Defining Modules to Control Scope

In this section, we'll talk about modules and other parts of the module system, namely _paths_ that allow you to name items and the `use` keyword that brings a path into scope.

First, we're going to start with a list of rules for easy reference when you're organizing your code in the future. Then we'll explain each of the rules in detail.

Modules Cheat Sheet

Here we provide a quick reference on how modules, paths and the `use` keyword work in the compiler, and how most developers organize their code. We'll be going through examples of each of these rules throughout this chapter, but this is a great place to refer to as a reminder of how modules work. You can create a new Scarb project with `scarb new backyard` to follow along.

- **Start from the crate root**: When compiling a crate, the compiler first looks in the crate root file (_src/lib.cairo_) for code to compile.
- **Declaring modules**: In the crate root file, you can declare new modules; say, you declare a "garden" module with `mod garden;`. The compiler will look for the module's code in these places:
  - Inline, within curly brackets that replace the semicolon following `mod garden`.
    ```cairo,noplayground
    // crate root file (src/lib.cairo)
    mod garden {
        // code defining the garden module goes here
    }
    ```
  - In the file _src/garden.cairo_.
- **Declaring submodules**: In any file other than the crate root, you can declare submodules. For example, you might declare `mod vegetables;` in _src/garden.cairo_. The compiler will look for the submodule's code within the directory named for the parent module in these places:
  - Inline, directly following `mod vegetables`, within curly brackets instead of the semicolon.
    ```cairo,noplayground
    // src/garden.cairo file
    mod vegetables {
        // code defining the vegetables submodule goes here
    }
    ```
  - In the file _src/garden/vegetables.cairo_.
- **Paths to code in modules**: Once a module is part of your crate, you can refer to code in that module from anywhere else in that same crate, using the path to the code. For example, an `Asparagus` type in the `vegetables` submodule would be found at `crate::garden::vegetables::Asparagus`.
- **Private vs public**: Code within a module is private from its parent modules by default. This means that it may only be

accessed by the current module and its descendants. To make a module public, declare it with `pub mod` instead of `mod`. To make items within a public module public as well, use `pub` before their declarations. Cairo also provides the `pub(crate)` keyword, allowing an item or module to be only visible within the crate in which the definition is included.
- **The `use` keyword**: Within a scope, the `use` keyword creates shortcuts to items to reduce repetition of long paths. In any scope that can refer to `crate::garden::vegetables::Asparagus`, you can create a shortcut with `use crate::garden::vegetables::Asparagus;` and from then on you only need to write `Asparagus` to make use of that type in the scope.

Here we create a crate named `backyard` that illustrates these rules. The crate's directory, also named `backyard`, contains these files and directories:

```text
backyard/
% % %  Scarb.toml
% % %  src
    % % %  garden
    %    % % %  vegetables.cairo
    % % %  garden.cairo
    % % %  lib.cairo
```

The crate root file in this case is _src/lib.cairo_, and it contains:

Filename: src/lib.cairo

```cairo
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/no_listing_01_lib/src/lib.cairo:crate}}
```

The `pub mod garden;` line imports the `garden` module. Using `pub` to make `garden` publicly accessible, or `pub(crate)` if you really want to make `garden` only available for your crate, is optional to run our program here, as the `main` function resides in the same module as `pub mod garden;` declaration. Nevertheless, not declaring `garden` as `pub` will make it not accessible from any other package.

This line tells the compiler to include the code it finds in _src/garden.cairo_, which is:

Filename: src/garden.cairo

```cairo,noplayground
pub mod vegetables;
```

Here, `pub mod vegetables;` means the code in *src/garden/vegetables.cairo* is included too. That code is:

```cairo,noplayground
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/no_listing_02_garden/src/lib.cairo}}
```

The line `use crate::garden::vegetables::Asparagus;` lets us bring the `Asparagus` type into scope,
so we can use it in the `main` function.

Now let’s get into the details of these rules and demonstrate them in action!
Grouping Related Code in Modules
Modules let us organize code within a crate for readability and easy reuse. Modules also allow us to control the privacy of items, because code within a module is private by default. Private items are internal implementation details not available for outside use. We can choose to make modules and the items within them public, which exposes them to allow external code to use and depend on them.
As an example, let’s write a library crate that provides the functionality of a restaurant. We’ll define the signatures of functions but leave their bodies empty to concentrate on the organization of the code, rather than the implementation of a restaurant.
In the restaurant industry, some parts of a restaurant are referred to as _front of house_ and others as _back of house_. Front of house is where customers are; this encompasses where the hosts seat customers, servers take orders and payment, and bartenders make drinks. Back of house is where the chefs and cooks work in the kitchen, dishwashers clean up, and managers do administrative work.
To structure our crate in this way, we can organize its functions into nested modules. Create a new package named _restaurant_ by running `scarb new restaurant`; then enter the code in Listing {{#ref front_of_house}} into _src/lib.cairo_ to define some modules and function signatures. Here’s the front of house section:
Filename: src/lib.cairo
```cairo,noplayground
{{#include ../listings/ch07-managing-cairo-projects-with-packages-crates-and-modules/listing_01_basic_nested_modules/src/lib.cairo:front_of_house}}
```

{{#label front_of_house}}
Listing {{#ref front_of_house}}: A `front_of_house` module containing other modules that then contain functions
We define a module with the `mod` keyword followed by the name of the module (in this case, `front_of_house`). The body of the module then goes inside curly brackets. Inside modules, we can place other modules, as in this case with the modules `hosting` and `serving`. Modules can also hold definitions for other items, such as structs, enums, constants, traits, and functions.
By using modules, we can group related definitions together and name why they’re related. Programmers using this code can navigate the code based on the groups rather than having to read through all the definitions, making it easier to find the definitions relevant to them. Programmers adding new functionality to this code would know where to place the code to keep the program organized.
Earlier, we mentioned that _src/lib.cairo_ is called the crate root. The reason for this name is that the content of this file forms a module named after the crate name at the root of the crate’s module structure, known as the _module tree_.
Listing {{#ref module-tree}} shows the module tree for the structure in Listing {{#ref front_of_house}}.
```text
```

```
restaurant
 % % %  front_of_house
    % % %  hosting
    %    % % %  add_to_waitlist
    %    % % %  seat_at_table
    % % %  serving
       % % %  take_order
       % % %  serve_order
       % % %  take_payment
```

{{#label module-tree}}
Listing {{#ref module-tree}}: The module tree for the code in Listing {{#ref front_of_house}}

This tree shows how some of the modules nest inside one another; for example, `hosting` nests inside `front_of_house`. The tree also shows that some modules are _siblings_ to each other, meaning they're defined in the same module; `hosting` and `serving` are siblings defined within `front_of_house`. If module A is contained inside module B, we say that module A is the _child_ of module B and that module B is the _parent_ of module A. Notice that the entire module tree is rooted under the explicit name of the crate _restaurant_.

The module tree might remind you of the filesystem's directory tree on your computer; this is a very apt comparison! Just like directories in a filesystem, you use modules to organize your code. And just like files in a directory, we need a way to find our modules.

{{#quiz ../quizzes/ch07-02-defining-modules-to-control-scope.toml}}