Programmieren in Java

Vorlesung 09: Funktionen höherer Ordnung

Prof. Dr. Peter Thiemann, Manuel Geffken

Albert-Ludwigs-Universität Freiburg, Germany

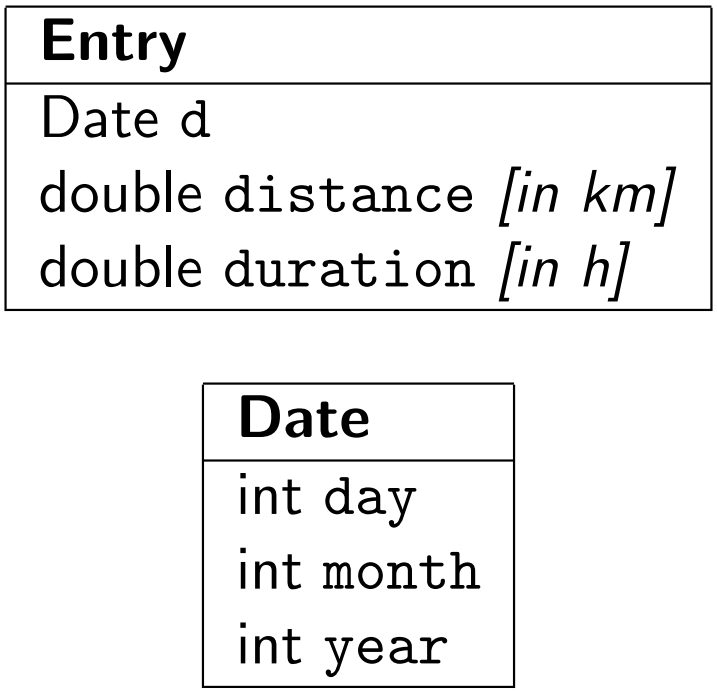
SS 2015

Inhalt

Erinnerung: Transformationen mit generischen Klassen und Interfaces

Wiederholung: Eintrag im ActivityLog

Klassendiagramm



Wiederholung: ActivityLog

Implementierung

```
1  // Eintrag in einem ActivityLog
2  public class Entry {
5      Date d;
6      double distance; // in km
7      double duration; // in h
22 }
```

```
1  public class ActivityLog {
2      private Collection<Entry> activities;
32 }
```

Wiederholung: Collections transformieren

Änderungsinterface

```
1  // transform an X into a U
2  public interface ITransform<X,U> {
3      public U transform (X x);
4  }
```

Statische generische Methode

```
1  public class Transform {
4      public static <X,Y> Collection<Y>
5          map(Collection<X> source, ITransform<X,Y> fun) {
6          Collection<Y> result = new ArrayList<Y>();
7          for (X item : source) {
8              result.add(fun.transform(item));
9          }
10         return result;
11     }
22 }
```

Wiederholung: Geschwindigkeiten ausrechnen

```
1 public class Speed implements ITransform<Entry,Double> {
2     public Double transform (Entry e) {
3         return e.getDistance() / e.getDuration();
4     }
5 }
```

Verwendung

```
1     Collection<Entry> log = ...;
2     Collection<Double> speeds = Transform.map(log, new Speed());
```

map als Instanz-Methode

map als Instanz-Methode formulieren

Ausgangspunkt

```
1  public static <X,Y> Collection<Y>
2  map(Collection<? extends X> src, ITransform<X,Y> fun) {...}
```

Aufruf:

```
1  Collection<CommentedEntry> cce = new ArrayList<CommentedEntry>();
2  Transform.map(cce, new Speed());
```

Noch Verbesserungspotential

- Nicht "objektorientiert"
  - Umständlicher Aufruf der statischen Methode
  - Schwer für den Programmierer herauszufinden, ob map-Methode existiert und in welcher Klasse diese definiert ist.
- Besser: map als Instanzmethode (nicht statisch)
- Wir möchten folgendes schreiben können: cce.map(**new** Speed());

map als Instanz-Methode formulieren

Zu transformierende Collection wird implizit als **this** übergeben

Erster Versuch

```
1 interface Collection<T> {
2   public <R> Collection<T> map(ITransform<T, R> fun);
3 }
```

- Mit diesem Typ **funktioniert** das Beispiel der Transformation von Collection<CommentedEntry> mit **new** Speed() **nicht mehr**.

