Component Dependencies

Working with components becomes more complex when we try to use one component inside another. As mentioned earlier, a component can only be embedded within a contract, meaning that it's not possible to embed a component within another component. However, this doesn't mean that we can't use one component inside another. In this section, we will see how to use a component as a dependency of another component.

Consider a component called `OwnableCounter` whose purpose is to create a counter that can only be incremented by its owner. This component can be embedded in any contract, so that any contract that uses it will have a counter that can only be incremented by its owner.

The first way to implement this is to create a single component that contains both counter and ownership features from within a single component. However, this approach is not recommended: our goal is to minimize the amount of code duplication and take advantage of component reusability. Instead, we can create a new component that _depends_ on the `Ownable` component for the ownership features, and internally defines the logic for the counter.

Listing {{#ref ownable_component}} shows the complete implementation, which we'll break down right after:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_03_component_dep/src/counter.cairo:full}}
```

{{#label ownable_component}}
<span class="caption">Listing {{#ref ownable_component}}: An OwnableCounter Component</span>

Specificities

Specifying Dependencies on Another Component

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_03_component_dep/src/counter.cairo:component_signature}}
```

In [chapter 8][cairo traits], we introduced trait bounds, which are used to specify that a generic type must implement a certain trait. In the same way, we can specify that a component depends on another component by restricting the `impl` block to be available only for contracts that contain the required component.

In our case, this is done by adding a restriction `impl Owner: ownable_component::HasComponent<TContractState>`, which indicates that this `impl` block is only available for contracts that contain an implementation of the `ownable_component::HasComponent` trait. This essentially means that the `TContractState' type has access to the ownable component. See [Components under the hood][component impl] for more information.

Although most of the trait bounds were defined using [anonymous parameters][anonymous generic impl operator], the dependency on the `Ownable` component is defined using a named parameter (here, `Owner`). We will need to use this explicit name when accessing the `Ownable`component within the`impl` block.

While this mechanism is verbose and may not be easy to approach at first, it is a powerful leverage of the trait system in Cairo. The inner workings of this mechanism are abstracted away from the user, and all you need to know is that when you embed a component in a contract, all other components in the same contract can access it.
[cairo traits]: ./ch08-02-traits-in-cairo.md
[component impl]: ch16-02-01-under-the-hood.md#inside-components-generic-impls
[anonymous generic impl operator]: ./ch08-01-generic-data-types md#anonymous-generic-implementation-parameter--operator

Using the Dependency

Now that we have made our `impl` depend on the `Ownable` component, we can access its functions, storage, and events within the implementation block. To bring the `Ownable` component into scope, we have two choices, depending on whether we intend to mutate the state of the `Ownable` component or not.
If we want to access the state of the `Ownable` component without mutating it, we use the `get_dep_component!` macro. If we want to mutate the state of the `Ownable` component (for example, change the current owner), we use the `get_dep_component_mut!` macro.
Both macros take two arguments: the first is `self`, either as a snapshot or by reference depending on mutability, representing the state of the component using the dependency, and the second is the component to access.

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_03_component_dep/src/counter.cairo:increment}}
```

In this function, we want to make sure that only the owner can call the `increment` function. We need to use
the `assert_only_owner` function from the `Ownable` component. We'll use the `get_dep_component!` macro which will return a snapshot of the requested component state, and call `assert_only_owner` on it, as a method of that component.
For the `transfer_ownership` function, we want to mutate that state to change the current owner. We need to use the `get_dep_component_mut!` macro, which will return the requested component state as a mutable reference, and call `transfer_ownership` on it.

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_03_component_dep/src/counter.cairo:transfer_ownership}}
```

It works exactly the same as `get_dep_component!` except that we need to pass the state as a `ref` so we can
mutate it to transfer the ownership.