Rust 2025

clase 8

Temario

- Mocking
- Linters
- Pending issues
- Programación concurrente
- Características avanzadas del lenguaje

Mocking

Unit testing + mocking

El mocking es una práctica/herramienta para crear objetos fake en nuestros tests, evitando la necesidad de armar un contexto muy grande o, cuando usamos libs de terceros y/o llamadas a otros servicios que nos proveen información donde no se puede simular todos los casos límites al momento de correr el test.

Unit testing + mocking

Ejemplo usando: faux -> https://crates.io/crates/faux

```
struct Calculadora{id:u8}
impl Calculadora {
    pub fn calcular_anio_nacimiento(&self, per: &Persona) -> u32{
        2025  - per.edad as u32
    }
}
struct Persona{
    dni:u8,
    edad:i32,
    calculadora: Calculadora,
}
impl Persona {
    fn calcular_anio_nacimiento(&self)-> u32{
        self.calculadora.calcular_anio_nacimiento(&self)
    }
    fn calcular_si_debe_aplicar_vacuna(&self, anio:u32)-> bool{
        if self.calcular_anio_nacimiento() > anio{true}else{false}
    }
}
```

Unit testing + mocking

```rust
#[derive(Default, Clone, Debug)]
struct Persona{
    dni:u8,
    edad:i32,
    calculadora: Calculadora,
}
impl PartialEq for Persona{
    fn eq(&self, other: &Self) -> bool {
        self.dni == other.dni
    }
}
impl Persona {
    fn calcular_anio_nacimiento (&self)-> u32{
        self.calculadora.calcular_anio_nacimiento (&self)
    }
    fn calcular_si_debe_aplicar_vacuna (&self, anio:u32)-> bool{
        if self.calcular_anio_nacimiento () > anio{true}else{false}
    }
}
```

Unit testing + mocking

```
#[faux::create]
#[derive(Default, Clone, Debug)]
struct Calculadora{
    id:u8
}

#[faux::methods]
impl Calculadora {
    pub fn calcular_anio_nacimiento(&self, per: &Persona) -> u32{
        2023  - per.edad as u32
    }
}
```

Unit testing + mocking

```
#[test]
fn calcular_si_debe_aplicar_vacuna_test_1(){
    let mut calc = Calculadora::faux();
    let mut per = Persona::default();
    faux::when!(
        calc.calcular_anio_nacimiento(per.clone()))
        .then_return(2001);
    per.calculadora = calc;
    let r = per.calcular_si_debe_aplicar_vacuna(2000);
    assert!(r);
    let r = per.calcular_si_debe_aplicar_vacuna(2001);
    assert!(!r, "Se esperaba false pero devolvio {}", r);
}
```

Unit testing + mocking

artículo para leer al respecto:

https://blog.logrocket.com/mocking-rust-mockall-alternatives/#mocking-in-unit-testing

Linters

Linters

Lint: regla que el código debe seguir

Linter: herramienta que chequea lints.

```
warning: unused variable: `semi`
 --> src/main.rs:3:9
  |
3 |     let semi = "Seminario Rust 2023";
  |         ^^^^ help: if this is intentional, prefix it with an underscore: `_semi`
  |
  = note: `#[warn(unused_variables)]` on by default
```

https://doc.rust-lang.org/rustc/lints/index.html

Linters: clippy

https://github.com/rust-lang/rust-clippy

cargo clippy

rustup component add clippy

Linters: clippy

```
fn main() {
    let valor = 10;
    if valor > 10{
        //hace algo con 10
    }else{
        if valor > 0 {
            //hace otra cosa porque es 0
        }
    }
}
```

Pending Issues

Alcance y visibilidad

Para definir a un elemento(`fn`, `struct`, `enum`, `mod`) como público se utiliza la palabra clave `pub`

delante de su definición, y esto indica que puede accedido desde fuera de donde fue declarado. En cambio si no se específica con `pub` el compilador de rust hará que sea privado y solo puede accederse en donde fue definido.

