Lecture 2: Match and Option API

Barinov Denis

February 26, 2025!

barinov.diu@gmail.com

Match

`match` is one of things that will help you to work with `enum`.

```
let x = MyEnum::First;
match x {
    MyEnum::First => println!("First"),
    MyEnum::Second => {
        for i in 0..5 { println!("{i}"); }
        println!("Second");
    },
    _ => println!("Matched something!"),
}
```

The _ symbol

- _ matches everything in `match` (called wildcard).
- Used for inference sometimes:

```
// Rust does not know here to what type
// you want to collect
let mut vec: Vec<_> = (0..10).collect();
vec.push(42u64);
```

- And to make a variable unused:

```
let _x = 10;
// No usage of _x, no warnings!
```

Match

`match` can match multiple objects at a time:

```
let x = OneMoreEnum::<i32>::Ein(2);
let y = MyEnum::First;
match (x, y) {
    (OneMoreEnum::Ein(a), MyEnum::First) => {
        println!("Ein! - {a}");
    },
    // Destructuring
    (OneMoreEnum::Zwei(a, _), _) => println!("Zwei! - {a}"),
    _ => println!("oooof!"),
}
```

Match

There's feature to match different values with same code:

```
let number = 13;
match number {
    1 => println!("One!"),
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    13..=19 => println!("A teen"),
    _ => println!("Ain't special"),
}
```

Match

And we can apply some additional conditions called guards:

```
let pair = (2, -2);
println!("Tell me about {:?}", pair);
match pair {
    (x, y) if x == y => println!("These are twins"),
    // The ^ `if condition` part is a guard
    (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
    (x, _) if x % 2 == 1 => println!("The first one is odd"),
    _ => println!("No correlation..."),
}
```

Match

Match is an expression too:

```
let x = 13;
let res = match x {
    13 if foo() => 0,
    // You have to cover all of the possible cases
    13 => 1,
    _ => 2,
};
```

Match

Ignoring the rest of the tuple:

```
let triple = (0, -2, 3);
println!("Tell me about {:?}", triple);
match triple {
    (0, y, z) => {
        println!("First is `0`, `y` is {y}, and `z` is {z}")
    },
    // `..` can be used to ignore the rest of the tuple
    (1, ..) => {
        println!("First is `1` and the rest doesn't matter")
    },
    _ => {
        println!("It doesn't matter what they are")
    },
}
```

Match

Let's define a struct:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

let foo = Foo { x: (1, 2), y: 3 };
```

Match

Destructuring the struct:

```
match foo {
    Foo { x: (1, b), y } => {
        println!("First of x is 1, b = {},  y = {} ", b, y);
    },
    Foo { y: 2, x: i } => {
        println!("y is 2, i = {:?}", i);
    },
    Foo { y, .. } => { // ignoring some variables:
        println!("y = {}, we don't care about x", y)
    },
    // Foo { y } => println!("y = {}", y),
    // error: pattern does not mention field `x`
}
```

Match

Binding values to names:

```
match age() {
    0 => println!("I haven't celebrated my birthday yet"),
    n @ 1..=12 => println!("I'm a child of age {n}"),
    n @ 13..=19 => println!("I'm a teen of age {n}"),
    n => println!("I'm an old person of age {n}"),
}
```

Match

Binding values to names + arrays:

```
let s = [1, 2, 3, 4];
let mut t = &s[..]; // or s.as_slice()
loop {
    match t {
        [head, tail @ ..] => {
            println!("{head}");
            t = &tail;
        }
        _ => break,
    }
} // outputs 1\n2\n\3\n4\n
```

if let

Sometimes we need only one enumeration variant to do something. Can we write it in a better way?

```
let optional = Some(7);
match optional {
    Some(i) => {
        println!("It's Some({i})");
    },
    _ => {},
    // ^ Required because `match` is exhaustive
};
```

if let

Sometimes we need only one enumeration variant to do something. Can we write it in a better way?

```
let optional = Some(7);
if let Some(i) = optional {
    println!("It's Some({i})");
}
```

while let

Same with `while`:

```
let mut optional = Some(0);
while let Some(i) = optional {
    if i > 9 {
        println!("Greater than 9, quit!");
        optional = None;
    } else {
        println!("`i` is `{i}`. Try again.");
        optional = Some(i + 1);
    }
}
```

Vector

```
let mut xs = vec![1, 2, 3];
// To declare vector with same element and
// specific count of elements, write
// vec![42; 113];
xs.push(4);
assert_eq!(xs.len(), 4);
assert_eq!(xs[2], 3);
```

Slices

We can create a slice to a vector or array. A slice is a contiguous sequence of elements in a collection.

```
let a = [1, 2, 3, 4, 5];
let slice1 = &a[1..4];
let slice2 = &slice1[..2];
assert_eq!(slice1, &[2, 3, 4]);
assert_eq!(slice2, &[2, 3]);
```

Panic!

In Rust, when we encounter an unrecoverable error, we `panic!`

```
let x = 42;
if x == 42 {
    panic!("The answer!")
}
```

There are some useful macros that `panic!`

- `unimplemented!`
- `unreachable!`
- `todo!`
- `assert!`
- `assert_eq!`

println!

The best tool for debugging, we all know.

