Rust

Enums, Pattern, Matches und Advanced Types

Struktur

- Enums
  - Definieren
  - Option<T>
- Patterns und Matches
  - Match-Operator
  - if let, while let
  - Vorkommen von Matches
  - Refutable und Irrefutable
- Advanced Types

Enums

Deklaration und Nutzen

- eigener Typ mit festgelegten Varianten

- ähnlich zu anderen bekannten Sprachen

Deklaration und Nutzen

```
enum Temperature {
    Warm,
    Cold,
}
```

Deklaration und Nutzen

```
let warm_water = Temperature::Warm;

fn water(water_temperature: Temperature) {}
```

Zusätzliche Werte

```
enum Temperature {
    Warm(f32),
    Cold(f32),
}
let warm_water = Temperature::Warm(42.3);
```

→ unterschiedliche Typen möglich

Unterschiedliche Typen

```
enum Location {
    None,
    Point { x: i32, y: i32 },
    Name(String),
    Area(String, i32),
}
```

Funktionen implementieren

```
impl Temperature {
        fn raise(&self) {
                // do stuff...
        }
}

let t = Temperature::Cold(5.0);
t.raise();
```

Option‹T›

- Rust kennt den Wert “*null*” nicht
- keine null-errors
- Möglichkeit nicht vorhandenen Wert auszudrücken:

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

Option‹T›

- Standardmäßig direkt eingebunden
- Aufruf über *None* und *Some(T)*
- Option:: entfällt bei Aufruf

```
let some_number = Some(5);

let some_string = Some("a string");


let absent_number: Option<i32> = None;
```

Option‹T›

- Option<T> ist eigener Typ
- deshalb nicht direkt als T verwendbar
- Konvertierung nötig → unwrap()

```
let some_number = Some(5);

let some_other_number = 3;

let result = some_number.unwrap() + some_other_number;
```

Aufgabe 1

Steuerung eines autonomen Fahrzeugs, Programmierung der Befehle (Action). Der Befehl soll verschiedene Ausprägungen haben können: Move (Koordinate -> x, y, benannt), GoTo (Ort, Name), Stop, Wait (Dauer). Zusätzlich soll an jedem Befehl die Funktion abort aufgerufen werden können, der ein Abbruch-Code übergeben wird. Dieser Abbruch-Code ist vom Typ i8, kann aber auch mal keinen Wert haben. In dieser Funktion wird der übergebene Code auf den Basis-Code base_code = 100 vom Typ i8 addiert. Zum Abschluss soll jeder der Befehle einmal ausgeführt werden (Koordinate 49, 8; Ziel Home, Warten 5) und der Befehl Wait abgebrochen werden (Code 505). Zur Umsetzung soll das Wissen über Enums genutzt werden.

Pattern und Matches

Match

- Kontrollflussoperator
- Testet eine Variable gegen unterschiedliche Vorgaben und gibt entsprechenden Wert zurück
- alle Fälle müssen abgedeckt sein
- die Vorgaben werden auch “Match-Arme” genannt
- Ähnlich switch-case in C# und Java

Match

```
enum Currency {
	EUR,
	GBP,
	USD,
	CAD,
	NZD,
}

fn value_compared_to_euro(currency: Currency) -> f32 {
	match currency {
		Currency:: EUR => 1.0,
		Currency:: GBP => 0.839,
		Currency:: USD => 1.42,
		Currency:: CAD => 1.13,
		Currency:: NZD => 1.61,
	}
}
```

Match

Ein Match-Arm kann auch ganze Blöcke enthalten:

```
fn value_compared_to_euro(currency: Currency) -> f32 {
        match currency {
                Currency:: EUR => {
                        println!(“Das macht nicht so viel Sinn!”);
                        1.0
                }
                Currency:: GBP => 0.839,
                …
        }
}
```

Match

Zusätzliche Werte können im Match-Arm verwendet werden

```
enum Country {
    USA,
    Canada,
    NewZealand,
}

enum Currency {
    Euro,
    Pfund,
    Dollar(Country),
}
```

```
fn value_compared_to_euro(currency: Currency) -> f32 {
    match currency {
        Currency:: Euro=> 1.0,
        Currency:: Pfund=> 0.839,
        Currency:: Dollar(country) => {
            // irgendwas mit country machen und einen
                entsprechenden Wert zurückgeben
        }
    }
}
```

Match mit Option‹T›

```
fn times_two(x: Option<i32>) -> Option<i32> {
        match x {
                None => None,
                Some(i) => Some(i * 2),
        }
}

let four = Some(4);
let eight = times_two(four);
let none = times_two(None);
```

Match mit Platzhalter

Platzhalter *other* ermöglicht das Sammeln aller nicht genannten Optionen in einer verwendbaren Variable

```
fn value_compared_to_euro(currency: Currency) -> f32 {
    match currency {
        Currency:: EUR => 1.0,
        Currency:: GBP => 0.839,
        other => handle_dollars(other),
    }
}
```