Alcance y visibilidad

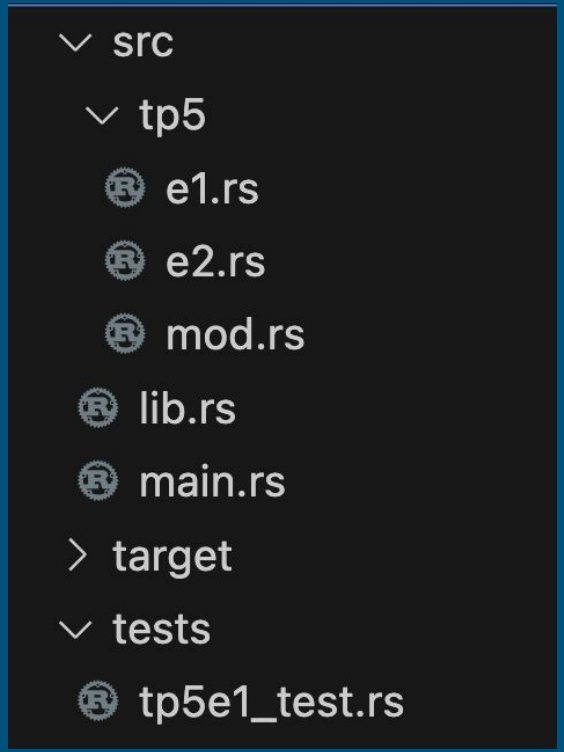
```
//ejemplo.rs
mod crate_helper_module {
    // esta función solo puede ser accedida en este rs
    pub fn crate_helper() {}

    //esta funcion solo puede ser accedida desde el scope de crate_helper_module
    fn implementation_detail() {}
}
// es es una funcion publica que puede ser accedida desde cualquier lugar, incluso
externamente
pub fn public_api() {}
```

Alcance y visibilidad

```
//ejemplo.rs
// Igual a public_api
pub mod submodule {
    use crate::ejemplo::crate_helper_module;
    pub fn my_method() {
        crate_helper_module::crate_helper();
    }
    // solo puede accederse en el scope de submodule
    fn my_implementation() {}
    #[cfg(test)]
    mod test {
        #[test]
        fn test_my_implementation() {
            // con super accedo jerarquicamente un nivel arriba de definiciones del módulo
            super::my_implementation();
        }
    }
}
```

Modularizando en cargo



Creando crates

```
//lib.rs
mod tp5;
use tp5::e1::Persona;
use tp5::e2::Vehiculo;
/// Crea un nuevo vehiculo y lo retorna
///Ejemplo
/// ```
/// use semi::nuevo_vehiculo;
/// let v = nuevo_vehiculo();
///
///```
pub fn nuevo_vehiculo() -> Vehiculo{
   Vehiculo { conductor: Persona {  }
    }
}
```

Creando crates

```
cargo doc --open
```

https://crates.io/me/

https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html

Atributos `#[]` o `#![]`

Es un metadato que se interpreta según la forma en que esté definido: `#[derive(Debug, Clone, ...)]`

`#[test]`

Los atributos internos, escritos con ! después de #, se aplican al elemento dentro del cual se declara el atributo. Los atributos externos, por el contrario, se aplican a lo que sigue al atributo.

https://doc.rust-lang.org/reference/attributes.html#built-in-attributes-index

Concurrencia

Concurrencia

Manejar la programación concurrente de manera segura y eficiente es otro de los principales objetivos de Rust. Tanto la programación concurrente, donde diferentes partes de un programa se ejecutan de forma independiente, y la programación paralela, donde diferentes partes de un programa se ejecutan al mismo tiempo, son cada vez más importantes a medida que más computadoras aprovechan sus múltiples procesadores.

