# Crates and Modules

Mustakimur R. Khandaker

# Introduction

To write large programs, organizing our code is critical.

- By grouping related functionality and separating code, we clarify where to find code that implements a particular feature.

In Rust, we can organize code by splitting it into multiple modules and then multiple files. A package (Cargo workspace) can contain multiple binary crates and optionally one library crate.

In addition, encapsulating implementation details lets us reuse code.

- Other code can communicate via the code's public interface.

These features, collectively referred to as the **module system**, include:

- **Packages:** A Cargo feature that lets us build, test, and share crates. (Have already discussed)
- **Crates:** A tree of modules that produces a library or executable. (Have already discussed)
- **Modules and use:** Let us control the organization, scope, and privacy of paths. (In this lecture)
- **Paths:** A way of naming an item, such as a struct, function, or module. (In this lecture)

# Modules

Modules let us organize code within a crate into groups for readability and easy reuse.

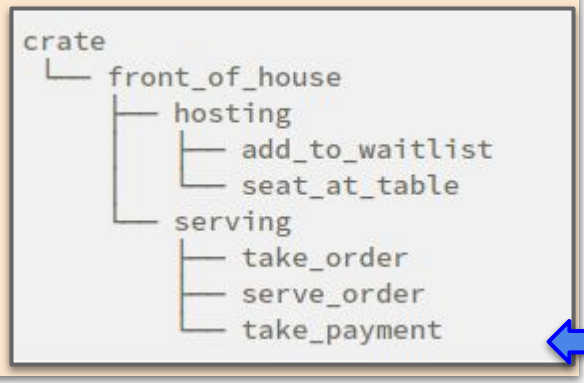
- Modules also control the privacy of items, which is whether an item can be used by outside code (public) or is an internal implementation detail and not available for outside use (private).

Let's create a new library named restaurant by running `cargo new --lib restaurant`.

The mod keyword is used create a new Module.

- Following the mod, we specify a name for the module and place curly brackets around the body of the module.
- Inside modules, we can have other modules.
- Modules can also hold definitions for other items:
  - structs
  - enums
  - constants
  - traits
  - functions.

crate at the root of the crate's module structure, known as the module tree.




```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

# Paths

To show Rust 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 (separated by double colons ::).

- **Absolute path** starts from a crate root by using a crate name or a literal crate.
- **Relative path** starts from the current module and uses self, super (acts like ..), or an identifier in the current module.

But, first we need to expose path using the pub keyword.

- Siblings/children in crate tree do not need an exposed path.
- By defaults, every content of any block is private.

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

```
fn serve_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
    }

    fn cook_order() {}
}
```

*The privacy rules apply to structs, enums, functions, and methods as well as modules.*

# Continue ...

In case of structure, pub keyword is required:

- For struct definition itself.
- For every field of that struct that needs to.

In case of enum, pub keyword is required just before enum definition.

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // Order a breakfast in the summer with Rye toast
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // Change our mind about what bread we'd like
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // The next line won't compile if we uncomment it; we're
not allowed
    // to see or modify the seasonal fruit that comes with the
meal
    // meal.seasonal_fruit = String::from("blueberries");
}
```

# use Keyword

We can bring a path into a scope once and then call the items in that path as if they're local items with the use keyword.

- Adding use and a path in a scope is similar to creating a symbolic link in the filesystem.
- Paths brought into scope with use also check privacy.

*You can also bring an item into scope with use and a relative path.*

Conventionally, we bring path upto parent for modules and functions; full path for structs, enums, and other items.

The exception to this idiom is if we're bringing two items with the same name into scope with use statements, because Rust doesn't allow that.

- E.g., If we specified `use std::fmt::Result` and `use std::io::Result`, we'd have two Result types in the same scope.

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;
// use self::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

# Continue ...

There's another solution of bringing two types of the same name into the same scope:

- after the path, we can specify as with a new local name (alias).

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

**Re-exporting Names with pub use:**
To enable the code that calls our code to refer to that name as if it had been defined in that code's scope, we combine pub and use.

- 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.

We can also shortening paths if nested. For example:

```
use std::cmp::Ordering;
use std::io;
```

```
use std::{cmp::Ordering, io};
```

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

If we want to bring all public items defined in a path into scope, we can specify that path followed by *, the glob operator.

```
use std::collections::*;
```

# < Crates and Modules />