



Rust 2025



clase 5



Temario

- Traits
- POO
- Closures



Traits



Trait ¿Qué es?

- Es una funcionalidad particular que tiene un tipo y puede compartir con otros tipos.
- Podemos usar traits para definir comportamiento de manera abstracta.
- Se pueden usar traits como límites en tipos de datos genéricos para determinada funcionalidad que el tipo genérico debe cumplir.
- Son similares a las interfaces llamadas en otros lenguajes pero con algunas diferencias.

Trait: ejemplo I

```
pub trait MulI32 {  
    fn mul(&self, other:i32) -> f64; // abstracto  
    fn hace_algo_concreto(&self){ // por defecto  
        println!("hace_algo_concreto");  
    }  
}
```

Trait : ejemplo I

```
impl MulI32 for f64{  
    fn mul(&self, other:i32) -> f64{  
        self * other as f64  
    }  
}  
  
fn main(){  
    let v1 = 2.8;  
    let v2 = 4;  
    let r = v1.mul(v2);  
    println!("{}", r);  
}
```

Trait : ejemplo II

```
struct Perro{}  
struct Gato{}  
fn main(){  
    let gato = Gato{};  
    let perro = Perro{};  
    println!("{}", gato.hablar(), perro.hablar());  
}
```

Trait : ejemplo II

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


Trait : limitando un generic

```
pub fn imprimir_hablar<T: Animal>(animal: &T) {  
    println!("Hablo! {}", animal.hablar());  
}  
  
fn main(){  
    let gato = Gato{};  
    let perro = Perro{};  
    imprimir_hablar(&gato);  
    imprimir_hablar(&perro);  
}
```

Trait : como parámetro

```
pub fn imprimir_hablar(animall1: &impl Animal, animal2: &impl Animal) {  
    println!("Hablando! {} {}", animal1.hablar(), animal2.hablar());  
}  
  
fn main() {  
    let gato = Gato{};  
    let perro = Perro{};  
    imprimir_hablar(&gato, &perro);  
}
```

Trait : como parámetro

```
pub fn imprimir_hablar(animal: &impl Animal) {  
    println!("Hablo! {}", animal.hablar());  
}  
  
fn main(){  
    let gato = Gato{};  
    let perro = Perro{};  
    imprimir_hablar(&gato);  
    imprimir_hablar(&perro);  
}
```

Trait : múltiples

```
pub fn imprimir_hablar(animal: &(impl Animal + OtroTrait)) {  
    println!("Hablo! {}", animal.hablar());  
}  
  
fn main() {  
    let gato = Gato{};  
    imprimir_hablar(&gato);  
}
```

Trait : múltiples con where

```
pub fn imprimir_hablar<T>(animal: &T)
where
    T: Animal + OtroTrait
{
    println!("Hablo! {}", animal.hablar());
}

fn main(){
    let gato = Gato{};
    imprimir_hablar(&gato);
}
```



P00



P00: elementos

- Clases
- Métodos
- Atributos
- Objetos

P00: conceptos

- Encapsulamiento
- Abstracción
- Polimorfismo
- Herencia
- Modularidad

P00: conceptos

Encapsulamiento: permite agrupar comportamiento y datos, y restringirlos a través de su interfaz

POO: Encapsulamiento en rust

```
//definición en ejemplos.rs
pub struct Ejemplo{
    atr1:i32,
    atr2:i32,
}

impl Ejemplo {
    pub fn new(atr1:i32, atr2:i32) -> Ejemplo{
        Ejemplo{atr1,atr2}
    }

    pub fn calcular(&self)-> i32{
        self.atr1 * self.atr2
    }
}
```

POO: Encapsulamiento en rust

```
//main.rs
mod ejemplos;
fn main(){
    let mut e = Ejemplo::new(3,4);
    e.atr1 = 5;
}
```

error[E0616]: field `atr1` of struct `Ejemplo` is private

--> src/main.rs:168:7

168

| e.atr1 = 5;

^^^^

private field

P00: conceptos

Abstracción: refiere a poder representar un objeto del mundo real con sus características apropiadas y que este pueda comunicarse con otros objetos sin saber cómo están realizadas sus implementaciones.

