variables **are not to be accessed** from outside the contract.

Finally, we need a variable that keeps track of **all shareholders’ addresses** and is **not accessible** from outside the contract.

Next, we must define a function that will be executed only once, at creation time. To it three positive integers must be passed – one for the initial supply of shares, one for the price at which the shares may be bought and one for the dividend that each share will receive. This function must be accessible from outside the contract.

In the function’s body, we must specify which address will be treated as the owner. In addition, the supply of shares must be allocated to the address of the contract’s owner. Finally, the price and dividend must be assigned to variables that we will use later. The next screenshot will help you in writing the function:



One important thing to notice here is the usage of **1 ether**. The reason for this is that floating-point numbers are not yet implemented in Solidity. Hence, if you need an amount equal to 0.1 ether you will use 100 Finney or another denomination. Since we want to set the price and dividend in ether, we need to **multiply** the passed values by **1 ether**. Otherwise, if we pass for a price the value of 1 and do not convert it properly, the price per share will be 1 Wei and this is not our intended result.

The next function we will need must return the price per share (in the appropriate units) so that a potential investor may see it. The function must be accessible from outside the contract and must promise not to alter its state. The next screenshot will help you in your task:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Function GetPricePerShare 1.png

Here, in order to for the price to be **quoted it ether** we need to divide the price by **1 ether**. Otherwise a number that begins with 1 and followed by a lot of zeroes. Since we divide two positive integers, the result will be a positive integer itself.

Another functionality that we will add is a function that calculates the amount of ether a number of shares will cost the investor. The function will not alter the state of the contract and should be accessible from outside of it. The next screenshot will help you when you write this function:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Function CalculateTransactionWorth 1.png

Again, if we are using units of ether we need to convert the calculated amount properly. Since we are using three positive integers, the result will also be a positive integer.

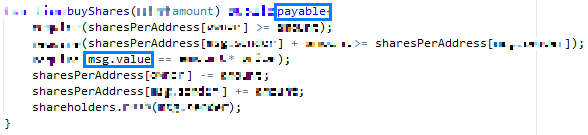
After the potential investor has seen the price per share and how much a certain amount will cost him/her in ether, it is time to write a function that will allow him to buy a certain number of shares. If everything is to work as expected, we need to state a few **requirements**:

* There must be enough number of shares **available** (at the beginning all shares are owned by the contract’s creator);
* We must protect the buyer from an **overflow**;
* We must be sure that the right amount of **ether** is send (this amount depends on the number of shares and the price per share);

If all the requirements are met then we must:

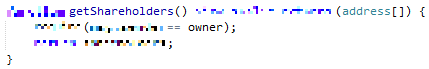
* **Increase** the number of shares for the buyer’s address;
* **Decrease** the amount available for sale (i.e. decrease the amount the owner of the contract holds);
* **Add** the address to the list of investors.

The following screenshot should give you an idea how to write the function that must be accessible outside the contract:

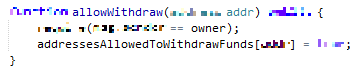


You must notice two things. The first one is that the function must be defined as **payable** if it is to work properly. What this means is that this function will **receive ether and increment the contract’s address ether balance with the amount received**. Hence, the price that the buyer pays in ether goes to the balance of the contract’s address. To access the amount of ether that the investor is sending we use **msg.value**. Therefore, this value must be equal to the number of shares to be bought multiplied with the price per share.

Next, we will implement a function that is accessible from outside the contract and that returns the list of all investors **if** **and only if** the contract’s owner calls the function. The next screenshot should give you a hint how to write this function:



After this function, we add another one that allows only the contract’s owner to select the addresses that can withdraw funds. The function must be accessible from outside the contract and must contain the requirement that the address that calls the function is the same as the address of the owner.

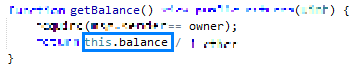
****

The next function allows the owner to **deposit ether to the contract’s address**. It must be accessible from outside the contract. To fulfill the implementation of this functionality correctly we must require that the address that is calling the function is the owner’s.