Das alte Problem

Der folgende Code liefert einen Typfehler:

```
1 Collection<CommentedEntry> cce = ...
2 Collection<Double> cd = cce.map(new Speed()); // type error
```

map als Instanz-Methode formulieren II

Dualität der Typschranken ausnutzen

Zweiter Versuch mit Typschranken:

```
1 interface Collection<T> {
2   public <T,R> Collection<R>
3     map(ITransform<? super T, R> fun) {...}
4 }
```

- Mit diesem Typ funktioniert das Beispiel (Transformation von Collection<CommentedEntry> mit **new** Speed()) wieder!

Wir haben unser Ziel erreicht.

```
1   Collection<CommentedEntry> cce = ...
2   Collection<Double> cd = cce.map(new Speed()); // OK
```

map als Instanz-Methode formulieren II

Dualität der Typschranken ausnutzen

Zweiter Versuch mit Typschranken:

```
1 interface Collection<T> {
2   public <T,R> Collection<R>
3     map(ITransform<? super T, R> fun) {...}
4 }
```

- Mit diesem Typ funktioniert das Beispiel (Transformation von Collection<CommentedEntry> mit **new** Speed()) wieder!

Wir haben unser Ziel erreicht.

```
1   Collection<CommentedEntry> cce = ...
2   Collection<Double> cd = cce.map(new Speed()); // OK
```

Können wir alle Programme mit der Instanz-Methode schreiben, die wir zuvor mit der statischen Methode geschrieben haben?

c.map(fun) kompiliert $\iff$ Transform.map(c, fun) kompiliert?

map als Instanz-Methode formulieren II

Warum funktioniert der zweite Versuch?

Dualität der Typschranken ausnutzen.

```
1  public static <T,R> Collection<R>
2    map(Collection<? extends T> src, ITransform<T, R> fun) {...}
```

is äquivalent zu

```
1  public static <T,R> Collection<R>
2    map(Collection<T> src, ITransform<? super T, R> fun) {...}
```

Sei X_1 das Typargument der Collection und X_2 das Typargument des konkreten ITransform

```
1 Collection<X1> col = ...
2 Collection<Double> cd = Transform.map(col, new ITransform<X2> { ... });
```

Es gilt X_1 **extends** $X_2 \iff X_2$ **super** X_1.

map als Instanz-Methode formulieren II

Warum funktioniert der zweite Versuch?

Dualität der Typschranken ausnutzen.

```
1  public static <T,R> Collection<R>
2    map(Collection<? extends T> src, ITransform<T, R> fun) {...}
```

is äquivalent zu

```
1  public static <T,R> Collection<R>
2    map(Collection<T> src, ITransform<? super T, R> fun) {...}
```

Aufrufe unserer map-Instanz-Methode kompilieren genau dann, wenn entsprechende Aufrufe der statischem map-Methode kompilieren

c.map(fun) kompiliert $\iff$ Transform.map(c, fun) kompiliert ✓

Anwendbarkeit von map

Angenommen wir haben eine Methode

```
1  printCollection(Collection<Object> c) { ... }
```

in einer externen grafischen Bibliothek, die die Collection in geeigneter Weise ausgibt. Jedes einzelne Objekt wird dabei mittels seiner toString-Methode ausgegeben.
Wir würden also gerne folgenden Code schreiben

```
1  // type mismatch: cannot convert from Collection<Double> to Collection<Object>
2  printCollection(cce.map(new Speed())); // type error
```

Das Problem ist wieder, dass

Collection<Double> **extends** Collection<Object>

in Java **nicht** gilt, obwohl Double **extends** Object gilt.

Lösung: Noch besserer Typ für map

Weniger Vorgaben Rückgabetyp des konkreten ITransform

```
1 interface Collection<T> {
2   public <R> Collection<R> map(ITransform<? super T, ? extends R> fun);
3 }
```

- Nun funktioniert printCollection(cce.map(**new** Speed())); für T=CommentedEntry, R=Object obwohl Speed **implements** ITransform<Entry,Double>
- Die Implementierung von map bleibt gleich!

