Method Syntax
Methods are similar to functions: we declare them with the `fn` keyword and a name, they can have parameters and a return value, and they contain some code that’s run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct (or an enum which we cover in [Chapter 6][enums]), and their first parameter is always `self`, which represents the instance of the type the method is being called on.
Defining Methods
Let’s change the `area` function that has a `Rectangle` instance as a parameter and instead make an `area` method defined on the `Rectangle` struct, as shown in Listing {{#ref area-method}}
```cairo, noplayground
{{#include ../listings/ch05-using-structs-to-structure-related-data/
no_listing_01_define_methods/src/lib.cairo:all}}
```
{{#label area-method}}
Listing {{#ref area-method}}: Defining an `area` method on the `Rectangle` struct.
To define the function within the context of `Rectangle`, we start an `impl` (implementation) block for a trait `RectangleTrait` that defines the methods that can be called on a `Rectangle` instance. As impl blocks can only be defined for traits and not types, we need to define this trait first - but it's not meant to be used for anything else. Everything within this `impl` block will be associated with the `Rectangle` type. Then we move the `area` function within the `impl` curly brackets and change the first (and in this case, only) parameter to be `self` in the signature and everywhere within the body. In `main`, where we called the `area` function and passed `rect1` as an argument, we can instead use _method syntax_ to call the `area` method on our `Rectangle` instance. The method syntax goes after an instance: we add a dot followed by the method name, parentheses, and any arguments.
In the signature for `area`, we use `self: @Rectangle` instead of `rectangle: @Rectangle`.
Methods must have a parameter named `self`, for their first parameter, and the type of `self` indicates the type that method can be called on. Methods can take ownership of `self`, but `self` can also be passed by snapshot or by reference, just like any other parameter.
> There is no direct link between a type and a trait. Only the type of the `self` parameter of a method defines the type from which this method can be called. That means, it is technically possible to define methods on multiple types in a same trait (mixing `Rectangle` and `Circle` methods, for example). But **this is not a recommended practice** as it can lead to confusion.
The main reason for using methods instead of functions, in addition to providing method syntax, is for organization. We’ve put all the things we can do with an instance of a type in one `impl` block rather than making future users of our code search for capabilities of `Rectangle` in various places in the library we provide.
The `generate_trait` Attribute
If you are familiar with Rust, you may find Cairo's approach confusing because methods cannot be defined directly on types. Instead, you must define a [trait](./ch08-02-traits-in-

cairo.md) and an implementation of this trait associated with the type for which the method is intended.
However, defining a trait and then implementing it to define methods on a specific type is verbose, and unnecessary: the trait itself will not be reused.
So, to avoid defining useless traits, Cairo provides the `#[generate_trait]` attribute to add above a trait implementation, which tells to the compiler to generate the corresponding trait definition for you, and let's you focus on the implementation only.
Both approaches are equivalent, but it's considered a best practice to not explicitly define traits in this case.
The previous example can also be written as follows:
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_02_gen_trait/src/lib.cairo}}
```
Let's use this `#[generate_trait]` in the following chapters to make our code cleaner.
Snapshots and References
As the `area` method does not modify the calling instance, `self` is declared as a snapshot of a `Rectangle` instance with the `@` snapshot operator. But, of course, we can also define some methods receiving a mutable reference of this instance, to be able to modify it.
Let's write a new method `scale` which resizes a rectangle of a `factor` given as parameter:
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_03_references/src/lib.cairo:trait_impl}}
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_03_references/src/lib.cairo:main}}
```
It is also possible to define a method which takes ownership of the instance by using just `self` as the first parameter but it is rare. This technique is usually used when the method transforms `self` into something else and you want to prevent the caller from using the original instance after the transformation.
Look at the [Understanding Ownership](ch04-00-understanding-ownership.md) chapter for more details about these important notions.
Methods with Several Parameters
Let’s practice using methods by implementing another method on the `Rectangle` struct. This time we want to write the method `can_hold` which accepts another instance of `Rectangle` and returns `true` if this rectangle can fit completely within self; otherwise, it should return false.
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_04_some_params/src/lib.cairo:trait_impl}}
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_04_some_params/src/lib.cairo:main}}
```
Here, we expect that `rect1` can hold `rect2` but not `rect3`.

Associated functions

We call _associated functions_ all functions that are defined inside an `impl` block that are associated to a specific type. While this is not enforced by the compiler, it is a good practice to keep associated functions related to the same type in the same `impl` block - for example, all functions related to `Rectangle` will be grouped in the same `impl` block for `RectangleTrait`.

Methods are a special kind of associated function, but we can also define associated functions that don't have `self` as their first parameter (and thus are not methods) because they don't need an instance of the type to work with, but are still associated with that type.

Associated functions that aren't methods are often used for constructors that will return a new instance of the type. These are often called `new`, but `new` isn't a special name and isn't built into the language. For example, we could choose to provide an associated function named `square` that would have one dimension parameter and use that as both width and height, thus making it easier to create a square `Rectangle` rather than having to specify the same value twice:

Let's create the function `new` which creates a `Rectangle` from a `width` and a `height`, `square` which creates a square `Rectangle` from a `size` and `avg` which computes the average of two `Rectangle` instances:

```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/
no_listing_05_class_methods/src/lib.cairo:trait_impl}}
{{#include ../listings/ch05-using-structs-to-structure-related-data/
no_listing_05_class_methods/src/lib.cairo:main}}
```

To call the `square` associated function, we use the `::` syntax with the struct name; `let sq = Rectangle::square(3);` is an example. This function is namespaced by the struct: the `::` syntax is used for both associated functions and namespaces created by modules. We'll discuss modules in [Chapter 7][modules].

Note that the `avg` function could also be written as a method with `self` as the first rectangle. In this case, instead of using the method with `RectangleTrait::avg(@rect1, @rect2)`, it would be called with `rect1.avg(rect2)`.

Multiple Traits and `impl` Blocks

Each struct is allowed to have multiple `trait` and `impl` blocks. For example, the following code is equivalent to the code shown in the _Methods with several parameters_ section, which has each method in its own `trait` and `impl` blocks.

```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/
no_listing_06_multiple_traits/src/lib.cairo:here}}
```

There's no strong reason to separate these methods into multiple `trait` and `impl` blocks here, but this is valid syntax.

{{#quiz ../quizzes/ch05-03-method-syntax.toml}}

[enums]: ./ch06-01-enums.md

[modules]: ./ch07-02-defining-modules-to-control-scope.md