Modularidad

Cuando nuestro programa empieza a crecer nos enfrentamos con un codigo desorganizada todo en un solo archivo. Por esta razon, Rust nos ayuda con la organización de nuestro codigo, agrupando y separando distintas responsabilidades, funciones y características.

Rust usa un sistema de crates, packages, módulos y rutas para mantener el código limpio y reutilizable.

- **Packages** (Paquetes): Una función de carga que le permite construir, probar y compartir cajas
- **Crates** (Cajas): Un árbol de módulos que produce una biblioteca o ejecutable
- **Modules** (Módulos): Le permiten controlar la organización, el alcance y la privacidad de Caminos
- **Paths** (Rutas): una forma de nombrar un elemento, como una estructura, función o módulo

Packages and Crates

Un crate es la unidad mínima de compilación en Rust.

- Puede ser un ejecutable *"binary crate"*.
- O puede ser una librería reutilizable *"library crate"*.

Cuando usas `cargo build`, Cargo compila un crate.

Tipos de crates

1. Binary crate
   - Contiene un main.rs.
   - Produce un ejecutable.

   Tiene un punto de entrada: la función `fn main()`

   Ejemplo: `cargo new hola_mundo`

   Esto crea un binary crate con src/main.rs.

   ```rust
   fn main() {
       println!("Hola, mundo!");
   }
   ```

   Se compila a un programa ejecutable.

2. Library crate
   - Contiene un lib.rs.
   - No tiene main().
   - Produce una librería que otros programas pueden usar.

   Ejemplo (lib.rs): `cargo new --lib mi_lib`

   Esto crea una library crate con src/lib.rs.

   ```rust
   pub fn suma(a: i32, b: i32) -> i32 {
       a + b
   }
   ```

   Se puede usar desde otro crate con:

   ```rust
   use mi_lib::suma;
   ```

Crates externos

Los crates externos son librerías publicadas en crates.io.

Se añaden en Cargo.toml:

```
[dependencies]
rand = "0.8"
```

Un Package es un conjunto de crates administrado con Cargo.

Cada package tiene un archivo Cargo.toml que describe:

- Dependencias.
- Nombre.
- Versión.
- Autores.

Ejemplo de estructura:

```
mi_proyecto/      <- package
 ├── Cargo.toml
 └── src/
      ├── main.rs <- binary crate
      └── lib.rs <- library crate
```

Module

- Los módulos sirven para organizar el código dentro de un crate.
- Agrupan funciones, structs, enums, etc.
- Permiten controlar la visibilidad (pub o privado).
- Evitan que todo quede en un solo archivo gigante.
- Se definen con `mod`.
- Pueden estar:
  - En el mismo archivo.
  - En archivos separados (mod nombre; busca nombre.rs o nombre/mod.rs).

Ejemplo en el mismo archivo:

```
mod matematicas {
    pub fn suma(a: i32, b: i32) -> i32 {
        a + b
    }

    fn resta(a: i32, b: i32) -> i32 {
        a - b
    }
}

fn main() {
    println!("Suma: 5 + 3 = {}", matematicas::suma(5, 3));
    // println!("Resta: {}", matematicas::resta(5, 3)); // ERROR: resta no es pública
}
```

Aquí `suma` es pública “`pub`” pero `resta` es privada.

El mismo ejemplo pero ahora usaremos `use` para acortar rutas:

```
mod matematicas {
    pub fn sumar(a: i32, b: i32) -> i32 {
        a + b
    }
}

use matematicas::sumar; // importamos

fn main() {
    println!("2 + 3 = {}", sumar(2, 3)); // más corto
}
```

También puedes importar todo:

```
use matematicas::*;
```

Varios archivos

Cuando contamos con mas archivos como por ejemplo este:

```
src/
 ├── main.rs
 ├── utilidades.rs
```

1. `main.rs`

   ```
   mod utilidades; // le decimos que existe ese archivo

   fn main() {
       utilidades::saludo("Luis");
   }
   ```

2. `utilidades.rs`

   ```
   pub fn saludo(nombre: &str) {
       println!("Hola, {}!", nombre);
   }
   ```

Ahora el código está separado.

