LESSON SC 6 – *Access Control*

University of West Attica

Department of Electrical and Electronics Engineering

Ioannis Christidis

Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

In this lesson we will think about *access control* concepts.

And we will start with this question:

Who can register a car?

As we mentioned before, only the owner should be able to register cars on our SC **however, right now, anyone can do it**. Test it yourself, by deploying the ICR SC, change the account and try to register a car. You will see that anyone can register a car, which is something that we do not want.

A SC deployed on the Blockchain is publicly accessible and anyone can interact with the SC by calling its functions.

So, how can we fix this?

The `if` *statement* approach (NOT GOOD)

We want to implement some logic in the `registerCar` function that *checks if the `msg.sender` is the `owner` of the SC*. That way *we will only allow the owner to register cars*.

Let's see an example:

```
    function registerCar(address _mc, uint256 _price) public {
        if (msg.sender == owner) {
            // register car logic here
        }
    }
```

We can, easily, do this with an *`if` statement*, **BUT** there is a better and more efficient way.

The `if` statement will check if `msg.sender` is the `owner` and if they are **not**, it will not run the register car logic, but the transaction will still happen even if we did not alter the state. Basically, **we wasted gas for no reason**. Instead, we want to find another way.

The `require` approach

We talked about reverts of transactions before and we can use them to our advantage here. We can write code that *checks if the `msg.sender` is the `owner` and if it is not the transaction will revert saving us some gas.*

We can do this by using a `require` statement.

```
    require(msg.sender == owner, "msg sender is not the owner");
```

This `require` will first check if `msg.sender == owner`:

`true`: the function will continue normally.

`false`: *the function will not continue because the transaction will revert*. You can also add a `string` message like `"msg sender is not the owner"` to know why the function call failed.

It is generally a good practice to use `require` statements at the start of the function because as we learned if a transaction reverts the gas consumed up to the point of revert will still be lost.

This is a good approach, but we can be even more efficient.

The `custom error` approach

From `version 0.8.4` and above *solidity added the custom error functionality* which, as the name suggest, *allows us to create custom errors*.

```
error errorName(params);
```

Custom errors work similar to `require` statements (revert the transaction) but they are better for the following reasons:

- We do not need to add a `string` as error message which is the **WORST** type to use in solidity as they aqcuire more space in storage and consume more gas when used.
- We can add parameters which can be information about the reverts (very useful during the testing).

As a best practice we want custom errors to follow this structure for naming:

ContractName__ErrorName(params) but this is not mandatory. It helps however in case we have multiple SCs (which we will have later) so that we know from which came the error.

Custom errors in ICR

Let's create a `custom error` for `registerCar` function. The custom errors can be be defined inside or even outside of our SC. However, we will define it inside the SC above our `enum Status`.

```
error ICR__IsNotOwner(address _notOwner);
```

In the line above, you see the name of the error `ICR__IsNotOwner` which follows the best practices and as a parameter we add an `address`. This will be the `address` of the account that called our `registerCar` function AND is not the `owner` of the SC.

Notice that since the errors are defined outside of functions, they *can be used in multiple functions*, similar to how we can call a state variable from any function.

Now, we need to use this error in the `registerCar` function. We want to revert the function call only if the `msg.sender` is not the `owner` so we can add it inside an `if statement` at the start of the function as shown below.

```
    function registerCar(address _mc, uint256 _price) public {
        if (msg.sender != owner) {
            revert ICR__IsNotOwner(msg.sender);
        }
       // Rest of register car logic here
    }
```

Actions of Entities

Now, let's talk about who can change the `Status` of a car. For example, if I was a user of the app, should I be able to change the `status` of a car to `UNAVAILABLE`? Probably, no. So, let’s let's define the actions for each entity.

USER

- *Rent Cars*: Pay some ether to change a car's status to occupied only if the car is available for rent.

`(AVAILABLE => OCCUPIED)`

MC

- *Make cars available after a rental so they can be rented again.*

`(OCCUPIED => AVAILABLE)`

OWNER

- *Register Cars* (Already covered)
- *Change cars' status to available (for rent) or unavailable (for service or something similar) when the car is not occupied (not rented by a user).*

`(UNAVAILABLE <=> AVAILABLE)`

Before Functions

Each of the bullets on the previous slide represent a function that we can build. `registerCar` is already created so *we must change the* `changeCarStarus` *function to 3 different functions one for each entity* (we will see later why we decided to split them instead of adding them all in one).