The next screenshot will help you in writing the function:

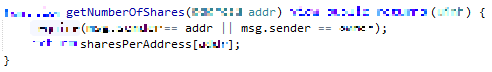
C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Function DepositEarnings 1.png

It would be useful to have a function that returns the contract’s balance, a positive integer (remember that it needs to be **converted appropriately**). This function must return a value only if called by the owner. In addition, it must promise not to change the contract’s state and be accessible from outside the contract. The next screenshot should give you an idea how to write this function:



Notice that the amount of ether available to the contract’s address is accessible through the usage of **this.balance**.

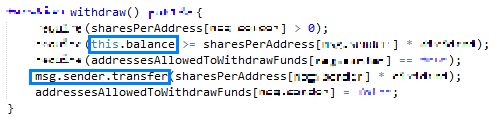
We will also need to be able to check the number of shares that an address holds. The function needs to meet certain criteria. The first one states that an investor may check only his/hers number of shares. The second one states that the owner of the contract may check the number of shares for every investor. In addition, the function must be accessible from outside the contract and must promise not to alter its state. The return data type must be a positive integer and the function must be passed an address. The next screen shot will give you an idea how to write this function:



Our final function will allow an address to withdraw funds. To be implemented properly it must fulfil a few requirements:

* The address must have more than 0 share allocated to it;
* The contract’s balance must have an amount of ether at least equal to the one that is about to be withdrawn (this amount depends on the number of shares and the paid dividend);
* The address must be allowed to withdraw funds;

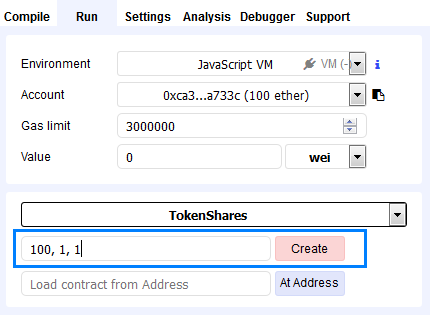
If these requirements are met then a **transfer** is done from the contract’s address in the appropriate amount of ether. Finally, after the withdrawal, it must not be possible to make another one. The next screenshot will aide you in writing the function:



To check whether there is enough ether in the **contract’s balance** we use **this.balance** and then we compare it to the required amount (equal to the number of shares multiplied with the dividend per share). To transfer ether from the **contract’s balance** to the investor’s address we use **msg.sender.transfer** and in brackets, we place the appropriate amount.

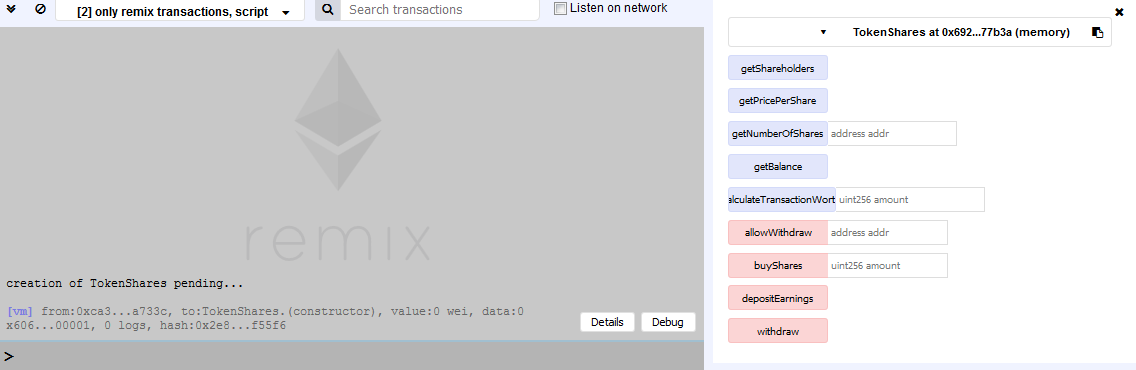
Now to test what we have done!

When deploying the contract, we must specify the number of shares, the price per share and the dividend that is going to be paid per share:



In this example, we will have 100 shares with price of 1 ether and dividend equal to 1 ether.

