Traits in Cairo

A trait defines a set of methods that can be implemented by a type. These methods can be called on instances of the type when this trait is implemented.

A trait combined with a generic type defines functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use _trait bounds_ to specify that a generic type can be any type that has certain behavior.

> Note: Traits are similar to a feature often called interfaces in other languages, although with some differences.

While traits can be written to not accept generic types, they are most useful when used with generic types. We already covered generics in the [previous chapter][generics], and we will use them in this chapter to demonstrate how traits can be used to define shared behavior for generic types.

[generics]: ./ch08-01-generic-data-types.md

Defining a Trait

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

For example, let's say we have a struct `NewsArticle` that holds a news story in a particular location. We can define a trait `Summary` that describes the behavior of something that can summarize the `NewsArticle` type.

```cairo,noplayground
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_14_simple_trait/src/lib.cairo:trait}}
```

{{#label first_trait_signature}}
 Listing {{#ref first_trait_signature}}: A `Summary` trait that consists of the behavior provided by a `summarize` method

In Listing {{#ref first_trait_signature}}, we declare a trait using the `trait` keyword and then the trait's name, which is `Summary` in this case.

We've also declared the trait as `pub` so that crates depending on this crate can make use of this trait too, as we'll see in a few examples.

Inside the curly brackets, we declare the method signatures that describe the behaviors of the types that implement this trait, which in this case is `fn summarize(self: @NewsArticle) -> ByteArray;`. After the method signature, instead of providing an implementation within curly brackets, we use a semicolon.

> Note: the `ByteArray` type is the type used to represent strings in Cairo.

As the trait is not generic, the `self` parameter is not generic either and is of type `@NewsArticle`. This means that the `summarize` method can only be called on instances of `NewsArticle`.

Now, consider that we want to make a media aggregator library crate named _aggregator_ that can display summaries of data that might be stored in a `NewsArticle` or `Tweet` instance. To do this, we need a summary from each type, and we'll request that summary by calling a summarize method on an instance of that type.

By defining the `Summary` trait on generic type `T`, we can implement the `summarize`

method on any type we want to be able to summarize.
```cairo,noplayground
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_15_traits/src/lib.cairo:trait}}
```
{{#label trait_signature}}
 Listing {{#ref trait_signature}}: A `Summary` trait that consists of the behavior provided by a `summarize` method for a generic type
Each type implementing this trait must provide its own custom behavior for the body of the method. The compiler will enforce that any type that implements the `Summary` trait will have the method `summarize` defined with this signature exactly.
A trait can have multiple methods in its body: the method signatures are listed one per line and each line ends in a semicolon.
Implementing a Trait on a Type
Now that we've defined the desired signatures of the `Summary` trait's methods, we can implement it on the types in our media aggregator. The following code shows an implementation of the `Summary` trait on the `NewsArticle` struct that uses the headline, the author, and the location to create the return value of `summarize`. For the `Tweet` struct, we define `summarize` as the username followed by the entire text of the tweet, assuming that tweet content is already limited to 280 characters.
```cairo,noplayground
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_15_traits/src/lib.cairo:impl}}
```
{{#label trait_impl}}
 Listing {{#ref trait_impl}}: Implementation of the `Summary` trait on `NewsArticle` and `Tweet`
Implementing a trait on a type is similar to implementing regular methods. The difference is that after `impl`, we put a name for the implementation, then use the `of` keyword, and then specify the name of the trait we are writing the implementation for.
If the implementation is for a generic type, we place the generic type name in the angle brackets after the trait name.
Note that for the trait method to be accessible, there must be an implementation of that trait visible from the scope where the method is called. If the trait is `pub` and the implementation is not, and the implementation is not visible in the scope where the trait method is called, this will cause a compilation error.
Within the `impl` block, we put the method signatures that the trait definition has defined. Instead of adding a semicolon after each signature, we use curly brackets and fill in the method body with the specific behavior that we want the methods of the trait to have for the particular type.
Now that the library has implemented the `Summary` trait on `NewsArticle` and `Tweet`, users of the crate can call the trait methods on instances of `NewsArticle` and `Tweet` in the same way we call regular methods. The only difference is that the user must bring the trait into scope as well as the