```
let x = 42;
println!("{x}");
println!("The value of x is {}, and it's cool!", x);
println!("{:04}", x); // 0042
println!("{value}", value=x + 1); // 43
let vec = vec![1, 2, 3];
println!("{vec:?}");   // [1, 2, 3]
println!("{:?}", vec); // [1, 2, 3]
let y = (100, 200);
println!("{:#?}", y);
// (
//     100,
//     200,
// )
```

`Option`[1] and `Result`[2]

Let's remember their definitions:

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

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

[1] `Option` documentation
[2] `Result` documentation

Option API

Matching Option:

```
let result = Some("string");
match result {
    Some(s) => println!("String inside: {s}"),
    None => println!("Ooops, no value"),
}
```

Option API

Useful functions .unwrap() and .expect():

```
fn unwrap(self) -> T;
fn expect(self, msg: &str) -> T;
```

Option API

Useful functions .unwrap() and .expect():

```
let opt = Some(22022022);
assert!(opt.is_some());
assert!(!opt.is_none());
assert_eq!(opt.unwrap(), 22022022);
let x = opt.unwrap(); // Copy!

let newest_opt: Option<i32> = None;
// newest_opt.expect("I'll panic!");

let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!
```

`Option` API

We have a magic function:

```
fn as_ref(&self) -> Option<&T>; // &self is &Option<T>
```

Let's solve a problem:

```
let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!

let opt_ref = Some(Vec::<i32>::new());
assert_eq!(new_opt.as_ref().unwrap(), &Vec::<i32>::new());
let x = new_opt.unwrap(); // We used reference!
// There's also .as_mut() function
```

That means if type implements `Copy`, `Option` also implements `Copy`.

`Option` API

We can map `Option<T>` to `Option<U>`:

```
fn map<U, F>(self, f: F) -> Option<U>;
```

Example:

```
let maybe_some_string = Some(String::from("Hello, World!"));
// `Option::map` takes self *by value*,
// consuming `maybe_some_string`
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```

`Option` API

There's **A LOT** of different `Option` functions, enabling us to write beautiful functional code:

```
fn map_or<U, F>(self, default: U, f: F) -> U;
fn map_or_else<U, D, F>(self, default: D, f: F) -> U;
fn unwrap_or(self, default: T) -> T;
fn unwrap_or_else<F>(self, f: F) -> T;
fn and<U>(self, optb: Option<U>) -> Option<U>;
fn and_then<U, F>(self, f: F) -> Option<U>;
fn or(self, optb: Option<T>) -> Option<T>;
fn or_else<F>(self, f: F) -> Option<T>;
fn xor(self, optb: Option<T>) -> Option<T>;
fn zip<U>(self, other: Option<U>) -> Option<(T, U)>;
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

`Option` and ownership

There's two cool methods to control ownership of the value inside:

```
fn take(&mut self) -> Option<T>;
fn replace(&mut self, value: T) -> Option<T>;
fn insert(&mut self, value: T) -> &mut T;
```

The first one takes the value out of the `Option`, leaving a `None` in its place.

The second one replaces the value inside with the given one, returning `Option` of the old value.

The third one inserts a value into the `Option`, then returns a mutable reference to it.

Option API and ownership: take

```
struct Node<T> {
    elem: T,
    next: Option<Box<Node<T>>>,
}

pub struct List<T> {
    head: Option<Box<Node<T>>>,
}

impl<T> List<T> {
    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }
}
```

`Option` and optimizations

Rust guarantees to optimize the following types `T` such that `Option<T>` has the same size as `T`:

- `Box<T>`
- `&T`
- `&mut T`
- `fn`, `extern "C" fn`
- `#[repr(transparent)]` struct around one of the types in this list.
- `num::NonZero*`
- `ptr::NonNull<T>`

This is called the "null pointer optimization" or NPO.

`Result`

Functions return Result whenever errors are expected and recoverable. In the `std` crate, `Result` is most prominently used for I/O.

Results must be used! A common problem with using return values to indicate errors is that it is easy to ignore the return value, thus failing to handle the error. `Result` is annotated with the `#[must_use]` attribute, which will cause the compiler to issue a warning when a Result value is ignored.[3]

[3] The Error Model

Result API

We can match it as a regular enum:

```
let version = Ok("1.1.14");
match version {
    Ok(v) => println!("working with version: {:?}", v),
    Err(e) => println!("error: version empty"),
}
```

`Result` API

We have pretty the same functionality as in `Option`:

```
fn is_ok(&self) -> bool;
fn is_err(&self) -> bool;
fn unwrap(self) -> T;
fn unwrap_err(self) -> E;
fn expect_err(self, msg: &str) -> E;
fn expect(self, msg: &str) -> T;
fn as_ref(&self) -> Result<&T, &E>;
fn as_mut(&mut self) -> Result<&mut T, &mut E>;
fn map<U, F>(self, op: F) -> Result<U, E>;
fn map_err<F, O>(self, op: O) -> Result<T, F>;
// And so on
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

Operator ?

Consider the following structure:

```
struct Info {
    name: String,
    age: i32,
}
```

Operator ?

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = match File::create("my_best_friends.txt") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };
    if let Err(e) = file
        .write_all(format!("name: {}\n", info.name)
        .as_bytes()) {
        return Err(e)
    }
    if let Err(e) = file
        .write_all(format!("age: {}\n", info.age)
        .as_bytes()) {
        return Err(e)
    }
    Ok(())
}
```

Operator ?

We can use the ? operator to make the code smaller!

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt")?;
    file.write_all(format!("name: {}\n", info.name).as_bytes())?;
    file.write_all(format!("age: {}\n", info.age).as_bytes())?;
    Ok(())
}
```

Beautiful, isn't it?

We can use it for `Option` too!

Homework

Questions?