Sie sind da

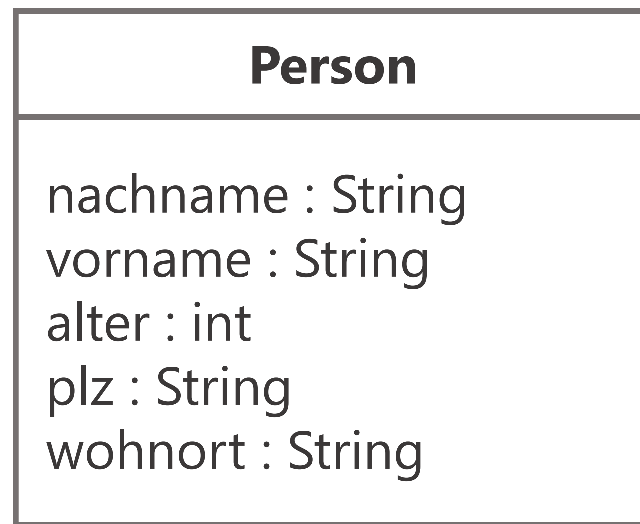
Lambda Expressions in Java 8

Rolf Borst

Lambda Expressions sind cool

Aus dem Internet

Stimmt das?



List<Person> personen

Aufgabe

Es sollen alle Personen über 30 Jahre ermittelt werden (Ü-30 Party).
Später soll auch nach Wohnorten, Namen, … gesucht werden können.

Wie machen wir das bisher?

Möglichkeit: Design Patterns

filtern(personen, **new FilterAlter(30)**)

filtern(personen, **new FilterName("Schuster")**)

filtern(personen, **new FilterWohnort("Stuttgart")**)

```
public interface Filter<T> {
    boolean istErfuellt(T e);
}
```

```
public <T> List<T> filtern(List<T> liste, Filter<T> filter) {
        List<T> treffer = new ArrayList<T>();
        for (T e : liste) {
                if (filter.istErfuellt(e)) {
                        treffer.add(e);
                }
        }
        return treffer;
}
```

Anonymous Inner Classes

```
List<Person> treffer = filtern(personen, new Filter<Person>() {
    @Override
    public boolean istErfuellt(Person p) {
        return p.getAlter() >= 30;
    }
});
```

Was wäre denn richtig cool?

```
filtern(personen, p.getAlter() >= 30);
```

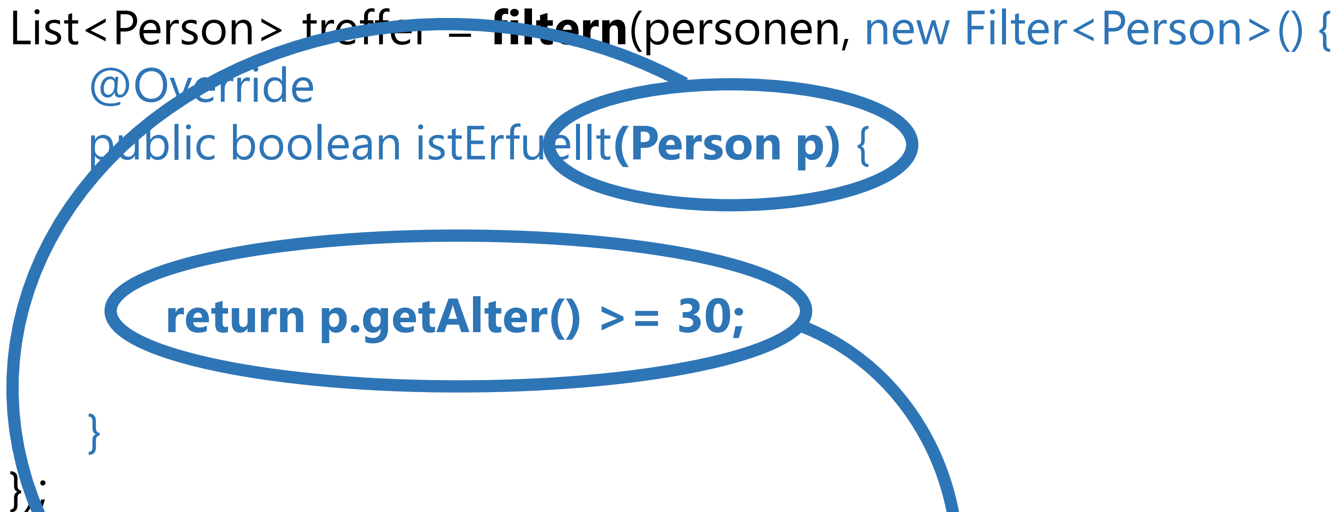
Was wäre denn richtig cool?

```
filtern(personen, p.getAlter() >= 30);
```