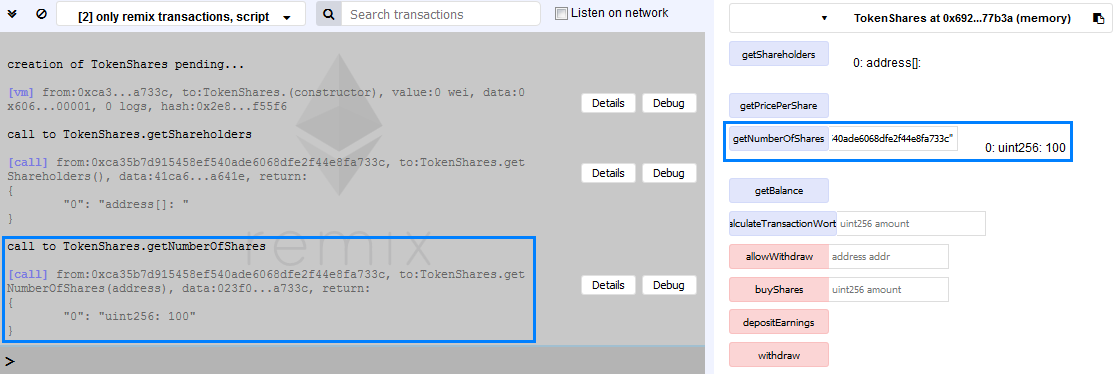
After clicking the **[Create] button** (notice the address you are using to deploy the contract), you should see something similar to the following screenshot:



Clicking **[getShareholders] button** will present you with the following information:

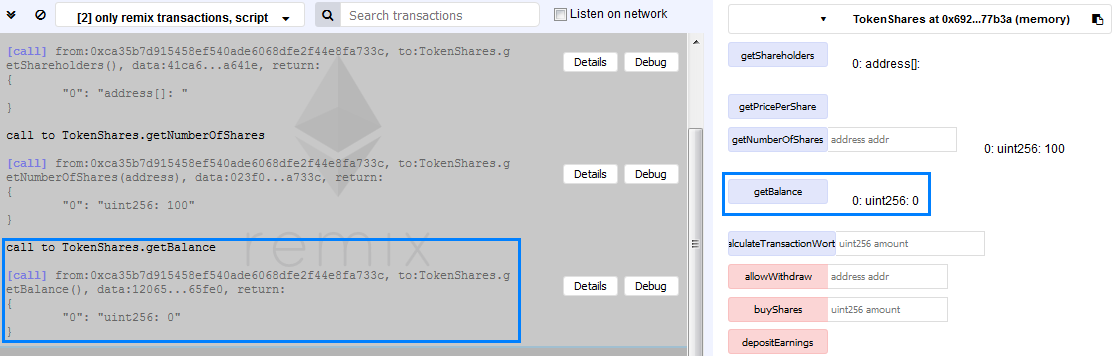
C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\First GetShareholders.PNG

This is normal since there is no one that has bought shares. Let us check the number of shares of the address that was used to create the contract.



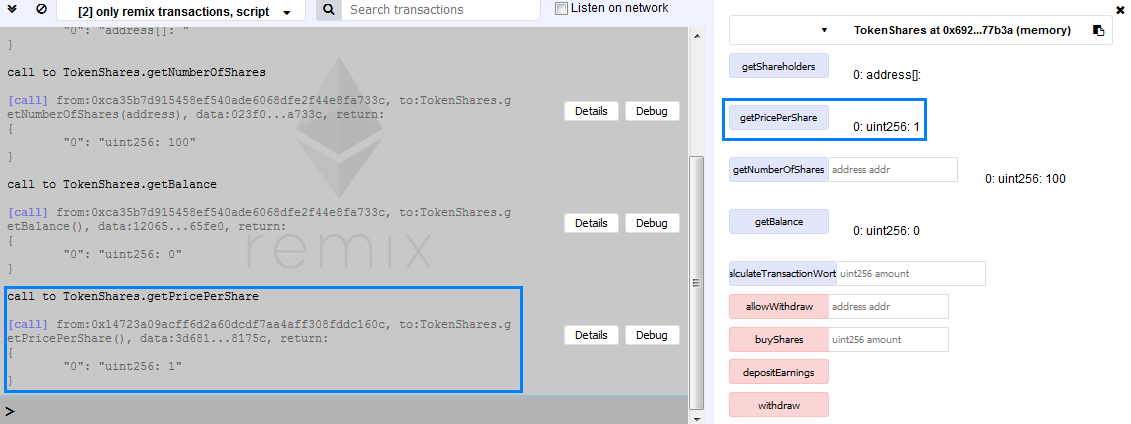
The number of shares that are allocated the address is equal to 100.