Varios archivos en carpetas

Supongamos esta estructura:

```
src/
 ├── main.rs
 ├── matematicas/
 │     ├── mod.rs
 │     ├── suma.rs
 │     └── resta.rs
```

1. `main.rs`:

   ```
   mod matematicas;

   fn main() {
       println!("2 + 3 = {}", matematicas::suma::sumar(2, 3));
   }
   ```

2. `matematicas/mod.rs`:

```rust
pub mod suma;
pub mod resta;
```

3. `matematicas/suma.rs`:

```rust
pub fn sumar(a: i32, b: i32) -> i32 {
    a + b
}
```

4. `matematicas/resta.rs`:

```rust
pub fn restar(a: i32, b: i32) -> i32 {
    a - b
}
```

Asi conseguimos un módulo matematicas con submódulos suma y resta.

Submódulos

Los módulos pueden tener submódulos:

```rust
mod libreria {
    pub mod libros {
        pub fn listar() {
            println!("Lista de libros...");
        }
    }
}

fn main() {
    libreria::libros::listar();
}
```

Rutas (paths)

Las rutas indican cómo encontrar funciones, structs, enums, etc., dentro de los módulos.

Ruta absoluta: comienza desde la raíz del crate.

Ruta relativa: parte desde el módulo actual `self::` o `super::`.

Ejemplo:

```rust
mod animales {
    pub mod perro {
        pub fn ladrar() {
            println!("Guau!");
        }
    }
}

fn main() {
    // Ruta absoluta
    crate::animales::perro::ladrar();

    // Ruta relativa
    animales::perro::ladrar();
}
```

Use

Es como un atajo o importación de ruta. En vez de escribir una ruta larga cada vez `crate::modulo::submodulo::funcion`, con `use` puedes traer ese nombre al scope actual y usarlo directamente.

En palabras simples : `use` ahorra escribir y hace el código más legible.

Ejemplo sin `use`:

```
mod util {
    pub fn saludar() {
        println!("Hola!");
    }
}

fn main() {
    crate::util::saludar(); // siempre tengo que escribir la ruta completa
}
```

Ejemplo con `use`:

```
mod util {
    pub fn saludar() {
        println!("Hola!");
    }
}

use crate::util::saludar;

fn main() {
    saludar(); // gracias al use ya puedo llamarla directo
}
```

super::

- `super::` sirve para acceder al módulo padre desde un submódulo.
- Es como decir: *"sube un nivel en la jerarquía de módulos"*.

Supongamos esta estructura de archivos:

```
src/
 └── main.rs
```

Código:

```
mod padre {
    pub fn saludo_padre() {
        println!("👋 Hola desde el padre");
    }

    pub mod hijo {
        pub fn saludo_hijo() {
            println!("💪 Hola desde el hijo");
        }

        pub fn usar_padre() {
            // Usamos super:: para subir al módulo padre
            super::saludo_padre();
        }
```

```
    }
}

fn main() {
    padre::hijo::saludo_hijo();   // Llamada directa al hijo
    padre::hijo::usar_padre();    // El hijo usa una función del padre
}
```

- padre es el módulo principal.
- hijo es un submódulo dentro de padre.
- Desde hijo, usamos `super::saludo_padre()` para subir al módulo padre.

