LESSON INTRO 4 – Your First Smart Contract

University of West Attica

Department of Electrical and Electronics Engineering

Ioannis Christidis

Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

This is a continuation of the Remix IDE introduction.

We will create our first SC using solidity and interact with it.

Understand the *read/write* functions.

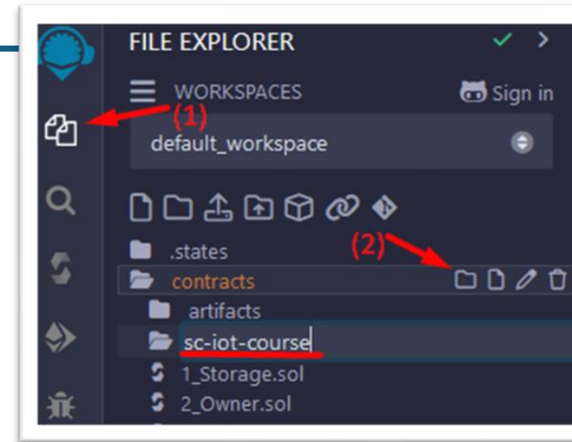
Understand the *state* in blockchain.

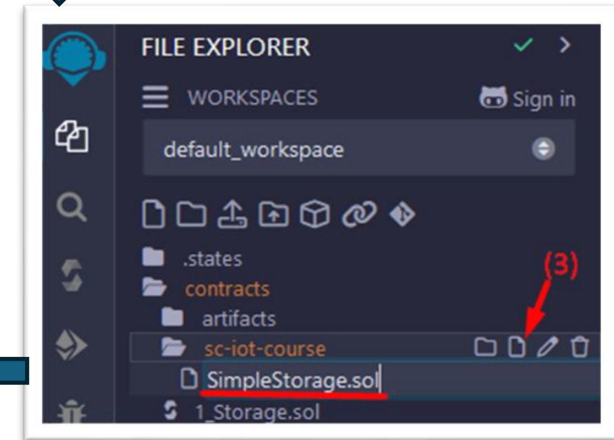
Understand *gas*.

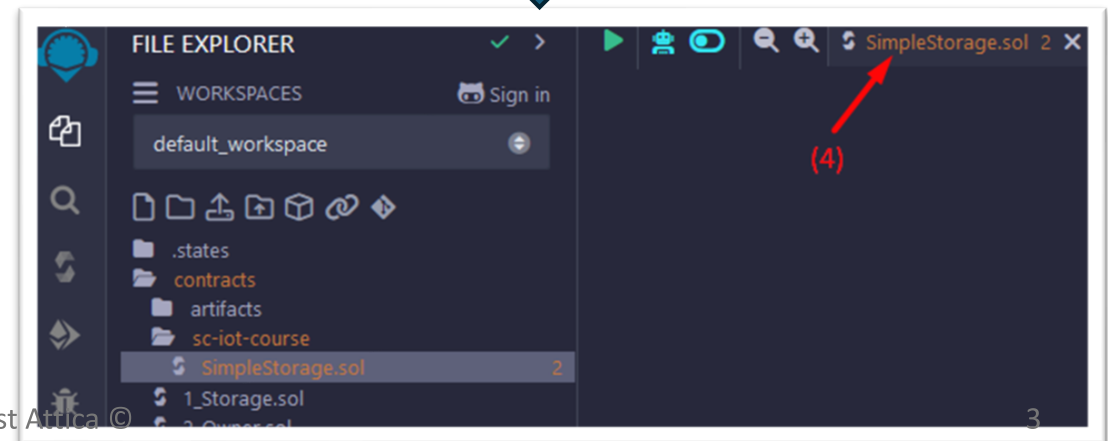
Understand how *transactions and reverts* work on blockchain.

How to write your first Smart Contract

1) To create your first SC, go into the *file explorer*.
2) In the `contracts` folder make a new folder called `sc-iot-course`.
3) Inside that folder create a new file and name it *`SimpleStorage.sol`*.
4) The new *file* will open in *code editor* as an emptry file.
5) Let's start coding.







Choose a *License*

In the new file that appears in the *code editor* we will write the first SC.

