Wer rastet, der rostet

Ein Rust-Kurs für Anfänger

Hendrik Wolff

Anton Obersteiner

Sommersemester 2023

Enums: Overview

- define a type by *enumerating* its possible *variants*
- can represent only *one* variant at a time
- hold associated data like a struct or a tuple
- access variants with the *namespace* operator `::`

Enums: Overview

- define a type by *enumerating* its possible *variants*
- can represent only *one* variant at a time
- hold associated data like a struct or a tuple
- access variants with the *namespace* operator `::`

```
#[derive(Debug)]
enum MouseButton {
    Left,
    Middle,
    Right,
}

println!("{:?}", MouseButton::Left);
// prints `Left`
println!("{:?}", MouseButton::Right as u8);
// prints `2`
```

Enums: Overview

- define a type by *enumerating* its possible *variants*
- can represent only *one* variant at a time
- hold associated data like a struct or a tuple
- access variants with the *namespace* operator `::`

```rust
#[derive(Debug)]
enum MouseButton {
    Left,
    Middle,
    Right,
}

println!("{:?}", MouseButton::Left);
// prints `Left`
println!("{:?}", MouseButton::Right as u8);
// prints `2`
```

```rust
#[derive(Debug)]
enum WindowEvent {
    Mouse {
        btn: MouseButton,
        x: u32,
        y: u32,
    },
    KeyPress(u16), // contains keycode
    FocusLost,
    FocusGained,
}
```

Non-existent values: the Option type

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

- generic over its encapsulated type
- globally usable, no need to prefix `Option::`
- None ⇒ there is no value
- Some ⇒ there is a value

```
let m_btn = Some(MouseButton::Left);
```

The match expression

```
let my_bool = true;
match my_bool {
    true => {},
    false => {},
}

let btn = MouseButton::Middle;
match btn {
    MouseButton::Left => { println!("left") },
    MouseButton::Middle => { println!("middle") },
    MouseButton::Right => { println!("right") },
}
```

The match expression

```rust
let num = Some(12);
let num = match num {
    Some(x) => x * 2,
    None => 9,
};

// binding with a catch all pattern
match num {
    3 => println!("three"),
    5 => prinltn!("five"),
    other => println!("got {other}"),
}
```

```rust
// the _ placeholder
match num {
    3 => println!("still three"),
    _ => {} // ignores other possibilities
}
```

Exercise: Temperature conversion (mit mat(s)ch spielen)

Write 3 methods for the `Temperature` type that convert the inner value to the desired unit. The functions should be called `to_celsius`, `to_fahrenheit`, `to_kelvin`. They should return the corresponding integer value; `i32`, `i32`, `u32` respectively. Use the `match` statement. You can ignore decimals and rounding errors.

```
enum Temperature {
    Celsius(i32),
    Fahrenheit(i32),
    Kelvin(u32),
}
```

At the end you should be able to do this:

```
let temp1 = Temperature::Celsius(-20);
println!("temp1: {}°C", temp1.to_celsius()); // -20
println!("temp1: {}°F", temp1.to_fahrenheit()); // -4
println!("temp1: {}K", temp1.to_kelvin()); // 253
```

Matching literals

```
let n = 1;

match n {
    1 => println!("one"),
    2 => println!("two"),
    _ => println!("anything"),
}
```

Matching named variables

```
let n = 1;

match n {
    1 => println!("one"),
    2 => println!("two"),
    n => println!("n = {n}"),
}
```

Matching multiple patterns

- using the *or* operator | or using a range expression

```
let n = 1;

match n {
    1 | 2 => println!("one or two"),
    3..=6 => println!("[3,6]"),
    7..10 | 13 => println!("[7,10) or 13"),
    n => println!("n = {n}"),
}
```

