Bringing Paths into Scope with the `use` Keyword
Having to write out the paths to call functions can feel inconvenient and repetitive. Fortunately, there's a way to simplify this process: we can create a shortcut to a path with the `use` keyword once, and then use the shorter name everywhere else in the scope.
In Listing {{#ref use-keyword}}, we bring the `crate::front_of_house::hosting` module into the
scope of the `eat_at_restaurant` function so we only have to specify
`hosting::add_to_waitlist` to call the `add_to_waitlist` function in
`eat_at_restaurant`.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_06_use/src/lib.cairo:use}}
```
{{#label use-keyword}}
<span class="caption">Listing {{#ref use-keyword}}: Bringing a module into scope with `use`</span>
Adding `use` and a path in a scope is similar to creating a symbolic link in the filesystem. By adding `use crate::front_of_house::hosting;` in the crate root, `hosting` is now a valid name in that scope, just as though the `hosting` module had been defined in the crate root.
Note that `use` only creates the shortcut for the particular scope in which the `use` occurs. Listing {{#ref use-scope}} moves the `eat_at_restaurant` function into a new child module named `customer`, which is then a different scope than the `use` statement, so the function body won't compile:
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_07_use_and_scope/src/lib.cairo:wrong-path}}
```
{{#label use-scope}}
<span class="caption">Listing {{#ref use-scope}}: A `use` statement only applies in the scope it's in.</span>
The compiler error shows that the shortcut no longer applies within the `customer` module:
```shell
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_07_use_and_scope/output.txt}}
```
Creating Idiomatic `use` Paths
In Listing {{#ref use-keyword}}, you might have wondered why we specified `use crate::front_of_house::hosting`
and then called `hosting::add_to_waitlist` in `eat_at_restaurant` rather than specifying the `use` path all the way out to
the `add_to_waitlist` function to achieve the same result, as in Listing {{#ref unidiomatic-

use}}.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_08_unidiomatic_use/src/lib.cairo:unidiomatic-path}}
```
{{#label unidiomatic-use}}
<span class="caption">Listing {{#ref unidiomatic-use}}: Bringing the `add_to_waitlist` function into scope with `use`, which is unidiomatic</span>
Although both Listing {{#ref use-keyword}} and {{#ref unidiomatic-use}} accomplish the same task, Listing {{#ref use-keyword}} is
the idiomatic way to bring a function into scope with `use`. Bringing the
function's parent module into scope with `use` means we have to specify the
parent module when calling the function. Specifying the parent module when
calling the function makes it clear that the function isn't locally defined
while still minimizing repetition of the full path. The code in Listing {{#ref unidiomatic-use}} is
unclear as to where `add_to_waitlist` is defined.
On the other hand, when bringing in structs, enums, traits, and other items with `use`, it's idiomatic to specify the full path. Listing {{#ref idiomatic-use}} shows the idiomatic way to bring the core library's `BitSize` trait into the scope, allowing to call `bits` method to retrieve the size in bits of a type.
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_09_idiomatic_import/src/lib.cairo}}
```
{{#label idiomatic-use}}
<span class="caption">Listing {{#ref idiomatic-use}}: Bringing `BitSize` trait into scope in an idiomatic way</span>
There's no strong reason behind this idiom: it's just the convention that has
emerged in the Rust community, and folks have gotten used to reading and writing Rust code this way.
As Cairo shares many idioms with Rust, we follow this convention as well.
The exception to this idiom is if we're bringing two items with the same name
into scope with `use` statements, because Cairo doesn't allow that.
Providing New Names with the `as` Keyword
There's another solution to the problem of bringing two types of the same name
into the same scope with `use`: after the path, we can specify `as` and a new
local name, or _alias_, for the type. Listing {{#ref as-keyword}} shows how you can rename an import with `as`:
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_10_as_keyword/src/lib.cairo}}
```
{{#label as-keyword}}

<span class="caption">Listing {{#ref as-keyword}}: Renaming a trait when it’s brought into scope with the `as` keyword</span>
Here, we brought `ArrayTrait` into scope with the alias `Arr`. We can now access the trait's methods with the `Arr` identifier.
Importing Multiple Items from the Same Module
When you want to import multiple items (like functions, structs or enums)
from the same module in Cairo, you can use curly braces `{}` to list all of
the items that you want to import. This helps to keep your code clean and easy
to read by avoiding a long list of individual `use` statements.
The general syntax for importing multiple items from the same module is:
```cairo
use module::{item1, item2, item3};
```
Here is an example where we import three structures from the same module:
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_11_multiple_items/src/lib.cairo}}
```
{{#label multiple-imports}}
<span class="caption">Listing {{#ref multiple-imports}}: Importing multiple items from the same module</span>
Re-exporting Names in Module Files
When we bring a name into scope with the `use` keyword, the name available in
the new scope can be imported as if it had been defined in that code’s scope.
This technique is called _re-exporting_ because we’re bringing an item into scope,
but also making that item available for others to bring into their scope, with the `pub` keyword.
For example, let's re-export the `add_to_waitlist` function in the restaurant example:
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_12_pub_use/src/lib.cairo:reexporting}}
```
{{#label reexporting}}
<span class="caption">Listing {{#ref reexporting}}: Making a name available for any code to use from a new scope with `pub use`</span>
Before this change, external code would have to call the `add_to_waitlist`
function by using the path `restaurant::front_of_house::hosting::add_to_waitlist()`.
Now that this `pub use` has re-exported the `hosting` module from the root module, external code
can now use the path `restaurant::hosting::add_to_waitlist()` instead.
Re-exporting is useful when the internal structure of your code is different
from how programmers calling your code would think about the domain. For
example, in this restaurant metaphor, the people running the restaurant think
about “front of house” and “back of house.” But customers visiting a restaurant
probably won’t think about the parts of the restaurant in those terms. With

`pub use`, we can write our code with one structure but expose a different structure. Doing so makes our library well organized for programmers working on the library and programmers calling the library.

Using External Packages in Cairo with Scarb

You might need to use external packages to leverage the functionality provided by the community. Scarb allows you to use dependencies by cloning packages from their Git repositories. To use an external package in your project with Scarb, simply declare the Git repository URL of the dependency you want to add in a dedicated `[dependencies]` section in your _Scarb.toml_ configuration file. Note that the URL might correspond to the main branch, or any specific commit, branch or tag. For this, you will have to pass an extra `rev`, `branch`, or `tag` field, respectively. For example, the following code imports the main branch of _alexandria_math_ crate from _alexandria_ package:

```cairo
[dependencies]
alexandria_math = { git = "https://github.com/keep-starknet-strange/alexandria.git" }
```

while the following code imports a specific branch (which is deprecated and should not be used):

```cairo
[dependencies]
alexandria_math = { git = "https://github.com/keep-starknet-strange/alexandria.git",
branch = "cairo-v2.3.0-rc0" }
```

If you want to import multiple packages in your project, you need to create only one `[dependencies]` section and list all the desired packages beneath it. You can also specify development dependencies by declaring a `[dev-dependencies]` section.

After that, simply run `scarb build` to fetch all external dependencies and compile your package with all the dependencies included.

Note that it is also possible to add dependencies with the `scarb add` command, which will automatically edit the _Scarb.toml_ file for you. For development dependencies, just use the `scarb add --dev` command.

To remove a dependency, simply remove the corresponding line from your _Scarb.toml_ file, or use the `scarb rm` command.

{{#quiz ../quizzes/ch07-04-bringing-paths-into-scope.toml}}