Programmieren in Java

Vorlesung 08: Generics II

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

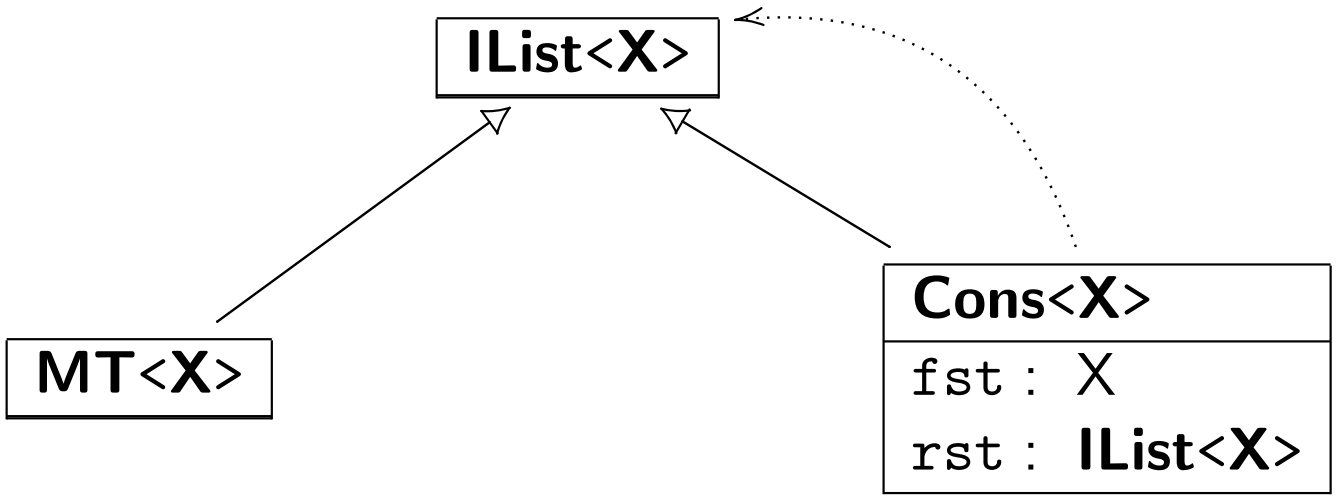
Inhalt

Erinnerung: Generische Klassen und Interfaces

Generics

- *Generische Klassen, Interfaces und Methoden* erlauben die Abstraktion von den konkreten Typen der Objekte, die in Instanzvariablen und lokalen Variablen gespeichert werden oder als Parameter übergeben werden.
- Hauptverwendungsbereiche:
  - Containerklassen (Collections)
  - Abstraktion eines Deklarationsmusters

Generische Listen



- **IList<X>** ist ein *generisches Interface*
- **MT<X>** und **Cons<X>** sind *generische Klassen*
- **X** ist dabei eine *Typvariable*
- **X** steht für einen beliebigen Referenztyp (Klassen- oder Interfacetyp), **nicht** für einen primitiven Typ

Implementierung: Generische Listen

```
1  // Listen mit beliebigen Elementen
2  interface IList<X> {
16 }
```

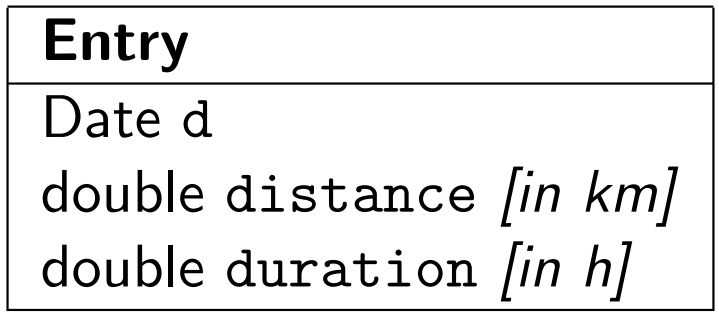
```
1  // Variante leere Liste
2  class MT<X> implements IList<X> {
3      public MT() {}
16 }
```

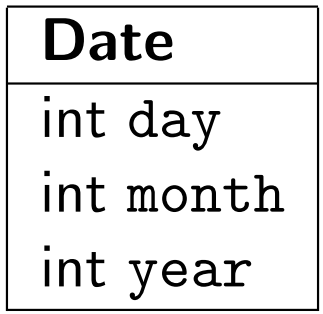
```
1  // Variante nicht-leere Liste
2  class Cons<X> implements IList<X> {
3      private X fst;
4      private IList<X> rst;
5
6      public Cons (X fst, IList<X> rst) {
7          this.fst = fst;
8          this.rst = rst;
9      }
27 }
```