Concurrencia: creando hilos

```rust
use std::thread;
use std::time::Duration;
fn main() {
    thread::spawn(||{
        println!("Soy el hijo #1");
    });
    thread::spawn(||{
        println!("Soy el hijo #2");
    });
    thread::sleep(Duration::from_millis(500));
    println!("soy el principal!");
}
```

Concurrencia: creando hilos

```
use std::thread;
fn main() {
    let handle1 = thread::spawn(||{
        println!("Soy el hijo #1");
    });
    let handle2 = thread::spawn(||{
        println!("Soy el hijo #2");
    });
    println!("soy el principal!");
    handle1.join().unwrap();
    handle2.join().unwrap();
}
```

Concurrencia: compartiendo data

```rust
use std::thread;

fn main() {
    let data = "Sem Rust!".to_string();
    let handle = thread::spawn(||{
        println!("Soy el hijo #1 {}", data);
    });
    println!("soy el principal! {}", data);
    handle.join().unwrap();
}
```

```
error[E0373]: closure may outlive the current function, but it borrows `data`, which is owned by the current function
 --> src/main.rs:5:32
  |
5 |     let handle = thread::spawn(||{
  |                                ^^ may outlive borrowed value `data`
6 |         println!("Soy el hijo #1 {}", data);
  |                                       ---- `data` is borrowed here
  |
```

Concurrencia: compartiendo data

```rust
use std::thread;
fn main(){
  let data = "Sem Rust!".to_string();
  let handle = thread::spawn(move ||{
       println!("Soy el hijo #1 {}", data);
  });
  println!("soy el principal! {}", data);
  handle.join().unwrap();
}
```

```
let data = "Sem Rust!".to_string();
    ---- move occurs because `data` has type `String`, which does not implement the `Copy` trait
let handle = thread::spawn(move ||{
                           ------- value moved into closure here
     println!("Soy el hijo #1 {}", data);
                                   ---- variable moved due to use in closure
});
println!("soy el principal! {}", data);
                                 ^^^^ value borrowed here after move
```

Concurrencia: compartiendo data

```
use std::thread;
use std::sync::{Arc,Mutex};
fn main(){
  let data = Mutex::new("Sem Rust!".to_string());
  let data_arc = Arc::new(data);
  let data_c1 = data_arc.clone();
  let handle = thread::spawn(move ||{
       let data_h = data_c1.lock().unwrap();
       println!("Soy el hijo #1 {}", *data_h);
  });
  println!("soy el principal! {}",*data_arc.lock().unwrap());
  handle.join().unwrap();
}
```

Concurrencia: compartiendo data

```
use std::thread;
use std::sync::{Arc,Mutex};
fn main(){
  let data = Mutex::new("Sem Rust!".to_string());
  let data_arc = Arc::new(data);
  let data_c1 = data_arc.clone();
  let handle = thread::spawn(move ||{
        let mut data_h = data_c1.lock().unwrap();
        *data_h = String::from("Seminario de Rust!");
        println!("Soy el hijo #1 {}", *data_h);
  });
  handle.join().unwrap();
  println!("soy el principal! {}",*data_arc.lock().unwrap());
}
```

Concurrencia: envío de mensajes entre hilos

```
use std::thread;
use std::sync::mpsc;
fn main(){
    let (tx, rx) = mpsc::channel();
    let handle = thread::spawn(move ||{
        let data= "Seminario de Rust!";
        tx.send(data).unwrap();
    });
    handle.join().unwrap();
    println!("soy el principal! {}",rx.recv().unwrap());
}
```

Concurrencia: envío de mensajes entre hilos

```
use std::thread;
use std::sync::mpsc;
fn main(){
    let (tx, rx) = mpsc::channel();
    let tx2 = tx.clone();
    thread::spawn(move ||{
        let data= "Seminario";
        tx.send(data).unwrap();
    });
    let handle2 = thread::spawn(move ||{
        let data= "de Rust!";
        tx2.send(data).unwrap();
    });
    handle2.join().unwrap();
    println!("soy el principal! {} {}",rx.recv().unwrap(), rx.recv().unwrap());
}
```

Concurrencia: async

```
use std::thread;
use std::time::Duration;
fn main(){
    for id in 1..3{
        println!("comenzando con tarea:{}", id);
        tarea_de_io_que_demora(id);
    }
}
fn tarea_de_io_que_demora(id:u8){
    thread::sleep(Duration::from_secs(2));
    println!("Termine! {}", id);
}
```

