

12.

Iterator & Closures

Iteratoren

```
trait Iterator {  
    type Item;  
    fn next(&mut self)  
        -> Option<Self::Item>;  
}
```

- **Iterator:**
 - „Kann ein optionales Element liefern“ → **next()**
 - **None** heißt typischerweise „kein Element mehr vorhanden“
 - Oft redet man über „eine Sequenz von Elementen“
 - Besitzt viele *Default Methods* (viele sog. Iterator Adaptoren)
- **DoubleEndedIterator**
 - Kann außerdem „optionales Element von hinten liefern“ → **next_back()**
- **ExactSizeIterator**
 - Kennt Anzahl an Elementen → **len()** und **is_empty()**

Iteratoren \Leftrightarrow Collections

- **IntoIterator**
 - „In Iterator konvertierbar“
 - Wird für **for**-Schleife genutzt
- **FromIterator<A>**
 - „Aus Iterator erstellbar“
- **Extend<A>**
 - „Von Iterator erweiterbar“
 - Fast die gleichen Implementierungen wie **FromIterator<A>**

```
impl<T> IntoIterator for Vec<T> { ... }  
impl<T> IntoIterator for & Vec<T> { ... }  
impl<T> IntoIterator for &mut Vec<T> { ... }
```

```
impl<T> FromIterator<T> for Vec<T> { ... }  
impl FromIterator<char> for String { ... }  
impl FromIterator<&str> for String { ... }  
impl FromIterator<String> for String { ... }
```

```
let mut v = vec![0, 1, 2];  
v.extend(3..9); // v now contains 0 ... 9
```

Iterator für Vec<T>

```
// Not possible! We need the
// current position!
impl<T> Iterator for Vec<T> { ... }

// We need an own type!
struct Iter<T> {
    v: &Vec<T>, // simplified!!!
    pos: usize,
}

let v = vec![1, 3, 5];
let it = Iter { v: &v, pos: 0 };
for x in it { ... }
```

```
impl<T> Iterator for Iter<T> {
    type Item = &T;
    fn next(&mut self)
        -> Option<Self::Item>
    {
        if self.pos == self.v.len() {
            None
        } else {
            self.pos += 1;
            Some(&self.v[self.pos - 1])
        }
    }
}
```

Vereinfacht!

Echte Implementation benötigt
explizite Lifetimes...

Iterator für Vec<T>

```
// We need an own type!  
struct Iter<T> {  
    v: &Vec<T>, // simplified!!!  
    pos: usize,  
}  
  
impl<T> Iterator for Iter<T> { ... }  
  
let v = vec![1, 3, 5];  
for x in &v { ... }
```

Vereinfacht!

Echte Implementation benötigt
explizite Lifetimes...

```
impl<T> IntoIterator for &Vec<T> {  
    type Item = &T;  
    type IntoIter = Iter<T>;  
    fn into_iter(self) -> Self::IntoIter {  
        Iter {  
            v: self,  
            pos: 0,  
        }  
    }  
}
```

```
impl<T> Vec<T> {  
    pub fn iter(&self) -> Iter<T> {  
        self.into_iter()  
    }  
}
```

Iteratoren \Leftrightarrow Collections


Pro Collection C:

- Meist drei **IntoIterator** Implementationen
 - *Immutable Reference* (... **for &C**)
 - *Mutable Reference* (... **for &mut C**)
 - *By Value/mit Ownership* (... **for C**)
- Meist zwei Methoden zum manuellen Aufrufen
 - **fn iter()** (für *immutable references*, wie **(&c).into_iter()**)
 - **fn iter_mut()** (für *mutable references*, wie **(&mut c).into_iter()**)
- Manchmal Methoden für spezielle Iteratoren
 - **str::chars()** und **str::bytes()**
 - **HashMap::keys()**

Syntaxzucker: for-Schleife

```
for <pattern> in <expr> {  
    <block>  
}
```

Explizite Version (fast) nie nötig



```
let mut it = <expr>.into_iter();  
  
while let Some(<pattern>) = it.next() {  
    <block>  
}
```

Iterator Helpermethoden 1

```
/// Consumes the iterator, counting the number  
/// of iterations and returning it.
```

```
fn count(self) -> usize { ... }
```

```
/// Consumes the iterator, returning the last  
/// element.
```

```
fn last(self) -> Option<Self::Item> { ... }
```

```
/// Consumes the n first elements of the iterator, then  
/// returns the `next()` one.
```

```
fn nth(&mut self, n: usize) -> Option<Self::Item> { ... }
```

```
(2..5).count(); // --> 3
```

```
(2..).count(); // will overflow ...  
// ... eventually ;-)
```

```
(2..5).last(); // --> Some(4)
```

```
(2..).nth(7); // --> Some(9)
```