Before we start creating these functions, let's look the function `changeCarStatus`.

```
    function changeCarStatus(uint256 _carId, Status _status) public {
        if(_carId < nextCarId){ // notice this line
            Car storage car = cars[_carId];
            car.status = _status;
        }
    }
```

As you can see, the first thing we do in this function is check the ID of the car we want to change its `Status` is less than `nextCarId`.

We did that to make sure that the car is registered before altering the state of the SC, but even if it is not registered the transaction will still happen.

So, to avoid that, we will create a new custom error:

```
  error ICR__InvalidCar(uint256 _carId);
```

Modifiers

This error we created is very useful since we are going to use it in the 3 functions that change the status of cars but instead of adding it with an if statement every time, we will add it on a `custom modifier`.

`custom modifiers` *are functions that can be applied to other functions in a SC. They are used to modify the behavior of functions or enforce certain conditions before the function's logic is executed.*

Bellow you will see a custom modifier for our SC named `carIsValid` that takes as input the ID of a car. Then, it checks if it is bigger than `nextCarId`. If it is, it will revert with our custom error, else it will continue with the function.

```
    modifier carIsValid (uint256 _carId) {
        if (_carId >= nextCarId) {
            revert ICR__InvalidCar(_carId);
        }
        _; // placeholder where the function logic will be inserted
    }
```

You can *define a* `custom modifier` *inside the SC using the modifier keyword.* *The `` `_;` `` symbol defines where the function logic wil be inserted. In our example it will first run the check and then the actual function*.

Rent Car Function with Modifier

Now let's create one of the functions that changes the status. We will start with the easiest one which is for the user. We want the user to be able to rent cars which means changing the Status from AVAILABLE to OCCUPIED.

We will call this function rentCarand it will take as input the ID of the car to be rented.

```
    function rentCar(uint256 _carId) public carIsValid(_carId) {}
```

Notice that after *the public keyword*, which *is also a modifier but not custom*, we added the carIsValid modifier and we add as input the ID of the car for rent.

We will also create a custom error that reverts if the current status of the car is incorrect. For example, for the rentCar function we want the car status to be AVAILABLE.

```
 error ICR__CarIsNotInTheCorrectStatus(uint256 _carId, Status _status);
```

If it is, we will change its status to OCCUPIED.

```
    function rentCar(uint256 _carId) public carIsValid(_carId) {
        if (cars[_carId].status != Status.AVAILABLE) {
            revert ICR__CarIsNotInTheCorrectStatus(_carId, cars[_carId].status);
        }
        Car storage car = cars[_carId];
        car.status = Status.OCCUPIED;
    }
```

Microcontrollers' Change Status Function

Now, let's create the function that changes the `status` for the microcontroller.

First, we need to check if the car is valid, and we can do that by adding our custom modifier `carIsValid`.

Next, we need to check if the `address` of the car's microcontroller (`car.mc`) is the same as the `msg.sender.` if it is, change the car's `status`.

Try to make a custom error for that.

Then, we need to check if the status of the car is `OCCUPIED`.

And if all these checks pass, we change the car's status to `AVAILABLE`.

```
    function changeCarStatusMc(uint256 _carId) public carIsValid(_carId) {
        if (msg.sender != cars[_carId].mc) {
            revert ICR__IsNotMc(_carId, msg.sender);
        }
        if (cars[_carId].status != Status.OCCUPIED) {
            revert ICR__CarIsNotInTheCorrectStatus(_carId, cars[_carId].status);
        }
        Car storage car = cars[_carId];
        car.status = Status.AVAILABLE;
    }
```

Owner Changes Status Function

Similarly, we create the function of the owner.

Try to do it without copy, pasting the code for training.

TIP: We already have all the custom errors we need for that.

```
    function changeCarStatusOwner(uint256 _carId) public carIsValid(_carId) {
        if (msg.sender != owner) {
            revert ICR__IsNotOwner(msg.sender);
        }
        if (cars[_carId].status == Status.OCCUPIED) {
            revert ICR__CarIsNotInTheCorrectStatus(_carId, cars[_carId].status);
        }
        Car storage car = cars[_carId];
        if(car.status == Status.AVAILABLE) {
            car.status = Status.UNAVAILABLE;
        } else {
            car.status = Status.AVAILABLE;
        }
    }
```

Outro

Good job! This was a hard lesson.

We learned about *access control concepts* and saw a few ways to implement them:

`If/then/else statements`

`Require statements`

`Custom errors`

`Custom modifiers`

On the next lesson, we will make the `rentCar` function `payable` which means, that the users will have to pay ETH to call it.