As of May 2020, every SC starts with the following line of code.

```
// SPDX-License-Identifier: {LICENSE}
```

SPDX stands for *Software Package Data Exchange* and it is used to declare the *software licence* of use. Specifying a license explicitly states how the code can be used, modified, and distributed.

The two most common licences are:

- **Copyleft GPL-3.0 License**: Any SC that interacts with or builds upon your *GPL-3.0* licensed SC must also be open-sourced and licensed under *GPL-3.0*.
- **Permissive MIT License**: SCs under the *MIT* License can be freely used and adapted without requiring derivative works to be open-sourced.

For now, we will choose the *Permissive MIT License* so type the following line of code.

```
// SPDX-License-Identifier: MIT
```

Specify the *Solidity Version*

The following line is also mandatory, and it specifies the *compiler version* that the SC is compatible with. Solidity introduce changes fast and some of them may affect your code, so it is important to specify the version of solidity that your code is inteneded to work with.

```
pragma solidity 0.8.26;
```

You can also specify *multiple versions* that you code is compatible, just be aware of breaking changes between versions.

```
pragma solidity >=0.8.2 <0.9.0;
```

There is also the option of specifing a version that is compatible with next versions that do not have breaking changes. This allows for more flexibility. In the example bellow the line signifies that the SC will be compatible with version `0.8.2` and up to the next version with a breking change (e.g. `0.9.2`)

```
pragma solidity ^0.8.2;
```

You can find more about the changes here.

In this course we will use version `0.8.26`.

Declare your *SC's name*

To start let's define an empty SC named `SimpleStorage`.

```
contract SimpleStorage {}
```

Similarly to how classes work in other programming languages, a SC is a collection of *functions and state variables*. When we deploy the SC in the blockchain, its functionality will be available to anyone *internally (inside the blockchain) and externally*.

The `SimpleStorage` is the most basic SC used to teach SC coding. What we will build is a very SC with the following features:

- A *state variable* named `favouriteNumber` that can store an *unsigned integer* as value
- A *setter function* that allows us to store an *unsigned integer* as value to `favouriteNumber`
- A *getter function* that allows us to see the value of `favouriteNumber`

Generally, solidity is a language that relies heavily on the concept of *getters and setters* to interact with SC data.

Declare a *State Variable*

Let's create a state variable inside the SC:

```
contract SimpleStorage {
    uint favouriteNumber;
}
```

`uint` stands for unsigned integer, meaning that it can only store values from *0 and above*.

Since we did not provide any initial value, it will take the default value which is `0`.

We could declare it with a starting value as:

```
uint favouriteNumber = 0;
```

In solidity, we avoid using *integers* and instead we prefer to use *unsigned integers*. There are a few reasons for it that **will be explained later**, on the course but to outline them:

- **Gas Efficiency**
- **Range of Values**
- **Overflow/Underflow**

Create a <u>*Setter Function*</u>

Let's create our first *function*:

```
function setFavouriteNumber(uint _favouriteNumber) public {
  favouriteNumber = _favouriteNumber;
}
```

This is a realy simple function that we call `setFavouriteNumber`.

It takes *one parameter* as input which is a `uint` we call `_favouriteNumber`.

The (_) before the *parameter name* is used to differentiate it from the state variable name. This just makes the code more readable.

`public` is the *visibility modifier* of the function. Every *function and state variable** has a *visibility modifier* in solidity. (*In case of `favouriteNumber`, since we did not specify the *visibility*, it is set to `internal` by default, however in a function it is mandatory to specify the *visibility*. We will explain `modifiers` later.)

Making this function `public` means that anyone can call this function.

`favouriteNumber = _favouriteNumber`: This line assigns the input `_favouriteNumber` to the `favouriteNumber` variable in the *SC's state*.

Function *Gas Cost*

****IMPORTANT****

Since this function ***changes the state variable*** it also **modifies the state of the blockchain**.

Functions that change the state of the blockchain COST GAS.

This is because, every time we change data on the blockchain, we are *writing to its state*, which requires computational resources and incurs a *gas fee*. This is also the case for any type of blockchain transaction like sending ETH between two accounts in the Ethereum blockchain.

Generally, when we are writing on the blockchain we are actually *making a transaction*.