POO: Abstracción en rust

```
//main.rs
```

```
mod ejemplos;
```

```
fn main(){
```

```
    let e = Ejemplo::new(3,4);
```

```
    e.calcular();
```

```
}
```

P00: conceptos

Polimorfismo: distintos tipos de objetos tienen la misma interfaz de comunicación pero su implementación es distinta. Es decir, entienden el mismo mensaje pero se comportan de manera diferente.

POO: polimorfismo en rust

```
//main.rs
use std::collections::LinkedList;
use std::collections::VecDeque;
fn main(){
    let mut list = LinkedList::new();
    let mut vecdeque = VecDeque::new();
    list.push_back(3);
    vecdeque.push_back(3);
    list.clear();
    vecdeque.clear();
}
```

POO: polimorfismo en rust

```
trait PushBack<T>{  
    fn push_back(&mut self, elt:T);  
}  
  
impl<T> PushBack<T> for LinkedList<T>{  
    fn push_back(&mut self, elt:T){  
        self.push_back(elt);  
        println!("acá podría ir otra lógica! sobre linkedlist!" );  
    }  
}  
  
impl<T> PushBack<T> for VecDeque<T>{  
    fn push_back(&mut self, elt:T){  
        self.push_back(elt);  
        println!("acá podría ir otra lógica! sobre vecdeque!" );  
    }  
}
```


POO: polimorfismo en rust

```
use std::collections::LinkedList;
use std::collections::VecDeque;
fn main(){
    let mut list = LinkedList::new();
    let mut vecdeque = VecDeque::new();
    PushBack::push_back(&mut vecdeque, 3);
    PushBack::push_back(&mut list, 3);
    println!("{:#?}", list);
    println!("{:#?}", vecdeque);
}
```

