3.

Ownership-System (Teil 1)

Das Ownership-System

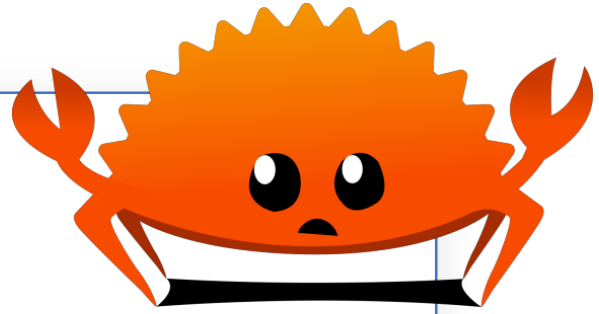
Wozu?

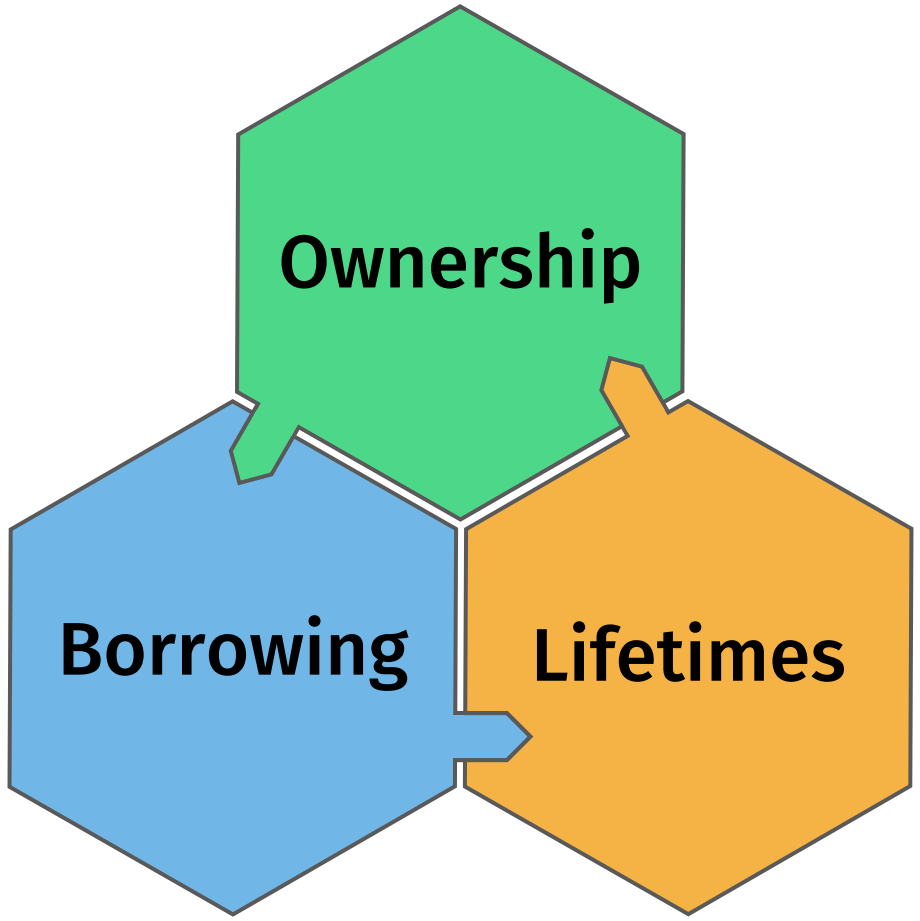
- Sicherheitsgarantien zur Kompilierzeit
- Kann zu anderen Zwecken genutzt werden

Warum lernen?

- Vorraussetzung für weitere Themen
- Vermittelt viele Konzepte u. Grundlagen

Hauptgrund für Rust's steile Lernkurve...





Die wichtigen Konzepte

Das Ownership-System

Wozu?