Match mit Platzhalter

Platzhalter _ ermöglicht das Sammeln aller nicht genannten Optionen in einem Arm, “vergisst” jedoch die Variable

```
fn value_compared_to_euro(currency: Currency) -> f32 {
    match currency {
        Currency:: EUR => 1.0,
        Currency:: GBP => 0.839,
        _ => 0.0, // Dollar sind hier nichts wert!
    }
}
```

Aufgabe 2

Erweitert Action aus Aufgabe 1 um Fly, dem ebenfalls x- und y-Koordinate übergeben werden soll und Abort. Schreibt eine Funktion, der eine Action übergeben wird. Bei Move soll die Distanz der Zielposition zur aktuellen Position (13, 42) berechnet und ausgegeben werden, bei GoTo das Ziel, bei Wait “Waiting for x seconds...” bei Stop nichts, Abort nichts und bei Fly ebenfalls die Distanz zur Zielposition. Es dürfen nur 5 Match-Arme verwendet werden.

if let

- kurze Schreibweise für Match
- nur ein Arm wird benötigt
- alle anderen Fälle werden gesammelt oder verworfen
- else anstatt von _ => ...

if let

```
let password = “123456”

match check_password {

    “password” => println!(“Correct!”),

    _ => (),

}
```

if let

```
let password = “123456”;

if let “password” = password {

        println!(“Correct!”);

}
```

if let

```
let password = “123456”;

if let “password” = password {

        println!(“Correct!”);

} else {

        println!(“{} is not the correct password!”, password);

}
```

Pattern

- Syntax zum Matchen von einfachen und komplexen Strukturen
- nutzt Literale
- destrukturierte arrays, enums, structs und tupel
- Variablen
- Wildcards
- Platzhalter

Bereits auf den vorherigen Folien gesehen!

Pattern Vorkommen

- Match-Arme
- if let
- while let
- for-Schleifen
- let
- Funktionsparameter

Pattern Syntax - Literale

```
let x = 1;

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

Pattern Syntax - Variablen

```
let x = 1;
let t1 = 1;
let t2 = 2;

match x {
        t1 => println!("one: {}", t1),
        t2 => println!("two: {}", t2),
        _ => println!("anything"),
}
```

Pattern Syntax - Mehrere Pattern

```
let x = 1;


match x {
        1 | 2 => println!("one or two"),
        5 | 6 => println!("five or six"),
        _ => println!("other number"),
}
```

Pattern Syntax - Ranges

```
let x = 1;


match x {
        1..=10 => println!("one to ten"),
        11..=20 => println!("eleven to twenty"),
        _ => println!("bigger than twenty"),
}
```

Pattern Syntax - Destrukturierung (Struct)

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

fn main() {
	let p = Point { x: 0, y: 7 };
	let Point { x, y } = p;
	println!(“x: {}”, x); // => x: 0
	println!(“y: {}”, y); // => y: 7
}
```

Pattern Syntax - Destrukturierung (Enum)

```
enum Location {
	None,
	Point { x: i32, y: i32 },
	Name(String),
	Area(String, i32),
}

fn main() {
	let loc = Location::None;
	match loc {
		Location::None => println!("None"),
		Location::Point { x: i32, y: i32 } => println!("x: {}, y : {}", x, y),
		Location::Name(name) => println!("Location name: {}", name),
		Location::Area(name, distance) => println!("Area name: {}, Distance: {}", name, distance),
	}
}
```

Pattern Syntax - Destrukturierung (Schachtelung)

Pattern können beliebig gemischt und geschachtelt werden. Hier sieht man es anhand von Structs und Tupeln:

```
let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

Pattern Syntax - Werte ignorieren

Ignorieren mit _:

```
fn my_func(_: i32, y: i32) {
	println!("Nur der zweite Parameter wird genutzt: {}", y);
}


let numbers = (2, 4, 8, 16, 32);


match numbers {
	(first, _, third, _, fifth) => {
		println!("Some numbers: {}, {}, {}", first, third, fifth)
	}
}
```

Pattern Syntax - Werte ignorieren

Durch Voranstellen von _ bei einer Variable wird die unused-Warnung unterdrückt.

```
fn main() {
    let x = 10; // → Warnung wegen unused x
    let _y = 5; // → Keine Warnung,
}
```

Die Zuweisung einer existierenden Variable zu _varname übergibt trotzdem die Ownership.

Pattern Syntax - Werte ignorieren

```
let my_tup = (1, 2, 3, 4, 5)


match my_tup {
        (first, .., last) => {
                // nur 1 und 5 werden zugewiesen, alle Werte dazwischen werden ignoriert
        }
}
```

.. darf nur einmal verwendet werden, (.., second, ..) ist nicht erlaubt!

Pattern Syntax - Match Guards

- Zusätzliche Bedingungen für Match-Arme
- Zum Umsetzen für komplexere Konzepte

