



P00



P00

- Clases
- Objetos
- Métodos
- Atributos

P00: conceptos

- Abstracción y encapsulamiento
- 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);
}
```

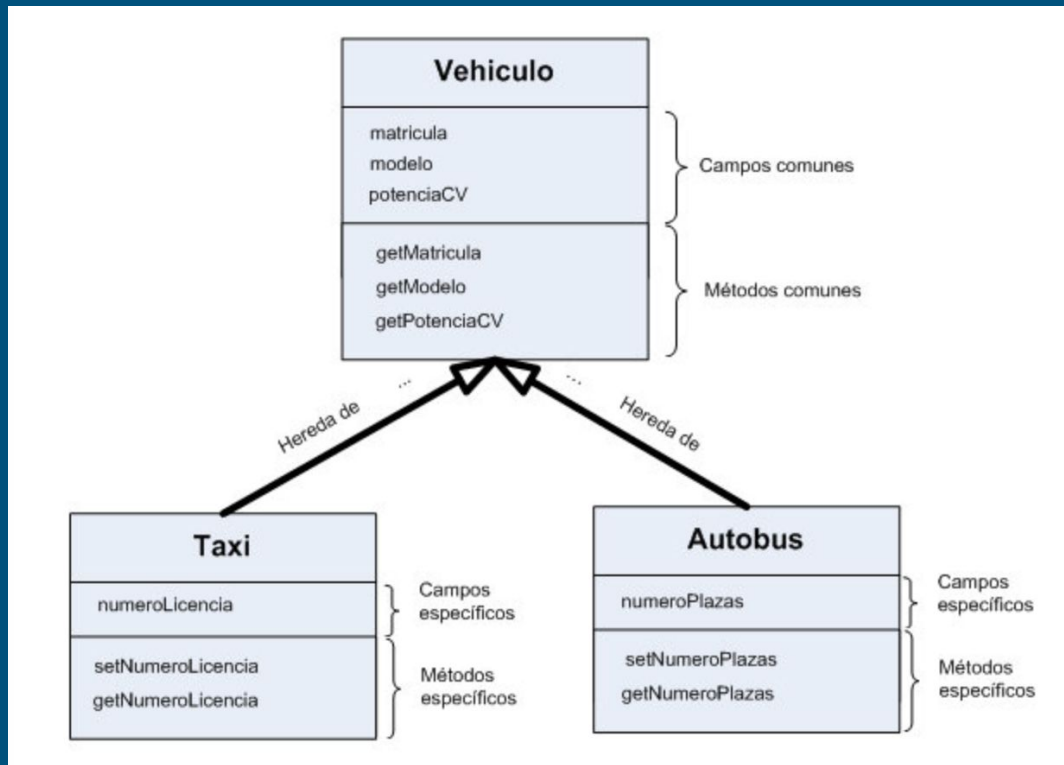
POO: herencia

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

```
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: 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: 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 