LESSON SC 12 – ICR Factory

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 create the SC used by the IIoTC.

As we mentioned before, an entity of our project is the **Innovative IoT Company** (*IIoTC*) that provides the installation of the microcontrollers to the cars.

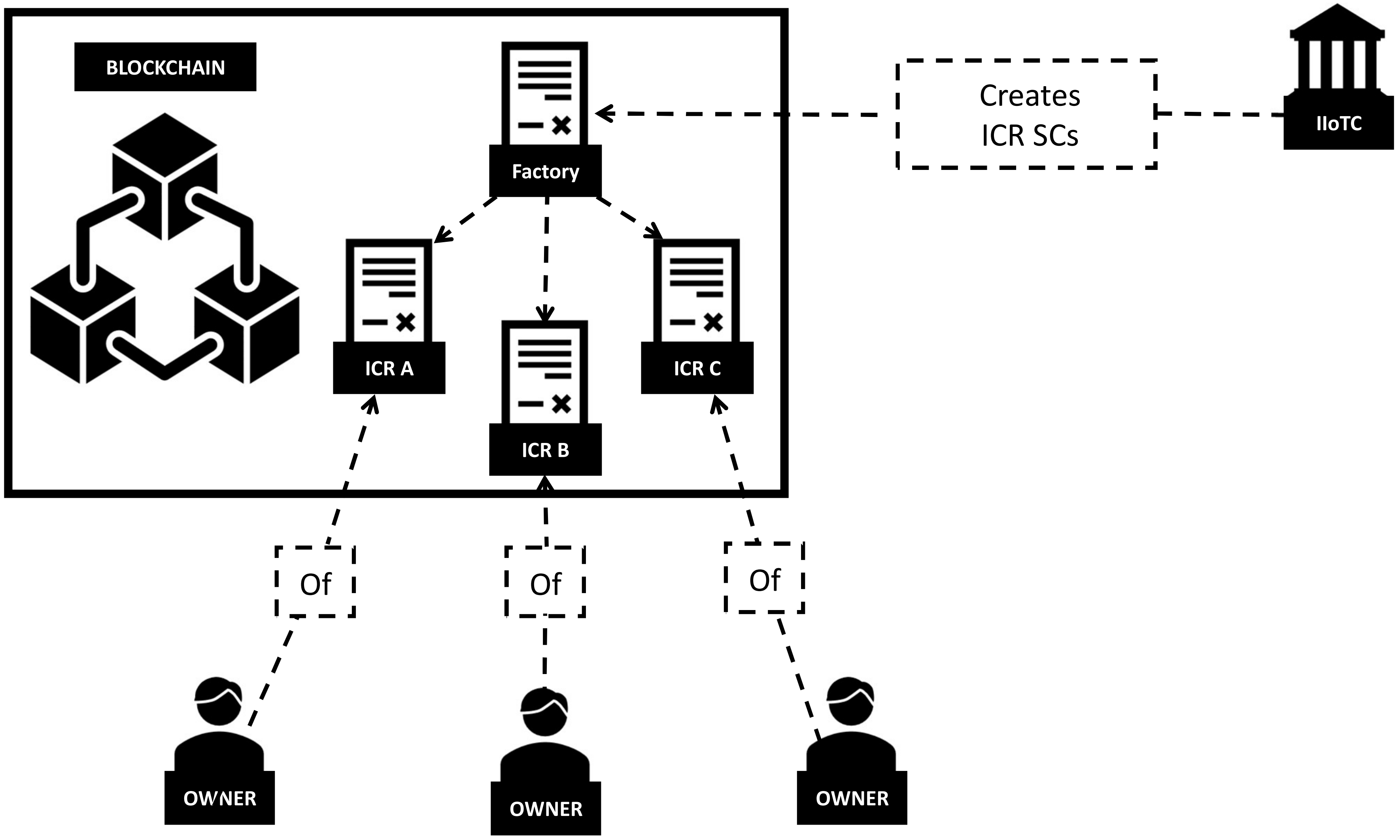
We also mentioned that car fleet owners will be able to list their cars for rent.