Destructuring: structs

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

let p = Point { x: 0, y: 7 };
match p {
    Point { x, y: 0, z: 0 } => println!("On the x axis at {x}"),
    Point { x: 0, y, z: 0 } => println!("On the y axis at {y}"),
    Point { x: 0, y: 0, z } => println!("On the z axis at {z}"),
    Point { x: a, y: b, z: c } => { // `x`, `y`, `z` are renamed to `a`, `b`, `c`
        println!("On neither axis: ({a}, {b}, {c})");
    }
}
```

Destructuring: enums

```
enum WindowEvent {
    Mouse { btn: MouseButton, x: u32, y: u32 },
    KeyPress(u16),
    FocusLost,
    FocusGained,
}
let event = WindowEvent::Mouse { btn: MouseButton::Left, x: 20, y: 100 };
match event {
    WindowEvent::Mouse { btn: MouseButton::Left | MouseButton::Right, x, y } => { ... },
    WindowEvent::Mouse { btn: MouseButton::Middle, x, y } => { ... },
    WindowEvent::KeyPress(keycode) => { ... },
    WindowEvent::FocusLost => { ... },
    WindowEvent::FocusGained => { ... },
}
```

Ignoring values

- with the _ catch-all

```
fn foo(_: u16, b: u16) {
    println!("we only need b: {b}");
}

fn main() {
    foo(5, 6);

    let numbers = (1, 2, 3, 4, 5);
    match numbers{
        (_, b, _, d, _) => println!("(_, {b}, _, {d}, _)"),
    }
}
```

Ignoring values

- with the .. range

```
let numbers = (1, 2, 3, 4, 5);
match numbers {
    (first, .., last) => {
        println!("Some numbers: {first}, {last}");
    }
}

let origin = Point { x: 0, y: 0, z: 0 };
let Point { x, .. } = origin;
println!("x is {x}");
```

Match guards

- condition, attached to a pattern
- applies to the entire pattern

```
let num = Some(4);
match num {
    Some(x) if x % 2 == 0 => println!("The number {} is even", x),
    Some(x) => println!("The number {} is odd", x),
    None => {},
}

let (x, cond) = (4, false);
match x {
// (         ) if cond =>
    4 | 5 | 6  if cond => println!("yes"),
    _ => println!("no"),
}
```

@ Bindings

```
enum Question {
    Input { n: i32 },
}

let q = Question::Input { n: 24 };
let answer = match q {
    Question::Input { n: num @ 24 } => num + 18,
    Question::Input { n: num @ 50..=100 | num @ 13 } => {
        eprintln!("We don't want number {num}!");
        12 * 4 - 6
    },
    _ => 6 * 7,
}
println!("Answer to the Ultimate Question of Life, the Universe, and Everything is {answer}");
```

The match expression

```rust
let num = Some(12);
let num = match num {
    Some(x) => x * 2,
    None => 9,
};

// binding with a catch all pattern
match num {
    3 => println!("three"),
    5 => prinltn!("five"),
    other => println!("got {other}"),
}
```

```rust
// the _ placeholder
match num {
    3 => println!("still three"),
    _ => {} // ignores other possibilities
}
```

Exercise: Temperature conversion (mit mat(s)ch spielen)

Write 3 methods for the `Temperature` type that convert the inner value to the desired unit. The functions should be called `to_celsius`, `to_fahrenheit`, `to_kelvin`. They should return the corresponding integer value; `i32`, `i32`, `u32` respectively. Use the `match` statement. You can ignore decimals and rounding errors.

