Rust 2023

clase 3

Temario

- Structs
- Enums
- Option

Structs

Structs: ¿Qué son?

Es un tipo de dato personalizado que permite empaquetar y nombrar valores relacionados que forman un conjunto de datos. Son similares, en la programación orientada a objetos al conjunto de atributos que tiene una clase.

Structs: ¿Cómo se definen?

Se definen con la palabra clave struct de la siguiente manera:

```
struct NombreDelStruct{
    nombre_atributo_1: tipo1,
    nombre_atributo_2: tipo2,
    nombre_atributo_n: tipo_n
}
```

Structs: ¿Cómo se definen?

Su definición no necesariamente tiene que ser dentro de la función main

```
struct Persona{
    nombre: String,
    apellido: String,
    dni: i32
}
fn main() {
   let persona1= Persona{
        nombre:"Lionel".to_string(),
        apellido: "Messi".to_string(),
        dni:1,
   };
   println!("nombre: {} apellido:{} dni:{}", persona1.nombre, persona1.apellido, persona1.dni);
}
```

Structs: init shorthand

```
fn main() {
  let persona1= nueva_persona(
        "Lionel".to_string(),
        "Messi".to_string(),
        1
  );
  println!("nombre: {} apellido:{} dni:{}", persona1.nombre, persona1.apellido, persona1.dni);
}
fn nueva_persona(nombre: String, apellido: String, dni: i32) -> Persona{
    Persona {
        apellido,
        dni,
        nombre,
    }
}
```

Structs: modificaciones

```
fn main() {
    let mut persona1= nueva_persona(
        "Lionel".to_string(),
        "Messi".to_string(),
        1
    );
    println!("nombre: {} apellido:{} dni:{}", persona1.nombre, persona1.apellido, persona1.dni);
    persona1.dni = 99;
    println!("nombre: {} apellido:{} dni:{}", persona1.nombre, persona1.apellido, persona1.dni);
}
```

Structs: creando instancias desde data de otra instancia

```
fn main() {
    let persona1= nueva_persona(
        "Lionel".to_string(),
        "Messi".to_string(),
        1
    );
    let persona2 = Persona{
        nombre:"Thiago".to_string(),
        ..persona1
    };
    println!("nombre: {} apellido:{} dni:{}", persona2.nombre, persona2.apellido, persona2.dni);
}
```

Structs: Tuple struct

```
struct Coordenada(f64, f64);

fn main() {
  let la_plata = Coordenada(-34.9213094, -57.9555699);
  println!("latitud: {} longirud:{}", la_plata.0, la_plata.1);
}
```

Structs: Implementando métodos

```
struct Coordenada(f64, f64);
impl Coordenada {
    fn es_la_plata(self) -> bool{
        let (latitud,longitud) = (-34.9213094, -57.9555699);
        if self.0==latitud && self.1 == longitud{
            return true;
        }
        false
    }
}
fn main() {
  let la_plata = Coordenada(-34.9213094, -57.9555699);
  println!("es la plata? {}", la_plata.es_la_plata());
}
```

Structs: Implementando métodos, otro ej

```
struct Rectangulo {
    ancho: u32,
    altura: u32,
}
impl  Rectangulo{
    fn area(self) -> u32{
        self.ancho * self.altura
    }
}
fn main() {
  let rec1 = Rectangulo{ancho:3, altura:7};
  println!("el area del rectangulo es: {}", rec1.area());
}
```

Structs: Implementando métodos, otro ej

```
fn main() {
  let rec1 = Rectangulo{ancho:3, altura:7};
  println!("rectangulo es:{}", rec1);
}
```

```
error[E0277]: `Rectangulo` doesn't implement `std::fmt::Display`
  --> src/main.rs:62:45
   |
62 |     println!("el area del rectangulo es:{}", rec1);
   |                                             ^^^^ `Rectangulo` cannot be formatted with the default form
atter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Rectangulo`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the
 macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
```