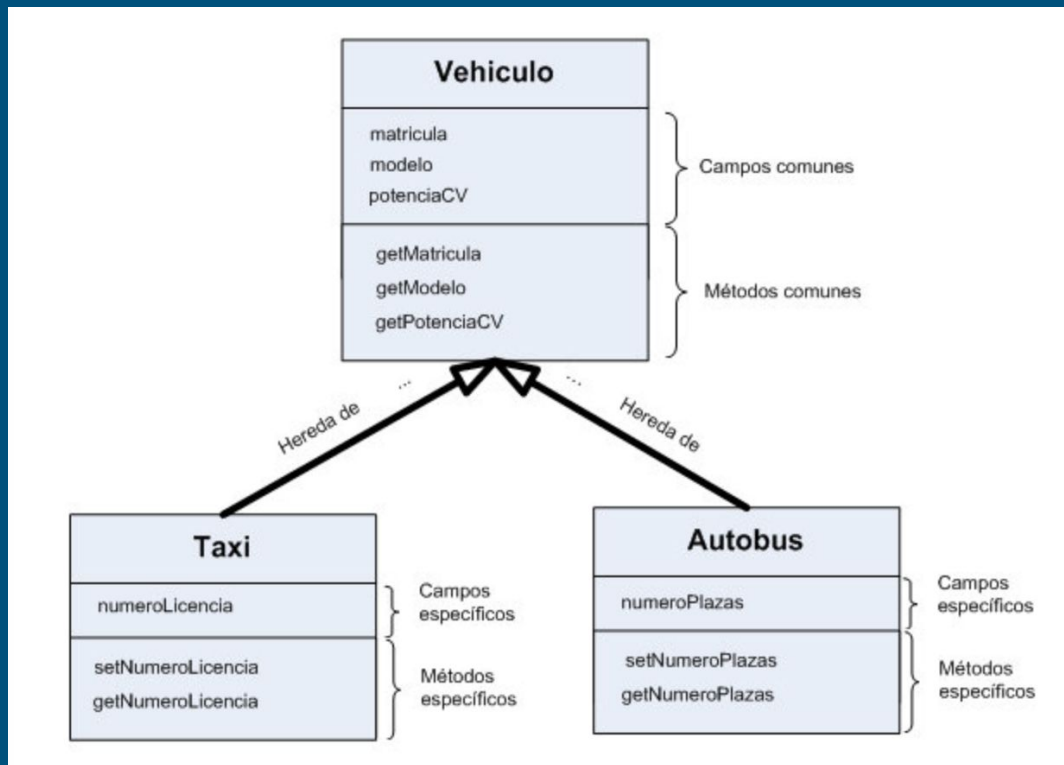
POO: herencia

La herencia es un mecanismo mediante el cual un objeto puede heredar elementos de la definición de otro objeto, obteniendo así los datos y el comportamiento del objeto principal sin tener que definirlos nuevamente.

Una de las razones principales para usar herencia es la reutilización del código ya que con ella se puede implementar un comportamiento particular para un tipo y permite reutilizar esa implementación para un tipo diferente o subtipo.

Si un lenguaje debe tener herencia para ser un lenguaje orientado a objetos, entonces Rust no lo es. No hay forma de definir una estructura que herede los campos de la estructura principal y las implementaciones de métodos.

POO: herencia-compartiendo comportamiento en rust



POO: herencia-compartiendo comportamiento en rust

```
struct DatosVehiculo {  
    matricula: String,  
    modelo: i32,  
    potencia: i32,  
}  
  
trait Vehiculo {  
    fn get_matricula (&self, datos: &DatosVehiculo) -> String {  
        datos.matricula.clone()  
    }  
    fn get_modelo (&self, datos: &DatosVehiculo) -> i32 {  
        datos.modelo  
    }  
    fn get_potencia (&self, datos: &DatosVehiculo) -> i32 {  
        datos.potencia  
    }  
}
```

POO: herencia-compartiendo comportamiento en rust

```
struct Taxi{  
    datos_vehiculo: DatosVehiculo,  
    numero_licencia: i32  
}  
  
impl Taxi {  
    fn new(numero_licencia:i32, matricula:String, modelo:i32, potencia:i32) -> Taxi{  
        Taxi{  
            datos_vehiculo: DatosVehiculo{matricula, modelo, potencia},  
            numero_licencia  
        }  
    }  
}  
  
impl Vehiculo for Taxi {}
```

POO: herencia-compartiendo comportamiento en rust

```
struct Autobus{
    datos_vehiculo: DatosVehiculo,
    numero_plazas:i32,
}

impl Autobus {
    fn new(numero_plazas:i32, matricula:String, modelo:i32, potencia:i32) -> Autobus{
        Autobus{
            datos_vehiculo: DatosVehiculo{matricula, modelo, potencia},
            numero_plazas
        }
    }
}

impl Vehiculo for Autobus {}
```





POO: herencia-compartiendo comportamiento en rust

```
fn main(){  
    let t = Taxi::new(  
        1, "u".to_string(), 2002, 145);  
    let mat_t = t.get_matricula(&t.datos_vehiculo);  
  
    let a = Autobus::new(  
        20, "u".to_string(), 2002, 145);  
    let mat_a = a.get_matricula(&a.datos_vehiculo);  
}
```

POO: modularidad en rust

Rust nos brinda esta característica, a través de la creación y uso de módulos, como lo estuvimos haciendo para resolver los tps, como así también la importación de libs (crates).

POO: en rust

- Abstracción y encapsulamiento 
- Polimorfismo 
- Herencia 
- Modularidad 



Closures



Closures: ¿Qué son?

Los closures son funciones anónimas que se pueden guardar en una variable o pasar como argumentos a otras funciones.

Se puede crear el closure en un lugar y luego llamarlo en otro para evaluar algo en un contexto diferente.