Lambda Expressions

```
public interface Filter<T> { … }

List<Person> treffer = filtern(personen, new Filter<Person>() {
    @Override
    public boolean istErfuellt(Person p) {


        return p.getAlter() >= 30;

    }
});
```

```
public interface Filter<T> { … }

List<Person> treffer = filtern(personen, new Filter<Person>() {
    @Override
    public boolean istErfuellt(Person p) {

        return p.getAlter() >= 30;

    }
});
```



```
filtern(personen, (Person p) -> { return p.getAlter() >= 30; } );
```

```
filtern(personen, p -> p.getAlter() >= 30 );
```

```
public void verarbeiten(List<Person> personen, int alter) {
    ...

    treffer = filtern(personen, p -> p.getAlter() >= alter);
}
```

Erzeuger

Funktionalität

```
public <T> List<T> filtern(List<T> liste, Filter<T> filter) {
    List<T> treffer = new ArrayList<T>();
    for (T e : liste) {
        if (filter.istErfuellt(e)) {
            treffer.add(e);
        }
    }
    return treffer;
}
```

Aufrufer

```
filtern(personen, p -> p.getAlter() >= 30 );
```

Methodensignatur: ...

```
filtern(List<T> liste, Filter<T> filter) { ... }
```

```
@FunctionalInterface
public interface Filter<T> {

  boolean istErfuellt(T e);

  default boolean nichtErfuellt(T e) {
    return !istErfuellt(e);
  }

}
```

Interface mit genau **einer** abstrakten Methode

Functional Interface

Interface darf **default-Methoden** besitzen

(Person p1, Person p2) -> ...

(p1, p2) -> ... (Typ ermittelbar)

p -> ... (Nur ein Argument)

() -> ... (Kein Argument)

filtern(personen, **p ->** p.getAlter() >= 30);

```
filtern(personen, p -> p.getAlter() >= 30 );
```

Rechte Seite

```
... -> {  int min= 30;
          return p.getAlter() >= min;
       }
```

(Ein Statement) `... -> p.getAlter() >= 30`

Auf was kann ich eigentlich in der Lambda Expression zugreifen?

Auf alles...

```
public class DatenService {

    private int minalter;

    ...

    public void verarbeiten(List<Person> personen, int alterVon) {
        int alter = einlesenAlterVonGUI();
        ...
        List<Person> treffer = filtern(personen, p -> p.getAlter() >= alter);
    }

    public int ermittelnMindestalter() {
        return ...;
    }

}
```

… mit Einschränkungen

```
public class DatenService {

    private int minalter;

    ...

    public void verarbeiten(List<Person> personen, int alterVon) {
        int alter = einlesenAlterVonGUI();
        ...
        List<Person> treffer = filtern(personen, p -> p.getAlter() >= alter);
    }

    public int ermittelnMindestalter() {
        return ...;
    }

}
```