types. Here’s an example of how a crate could use our `aggregator` crate:
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_15_traits/src/lib.cairo:main}}
```
{{#label trait_main}}
This code prints the following:
```shell
{{#include ../listings/ch08-generic-types-and-traits/no_listing_15_traits/output.txt}}
```
Other crates that depend on the _aggregator_ crate can also bring the `Summary` trait into scope to implement `Summary` on their own types.
Default Implementations
Sometimes it’s useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. Then, as we implement the trait on a particular type, we can keep or override each method’s default behavior.
In Listing {{#ref default_impl}} we specify a default string for the `summarize` method of the `Summary` trait instead of only defining the method signature, as we did in Listing {{#ref trait_signature}}.
Filename: src/lib.cairo
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/listing_default_impl/src/lib.cairo:trait}}
```
{{#label default_impl}}
 Listing {{#ref default_impl}}: Defining a `Summary` trait with a default implementation of the `summarize` method
To use a default implementation to summarize instances of `NewsArticle`, we specify an empty `impl` block with `impl NewsArticleSummary of Summary<NewsArticle> {}`. Even though we’re no longer defining the `summarize` method on `NewsArticle` directly, we’ve provided a default implementation and specified that `NewsArticle` implements the `Summary` trait. As a result, we can still call the `summarize` method on an instance of `NewsArticle`, like this:
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/listing_default_impl/src/lib.cairo:main}}
```
This code prints `New article available! (Read more...)`.
Creating a default implementation doesn’t require us to change anything about the previous implementation of `Summary` on `Tweet`. The reason is that the syntax for overriding a default implementation is the same as the syntax for implementing a trait method that doesn’t have a default implementation.
Default implementations can call other methods in the same trait, even if those other methods don’t have a default implementation. In this way, a trait can provide a lot of useful functionality and only require implementors to specify a small part of it. For

example, we could define the `Summary` trait to have a `summarize_author` method whose implementation is required, and then define a `summarize` method that has a default implementation that calls the `summarize_author` method:
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/
no_listing_default_impl_self_call/src/lib.cairo:trait}}
```
To use this version of `Summary`, we only need to define `summarize_author` when we implement the trait on a type:
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/
no_listing_default_impl_self_call/src/lib.cairo:impl}}
```
After we define `summarize_author`, we can call `summarize` on instances of the `Tweet` struct, and the default implementation of `summarize` will call the definition of `summarize_author` that we've provided. Because we've implemented `summarize_author`, the `Summary` trait has given us the behavior of the `summarize` method without requiring us to write any more code.
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/
no_listing_default_impl_self_call/src/lib.cairo:main}}
```
This code prints `1 new tweet: (Read more from @EliBenSasson...)`.
Note that it isn't possible to call the default implementation from an overriding implementation of that same method.
<!-- TODO: NOT AVAILABLE IN CAIRO FOR NOW move traits as parameters here -->
<!-- ## Traits as parameters
Now that you know how to define and implement traits, we can explore how to use traits to define functions that accept many different types. We'll use the `Summary` trait we implemented on the `NewsArticle` and `Tweet` types to define a `notify` function that calls the `summarize` method
on its `item` parameter, which is of some type that implements the `Summary` trait. To do this, we use the `impl Trait` syntax.
Instead of a concrete type for the `item` parameter, we specify the `impl`
keyword and the trait name. This parameter accepts any type that implements the specified trait. In the body of `notify`, we can call any methods on `item`
that come from the `Summary` trait, such as `summarize`. We can call `notify`
and pass in any instance of `NewsArticle` or `Tweet`. Code that calls the function with any other type, such as a `String` or an `i32`, won't compile because those types don't implement `Summary`. -->
<!-- TODO NOT AVAILABLE IN CAIRO FOR NOW Using trait bounds to conditionally implement methods -->
Managing and Using External Trait
To use traits methods, you need to make sure the correct traits/implementation(s) are imported. In some cases you might need to import not only the trait but also the implementation if they are declared in separate modules.

If `CircleGeometry` implementation was in a separate module/file named _circle_, then to define `boundary` method on `Circle` struct, we'd need to import `ShapeGeometry` trait in the _circle_ module.
If the code were to be organized into modules like in Listing {{#ref external_trait}} where the implementation of a trait is defined in a different module than the trait itself, explicitly importing the relevant trait or implementation would be required.
```cairo,noplayground
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_17_generic_traits/src/lib.cairo}}
```
{{#label external_trait}}
 Listing {{#ref external_trait}}: Implementing an external trait
Note that in Listing {{#ref external_trait}}, `CircleGeometry` and `RectangleGeometry` implementations don't need to be declared as `pub`. Indeed, `ShapeGeometry` trait, which is public, is used to print the result in the `main` function. The compiler will find the appropriate implementation for the `ShapeGeometry` public trait, regardless of the implementation visibility.
Impl Aliases
Implementations can be aliased when imported. This is most useful when you want to instantiate generic implementations with concrete types. For example, let's say we define a trait `Two` that is used to return the value `2` for a type `T`. We can write a trivial generic implementation of `Two` for all types that implement the `One` trait, simply by adding twice the value of `one` and returning it. However, in our public API, we may only want to expose the `Two` implementation for the `u8` and `u128` types.
```cairo,noplayground
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/listing_impl_aliases/src/lib.cairo}}
```
{{#label impl-aliases}}
 Listing {{#ref impl-aliases}}: Using impl aliases to instantiate generic impls with concrete types
We can define the generic implementation in a private module, use an impl alias to instantiate the generic implementation for these two concrete types, and make these two implementations public, while keeping the generic implementation private and unexposed. This way, we can avoid code duplication using the generic implementation, while keeping the public API clean and simple.
Negative Impls
> Note: This is still an experimental feature and can only be used if `experimental-features = ["negative_impls"]` is enabled in your _Scarb.toml_ file, under the `[package]` section.
Negative implementations, also known as negative traits or negative bounds, are a mechanism that allows you to express that a type does not implement a certain trait when defining the implementation of a trait over a generic type. Negative impls enable you to write implementations that are applicable only when another implementation does not exist in the current scope.

For example, let's say we have a trait `Producer` and a trait `Consumer`, and we want to define a generic behavior where all types implement the `Consumer` trait by default. However, we want to ensure that no type can be both a `Consumer` and a `Producer`. We can use negative impls to express this restriction.
In Listing {{#ref negative-impls}}, we define a `ProducerType` that implements the `Producer` trait, and two other types, `AnotherType` and `AThirdType`, which do not implement the `Producer` trait. We then use negative impls to create a default implementation of the `Consumer` trait for all types that do not implement the `Producer` trait.
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_18_negative_impl/src/lib.cairo}}
```
{{#label negative-impls}}
 Listing {{#ref negative-impls}}: Using negative impls to enforce that a type cannot implement both `Producer` and `Consumer` traits simultaneously
In the `main` function, we create instances of `ProducerType`, `AnotherType`, and `AThirdType`. We then call the `produce` method on the `producer` instance and pass the result to the `consume` method on the `another_type` and `third_type` instances. Finally, we try to call the `consume` method on the `producer` instance, which results in a compile-time error because `ProducerType` does not implement the `Consumer` trait.
{{#quiz ../quizzes/ch08-02-traits.toml}}