Anatomy of a Simple Contract
This chapter will introduce you to the basics of Starknet contracts using a very simple smart contract as example. You will learn how to write a contract that allows anyone to store a single number on the Starknet blockchain.
Let's consider the following contract for the whole chapter. It might not be easy to understand it all at once, but we will go through it step by step:
```cairo,noplayground
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_01_simple_contract/src/lib.cairo:all}}
```
{{#label simple-contract}}
<span class="caption">Listing {{#ref simple-contract}}: A simple storage contract</
span>
What Is this Contract?
Contracts are defined by encapsulating state and logic within a module annotated with the `#[starknet::contract]` attribute.
The state is defined within the `Storage` struct, and is always initialized empty. Here, our struct contains a single field called `stored_data` of type `u128` (unsigned integer of 128 bits), indicating that our contract can store any number between 0 and \\({2^{128}} - 1 \\).
The logic is defined by functions that interact with the state. Here, our contract defines and publicly exposes the functions `set` and `get` that can be used to modify or retrieve the value of the stored variable.
You can think of it as a single slot in a database that you can query and modify by calling functions of the code that manages the database.
The Interface: the Contract's Blueprint
```cairo,noplayground
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_01_simple_contract/src/lib.cairo:interface}}
```
{{#label interface}}
<span class="caption">Listing {{#ref interface}}: A basic contract interface</span>
Interfaces represent the blueprint of the contract. They define the functions that the contract exposes to the outside world, without including the function body. In Cairo, they're defined by annotating a trait with the `#[starknet::interface]` attribute. All functions of the trait are considered public functions of any contract that implements this trait, and are callable from the outside world.
> The contract constructor is not part of the interface. Nor are internal functions.

All contract interfaces use a generic type for the `self` parameter, representing the contract state. We chose to name this generic parameter `TContractState` in our interface, but this is not enforced and any name can be chosen.
In our interface, note the generic type `TContractState` of the `self` argument which is passed by reference to the `set` function. Seeing the `self` argument passed in a contract function tells us that this function can access the state of the contract. The `ref` modifier implies that `self` may be modified, meaning that the storage variables of the contract may be modified inside the `set` function.

On the other hand, the `get` function takes a snapshot of `TContractState`, which immediately tells us that it does not modify the state (and indeed, the compiler will complain if we try to modify storage inside the `get` function).
By leveraging the [traits & impls](./ch08-02-traits-in-cairo.md) mechanism from Cairo, we can make sure that the actual implementation of the contract matches its interface. In fact, you will get a compilation error if your contract doesn’t conform with the declared interface. For example, Listing {{#ref wrong-impl}} shows a wrong implementation of the `ISimpleStorage` interface, containing a slightly different `set` function that doesn't have the same signature.
```cairo,noplayground
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_02_wrong_impl/src/lib.cairo:impl}}
```
{{#label wrong-impl}}
<span class="caption">Listing {{#ref wrong-impl}}: A wrong implementation of the interface of the contract. This does not compile, as the signature of `set` doesn't match the trait's.</span>
Trying to compile a contract using this implementation will result in the following error:
```shell
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_02_wrong_impl/output.txt}}
```
Public Functions Defined in an Implementation Block
Before we explore things further down, let's define some terminology.
- In the context of Starknet, a _public function_ is a function that is exposed to the outside world. A public function can be called by anyone, either from outside the contract or from within the contract itself. In the example above, `set` and `get` are public functions.
- What we call an _external_ function is a public function that can be directly invoked through a Starknet transaction and that can mutate the state of the contract. `set` is an external function.
- A _view_ function is a public function that is typically read-only and cannot mutate the state of the contract. However, this limitation is only enforced by the compiler, and not by Starknet itself. We will discuss the implications of this in a later section. `get` is a view function.
```cairo,noplayground
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_01_simple_contract/src/lib.cairo:impl}}
```
{{#label implementation}}
<span class="caption">Listing {{#ref implementation}}: `SimpleStorage` implementation</span>
Since the contract interface is defined as the `ISimpleStorage` trait, in order to match the interface, the public functions of the contract must be defined in an implementation of this trait — which allows us to make sure that the implementation of the contract matches its interface.

However, simply defining the functions in the implementation block is not enough. The implementation block must be annotated with the `#[abi(embed_v0)]` attribute. This attribute exposes the functions defined in this implementation to the outside world — forget to add it and your functions will not be callable from the outside. All functions defined in a block marked as `#[abi(embed_v0)]` are consequently _public functions_.

Because the `SimpleStorage` contract is defined as a module, we need to access the interface defined in the parent module. We can either bring it to the current scope with the `use` keyword, or refer to it directly using `super`.

When writing the implementation of an interface, the `self` parameter in the trait methods **must** be of type `ContractState`. The `ContractState` type is generated by the compiler, and gives access to the storage variables defined in the `Storage` struct. Additionally, `ContractState` gives us the ability to emit events. The name `ContractState` is not surprising, as it’s a representation of the contract’s state, which is what we think of `self` in the contract interface trait.

When `self` is a snapshot of `ContractState`, only read access is allowed, and emitting events is not possible.

Accessing and Modifying the Contract's State

Two methods are commonly used to access or modify the state of a contract:

- `read`, which returns the value of a storage variable. This method is called on the variable itself and does not take any argument.

```cairo,noplayground
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_01_simple_contract/src/lib.cairo:read_state}}
```

- `write`, which allows to write a new value in a storage slot. This method is also called on the variable itself and takes one argument, which is the value to be written. Note that `write` may take more than one argument, depending on the type of the storage variable. For example, writing on a mapping requires 2 arguments: the key and the value to be written.

```cairo,noplayground
{{#include ../listings/ch13-introduction-to-starknet-smart-contracts/
listing_01_simple_contract/src/lib.cairo:write_state}}
```

> Reminder: if the contract state is passed as a snapshot with `@` instead of passed by reference with `ref`, attempting to modify the contract state will result in a compilation error.

This contract does not do much apart from allowing anyone to store a single number that is accessible by anyone in the world. Anyone could call `set` again with a different value and overwrite the current number. Nevertheless, each value stored in the storage of the contract will still be stored in the history of the blockchain. Later in this book, you will see how you can impose access restrictions so that only you can alter the number.