Paths for Referring to an Item in the Module Tree

To show Cairo where to find an item in a module tree, we use a path in the same way we use a path when navigating a filesystem. To call a function, we need to know its path.

A path can take two forms:

- An _absolute path_ is the full path starting from a crate root. The absolute path begins with the crate name.
- A _relative path_ starts from the current module.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (`::`).

To illustrate this notion let's take back our example Listing {{#ref front_of_house}} for the restaurant we used in the last chapter. We have a crate named _restaurant_ in which we have a module named `front_of_house` that contains a module named `hosting`. The `hosting` module contains a function named `add_to_waitlist`. We want to call the `add_to_waitlist` function from the `eat_at_restaurant` function. We need to tell Cairo the path to the `add_to_waitlist` function so it can find it.

Filename: src/lib.cairo

```cairo,noplayground
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_02_paths/src/lib.cairo:paths}}
```

{{#label path-types}}
Listing {{#ref path-types}}: Calling the `add_to_waitlist` function using absolute and relative paths

The `eat_at_restaurant` function is part of our library's public API, so we mark it with the `pub` keyword. We'll go into more detail about `pub` in the ["Exposing Paths with the `pub` Keyword"][pub] section.

The first time we call the `add_to_waitlist` function in `eat_at_restaurant`, we use an absolute path. The `add_to_waitlist` function is defined in the same crate as `eat_at_restaurant`. In Cairo, absolute paths start from the crate root, which you need to refer to by using the crate name. You can imagine a filesystem with the same structure: we'd specify the path _/front_of_house/hosting/add_to_waitlist_ to run the _add_to_waitlist_ program; using the crate name to start from the crate root is like using a slash (`/`) to start from the filesystem root in your shell.

The second time we call `add_to_waitlist`, we use a relative path. The path starts with `front_of_house`, the name of the module defined at the same level of the module tree as `eat_at_restaurant`. Here the filesystem equivalent would be using the path _./front_of_house/hosting/add_to_waitlist_. Starting with a module name means that the path is relative to the current module.

Let's try to compile Listing {{#ref path-types}} and find out why it won't compile yet! We get the following error:

```shell
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_02_paths/output.txt}}
```

The error messages say that module `hosting` and the `add_to_waitlist` function are

not visible. In other words, we have the correct paths for the `hosting` module and the `add_to_waitlist` function, but Cairo won't let us use them because it doesn't have access to them. In Cairo, all items (functions, methods, structs, enums, modules, and constants) are private to parent modules by default. If you want to make an item like a function or struct private, you put it in a module.

Items in a parent module can't use the private items inside child modules, but items in child modules can use the items in their ancestor modules. This is because child modules wrap and hide their implementation details, but the child modules can see the context in which they're defined. To continue with our metaphor, think of the privacy rules as being like the back office of a restaurant: what goes on in there is private to restaurant customers, but office managers can see and do everything in the restaurant they operate.

Cairo chose to have the module system function this way so that hiding inner implementation details is the default. That way, you know which parts of the inner code you can change without breaking outer code. However, Cairo does give you the option to expose inner parts of child modules' code to outer ancestor modules by using the `pub` keyword to make an item public.

[pub]: ./ch07-03-paths-for-referring-to-an-item-in-the-module-tree.md#exposing-paths-with-the-pub-keyword

Exposing Paths with the `pub` Keyword

Let's return to the previous error that told us the `hosting` module and the `add_to_waitlist` function are not visible. We want the `eat_at_restaurant` function in the parent module to have access to the `add_to_waitlist` function in the child module, so we mark the `hosting` module with the `pub` keyword, as shown in Listing {{#ref pub-keyword-not-compiling}}.

Filename: src/lib.cairo

```cairo,noplayground
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_03_pub_keyword/src/lib.cairo}}
```

{{#label pub-keyword-not-compiling}}
Listing {{#ref pub-keyword-not-compiling}}: Declaring the `hosting` module as `pub` to use it from `eat_at_restaurant`

Unfortunately, the code in Listing {{#ref pub-keyword-not-compiling}} still results in an error.