Permiten la reutilización de código

Closures: en el contexto

```
#[derive(Debug, PartialEq, Copy, Clone)]  
enum Color {  
    Rojo,  
    Azul,  
}  
struct Inventario {  
    remeras: Vec<Color>,  
}
```

Closures: en el contexto

```
impl Inventario {  
    fn get_color(&self, color_favorito: Option<Color>) -> Color {  
        color_favorito.unwrap_or_else(|| self.mas_stockeado())  
    }  
    fn mas_stockeado(&self) -> Color {  
        let mut num_rojo = 0;  
        let mut num_azul = 0;  
        for color in &self.remeras {  
            match color {  
                Color::Rojo => num_rojo += 1,  
                Color::Azul => num_azul += 1,  
            }  
        }  
        if num_rojo > num_azul {  
            Color::Rojo  
        } else {  
            Color::Azul  
        }  
    }  
}
```

Closures: en el contexto

```
fn main() {  
    let store = Inventario {  
        remeras: vec![Color::Azul, Color::Rojo, Color::Azul],  
    };  
    let u_pref1 = Some(Color::Rojo);  
    let get_color1 = store.get_color(u_pref1);  
    println!( "El usuario prefiere {:?} y obtiene {:?}",  
        u_pref1, get_color1  
    );  
    let u_pref2 = None;  
    let get_color2 = store.get_color(u_pref2);  
    println!( "El usuario prefiere: {:?} y obtiene {:?}",  
        u_pref2, get_color2  
    );  
}
```

Closures: inferencia de tipos

```
fn main() {  
    fn sumar_v1 (x: u32, y:u32) -> u32 { x + y }  
    let sumar_v2 = |x: u32, y:u32| -> u32 { x + y };  
    let sumar_v3 = |x, y| { x + y };  
    let sumar_v4 = |x, y| x + y ;  
  
    sumar_v1(3, 3);  
    sumar_v2(3, 3);  
    sumar_v3(3, 3);  
    sumar_v4(3, 3);  
}
```

Closures: inferencia de tipos

```
fn main() {  
    let sumer_v3 = |x, y| { x + y };  
    let sumar_v4 = |x, y| x + y ;  
  
    let c1 = Caja{dato:2};  
    let c2 = Caja{dato:5};  
  
    println!("{}", sumer_v3(c1, c2));  
    println!("{}", sumar_v4(c1, c2));  
}
```


Closures: inferencia de tipos

```
use std::ops::Add;

#[derive(Clone, Copy)]
struct Caja{
    dato:i32
}

impl Add for Caja{
    type Output = i32;

    fn add(self, otro: Self) -> i32 {
        self.dato + otro.dato
    }
}
```

Closures: referencias y ownership

```
fn main() {  
    let list = vec!["uno".to_string(), "dos".to_string(), "tres".to_string()];  
    println!("Antes de definir el closure: {:?}", list);  
  
    let pide_prestado = || println!("Desde el closure: {:?}", list);  
  
    println!("Antes de llamar al closure: {:?}", list);  
    pide_prestado();  
    println!("Luego de llamar al closure {:?}", list);  
}
```

Closures: referencias y ownership

```
fn main() {  
    let mut list = vec![1, 2, 3];  
    println!("Antes de definir el closure: {:?}", list);  
  
    let mut prestado_mutable = || list.push(7);  
  
    prestado_mutable();  
    println!("Luego de llamar al closure: {:?}", list);  
}
```

```
let mut list = vec![1, 2, 3];
```

[illegible]

Closures: como parámetros

```
#[derive(Debug)]  
struct Rectangulo {  
    ancho: u32,  
    alto: u32,  
}  
  
fn main() {  
    let mut arreglo = [  
        Rectangulo { ancho: 10, alto: 1 },  
        Rectangulo { ancho: 3, alto: 5 },  
        Rectangulo { ancho: 7, alto: 12 },  
    ];  
  
    arreglo.sort_by_key(|r| r.ancho);  
  
    println!("{:#?}", arreglo);  
}
```