Now, let us see the current balance of the contract. Click the **[getBalance] button**. You should see something similar to the next screenshot:



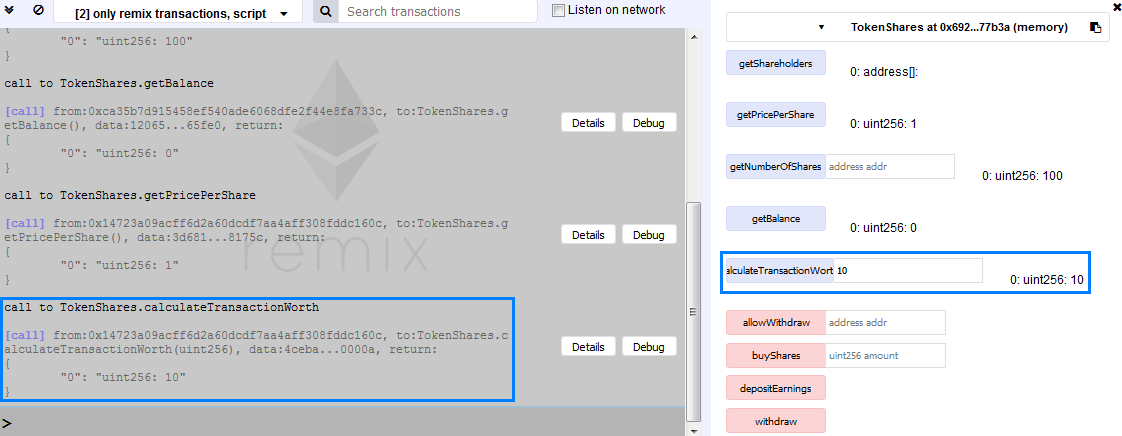
So far, the balance is equal to 0, as it should be. Good. Let’s change to another address and do some tests.

First, we check the price per share by clicking the **[getPricePerShare] button**.

****

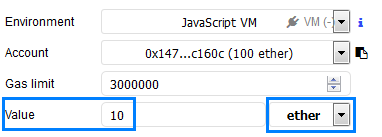
Remember that the price is quoted in **ether**.

Now we will check how much it will cost us to buy 10 shares by clicking the **[calculateTransactionWorth] button** and passing to it the integer 10.

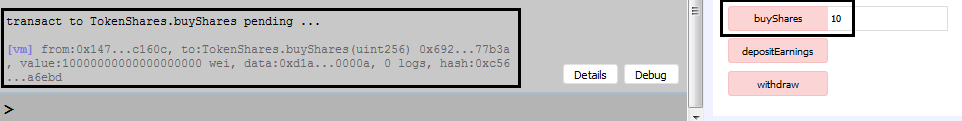


The result is equal to 10 as it should be.

Next, we want to buy 10 shares hence we must send 10 ethers to the **contract’s address**. This we will achieve through the field with label **“Value”**.



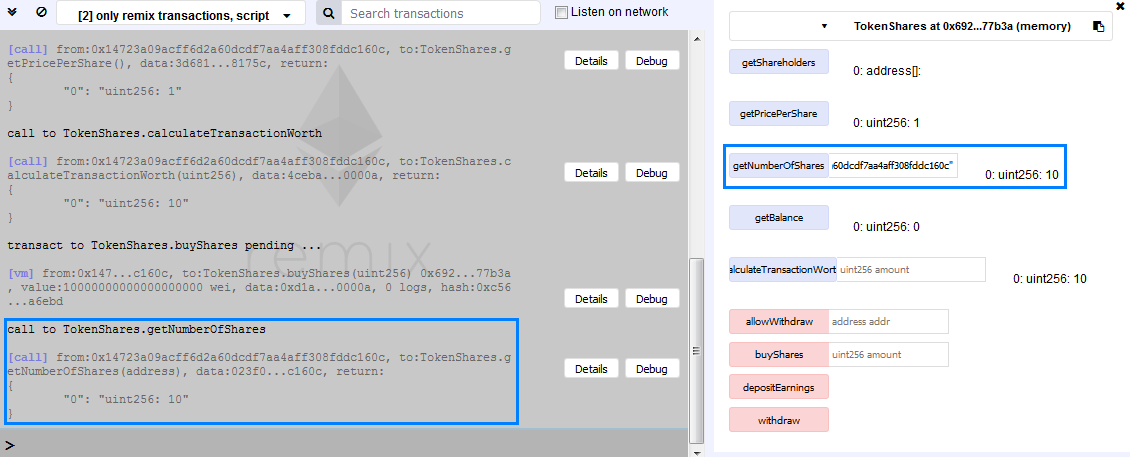
Make sure that you choose **ether** from the options in the drop-down menu. Then, in the field next to the **[buyShares]** write 10 and click the button.