Generische Suche

Vorspiel: Eintrag im ActivityLog

Klassendiagramm





Eintrag im 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 }
```

Generische Auswahl

Filtere aus einem **ActivityLog** *diejenigen Einträge aus, die ein bestimmtes Auswahlkriterium erfüllen.*

Beispiele

- Finde alle Aktivitäten von mehr als 10km Länge.
- Finde alle Aktivitäten im Juni 2003.
- ...

Generische Auswahl

Funktional

Alter Ansatz

Entwickle Methoden in **ActivityLog**

- `Collection<Entry> distanceLongerThan (double length);`
- `Collection<Entry> inMonth (int month, int year);`
- `...`

denen allen das Durchlaufen der Collection und das Zusammenstellen des Ergebnisses gemeinsam ist.

Generische Auswahl

Funktional

Alter Ansatz

Entwickle Methoden in **ActivityLog**

- `Collection<Entry> distanceLongerThan (double length);`
- `Collection<Entry> inMonth (int month, int year);`
- `...`

denen allen das Durchlaufen der Collection und das Zusammenstellen des Ergebnisses gemeinsam ist.

Generischer Ansatz

Entwickle *eine* Methode mit dieser Funktionalität und parametrisiere sie so, dass alle anderen Methoden Spezialfälle davon werden.

Erinnerung: Alter Ansatz

```
5     public Collection<Entry> distanceLongerThan (double length) {
6         Collection<Entry> result = new ArrayList<Entry>();
7         for (Entry e : this.activities) {
8             if (e.distanceLongerThan(length)) {
9                 result.add(e);
10            }
11        }
12        return result;
13    }
```

```
1  // Eintrag in einem ActivityLog
2  public class Entry {
17        public boolean distanceLongerThan(double length) {
18            return distance >= length;
19        }
22 }
```

Neu: Generischer Ansatz

Generische Auswahl

- Definiere das Auswahlkriterium durch ein separates Interface **ISelect**, welches von Elementtypen erfüllt sein soll.
- Dieses Interface muss über den Elementtypen parametrisiert sein:

```
1 // generische Auswahl
2 interface ISelect<X> {
3       // ist obj das Gesuchte?
4       public boolean selected (X obj);
5 }
```

Entwurfsmuster *Strategy*

- Suche mit abstrakter Selektion
- Instantiiert durch konkrete Selektionen

Implementierung der generischen Auswahl

. . . durch Methode `filter` in ActivityLog

```
16    public Collection<Entry> filter (ISelect<Entry> pred) {
17        Collection<Entry> result = new ArrayList<Entry>();
18        for (Entry e : this.activities) {
19            if (pred.selected(e)) {
20                result.add(e);
21            }
22        }
23        return result;
24    }
```

- Auswahlmethode filter parametrisiert über Auswahlkriterium ISelect<Entry>
- Anwendung des Auswahlkriteriums durch pred.selected(e)

Beispiel: Implementierung des Auswahlkriteriums

```
1  // teste ob ein Entry eine längere Entfernung enthält
2  public class DistanceLongerThan implements ISelect<Entry> {
3      private final double limit;
4      public DistanceLongerThan (double limit) {
5          this.limit = limit;
6      }
7
8      public boolean selected (Entry e) {
9          return e.distanceLongerThan(this.limit);
10     }
11 }
```

Beispiel: Implementierung des Auswahlkriteriums

```
1  // teste ob ein Entry in einem bestimmten Monat liegt
2  public class EntryInMonth implements ISelect<Entry> {
3      private final ISelect<Date> selectdate;
4      public EntryInMonth (int month, int year) {
5          selectdate = new DateInMonth(month, year);
6      }
7      public boolean selected (Entry e) {
8          return selectdate.selected (e.getDate());
9      }
10 }
```

```
1  // teste ob ein Date in einem bestimmten Monat liegt
2  public class DateInMonth implements ISelect<Date> {
3      private final int month;
4      private final int year;
5      public DateInMonth (int month, int year) {
6          this.month = month; this.year = year;
7      }
8      public boolean selected (Date d) {
9          return d.getMonth() == this.month && d.getYear() == this.year;
10     }
11 }
```

Verwendung der generischen Auswahl

Aktivitäten von mehr als 10km Länge

```
1   ActivityLog myLog = ...;
2   ISelect<Entry> moreThan10 = new DistanceLongerThan (10);
3   Collection<Entry> myLongDist = myLog.filter (moreThan10);
```

Verwendung der generischen Auswahl

Aktivitäten von mehr als 10km Länge

```
1   ActivityLog myLog = ...;
2   ISelect<Entry> moreThan10 = new DistanceLongerThan (10);
3   Collection<Entry> myLongDist = myLog.filter (moreThan10);
```

Aktivitäten im Juni/Juli 2003