Structs: Implementando métodos, otro ej

```
fn main() {
    let rec1 = Rectangulo{ancho:3, altura:7};
    println!("el area del rectangulo es: {:?}", rec1);
}
```

```
error[E0277]: `Rectangulo` doesn't implement `Debug`
  --> src/main.rs:62:47
   |
62 |     println!("el area del rectangulo es:{:?}", rec1);
   |                                               ^^^^ `Rectangulo` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Rectangulo`
   = note: add `#[derive(Debug)]` to `Rectangulo` or manually `impl Debug for Rectangulo`
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the
 macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider annotating `Rectangulo` with `#[derive(Debug)]`
   |
14 | #[derive(Debug)]
```

Structs: Implementando métodos, otro ej

```
#[derive(Debug)]
struct Rectangulo {
    ancho: u32,
    altura: u32,
}
fn main() {
   let rec1 = Rectangulo{ancho:3, altura:7};
   println!("rectangulo es:{:?}", rec1);
}
```

Structs: funciones asociadas

Todas las funciones definidas dentro de un bloque impl se denominan funciones asociadas. Están asociadas con el tipo que lleva el nombre de impl.

Podemos definir funciones asociadas que no tienen self como su primer parámetro (y por lo tanto no son métodos) porque no necesitan una instancia del tipo para trabajar.

Las funciones asociadas que no son métodos a menudo se usan como constructores por ej. que devolverán una nueva instancia de la estructura. Estos a menudo se suelen definir como new, pero new no es un nombre especial y no está integrado en el lenguaje.

Structs: funciones asociadas ejemplos

```
impl Rectangulo{
    fn area(&self) -> u32{
        self.ancho * self.altura
    }
    fn new(ancho: u32, altura: u32) -> Rectangulo{
        Rectangulo { ancho, altura}
    }
}
fn main() {
  let rec1 = Rectangulo::new(3,7);
  println!("rectangulo es: {:?}", rec1);
  println!("el area del rectangulo es: {}", rec1.area());
}
```

Enums

Enums: enumeration

Es un tipo de dato que permite definir distintas variaciones.

Para definirlo se utiliza la siguiente sintaxis:

```
enum NombreEnum{
        VARIACION1,
        VARIACION2,
        VARIACION3,
        ...
}
```

Enums: ejemplos

```
enum Rol{
    PADRE,
    HIJO,
}
struct Persona{
    nombre: String,
    apellido: String,
    dni: i32,
    rol:Rol,
}
fn main() {
   let per1 = Persona{nombre:"Lionel".to_string(), apellido:"Messi".to_string(), dni:1, rol: Rol::PADRE,};
   println!("el rol de:{} es:{:?}", per1.nombre, per1.rol);
}
```

Enums: ejemplos => con valores

```
enum Rol{
    PADRE(i32),
    HIJO(i32),
}
fn main() {
   let per1 = Persona{
    nombre:"Lionel".to_string(),
    apellido:"Messi".to_string(),
    dni:1,
    rol: Rol::PADRE(5),
   };
   match  per1.rol{
        Rol::PADRE(valor) => println!("{}", valor),
         _ => (),
   };
}
```

Enums: ejemplos => con Struct

```
struct StructPadre{}
struct StructHijo{}
impl StructPadre {
    fn hace_algo(self){
        println!("soy un padre que hace algo");
    }
}
impl StructHijo {
    fn hace_algo(self){
        println!("soy un hijo que hace algo");
    }
}
```

Enums: ejemplos => con Struct cont..

```
enum Rol{
    PADRE(StructPadre),
    HIJO(StructHijo),
}
impl Rol{
    fn hace_algo(self){
        match self {
            Rol::PADRE(instancia) => instancia.hace_algo(),
            Rol::HIJO(instancia) => instancia.hace_algo(),
        }
    }
}
```

Enums: ejemplos => con Struct cont..

```
fn main() {
  let per1 = Persona{
    nombre:"Lionel".to_string(),
    apellido:"Messi".to_string(),
    dni:1,
    rol: Rol::PADRE(StructPadre{}),
  };
  per1.rol.hace_algo();
}
```

Option

Option: ¿Qué es?

Option es un enum que está disponible es la lib standard

Este enum tiene 2 posibles variantes que son Some() y None

Rust nos obliga a que en caso de que tengamos algún campo que no sepamos el valor, es decir vacío o nulo tenemos que manejar explícitamente y de manera obligatoria en código el caso. De esta forma se evitan los errores del tipo Null Pointer Exception de otros lenguajes.