Concurrencia: async

```
use std::thread;
use std::time::Duration;
fn main(){
    for id in 1..3{
        println!("comenzando con tarea:{}", id);
        thread::spawn(move ||{
            tarea_de_io_que_demora(id);
        });
    }
}
fn tarea_de_io_que_demora(id:u8){
    println!("Termine! {}", id);
}
```

Concurrencia: async

```
use std::thread;
use std::time::Duration;
fn main(){
    for id in 1..3{
        println!("comenzando con tarea:{}", id);
        thread::spawn(move ||{
            tarea_de_io_que_demora(id);
        });
    }
    thread::sleep(Duration::from_secs(2));
}
fn tarea_de_io_que_demora(id:u8){
    println!("Termine! {}", id);
}
```

Concurrencia: async

```
fn main(){
    for id in 1..3{
        println!("comenzando con tarea:{}", id);
        let  r = tarea_de_io_que_demora(id).await;
    }
}
async fn tarea_de_io_que_demora(id:u8){
    println!("Termine! {}", id);
}
```

```
error[E0728]: `await` is only allowed inside `async` functions and blocks
 --> src/main.rs:5:44
  |
2 | fn main(){
  |    ---- this is not `async`
...
5 |         let  r = tarea_de_io_que_demora(id).await;
  |                                            ^^^^^^ only allowed inside `async` functions and blocks
```

Concurrencia: async runtime

async runtimes:

tokio-> https://tokio.rs

async-std -> https://async.rs

smol -> https://github.com/smol-rs/smol

Concurrencia: async runtime -> tokio

```
#[tokio::main]
async fn main(){
    for id in 1..3{
        println!("comenzando con tarea: {}", id);
        let   r = tarea_de_io_que_demora(id).await;
    }
}
async fn tarea_de_io_que_demora(id:u8){
    println!("Termine!  {}", id);
}
```

Concurrencia: async runtime -> tokio

```
use tokio::join;
#[tokio::main]
async fn main(){
      println!("comenzando con tarea:{}", 1);
      let  r1 = tarea_de_io_que_demora(1);

      println!("comenzando con tarea:{}", 2);
      let  r2 = tarea_de_io_que_demora(2);
      join!(r1, r2);
}
async fn tarea_de_io_que_demora(id:u8){
  println!("Termine! {}", id);
}
```

Concurrencia: async runtime -> async-std

```
use async_std::task;
fn main(){
  for id in 1..3{
        println!("comenzando con tarea:{}", id);
        task::block_on(
              tarea_de_io_que_demora(id)
        );
  }
}
async fn tarea_de_io_que_demora(id:u8){
  println!("Termine! {}", id);
}
```

Concurrencia: async runtime -> async-std

```
use async_std::task;
use std::{thread, time::Duration};
fn main(){
    let mut v = Vec::new();
    for id in 1..3{
        println!("comenzando con tarea:{}", id);
        let s = task::spawn(tarea_de_io_que_demora(id));
        v.push(s);
    }
    for i in v{
        task::block_on(i);
    }
}
async fn tarea_de_io_que_demora(id:u8){
    println!("Termine! {}", id);
    thread::sleep(Duration::from_secs(2));
}
```