```
1   ActivityLog myLog = ...;
2   ISelect<Entry> inJune2003 = new EntryInMonth (6, 2003);
3   Collection<Entry> myJune = myLog.filter (inJune2003);
4   // ... in July
5   Collection<Entry> myJuly = myLog.filter (new EntryInMonth (7, 2003));
```

Alternative: Generische Methode

```
1  public class Filter {
2      // generic method
3      public static <X> Collection<X> filter (Collection<X> src, ISelect<X> pred) {
4          Collection<X> result = new ArrayList<X>();
5          for (X elem : src) {
6              if (pred.selected(elem)) {
7                  result.add(elem);
8              }
9          }
10         return result;
11     }
12 }
```

- Gewöhnliche Klasse mit (statischer) generischer Methode
- Einführen von Typvariablen durch <X> vor dem Ergebnistyp der Methode
- Nicht mehr spezifisch für einen bestimmten Elementtyp

Verwendung der generischen Auswahl

Implementierung in ActivityLog

```
27      public Collection<Entry> filter (ISelect<Entry> pred) {
28          return Filter.filter (this.activities, pred);
29      }
```

Verwendung der generischen Auswahl

Implementierung in ActivityLog

```
27      public Collection<Entry> filter (ISelect<Entry> pred) {
28          return Filter.filter (this.activities, pred);
29      }
```

Aktivitäten von mehr als 10km Länge

```
1 ActivityLog myLog = ...;
2 ISelect<Entry> moreThan10 = new DistanceLongerThan (10);
3 Collection<Entry> myLongDist = Filter.filter(myLog.getActivities(), moreThan10);
```

Verwendung der generischen Auswahl

Implementierung in ActivityLog

```
27      public Collection<Entry> filter (ISelect<Entry> pred) {
28          return Filter.filter (this.activities, pred);
29      }
```

Aktivitäten von mehr als 10km Länge

```
1 ActivityLog myLog = ...;
2 ISelect<Entry> moreThan10 = new DistanceLongerThan (10);
3 Collection<Entry> myLongDist = Filter.filter(myLog.getActivities(), moreThan10);
```

Aktivitäten im Juli 2003

```
1 ActivityLog myLog = ...;
2 Collection<Entry> myJuly =
3   Filter.filter (myLog.getActivities(), new EntryInMonth (7, 2003));
```

Collections transformieren

Listen transformieren

Aufgabe: Ändere alle Einträge im ActivityLog von km auf Meilen.

- Das Abändern von Einträgen macht auch für andere Collections Sinn.

$\Rightarrow$ entwerfe generische Methode

$\Rightarrow$ entwerfe zunächst ein allgemeines Änderungsinterface

Listen transformieren

Aufgabe: Ändere alle Einträge im ActivityLog von km auf Meilen.

- Das Abändern von Einträgen macht auch für andere Collections Sinn.

$\Rightarrow$ entwerfe generische Methode

$\Rightarrow$ entwerfe zunächst ein allgemeines Änderungsinterface

Änderungsinterface

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

Collections transformieren

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 }
```

- Ursprüngliche Collection bleibt unverändert
- Ergebnis in neuer Collection (mit anderer Reihenfolge)
- Transformation kann den Typ der Elemente ändern

Km in Meilen umwandeln

```
1  public class ChangeKmToMiles implements ITransform<Entry,Entry> {
2      // Umrechnungsformel
3      private static double kmToMiles (double km) {
4          return km * 0.6214;
5      }
6      // Transformation
7      public Entry transform (Entry e) {
8          return new Entry (e.getDate(),
9                            kmToMiles(e.getDistance()),
10                           e.getDuration());
11     }
12 }
```

Km in Meilen umwandeln

```
1  public class ChangeKmToMiles implements ITransform<Entry,Entry> {
2      // Umrechnungsformel
3      private static double kmToMiles (double km) {
4          return km * 0.6214;
5      }
6      // Transformation
7      public Entry transform (Entry e) {
8          return new Entry (e.getDate(),
9                            kmToMiles(e.getDistance()),
10                           e.getDuration());
11     }
12 }
```

Verwendung

```
1          Collection<Entry> logInKm = ...;
2          ITransform<Entry> kmToMiles = new ChangeKmToMiles ();
3          Collection<Entry> logInMiles = Transform.map (logInKm, kmToMiles);
```

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());
```

Typschranken

Anwendbarkeit von map

Angenommen, wir erweitern Entry um ein Kommentarfeld.