ITransform<T,R> in der Java-Bibliothek

Vordefiniert als java.util.function.Function

```
1  interface Function<T,R> {
2    R apply(T t);
3    ... // weitere default-Methoden
4  }
```

map in der Java-Bibliothek

In Klasse `java.util.stream.Stream`

```
1  interface Stream<T> {
2    <R> Stream<T> map(Function<? super T, ? extends R> fun);
3  }
```

Anwendung auf Collection

```
1  Collection<CommentedEntry> cce = ...
2  Collection<Double> cd = cce.stream().map(new Speed()); // type error
```

Zur Erinnerung

```
1  public class Speed implements ITransform<Entry,Double> { ... }
```

Speed ist keine Function!

Anwendung von `map` zur Transformation von Log-Einträgen in Geschwindigkeiten

Mit anonymer Klasse, die Function implementiert

```
1 Collection<CommentedEntry> cce = ...
2 Collection<Double> cd = cce.stream().map(new Function<Entry, Double>() {
3   @Override
4   public Double apply(Entry e) {
5     return e.getDistance() / e.getDuration
6   }
7 });
```

Problem

Trotz Verwendung von anonymer Klasse großer syntaktischer Overhead bei einfachen Funktionen

Lösung

Lambda-Ausdrücke

Lambda-Ausdrücke

Anwendung von `map` mit Lambda-Ausdruck

Elegantere Lösung mit Lambda-Ausdruck

```
1 Collection<Double> cd = cce.stream().map(
2   (Entry: e) -> {return e.getDistance() / e.getDuration();});
```

Allgemeine Syntax von Lambda-Ausdrücken:

- Liste formaler Parameter $(x_1, \ldots, x_n)$. Ein einzelner Parameter kann ungeklammert geschrieben werden. Typen können inferiert werden.
- Pfeil ->
- Ein Body, der aus einem einzelnen Ausdruck oder einem Statement Block besteht.

D.h. noch kompakter kann man schreiben

```
1 Collection<Double> cd = cce.stream().map(
2   e -> e.getDistance() / e.getDuration());
```

Lambda-Ausdrücke

- Kompakte Syntax um Verhalten (Code) als Wert (Daten) anzugeben
- hilfreich beim Aufruf von Funktionen höherer Ordnung
- Lambda-Ausdrücke vermeiden
  1. das implementierte Interface zu referenzieren
  2. meist Typen anzugeben
  3. die implementierte Methode zu referenzieren

```
1 Collection<Double> cd = cce.stream().map(
2   e -> e.getDistance() / e.getDuration());
```

Lambda-Ausdrücke

- Kompakte Syntax um Verhalten (Code) als Wert (Daten) anzugeben
- hilfreich beim Aufruf von Funktionen höherer Ordnung
- Lambda-Ausdrücke vermeiden
  1. das implementierte Interface zu referenzieren (aus Kontext inferiert)
  2. meist Typen anzugeben (in der Regel aus Kontext inferierbar)
  3. die implementierte Methode zu referenzieren (Wie geht das?)

```
1 Collection<Double> cd = cce.stream().map(
2   e -> e.getDistance() / e.getDuration());
```

Lambda-Ausdrücke

- Kompakte Syntax um Verhalten (Code) als Wert (Daten) anzugeben
- hilfreich beim Aufruf von Funktionen höherer Ordnung
- Lambda-Ausdrücke vermeiden
  1. das implementierte Interface zu referenzieren (aus Kontext inferiert)
  2. meist Typen anzugeben (in der Regel aus Kontext inferierbar)
  3. die implementierte Methode zu referenzieren (→ Functional Interfaces.)

Wo kann man Lambda-Ausdrücke verwenden?

Implementieren von **Interfaces mit nur einer anonymen Methode**

- Solche Interfaces heißen *Functional Interfaces*
- Der Compiler inferiert welche Methode der λ-Ausdruck implementiert
- Hinweis: **default/static**-Methoden sind **nicht** anonym!

