

6.

Structs, impl-Block und Methodensyntax

Typen in Rust

- Bisher:
 - Primitive Typen (`u32`, `f32`, `bool`, ...)
 - *Anonyme*, nutzerdefinierte Typen
 - Tuple: `(T, U, ...)`
 - Arrays: `[T; N]`
 - Slices: `[T]`
- Jetzt: *benannte*, nutzerdefinierte Typen
 - Struct
 - Tuple-Struct
 - Enum

Structs

- Vergleichbar mit Java-Klassen
- Typdefinition enthält nur Information über Felder
 - Keine Methoden oder ähnliches!

```
struct Point {  
    x: f32,  
    y: f32,  
}  
  
struct Student {  
    id: u32,  
    name: String,  
}
```

```
let point = Point {  
    x: 5.0,  
    y: 3.14,  
};  
  
let peter = Student {  
    id: 123456,  
    name: „Lustig“.into(),  
};  
println!("{}", peter.name);
```

Syntax

Struct Definition

```
struct <TypeName> {  
    <field_name_1>: <field_type_1>,  
    <field_name_2>: <field_type_2>,  
    ...,  
}
```

Struct Initializer*

```
<TypeName> {  
    <field_name_1>: <value_1>,  
    <field_name_2>: <value_2>,  
    ...,  
}
```

- Letztes Komma nicht *nötig*
 - Liste in einer Zeile → letztes Komma weglassen
 - Liste in mehreren Zeilen → letztes Komma hinzufügen
- Kann auch in Funktion definiert werden
- „Muss ich immer alle Felder manuell initialisieren?!“

Konstruktor

- Lösung mit schon bekannten Techniken? (**Point** mit **[0, 0]**)

```
fn origin() -> Point {  
    Point {  
        x: 0.0,  
        y: 0.0,  
    }  
}
```

```
let point = origin();  
println!(  
    "{}, {}",  
    point.x,  
    point.y,  
); // output: 0, 0
```

- *Unschön*: Funktion ist global, nicht verbunden mit **Point**
- *Schön*: Name kann Funktion beschreiben (i. G. z. Konstruktoren)

Assoziierte Funktionen

Wie statische Methoden
in C++, Java, ...!

- Funktion lebt im Namensraum des Typen

```
impl Point {  
  fn origin() -> Point {  
    Point {  
      x: 0.0,  
      y: 0.0,  
    }  
  }  
}
```

```
let point = Point::origin();  
//          ^^^^^^^  
println!(  
  "{}, {}",  
  point.x,  
  point.y,  
); // output: 0, 0
```

- Definition im **impl**-Block des Typen
- Zugriff via **Type::function** (:: → Trennzeichen für Namenspfade)

Assoziierte Funktionen

- Beispiel: **String**

- `new()` → erzeugt leeren String
 - `new` ist kein Keyword!

- `with_capacity()` → leerer String, aber allokiert bereits einen Buffer

- Besonderheit in **impl**-Block:

→ „Self“ steht für aktuellen Typen

```
let a = String::new();
let b = String::with_capacity(10);
```

```
impl Point {
    fn origin() -> Self {
        Point {
            x: 0.0,
            y: 0.0,
        }
    }
}
```

Funktioniert leider
([noch](#)) nicht beim
Struct-Initializer 😞

Copy und Clone von neuen Typen

- Ist `Point` `Copy`? Ist `Point` `Clone`? → **Nope**

```
#[derive(Clone, Copy)]  
struct Point {  
    x: f32,  
    y: f32,  
}
```

- `Copy` und `Clone` können „derived“ werden
 - `#[derive(Clone)]` für `Clone`
 - `#[derive(Clone, Copy)]` für `Copy` (und `Clone`)

- `#[...]` sind „Attribute“
 - Für unterschiedlichste Zwecke (später mehr)
 - Beziehen sich auf das darauffolgende Item
 - `#![...]` Attribute beziehen sich auf Elternitem

Können wir einen `Point` eigentlich mit `{:?}` ausgeben?

→ `#[derive(Debug)]`

Instanzgebundene Funktionen

- Lösung mit schon bekannten Techniken? (`move_up()`, `y + 1`)

```
fn move_up(old: &Point) -> Point {
    Point {
        x: old.x,
        y: old.y + 1,
    }
}
```

```
fn move_up(p: &mut Point) {
    p.y += 1;
}

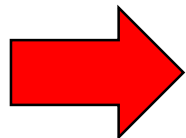
let mut p = Point::origin();
move_up(&mut p);
```

- *Schön*: funktioniert einfach so!
- *Unschön*:
 - Steht nicht in Verbindung mit dem **Point**-Typen
 - Geschachtelte Aufrufe sind unübersichtlich: `d(c(3, b(a(p))))`

Self als Parameter

„Steht nicht in Verbindung mit dem **Point**-Typen“

```
impl Point {  
    fn move_up(p: &mut Self) {  
        p.y += 1;  
    }  
}  
  
let mut p = Point::origin();  
Point::move_up(&mut p);
```