```
1 public class CommentedEntry extends Entry {
2     final String comment;
3     public CommentedEntry (Date d, double distance, double duration, String comment)
4         super(d, distance, duration);
5         this.comment = comment;
6     }
7 }
```

Dann ist Transform.map(..., **new** Speed()) nicht ohne weiteres auf eine Collection<CommentedEntry> anwendbar. Insbesondere liefert der folgende Code einen Typfehler:

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

Anwendbarkeit von map II

Was ist die Ursache des Typfehlers?
Betrachte die Beteiligten:

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

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

Damit ergibt sich folgender Widerspruch für die Belegung von X:

- **new** Speed() : ITransform<Entry,Double> also X=Entry und Y=Double
- cce : Collection<CommentedEntry> also X=CommentedEntry

Anwendbarkeit von map II

Was ist die Ursache des Typfehlers?
Betrachte die Beteiligten:

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

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

Damit ergibt sich folgender Widerspruch für die Belegung von X:

- **new** Speed() : ITransform<Entry,Double> also X=Entry und Y=Double
- cce : Collection<CommentedEntry> also X=CommentedEntry

Achtung

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

Vererbung und generische Klassen

Warum ist Collection<CommentedEntry> nicht verwendbar, wenn Collection<Entry> verlangt ist?
Angenommen doch, dann wird folgender Code akzeptiert:

```
1 void m(Collection<Entry> ce) {
2   ce.add(new Entry(1, 0.05);
3 }
4 ...
5 Collection<CommentedEntry> cce = new ArrayList<CommentedEntry>();
6 m(cce);
7 for(CommentedEntry e : cce) {
8   cce.getComment(); // run-time error!
9 }
```

Dieser Code liefert einen Laufzeitfehler, da das von m hinzugefügte Objekt keine getComment() Methode besitzt.

Vererbung und generische Klassen

Abhilfe erfolgt in Java durch **Typschranken** und **Wildcards**

Beobachtung

- Falls eine Methode von einem Objekt von Typ Collection<A> **nur liest**, dann darf für A auch ein beliebiger Subtyp (z.B. Subklasse) eingesetzt werden.
  Schreibweise: Collection<? **extends** A>
- Falls eine Methode in ein Objekt vom Typ Collection<A> **nur schreibt**, dann darf für A ein beliebiger Supertyp (z.B. Superklasse) eingesetzt werden.
  Schreibweise: Collection<? **super** A>

Besserer Typ für map

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

- Mit diesem Typ funktioniert das Beispiel (Transformation von Collection<CommentedEntry> mit **new** Speed())
- Die Implementierung von map (d.h., der Code) bleibt **gleich**!

Weitere Beispiele aus dem Collection Interface

```
1  public interface Collection<E> {
2    // adds the element o
3    boolean add (E o);
4    // adds all elements in collection c
5    boolean addAll (Collection<? extends E> c);
6    // removes element o
7    public boolean remove (Object o);
8    // removes all elements
9    public void clear();
10   // removes all elements in c
11   public boolean removeAll(Collection<?> c);
12   // removes all elements not in c
13   public boolean retainAll(Collection<?> c);
14 }
```

- Abkürzung: <?> für <? **extends** Object>

Zusammenfassung: Subtyping und generische Klassen

- Für generische Klassen gelten nur deklarierte Subtyp-Beziehungen.
- Insbesondere:
  - Falls A Subklasse von B, dann *gilt nicht*, dass `Collection<A>` Subtyp von `Collection<B>` ist.
  - `Collection<A>` und `Collection<B>` haben keinerlei (Vererbungs-) Beziehung zueinander.
  - Gilt analog für alle anderen generischen Klassen.
- Aber falls A Subklasse von B, dann ...
  - `Collection<? extends B>` ist Supertyp von `Collection<A>`
  - Elemente von `Collection<? extends B>` können nur als `B` gelesen werden.

Intermezzo: Geschachtelte Klassen (Nested Classes)

Geschachtelte Klassen

- Eine Klasse, die **ISelect** oder **ITransform** implementiert, ist immer nur im Kontext von einer bestimmten Klasse sinnvoll.
- In Java kann dies durch eine **geschachtelte Klasse** ausgedrückt werden.
- Das eine vollständige Klassendefinition, die innerhalb einer anderen Klasse erfolgt.

Beispiel: Geschachtelte Klassen

```
1  public class Entry {
2    private double distance;
3    public static class DistanceLongerThan implements ISelect<Entry> {
4      private final double length;
5      public DistanceLongerThan (double length) { this.length = length; }
6      public boolean selected(Entry e) {
7        return e.distance >= length;
8      }
9    }
10   // rest of Entry class
11 }
```

- Private Felder der umschliessenden Klasse sind sichtbar
- **Keine Vererbungsbeziehung** zur umschliessenden Klasse!

Verwendung

