Executing Code from Another Class

In previous chapters, we explored how to call external _contracts_ to execute their logic and update their state. But what if we want to execute code from another class without updating the state of another contract? Starknet makes this possible with _library calls_, which allow a contract to execute the logic of another class in its own context, updating its own state.

Library calls

The key differences between _contract calls_ and _library calls_ lie in the execution context of the logic defined in the class. While contract calls are used to call functions from deployed **contracts**, library calls are used to call stateless **classes** in the context of the caller.

To illustrate this, let's consider two contracts _A_ and _B_.

When A performs a _contract call_ to the **contract** B, the execution context of the logic defined in B is that of B. As such, the value returned by `get_caller_address()` in B will return the address of A, `get_contract_address()` in B will return the address of B, and any storage updates in B will update the storage of B.

However, when A uses a _library call_ to call the **class** of B, the execution context of the logic defined in B is that of A. This means that the value returned by `get_caller_address()` in B will be the address of the caller of A, `get_contract_address()` in B's class will return the address of A, and updating a storage variable in B's class will update the storage of A.

Library calls can be performed using the dispatcher pattern presented in the previous chapter, only with a class hash instead of a contract address.

Listing {{#ref expanded-ierc20-library}} describes the library dispatcher and its associated `IERC20DispatcherTrait` trait and impl using the same `IERC20` example:

```cairo,noplayground
{{#include ../listings/ch15-starknet-cross-contract-interactions/
listing_04_expanded_ierc20_library/src/lib.cairo}}
```

{{#label expanded-ierc20-library}}
Listing {{#ref expanded-ierc20-library}}: A simplified example of the `IERC20DLibraryDispatcher` and its associated trait and impl

One notable difference with the contract dispatcher is that the library dispatcher uses `library_call_syscall` instead of `call_contract_syscall`. Otherwise, the process is similar.

Let's see how to use library calls to execute the logic of another class in the context of the current contract.

Using the Library Dispatcher

Listing {{#ref library-dispatcher}} defines two contracts: `ValueStoreLogic`, which defines the logic of our example, and `ValueStoreExecutor`, which simply executes the logic of `ValueStoreLogic`'s class.

We first need to import the `IValueStoreDispatcherTrait` and `IValueStoreLibraryDispatcher` which were generated from our interface by the compiler. Then, we can create an instance of `IValueStoreLibraryDispatcher`, passing in the `class_hash` of the class we want to make library calls to. From there, we can call the functions defined in that class, executing its logic in the context of our contract.

```cairo,noplayground
```

{{#include ../listings/ch15-starknet-cross-contract-interactions/
listing_05_library_dispatcher/src/lib.cairo}}
```

{{#label library-dispatcher}}
<span class="caption">Listing {{#ref library-dispatcher}}: An example contract using a Library Dispatcher</span>
When we call the `set_value` function on `ValueStoreExecutor`, it will make a library call to the `set_value` function defined in `ValueStoreLogic`. Because we are using a library call, `ValueStoreExecutor`'s storage variable `value` will be updated. Similarly, when we call the `get_value` function, it will make a library call to the `get_value` function defined in `ValueStoreLogic`, returning the value of the storage variable `value` - still in the context of `ValueStoreExecutor`.
As such, both `get_value` and `get_value_local` return the same value, as they are reading the same storage slot.
## Calling Classes using Low-Level Calls
Another way to call classes is to directly use `library_call_syscall`. While less convenient than using the dispatcher pattern, this syscall provides more control over the serialization and deserialization process and allows for more customized error handling.
Listing {{#ref library_syscall}} shows an example demonstrating how to use a `library_call_syscall` to call the `set_value` function of `ValueStore` contract:
```cairo,noplayground
{{#include ../listings/ch15-starknet-cross-contract-interactions/listing_07_library_syscall/
src/lib.cairo}}
```

{{#label library_syscall}}
Listing {{#ref library_syscall}}: A sample contract using `library_call_syscall` system call
To use this syscall, we passed in the class hash, the selector of the function we want to call and the call arguments.
The call arguments must be provided as an array of arguments, serialized to a `Span<felt252>`. To serialize the arguments, we can simply use the `Serde` trait, provided that the types being serialized implement this trait. The call returns an array of serialized values, which we'll need to deserialize ourselves!
Summary
Congratulations for finishing this chapter! You have learned a lot of new concepts:
- How _Contracts_ differ from _Classes_ and how the ABI describes them for external sources
- How to call functions from other contracts and classes using the _Dispatcher_ pattern
- How to use _Library calls_ to execute the logic of another class in the context of the caller
- The two syscalls that Starknet provides to interact with contracts and classes
You now have all the required tools to develop complex applications with logic spread across multiple contracts and classes. In the next chapter, we will explore more advanced topics that will help you unleash the full potential of Starknet.
```