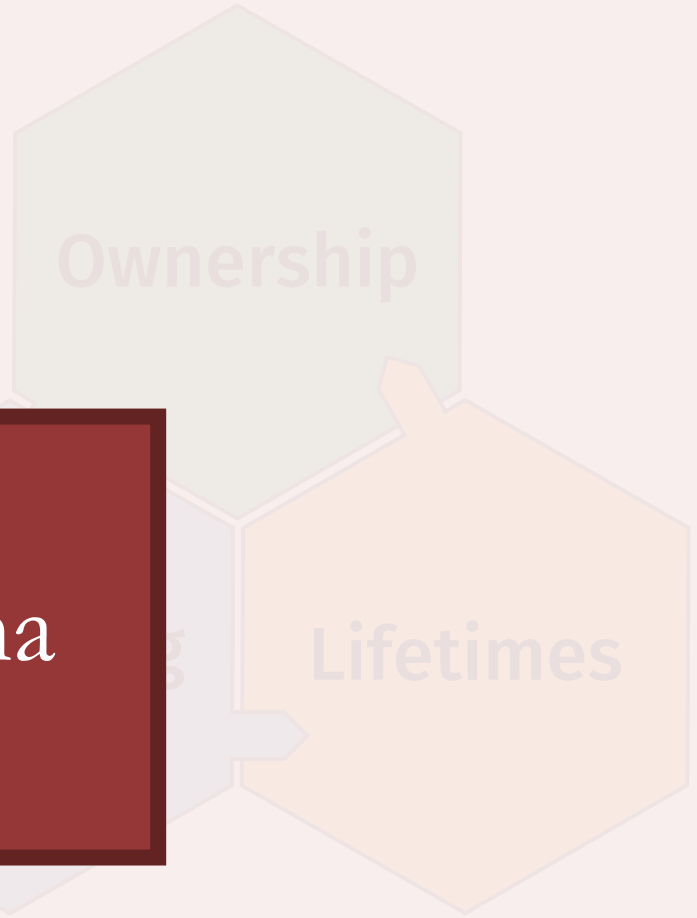
- Sicherheitsgarantien zur Kompilierzeit
- Kann zu and

Warum lern

- Vorraussetzu
- Vermittelt viele Konzepte u. Grundlagen

Hauptgrund für Rust's steile Lernkurve...

Die wichtigen Konzepte



Wichtig:

Wir besprechen das Thema *erstmal* nur grob!