```
enum Temperature {
    Celsius(i32),
    Fahrenheit(i32),
    Kelvin(u32),
}
```

At the end you should be able to do this:

```
let temp1 = Temperature::Celsius(-20);
println!("temp1: {}°C", temp1.to_celsius()); // -20
println!("temp1: {}°F", temp1.to_fahrenheit()); // -4
println!("temp1: {}K", temp1.to_kelvin()); // 253
```

Non-exhaustive pattern matching with `if let`

```
fn eval(maybe_inner: Option<i32>) -> bool {
    match maybe_inner {
        Some(inner) => {
            let result = do_some_crazy_computation(inner);
            println!("We have an answer: {:?}", result);
            result
        },
        None => false
    }
}
```

Non-exhaustive pattern matching with `if let`

```
fn eval(maybe_inner: Option<i32>) -> bool {
    if let Some(inner) = maybe_inner {

            let result = do_some_crazy_computation(inner);
            println!("We have an answer: {:?}", result);
            return result;
    }
    false
}
```

Task: 'click'

- implement a `WindowEvent` enum like in the slides

```
fn main() {
    let lost = WindowEvent::FocusLost;



}
```

Task: 'click'

- implement a `WindowEvent` enum like in the slides
- create some instances of it

```
fn main() {
    let lost = WindowEvent::FocusLost;
    let click = WindowEvent::Mouse { btn: MouseButton:: Left, x: 20, y: 100 };



}
```

Task: 'click'

- implement a `WindowEvent` enum like in the slides
- create some instances of it
- use `if let` to print the coordinates of clicks

```
fn main() {
    let lost = WindowEvent::FocusLost;
    let click = WindowEvent::Mouse { btn: MouseButton:: Left, x: 20, y: 100 };
    print_coordinates_if_click(lost); // lost prints nothing, click does


}
```

Task: 'Patterned Bars'

- presumes you have done task ‘bars’

```
##### 5
### 3
######## 8
###### 6
```

```
pub struct Bars {
  width: i32,
  values: [i32; HEIGHT],


}
```

Task: 'Patterned Bars'

- presumes you have done task ‘bars’
- the normal pattern can be set to something other than #

```
##### 5
### 3
######## 8
###### 6
```

```
pub struct Bars {
  width: i32,
  values: [i32; HEIGHT],
  // set to '#' in Bars::new()
  normal_pattern: char
}
```

Task: 'Patterned Bars'

- presumes you have done task ‘bars’
- the normal pattern can be set to something other than #
- add an `enum Pattern` for patterned bars

```
##### 5     #Pattern::Normal
::: 3       #Pattern::Single(':')
//////// 8  #Pattern::Single('/')
###### 6
```

```rust
pub enum Pattern {
  Normal,         // needs no extra infos
  Single(char),   // stores a char


}
```

Task: 'Patterned Bars'

- presumes you have done task ‘bars’
- the normal pattern can be set to something other than `#`
- add an `enum Pattern` for patterned bars
- include `Normal`, `Single(char)`

```
##### 5
::: 3
/////// 8
###### 6
```

```rust
pub enum Pattern {
  Normal,        // needs no extra infos
  Single(char),  // stores a char


}
```

Task: 'Patterned Bars'

- presumes you have done task ‘bars’
- the normal pattern can be set to something other than `#`
- add an `enum Pattern` for patterned bars
- include `Normal`, `Single(char)`
- use them in Bars: in the `struct`, in `print`, …
- e.g. to mark the max and min values

```
##### 5
::: 3
/////// 8
###### 6
```

```rust
pub enum Pattern {
  Normal,         // needs no extra infos
  Single(char),   // stores a char


}
```

Task: 'Patterned Bars'

- presumes you have done task ‘bars’
- the normal pattern can be set to something other than `#`
- add an `enum Pattern` for patterned bars
- include `Normal`, `Single(char)`
- use them in Bars: in the `struct`, in `print`, ...
- e.g. to mark the max and min values
- maybe add a `Multiple(&'static str)` to `Pattern`

```
##### 5     #Pattern::Normal
::: 3       #Pattern::Single(':')
/¯\_/¯\_ 8  #Pattern::Multiple("/¯\_")
###### 6
```

```rust
pub enum Pattern {
  Normal,         // needs no extra infos
  Single(char),   // stores a char
  // stores a string literal
  Multiple(&'static str)
}
```