LESSON SC 7 – Payable Functions

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 learn how to make a function `payable` in a SC, enabling it to accept ETH.

In the last lesson we created a function called `rentCar`. This function can be used by users to rent cars, but we did not add the payment functionality. So, we will do this now.

Payable Functions

To enable a SC to accept ETH, we can add the `payable` *modifier in a function.* So, if a user calls this function, they can also *send ETH to the SC.*

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

But what we want in the `registerCar` function, is to pay the `price` that the `owner` defines for each car. To do that, we need to check if the ETH the user sends to the SC is not less than the price of the car, so we can create a custom error for that.

```
error ICR__NotEnoughEther(uint256 _price);
```

The global variable `msg.value` *represents the amount of Wei (the smallest unit of Ether) sent with a transaction.*

In the `rentCar` function, we can check if the `msg.value` is less than the car's price. If it is the function will revert with our custom error else, it will proceed to the rest of the logic.

```
        if (msg.value < cars[_carId].price) {
            revert ICR__NotEnoughEther(msg.value);
        }
```

Trying Payable functions

Let's test this on Remix IDE.

Compile and deploy the ICR SC.

With the owner account, register a new car using the registerCar function:

The new car should have a price that is not 0 (e.g. 10). This means that the price of the car will be 10 wei or 10*10^(-18) ether.

```
// 1 ether = 1_000_000_000_000_000_000 wei
// 1 wei = 0,000_000_000_000_000_001 ether
// 10 wei = 0,00_000_000_000_000_001 ether
```

For the mc address choose one address (other than the owner's) from the *Accounts* list.

Call the function changeCarStatusOwner to make the car AVAILABLE (for car ID add 0 which is the ID of the car we just registered).

Now let's call the rentCar function to rent the car (with ID 0) without adding any ETH and we will get this error.

To differentiate the payable functions their buttons are red (like the rentCar button)

```
Error provided by the contract:
ICR__NotEnoughEther
Parameters:
{
 "_price": {
  "value": "0"
 }
}
```

Trying `Payable functions` (2)

In the *VALUE* field on *RUN AND DEPLOY TRANSACTIONS* you can add the amount of ETH you want to pay.

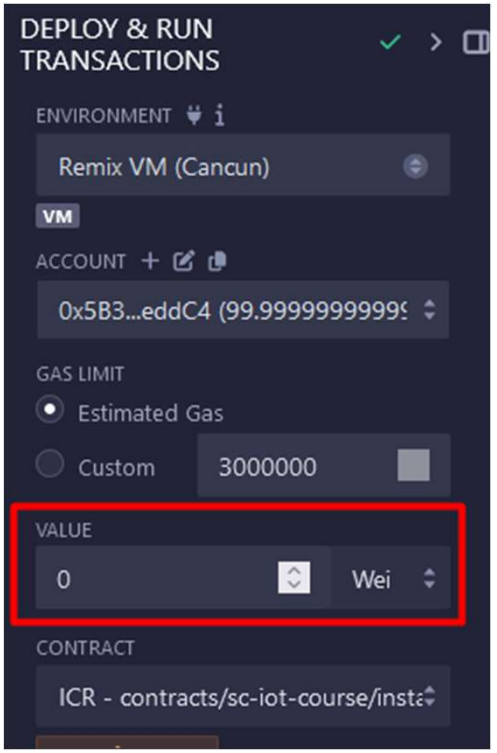
You can add *VALUE* in `Wei, Gwei or Ether`.

After you add an amount, press the `rentCar` button with car *ID = 0*.

Try paying more, less or equal to the car's price to see the results. Just note that every time the transaction passes the `status` of the car changes to `OCCUPIED`, so you need to change it back to `AVAILABLE` with the microcontroller's account by calling `changeCarStatusMc`.

Now, consider what will happen if *the owner of a car register it with 0 price by accident*. *They will not be able to change the price of the car because we did not provide such functionality.*

We are going to add a bit more functionality to the `registerCar` function that checks if the price inserted is reasonable.



Constant Variables

We will add a minimum `0.1 ETH` and maximum `10 ETH` limit on car prices.

We will assume that these limits will NEVER need to change.

We can add them in our SC as `constant variables`

A `constant variable` in Solidity is *a state variable whose value is fixed and cannot be modified after it is initialized*. *Constant variables are stored directly in the SC's bytecode instead of storage, which reduces gas costs*.

```
    uint256 public LOWER_PRICE_LIMIT = 0.1 ether; // 0.1 ether = 10^17 wei
    uint256 public UPPER_PRICE_LIMIT = 10 ether; // 10 ether = 10 * 10^18 wei
```

We will create two new errors and implement them in the `registerCar` function with `if statements` to check if the `_price` added as input is within bounds.

```
    function registerCar(address _mc, uint256 _price) public {
        // other checks
        if (_price < LOWER_PRICE_LIMIT) {
            revert ICR__PriceIsTooSmall(_price);
        }
        if (_price > UPPER_PRICE_LIMIT) {
            revert ICR__PriceIsTooBig(_price);
        }
        // rest of the functionality here
    }
```

Outro

Good job!

We learned about a few more things about:

`Payments`

`Msg.value`

`Constant variables`

Now on the next lesson we will talk about `events`.