

**17.**

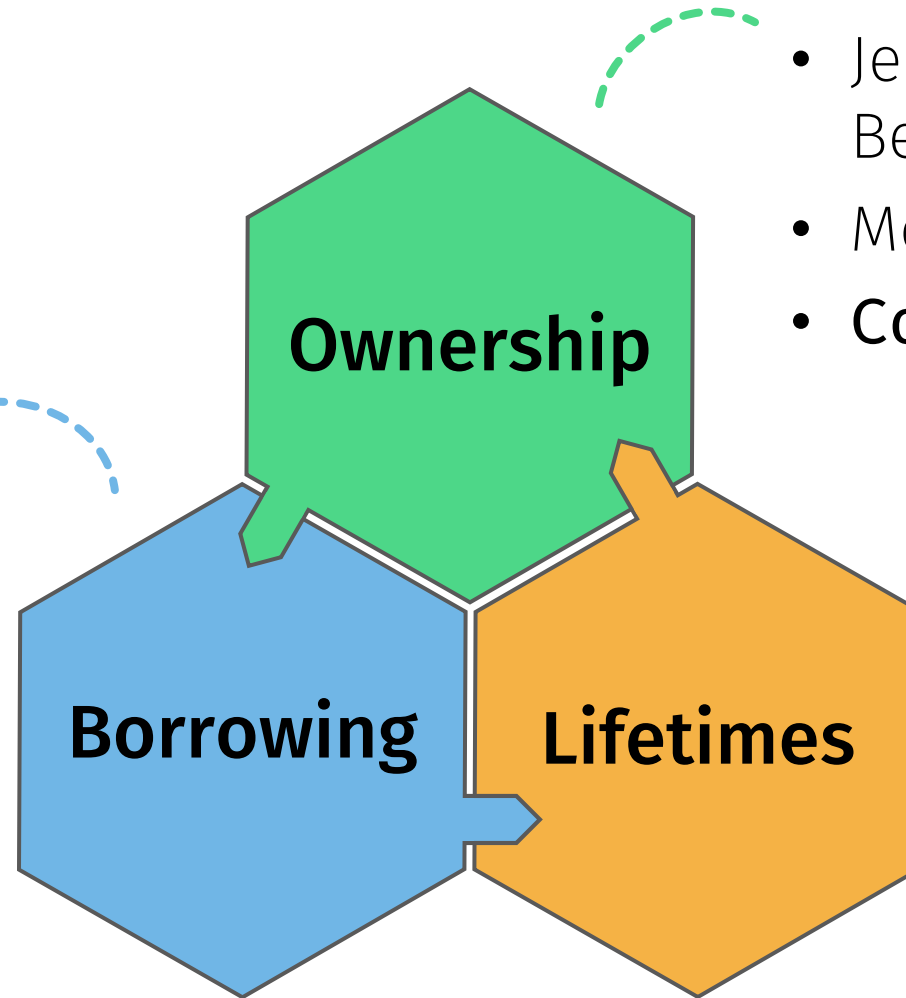
---

Lifetimes

---

# Das Ownership-System

- Nutzung ohne Ownership
- Immutable/Mutable
- Aliasing *xor* Mutability



- Jede Variable hat *einen* Besitzer
- Move Semantics
- **Copy**-Types

???

# Was wir schon wissen

- Referenz zeigt immer auf gültiges Objekt
  - Kein „*use after free*“
  - Scope von Referenz kleiner als Scope von referenziertem Wert!
- Scope von Variablen wie Stack
  - LIFO Prinzip

**error:** `x` does not live long enough

[...]

