Interacting with Another Contract
In the previous section, we introduced the dispatcher pattern for contract interactions. This chapter will explore this pattern in depth and demonstrate how to use it.
The dispatcher pattern allows us to call functions on another contract by using a struct that wraps the contract address and implements the dispatcher trait generated by the compiler from the contract class ABI. This leverages Cairo's trait system to provide a clean and type-safe way to interact with other contracts.
When a [contract interface][interfaces] is defined, the compiler automatically generates and exports multiple dispatchers. For instance, for an `IERC20` interface, the compiler will generate the following dispatchers:
- _Contract Dispatchers_: `IERC20Dispatcher` and `IERC20SafeDispatcher`
- _Library Dispatchers_: `IERC20LibraryDispatcher` and `IERC20SafeLibraryDispatcher`
These dispatchers serve different purposes:
- Contract dispatchers wrap a contract address and are used to call functions on other contracts.
- Library dispatchers wrap a class hash and are used to call functions on classes. Library dispatchers will be discussed in the next chapter, ["Executing code from another class"][library dispatcher].
- _'Safe'_ dispatchers allow the caller to handle potential errors during the execution of the call.
> Note: As of Starknet 0.13.2, error handling in contract calls is not yet available. This means that if a contract call fails, the entire transaction will fail. This will change in the future, allowing safe dispatchers to be used on Starknet.
Under the hood, these dispatchers use the low-level [`contract_call_syscall`][syscalls], which allows us to call functions on other contracts by passing the contract address, the function selector, and the function arguments. The dispatcher abstracts away the complexity of this syscall, providing a clean and type-safe way to interact with other contracts.
To effectively break down the concepts involved, we will use the `ERC20` interface as an illustration.
[interfaces]: ./ch13-02-anatomy-of-a-simple-contract.md#the-interface-the-contracts-blueprint
[syscalls]: ./appendix-08-system-calls.md
[library dispatcher]: ./ch15-03-executing-code-from-another-class.md
The Dispatcher Pattern
We mentioned that the compiler would automatically generate the dispatcher struct and the dispatcher trait for a given interface. Listing {{#ref expanded-ierc20dispatcher}} shows an example of the generated items for an `IERC20` interface that exposes a `name` view function and a `transfer` external function:
```cairo,noplayground
{{#include ../listings/ch15-starknet-cross-contract-interactions/listing_02_expanded_ierc20_dispatcher/src/lib.cairo}}
```
{{#label expanded-ierc20dispatcher}}
<span class="caption">Listing {{#ref expanded-ierc20dispatcher}}: A simplified example

of the `IERC20Dispatcher` and its associated trait and impl</span>
As you can see, the contract dispatcher is a simple struct that wraps a contract address and implements the `IERC20DispatcherTrait` generated by the compiler. For each function, the implementation of the trait will contain the following elements:
- A serialization of the function arguments into a `felt252` array, `__calldata__`.
- A low-level contract call using `contract_call_syscall` with the contract address, the function selector, and the `__calldata__` array.
- A deserialization of the returned value into the expected return type.
Calling Contracts Using the Contract Dispatcher
To illustrate the use of the contract dispatcher, let's create a simple contract that interacts with an ERC20 contract. This wrapper contract will allow us to call the `name` and `transfer_from` functions on the ERC20 contract, as shown in Listing {{#ref contract-dispatcher}}:
```cairo,noplayground
{{#rustdoc_include ../listings/ch15-starknet-cross-contract-interactions/listing_03_contract_dispatcher/src/lib.cairo:here}}
```
{{#label contract-dispatcher}}
<span class="caption">Listing {{#ref contract-dispatcher}}: A sample contract which uses the dispatcher pattern to call another contract</span>
In this contract, we import the `IERC20Dispatcher` struct and the `IERC20DispatcherTrait` trait. We then wrap the address of the ERC20 contract in an instance of the `IERC20Dispatcher` struct. This allows us to call the `name` and `transfer` functions on the ERC20 contract.
Calling `transfer_token` external function will modify the state of the contract deployed at `contract_address`.
Calling Contracts using Low-Level Calls
Another way to call other contracts is to directly use the `call_contract_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 syscalls}} shows an example demonstrating how to call the `transfer_from` function of an `ERC20` contract with a low-level `call_contract_sycall` syscall:
```cairo,noplayground
{{#include ../listings/ch15-starknet-cross-contract-interactions/listing_06_syscalls/src/lib.cairo}}
```
{{#label syscalls}}
<span class="caption">Listing {{#ref syscalls}}: A sample contract using `call_contract_sycall` syscall</span>
To use this syscall, we passed in the contract address, 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!