What happened? Adding the `pub` keyword in front of `mod hosting;` makes the module public. With this change, if we can access `front_of_house`, we can access `hosting`. But the contents of `hosting` are still private; making the module public doesn't make its contents public. The `pub` keyword on a module only lets code in its ancestor modules refer to it, not access its inner code. Because modules are containers, there's not much we can do by only making the module public; we need to go further and choose to make one or more of the items within the module public as well.

Let's also make the `add_to_waitlist` function public by adding the `pub` keyword before its definition, as in Listing {{#ref pub-keyword}}.

Filename: src/lib.cairo

```cairo,noplayground
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_04_pub_compiles/src/lib.cairo}}
```

{{#label pub-keyword}}
Listing {{#ref pub-keyword}}: Declaring the `hosting` module as `pub` to use it from `eat_at_restaurant`

Now the code will compile! To see why adding the `pub` keyword lets us use these paths in `add_to_waitlist` with respect to the privacy rules, let's look at the absolute and the relative paths.

In the absolute path, we start with the crate root, the root of our crate's module tree. The `front_of_house` module is defined in the crate root. While `front_of_house` isn't public, because the `eat_at_restaurant` function is defined in the same module as `front_of_house` (that is, `front_of_house` and `eat_at_restaurant` are siblings), we can refer to `front_of_house` from `eat_at_restaurant`. Next is the `hosting` module marked with `pub`. We can access the parent module of `hosting`, so we can access `hosting` itself. Finally, the `add_to_waitlist` function is marked with `pub` and we can access its parent module, so this function call works!

In the relative path, the logic is the same as the absolute path except for the first step: rather than starting from the crate root, the path starts from `front_of_house`. The `front_of_house` module is defined within the same module as `eat_at_restaurant`, so the relative path starting from the module in which `eat_at_restaurant` is defined works. Then, because `hosting` and `add_to_waitlist` are marked with `pub`, the rest of the path works, and this function call is valid!

{{#quiz ../quizzes/ch07-03-paths-in-module-tree-1.toml}}

Starting Relative Paths with `super`

We can construct relative paths that begin in the parent module, rather than the current module or the crate root, by using `super` at the start of the path. This is like starting a filesystem path with the `..` syntax. Using `super` allows us to reference an item that we know is in the parent module, which can make rearranging the module tree easier when the module is closely related to the parent, but the parent might be moved elsewhere in the module tree someday.

Consider the code in Listing {{#ref relative-path}} that models the situation in which a chef fixes an incorrect order and personally brings it out to the customer. The function `fix_incorrect_order` defined in the `back_of_house` module calls the function `deliver_order` defined in the parent module by specifying the path to `deliver_order` starting with `super`:

Filename: src/lib.cairo

```cairo,noplayground
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_05_super/src/lib.cairo}}
```

{{#label relative-path}}
Listing {{#ref relative-path}}: Calling a function using a relative path starting with `super`

Here you can see directly that you access a parent's module easily using `super`,

which wasn't the case previously.
Note that the `back_of_house` is kept private, as external users are not supposed to interact with the back of house directly.
Making Structs and Enums Public
We can also use `pub` to designate structs and enums as public, but there are a few extra details to consider when using `pub` with structs and enums.
- If we use `pub` before a struct definition, we make the struct public, but the struct's fields will still be private. We can make each field public or not on a case-by-case basis.
- In contrast, if we make an enum public, all of its variants are then public. We only need the `pub` before the `enum` keyword.
There's one more situation involving `pub` that we haven't covered, and that is our last module system feature: the `use` keyword. We'll cover `use` by itself first, and then we'll show how to combine `pub` and `use`.
{{#quiz ../quizzes/ch07-03-paths-in-module-tree-2.toml}}