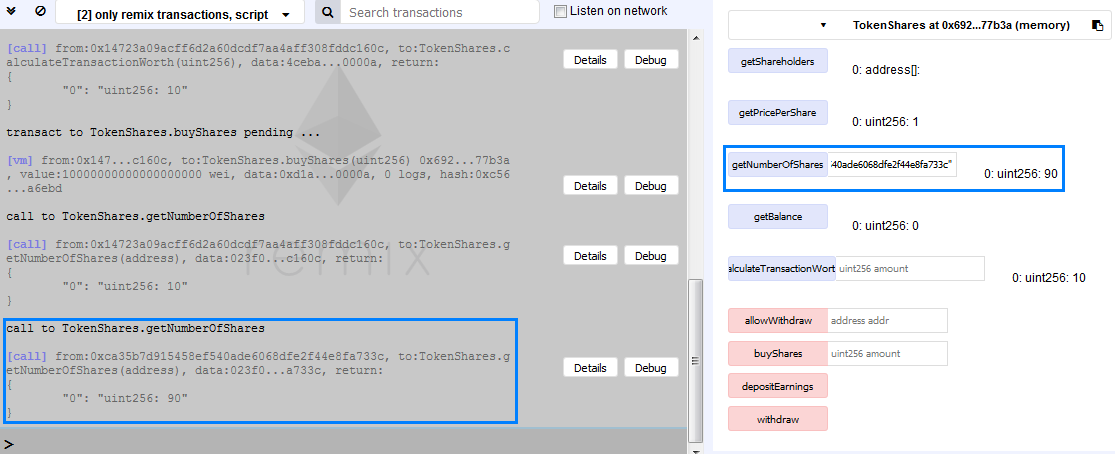
From the information given in the gray box it seems that the transaction was successful. Notice that the available ether in the account has decreased with little more than 10 ether due to transaction costs.

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Account.PNG

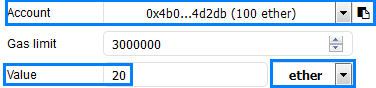
Now let’s check the number of shares of this address. Copy the address and place it in the appropriate field. You should see that 10 shares have been allocated to it:



Also, let’s check whether the number of shares that the owner holds has decreased with the same amount. Change back to the address that was used to create the account (remember **only** the owner can check the number of shares that an address holds, other users may check only their holdings), copy it and place it in the appropriate field. You should see something similar to the following screenshot:



Now let’s use a third address to buy 20 shares.



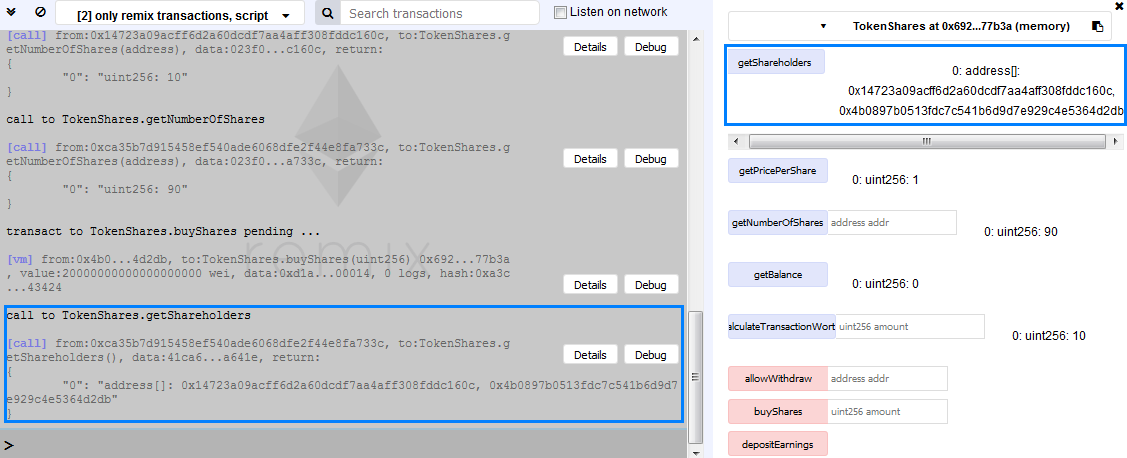
Write the amount next to **[buyShares]** and click the button.



