

LESSON SC 10 – Withdraw Ether – Time in Blockchain

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:

- 1. how to withdraw ETH
- 2. how time works in blockchain



Low Level Call

In the ICR SC, we have the function rentCar that users can use by paying ETH to rent a car. This ETH is stored in the SC. If the SC is deployed as is, there will be NO WAY for us to retrieve that ETH. We need to create a function for the owner to retrieve the ETH from the SC. There are multiple methods to do that but, in this case, we will use a <u>low level call</u>.

In Solidity, the address variable has a method named call() through which you can call every function you want, and if the address implements it, it will be triggered. It bypasses Solidity's type safety mechanisms, giving developers more flexibility but also posing significant risks.

<u>No Type Checking</u>: The call function does not enforce type checking of the arguments or return values.

<u>No Reversion Propagation</u>: Unlike higher-level Solidity functions, a low-level call does not automatically propagate reverts.

This means that we need to manually check whether the call was successful by examining the boolean value it returns.

Withdraw ETH



Let's create a function called withdraw, that only owner can use to withdraw all the ETH from the SC.

```
function withdraw() external {
    if (msg.sender != owner) {
        revert ICR_IsNotOwner(msg.sender);
    }
    // call
    (bool callSuccess,) = payable(owner).call{value: address(this).balance}("");
    if (!callSuccess) {
        revert ICRRegistry_WithdrawFailed();
    }
}
```

In the function above, after we check if msg.sender is the owner we see a weird line of code. Let's explain it.

(bool callSuccess,): There are some functions that return more than one result like the *example* bellow.

```
function returnMoreThanOne() public pure returns (uint a, uint b, uint c){
    a = 0;
    b = 0;
    c = 0;
}
```



Withdraw ETH(2)



In another function we can get the returned values of returnMoreThanOne function like this:

```
function addABC() public pure returns (uint res) {
   (uint a, uint b, uint c) = returnMoreThanOne();
   res = a + b + c;
}
```

And if we only want some of the returned results we can do:

```
function addAC() public pure returns (uint res) {
    (uint a, , uint c) = returnMoreThanOne();
    res = a + c;
}
```

So back to the weird line of code:

(bool callSuccess,) = payable(owner).call{value: address(this).balance}("");

The low-level call returns 2 values, but we only need the first, which is a bool, <u>that is</u> <u>true if the call was successful</u>.

payable(owner) : this is called <u>typecast</u>, and <u>we can use in almost every variable type to change</u> <u>it to another, if applicable</u>. In this case we are changing an address to an address payable (check LESSON INTRO 5 - Variables) and the reason we do this, is because, it is the only way for an address to accept ETH.





Withdraw ETH(3)

.call{}(""): this is the low-level call function we discussed. In this case we are using it to send ETH to the owner address.

value: address(this).balance: the amount of ETH we send is equal to the value. The value is equal to the balance of "this" address which is our SC. The "this" keyword can be used to define our SC's address without it being deployed yet.

So essentially, the line bellow says, "*return true if the transferring of all the ETH, "this" SC (ICR) has, to the owner was successful, else return false*".

(bool callSuccess,) = payable(owner).call{value: address(this).balance}("");

After that, we *check if the call was successful*, and if it wasn't, we revert with a custom error.

if (!callSuccess) {
 revert ICRRegistry_WithdrawFailed();

<u>Time in Blockchain</u>



At the description of ICR project, we mentioned that each car will be rented for one day. This means that after a user rents a car, the microcontroller should wait 24 hours and then call the function changeCarStatusMc to make the car AVAILABLE for rent. But to ensure that 24 hours have passed, we can check it inside the blockchain.

The block.timestamp property provides the <u>timestamp of the current block in seconds since</u> <u>the Unix epoch (January 1, 1970, at 00:00:00 UTC).</u>

At the time that this course is written the current block.timestamp = 1735994708 seconds.

block.timestamp is an unsigned integer (uint) so we can use it in arithmetic operations.

There are time units in solidity such as seconds, minutes, hours, days and weeks.

- 1 == 1 seconds
- 1 minutes == 60 seconds
- 1 hours == 60 minutes
- 1 days == 24 hours
- 1 weeks == 7 days



Implement Time in ICR

In our SC we will create a constant variable named RENT_TIME.

```
uint256 internal constant RENT_TIME = 1 days;
```

Inside the Car struct we will add a new uint256 called timeOfLastRent. This will be updated every time a user rents a car.



Remember to update the registerCar function so that it also adds the timeOfLastRent. Since the car has not been rented yet, add zero.

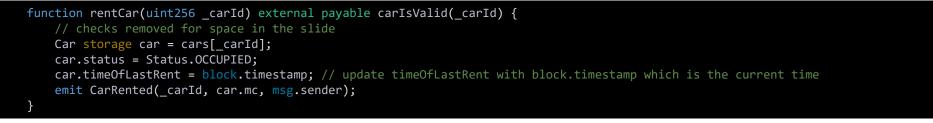
```
function registerCar(address _mc, uint256 _price) external {
    // checks removed for space in the slide
    Car memory car = Car(_mc, _price, Status.UNAVAILABLE, 0); // Added the 0 after the Status
    uint256 currentCarId = nextCarId;
    cars[currentCarId] = car;
    nextCarId++;
    emit CarRegistered(currentCarId, _mc);
}
```

Implement Time in ICR Functions



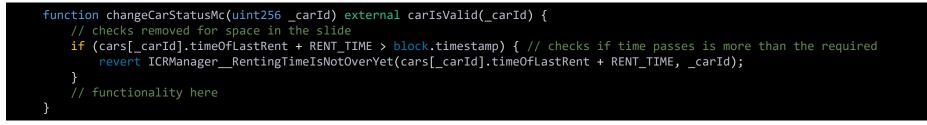
We also need to change the rentCar function to update the timeOfLastRent.

Here we will use the block.timestamp we mentioned before to get the current time.



In the changeCarStatusMc function we can check if the timeOfLastRent + RENT_TIME (24 hours) is less than or equal to the current time or block.timestamp:

- true: the function will proceed
- false: the function will revert with a custom error.





Outro

Be Careful when using timestamp!!

Miners or Validators can slightly adjust the timestamp of a block to gain advantages in specific scenarios. Avoid using block.timestamp for critical decisions (e.g., randomness).

Time on the blockchain is approximate due to network latency and block intervals. Do not rely on it for highly precise timing requirements.

This was a lot of information!

We learned how to:

- withdraw ETH from a SC
- use low level calls.
- use block.timestamp.

In the next lesson we will do some "twicking" to the code.