Müssen **final** oder **effektiv final** sein

Das geht nicht…

```
for (int alter = 20; alter <= 70; alter++) {
    List<Person> treffer = filtern(personen, p -> p.getAlter() == alter );
    anzeigenTreffer(treffer);
}
```

Das geht nicht...

```
for (int alter = 20; alter <= 70; alter++) {
    List<Person> treffer = filtern(personen, p -> p.getAlter() == alter );
    anzeigenTreffer(treffer);
}
```

... das dagegen schon

```
for (int alter = 20; alter <= 70; alter++) {
    int suchalter = alter;
    List<Person> treffer = filtern(personen, p -> p.getAlter() == suchalter );
    anzeigenTreffer(treffer);
}
```

```
public int berechnenAlter(Person p) throws AlterException {
    ...
}

public void verarbeiten(List<Person> personen) throws AlterException {
    treffer = filtern(personen, p -> berechnenAlter(p) >= 30);
}
```

Das geht **nicht**

```
public int berechnenAlter(Person p) throws AlterException {
    ...
}

public void verarbeiten(List<Person> personen) throws AlterException {
    treffer = filtern(personen, p -> berechnenAlter(p) >= 30);
}
```

Das geht **nicht**

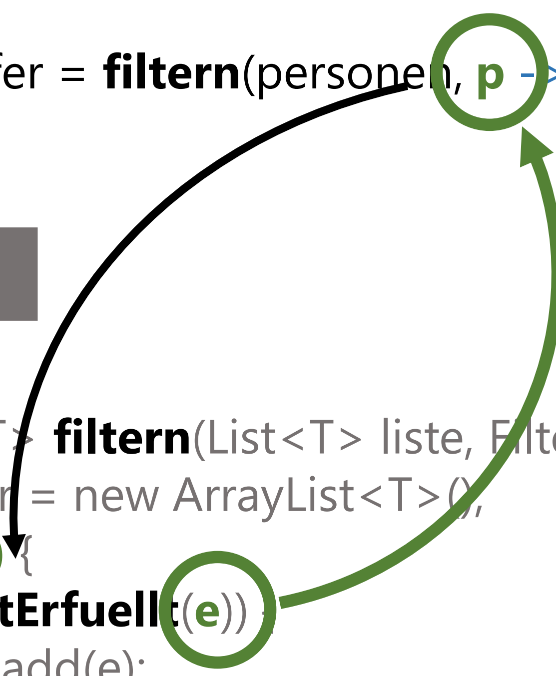
Interface definiert **Vertrag**

```
public interface Filter<T> {
    boolean istErfuellt(T e);
}
```

```
public int berechnenAlter(Person p) throws AlterException {
      ...
}

public void verarbeiten(List<Person> personen) {
      treffer = filtern(personen, p -> {
                        try {
                              berechnenAlter(p) >= 30);
                        } catch (AlterException) {
                              return false;
                        }
                  }
}
```

Eine mögliche Lösung

```
public void verarbeiten(List<Person> personen, int alter) {
    ...

    treffer = filtern(personen, p -> p.getAlter() >= alter);
}
```



Erzeuger

Funktionalität

```
public <T> List<T> filtern(List<T> liste, Filter<T> filter) {
    List<T> treffer = new ArrayList<T>();
    for (T e : liste) {
        if (filter.istErfuellt(e)) {
            treffer.add(e);
        }
    }
    return treffer;
}
```

Aufrufer

Design Pattern

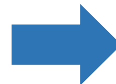
1 Unser Filter-Interface...

```
public interface Filter<T> {
    boolean istErfuellt(T e);
}
```

2 Unsere filtern-Methode...

```
public List<Person> filtern(...) {
    ...
}
```

3 Unsere Filter-Klassen...



Lambda Expressions

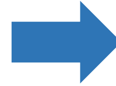
AlterFilter, WohnortFilter, NameFilter, ...