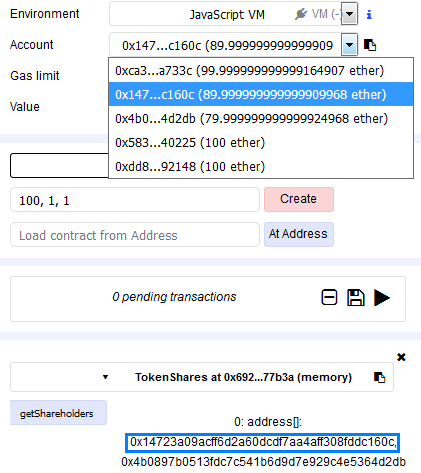
Again, verify that an amount of little more than 20 ether were spent:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Account 2.PNG

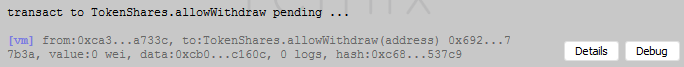
Now is the moment to check the addresses of the investors. To do this first we must change back to the address that created the contract and then click the **[getShareholders] button**. The result should look like the following screenshot:



You should be able to see the addresses of our two investors. Let’s take the first one and allow it to withdraw its dividend. To do this we first note the amount of ether that the address has before withdrawal:



Then, using the owner’s address, we copy and paste the address of the investor next to the **[allowWithdraw]** and click the button. In the gray box, you will get a feedback similar to the one in the next screenshot:

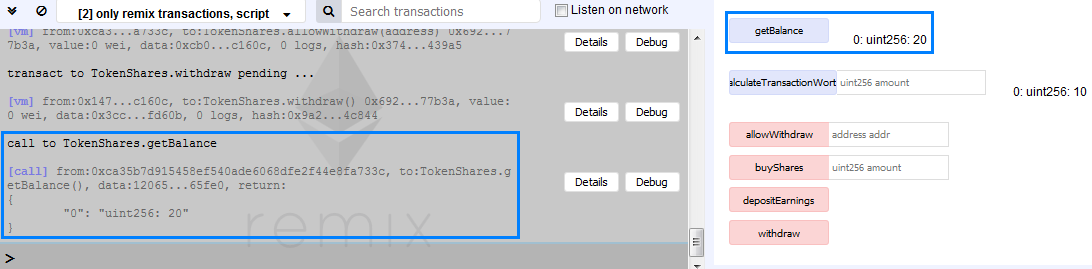


Now, from **Account** select the address that was just allowed to withdraw funds and click the **[withdraw] button**.

First, notice the amount of available ether after the call to **withdraw** – it should have increased with 10 ether (keep in mind that there are some transaction costs):

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Account Final Balance.PNG

Use the owner’s address and check the contract’s balance. You should see something similar to the following screenshot:



The result is 20 because we sold shares for 30 ether and then allowed an address to withdraw 10 ether.

Through the owner’s address, we will deposit 5 additional ether to the contract.

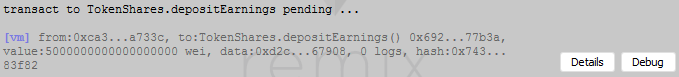
First, notice the available amount of ether:

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Account 3.PNG

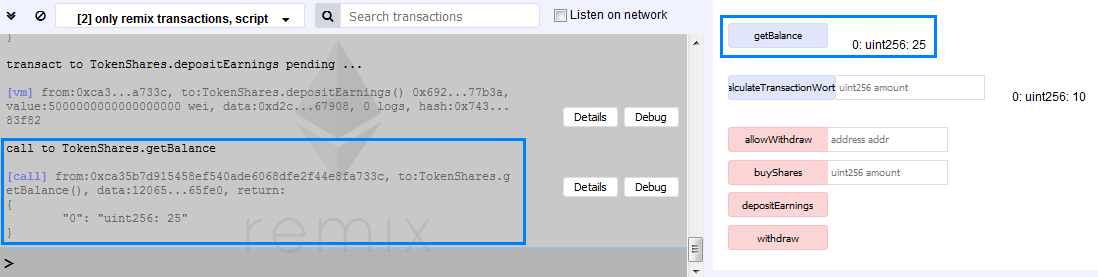
Write 5 next to in the field with label **Value** and make sure that from the drop-down menu **ether** is chosen.

