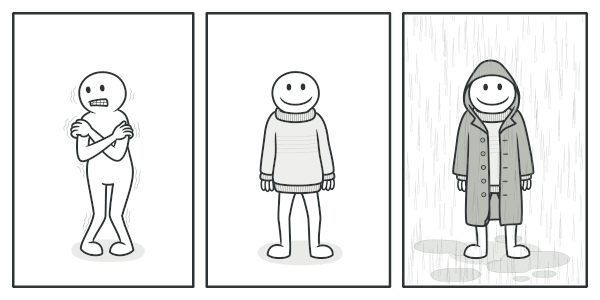
**Solidity Tutorial: all about Modifiers**



Modifiers are analog to the decorator pattern (source: <https://refactoring.guru/design-patterns/decorator>)

In Solidity, **Modifiers**express what actions are occurring in a declarative and readable manner. They are similar to the [**decorator pattern**](https://refactoring.guru/design-patterns/decorator)used in **Object Oriented Programming.**

**What is exactly a Modifier in Solidity ?**

The Solidity documentation define a modifier as follow:

A function modifier is a compile-time source code roll-up.

It can be used to amend the semantics of functions in a declarative way.

From this definition, we can understand that a modifier aims to change the behaviour of the function to which it is attached.

For instance, automatically checking a condition prior to executing the function (this is mainly what they are used for).

Modifiers are useful because they **reduce code redundancy**. You can re-use the same modifier in multiple functions if you are checking for the same condition over your smart contract.

**When to use a modifier in Solidity?**

The main use case of modifiers is for **automatically checking a condition, prior to executing a function**. If the function does not meet the modifier requirement, an exception is thrown, and the function execution stops.

**How to create and use Modifiers ?**

**Modifiers**can be created (declared) as follow:

modifier MyModifier {  
 // modifier code goes here...  
}

You can write a modifier with or without arguments. If the modifier does not have argument, you can omit the parentheses () .

modifier ModifierWithArguments(uint a) {  
 // ...  
}modifier ModifierWithoutArguments() {  
 // ...  
}modifier ModifierWithoutArguments {  
 // ...  
}

**How do modifiers work?**

Let’s start with some basic examples below.

modifier onlyOwner {  
 require(msg.sender == owner);  
 \_;  
}

**The \_; symbol**

The symbol \_; is called a merge wildcard. It merges the function code with the modifier code where the \_; is placed. *(give example in code).*

In other terms, the body of the function (to which the modifier is attached to) will be inserted where the special symbol \_; appears in the modifier’s definition.

Using the terms of the Solidity docs, it *“returns the flow of execution to the original function code”.*

A modifier must have the symbol \_; within its body in order to execute. It is mandatory *(does Solidity throw an error if it is not the case?).*

**Where to place the \_; ?**

The place where you write the \_; symbol will decide if the function has to be executed before, in between or after the modifier code.

modifier SomethingBefore {  
 require(/\* check something first \*/);  
 \_; // resume with function execution  
}// Do one where modifier is placed in the middlemodifier SomethingAfter {  
 \_; // run function first  
 require(/\* then check something \*/)  
}

As shown in the example above, you can place the \_; at the beginning, middle or the end of your modifier body.

In practice, (especially until you understand how modifiers work really well), the safest usage pattern is to place the \_; at the end. In this scenario, the **modifier** serves for consistent validation check, so to check a condition upfront and then carry on. The code snippet below show show this as example:

function isOkay() public view returns(bool) {  
 // do some validation checking  
 return true;  
}function isAuthorised(address \_user) public view returns(bool) {  
 // logic that checks that \_user is authorised  
 return true;   
}modifier OnlyIfOkAndAuthorised {  
 require(isOkay());  
 require(isAuthorised(msg.sender));  
 \_;  
}

*NB: return true are just placeholder for more meaningful results.*

**Passing arguments to Modifiers.**

Modifiers can also accept arguments. Like a function, you just have to pass the variable type + name between parentheses in front of the modifier name.

modifier Fee (uint \_fee) {  
 if (msg.value >= \_fee) {  
 \_;  
 }  
}

Using the example above, you can ensure that a user (or contract) that call one of your contract function has sent some ethers to pay a pre-required fee.

