Programmieren in Java

Vorlesung 07: Generics

Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2013

Inhalt

Generics

- Generische Klassen und Interfaces
- Exkurs: Wrapperklassen
- Generische Suche
- Collections transformieren
- Vergleichen: equals, hashCode und compareTo

Generische Klassen und Interfaces

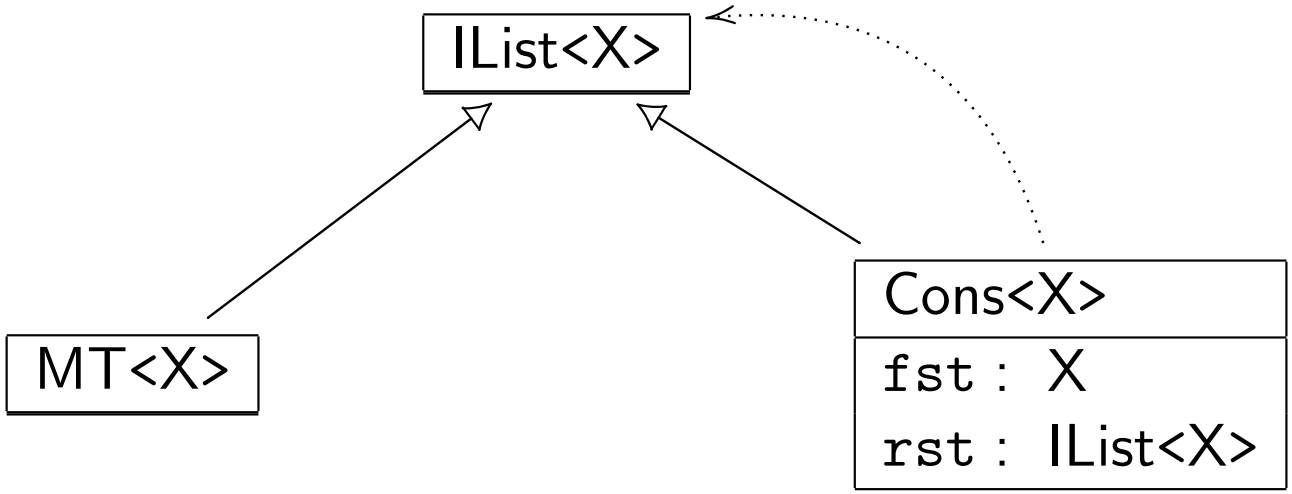
Generics

- *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

Generisches Paar

```
public class GenericPair<X,Y> {
    private X fst;
    private Y snd;
    GenericPair(X fst, Y snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public X getFst() {
        return this.fst;
    }
    public Y getSnd() {
        return this.snd;
    }
}
```

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

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

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

```
// Variante nicht-leere Liste
class Cons<X> implements IList<X> {
    private X fst;
    private IList<X> rst;

    public Cons (X fst, IList<X> rst) {
        this.fst = fst;
        this.rst = rst;
    }
}
```

Verwendung von generischen Listen

Liste von int bzw. Integer

- Achtung: **Typvariablen können nur für Referenztypen stehen!**
- Anstelle von primitiven Typen müssen die Wrapperklassen verwendet werden (Konversion von Werten automatisch dank *Autoboxing*)

```
// Aufbau der Liste
IList<Integer> i1 = new MT<Integer> ();
IList<Integer> i2 = new Cons<Integer> (32168, i1);
IList<Integer> i3 = new Cons<Integer> (new Integer ("32768"), i2);
IList<Integer> i4 = new Cons<Integer> (new Integer (-14), i3);
```

Exkurs: Wrapperklassen

- Für jeden primitiven Datentyp stellt Java eine Klasse bereit, deren Instanzen einen Wert des Typs in ein Objekt verpacken.
- Beispiele

primitiver Typ	Wrapperklasse
int	java.lang.Integer
double	java.lang.Double
boolean	java.lang.Boolean

- Klassen- und Interfacetypen heißen (im Unterschied zu primitiven Typen) auch *Referenztypen*.

