

# LESSON SC 9 – Inheritance

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 the concept of inheritance! We will also look on visibility modifiers and immutability.



## Inheritance

Inheritance in Solidity allows one SC to derive properties and functions from another SC.

We can use that to split our SC's functionality to "Registry" that handles the registration of new cars and "Manager" that handles the status of the cars.

Create two new files called ICRRegistry.sol and ICRManager.sol on the same folder.

We will make the ICRRegistry(parent) to inherit its properties to ICRManager (child) with the "is" keyword.

<pre>// SPDX-License-Identifier: MIT pragma solidity 0.8.26; contract ICRRegistry {} To import the ICRRegistry to ICRManager do the following:</pre>				
<pre>// SPDX-License-Identifier: MIT pragma solidity 0.8.26;</pre>				
<pre>import {ICRRegistry} from "./ICRRegistry.sol"; // import Contract from File</pre>				
<pre>contract ICRManager is ICRRegistry {}</pre>				



# Visibility Modifiers(1)

In *LESSON INTRO 4* we talked a bit about visibility modifiers but since we are learning about inheritance we should discuss them more here.

Visibility modifiers in Solidity define the scope and accessibility of functions and state variables within and outside of SCs. They determine who can call a function or access a state variable and from where.

There are four ways to access a variable or a function:

- From within the SC
- From a derived (child) SC
- <u>From an external SC (a SC that is deployed in the same blockchain but does not inherit from our SC and YES this is possible, and we will learn it later)</u>
- From an external account that is not a SC (a user or entity outside of the blockchain)



# Visibility Modifiers(2)

There are four visibility modifier in solidity:

- Public variable/function: Accessible by anyone
- Private <u>variable/function</u>: Accessible only by the SC that is initialized
- Internal <u>variable/function</u>: Accessible only by the SC that is initialized and SCs that inherit from it (children)
- External <u>function</u>: Accessible only from outside the SC (cannot be called from functions inside the SC or its children)

Check the table below to see the comparisons:

Modifier	This SC	Children SC	External SC	External Account
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
internal	Yes	Yes	No	No
external	No	No	Yes	Yes



## Visibility Modifiers(3)

It is generally a best practice to <u>use the least permissive visibility possible</u>. We cannot always do that especially since we want to have ways for our entities to interact with our SC, but we can always minimize the risks if we are less permissive.

For example, we can make our state variable private or internal and add functions that allow their retrieval instead of making them public.

```
uint256 private nextCarId;
function getNextCarId() external view returns (uint256) {
    return nextCarId;
}
```

Since we do not want the registerCar function to be used inside the SC, we can make it external.

function registerCar(address \_mc, uint256 \_price) external {}

We generally want to think. who should have access to the state variable or function and use the appropriate visibility modifier.



## Pure functions

If we <u>make our constant variables private and create getter functions to retrieve them we</u> <u>have to make the functions pure</u>.

<u>Pure functions</u> <u>do not read or modify any state variables of the SC.</u> They rely only on their input parameters and perform computations without interacting with the blockchain's state.

```
function add(uint256 a, uint256 b) public pure returns (uint256) {
   return a + b;
```

You can also <u>retrieve constant variables with pure functions</u> because these variables do not rely on the SC's mutable state and are treated as compile-time constants.

```
uint256 private constant LOWER_PRICE_LIMIT = 0.1 ether;
function getLowerPriceLimit() public pure returns (uint256) {
    return LOWER_PRICE_LIMIT;
}
```





In the table below, you can see the comparison between pure, view and other functions. Other functions are functions that modify the state and payable functions.

Feature	Pure	View	Other
Read State	No	Yes	Yes
Modifies State	No	No	Yes
Gas Cost	No (when called externally)	No (when called externally)	Yes



## Immutability

Similar to constant variables, if you have a state variable that is set only once and should not change ever again (like owner in our case) you can make it immutable.

<u>Immutable variables are a type of state variable in Solidity whose value is set only once, during</u> <u>the SC's deployment via the constructor, and cannot be modified thereafter.</u> They offer an alternative to constant variables when the value is not known at compile time but is fixed after deployment.

<u>Immutable variables are stored in the bytecode of the SC instead of the storage</u>, making them cheaper to access compared to regular state variables.

Let's make the owner immutable since we do not want them to change after deployment.



## Restructuring: ICRRegistry (1)



Let's restructure our code. We will start with ICRRegistry and first let's add the errors.