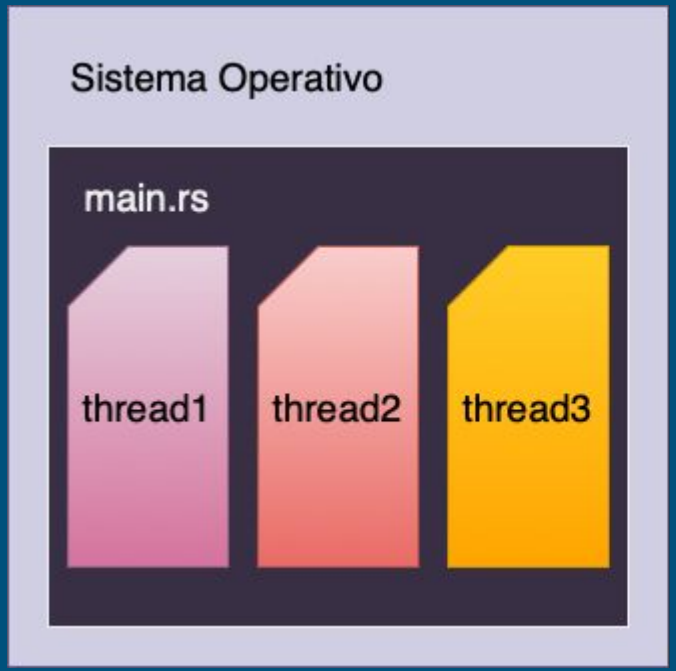
Concurrencia: async runtime -> smol

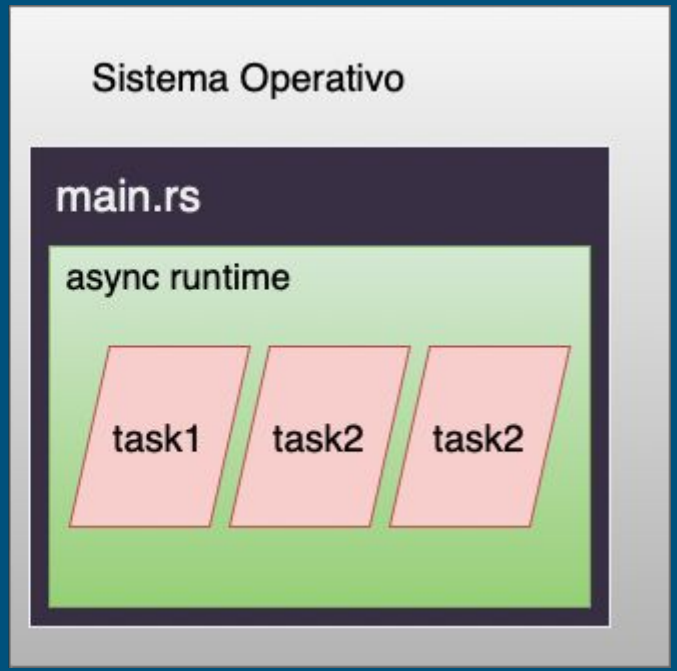
```
fn main(){
    for id in 1..3{
        println!("comenzando con tarea:{}", id);
        let r = smol::block_on(
            tarea_de_io_que_demora(id));
    }
}
async fn tarea_de_io_que_demora(id:u8){
    println!("Termine! {}", id);
}
```

Concurrencia: async runtime -> smol

```
use std::{thread, time::Duration};
use async_executor::Executor;
use futures_lite::future;
fn main(){
    let ex = Executor::new();
    let t1 = ex.spawn(tarea_de_io_que_demora(1));
    let t2= ex.spawn(tarea_de_io_que_demora(2));
    println!("t1: {:#?}", t1);
    future::block_on(ex.run(t1));
    println!("t2: {:#?}", t2);
    future::block_on(ex.run(t2));
}
async fn tarea_de_io_que_demora(id:u8){
    println!("Termine! {}", id);
    thread::sleep(Duration::from_secs(3));
}
```

Concurrencia: threads vs async runtime





Concurrencia: threads vs async runtime

- ➔ Número pequeño de tareas y cada una de ellas consume mucho cpu -> Threads
- ➔ Muchas tareas y con operaciones IO -> Async

Características avanzadas

Características avanzadas

- dyn
- Unsafe Rust
- Advanced traits
- Advanced types
- Advanced Functions and Closures

dyn: dynamic

La palabra clave dyn se usa para que llamadas a métodos en el trait asociado se resuelvan dinámicamente.

A diferencia de los parámetros genéricos o implementaciones de trait , el compilador no conoce el tipo concreto que se está pasando. Como tal, una referencia de dyn trait contiene dos punteros. Un puntero va a los datos (por ejemplo, una instancia de una estructura). Otro puntero va a un mapa de nombres de llamadas de método a punteros de función (conocido como tabla de método virtual o vtable).