Erstmal: *Variable Binding*

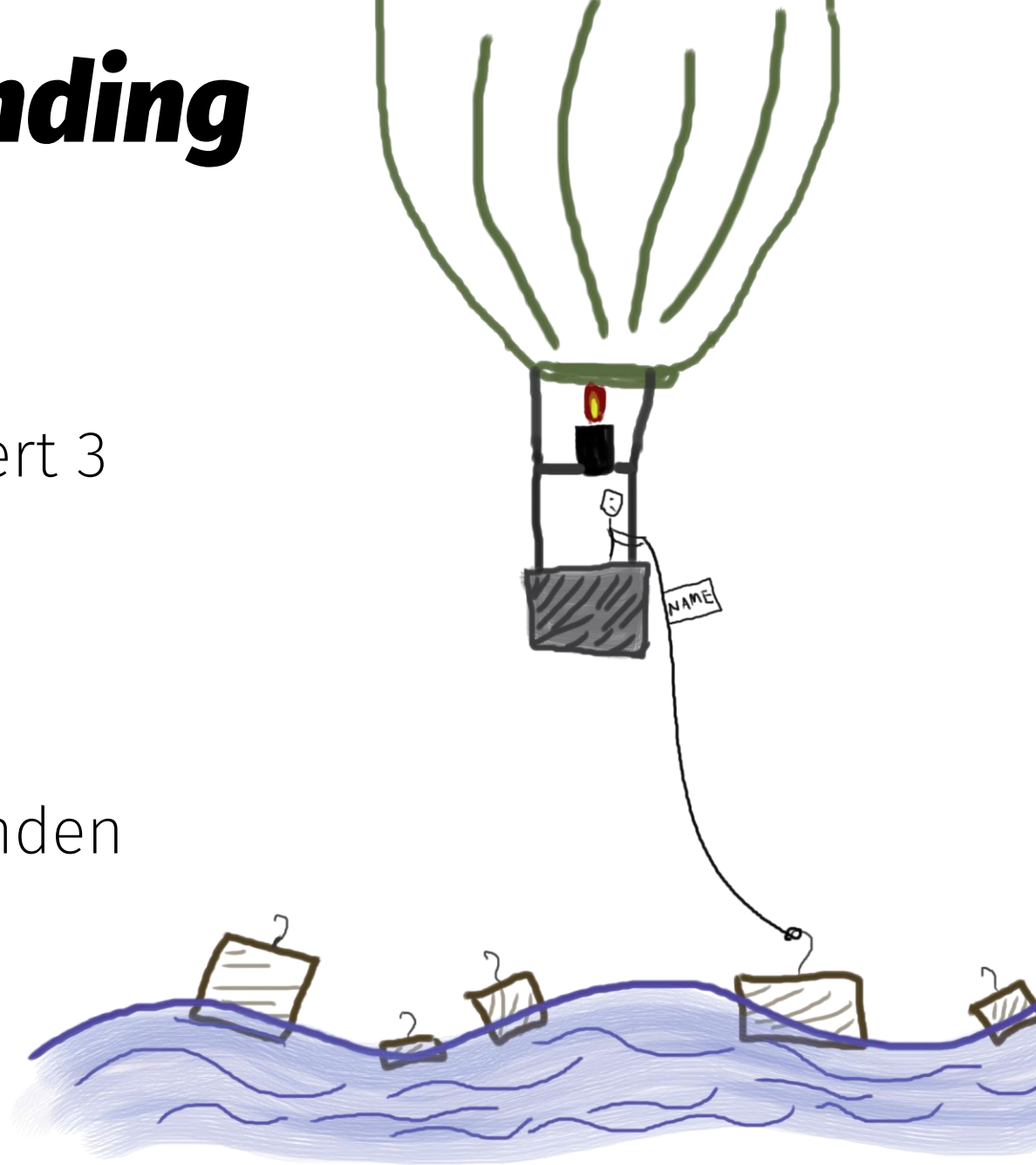
```
let a = 3;
```

- **a** ist nur ein Name, der an den Wert 3 gebunden ist

```
let (x, _) = returns_tuple();
```

- Zweites Tupleelement nicht gebunden

„*See*, auf dem *Werte* schwimmen.
Manche *binden* wir an *Namen*.“



Ownership

- Variable Binding *besitzt* den „Wert“ (oder „Objekt“, „Instanz“, …)
- Wenn Besitzer *out-of-scope* → Wert wird zerstört

```
let a: String = "hi".to_string();
println!("a: {}", a);

let b = a;
println!("a: {}", a);
```

```
error[E0382]: use of moved value: `a`
 --> <anon>:6:23
  |
5 | let b = a;
  |     - value moved here
6 | println!("a: {}", a);
  |                   ^ value used here after move
```

- *Move Semantics*: Ownership wird übertragen (an **b**)
  - Kein impliziter Klon/implizite Referenz

Move & Funktionen

```rust
fn greet(name: String) {
    println!("Hello {}!", name);
}
fn say_goodbye(name: String) {
    println!("Goodbye {}!", name);
}

let peter = "Peter".to_string();
greet(peter);
say_goodbye(peter);
```

```
error[E0382]: use of moved value: `peter`
 --> <anon>:8:7
  |
7 | greet(peter);
  |       ----- value moved here
8 | say_goodbye(peter);
  |             ^^^^^ value used here after move
```

Wie reparieren wir das?

Move & Funktionen

```
fn greet(name: String) -> String {
    println!("Hello {}!", name);
    name
}
fn say_goodbye(name: String) -> String {
    println!("Goodbye {}!", name);
    name
}

let peter = "Peter".to_string();
let peter = greet(peter);
say_goodbye(peter);
```

Das funktioniert zwar, aber …

… meh!

Borrowing

```
fn greet(name: &String) {
    println!("Hello {}!", name);
}
fn say_goodbye(name: &String) {
    println!("Goodbye {}!", name);
}

let peter = "Peter".to_string();
greet(&peter);
say_goodbye(&peter);
```

- **&** im Expression-Position:
  - Leiht Wert aus/erzeugt Referenz auf Wert
- **&** in Typposition:
  - Referenztyp
  - **&T** → „Ein ausgeliehenes **T**“ oder „Referenz auf ein **T**“
- Pointer auf Maschinenebene!

Was ist wenn **`greet()`** den Namen ändert? 

Mutable Borrows

```
fn greet(name: &mut String) {
    println!("Hello {}!", name);
    *name = "Susi".to_string();
}
fn say_goodbye(name: &String) {
    println!("Goodbye {}!", name);
}

let mut peter = "Peter".to_string();
greet(&mut peter);
say_goodbye(&peter);
```

- `&mut` im Expression-Position:
  - Leiht Wert veränderbar aus/erzeugt *mutable* Referenz
- `*` im Expression-Position:
  - Dereferenziert Wert, „holt den Wert hinter der Referenz“
- `&mut` in Typposition:
  - Mutable-Referenztyp
  - `&mut T` → „Ein veränderbar ausgeliehenes `T`“ oder „mutable Referenz auf ein `T`“

Borrowing: Einschränkungen

Zu einem bestimmten Zeitpunkt:

Beliebig **viele** *immutable* Borrows …

— oder —

… genau **einen** *mutable* Borrow

Aliasing *xor* Mutability

- *Aliasing*: ein Wert wird gleichzeitig mehrfach referenziert
- Wenn beides zusammen: i.d.R. schlimme Dinge

