Lesson 4 Outline

- Recap Lesson 3
- Generics
- Traits
- Lifetime
- Practice & Examples

https://github.com/RustRome/corso-rust

Recap Lesson 3

Structs

```
struct User {
    username : String,
    email : String
}

impl User {

    fn new(username: String,email : String) -> User {
        User {username,email }
    }

    fn hello(&self) -> String {
        format!("Hello {}", self.username)
    }
    fn change_email(&mut self,email : String) {
        self.email = email;
    }

}
```

Structs

```
let mut user = User::new(String::from("wolf4ood"),String::from("enrico.risa@gmail.com"));

println!("{}", user.hello());

user.change_email(String::from("test@gmail.com"));
```

Enum

```
enum Command {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Pattern matching

- match keyword
- Enum (mandatory)
- Structs
- Bool
- Numbers
- Strings
- Tuple
- .....

Pattern matching

```
enum Command {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Command {
    fn exec(&self)  {
        match self {
            Command::Quit => println!("Quit"),
            Command::Move {x,y} => println!("Moving to {}-{}", x,y),
            Command::Write(s) => println!("Writing {}", s),
            Command::ChangeColor(r,g,b) => {
              println!("Changing color to {}-{}-{}",r,g,b),
            }
        }
    }
}

let command = Command::Move { x : 0 , y : 0};
command.exec();
// Moving to 0-0
```

Generics

Generics

1. Result<T,E> - Error Handling
    a. Ok(T)
    b. Err(E)

2. Option<T> - Null Handling
    a. Some(T)
    b. None

Generics

```
enum Option<T> {
    Some(T),
    None,
}


enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Generics - Structs

```
struct Point<T> {
    x: T,
    y: T,
}

let integer = Point { x: 5, y: 10 };
let float = Point { x: 1.0, y: 4.0 };
```

Generics - Methods

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

let p = Point { x: 5, y: 10 };

println!("p.x = {}", p.x());
```

Generics - Single Impl

```
struct Point<T> {
    x: T,
    y: T,
}


impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
let p = Point { x: 5.0, y: 10.0 };

println!("distance = {}", p.distance_from_origin());
```

Generics - Single Impl

```
struct Point<T> {
    x: T,
    y: T,
}


impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
let p = Point { x: 5, y: 10 };

println!("distance = {}", p.distance_from_origin());

//Compiling playground v0.0.1 (/playground)
//error[E0599]: no method named `distance_from_origin` found for type `Point<{integer}>` in
the current scope
//--> src/main.rs:19:33
```

Generics

- Monomorphization (multiple version)
- No runtime cost
- Code reduction
- Additional compilation time

Generic - Performance example

```
enum Option<T> {
    Some(T),
    None,
}

let integer = Some(5);
let float = Some(5.0);
```

Generic - Performance example

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Generic Behaviour

```
struct Pair<T> {
    first : T,
    second : T
}


impl<T> Pair<T> {
    fn bigger(&self) -> &T {
        if self.first > self.second {
            &self.first
        } else {
            &self.second
        }
    }
}
```

Generic Behaviour - Error

```
let p = Pair { first: 5, second: 10 };

println!("bigger = {}", p.bigger());

//Compiling playground v0.0.1 (/playground)
//error[E0369]: binary operation `>` cannot be applied to type `T`
//--> src/main.rs:9:23
//  |
//9 |         if self.first > self.second {
//  |            ---------- ^ ----------- T
//  |            |
//  |            T
//  |
//  = note: `T` might need a bound for `std::cmp::PartialOrd`
```

Traits

Traits

- Defines common behaviour among differents Types
- Enables a sort of OOP programming in rust
- Zero-cost, Very Optimized
- Allows dynamic dispatch (with some restrictions)
- They can be used with Generics

declaration

```
pub trait IsNumber {
  pub fn is_number(&self) → bool;
}
```

implementation

```
// IsNumber is in the same source file
// or is imported from a different module
struct Wrapper(u32);

impl IsNumber for Wrapper {
  fn is_number(&self) -> bool {
    true
  }
}
```

```rust
trait IsNumber {
    fn is_number(&self) -> bool;
}

impl IsNumber for i32 {
    fn is_number(&self) -> bool { true }
}

6.is_number() // true
```

imported and mixed

```
use std::fmt::Debug;

trait PrintDebug
where
    Self: Debug,
{
    fn print_debug(&self);
}
```

default implementation

```
trait PrintDebug
where
    Self: Debug,
{
    fn print_debug(&self) {
        println!("{:?}" self);
    }
}

impl PrintDebug for MyStruct {} // Debug required for MyStruct
```

Bound on Generic structs

```rust
use std::io::Write;

struct App<IO> {
    io: IO,
}

impl<IO: Write> for App<IO> {
    fn save(&self, buf: &[u8]) -> usize {
        self.io.write(buf)
    }
}
```

Specific types can be declared inside

```rust
trait MyMath<T> {
    type Output;
    fn my_math(&self, other: T) -> Self::Output;
}

impl MyMath<i64> for i32 {
    type Output = i128;
    fn my_math(&self, other: i64) -> i128 {
        // or (*self as i64 * other) as i128
        let res = *self as i64 * other;
        res as i128
    }
}
4.my_math(35);
```

Zero cost

```
// Rust resolves Traits in something like this
MyStruct_impl_trait(&self) {
    // ...
}

MyStruct_impl_trait() {
    // ...
}
```

dyn: Traits Objects (dynamic dispatch)

```rust
struct MyStruct<'a> {
    t: &'a dyn Trait,
}

// Allocated
struct MyStruct {
    t: Box<dyn Trait>,
}
```

dyn limitations: Generics