Iterator Helpermethoden 2

```
/// Transforms an iterator into a collection.
fn collect<B>(self) -> B
    where B: FromIterator<Self::Item> { ... }

/// Returns the maximum element of an iterator.
/// Also: min(), sum(), product()
fn max(self) -> Option<Self::Item>
    where Self::Item: Ord { ... }

/// Is equal to other? Also: cmp(), ne(), ...
fn eq<I>(self, other: I) -> bool
    where I: IntoIterator,
           Self::Item: PartialEq<I::Item> { ... }
```

```
// type annotation or turbofish
// (::<>) necessary
let v: Vec<_> = (3..7).collect();
let v: Vec<_> =
    "hi".chars().collect();

(3..7).max(); // 6
(1..101).sum::i32(); // 5050

let v = vec![1, 2, 3];
(1..4).eq(v); // true
```

Iterator Adaptoren 1

- Nehmen Iterator, geben anderen Iterator zurück
- „*Iterator Wrapper*“ kapseln anderen Iterator
 - Speichern „original Iterator“ in sich
 - Verändern Verhalten von `next()`

```
fn take(self, n: usize)
    -> Take<Self> { ... }
```

```
// Will print "4 5"
for i in (4..9).take(2) {
    println!("{}", i);
}
```

```
pub struct Take<I> {
    iter: I,
    n: usize,
}
```

```
impl<I> Iterator for Take<I>
    where I: Iterator
{
    type Item = I::Item;
    fn next(&mut self)
        -> Option<Self::Item>
    {
        if self.n != 0 {
            self.n -= 1;
            self.iter.next()
        } else {
            None
        }
    }
}
```

Iterator Adaptoren 2

- `skip(n)`: Überspringt die ersten `n` Elemente

```
// yields: 2, 3, 4  
(0..5).skip(2);
```

- `enumerate()`: Fügt Index zu jedem Element hinzu

```
let v = vec!['a', 'b', 'c'];  
let it = v.into_iter().enumerate();  
  
// yields: (0, 'a'), (1, 'b'), (2, 'c')  
for (index, c) in it { ... }
```

Alles kombinierbar!

```
// yields 5, 6, 7, 8, 9  
(0..  
    .take(10)  
    .skip(5)
```

Iterator Adaptoren 3

- `cycle()`: Wiederholt Iterator für immer

```
// yields: 1, 2, 3, 1, 2, 3, 1, 2, ...  
(1..4).cycle();
```

```
fn cycle(self) -> Cycle<Self>  
  where Self: Clone { ... }
```

- `rev()`: Dreht Iterator um

```
// yields: 3, 2, 1  
(1..4).rev();
```

```
fn rev(self) -> Rev<Self>  
  where Self: DoubleEndedIterator { ... }
```

- `cloned()`: Wandelt Referenzen durch Klonen in Werte um

```
vec![1, 2, 3]  
  .iter()    // would yield: &1, &2, &3  
  .cloned() // yields: 1, 2, 3
```

Iterator Adaptoren 4

- **chain()**: Hängt zwei Iteratoren hintereinander

```
// yields: 1, 2, 3, 7, 8, 9  
(1..4).chain(vec![7, 8, 9]);
```

```
fn chain<U>(self, other: U) -> ...  
  where U: IntoIterator<Item=Self::Item>
```

- **zip()**: Paart Elemente zweier Iteratoren im Gleichschritt

```
let it = (7..10)  
  .zip(vec!['a', 'b', 'c']);  
  
// yields (7, 'a'), (8, 'b'), (9, 'c')  
for (x, y) in it { ... }
```

```
fn zip<U>(self, other: U)  
  -> Zip<Self, U::IntoIter>  
  where U: IntoIterator { ... }
```

Zuerst: Funktionen als Variable

```
let greeter = match party.kind {  
    PartyKind::Formal => hello,  
    PartyKind::Informal => wassup,  
};
```

```
greeter("Peter");  
greeter("Heike");  
greeter("Jörg");
```

```
fn hello(name: &str) {  
    println!("Hello {}!", name);  
}  
  
fn wassup(name: &str) {  
    println!("Wassuuuuup {}!", name);  
}
```

- *Funktionspointer*: Adresse von Funktion
- Kann ohne Probleme in Variable gespeichert werden
- Verhalten/Algorithmus kann weitergereicht werden

Zuerst: Funktionen als Variable

```
greet_all_guests(wassup);  
if people_think_i_am_crazy() {  
    greet_all_guests(hello);  
}
```

```
fn greet_all_guests(greeter: fn(&str)) {  
    greeter("Peter");  
    greeter("Heike");  
    greeter("Jörg");  
}
```

Funktionspointer
Typ

- Libraries können ein Gerüst für beliebigen Algorithmus bieten
- In Java: Ähnliches Verhalten via **Runnable**
- **Beispiel:** Sortieren mit Comparator