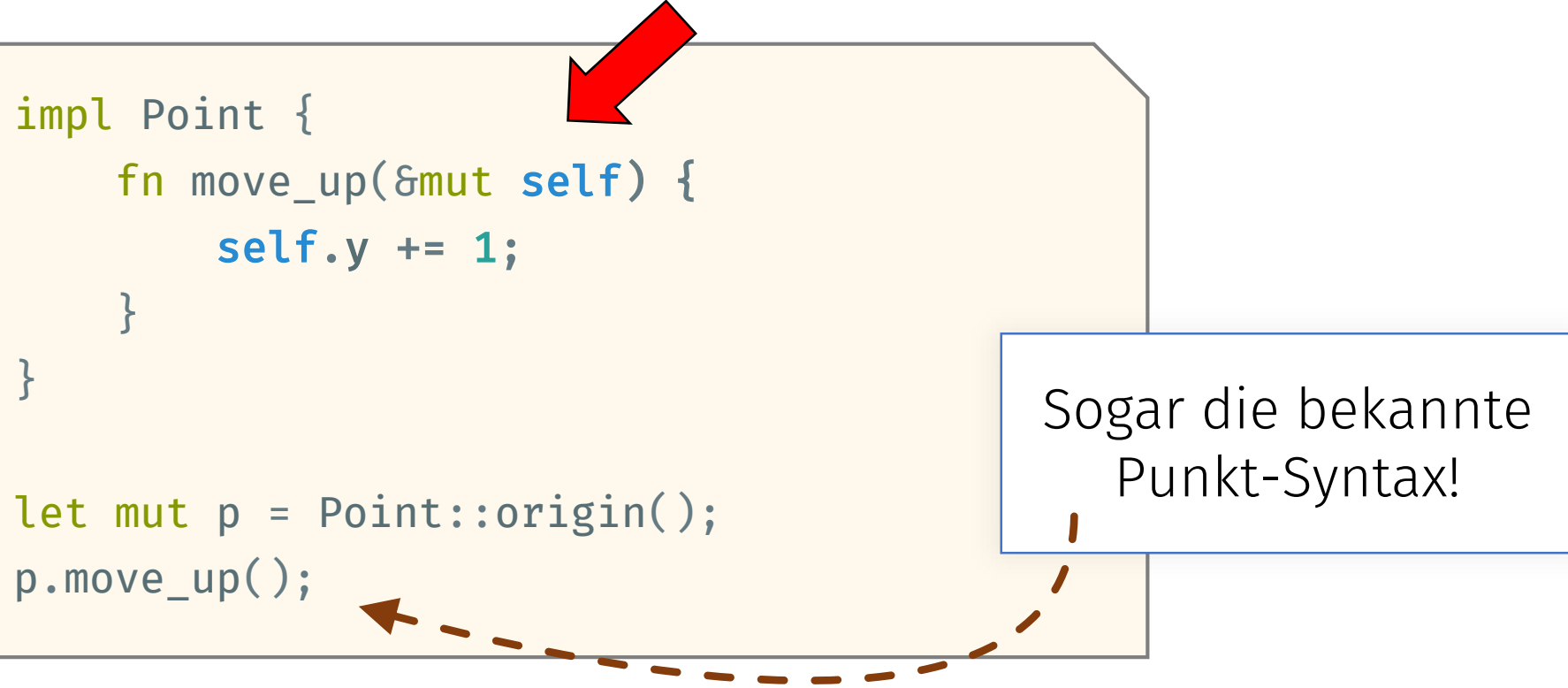


Immer noch umständlicher Aufruf!

self-Parameter

Spezielle Syntax für ersten **Self** Parameter:

```
impl Point {  
    fn move_up(&mut self) {  
        self.y += 1;  
    }  
}  
  
let mut p = Point::origin();  
p.move_up();
```



Sogar die bekannte
Punkt-Syntax!

self-Parameter

- **self** \approx **this** aus C++/Java
 - Muss aber explizit als Parameter deklariert werden
 - In C++ und Java ist **this**-Parameter versteckt
 - **self muss** zum Zugriff auf Felder genutzt werden!

```
impl Point {  
    fn move_up(&mut self) {  
        y += 1; // error!  
    }  
}
```

	Rust	C++ ¹	Java
Statische Methode/ Assoziierte Funktion	<code>fn foo(args...) {}</code>	<code>static void foo(args...) {}</code>	<code>static void foo(args...) {}</code>
Immutable self / this	<code>fn foo(&self, args...) {}</code>	<code>void foo(args...) const {}</code>	<i>nicht möglich</i>
Mutable self / this	<code>fn foo(&mut self, args...) {}</code>	<code>void foo(args...) {}</code>	<code>void foo(args...) {}</code>
Consuming self / this	<code>fn foo(self, args...) {}</code>	<code>void foo(args...) && {}</code>	<i>nicht möglich</i>

