Lifetime

Lifetime

Cada referencia en Rust tiene una vida útil, que es el alcance para el cual esa referencia es válida.

La mayoría de las veces el tiempo de vida de las referencias se infieren al igual que la mayoría de las veces se infieren los tipos.

Lifetime es la manera que tiene el compilador de Rust de asegurar que un lugar de memoria es válido para una referencia.

Lifetime: ejemplos

```rust
fn main() {
    let dato1: &i32;
    {
        let otro_scope = 2;
        dato1 = &otro_scope;
    }
    println!("{}", dato1);
}
```

```
error[E0597]: `otro_scope` does not live long enough
  --> src/main.rs:15:17
   |
15 |         dato1 = &otro_scope;
   |                 ^^^^^^^^^^^ borrowed value does not live long enough
16 |
17 |     }
   |     - `otro_scope` dropped here while still borrowed
18 |     println!("{}", dato1);
   |                    ----- borrow later used here
```

Lifetime: ejemplos

```
fn main() {
    let d1 = "str1";
    let d2 = "str2";
    let r = crear(d1, d2);
    println!("{}", r.as_str());
}
fn crear(data1: &str, data2: &str)-> &String{
    let resultado:String = data1.to_string().add(data2);
    &resultado
}
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:18:38
   |
18 | fn crear(data1: &str, data2: &str)-> &String{
   |                 ----          ----     ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the signature does not say whether it
is borrowed from `data1` or `data2`
help: consider introducing a named lifetime parameter
   |
18 | fn crear<'a>(data1: &'a str, data2: &'a str)-> &'a String{
   |         ++++         ++              ++         ++
```

Lifetime: ejemplos

```rust
use std::{ops::Add};
fn main() {
    let d1 = "str1";
    let d2 = "str2";
    let r = crear(d1, d2);
    println!("{}", r);
}
fn crear<'a>(data1: &'a str, data2: &'a str)-> &'a str{
    let d1 = data1.to_string();
    let resultado:&str = d1.add(data2).as_str();
    &resultado
}
```

```
error[E0515]: cannot return value referencing temporary value
  --> src/main.rs:11:5
   |
10 |     let resultado:&str = d1.add(data2).as_str();
   |                          ------------- temporary value created here
11 |     &resultado
   |     ^^^^^^^^^^ returns a value referencing data owned by the current function
```

Lifetime: ejemplos

```rust
fn main() {
    let string1 = String::from("Seminario de:");
    let string2 = "Rust!!!";
    let result = mas_largo(string1.as_str(), string2);
    println!("El mas largo es:{}", result);
}
fn mas_largo(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:35
  |
9 | fn mas_largo(x: &str, y: &str) -> &str {
  |                 ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `
x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn mas_largo<'a>(x: &'a str, y: &'a str) -> &'a str {
  |             ++++     ++          ++          ++
```

Lifetime: ejemplos

```
fn mas_largo(x: &'a str, y: &'b str) -> &'???? str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime: ejemplos

```
fn main() {
    let string1 = String::from("Seminario de:");
    let string2 = "Rust!!!";
    let result = mas_largo(string1.as_str(), string2);
    println!("El mas largo es:{}", result);
}
fn mas_largo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime: ejemplos

```
fn main() {
    let string1 = String::from("Seminario de:");
    let string2 = "Rust!!!";
    let result = mas_largo(string1.as_str(), string2);
    println!("El mas largo es:{}", result);
}
fn mas_largo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime: ejemplos

```
fn mayor<'a>(x: &'a i32, y: &'a i32) -> &'a i32 {
    if x > y { x } else { y }
}
fn main() {
    let a = 10;
    let r;
    {
        let b = 20;
        r = mayor(&a, &b); // ⚠️ aquí 'a = lifetime de b (el menor de los dos)
    }
    println!("Mayor: {}", r); // ❌ Error: b ya no está vivo
}
```

Lifetime: ejemplos

`&a vive todo main`

`&b vive solo dentro del bloque`

`mayor podría devolver &b`

`Entonces 'a = lifetime de b`

`r no puede usarse después del bloque porque podría estar apuntando a b → Rust te impide compilar`

Lifetime: sintaxis

```
&i32        // una referencia
&'a i32     // una referencia con explicito lifetime
&'a mut i32 // una referencia mutable con explicito lifetime
```

Tests

Unit testing

En desarrollo de software es la práctica en la cual se crean pruebas automatizadas para verificar el correcto funcionamiento individual de las unidades de código más pequeñas, como funciones, métodos o clases. Estas pruebas se enfocan en aislar y probar una unidad de código de forma independiente, sin depender de otras partes del sistema.

Unit testing: algunas ventajas

- ★ **Detección temprana de errores**: permiten identificar y corregir errores en una etapa temprana del desarrollo, lo que ayuda a evitar que se propaguen y se conviertan en problemas más difíciles y costosos de solucionar en etapas posteriores.

- ★ **Mejora de la calidad del código:** Al escribir pruebas unitarias, los desarrolladores deben pensar en cómo utilizar y probar sus propias funciones y clases. Esto promueve la escritura de código más limpio, modular y de alta calidad, lo que facilita su mantenimiento y extensión.

Unit testing: ventajas

- ★ **Facilita la refactorización:** Las pruebas unitarias proporcionan un nivel de seguridad al refactorizar el código. Si las pruebas pasan correctamente después de realizar cambios, se tiene la confianza de que las funcionalidades previamente probadas siguen intactas.

- ★ **Documentación viva:** Las pruebas unitarias actúan como una forma de documentación viva del código. Al leer las pruebas, se obtiene una comprensión clara de cómo se espera que funcione cada unidad de código.

Unit testing: importante

Unit testing no asegura que nuestro código no tenga errores sino que es una buena práctica para reducirlos

Unit testing: en rust

```
#[test]
```

```
assert!(expresion);
```

```
assert_eq!(v1, v2);
```

```
assert_ne!(v1, v2);
```

Unit testing: en rust

```
comandos:
    cargo test
    cargo test nombre_del_test
    cargo test nombre_con_el_que_empieza


#[ignore]
#[should_be_panic(expected="mensaje del panic")]
```

Unit testing: en rust

```
//Definir una función llamada contar_letras que reciba un
//&str y un char,
//y retorne cuántas veces aparece el carácter en el string.
```