```
1  ISelect<Entry> = new Entry.DistanceLongerThan( 21.195 );
```

Anonyme Klassen

- Oft wird eine geschachtelte Klasse nur einmal benötigt.
- Dann ist es sinnlos, ihr extra einen Namen zu geben.
- Typische Verwendungen
  - Callback Aktionen bei Programmierung von GUIs
  - Einmalige Selektoren oder Transformer
  - Vergleichsoperationen (siehe unten)

Beispiel: Anonyme Klassen

Als Argument zur Filter-Methode

```
1  ActivityLog al = ...;
2  Collection<Entry> clong = al.filter (new ISelect<Entry>() {
3    public boolean selected(Entry e) {
4      return e.getDistance() >= 21.195;
5    }
6  });
```

Syntax Anonyme Klasse

- **new** mit Name eines Interface oder abstrakter Klasse
- gefolgt vom Rumpf einer Klasse mit Deklaration von Feldern und Methoden
- alle abstrakten Methoden (bzw. Interfacemethoden) müssen implementiert werden.
- (wird intern in eine geschachtelte Klasse umgewandelt)

Vergleichen von Objekten

Vergleichen

```
1 package java.lang;
2 public interface Comparable<T> {
3   int compareTo (T that);
4 }
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Verwendung

```
1       Integer i1 = new Integer (42);
2       Integer i2 = new Integer (4711);
3       int result = i1.compareTo (i2);
4       // result < 0
```

Vergleichbar machen

```
1  public class Date implements Comparable<Date> {
2    ...
3    // Vergleich für Comparable<Date>
4    public int compareTo (Date that) {
5      if (this.year < that.year ||
6            this.year == that.year && this.month < that.month ||
7            this.year == that.year && this.month == that.month
8              && this.day < that.day) {
9        return -1;
10     } else if (this.year == that.year && this.month == that.month
11                     && this.day == that.day) {
12       return 0;
13     } else {
14       return 1;
15     }
16   }
17 }
```

Vergleichbar machen

Achtung!

- Eine Implementierung von Comparable<T> muss eine totale Ordnung auf Objekten vom Typ T definieren.
  - reflexiv
  - transitiv
  - antisymmetrisch
  - total
- compareTo muss mit der Implementierung von equals kompatibel sein:
  - x.compareTo (y) == 0 genau dann, wenn x.equals (y)
- java.util.Collection verlässt sich darauf...

Maximum einer Collection

Erste Implementierung

```
1   // maximum of a non-empty collection
2   public static <T extends Comparable<T>> T max(Collection<T> coll) {
3     T candidate = coll.iterator().next();
4     for (T elem : coll) {
5       if (candidate.compareTo(elem) < 0) {
6         candidate = elem;
7       }
8     }
9     return candidate;
10  }
```

- Funktioniert für alle Typen `T`, die `Comparable<T>` implementieren.
- Effizienz kann verbessert werden. Wie?

Beispiele

- `Integer`

```
1  List<Integer> ints = Arrays.asList(0, 1, 2);
2  assert max(ints) == 2;
```

- `String`

```
1  List<String> strs = Arrays.asList("zero", "one", "two");
2  assert max(strs) == "zero";
```

- **Nicht** für `Number`

```
1  List<Number> nums = Arrays.asList(1,2,3.14);
2  assert max(nums) == 3.14; // type error
```

Verbesserte Signatur

- Ausgangspunkt

```
1  public static <T extends Comparable<T>>
2  T max(Collection<T> coll);
```

- Die Eingabecollection darf auch einen Subtyp von `T` als Elementtyp haben, **da aus ihr nur gelesen wird.**

```
1  public static <T extends Comparable<T>>
2  T max(Collection<? extends T> coll);
```

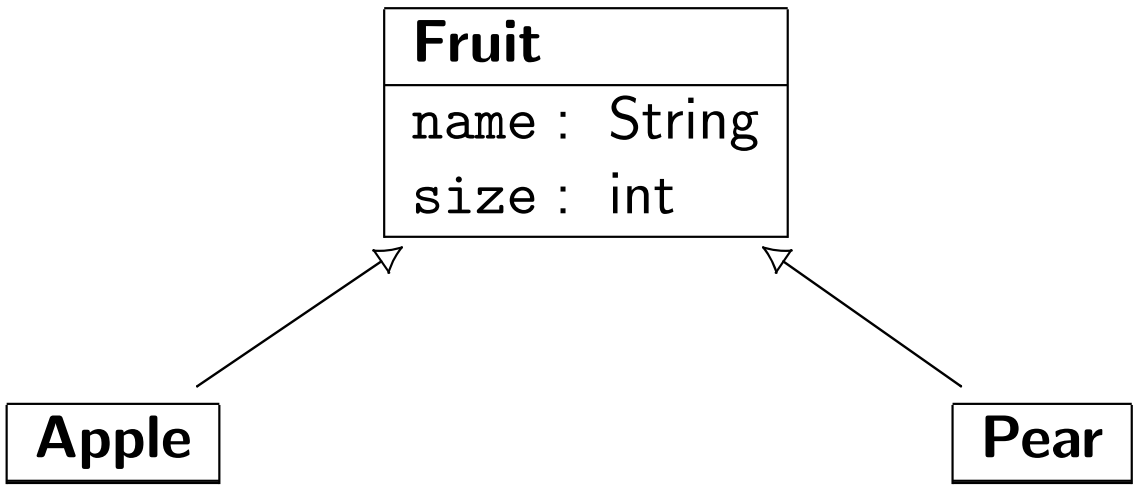
- Die Vergleichsoperation könnte auch auf einem Supertyp von `T` definiert sein.

```
1  public static <T extends Comparable<? super T>>
2  T max(Collection<? extends T> coll);
```

