Contract Functions
In this section, we are going to be looking at the different types of functions you could encounter in Starknet smart contracts.
Functions can access the contract's state easily via `self: ContractState`, which abstracts away the complexity of underlying system calls (`storage_read_syscall` and `storage_write_syscall`). The compiler provides two modifiers: `ref` and `@` to decorate `self`, which intends to distinguish view and external functions.
1. Constructors
Constructors are a special type of function that only runs once when deploying a contract, and can be used to initialize the state of a contract.
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:constructor}}
```
Some important rules to note:
1. A contract can't have more than one constructor.
2. The constructor function must be named `constructor`, and must be annotated with the `#[constructor]` attribute.
The `constructor` function might take arguments, which are passed when deploying the contract. In our example, we pass some value corresponding to a `Person` type as argument in order to store the `owner` information (address and name) in the contract.
Note that the `constructor` function **must** take `self` as a first argument, corresponding to the state of the contract, generally passed by reference with the `ref` keyword to be able to modify the contract's state. We will explain `self` and its type shortly.
2. Public Functions
As stated previously, public functions are accessible from outside of the contract. They are usually defined inside an implementation block annotated with the `#[abi(embed_v0)]` attribute, but might also be defined independently under the `#[external(v0)]` attribute.
The `#[abi(embed_v0)]` attribute means that all functions embedded inside it are implementations of the Starknet interface of the contract, and therefore potential entry points.
Annotating an impl block with the `#[abi(embed_v0)]` attribute only affects the visibility (i.e., public vs private/internal) of the functions it contains, but it doesn't inform us on the ability of these functions to modify the state of the contract.
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:impl_public}}
```
> Similarly to the `constructor` function, all public functions, either standalone functions annotated with the `#[external(v0)]` or functions within an impl block annotated with the `#[abi(embed_v0)]` attribute, **must** take `self` as a first argument. This is not the case for private functions.
External Functions
External functions are _public_ functions where the `self: ContractState` argument is

passed by reference with the `ref` keyword, which exposes both the `read` and `write` access to storage variables. This allows modifying the state of the contract via `self` directly.
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:external}}
```
View Functions
View functions are _public_ functions where the `self: ContractState` argument is passed as snapshot, which only allows the `read` access to storage variables, and restricts writes to storage made via `self` by causing compilation errors. The compiler will mark their _state_mutability_ to `view`, preventing any state modification through `self` directly.
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:view}}
```
State Mutability of Public Functions
However, as you may have noticed, passing `self` as a snapshot only restricts the storage write access via `self` at compile time. It does not prevent state modification via direct system calls, nor calling another contract that would modify the state.
The read-only property of view functions is not enforced on Starknet, and sending a transaction targeting a view function _could_ change the state.
<!-- TODO: add an example of a view function that could modify the state using low-level syscalls -->
In conclusion, even though external and view functions are distinguished by the Cairo compiler, **all public functions** can be called through an invoke transaction and can potentially modify the Starknet state. Moreover, all public functions can be called with the `starknet_call` RPC method, which will not create a transaction and hence will not change the state.
> **Warning:** This is different from the EVM where a `staticcall` opcode is provided, which prevents storage modifications in the current context and subcontexts. Hence developers **should not** have the assumption that calling a view function on another contract cannot modify the state.
Standalone Public Functions
It is also possible to define public functions outside of an implementation of a trait, using the `#[external(v0)]` attribute. Doing this will automatically generate an entry in the contract ABI, allowing these standalone public functions to be callable by anyone from outside. These functions can also be called from within the contract just like any function in Starknet contracts. The first parameter must be `self`.
Here, we define a standalone `get_contract_name` function outside of an impl block:
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:standalone}}
```
3. Private Functions

Functions that are not defined with the `#[external(v0)]` attribute or inside a block annotated with the `#[abi(embed_v0)]` attribute are private functions (also called internal functions). They can only be called from within the contract.
They can be grouped in a dedicated impl block (e.g., in components, to easily import internal functions all at once in the embedding contracts) or just be added as free functions inside the contract module.
Note that these 2 methods are equivalent. Just choose the one that makes your code more readable and easy to use.
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:state_internal}}
```
> Wait, what is this `#[generate_trait]` attribute? Where is the trait definition for this implementation? Well, the `#[generate_trait]` attribute is a special attribute that tells the compiler to generate a trait definition for the implementation block. This allows you to get rid of the boilerplate code of defining a trait with generic parameters and implementing it for the implementation block. With this attribute, we can simply define the implementation block directly, without any generic parameter, and use `self: ContractState` in our functions.
The `#[generate_trait]` attribute is mostly used to define private impl blocks. It might also be used in addition to `#[abi(per_item)]` to define the various entrypoints of a contract (see [next section][abi per item section]).
> Note: using `#[generate_trait]` in addition to the `#[abi(embed_v0)]` attribute for a public impl block is not recommended, as it will result in a failure to generate the corresponding ABI. Public functions should only be defined in an impl block annotated with `#[generate_trait]` if this block is also annotated with the `#[abi(per_item)]` attribute.
[abi per item section]: ./ch14-02-contract-functions.md#4-abiper_item-attribute
`[abi(per_item)]` Attribute
You can also define the entrypoint type of functions individually inside an impl block using the`#[abi(per_item)]` attribute on top of your impl. It is often used with the `#[generate_trait]` attribute, as it allows you to define entrypoints without an explicit interface. In this case, the functions will not be grouped under an impl in the ABI. Note that when using `#[abi(per_item)]` attribute, public functions need to be annotated with the `#[external(v0)]` attribute - otherwise, they will not be exposed and will be considered as private functions.
Here is a short example:
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
no_listing_01_abi_per_item_attribute/src/lib.cairo}}
```
In the case of `#[abi(per_item)]` attribute usage without `#[generate_trait]`, it will only be possible to include `constructor`, `l1-handler` and `internal` functions in the trait implementation. Indeed, `#[abi(per_item)]` only works with a trait that is not defined as a Starknet interface. Hence, it will be mandatory to create another trait defined as interface to implement public functions.