En consola:

```
👶 Hola desde el hijo
👋 Hola desde el padre
```

Ejemplo con varios niveles

Estructura:

```
src/
 └── main.rs
```

Código:

```
mod abuelo {
    pub fn saludo_abuelo() {
        println!("👴 Hola desde el abuelo");
    }

    pub mod padre {
        pub fn saludo_padre() {
            println!("👨 Hola desde el padre");
        }

        pub mod hijo {
            pub fn saludo_hijo() {
                println!("👶 Hola desde el hijo");
            }

            pub fn usar_padre_y_abuelo() {
                super::saludo_padre();       // sube al módulo padre
                super::super::saludo_abuelo(); // sube dos niveles
            }
        }
    }
}

fn main() {
    abuelo::padre::hijo::saludo_hijo();
    abuelo::padre::hijo::usar_padre_y_abuelo();
}
```

Salida:

```
👶 Hola desde el hijo

👨 Hola desde el padre
```

```
👴 Hola desde el abuelo
```

- `super::` : sube un nivel en la jerarquía de módulos.
- `super::super::` : sube dos niveles, y así sucesivamente.
- Se usa para organizar mejor el código y evitar rutas largas desde los submódulos.

self::

- `self::` significa "este mismo módulo".

Se usa para:

- Referirse a elementos dentro del mismo módulo (funciones, structs, etc.).
- Dejar más explícito que estás trabajando con el módulo actual.
- Evitar confusión cuando importas nombres desde fuera.

Muchas veces no es obligatorio escribir `self::`, pero ayuda a dar claridad en proyectos grandes.

Ejemplo básico:

```rust
mod calculadora {
    // Función pública
    pub fn suma(a: i32, b: i32) -> i32 {
        a + b
    }

    // Función pública que llama a otra dentro del mismo módulo
    pub fn operaciones() {
        let resultado = self::suma(2, 3); // self:: -> este mismo módulo
        println!("Resultado: {}", resultado);
    }
}

fn main() {
    calculadora::operaciones();
}
```

Aquí `self::suma` indica que estamos llamando a suma dentro del módulo calculadora.

Podríamos haber escrito solo suma(2, 3), pero `self::` lo hace explícito.

Ejemplo con submódulos:

```rust
mod libreria {
    pub fn info() {
        println!("📚 Bienvenido a la librería!");
    }

    pub mod libros {
        pub fn listar() {
            println!("Lista de libros...");
        }

        pub fn mostrar() {
            // Llamamos a otra función del mismo módulo
            self::listar();
        }
```

```
    }
}

fn main() {
    libreria::libros::mostrar();
}
```

Salida:

```
Lista de libros...
```

Aquí `self::listar()` aclara que la función listar está dentro del mismo módulo libros.

Ejemplo con use self::

A veces `self::` se usa junto con `use` para importar del mismo módulo:

```
mod util {
    pub fn limpiar() {
        println!("🧹 Limpiando...");
    }

    pub fn ejecutar() {
        // Importamos funciones del mismo módulo
        use self::limpiar;

        limpiar();
    }
}

fn main() {
    util::ejecutar();
}
```

- `self::` : módulo actual.
- Se usa para:
  - Llamar funciones o usar elementos dentro del mismo módulo.
  - Hacer explícita la ruta (especialmente en proyectos grandes).
  - Evitar ambigüedades con otros módulos.
- Muchas veces se puede omitir, pero es útil como buena práctica de claridad.

crate::

- `crate::` : significa la raíz del crate actual.
- Desde cualquier módulo dentro del crate, crate:: te permite navegar desde la raíz, sin importar dónde estés.

Es parecido a una ruta absoluta dentro de tu proyecto.

Mientras que `self::` y `super::` son rutas relativas.

Supongamos esta estructura de archivos:

```
src/
 └── main.rs
```

Código:

```rust
mod util {
    pub fn saludar() {
        println!("👋 Hola desde util!");
    }
}

fn main() {
    // Desde la raíz del crate (main.rs es la raíz)
    crate::util::saludar();
}
```

Aquí `crate::util::saludar()` es lo mismo que `util::saludar()`.

Pero si estuvieras en un submódulo más profundo, `crate::` siempre te lleva a la raíz.