= **note:** values in a scope are dropped in the opposite order they are created

```
// x does not live long enough!  
fn return_stack() -> &u64 {  
    let x = 0u64;  
    &x  
}
```

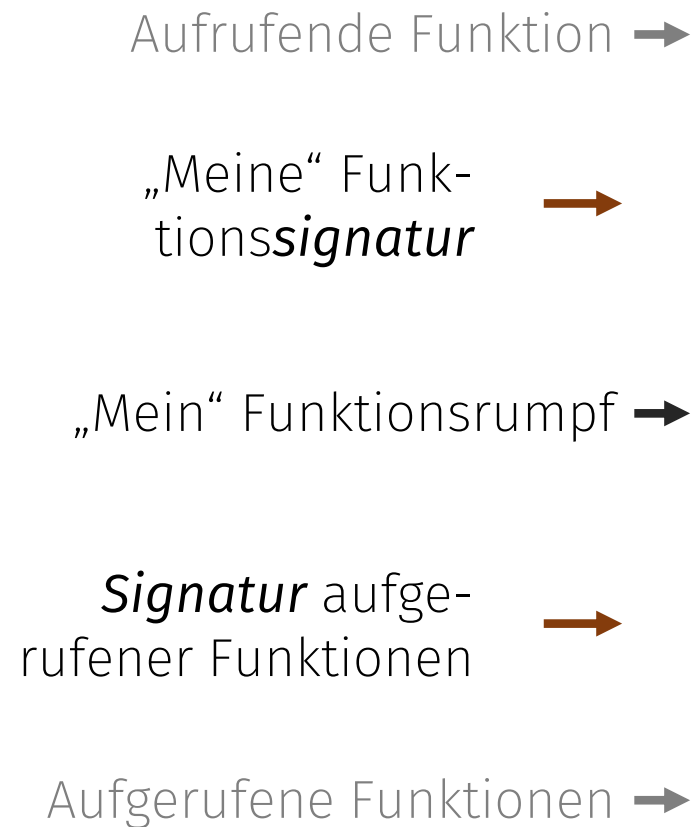
```
// x does not live long enough!  
let r: &u64 = {  
    let x = 0u64;  
    &x  
};
```

```
fn foo() {  
    let r: &u64;  
    let x = 0u64;  
    r = &x;  
}
```



# Compiler Analysen

- Compiler stellt sicher:
  - Keine Referenz lebt länger als der referenzierte Wert
  - Aliasing *xor* Mutability
- Analyse von **fn**-Rümpfen
- Nutzt zur Analyse nur:
  - Eigene Signatur
  - Eigenen Funktionsrumpf
  - Signatur von aufgerufenen Funktionen



Compiler guckt nicht **in** aufgerufene Funktionen!

# Quiz

```
fn foo(i: &u8) -> &u8 { ... }
```

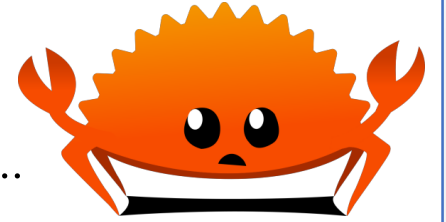
```
// Is this safe?
```

```
let r: &u8 = {  
    let x = 3;  
    foo(&x)  
};
```

```
// Is this safe?
```

```
let mut y = 3;  
let r = foo(&y);  
x += 1;
```

Kommt auf den Funktionsrumpf an...



Wäre *unsicher*:

```
fn foo(i: &u8) -> &u8 {  
    i  
}
```

Wäre *sicher*:

```
static STATIC_NUM: u8 = 27;  
  
fn foo(i: &u8) -> &u8 {  
    println!("{}", i);  
    &STATIC_NUM  
}
```

# Quiz

```
fn bar(i: &u8, j: &u8) -> &u8 { ... }  
  
// Is this safe?  
let a = 100;  
let r: &u8 = {  
    let b = 200;  
    bar(&a, &b)  
};
```

- Analyse ohne Rumpf unmöglich
  - Immer konservativ sein und annehmen, es ist unsicher?

Wäre *unsicher*:

```
fn bar(i: &u8, j: &u8) -> &u8 {  
    j  
}
```

Wäre *sicher*:

```
fn bar(i: &u8, j: &u8) -> &u8 {  
    i  
}
```



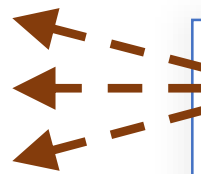
Nötige Informationen in Signatur annotieren!