Anonymous Inner Classes

Design Pattern

1 Unser Filter-Interface...

```
public interface Filter<T> {
    boolean istErfuellt(T e);
}
```



... gibt es schon

```
public interface Predicate<T> {
    boolean test(T e);
}
```

Package **java.util.function**

Function, Consumer, Supplier, ...

2 Unsere filtern-Methode...

```
public List<Person> filtern(...) {
    ...
}
```

3 Unsere Filter-Klassen...

AlterFilter, WohnortFilter, NameFilter, ...
Anonymous Inner Classes

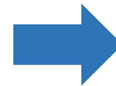
Design Pattern

1 Unser Filter-Interface...

```
public interface Filter<T> {
    boolean istErfuellt(T e);
}
```

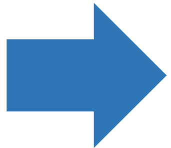
2 Unsere filtern-Methode...

```
public List<Person> filtern(...) {
    ...
}
```



... gibt es schon

Erweiterungen bei den **Collections**
Streams



3 Unsere Filter-Klassen...

AlterFilter, WohnortFilter, NameFilter, ...
Anonymous Inner Classes

0..n Intermediate Operations

Eintrittskarte

```
List<Person> personen = ... ;

List<Person> treffer = personen.stream()            → Stream<Person>
        .filter(p -> p.getAlter() >= 30)            → Stream<Person>
        .collect(Collectors.toList());              → List<Person>
```

Terminal Operation

0..n Intermediate Operations

filter
map
distinct
sorted
limit
skip
...

```
List<Person> personen = ... ;

List<Person> treffer = personen.stream()
        .filter(p -> p.getAlter() >= 30)
        .collect(Collectors.toList());
```

Terminal Operation

forEach
reduce
min
max
count
findFirst
anyMatch
allMatch

collect
toList
groupingBy
joining

Aufgabe

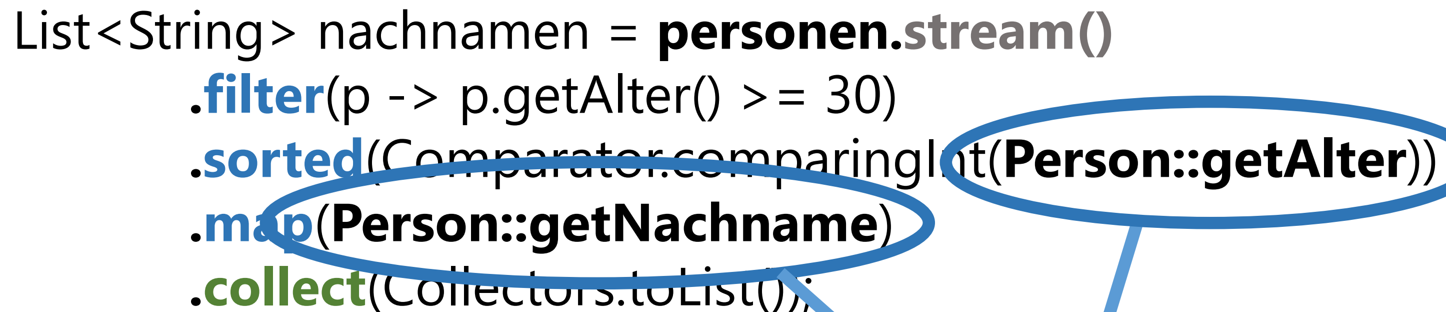
Es soll eine **Namensliste** der Personen **über 30 Jahre** erstellt werden.
Die Liste soll nach dem **Alter sortiert** sein.

```
List<Person> personen = ... ;

List<String> nachnamen = personen.stream()
        .filter(p -> p.getAlter() >= 30)
        .sorted(Comparator.comparingInt(Person::getAlter))
        .map(Person::getNachname)
        .collect(Collectors.toList());
```

Aufgabe

Es soll eine **Namensliste** der Personen **über 30 Jahre** erstellt werden.
Die Liste soll nach dem **Alter sortiert** sein.

```
List<Person> personen = ... ;
```