Ejemplo con submódulos

Estructura:

```
src/
 └── main.rs
```

Codigo:

```rust
mod util {
    pub fn limpiar() {
        println!("🧹 Limpiando...");
    }

    pub mod avanzado {
        pub fn ejecutar() {
            // Desde aquí accedemos a la raíz del crate
            crate::util::limpiar();
        }
    }
}

fn main() {
    util::avanzado::ejecutar();
}
```

- Estamos en `util::avanzado`.
- Para llamar a `util::limpiar()` podríamos usar `super::super::util::limpiar()`, pero es más fácil y claro usar `crate::util::limpiar()`.

Ejemplo en librerías (lib.rs)

Estructura:

```
src/
 ├── lib.rs
 └── main.rs
```

lib.rs:

```rust
pub mod calculadora {
    pub fn sumar(a: i32, b: i32) -> i32 {
        a + b
    }
}
```

main.rs:

```
fn main() {
    let resultado = crate::calculadora::sumar(5, 3);
    println!("Resultado: {}", resultado);
}
```

Aquí `crate::` significa “este crate de librería es en la que estoy”.

Control de privacidad

Es privado por defecto y solo accesible dentro del mismo módulo.

Con `pub` podemos usarlo en archivos accesible desde fuera.

- pub(crate): accesible solo dentro del crate.
- pub(super): accesible solo desde el módulo padre.

Ejemplo basico:

```
mod a {
    pub(crate) fn solo_crate() {}
    pub(super) fn solo_modulo_padre() {}
}
```

Estructura del proyecto

```
src/
 ├── main.rs
 ├── utilidades/
 │     ├── mod.rs
 │     ├── texto.rs
 │     └── numeros.rs
```

1. `main.rs` : Aquí arrancamos el programa y usamos los módulos.

   ```
   mod utilidades; // importamos el módulo "utilidades"

   use utilidades::texto;
   use utilidades::numeros;

   fn main() {
       println!("=== Proyecto con módulos ===");

       // Usamos el módulo de texto
       texto::saludo("Luis");

       // Usamos el módulo de números
       let suma = numeros::sumar(5, 3);
       println!("5 + 3 = {}", suma);

       // Ejemplo con pub(crate) → se puede usar dentro del mismo crate
       let cuadrado = numeros::cuadrado(4);
       println!("4^2 = {}", cuadrado);

       // Ejemplo con pub(super) → lo uso desde el padre (mod.rs)
       utilidades::texto::imprimir_desde_super();
   }
   ```

2. `utilidades/mod.rs`

- Archivo principal del módulo utilidades.
- Aquí agrupamos los submódulos.

3. `utilidades/texto.rs` : Aquí jugamos con `pub` y `pub(super)`.

```rust
pub fn saludo(nombre: &str) {
    println!("Hola, {}!", nombre);
}

// Esta función es accesible solo para el módulo padre (mod.rs)
pub(super) fn secreto_super() {
    println!("Soy un secreto, solo visible desde utilidades::mod.rs");
}

// Esta función llama a la de arriba
pub fn imprimir_desde_super() {
    println!("Llamando a secreto_super desde texto:");
    super::texto::secreto_super(); // uso de pub(super)
}
```

4. `utilidades/numeros.rs` : Aquí jugamos con pub y pub(crate).

```rust
pub fn sumar(a: i32, b: i32) -> i32 {
    a + b
}

// pub(crate) → accesible solo dentro de este crate (no fuera del proyecto)
pub(crate) fn cuadrado(n: i32) -> i32 {
    n * n
}
```

Resultado cuando corremos con el siguiente comando: `cargo run`

Salida en consola:

```
=== Proyecto con módulos ===
Hola, Luis!
5 + 3 = 8
4^2 = 16
Llamando a secreto_super desde texto:
Soy un secreto, solo visible desde utilidades::mod.rs
```

- `pub` : cualquiera puede acceder (incluso otros proyectos).
- `pub(crate)` : solo accesible dentro del crate actual.
- `pub(super)` : accesible solo por el módulo padre inmediato.
- sin `pub` : solo dentro del mismo archivo.