**You will notice that we will use the terms read/write on the blockchain*

**Read means that we just view data from the chain (free)*

** Write means we are making a transaction (costs gas)*

We can think of **gas** as a unit that measures the computational effort required to execute operations on the blockchain. Every action, like storing data, performing calculations, or sending tokens, costs a certain amount of gas based on its complexity.

Function *Gas Cost* (2)

Purpose of Gas:

Gas fees incentivize *miners or validators to process transactions*, as these fees are paid to them for their work in *validating transactions*.

Gas Calculations:

Gas is paid in the native currency of the blockchain (e.g. ETH) by account that calls the function. The total gas cost of a function depends on two factors:

1) *Gas Amount*: The amount of gas an operation requires.

2) *Gas Price*: The price per unit of gas, which can vary based on network demand.

We will not learn about *Gas Amount* calculations in this course but there are development frameworks that calculate them automatically.

You can find the current *Gas Price* of a blockchain by googling it.

Function *Gas Cost* (3)

Key Takeaway:

Any SC that alters the state of the blockchain, creates a transaction and costs gas which is paid by the one calling the function.

When we are creating a SC, we need to be careful with functions that alter the state.

In the case of our SC, it is impossible to avoid making a transaction since we want to save our favorite number in the state (**VERY IMPORTANT :P**) but the cost will be very small since we are only making one change in the state.

Create a <u>*Getter Function*</u>

Let's continue by making a function that retrieves our favorite word.

```
function getFavouriteNumber() public view returns (uint) {
  return  favouriteNumber;
}
```

The function above is called `getFavouriteNumber` and it returns our favorite number.

This function get <u>no parameters</u> as input.

This function also has the `public` <u>*visibility modifier*</u> so anyone can call it.

The `view` keyword is also a <u>*modifier*</u>. It specifies that this function is "**read-only**". This means that we can be sure that it will not alter the state so we can call it <u>*without paying any gas*</u>. This is true because we just want to read the `favouriteNumber` state variable, but we do not alter it.

When a function just reads from the blockchain we should mark it as `view`.

Next, this function specifies what is expected to be returned `returns (uint)`, so we are expecting a `uint` to be returned.

Finaly, with the `return` keyword we retrieve the favorite number.

Coding Completed

And now our SC is completed. We have created:

1) A *state variable* that stores an unsigned integer called `favouriteNumber`.
2) A *setter function*, called `setFavouriteNumber`, that changes the state variable.
3) A *getter function*, called `getFavouriteNumber`, that returns the state variable.

Your SC should look similar to the following snippet:

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.26;
// Smart contracts are mainly built with the getter and setter functionality
contract SimpleStorage {
    // This is unsigned integer. Since its value is not defined it is considered 0
    uint favouriteNumber;
    // This is the setter function that sets the number to a specific value
    function setFavouriteNumber(uint _favouriteNumber) public { // writes on the blockchain (costs gas)
        favouriteNumber = _favouriteNumber;
    }
    // this is the getter function that retrieves the value of the number
    function getFavouriteNumber() public view returns (uint) { // reads from the blockchain
        return  favouriteNumber;
    }
}
```

Compile and Deploy the Contract

While you are on the `Simplestorage.sol` go to the compiler from the sidebar, and in the compiler search for version `0.8.26+commit.8a97fa7a`.

Click the button *Compile SimpleStorage.sol* and if everything went well you will have a successful compile (tick icon on the compiler icon on the sidebar).

Navigate to *Deploy and Run Transactions* from the sidebar and choose the environment *Remix VM (Cancun)*.

Notice that in the *Accounts* field each *account has an address and 100 ETH*.

Make sure that `SimpleStorage` appears in the *Contract* field and click the *Deploy* button.

(If you could not follow any of these steps check LESSON INTRO 3 – Intro to Remix)

Congratulation!!! You just wrote, compiled and deployed your first smart contract!!!

Now the next step is to interact with it.

Results of Deployment (1)

You will notice that now that we deployed our SC, **three** things happened.

The **first** is that if you look at the *accounts* field you will see that the *active account* (the one that appears on the dropdown while it is closed) does not have *100 ETH* as it was supposed to, but instead it has *99.999...7 ETH*. This is because *when we deployed the SC, we altered the state of the blockchain by making a transaction* and that, *costs gas*. The *active account* was the one that deployed the SC, so it was the one that payed the gas cost.

The **second** is that in the *console* (*under the code editor*) a tick icon appeared with some weird stuff similar to this:

If you click on it, it will open up revealing the full information. This is *the information of the transaction* that happened when the SC was deployed.

Results of Deployment (2) - Output Explanation

Let's explain the fields that appeared in the transaction details.

`status`: Indicates if the transaction was successful.

`transaction hash`: The _unique id_ for the transaction inside the block.

`block hash`: The _block identifier_ that was mined (successfully added to the blockchain).

`block number`: The _number of the block_ in which the transaction was mined.

`contract address`: Like the accounts we use, **every SC has its own address in the blockchain**. This is important because we will use the SC address to interact with it.

`from`: The address of the account that made the transaction (the active account that deployed the SC)

`to`: Now, this is a bit tricky! Go to the next slide to see more!

Results of Deployment (3) - Output Explanation

Normally in a transaction in the real world with actual money we have three fields:

1. `from`: Who sends money
2. `value`: How much money
3. `to`: Who recieves the money

The same logic, but with a few more info, applies in the blockchain when we call a function of a SC.

During the deployment of a SC, we pay gas for the transaction to be executed but this is not the `value` from above, just the cost of the transaction, so the `value` here is `0`.

You will notice that the `value` does not appear in the details of the transaction, but it does appear here.