# Lifetime Annotationen

- Für unser Problem:
  - Annotieren, wie lange Wert hinter zurückgegebener Referenz lebt

```
fn bar(i: &u8, j: &u8) -> &u8 { ... }
```

- Wie lange könnte der Wert hinter zurückgegebener Referenz leben?
  - So lange wie der Wert hinter **i**
  - So lange wie der Wert hinter **j**
  - Für immer (Wert lebt in **.data**)
- Lifetimes kann man sich nicht aus den Fingern saugen!



**Das waren alle Möglichkeiten!**

Von Referenz-Argumenten oder  
**.data** ausgeliehen!

# Lifetime Annotationen

```
fn bar<T>
```

- „Für irgendeinen Typen **T** ...“

```
fn bar<'a>
```

- „Für irgendeine Lifetime **a** ...“

```
&'a T
```

- „Eine Referenz auf ein **T**, welches (mindestens) für die Lifetime **a** lebt“

```
fn bar<'a>(i: &'a u8, j: &u8)
    -> &'a u8
{
    i
}
```

```
fn bar<'a>(i: &u8, j: &'a u8)
    -> &'a u8
{
    j
}
```

```
fn bar(i: &u8, j: &u8)
    -> &'static u8
{
    &STATIC_U8
}
```



# Beispiel

```
fn without_prefix(s: &str, prefix: &str)
    -> &str
{
    if s.starts_with(prefix) {
        &s[prefix.len()..]
    } else {
        s
    }
}
```

```
// "cdef"
```

```
without_prefix("abcdef", "ab");
```

```
// "ann-kristin"
```

```
without_prefix("ann-kristin", "anna");
```

**error[E0106]:** missing lifetime specifier

[...]

= **help:** this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `s` or `prefix`

```
fn without_prefix<'a>(
    s: &'a str,
    prefix: &str
) -> &'a str {
    ...
}
```

# Lifetimes sind immer da!

- Jede Referenz hat immer eine Lifetime mit sich assoziiert
  - Compiler besitzt Beschreibung des Scopes
  - Wir können Scope nur benennen, nicht definieren!
    - Wir können uns den Scope/die Lifetime nicht aussuchen!
    - **'static'** teilweise speziell, trotzdem nur ein Name
- Als lokale Variable:
  - Benennung nicht nötig (Typinferenz)
  - Benennung meist auch unmöglich
    - Ausnahme: **'static'**

```
let x = 3;
{
    let y = 4;
    // We can't (and don't need
    // to) specify the lifetime!
    let r: &u8 = &y;
}
```

# Lifetimes von Funktionsargument

- Lifetime kann mit jedem Aufruf variieren
  - Wie Generics: „Für eine beliebige Lifetime 'a ...“

```
fn print(s: &str)
```