- Definition in `java.util.Collections` noch etwas komplizierter
- Rumpf der Methode bleibt unverändert

Fruchtige Beispiele

Äpfel und Birnen



Zwei Wahlmöglichkeiten beim Entwurf der Vergleichsoperation:

- Äpfel und Birnen dürfen verglichen werden
- Äpfel und Birnen dürfen **nicht** verglichen werden

Möglichkeit 1: Vergleich von Äpfel und Birnen erlaubt

```
1  abstract class Fruit implements Comparable<Fruit> {
2      protected String name;
3      protected int size;
4      protected Fruit (String name, int size) {
5          this.name = name; this.size = size;
6      }
7      public boolean equals (Object o) {
8          if (o instanceof Fruit) {
9              Fruit that = (Fruit) o;
10             return this.name.equals (that.name) && this.size == that.size;
11         } else { return false; }
12     }
13     public int hashCode() {
14         return name.hashCode()*29 + size;
15     }
16     public int compareTo (Fruit that) {
17         return this.size < that.size ? -1 :
18             this.size > that.size ? 1 : this.name.compareTo(that.name);
19     }
20 }
```

Äpfel und Birnen

```
1 class Apple extends Fruit {
2     public Apple (int size) {
3         super ("Apple", size);
4     }
5 }
```

```
1 class Pear extends Fruit {
2     public Pear (int size) {
3         super ("Pear", size);
4     }
5 }
```

Test mit Vergleichen

```
1  class ApplePearTest {
2      public static void main (String[] arg) {
3          Apple a1 = new Apple(1); Apple a2 = new Apple(2);
4          Pear o3 = new Pear(3); Pear o4 = new Pear(4);
5
6          List<Apple> apples = Arrays.asList(a1,a2);
7          assert Collections.max(apples).equals(a2);
8
9          List<Pear> pears = Arrays.asList(o3,o4);
10         assert Collections.max(pears).equals(o4);
11
12         List<Fruit> mixed = Arrays.<Fruit>asList(a1,o3);
13         assert Collections.max(mixed).equals(o3); // ok
14     }
15 }
```

Einschub: Signatur von `max`

- Die allgemeine Signatur von `max` war

```
1  public static <T extends Comparable<? super T>>
2  T max(Collection<? extends T> coll);
```

- Für `Fruit` ist dies erforderlich, da sonst `max` nicht auf pears anwendbar wäre.

  Pear **extends** Comparable<Pear> **gilt nämlich nicht**!
- Aber Pear **extends** Comparable<? **extends** Fruit> ist erfüllt, denn
  - Pear **extends** Fruit
  - Fruit **extends** Comparable<Fruit>

  implizieren, dass
  - Fruit **super** Pear
  - Pear **extends** Comparable<Fruit>

Möglichkeit 2: kein Vergleich von Äpfeln mit Birnen

```
1  abstract class Fruit1 {
2      protected String name;
3      protected int size;
4      protected Fruit1 (String name, int size) {
5          this.name = name; this.size = size;
6      }
7      public boolean equals (Object o) {
8          if (o instanceof Fruit1) {
9              Fruit1 that = (Fruit1) o;
10             return this.name.equals (that.name) && this.size == that.size;
11         } else { return false; }
12     }
13     public int hashCode() {
14         return name.hashCode()*29 + size;
15     }
16     protected int compareTo (Fruit1 that) {
17         return this.size < that.size ? -1 :
18                 this.size > that.size ? 1 : this.name.compareTo (that.name);
19     }
20 }
```

Äpfel und Birnen

```
1 class Apple1 extends Fruit1 implements Comparable<Apple1> {
2     public Apple1 (int size) {
3         super ("Apple", size);
4     }
5     public int compareTo (Apple1 a) {
6         return super.compareTo(a);
7     }
8 }
```

```
1 class Pear1 extends Fruit1 implements Comparable<Pear1> {
2     public Pear1 (int size) {
3         super ("Pear", size);
4     }
5     public int compareTo (Pear1 that) {
6         return super.compareTo (that);
7     }
8 }
```

