Advanced Functions

Function Modifiers

Copy page

Build custom function modifiers to efficiently modify functionality.

Function modifiers allow you to efficiently change the behavior of functions. In some ways, it’s similar to inheritance, but there are restrictions, particularly in variable scope.

Objectives

By the end of this lesson you should be able to:

- Use modifiers to efficiently add functionality to multiple functions

Adding a Simple OnlyOwner Modifier

By default, `public` functions can be called by **anyone**, without restriction. Often this is desirable. You want any user to be able to see what NFTs are for sale on your platform, sign up for a service, or read various items stored in state.

However, there will be many functions you **don't** want any user to be able to do, such as setting the fee for using the app, or withdrawing all funds in the contract! A common pattern to protect these functions is to use `modifier` s to make sure that only the owner can call these functions.

For a production app, you'll want to use a more robust implementation of `onlyOwner` , such as the one provided by OpenZeppelin.

Adding an Owner

The address of the deployer of a contract is **not** included as an accessible property. To make it available, add it as a state variable and assign `msg.sender` in the `constructor` .

Reveal code

Creating an `onlyOwner` Modifier

to modify state. Modifiers must have a special `_` character, which serves as a placeholder for where the code contained within the modified function will run.

Create a simple `onlyOwner` modifier, which returns an `error` of `NotOwner` with the sending address if the sender is not the owner.

Reveal code

Test your `modifier` by adding a function that uses it:

Reveal code

To test, deploy your contract and call the `iOwnThis` function. You should see the message “You own this!”.

Next, switch the *Account*, and try the function again. You should see an error in the console:

Ask a question... Ctrl+I

```
call to Modifiers.iOwnThis errored: VM error: revert.

revert
    The transaction has been reverted to the initial state.
Error provided by the contract:
NotOwner
Parameters:
{
 "_msgSender": {
  "value": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"
 }
}
Debug the transaction to get more information.
```

Always verify the output of a function call in the console. The result that appears under the button for the function is convenient, but it does **not** clear or change if a subsequent call reverts.

Modifiers and Variables

Modifiers can have parameters, which essentially work the same as in functions. These parameters can be independent values, or they can overlap with the arguments provided to a function call.

Modifiers with Parameters

Modifier parameters can be the arguments provided to the functions they modify. You can perform calculations and trigger errors based on these values.

```
error NotEven(uint number);

modifier onlyEven(uint _number) {
    if(_number % 2 != 0) {
        revert NotEven(_number);
    }
    _;
}

function halver(uint _number) public pure onlyEven(_number) returns (uint)
    return _number / 2;
}
```

Independent Scope

While `modifiers` are used to modify functions and can share inputs, they have separate scopes. The following example will **not** work:

```
// Bad code example, does not work
modifier doubler(uint _number) {
    _number *= 2;
    _;
}

function modifierDoubler(uint _number) public pure doubler(_number) returns
    return _number; // Returns the original number, NOT number * 2
}
```

Conclusion

Function `modifier` s are an efficient and reusable way to add checks, trigger errors, and control function execution. In this lesson, you've seen examples of how they can be used to abort execution under certain conditions. You've also learned that they have separate scopes and cannot be used to modify variables within the function they modify.

‹ Function Modifiers Structs ›

base docs

B e.org Blog Privacy Policy Terms of Service Cookie Policy