=

```
fn print<'a>(s: &'a str)
```

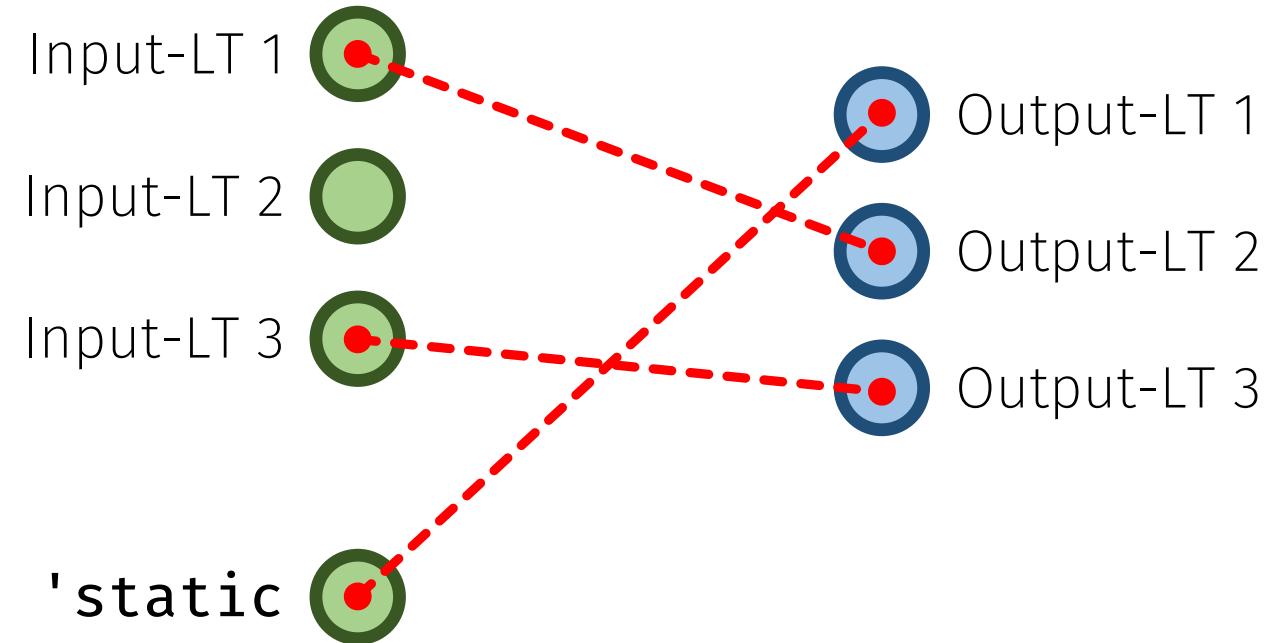
- Wenn Lifetime nicht benannt: Anonyme Lifetime
  - Funktioniert genau so wie benannte Lifetime
  - Lifetime nur zum Zuordnen und Bounden benennen

## Namensgebung

- Oft nur 'a, 'b, ...
- Sonst klein geschrieben und ein Wort ('input)

# Das Lifetime-Spiel

- Gegeben: *Input-Lifetimes*
- Müssen zugewiesen werden: *Output-Lifetimes*
  - Entweder von *Input-Lifetime*
  - Oder **'static'**
- Einfache Fälle werden von „Lifetime Elision“ abgedeckt



# Lifetime Elision

## Regeln zum Weglassen von Lifetime-Parametern:

1. Jede weggelassene Lifetime der Argumente wird ein eigener Lifetime-Parameter
2. Wenn es nur eine Input-Lifetime gibt, wird diese allen weggelassenen Output-Lifetimes zugewiesen
3. Wenn es mehrere Input-Lifetimes gibt, aber eine davon ist die von „**&self**“ oder „**&mut self**“, wird diese allen Output-Lifetimes zugewiesen

```
fn f(a: &u8)
    -> &u8
```

Regel 2

```
fn f<'a>(a: &'a u8)
    -> &'a u8
```

```
// impl Option<T>:
fn ref_or_print(&self, e: &str)
    -> &T
```

Regel 3

```
// impl Option<T>:
fn ref_or_print<'a>(&'a self, e: &str)
    -> &'a T
```

# Fehler bei Lifetime-Elision

```
enum Redlight { Green, Yellow, Red }

impl Redlight {
    fn as_str(&self) -> &str {
        use Redlight::*;

        match *self {
            Green => "green",
            Yellow => "yellow",
            Red => "red",
        }
    }
}
```

```
// error: does not live long enough
let s = {
    let rl = Redlight::Red;
    rl.as_str()
};
```

- Rückgabebetyp manuell Lifetime 'static' zuweisen, damit Code funktioniert!

Lifetime von Referenzen automatisch runtergestuft („Typumwandlung“)

# Lifetime mehrmals „nutzen“

```
// error: missing lifetime specifier
fn choose(x: &str, y: &str) -> &str {
    if random() {
        x
    } else {
        y
    }
}
```

```
// x and y have "the same" lifetime
fn choose<'a>(x: &'a str, y: &'a str)
    -> &'a str
{ ... }

// works (lifetime downgrade):
let s = "hi".to_string();
choose(&s, "bye");
```

- Ein Lifetime-Parameter in mehreren Referenzen:
  - Beide haben „gleiche“ Lifetime
- Unterschiedliche Lifetimes durch automatische Umwandlung möglich (die längere wird runtergestuft)