Test mit Vergleichen

```
1  class ApplePearTest1 {
2      public static void main (String[] arg) {
3          Apple1 a1 = new Apple1(1); Apple1 a2 = new Apple1(2);
4          Pear1 o3 = new Pear1(3); Pear1 o4 = new Pear1(4);
5
6          List<Apple1> apples = Arrays.asList(a1,a2);
7          assert Collections.max(apples).equals(a2);
8
9          List<Pear1> pears = Arrays.asList(o3,o4);
10         assert Collections.max(pears).equals(o4);
11
12         List<Fruit1> mixed = Arrays.<Fruit1>asList(a1,o3);
13         assert Collections.max(mixed).equals(o3); // type error
14     }
15 }
```

Andere Ordnungen

Alternative Vergleiche

- Eine Anwendung benötigt manchmal eine andere als die "natürliche" Ordnung.
- Beispiel:
  - Früchte dem Namen nach vergleichen
  - Strings der Länge nach vergleichen
- Java stellt dafür das `Comparator` Interface bereit.

Comparator

```
1  interface Comparator<T> {
2    public int compare(T o1, T o2);
3  }
```

- compare (x,y) liefert
  - < 0, falls x kleiner als y
  - $= 0$, falls x gleich y
  - > 0, falls x größer als y
- Forderung: Konsistenz mit `equals` bei Verwendung mit sorted set or sorted map
  - Konsistenz: `compare (x,y) == 0` genau dann, wenn `x.equals(y)`

Beispiel: Strings zuerst der Länge nach vergleichen

```
1  class SizeOrder implements Comparator<String> {
2    public SizeOrder () {}
3    public int compare (String x, String y) {
4      return x.length() < y.length() ? −1 :
5             x.length() > y.length() ? 1 :
6             x.compareTo(y);
7    }
8  }
```

Verwendung

```
1  assert "two".compareTo("three") > 0;
2  assert new SizeOrder().compare ("two", "three") < 0;
```

Beispiel nochmal mit anonymer Klasse

```
1  Comparator<String> sizeOrder =
2  new Comparator<String> () {
3    public int compare (String x, String y) {
4      return x.length() < y.length() ? −1 :
5             x.length() > y.length() ? 1 :
6             x.compareTo(y);
7    }
8  };
```

Verwendung

```
1  assert "two".compareTo("three") > 0;
2  assert sizeOrder.compare ("two", "three") < 0;
```

Comparator in der Java-Bibliothek

- Die Java-Bibliothek enthält immer beide Varianten, für `Comparable` und für `Comparator`
- Beispiel: `max`

```
1  public static <T extends Comparable<? super T>>
2  T max(Collection<? extends T> coll);
```

```
1  public static <T>
2  T max(Collection<? extends T> coll, Comparator<? super T> cmp);
```

- Analog für `min`

Beispiele mit Comparator

```
1  Collection<String> strings = Arrays.asList("from", "aaa", "to", "zzz");
2  assert max(strings).equals("zzz");
3  assert min(strings).equals("aaa");
4  assert max(strings, sizeOrder).equals("from");
5  assert min(strings, sizeOrder).equals("to");
```

Maximum mit Comparator

```
1   public static <T>
2   T max(Collection <? extends T> coll, Comparator<? super T> comp) {
3     Iterator<? extends T> iter = coll.iterator();
4     T candidate = iter.next();
5     while(iter.hasNext()) {
6       T elem = iter.next();
7       if (comp.compare(candidate, elem) < 0) {
8         candidate = elem;
9       }
10    }
11    return candidate;
12  }
```

Natürliche Ordnung als Comparator

Mit anonymer Klasse

```
1  public static <T extends Comparable<? super T>>
2  Comparator<T> naturalOrder() {
3    return
4    new Comparator<T> () {
5      public int compare(T t1, T t2) {
6        return t1.compareTo(t2);
7      }
8    };
9  }
```

Umdrehen der Ordnung

Mit anonymer Klasse

```
1   public static <T>
2   Comparator<T> reverseOrder(final Comparator<T> comp) {
3     return
4     new Comparator<T> () {
5       public int compare(T t1, T t2) {
6         return comp.compare(t2, t1);
7         // alternativ:
8         // return -comp.compare(t1, t2);
9       }
10    };
11  }
```

Implementierung des Minimums

- Mit Comparable:

```
1 public static <T extends Comparable<? super T>>
2 T min(Collection<? extends T> coll) {
3   return min(coll, Comparison.<T>naturalOrder());
4 }
```