¹ C++ ist „unsicherer“, da das Typsystem umgangen werden kann (z.B. `const_cast`)

Arten von Funktionen (bisher)

„Freie Funktionen“

- *Nicht* in `impl`-Block
- `fn foo(arg1: type1, ...)`

„Assoziierte Funktionen“

- *In* `impl`-Block
- `fn foo(arg1: type1, ...)`

„Methoden“

- *In* `impl`-Block
- `fn foo(self, arg1: type1, ...)`
- `fn foo(&self, arg1: type1, ...)`
- `fn foo(&mut self, arg1: type1, ...)`

Zugriff auf
`Self`-Typen

Zugriff auf
`self`-Instanz

Beispiel

```
impl Monster {  
    /// Returns a monster with the specified  
    /// strength.  
    fn with_strength(strength: u8) -> Self {  
        Monster {  
            health: 100,  
            strength: strength,  
        }  
    }  
  
    /// Returns a monster with strength 10.  
    fn weak() -> Self {  
        Self::with_strength(10)  
    }  
}
```

```
/// An evil monster.  
///  
/// A new monster has 100 health points. It  
/// gets weaker when the health is low.  
struct Monster {  
    health: u8,  
    strength: u8,  
}
```

```
// Note: all methods and functions are  
// usually defined in the same impl-block  
impl Monster {  
    /// Returns whether or not there are  
    /// any life points left.  
    fn is_alive(&self) -> bool {  
        self.health > 0  
    }  
}
```

Beispiel

```
impl Monster {  
    /// Returns the monster's current attack  
    /// strength. If the monster has less than  
    /// 20 health points, its attack is only  
    /// half as strong.  
    fn attack_strength(&self) -> u8 {  
        if self.health < 20 {  
            self.strength / 2  
        } else {  
            self.strength  
        }  
    }  
}
```

```
/// An evil monster.  
///  
/// A new monster has 100 health points. It  
/// gets weaker when the health is low.  
struct Monster {  
    health: u8,  
    strength: u8,  
}
```

```
impl Monster {  
    /// Reduces the monster's health points  
    /// according to the incoming attack's strength.  
    fn endure_attack(&mut self, strength: u8) {  
        self.health =  
            self.health.saturating_sub(strength);  
    }  
}
```

Beispiel

```
fn main() {  
    let mut wolfgang = Monster::weak();  
    let mut sabine = Monster::with_strength(13);  
  
    while wolfgang.is_alive() && sabine.is_alive() {  
        wolfgang.endure_attack(sabine.attack_strength());  
        sabine.endure_attack(wolfgang.attack_strength());  
  
        println!(  
            "Wolfgang: {} HP, Sabine: {} HP",  
            wolfgang.health,  
            sabine.health,  
        );  
    }  
}
```

```
Wolfgang: 87 HP, Sabine: 90 HP  
Wolfgang: 74 HP, Sabine: 80 HP  
Wolfgang: 61 HP, Sabine: 70 HP  
Wolfgang: 48 HP, Sabine: 60 HP  
Wolfgang: 35 HP, Sabine: 50 HP  
Wolfgang: 22 HP, Sabine: 40 HP  
Wolfgang: 9 HP, Sabine: 35 HP  
Wolfgang: 0 HP, Sabine: 30 HP
```


Private und Public?

- Derzeit nur zwei Modi
 - Mit **pub** Modifier → public
 - Ohne Modifier → module-internal
- „Module“ in späteren Kapiteln
 - Zurzeit alles in einem Modul!

```
struct Monster {  
    health: u8,  
    strength: u8,  
}  
  
struct Point {  
    ↗ pub x: f32,  
    ↘ pub y: f32,  
}  
  
impl Monster {  
    ↗ pub fn weak() -> Self { ... }  
    ↘ pub fn is_alive(&self) -> bool { ... }  
    fn internal_function() { ... }  
}
```

Consuming self

Eher selten!

Wofür ist `fn foo(self) da?`

- Für kleine **Copy**-Types: vermeidet Indirektion
- Wiederverwendung von Ressourcen

```
impl String {  
    fn into_bytes(self) -> Vec<u8> { ... }  
}
```

- Vermeidung von Clones, insb. bei generischen Typen

Auto-Borrowing

(oder Auto-Dereferencing)

- Punkt-Syntax wandelt automatisch um
 - Zwischen: Value/Borrow/MutBorrow types
- Sonst: immer explizit

```
let mut a = Foo;  
  
// all of those work  
a.takes_ref();  
a.takes_mut_ref();  
a.takes_value();
```

```
impl Foo {  
    fn takes_value(self) {}  
    fn takes_ref(&self) {}  
    fn takes_mut_ref(&mut self) {}  
}
```

```
let b = Foo;  
let c = &mut b;  
  
// these work as well  
c.takes_ref();  
c.takes_mut_ref();  
  
// this one works, *if* the  
// type implements `Copy`  
c.takes_value();
```