# Referenzen in anderen Typen

```
struct RefWrapper {  
    r: &u8,  
}  
  
// There is no output lifetime that  
// we need to assign...  
fn foo(x: &u8, y: &u8) -> RefWrapper {  
    RefWrapper { r: x }  
}  
  
// uhm... is that safe now?  
let r: RefWrapper = {  
    let x = 3;  
    foo(&x, &x)  
};
```

Funktioniert nicht

- Lifetimes verstecken unmöglich
- Typen bekommen Lifetime-Parameter

```
struct RefWrapper<'a> {  
    r: &'a u8,  
}  
  
fn foo<'a>(x: &'a u8, y: &u8)  
-> RefWrapper<'a>  
{  
    RefWrapper { r: x }  
}
```



# Referenzen in anderen Typen

```
struct RefWrapper<'a> {  
    r: &'a u8,  
}
```

- LT-Parameter drückt aus, dass etwas geborrowed ist
  - Es gelten Einschränkungen für Typen, die etwas referenzieren
- Referenzen: auch Typen mit LT-Parameter
  - Nur andere Syntax
  - Bisher gelernte Regeln gelten auch für eigene Typen mit LT-Parameter

```
fn wrap(x: &u8) -> RefWrapper {  
    RefWrapper { r: x }  
}
```

Lifetime Elision

# Methoden für RefWrapper

```
struct RefWrapper<'a> {  
    r: &'a u8,  
}  
  
impl<'a> RefWrapper<'a> {  
    fn cloned(&self) -> u8 {  
        *self.r  
    }  
  
    // Lifetime-Elision:  
    // out lifetime = self-lifetime  
    fn get(&self) -> &u8 {  
        self.r  
    }  
}
```

```
// error: does not live long enough!  
let num = 3;  
let r = {  
    let w = RefWrapper { r: &num };  
    w.get()  
};
```

```
impl<'a> RefWrapper<'a> {  
    fn get(&self) -> &'a u8 {  
        self.r  
    }  
}
```

Nicht optimal

Besser

# Beispiel: Digits Iterator

```
struct Digits {  
    s: String,  
    byte_pos: usize,  
}  
  
impl Digits {  
    fn new(s: &str) -> Self {  
        Digits {  
            s: s.to_string(),  
            byte_pos: 0,  
        }  
    }  
}
```

- Ziel:
  - Iterator über Zeichenkette
  - Ignoriert alle nicht-Ziffer-Zeichen
  - Equivalent zu:

```
s.chars()  
    .filter(|c| c.is_digit(10))
```

**Unnötiger Klon!**

Würde auch mit Referenz  
funktionieren!

# Beispiel: Digits Iterator

```
struct Digits<'a> {
    s: &'a str,
}

impl<'a> Digits<'a> {
    fn new(s: &'a str) -> Self {
        Digits {
            s: s,
        }
    }
}

fn main() {
    for d in Digits::new("h4xx0r") {
        println!("{}", d);
    }
}
```

```
impl<'a> Iterator for Digits<'a> {
    type Item = char;
    fn next(&mut self) -> Option<Self::Item> {
        loop {
            let c = match self.s.chars().nth(0) {
                Some(c) => c,
                None => return None,
            };
            let offset = c.len_utf8();
            self.s = &self.s[offset..];

            if c.is_digit(10) {
                return Some(c);
            }
        }
    }
}
```

# Beispiel: Digits Iterator

```
struct Digits<'a> {
    chars: Chars<'a>,
}

impl<'a> Digits<'a> {
    fn new(s: &'a str) -> Self {
        Digits {
            chars: s.chars(),
        }
    }
}

fn main() {
    for d in Digits::new("h4xx0r") {
        println!("{}", d);
    }
}
```

```
impl<'a> Iterator for Digits<'a> {
    type Item = char;
    fn next(&mut self) -> Option<Self::Item> {
        self.chars.find(|c| c.is_digit(10))
    }
}
```

- Nicht mehr manuell **&str** verwalten
- Fertigen **chars()** Iterator benutzen
  - **std::str::Chars**
- LT-Parameter immer außen sichtbar