Module system

Managing code structure in a growing project.
Testing and sharing code conveniently.

Module system consists of:

- **Packages**: A Cargo feature that lets you build, test, and share crates
- **Crates**: A tree of modules that produces a library or executable
- **Modules and `use`**: Let you control the organization, scope, and privacy of paths
- **Paths**: A way of naming an item, such as a struct, function, or module

Package structure

```
my-project
├── Cargo.lock                <-- actual dependencies' versions
├── Cargo.toml                <-- package configuration, dependency version requirements
└── src
    ├── configuration
    │   ├── run.rs
    │   └── mod.rs
    ├── lib.rs                <-- root of the lib crate
    ├── bin1
    │   ├── distribution.rs
    │   └── main.rs           <-- root of bin crate `bin1`
    └── bin2.rs               <-- root of bin crate `bin2`
```

Lib crates can be shared

- *crates.io* is the main crate repository.
- If you specify a dependency in `Cargo.toml`, it's fetched from `crates.io` automatically by *Cargo*.
- `lib.rs` is the root of a *lib crate*.

Binary crates can be executed

- `cargo run` executes the bin crate in your package.
- If you have multiple bin crates, you have to specify which to run:
  `cargo run --bin <bin_name>`
- Each bin crate in a package can import code from the lib crate there.

Modules: grouping related code (& encapsulation)

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

    // Alternatively, this could be located in `serving.rs` file and imported.
    mod serving {
        fn take_order() {}
        fn serve_order() {}
        fn take_payment() {}
    }
}
```

Modules: grouping related code (& encapsulation)

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Exports & imports

- exports: using privacy modifier (`pub` , `pub(crate)` , <no modifier>)

```
mod some_mod {
    struct ModulePublic;
    pub(super) struct ParentModulePublic;
    pub(crate) struct CratePublic;
    pub struct WorldPublic;
}
```

- imports: using `use` statement

```
use some_mod::CratePublic;
pub use some_mod::WorldPublic; // <-- re-export
```