Structs, traits y POO en Rust.

En este apartado vamos a tratar uno de los temas más densos de Rust, que son las estructuras, los traits y la orientación a objetos en general. Antes de empezar conviene hacerse una pregunta.

¿Es Rust orientado a objetos?.

La respuesta no es unánime. Bajo ciertas definiciones lo es, bajo otras no. Lo cierto es que aquí vamos a suponer que lo es, aunque no implemente ni **clases** ni **herencia**.

Structs en Rust.

Antes de avanzar a los objetos, vamos a ir con los **structs** o estructuras, similares a las que podemos tener en C. Las estructuras son datos agrupados por clave-valor.

```
struct Punto{
    x: i32,
    y: i32
}

struct Rectangulo{
    origen: Punto,
    ancho: i32,
    alto: i32
}

fn main(){
    let p = Punto {x: 50, y: 50};
    println!("Punto X: {}",p.x);
}
```

Se crean con la sintaxis de llave. Hasta aquí nada raro.

Pero en Rust, las estructuras pueden y deben llevar asociados métodos. Estos métodos se definen en las **traits**.

Piensa en ello como si fuesen interfaces. No obstante, algunas **traits** pueden tener implementación automática. Estas **traits** las llamaremos **derivables**. Usando la anotación

#[derive()] podemos indicarle a una estructura que derive de una **trait**. ¿Recuerdas las traits **Clone** y **Copy**? Son de este tipo y salvo que queramos cosas más especiales, con hacer una anotación nos bastará. ¿No te parece que Punto es una estructura que debe ser copiada en vez de movida? Pues simplemente añadimos las **traits** correspondientes.

```
#[derive(Copy,Clone)]
struct Punto{
    x: i32,
    y: i32
}

struct Rectangulo{
    origen: Punto,
    ancho: i32,
    alto: i32
}

fn main(){
    let p = Punto {x: 50, y: 50};
    println!("Punto X: {}",p.x);
}
```

Así, hemos modificado el comportamiento del **struct**.

Además, hemos ganado métodos en la estructura. Ahora podemos hacer p.clone() y obtener una copia. Esto ha sido gracias a la trait **Clone**, la cuál es necesaria para los elementos que quieran ser **Copy**. Esto además significa que podremos pasar la estructura a todas las funciones con genericidad cuya única condición sea que implementemos **Copy.**

Métodos asociados a la estructura

Hemos visto que **Clone** ha añadido un método a la estructura, como si fuese una clase de la que hubiéramos heredado. Nosotros también podemos definir métodos asociados a la estructura. Se hace con la palabra reservada **impl**.

```
#[derive(Copy,Clone)]
struct Punto{
    x: i32,
    y: i32
}

struct Rectangulo{
    origen: Punto,
    ancho: i32,
    alto: i32
}

impl Rectangulo{
    pub fn area(&self) -> i32{
        self.ancho*self.alto
    }
}

fn main(){
    let p = Punto {x: 50, y: 50};
    println!("Punto X: {}",p.x);
    let rectangulo = Rectangulo {origen: p,ancho: 20, alto: 20};
    println!("Área: {}",rectangulo.area());
}
```

Aquí hemos definido el método area asociado a Rectangulo.

Es importante destacar que aunque parezca una definición de clase, aquí no hay herencia posible y tampoco hay constructores. El primer argumento de la función area es **self**, o lo que es lo mismo, una referencia a la propia estructura.

Además vemos **pub**, lo que indica que es un método público.

Como hemos dicho, hay ciertas **traits** que se autoimplementan, pero otras requieren de código de integración.

Vamos a implementar dos **traits** en Rectangulo, **PartialOrd** para las comparaciones y **std::fmt::Display** para definir que se debe mostrar cuando hagamos **println!** a Rectangulo. **PartialOrd** a su vez requiere **PartialEq** presente.

```
use std::cmp::Ordering;

#[derive(Copy,Clone)]
struct Punto{
    x: i32,
    y: i32
}

struct Rectangulo{
    origen: Punto,
    ancho: i32,
    alto: i32
}

impl Rectangulo{
    pub fn area(&self) -> i32{
        self.ancho*self.alto
    }
}

impl PartialEq for Rectangulo{
    fn eq(&self,other: &Rectangulo) -> bool{
        self.area() == other.area()
    }
}

impl PartialOrd for Rectangulo{
    fn partial_cmp(&self, other: &Rectangulo) -> Option<Ordering>{
        if self.area() == other.area() {
            Some(Ordering::Equal)
        }else if self.area() > other.area() {
            Some(Ordering::Greater)
        }else{
            Some(Ordering::Less)
        }
    }
}

impl std::fmt::Display for Rectangulo{
    fn fmt(&self,f: &mut std::fmt::Formatter) -> std::fmt::Result{
        write!(f,"Origen: ({},{}) - Area:
{}",self.origen.x,self.origen.y,self.area())
    }

}

fn main(){
    let p = Punto {x: 50, y: 50};
    println!("Punto X: {}",p.x);
    let r1 = Rectangulo {origen: p,ancho: 20, alto: 20};
    println!("{}",r1);
```

```
    let r2 = Rectangulo {origen: Punto{x: 3, y: 4}, ancho: 30, alto: 30};
    println!("{}",r2);
    if r1 < r2 {
        println!("r2 es más grande");
    }

}
```

Aquí ya estamos viendo código Rust en estado puro. Sin embargo todavía no hemos visto como crear **traits** nosotros mismos.

Traits

Una **trait** es similar a una interfaz y deben de ser usadas mediante la composición, no la herencia. Piensa como si el objeto estuviese hecho con piezas de puzzle. Unas le dan una funcionalidad, otras otra y junto a código propio, se consigue crear un objeto completo. Así funciona la POO en Rust.

Imagina que existe la **trait** Pintable. Podremos crear métodos que admitan mediante genericidad restringida solo a objetos que implementen Pintable. Y si queremos que Rectangulo sea uno de ellos solo hay que implementar las funciones de Pintable sin definir. Pero Pintable puede incluir métodos ya hecho que funcionan con independencia del objeto en cuestión.
Esos métodos pueden ser reescritos si creemos conveniente.

```
impl Pintable for Rectangulo{
    fn pintar(&self) {
        // pintar rectangulo
    }
}

trait Pintable{
    fn pintar(&self);
    fn limpiar(){
        // limpiar pantalla
    }
}
```