```
[vm] from: 0x5B3...eddC4 to: SimpleStorage.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xec0...66386
status              0x1 Transaction mined and execution succeed
transaction hash    0xec0403e2cb73865cb24d282a558c1bed0452ab2226b6039789cbfe3d34966386
block hash          0x6e8d12457715c9f8922619afd734c2321f6f9a53cc4ec2db34b3d69f51462175
```

University of West Attica ©

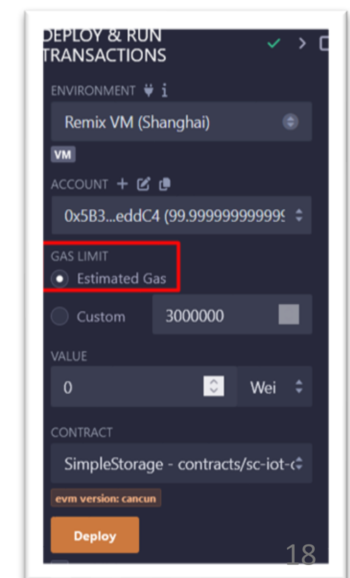
Results of Deployment (4) - Output Explanation

So, we do have a `from` which is the account we used to deploy the SC, but we also have a `to` property. So, the question is to who we send `0` `value`?

The answer is to the SC that is currently being deployed. Technically *when we send `value` to a SC, it recieves it through a function* (even if it has not been specified during the coding, but we will explain this more later). But since it has not been deployed yet, we can send it to the `constructor function` which is a function that runs only once during the SC's deployment. We will also learn more about this later. As a result, on the `to` property we see `SimpleStorage.(constructor)`.

Now let's continue with the rest of properties.

`gas`: The *maximum amount of gas allocated for the deployment*. This is also known as `gas limit` and it is set by the `deployer` (the account that deploys the SC) but in our case it was estimated by remix. You can see that in *Deploy and Run Transactions* sidebar. This limit acts as a cap, ensuring that the transaction doesn't consume an unlimited amount of gas.



Results of Deployment (5) - Output Explanation

`transaction cost`: The *actual cost of the transaction in gas*. If the transaction completes before reaching the gas limit, the remaining gas is refunded. However, if the transaction runs out of gas before completion, it will fail and revert any changes made up to that point.

`execution cost`: This is part of the `transaction cost` and it is the *actual gas required to perform the function* called. In our case since we deployed the SC it will be the `constructor`.

`input`: This is the *raw data sent with the transaction*. *In this case it includes the SC's bytecode* (binary representation of the SC's logic). *We will not dive deep into this consept in this course so do not worry about this for now. It is not mandatory for you to understand it.*