- Mit Comparator:

```
1 public static
2 <T> T min(Collection<? extends T> coll, Comparator<? super T> comp) {
3   return max(coll, reverseOrder(comp));
4 }
```

Lexikographisches Vergleichen für Listen aus Elementvergleich

```
1   public static <T>
2   Comparator<List<T>> listComparator(final Comparator<T> comp) {
3     return new Comparator<List<T>>() {
4       public int compare(List<T> l1, List<T> l2) {
5         int n1 = l1.size();
6         int n2 = l2.size();
7         for(int i = 0; i < Math.min(n1, n2); i++) {
8           int k = comp.compare(l1.get(i), l2.get(i));
9           if (k != 0) {
10            return k;
11          }
12        }
13        return n1 < n2 ? -1 :
14        n1 == n2 ? 0 : 1;
15      }
16    };
17  }
```

Aufzählungstypen (Enumerated Types)

Aufzählungstypen in Java 5

- Ein Aufzählungstyp enthält endlich viele benannte Elemente.
- Beispiel

```
1   enum Season { WINTER, SPRING, SUMMER, AUTUMN }
```

- Diese Werte können im Programm als Konstanten verwendet werden.
- Konvention: Konstanten werden komplett groß geschrieben.
- Die Implementierung von Aufzählungstypen erfolgt mit Hilfe einer generischen Klasse mit einer interessanten Typschranke.
- (Implementierung ist eingebaut im Java Compiler)

Implementierung von Season

```
1   enum Season { WINTER, SPRING, SUMMER, AUTUMN }
```

- Es gibt eine Klasse Season
- Von dieser Klasse gibt es genau vier Instanzen, eine für jeden Wert.
- Jeder Wert ist durch ein **static final** Feld in Season verfügbar.
- Season erbt von einer Klasse Enum, die das Grundgerüst der Implementierung liefert.
- (Implementierung nach Joshua Bloch, Effective Java)

Die Klasse Enum

```
1  public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
2      private final String name;
3      private final int ordinal;
4      protected Enum (String name, int ordinal) {
5          this.name = name; this.ordinal = ordinal;
6      }
7      public final String name() { return name; }
8      public final int ordinal() { return ordinal; }
9      public String toString() { return name; }
10     public final int compareTo(E o) {
11         return ordinal − o.ordinal;
12     }
13 }
```

Die Klasse Season

```
1  // corresponding to
2  // enum Season { WINTER, SPRING, SUMMER, AUTUMN }
3  final class Season extends Enum<Season> {
4    private Season(String name, int orginal) { super(name, ordinal); }
5    public static final Season WINTER = new Season ("WINTER", 0);
6    public static final Season SPRING = new Season ("SPRING", 1);
7    public static final Season SUMMER = new Season ("SUMMER", 2);
8    public static final Season AUTUMN = new Season ("AUTUMN", 3);
9    private static final Season[] VALUES = {WINTER, SPRING, SUMMER, AUTUMN};
10   public static Season[] values() { return VALUES.clone(); }
11   public static Season valueOf (String name) {
12     for (Season e : VALUES) {
13       if (e.name().equals (name)) { return e; }
14     }
15     throw new IllegalArgumentException();
16   }
17 }
```

Erklärung für die Typschranken

```
1 public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
```

```
1 final class Season extends Enum<Season> {
```

- Wofür ist Enum<E **extends** Enum<E>> notwendig?
- Die Klasse Season ist passend definiert:
  class Season **extends** Enum<Season>
- Da außerdem Enum<E> **implements** Comparable<E>, gilt weiter
  Enum<Season> **implements** Comparable<Season> und
  Season **extends** Comparable<Season>

Erklärung für die Typschranken

```
1 public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
```

```
1 final class Season extends Enum<Season> {
```

- Wofür ist Enum<E **extends** Enum<E>> notwendig?
- Die Klasse Season ist passend definiert:
  class Season **extends** Enum<Season>
- Da außerdem Enum<E> **implements** Comparable<E>, gilt weiter
  Enum<Season> **implements** Comparable<Season> und
  Season **extends** Comparable<Season>

$\Rightarrow$ Elemente von Season miteinander vergleichbar, **aber nicht** mit Elementen von anderen Aufzählungstypen!

So tut's nicht: zu einfach

- Ohne die Typschranken könnten Elemente von beliebigen Aufzählungstypen miteinander verglichen werden.
- Angenommen, es wäre
  - **class** Enum **implements** Comparable<Enum>
  - **class** Season **extends** Enum
- Dann gilt Season **extends** Comparable<Enum>, genau wie für jeden anderen Aufzählungstyp!
- (Vgl. Fruit-Beispiel)
- **Dieses Verhalten ist unerwünscht!**