Es probable que dyn trait produzca un código más pequeño que impl trait o parámetros genéricos, ya que el método no se duplicará para cada tipo concreto.

dyn: dynamic

```
pub trait Animal {
    fn hablar(&self) -> String;
}
struct Gato {}
struct Perro{}
impl Animal for Gato {
    fn hablar(&self) -> String {
        "Miauuuu!".to_string()
    }
}
impl Animal for Perro {
    fn hablar(&self) -> String {
        "Guauuuu!".to_string()
    }
}
```

dyn: dynamic

```
fn main() {
    let rand_n = 0.234;
    let animal = random_animal(rand_n);
    println!("El animal habla: {}", animal.hablar());
}
fn random_animal(random_number: f64) -> Animal{
    if random_number < 0.5 {
        Perro{}
    } else {
        Gato{}
    }
}
```

```
error[E0782]: trait objects must include the `dyn` keyword
 --> src/main.rs:6:40
  |
6 | fn random_animal(random_number: f64) ->Animal{
  |                                        ^^^^^^
  |
help: add `dyn` keyword before this trait
  |
6 | fn random_animal(random_number: f64) ->dyn Animal{
  |                                        +++
```

dyn: dynamic

```
fn random_animal(random_number: f64) ->Box<dyn Animal>{
    if random_number < 0.5 {
        Box::new(Perro{})
    } else {
        Box::new(Gato{})
    }
}
```

dyn: dynamic

```
fn main() {
    let mut animals:Vec<Box<dyn Animal>> = Vec::new();
    animals.push(Box::new(Perro{}));
    animals.push(Box::new(Gato{}));
    animals.push(Box::new(Gato{}));
    animals.push(Box::new(Gato{}));
    animals.push(Box::new(Perro{}));
    for a in animals{
        println!("{}", a.hablar());
    }
}
```

unsafe rust

Rust ofrece seguridad y la brinda por la inflexibilidad en las reglas que hemos visto, pero en determinados casos muy especiales esa flexibilidad nos limita para realizar código seguro que el compilador al ser tan rígido no nos dejaría, para ello, y que se vuelva flexible existe la sentencia `unsafe`
En la mayoría de los casos se utiliza para obtener una mejor performance.
más info:

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

https://blog.logrocket.com/unsafe-rust-how-and-when-not-to-use-it/

Advanced traits

Especificando un placeholder type: se puede asociar un tipo a un trait, esto permite una única implementación del trait para un tipo determinado.

Advanced traits

```
struct A;
pub trait Iterator1<T> {
    fn next1(&mut self) -> Option<T>;
}
impl Iterator1<i32> for A{
    fn next1(&mut self) -> Option<i32>{
        Some(8)
    }
}
impl Iterator1<f64> for A{
    fn next1(&mut self) -> Option<f64>{
        Some(8.0)
    }
}
```

Advanced traits

```
fn main() {
    let mut a = A{};
    let s1:i32 = a.next1().unwrap();
    let s2:f64 = a.next1().unwrap();
}
```

Advanced traits

```
pub trait Iterator2 {
    type Item;
    fn next2(&mut self) -> Option<Self::Item>;
}
impl Iterator2 for A {
    type Item = i32;
    fn next2(&mut self) -> Option<Self::Item>{
        Some(8)
    }
}
fn main() {
    let mut a = A{};
    let s= a.next2().unwrap();
}
```

Advanced types

Rust proporciona la capacidad de declarar un alias de tipo para dar otro nombre a un tipo existente. Para ello utilizamos la palabra clave type. Por ejemplo, podemos crear el alias Kilómetros para i32 así:

```
type Kilometros = i32;
fn main() {
    let x: i32 = 5;
    let y: Kilometros = 5;
    println!("x + y = {}", x + y);
}
```

Advanced types

```
use std::fmt;
use std::io::Error;
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

Advanced types

```
type Result<T> = std::result::Result<T, std::io::Error>;
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

https://www.youtube.com/watch?v=aI1oIFqh44Y

Advanced Functions and Closures

Punteros de funciones:

```
fn sumar_uno(x: i32) -> i32 {
    x + 1
}
fn dos_veces(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}
fn main() {
    let answer = dos_veces(sumar_uno, 5);
    println!("El resultado es: {}", answer);
}
```