`output`: The output in this case is *the bytecode generated by the constructor*. This *bytecode is stored on the blockchain as the SC's code*, making it available for interaction.

`decoded input`: The *data we added as input (parameters in a function call) in human readable form*. Since we did not provide any, it is empty.

`decoded output`: Typically *displays results or return data from a function call*. Since the constructor does not return any data it is also empty.

`logs and raw logs`: *Sometimes SC functions emit data without returning the data with the use of events* (we will talk about events later). *They are shown here in human readable and hexadecimal form*. Once again, the constructor does not emit any data so both fields are empty.

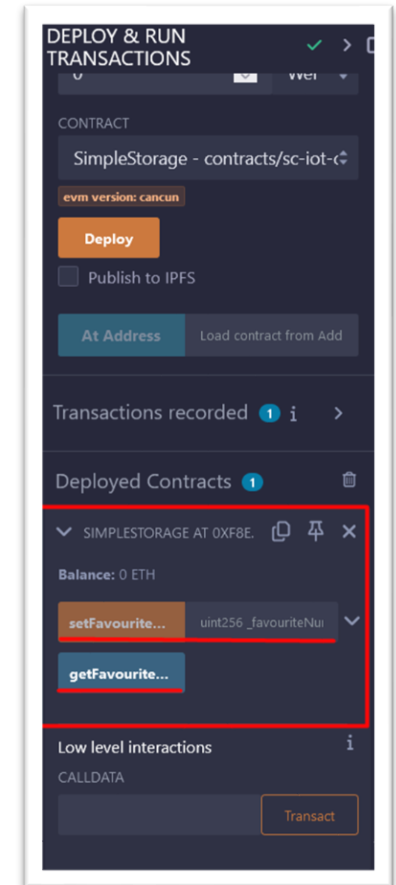
Results of Deployment (6)

As we mentioned all the way back in slide 15, **three** things happened when we deployed our SC. We talked about two of them so now we are going to talk about the **third** and the most exciting one. In *Deploy and Run Transactions* sidebar if you scroll down, you will notice a field named *Deployed SCs* and in it you will find our newly deployed `SimpleStorage` SC with its `address`. If you click on it two buttons will be revealed. These are *buttons that can be used to interact with our smart SC by calling its two functions*.

You will also notice that they have different colors (in the image they appear as orange and blue) and the reason is to differenciate functions that alter the state from the ones that only view from the blockchain (orange for writing on the blockchain, blue for reading). You will also notice that next to the orange button there is an input field. This input field is for the user to add an input.

Blue Button => `getFavouriteNumber`

Orange Button => `setFavouriteNumber` with an input field to add our favourite number.



Interacting with a Smart Contract (1)

First let's call the `getFavouriteNumber` by pressing the blue button.

We expect it to return `0` because we haven't set a favourite number yet.

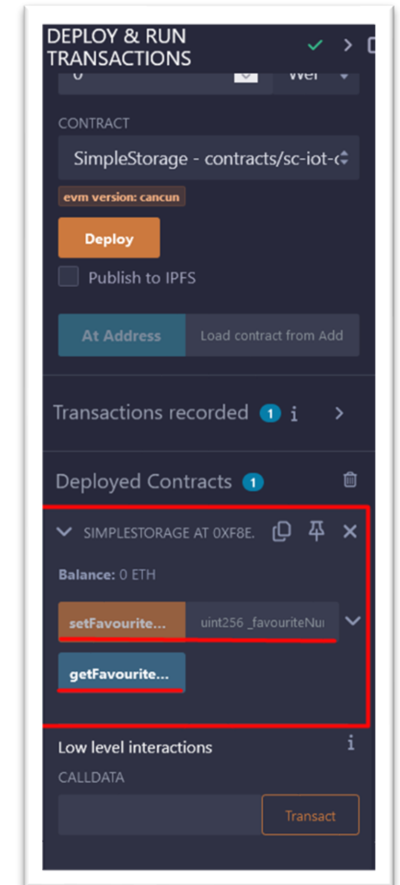
Under the button you will see the `0: uint256: 0`. The first `0` means the 1st item that is returned. The `uint256` is the `uint` that we expected as output in the function, but this will be explained more in the next lesson. The second `0` is the value of `favouriteNumber`

Now if we check the *console*, we will see a log similar to the following:

```
call to SimpleStorage.getFavouriteNumber

CALL    [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SimpleStorage.getFavouriteNumber() data: 0xe03...aefbf
```

If you click it to expand you will notice some differences from the previous log. The most important is that *this is not a transaction*, it is just *a call to view data* from the SC, so *it does not cost gas*. Let's continue on the next slide.



Interacting with a Smart Contract (2)

First, we notice that an `execution cost` is mentioned but *this cost would only be paid if the call happened from another SC* (YES, SCs can interact and we will see that on the next project). For us it is free! :P

The `to` field, now reads `SimpleStorage.getFavouriteNumber` which is indeed the function that we used.

The `input` we see here, contains data that allows the identification of the function we used to be executed by the EVM. We will not dive deep into this consept in this course so do not worry about this for now. It is not mandatory for you to understand it.

The `output` is `` `0x00..0` ``which is the `0` equivelent of a `` `uint` `` (more about this in the next lesson).

`decoded input` is empty since we do not have any parameters.

`decoded output` is what also appeared under the button we used to interact with the SC.

We still do not have any `logs` and `raw logs`.

University of West Attica ©

Interacting with a Smart Contract (3)

Now, let's add a favorite number. For this example, we will use the number `305`. If we add it in input field next to the `setFavouriteNumber` button and then press the button, we will see that a new log was added in the *console* similar to this.

```
transact to SimpleStorage.setFavouriteNumber pending ...

[vm] from: 0x5B3...eddC4 to: SimpleStorage.setFavouriteNumber(uint256) 0xf8e...9fBe8 value: 0 wei data: 0xacc...00131 logs: 0 hash: 0x7a4...51d08
```

This was a transaction since we change the state and that means that we paid some gas. If we inspect the log, we will notice a few differences from the transaction that we did when we deployed the SC.

We can see that we have different `transaction hash` and `block hash` and also the `block number` has gone up by `1`. This is because we are currently using a *VM blockchain* so there are no other transactions on it. *In a normal network the second transaction would still have a bigger `block number` but not necessarily by `1`*.

Continue on the next slide..

Interacting with a Smart Contract (4)

The `to` value is `SimpleStorage.setFavouriteNumber` which is the function that we used.

We can see the information about `gas costs` similarly to the deployment of the SC.

The `input` we see here, looks similar to the previous we saw, but it also contains information about the *parameters of the function*. In the `decoded input` where we have our `parameter "uint256 _favouriteNumber": "305"`. Once again, we will not dive deep into this consept in this course so do not worry about this for now. It is not mandatory for you to understand it.

Since this function does not return anything, we have an `output : ` **`0x`** which means that *nothing is returned*. This is also the case for the `decoded output`.

Finally, once again we do not have any `logs` or `raw logs`.

Now, if we press the button `getFavouriteNumber` again we will see the number `305` so we successfully added our favorite number in the blockchain.

Interacting with a Smart Contract (5)

Finally, let's try to add an input that is not a `uint` to see how the SC will behave.

For example, if you try to add a `string` like `"abc"` you will receive an error in the console.

```
transact to SimpleStorage.setFavouriteNumber errored: Error encoding arguments: Error: invalid BigNumber string (argument="value", value="aaa", code=INVALID_ARGUMENT, version=bignumber/5.7.0)
```

In this case, the transaction will revert with `"Error: invalid BigNumber string"`. The key point here is that *the input needs to match the parameter type else the transaction reverts*.

The case is similar if you try, for example, to add a negative number like `-13`

```
transact to SimpleStorage.setFavouriteNumber errored: Error encoding arguments: Error: value out-of-bounds (argument=null, value="-12", code=INVALID_ARGUMENT, version=abi/5.7.0)
```

Now the transaction reverts with `"Error: value out-of-bounds"` since we only allow unsigned integers.

IMPORTANT: *When a transaction reverts it undoes any changes that were attempted during execution, and any gas used up to that point is still consumed (minus any refunds for unspent gas). This is done to ensure the blockchain remains in a consistent state.*

Outro

Congratulations, you finished the most difficult lesson.

In the next lesson we will learn the different types of variable in solidity.