We will add all the errors in ICRRegistry SC but if you like you can split them. You can also change the name of the errors to state the exact name of the SC (instead of ICR\_\_ErrorName you can use ICRRegistry\_\_ErrorName or ICRManager\_\_ErrorName).

```
error ICR__CarIsNotInTheCorrectStatus(uint256 _carId, Status _status);
error ICR__IsNotOwner(address _notOwner);
error ICR__IsNotMc(uint256 _carId, address _notMc);
error ICR__InvalidCar(uint256 _carId);
error ICR__NotEnoughEther(uint256 _price);
error ICR__PriceIsTooSmall(uint256 _price);
error ICR__PriceIsTooBig(uint256 _price);
```

We will keep the event CarRegistered since we will have our registerCar function here.

We will also keep our state variables (with modified visibility), Car struct and modifier.

Lastly, we will create functions to retrieve the state variables since they are not public anymore.

## Restructuring: ICRRegistry (2)

```
enum Status {
    UNAVAILABLE, AVAILABLE, OCCUPIED
}
event CarRegistered(uint256 indexed _carId, address indexed _mc);
struct Car {
    address mc;
   uint256 price;
    Status status;
}
address internal immutable owner;
uint256 private constant LOWER_PRICE_LIMIT = 0.1 ether;
uint256 private constant UPPER_PRICE_LIMIT = 10 ether;
mapping (uint256 _carId => Car _car) internal cars;
uint256 internal nextCarId;
constructor() {
    owner = msg.sender;
}
```

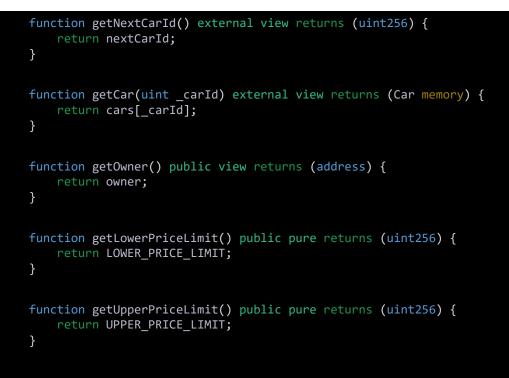




## Restructuring: ICRRegistry (3)

```
function registerCar(address _mc, uint256 _price) external {
    if (msg.sender != owner) {
        revert ICR__IsNotOwner(msg.sender);
   if (_price < LOWER_PRICE_LIMIT) {</pre>
        revert ICRRegistry_PriceIsTooSmall(_price);
    if (_price > UPPER_PRICE_LIMIT) {
        revert ICRRegistry__PriceIsTooBig(_price);
   Car memory car = Car(_mc, _price, Status.UNAVAILABLE);
   uint256 currentCarId = nextCarId;
   cars[currentCarId] = car;
    nextCarId++;
    emit CarRegistered(currentCarId, _mc);
}
modifier carIsValid (uint256 _carId) {
    if (_carId >= nextCarId) {
        revert ICR InvalidCar( carId);
    }
    _;
```

#### Restructuring: ICRRegistry (4)

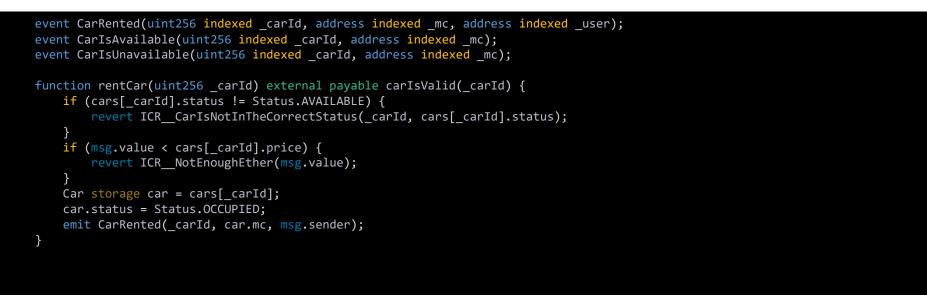






## Restructuring: ICRManager (1)

In the ICRManager we will add the rest of the events as well as the functions that change the status of a car.





## Restructuring: ICRManager (2)

```
function changeCarStatusMc(uint256 carId) external 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;
   emit CarIsAvailable( carId, car.mc);
}
function changeCarStatusOwner(uint256 carId) external 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;
       emit CarIsUnavailable(_carId, car.mc);
    } else {
       car.status = Status.AVAILABLE;
       emit CarIsAvailable( carId, car.mc);
```



#### Outro

This was a lot of information!

We learned about: Inheritance Visibility Immutability Pure functions And we restructured our SC!

In the next lesson we will learn how to retrieve ETH from the SC and how time works on the blockchain.