Lesson 3 Outline

- Recap Lesson 2
- Structs
- Enum
- Pattern Matching
- Practice & Examples

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

Recap Lesson 2

Ownership

Each value has a variable called **owner**, which

can be only **one at time.**

The value is **dropped** when the owner goes out of scope.

Ownership

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Ownership (Error)

```
error[E0382]: use of moved value: `s1`
 ⟶ src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
  not implement the `Copy` trait
```

Borrowing

References allow us to refer to values without taking the ownership.

They can be **immutable** (&) or **mutable** (&mut)

Exactly one mutable reference (exclusive)

Borrowing

```
let s1 = String::from("hello");
let s = &s1;

println!("{}, world!", s1);
println!("{}, world!", s);
```

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

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                      // -+ x goes into scope
                                //  |
    {                           //  |
        let y = &5;             // ---+ y goes into scope
        let f = Foo { x: y };   // ---+ f goes into scope
        x = &f.x;               //  | | error here
    }                           // ---+ f and y go out of scope
                                //  |
    println!("{}", x);          //  |
}                               // -+ x goes out of scope
```

String and &str

str is a slice of characters

String own its content, is allocated at runtime

Both implements **UTF-8**

String can be deferenced as **&str**

&str could be easily converted to **String**

Both have len, **String** add capacity to **&str**

Structs

Definition

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

Create new Object

```
let user = User {
        username : String::from("wolf4ood"),
        email : String::from("enrico.risa@gmail.com")
};
```

Shorthand field

```
let username = String::from("wolf4ood");
let email = String::from("enrico.risa@gmail.com");
let user = User {
        username,
        email
};
```

Field access

```
let user = User {
        username : String::from("wolf4ood"),
        email : String::from("enrico.risa@gmail.com")
};

println!("Username: {}", user.username);
```

Methods

```
impl User {

    fn hello(&self) -> String {
        format!("Hello {}",self.username)
    }

}
```

Methods (mut)

```
impl User {

    fn change_email(&mut self, email : String) {
       self.email = email;
    }

}
```

Common Pattern for Object creation

```
impl User {

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

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

Tuple Structs

```
// Declaration
struct Point(i32, i32);

// Allocation
let point = Point(0,0);

// Field access
println!("x: {}, y: {}",point.0,point.1);
```

Enums

Simple Enum

```
enum IpAddrKind {
    V4,
    V6,
}
```

Enum with Values

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

Standard Enum

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

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

Enum with Structured Values

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

Methods

```
impl Command {
    fn exec(&self)  {
     // How do i access the Enum data?
    }
}
```

Pattern Matching

Match on Enums

```
enum IpAddrKind {
    V4,
    V6
}

let ip = IpAddrKind::V4;

match ip  {
    IpAddrKind::V4 => {
        println!("Ipv4")
    },
    IpAddrKind::V6 => {
        println!("Ipv6")
    }
}
```

Match are exhaustive

```
enum IpAddrKind {
    V4,
    V6
}

let ip = IpAddrKind::V4;

match ip  {
    IpAddrKind::V4 => {
        println!("Ipv4")
    }
}
/*
   Compiling playground v0.0.1 (/playground)
error[E0004]: non-exhaustive patterns: `V6` not covered
  --> src/main.rs:17:11
   |
   */
```

_ to catch them all

```
enum IpAddrKind {
    V4,
    V6
}

let ip = IpAddrKind::V4;

match ip  {
    IpAddrKind::V4 => {
        println!("Ipv4")
    },
    _ => {
        println!("All the rest")
    }
}
```

Match several variants

```
enum IpAddrKind {
    V4,
    V6
}

let ip = IpAddrKind::V4;

match ip  {
    IpAddrKind::V4 | IpAddrKind::V6 => {
        println!("All ip type")
    }
}
```

Match on Enum with values

```
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
```

Match all the things

Match on bool

```
let active = false;

match active  {
    true => println!("Active"),
    false => println!("Not Active")
};
```

Match on strings

```
let username = "wolf4ood";

match username  {
    "wolf4ood" => println!("Hi Enrico"),
    _ => println!("Hello Stranger")
};
```

Match on numbers

```
let number = 13;

match number {
    1 ⇒ println!("One"),
    2 | 3 | 5 | 7  ⇒ println!("prime less than then"),
    13..=19 ⇒ println!("≥ 13 and ≤ 19"),
    // 13 ... 19 same
    _ ⇒ println!("nothing special"),
}
```

Match guards

```
 let pair = (0, -2);
    // TODO ^ Try different values for `pair`

println!("Tell me about {:?}", pair);
// Match can be used to destructure a tuple
match pair {
    // Destructure the second
    (0, y) => println!("First is `0` and `y` is `{:?}`", y),
    (x, 0) => println!("`x` is `{:?}` and last is `0`", x),
    _      => println!("It doesn't matter what they are"),
    // `_` means don't bind the value to a variable
}
```

One pattern match (if let)

```
let result : Result<String,String> = Err(String::from("My
error"));

if let Err(e) = result {
    println!("Error {}", e);
}
```

Destructured match

Struct

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

let p = Point{ x: 1, y: 2 };

match p {
    Point{ x: 1, y: 3 } ⇒ println!("just x == 1 and y == 3"),
    Point{ x: 2, .. } ⇒ println!("just x == 2"),
    Point{ y: 2, .. } ⇒ println!("just y == 2 order dosn't matter"),
    Point{ .. } ⇒ println!("every point"),
}
```

Struct tuple

```
struct Point(i32, i32);

let p = Point(1, 2);

match p {
    Point{ 0: 1, 1: 3 } ⇒ println!("just x == 1 and y == 3"),
    Point{ 0: 2, .. } ⇒ println!("just x == 2"),
    Point{ 1: 2, .. } ⇒ println!("just y == 2 order dosn't matter"),
    Point{ .. } ⇒ println!("every point"),
}
```

Grouped match

```
let ref_n = &3;

match ref_n {
    // error: ambiguous not allowed
    //&0..=5 ⇒ println!("≥ 0 and ≤ 5"),
    &(0..=5) ⇒ println!("≥ 0 and ≤ 5"),
    _ ⇒ println!("< 0 or > 5"),
}
```

Match array

```
let array = [1, 2, 3];
match array {
  [1, _, _] ⇒ println!("starts with one"),
  [a, b, c] ⇒ {
       println!("starts with {}", a);
       println!("and then {}", b);
       println!("and then {}", c);
  },
}
```

Binding match

```
let number = 3;

match number {
    n @ 1 => println!("is one {}",n),
    n @ 2..=4 => println!("is not five: {}",n),
    n => println!("is five or more: {}",n)
}
```

Binding inside Option

```
let number = Some(4);
match number {
  Some(n @ 0..=5) ⇒ println!("{} is ⩾ 0 and ⩽ 5", n),
  Some(n) ⇒ println!("{}", n),
  None ⇒ println!("None"),
}
```

Match references

```
let number = Some(4);
match number {
  Some(ref n) ⇒ println!("I've got the reference of {}", n),
  None ⇒ println!("None"),
}
```

Match mut references

```
let mut number = Some(4);
match number {
    Some(ref mut n) => {
        *n = 4;
        println!("I've got the mutable reference of {}", n);
    },
    None => println!("None"),
}
```

Next lesson

Giovedì 28 novembre 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