Advanced Functions and Closures

```
struct Jv1{}
impl Jv1 {
    fn new()-> Self{ Self{}}
    fn imprimir(&self, data:String, opcion:i32){
        println!("{} con opcion:{}", data, opcion)
    }
    fn exec(&mut self, n:i32){
        if n==1{
            // varias lineas de codigo que hacen algo mas
            self.imprimir("ejecutando rutina a".to_string(), n);
        }else if n==2{
            // varias lineas de codigo que hacen algo mas
            self.imprimir("ejecutando rutina b".to_string(), n);
        }else if n==3{
            // varias lineas de codigo que hacen algo mas
            self.imprimir("ejecutando rutina c".to_string(), n);
        }
    }
```

Advanced Functions and Closures

```
fn main() {
    let mut j = Jv1::new();
    j.exec(2);
}
```

Advanced Functions and Closures

```
struct Jv2{
    hm:HashMap<i32, fn(i32, &mut Self)>,
}
impl Jv2 {
    fn new()-> Self{
        let mut j = Self{hm:HashMap::new()};
        j.armar_hm();
        j
    }
    fn imprimir(&self, data:String, opcion:i32){
        println!("{} con opcion: {}", data, opcion);
    }
```

Advanced Functions and Closures

```
fn exec(&mut self, n:i32){
        let f = self.hm.get(&n).expect("Error no existe key");
        f(n, self);
    }
    fn add_fn(&mut self, f:fn(i32, j: &mut Self), n:i32){
        self.hm.insert(n, f);
    }
    fn armar_hm(&mut self){
        self.add_fn(Self::rutina_a, 1);
        self.add_fn(Self::rutina_b, 2);
    }
    fn rutina_a(n:i32, j:&mut Self){
        j.imprimir("ejecutando rutina a ".to_string(), n);
    }
    fn rutina_b(n:i32, j:&mut Self){
        j.imprimir("ejecutando rutina b".to_string(), n);
    }
}
```

Advanced Functions and Closures

```
    fn get_method(&mut self, opcion:i32) -> &fn(i32, &mut Self){
        let f = self.hm.get(&opcion).expect("Error no existe key");
        f
    }


fn main() {
    let mut j = Jv2::new();
    j.exec(2);
    let m = j.get_method(2);
    m(2, &mut j);
}
```

Advanced Functions and Closures

```
fn returns_closure() -> fn(i32) -> i32 {
    |x| x + 1
}
fn main() {
    let c = returns_closure();
    println!("{}",c(5))
}
```

Macros

Macros

Macro: código que escribe código (metaprogramación)

- Hemos usado los macros println! y vec! . Todas estos macros se expanden para producir más código que el código que ha escrito manualmente.
- La metaprogramación es útil para reducir la cantidad de código que se tiene que escribir y mantener, que también es uno de los roles de las funciones. Sin embargo, las macros tienen algunos beneficios adicionales que las funciones no tienen.
- La firma de una función debe declarar el número y el tipo de parámetros que tiene la función. Las macros, por otro lado, pueden tomar un número variable de parámetros: podemos llamar a println!("hola") con un argumento o println!("hola {}", nombre) con dos argumentos por ej.
- La desventaja de implementar un macro en lugar de una función es que las definiciones de macro son más complejas porque está escribiendo código Rust que escribe código Rust. Debido a esta indirección, las definiciones de macros son más difíciles de leer, comprender y mantener que las definiciones de funciones.
- Se deben definir macros o incluirlos en el alcance antes de llamarlas en un archivo

Macros

```
macro_rules! nombre_del_macro {
    $regla1 => {};
    $regla2 => {};
    $regla3 => {};
    $reglaN => {};
}
```

Macros

```
#[macro_export]
macro_rules! mi_vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
fn main() {
    let v = mi_vec![1,2,3];
    println!("{:?}", v);
}
```

Macros

```
fn main() {
    let v =
        {
            let mut temp_vec = Vec::new();
            temp_vec.push(1);
            temp_vec.push(2);
            temp_vec.push(3);
            temp_vec
        };
    println!("{:?}", v);
}
```

mas info: https://veykril.github.io/tlborm/introduction.html