Option: Ejemplo I

```
struct Persona{
    nombre: String,
    apellido: String,
    dni: Option<i32>,
    rol:Rol,
}
impl Persona {
    fn new(nombre:String, apellido:String, rol:Rol, dni:Option<i32>) -> Persona{
        Persona{
            nombre,
            apellido,
            dni,
            rol
        }
    }
}
```

Option: Ejemplo I cont.

```
fn main() {
  let nombre = "Lionel".to_string();
  let apellido = "Messi".to_string();
  let rol = Rol::PADRE(StructPadre {});
  let dni = None;
  let per1 = Persona::new(nombre, apellido, rol, dni);
  println!("la persona:{} tiene el dni:{:?}", per1.apellido, per1.dni);
}
```

Option: Ejemplo I con valor

```
fn main() {
  let nombre = "Lionel".to_string();
  let apellido = "Messi".to_string();
  let rol = Rol::PADRE(StructPadre {});
  let dni = Some(1);
  let per1 = Persona::new(nombre, apellido, rol, dni);
  match per1.dni {
   Some(valor) => println!("el dni de: {} es: {}", per1.apellido, valor),
   None => println!("{} no tiene nro de dni registrado", per1.apellido)
  }
}
```

Option: Ejemplo II con otro struct

```
struct DNI{
    tipo: char,
    nro: u32,
}
struct Persona{
    nombre: String,
    apellido: String,
    dni: Option<DNI>,
    rol:Rol,
}
impl Persona {
    fn new(nombre:String, apellido:String, rol:Rol, dni:Option<DNI>) -> Persona{
        Persona{nombre, apellido, dni, rol}
    }
}
```

Option: Ejemplo II con otro struct cont.

```
fn main() {
  let nombre = "Lionel".to_string();
  let apellido = "Messi".to_string();
  let rol = Rol::PADRE(StructPadre {});
  let dni = Some(DNI{tipo:'A', nro:1});
  let per1 = Persona::new(nombre, apellido, rol, dni);
  match per1.dni {
   Some(valor) => println!("el dni de: {} es: {}", per1.apellido, valor.nro),
   None => println!("{} no tiene nro de dni registrado", per1.apellido)
  }
}
```

Option: Ejemplo II con otro struct cont.

```
fn main() {
  let nombre = "Lionel".to_string();
  let apellido = "Messi".to_string();
  let rol = Rol::PADRE(StructPadre {});
  let dni = Some(DNI{tipo:'A', nro:1});
  let per1 = Persona::new(nombre, apellido, rol, dni);
  if per1.dni.is_none(){
     println!("{} no tiene nro de dni registrado", per1.apellido);
  }else{
       println!("el dni de: {} es: {:?}", per1.apellido, per1.dni.unwrap());
  }
}
```

Option: if let

```
fn main() {
  let nombre = "Lionel".to_string();
  let apellido = "Messi".to_string();
  let rol = Rol::PADRE(StructPadre {});
  let dni = Some(DNI{tipo:'A', nro:1});
  let per1 = Persona::new(nombre, apellido, rol, dni);
  if let Some(data) = per1.dni  {
      println!("el dni de: {} es: {}", per1.apellido, data.nro);
  }else{
      println!("{} no tiene nro de dni registrado", per1.apellido);
  }
}
```

Option: let else

```
fn main() {
  let nombre = "Lionel".to_string();
  let apellido = "Messi".to_string();
  let rol = Rol::PADRE(StructPadre {});
  let dni = Some(DNI{tipo:'A', nro:1});
  let per1 = Persona::new(nombre, apellido, rol, dni);
  let Some(data) = per1.dni else{
       panic!("{} no tiene dni", per1.apellido);
  };
}
```

Option: while let

```
fn main() {
    let mut cantidad = Some(5);
    loop{
        match cantidad {
            Some(valor) => {
                if valor > 0 {
                    println!("{}", valor);
                    cantidad = Some(valor -1);
                }else{
                    cantidad = None;
                }
            },
            None => {break;}
        }
    }
}
```

Option: while let cont.

```
fn main() {
    let mut cantidad = Some(5);
    while let Some(valor) = cantidad  {
        if valor > 0{
            println!("{valor}");
            cantidad = Some(valor -1);
        }else{
            cantidad = None;
        }
    }
}
```