```
// Not Allowed: Size of T must be knowed at compile time
trait MyTrait<T: Debug> {
    fn my_trait(&self, t: T);
}

let mt: &dyn MyTrait = MyStructImplMyTrait{};
```

Simple Workaround could be

```rust
trait MyTrait {
    fn my_trait(&self, t: &dyn Debug);
}
// or
trait MyTrait {
    fn my_trait(&self, t: Box<dyn Debug>);
}
```

Limitations: traits functions cannot return **Self**

```
trait New {
    fn new() → Self;
}
// Error: Self not allowed
// note: Trait cannot require Self : Sized
let o = MyStructImplNew{} as &dyn New;
```

Self and Generics are not object-safe in Trait Objects

Trait are powerful and very efficient

Operator overloading

Very Optimized

Lifetime

Lifetimes

Remember that the owner has always the ability to destroy (deallocate) a resource!

Define the scope for which a reference is valid.

Every reference has a lifetime.

Most of the time inferred by the compiler.
In more complex scenarios compiler **needs an hint**.

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}
```

```
fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}
```

first borrowing

second borrowing
(we return something that is **not ours**)

```
fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}
```

first borrowing

second borrowing
(we return something that is **not ours**)

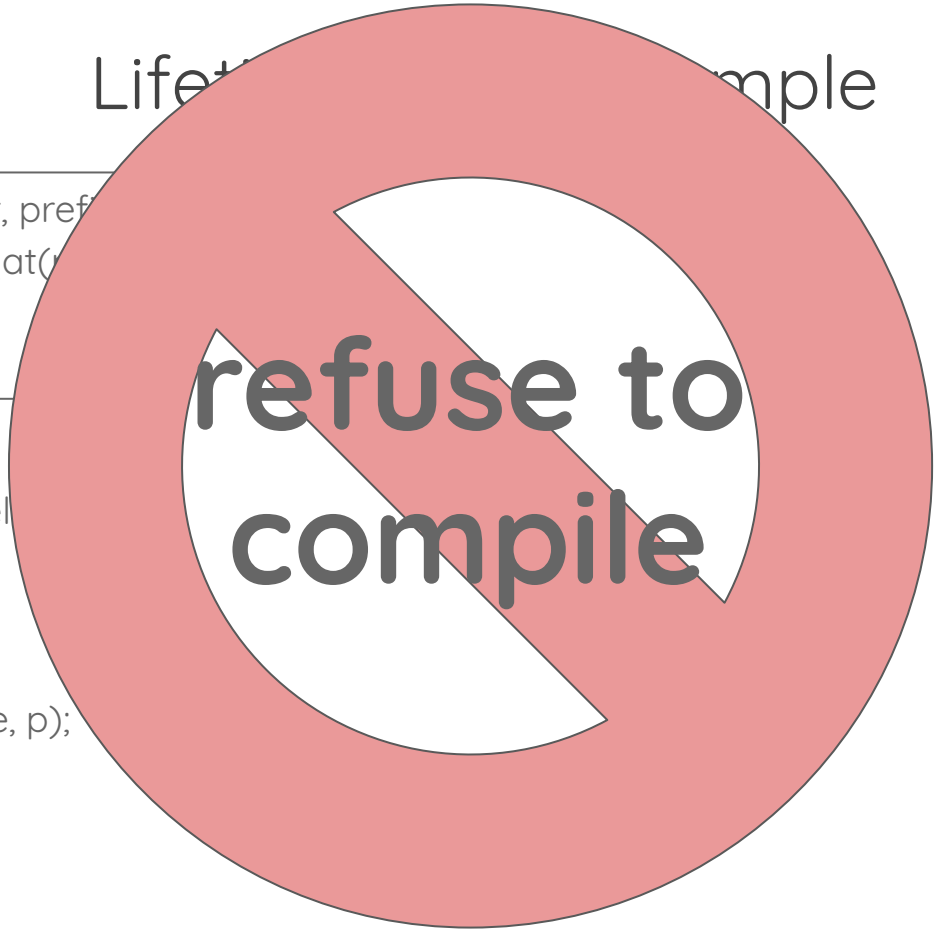
```
fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

we know that “s2” is valid as long as “line” is valid, but **compiler doesn’t know**

Lifet… …mple

```
fn skip_prefix(line: &str, pref
    let (s1,s2) = line.split_at(
    s2
}
```

```
fn print_hello() {
    let line = "lang:en=Hel
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```



Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let prefix = "lang:en=";
        v = skip_prefix(line, prefix);
    }
    println!("{}", v);
}
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:37:45
   |
37 | fn skip_prefix(line: &str, prefix: &str) -> & str {
   |                                             ^ expected lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `line` or `prefix`
```

Lifetime - example reviewed

```
fn skip_prefix<'a>(line: &'a str, prefix: &str) -> &'a str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - example reviewed

```
fn skip_prefix<'a>(line: &'a str, prefix: &str) -> &'a str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

borrowing source is now explicit, through the **lifetime parameter**

Hello World!

Lifetime Elision

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

The static lifetime

```
let s: &'static str = "I have a static lifetime.";
```

Lifetime

- Every reference has a lifetime
- Lifetime parameter (similar to Generics)
- Lifetime elision rules
- Can be applied to
  - Functions
  - Structs
  - Enum
  - Tuple
  - ...
- Checked at compile time

Next lesson

Martedì 3 Dicembre orario 18-20

Per info e domande:

alex179ohm@gmail.com

enrico.risa@gmail.com

Oggetto: **corso rust sapienza**

Slack: http://rust-italia.herokuapp.com channel: #rust-roma