But, the `ICR` SC only has one `owner`, so we need a SC for IIOTC that handles the creation (*deployment*) of multiple `ICR` SCs (one `ICR` SC per `owner`).

So, let's discuss what this SC needs to do:

We will call this SC `ICRFactory.`

The company will use `ICRFactory` to deploy `ICR` SCs for car owners.


BLOCKCHAIN
Factory
ICR A
ICR B
ICR C
Creates ICR SCs
IIoTC
Of
Of
Of
OWNER
OWNER
OWNER

ICR and ICR Factory

Create a new file called ICR.sol and import ICRManager. Then, using the "is" keyword, inherit ICRManager to the ICR.

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

pragma solidity 0.8.26;

import {ICRManager} from "./ICRManager.sol";

contract ICR is ICRManager {}
```

Now let's create the ICRFactory.

Create a new file called ICRFactory.sol and import the ICR SC.

ICRFactory will **NOT** inherit from the ICR SC, it will just use it as a template to deploy multiple copies of it.

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

pragma solidity 0.8.26;

import {ICR} from "./ICR.sol";

contract ICRFactory {}
```

ICR Factory

Let's create a mapping that stores the addresses of all ICRs that are deployed from ICRFactory and a counter that keeps track of the number ICRs created.

```
    mapping(uint256 _icrId => address _icrAddress) private s_icrs;
    uint256 private s_nextIcrId;
```

Each ICR has an owner and each owner can only own one ICR SC so let's also keep track of each owner that owns an ICR SC.

```
    mapping(address _user => bool _hasSc) private s_userHasSc;
```

Before we start creating the main functions, let's create a custom error for a user that is already owner of an ICR SC.

We should also ensure owners cannot be "0x0" (address(0)) as this will break our functionality since "0x0" is not an actual account address. (*We should also do this in the registerCar function for the _mc addresses*)

Let's also create an event that will be emitted after a new ICR is deployed.

```
    error ICRFactory__AlreadyHasSmartContract(address _user);
    error ICRFactory__UserCannotBeAddressZero();

    event IcrDeployed(uint256 indexed _icrId, address indexed _icrAddress, address indexed  _icrOwner);
```

Here you can also create the getter functions to retrieve the state variables and mappings since they are private.

Deploy SC from SC

To deploy a SC from another SC we need to use the "new" keyword and define the SC. We also need the SC code that we want to deploy which in our case is the ICR we imported.

```
        new ICR();
```

Let's create a function that deploys ICR SCs. We will make it external for now and later we will add some access control (only company should use it). It will take as input the address of the account that will be the owner of the ICR SC to be deployed.

```
    function deployICR(address _user) external {
        if(s_userHasSc[_user]) {
            revert ICRFactory__AlreadyHasSmartContract(_user);
        }
        if(_user == address(0)) {
            revert ICRFactory__UserCannotBeAddressZero();
        }
        s_userHasSc[_user] = true;
        uint256 newIcrId = s_nextIcrId;
        ICR icr = new ICR();
        s_icrs[newIcrId] = address(icr);
        s_nextIcrId++;
        emit IcrDeployed(newIcrId, address(icr), icr.getOwner());
    }
```

We first check if _user already has an ICR and if they have, revert with our custom error, else we proceed to the functionality.

Next slide

deployICR Explanation

After the checks we update the state of `s_userHasSC` because `_user` will now be the owner of a new ICR.

```
        s_userHasSc[_user] = true;
```

Next, we define an ID for our new ICR and we make it equal to the `s_nextIcrId` state variable.

```
        uint256 newIcrId = s_nextIcrId;
```

Next, *we define a new instant of an* ICR *SC and we name it* `icr`. We deploy the new ICR SC and we make it equal to the `icr` instant for the duration of the function.

```
        ICR icr = new ICR();
```

We add the address of the `icr` to the `mapping` that keeps track of the ICR SCs deployed using typecast `address(icr)`.

```
      s_icrs[newIcrId] = address(icr);
```

We update the `s_nextIcrId` state variable.

```
        s_nextIcrId++;
```

And lastly, we `emit the event` we created with the ID of the new ICR, the `address` of the new ICR and the `address` of the `owner`. To get the `owner` of the new ICR SC we can just use the function `getOwner()` we created in the `ICRRegistry`. This is called `external function call from a SC`, and we will explain it more later.

```
        emit IcrDeployed(newIcrId, address(icr), icr.getOwner());
```

Deploy an `ICR` with `ICRFactory`

Compile and deploy the `ICRFactory` in Remix IDE.