**Let’s illustrate with a simple example of a contract that acts like a vault.**

You want to ensure that every user that want to take out money stored in the contract vault *pays a minimum fee of 2.5 % of an ether to the contract*.

A modifier with arguments can simulate this behaviour. See the code below.

pragma solidity ^0.8.0;contract Vault {  
   
 modifier fee(uint \_fee) {  
 if (msg.value != \_fee) {  
 revert("You must pay a fee to withdraw your ethers");  
 } else {  
 \_;  
 }  
 }  
   
 function deposit(address \_user, uint \_amount) external {  
 // ...  
 }  
   
 function withdraw(uint \_amount) external payable fee(0.025 ether) {  
 // ...  
 }  
}

An other example is to check for instance that it is not a specific address that is calling the function.

modifier notSpecificAddress (address \_user) {  
 if (\_user === msg.sender) throw;  
 \_;  
}function someFunction(address \_user) notSpecificAddress("0x......") {  
 // ...  
}

* See other examples on CryptoZombie, chapter 8 and 9.
* Look at the modifier nonReentrant from Open Zeppelin.

Can find more practical use cases below on how to use arguments in modifiers.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visibles from the function are visible in the modifier.

**Applying multiple Modifiers to a function.**

Multiple modifiers can be applied to a function. You can do this as follow:

contract OwnerContract { address public owner = msg.sender;  
 uint public creationTime = now; modifier onlyBy(address \_account) {  
 require(  
 msg.sender == \_account,  
 "Sender not authorized.  
 );  
 \_;  
 } modifier onlyAfter(uint \_time) {  
 require(  
 now >= \_time,  
 "Function called too early."  
 );  
 \_;  
 } function disown() public onlyBy(owner) onlyAfter(creationTime + 6 weeks) {  
 delete owner;  
 }}

The modifiers will be executed in the order they are defined, so *from left to right.*So in the example above, the function will check the following conditions prior to run:

1. **onlyBy(...)** : is the address calling the contract the owner?
2. **onlyAfter(...)** : Is there more than 6 weeks that the person has been the owner?

**Modifier Overriding**

Like functions, modifiers defined in one contract can be overriden by other contracts that derive from it. As the Solidity documentation suggest:

Modifiers are inheritable properties of contracts and may be overridden by derived contracts.

Therefore, **symbols introduced in a modifier are not visible in the function**(*because the modifier might change if it is overridden in the derived contract)*.

**Modifiers with Enums**

If you contract holds a global variable of type enum, you can check the value it holds by passing one of the available option as an argument to the modifier.

enum State { Created, Locked, Inactive }  
State state;modifier isState(State \_expectedState) {  
 require(state == \_expectedState);  
 \_;  
}

**Modifiers in Practice**

The following section describes some practical use cases for using modifiers.

**Only owner, or specific users**

[A good example is given on the Colony blog](https://blog.colony.io/writing-more-robust-smart-contracts-99ad0a11e948/). It demonstrates how to use a modifier to check that a function can be called and executed only by a specific address.

modifier OnlyBy(address \_account) {  
 require(msg.sender == \_account, “sender not authorised”);  
 \_;  
}

**Support Multiple Admins**

The article [*“Writing robust smart contracts in Solidity”*](https://blog.colony.io/writing-more-robust-smart-contracts-99ad0a11e948/#support-for-multiple-admin-accounts)gives an other example that extends from the previous one. Instead of allowing one user, we can support multiple specific users to run a function.

**Make the contract state discoverable to specific conditions**

You can use modifiers in conjunction with **getter functions**.

For instance, view functions that should be guarded by a certain combination of concerns can use the modifier

The idea is that clients, other contracts and other internal functions can use the getter function to fetch / check the individual concerns (if they want to).