```
let x = Some(2);

match x {
    Some(y) if y > 1 => println!(“Bigger than one”);
    Some(y) => println!(“Smaller than one”)
    None => (),
}
```

Pattern Syntax - Match Guards

- Abfrage des Guards is **kein** Pattern
- erstellt keine neuen Variablen
- in Guard kann deshalb auf äußere Variablen zugegriffen werden

```
let x = Some(2);
let y = 3;

match x {
        Some(n) if n > y => println!(“Bigger than {}”, y);
        Some(n) => println!(“Smaller than  {}”, y)
        None => (),
}
```

Pattern Syntax - @-Binding

Erschafft eine lokale Variable, die gleichzeitig getestet wird

```
enum Temperature {
        Warm { amount: f32 },
}

let temp = Temperature::Warm { amount: 32.0 };

match temp {
        Temperature ::Warm {
                amount: amount_var @ 0..=99,
        } => println!(“amount_var ({}) can be used here!”, amount_var),
        _ => (),
}
```

Aufgabe 3

Erstellt ein Match, das eine Zahl matcht und bei den Primzahlen bis 10 “Primzahl!”, bei Zahlen von 11-99 “Unter 100”, bei Zahlen von 107-121 “Diese Reihe von Zahlen ist willkürlich ausgewählt” und bei allen anderen Zahlen nichts ausgibt.

Refutability

- “Anfechtbarkeit” der Garantie des Matches
- irrefutable: Patterns, die immer matchen, wie zum Beispiel
  x = 0
- refutable: Patterns, die nicht immer matchen, zum Beispiel
  if let Some(x) = value.
  Da value auch None sein kann, matcht dieses Pattern nicht zwangsweise.

Refutability

- Teilweise nur refutable oder irrefutable Pattern erlaubt:
- *let* erlaubt nur irrefutable Pattern, *let Some(x) = value* ist nicht erlaubt
- Mit *if let* wäre das dafür möglich
- *if let* lässt nur refutable Pattern zu, *if let x = 5* ist also nicht möglich
- Match-Arme erwarten refutable, außer beim letzten, “sammelnden” Arm

→ Zum Beheben von Fehlern nützlich

Advanced Types

Newtype

- Umgehen der Einschränkung, dass entweder Trait oder Type in der lokalen Crate vorhanden sind
- Wrapper-struct um Type
- Wrapper implementiert Trait
- Wrapper fehlen die Funktionen des Types, da er als neuer Type gilt
- Trait *Deref* behebt das (muss von Wrapper implementiert werden)
- Alternativ einzelne Funktionen manuell implementieren

Newtype

```
struct Wrapper(typeOutsideOfCrate);

impl TraitOutsideOfCrate for Wrapper {
        fn traitFunction() {
                //...
        }
}

fn main() {
        let w = Wrapper(<literal of typeOutsideOfCrate>);
        w.traitFunction();
}
```

Newtype - Nutzen

- Typsicherheit gewährleisten (bei eigenen Typen)
- eine private innere API in eine eingeschränkte öffentliche API verpacken
- interne Implementationen zu verstecken

Aufgabe 4

Ihr habt den Typ “Bomb” mit den Funktionen move_to(destination: String), throw(), arm(), drop(), kick(). Da ihr die Bombe ein paar Kindern als Ersatz für ihren verlorenen Ball zum Spielen geben wollt, jedoch nicht dafür verantwortlich sein wollt, dass die Kinder sich in die Luft sprengen, sollen sie die Bombe nicht scharf machen können. Nutzt deshalb das Newtype-Pattern, um eine “SaveBomb” zu erstellen. Diese kann alles, was auch Bomb kann, jedoch kann die Funktion arm() nicht aufgerufen werden.

Typ-Synonyme

- type Meter = i32;
- Meter verhält sich exakt wie i32
- keine Typsicherheit, Meter und i32 können frei miteinander vertauscht werden
- Nutzen: lange Typen in kürzere, einfacher lesbare/verwendbare Typen verpacken → z.B. für Result<T, E>

Never (empty type)

```
fn something() -> ! {
    // do something...
}
```

- Typ -> *!*
- returnt niemals
- gibt deshalb niemals einen Typ zurück
- Kann in Match verwendet werden, um Arm nichts zurückgeben zu lassen
  → Arm muss an sich immer etwas zurückgeben
- Beispiele: panic!, loop

Dynamically Sized Types (DST)

- Rust muss immer wissen wie viel Speicher ein Typ braucht
- Unmöglich für Typen wie String
- Deshalb hat &str eigentlich zwei Werte: Start und Länge
- Deshalb wird *let text: &str* und nicht *let text: str* genutzt
- Traits sind ebenfalls DSTs
- Trait *Sized* bestimmt ob ein Typ DST ist oder nicht
- Alle generischen Funktionen implementieren implizit *Sized*

Dynamically Sized Types (DST)

```
fn generic<T: ?Sized>(t: &T) {
    // do stuff
}
```

- Durch ?Sized → T kann *Sized* sein, muss aber nicht

- Deshalb muss t: T zu t: &T werden