Methodenreferenzen

```
public interface Suche {
    boolean pruefen(Person p, String begriff);
}
```

```
public class PersonHelper {

    public static boolean ueberpruefen(Person p, String begriff) {
        ...
    }

}
```

suchen(personen, **PersonHelper::ueberpruefen**)

Alternative für: suchen(personen, **(p, b) -> PersonHelper.ueberpruefen(p, b)**)

```
public interface Suche {
    boolean pruefen(Person p, String begriff);
}
```

```
public class PersonPruefer {

    public boolean ueberpruefen(Person p, String begriff) {
        ...
    }

}
```

```
PersonPruefer pruefer = new PersonPruefer();
suchen( personen, pruefer::ueberpruefen )
```

Alternative für: `suchen( personen, (p, b) -> pruefer.ueberpruefen(p, b) )`

```
public interface Suche {
    boolean pruefen(Person p, String begriff);
}
```

```
public class Person {

    public boolean ueberpruefen(String begriff) {
        ...
    }

}
```

suchen(personen, **Person::ueberpruefen**)

Alternative für: suchen(personen, **(p, b) -> p.ueberpruefen(b)**)

```
public interface Erzeuger {
    Person erzeugen(String nachname, String vorname);
}
```

```
public class Person {

    public Person (String nachname, String vorname) {
        ...
    }

}
```

```
erweitern( personen, Person::new )
```

Alternative für: `erweitern( personen, (n, v) -> new Person(n, v) )`

```
List<Person> personen = ... ;

List<String> nachnamen = personen.stream()
        .filter(p -> p.getAlter() >= 30)
        .sorted(Comparator.comparingInt(Person::getAlter))
        .map(Person::getNachname)
        .collect(Collectors.toList());
```

Mehr Beispiele

```
Button speichernButton = new Button("Speichern");
speichernButton.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
                speichern();
        }
});
```

...mit Lambda Expressions

```
Button speichernButton = new Button("Speichern");
speichernButton.setOnAction(e -> speichern());
```

oder

```
Button speichernButton = new Button("Speichern");
speichernButton.setOnAction(this::speichern);
```

```
logger.debug( ermittelnDebugInfo() );
...

private String ermittelnDebugInfo() {
    // dauert sehr lange
}
```

```
if (logger.isDebugEnabled()) {
    logger.debug( ermittelnDebugInfo() );
}
...

private String ermittelnDebugInfo() {
    // dauert sehr lange
}
```

```
logger.debug( () -> ermittelnDebugInfo() );
```

oder

```
logger.debug( this::ermittelnDebugInfo );
```

Im Logging-Framework:

```
public void debug(MessageSupplier message)
    if (isDebugEnabled()) {
        writeLog( message.get() );      → ermittelnDebugInfo()
    }
}
```

```
try (Connection conn = getConnection()) {

    String sql = "SELECT artikelnr, bezeichnung FROM artikel  WHERE gruppe = ?";

    try (PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString(1, gruppe);

        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            Artikel artikel = new Artikel();
            artikel.setArtikelnr(rs.getInt("artikelnr"));
            artikel.setBezeichnung(rs.getString("bezeichnung"));
            artikelliste.add(artikel);
        }
        rs.close();

    }

}
```

```
List<Person> personen = datenErmitteln(
    () -> getConnection(),
    () -> "SELECT artikelnr, bezeichnung FROM artikel WHERE gruppe = ?",
    stmt -> stmt.setString(1, gruppe),
    rs -> { Artikel artikel = new Artikel();
            artikel.setArtikelnr(rs.getInt("artikelnr"));
            artikel.setBezeichnung(rs.getString("bezeichnung"));
            return artikel; }
);
```

Fazit

Lambda Expression **sind cool**

Sie sind da

Lambda Expressions in Java 8

Danke