As a result, the logic of it all (and your intent) remains very clear to reviewers.

The example above enables to keep the code organised while making the state discoverable to clients and ensure potentially that a complex logic is consistently applied.

You can then have several functions like above which have some tricky steps. Hopefully, they can be understood as individual concerns. Then when used in combination, you can reduce repetition and improve readability with a modifier.

**Data Validation**

An other excellent use case for modifiers is to verify data inputs.

Below are some examples, based on different data types.

modifier throwIfAddressIsInvalid(address \_target) {  
 if (\_target == 0x0) throw;  
 \_;  
}modifier throwIfIsEmptyString(string \_id) {  
 if (bytes(\_id).length == 0) throw;  
 \_;  
}modifier throwIfEqualToZero(uint \_id) {  
 if (\_id == 0) throw;  
 \_:  
}modifier throwIfIsEmptyBytes32(bytes32 \_id) {  
 if (\_id == "") throw;  
 \_:  
}

**Check that some time (=blocks) has passed**

modifier OnlyAfter(uint \_time) {  
 require(now >= \_time, "function called too early!");  
 \_;  
}

**Refund Ether sent by Accident**

The blockchain world does not allow for mistakes. For any ethers or other form of value sent by accident, there is no clerk to turn to for a claim, as there is no bank or central authority that monitor transactions.

However, you can make your smart contract to act like a good cashier, by noticing him what to do when accidents happen on the end-user.

modifier refundEtherSentByAccident() {  
 if(msg.value > 0) throw;  
 \_;  
}

**Charge a fee**

On the other end, your smart contract can reinforce payments, by ensuring for instance that a specific fee has been paid in order to perform an operation.

Modifier Fee (uint \_fee) {  
 if (msg.value >= \_fee) {  
 \_;  
 }  
}

**Send Change Back**

We saw before how a modifier could be used to refund ethers sent by accident. But how about the case where you send more than you should?

[Some code snippet from the Solidity documentation](https://docs.soliditylang.org/en/v0.5.12/common-patterns.html#restricting-access) provides good examples on how to do this.

An example of the implementation would look like this.

modifier GiveChangeBack(uint \_amount) {  
 \_;  
 **if** (msg.value > \_amount) {  
 msg.sender.transfer(msg.value - \_amount);  
 }  
}

**Switch between states**

Modifier can be used for complex design patterns. This include the [**state machine pattern.**](https://github.com/fravoll/solidity-patterns/blob/master/docs/state_machine.md)

**Prevent Re-Entrancy**

Look at the modifier *nonReentrant*from OpenZeppelin  
<https://docs.soliditylang.org/en/v0.5.3/contracts.html#function-modifiers>

**Restrict to Users / Disallow contracts to interact**

SUSHISWAP gives an excellent example of a modifier that [enables only **externally owned accounts**(= users) to claim their rewards on their contract](https://github.com/sushiswap/sushiswap/blob/f01b52e22cfc193c4653aee769606dd50fc07fc2/contracts/SushiMakerKashi.sol#L86-L90).

As a result, this prevent other smart contracts to call the specific function the modifier is attached to.

modifier onlyEOA() {  
 require(msg.sender == tx.origin, "Must use EOA");  
 \_;   
}

**only Humans**

This modifier complements the previous one, which checked if it was an Externally Owned Account (EOA). It checks at the caller address that no code is present, using the opcode *extcodesize(…)*opcodevia assembly

[This code was taken from the Youtube video describing the Acoraida Monica contract challenge.](https://www.youtube.com/watch?v=ozqOlUVKL1s)

modifier onlyHuman {  
 uint size;  
 address addr = msg.sender;  
 assembly { size := extcodesize(addr) }  
 require(size == 0, “only humans allowed! (code present at caller address)”);  
 \_;  
}

**Security Consideration about modifiers**

Modifiers could have dangerous in the previous versions of Solidity.

Before Solidity version 0.4.0, it was possible to skip the part after \_;