Borrowing: Einschränkungen

```
let mut orig = 3;
{
    let ib = &orig; // : &i32
    let ib2 = &orig;
    println!("{} is {}", ib, ib2);  // output: 3 is 3

    let mb = &mut orig;  // error: cannot borrow `a` as mutable because
                         //        it is also borrowed as immutable

    orig += 2;  // error: cannot assign to `orig` because it is borrowed
}

// ib and ib2 are out of scope: Everything is possible again
let mb = &mut orig;
*mb = 27;
```

Lifetimes

```
&'a u32
&'b mut String
```

Wenn explizit annotiert …

- Jede Referenz besitzt eine „Lifetime“
- Compiler …
  - … weiß, wie lange Original gültig
  - … sorgt dafür, dass alle Referenzen gültig sind
    - (keine Referenz lebt länger als Original)

- Oft nicht nötig, manuell Lifetimes zu annotieren
- Später tauchen wir viel tiefer ins Thema ein! 😉

Copy & Clone

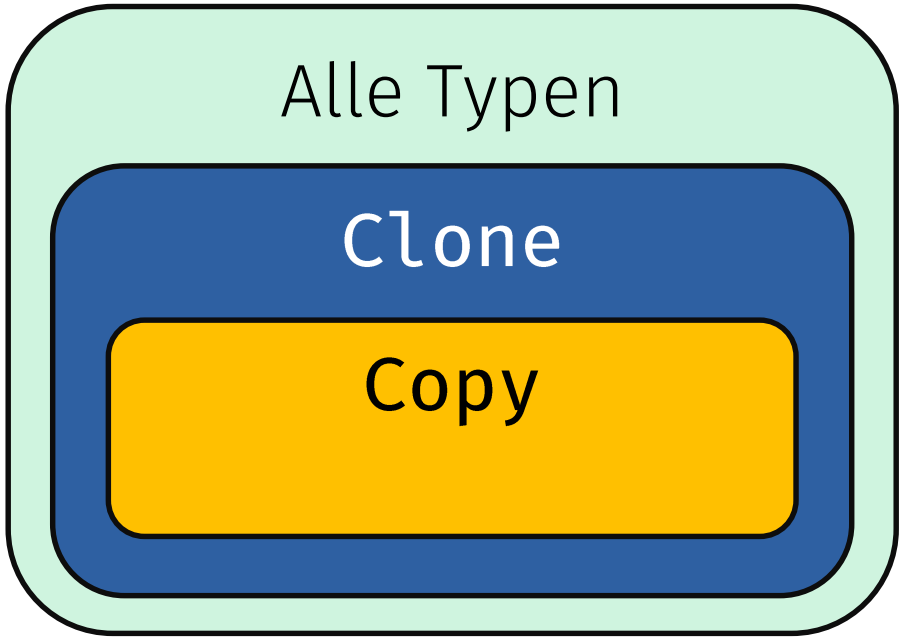
```
let a = 3; // i32 is Copy
let b = a;
println!("{}", a); // OK
```

- `Copy`-Typen (≈ C++'s POD-types)
  - „Types that can be copied by simply copying bits (i.e. **`memcpy`**)"
  - Beispiele: **`{integer}, {float}, bool, char,`** …
  - Gegenbeispiele: **`String`** (verwaltet zusätzlichen Speicher)
  - Keine *Move*-, sondern *Copy Semantics*!

- `Clone`-Typen
  - Typen, die dupliziert (geklont) werden können
  - Klonen immer explizit durch **`.clone()`**
  - Beispiel: **`String, {integer}, …`**

```
let a = "hi".to_string();
let b = a.clone();
println!("{}", a); // OK
```

Zusammenfassung



- *Move Semantics*: 
- *Copy Semantics*: 

- Werte haben immer einen Besitzer
- Borrowing mit „***aliasing*** *xor* ***mutability***“-Einschränkung

Wichtig:

Ein Borrow bedeutet, dass Daten referenziert werden, die ***woanders leben***! 🏠

Moment mal…

- Rust Compiler kann nicht immer wissen, dass etwas sicher ist!

➔ „`unsafe`-Rust“

```
unsafe {
    // here be dragons…
}
```

- Mehr Möglichkeiten
- Behandeln wir später!

Warum „zwei Sprachen“?

- Wenig unsicherer Code
  → wenige mögliche Fehlerquellen
- Einfach zu suchen/finden
- Hinter sicheren Abstraktionen versteckt

Wichtig

`unsafe`-Rust muss quasi *nie* benutzt werden!