Next choose an account from the account list (*remember the `address`*) and use it as a parameter for the `deployICR` function.

In the console, look at the `deployICR` transaction we just made and find the `logs`.

Remember that at the `logs`, we can see the `events` that were `emitted` during the transaction.

Find the event `IcrDeployed event` and look at the `arguments(args)`.

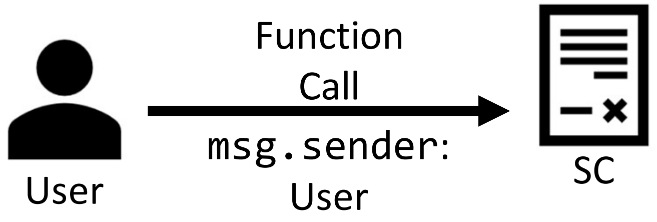
You will notice that we have three arguments:

1. ID of the new SC which is `0` because it is the first `ICR` we deployed
2. `Address` of the new `ICR` SC
3. `Address` of the `owner` **BUT,** you will notice that the `owner` is the `address` of the `ICRFactory`. This is because *when we are doing external function calls from another SC, the `msg.sender` is the SC that calls the external function* and in our `ICRRegistry's constructor`, we declare the `msg.sender` as the `owner`.

`msg.sender` on Function Calls

To understand this concept better let's see two examples:

Example 1: User calls a function on a SC. The `msg.sender` of the call is the user.

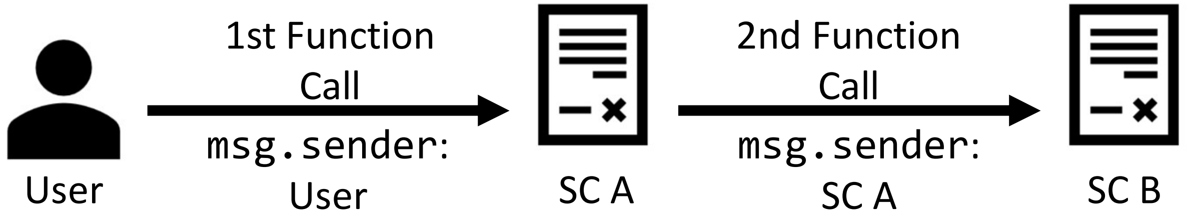


Example 2: User calls a function on SC A that calls a function on SC B.

In this case two function calls are made in total:

1st: User to SC A (`msg.sender` is the user)

2nd: SC A to AC B (`msg.sender` is SC A)



Mitigation

To fix this, we first need to make some adjustments to the `ICR` SC.

In the `constructor` of the `ICRRegistry` instead of declaring the `msg.sender` as the `owner`, we can add the `owner` as a parameter. We should also add a check if the `address` is not `address(0)` since this is an empty `address` and would break our SC's functionality. You can do that with a `require` statement or a `custom error`.

```
contract ICRRegistry {
    // ...
    constructor(address _owner) {
        require(_owner != address(0), "address 0 cannot be the owner");
        i_owner = _owner;
    }
    // ...
}
```

Since we added an argument in the `constructor` of `ICRRegistry` and this SC inherits its properties to `ICRManager` and then `ICR`, we need to update both of their `constructors` so that they can also accept this argument.

```
contract ICRManager is ICRRegistry {
    // ...
constructor(address _owner) ICRRegistry(_owner) {}
    // ...
}
```