Wichtige *Functional Interfaces*

Im Paket `java.util.function` bzw. `java.util`

Java-Bibliothek	**Java 2015 Kurs**
Function<T,R>	ITransform<T,R>
Predicate<T>	ISelect<T>
BiFunction<T,U,R>	
Comparator<T>	
⋮	

Beispiele

```
interface Function<T,R> {
  R apply(T t);
  ... // default/static−Methoden
}
```

```
interface Predicate<T> {
  boolean test(T t);
  ... // default/static−Methoden
}
```

Wichtige Funktionen höherer Ordnung

Hier werden Functional Interfaces verwendet

Java-Bibliothek: `java.util.stream.Stream`

```
1 interface Stream<T> {
2   <R> Stream<R> map(Function<? super T, ? extends R> fun);
3   Stream<T> filter(Predicate<? super T> predicate)
4   ...
5 }
```

Java 2015 Kurs

```
1 public class Transform {
2   public static <T,R> Collection<R>
3     map(Collection<T> source, ITransform<? super T, ? super R> fun)
4 }
5 public class Filter {
6   public static <T> Collection<R>
7     filter(Collection<T>, ISelect<T> pred);
8 }
```

Functional Interfaces

Beispiele für Verwendung

```
1 Collection<Double> cd = cce.stream().map(e -> e.getDistance() / e.getDuration());
2
3 Collection<Entry> moreThan10 = cce.stream().filter(e -> e.getDistance() > 10);
```

Functional Interfaces

Beispiele für Verwendung

```
1 Collection<Double> cd = cce.stream().map(e -> e.getDistance() / e.getDuration());
2
3 Collection<Entry> moreThan10 = cce.stream().filter(e -> e.getDistance() > 10);
```

Beispiel „Komposition“

```
1 Collection<Double> cd = cce.stream().
2   filter(e -> e.getDistance() > 10).
3     map(e -> e.getDistance() / e.getDuration());
```

Filtere die Einträge heraus, deren Distanz größer als 10 km ist und transformiere diese jeweils zu ihrer Geschwindigkeit.

Weitere Funktionen höherer Ordnung

Java-Bibliothek: `java.util.stream.Stream`

```
1 interface Stream<T> {
2   <R> Stream<R> map(Function<? super T, ? extends R> fun);
3   Stream<T> filter(Predicate<? super T> predicate);
4   boolean allMatch(Predicate<? super T> predicate);
5   Optional<T> max(Comparator<? super T> comparator);
6   Optional<T> min(Comparator<? super T> comparator);
7   ...
8 }
```

Weitere Funktionen höherer Ordnung

Java-Bibliothek: `java.util.stream.Stream`

```
1 interface Stream<T> {
2   <R> Stream<R> map(Function<? super T, ? extends R> fun);
3   Stream<T> filter(Predicate<? super T> predicate);
4   boolean allMatch(Predicate<? super T> predicate);
5   Optional<T> max(Comparator<? super T> comparator);
6   Optional<T> min(Comparator<? super T> comparator);
7   ...
8 }
```

Frage

Welches Muster haben die Typschranken der Functional Interfaces?

Varianz von Funktionen

Zentrale Frage

Wann gilt $f : T_1 \to R_1$ **extends** $g : T_2 \to R_2$?

Varianz von Funktionen

Zentrale Frage

Wann gilt $f : T_1 \to R_1$ **extends** $g : T_2 \to R_2$?

Antwort

Genau dann wenn

- T_2 **extends** T_1 und
- R_1 **extends** R_2

gilt. Man sagt eine Funktion ist kontravariant im Argumenttyp und kovariant im Rückgabetyp.

Robustheitsprinzip Es ist sicher eine Funktion f anstatt einer Funktion g einzusetzen wenn f einen allgemeineren Argumenttyp und einen spezielleren Rückgabetyp hat als g.

Varianz von Funktionen II

Achtung: Java-Generics sind invariant

In Java stehen die Typen Function<Entry, Double> und Function<CommentedEntry, Object> in keiner Subtyp-Beziehung, obwohl CommentedEntry <: Entry und Double <: Object!

Varianz von Funktionen II

Achtung: Java-Generics sind invariant

In Java stehen die Typen Function<Entry, Double> und Function<CommentedEntry, Object> in keiner Subtyp-Beziehung, obwohl CommentedEntry <: Entry und Double <: Object!

Daher

Typschranken für Ko- und Kontravarianz bei Benutzung des Functional Interface

```
1 interface Stream<T> {
2   <R> Stream<R> map(Function<? super T, ? extends R> fun);
3   Stream<T> filter(Predicate<? super T> predicate);
4   boolean allMatch(Predicate<? super T> predicate);
5   Optional<T> max(Comparator<? super T> comparator);
6   Optional<T> min(Comparator<? super T> comparator);
7    ...
8 }
```