**Bringing Paths into Scope with the use Keyword**

We can create a shortcut to a path with the use keyword once, and then use the shorter name everywhere else in the scope.

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, like any other paths.

src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

use crate::front\_of\_house::hosting;

pub fn eat\_at\_restaurant() {

hosting::add\_to\_waitlist();

}

**NOTE**

use only creates the shortcut for the particular scope in which the use occurs.

src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

use crate::front\_of\_house::hosting;

mod customer {

pub fn eat\_at\_restaurant() {

hosting::add\_to\_waitlist();

}

}

This will fail to build because the shortcut no longer applies within the customer module.

To fix this problem, move the use within the customer module too, or reference the shortcut in the parent module with *super::hosting* within the child customer module.

# Creating Idiomatic use Paths

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.

When bringing in structs, enums, and other items with use, it’s idiomatic to specify the full path.

It’s just the convention that has emerged.

Example

src/main.rs

use std::collections::HashMap;

fn main() {

let mut map = HashMap::new();

map.insert(1, 2);

}

**Exception**

If we are bringing two items with the same name into scope with use statements, because Rust doesn’t allow that.

src/lib.rs

use std::fmt;

use std::io;

fn function1() -> fmt::Result {

// --snip--

Ok(())

}

fn function2() -> io::Result<()> {

// --snip--

Ok(())

}

# 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.

src/lib.rs

use std::fmt::Result;

use std::io::Result as IoResult;

fn function1() -> Result {

// --snip--

Ok(())

}

fn function2() -> IoResult<()> {

// --snip--

Ok(())

}

# Re-exporting Names with pub use

When we bring a name into scope with the use keyword, the name available in the new scope is private. 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 can 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.

src/lib.rs

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();

}

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.

# Using External Packages

To use an external package rand in our project, we added this line to Cargo.toml:

Cargo.toml

rand = "0.8.3"

Adding rand as a dependency in Cargo.toml tells Cargo to download the rand package and any dependencies from crates.io and make rand available to our project.

Then, to bring rand definitions into the scope of our package, we added a use line starting with the name of the crate, rand, and listed the items we wanted to bring into scope.

use rand::Rng;

fn main() {

let secret\_number = rand::thread\_rng().gen\_range(1..=100);

}

The standard library is shipped with the Rust language, we don’t need to change Cargo.toml to include std.

# Using Nested Paths to Clean Up Large use Lists

We can use nested paths to bring multiple items into scope in one line.

We do this by specifying the common part of the path, followed by two colons, and then curly brackets around a list of the parts of the paths that differ.

// without nested path

use std::cmp::Ordering;

use std::io;

// with nested path

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

We can use a nested path at any level in a path, which is useful when combining two use statements that share a subpath.

// without nested path

use std::io;

use std::io::Write;

// with nested path

use std::io::{self, Write};

This line brings std::io and std::io::Write into scope.

# The Glob Operator

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::\*;

# References

<https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html#the-glob-operator>

# END OF FILE