Next slide

Mitigation (2)

```
contract ICR is ICRManager {
    // ...
    constructor(address _owner) ICRManager(_owner) {}
    // ...
}
```

In the example above and in the previous slide, you can see that *in the `constructor` of a SC, we can specify the `constructors` from other SCs that are inherited and pass the arguments they need.*

Now, let's update the `ICRFactory` to pass `_user` as parameter when the `ICR` is deployed.

```
    function deployICR(address _user) external {
        // … checks
        s_userHasSc[_user] = true;
        uint256 newIcrId = s_nextIcrId;
        ICR icr = new ICR(_user); // Pass _user as parameter here
        s_icrs[newIcrId] = address(icr);
        s_nextIcrId++;
        emit IcrDeployed(newIcrId, address(icr), icr.getOwner());
    }
```

Now, let's try to compile, deploy and use `deployICR` again to see that the `owner` is correctly defined in the `event IcrDeployed`.

Application Binary Interface

You will notice that the `ICR` SC deployed from `ICRFactory` does not appear in the sidebar under deployed SCs. However, we can add it manually. *In blockchain we can interact with **ANY** SC that is already deployed if we just have two things:*

1) *The address of the SC*
2) *The ABI of the SC (Application Binary Interface)*

What is the ABI of a SC:

The ABI (Application Binary Interface) of a SC is a JSON specification that defines how to interact with the SC on the blockchain. It acts as a bridge between the SC bytecode (on-chain) and the external applications or users that interact with it (off-chain).

We will use the ABI of our SC later to call its functions from the microcontroller.

The *ABI is generated when the SC is compiled*, and we can get it from Remix IDE. We will use the ABI of the `ICR` SC to interact with it from Remix IDE.

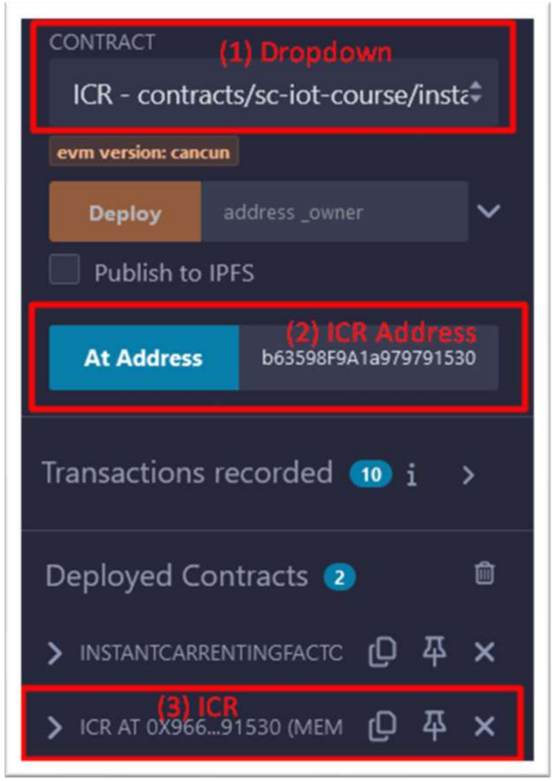
Interact with ICR Deployed from ICR Factory

In the CONTRACT field in DEPLOY AND RUN TRANSACTIONS you can see a dropdown of all the SCs we compiled.

You can also see the `ICR` SC. If you cannot see it the go to the `ICR.sol`, compile it and check again (do not deploy it).

1. Choose the `ICR` SC from the dropdown. This will fetch the ABI of `ICR` SC.
2. Under DEPLOY button there is an AT ADDRESS button and next to it there is an input field. Add the address of the `ICR` SC that was deployed from `ICRFactory` (find it in the `logs` of the transaction).
3. Press the AT ADDRESS button and the SC will appear under the Deployed SCs section.

Now you can try to interact with it. For example, check who the `owner` is.



Outro

Nice, we created a SC that creates other SCs.

We also learned a few things about external function calls from other SCs and how to interact with them from anywhere.

In the next lesson we are going to learn about ready to use SCs.