C:\Users\drumenov\Documents\PhD\P\Block\Week 3\Day 2\Screenshots\Problem 8\Value.PNG

Then the **[depositEarnings]** **button**. The feedback from the gray box should look similar to the next screenshot:



Finally, let’s again check the contract’s balance. You should see something similar to the following screenshot:



That’s it! You just implemented a token with more advanced functionality![[15]](#footnote-15)

## More Advanced Token: Company Shares (2)

Write a contract that creates an initial supply of shares. Their price and the amount of dividend must vary. Only the contract’s owner must be allowed to allocate additional funds to the contract. Furthermore, only the contract’s owner must be able to allow withdrawal of funds. In addition, it must be possible only for the contract’s owner to see the addresses of the shareholders. Addresses must be allowed to purchase available shares only if they pay the required price. Shareholders may withdraw funds only if allowed to. Prices and dividends must use different denominations of ether. The representation of the denomination must be presented in a user-friendly manner. All conditions check must be made through the use of modifiers.

### Hints

This time you are on your own.

# What to Submit?

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

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

1. Since **Wei** is the smallest ether denomination you will not notice a big difference in the available ether in the used account. You may check the following link to see the different denomination of ether - <https://converter.murkin.me/> [↑](#footnote-ref-1)
2. Feel free to copy-paste the code from the previous problem but we advise you to manually write code as much as possible. [↑](#footnote-ref-2)
3. Or rather, it is rendered inaccessible. It is not possible to delete something from the blockchain in the usual manner. All ether that is send to such a contract is irretrievable. [↑](#footnote-ref-3)
4. You still can send ether to the second contract through by writing the required amount in the field next to the label **Value** and pressing the **[(fallback)] button**. [↑](#footnote-ref-4)
5. If you try to access the contract`s functionality after you terminate the contract you will get an error – try it. [↑](#footnote-ref-5)
6. You are advised to try rewriting the code so that the balance of the contract to be terminated is sent to another contract, not directly back to the originator. [↑](#footnote-ref-6)
7. We advise you to write an event with different parameter and do some additional tests. [↑](#footnote-ref-7)
8. For the purpose of the current exercise no checks are made whether a valid address has been entered. Hence, you are free to use as an input whatever you choose. In addition, remember to pass the variables in the appropriate format, otherwise you will get an error. [↑](#footnote-ref-8)
9. You are advised to try this as an additional exercise. [↑](#footnote-ref-9)
10. Be careful when you are verifying this - notice that there must not be any additional spaces. A single comma separates the data types. Were you to add an additional space (or any other symbol for that matter) you will get a different hash value. [↑](#footnote-ref-10)
11. Up to three parameters can be indexed. This way it is possible to search for the respective arguments. For additional information - <https://solidity.readthedocs.io/en/develop/contracts.html?highlight=topic#events> [↑](#footnote-ref-11)
12. You are advised to verify this - simply remove the keyword **indexed** and the name of the index, create anew the contract, pass it some values and check the result. [↑](#footnote-ref-12)
13. For additional information see <https://ethereum.stackexchange.com/questions/6840/indexed-event-with-string-not-getting-logged> [↑](#footnote-ref-13)
14. If you were the return, the value of **now** you would see that it is in the UNIX timestamp format. [↑](#footnote-ref-14)
15. We recommend to rewriting the code but for conditions to use **modifiers** and add **text that show in what denomination** are the quoted values. Furthermore, you may try adding a function that **returns the amount of dividend** that a share receives and a function that **calculates how much dividend an address is about to receive**, depending on the number of shares. You may even try to create a contract that pays **a varying amount of dividend**. [↑](#footnote-ref-15)