Methoden von Wrappenklassen

- Wrapperklassen beinhalten (statische) Hilfsmethoden und Felder zum Umgang mit Werten des zugehörigen primitiven Datentyps.
- Vorsicht: Ab Version 5 konvertiert Java automatisch zwischen primitiven Werten und Objekten der Wrapperklassen. (*autoboxing*)

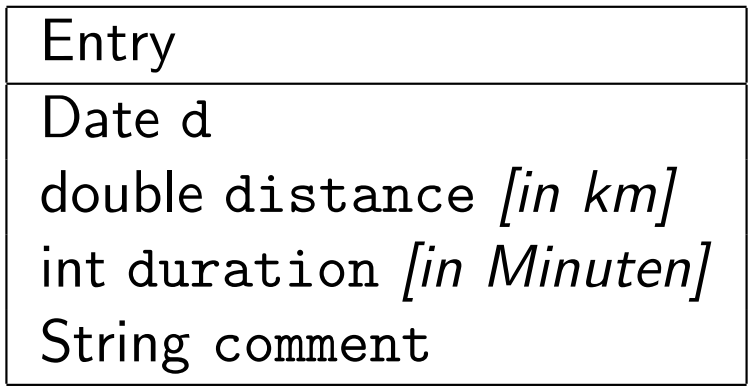
Beispiel: Integer (Auszug)

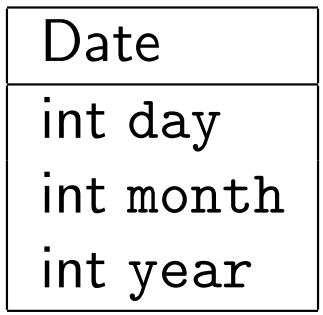
```
static int MAX_VALUE; // maximaler Wert von int
static int MIN_VALUE; // minimaler Wert von int

Integer (int value);
Integer (String s); // konvertiert String -> int

int compareTo(Integer anotherInteger);
int intValue();
static int parseInt(String s);
```

Generische Suche

Vorspiel: Eintrag im ActivityLog

Klassendiagramm





Eintrag im ActivityLog

Implementierung

```
1  // ein Eintrag in einem ActivityLog
2  class Entry {
3     Date d;
4     double distance; // in km
5     int duration; // in Minuten
6     String comment;
7
8     Entry(Date d, double distance, int duration, String comment) {
9        this.d = d;
10       this.distance = distance;
11       this.duration = duration;
12       this.comment = comment;
13    }
14 }
```

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

Generische Suche

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

Beispiele

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

Generische Suche

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 Suche

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

```
public Collection<Entry> distanceLongerThan (double length) {
    Collection<Entry> result = new ArrayList<Entry>();
    for (Entry e : this.activities) {
        if (e.distanceLongerThan(length)) {
            result.add(e);
        }
    }
    return result;
}
```

Generischer Ansatz

Generische Auswahl

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

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

Entwurfsmuster *Strategy*

- Suche mit abstrakter Selektion
- Instantiiert durch konkrete Selektionen

Instanzen der generischen Auswahl

```
// teste ein Entry ob er eine längere Entfernung enthält
class DistanceLongerThan implements ISelect<Entry> {
    private double limit;
    public DistanceLongerThan (double limit) {
        this.limit = limit;
    }

    public boolean selected (Entry e) {
        return e.distanceLongerThan(this.limit);
    }
}
```

Instanzen der generischen Auswahl

```
// teste ob ein Entry in einem bestimmten Monat liegt
class EntryInMonth implements ISelect<Entry> {
    private ISelect<Date> selectdate;
    public EntryInMonth (int month, int year) {
        this.selectdate = new DateInMonth(month, year);
    }
    public boolean selected (Entry e) {
        return this.selectdate.selected (e.d);
    }
}
```

```
// teste ob ein Date in einem bestimmten Monat liegt
class DateInMonth implements ISelect<Date> {
    private int month; private int yearM;
    public DateInMonth (int month, int year) {
        this.month = month; this.year = year;
    }
    public boolean selected (Date d) {
        return d.getMonth() == this.month && d.getYear == this.year;
    }
}
```

Implementierung der generischen Auswahl

... as `filter` in ActivityLog

```
public Collection<Entry> filter (ISelect<Entry> pred) {
    Collection<Entry> result = new ArrayList<Entry>();
    for (Entry e : this.activities) {
        if (pred.selected(e)) {
            result.add(e);
        }
    }
    return result;
}
```

Verwendung der generischen Auswahl

Aktivitäten von mehr als 10km Länge

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

Verwendung der generischen Auswahl

Aktivitäten von mehr als 10km Länge

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

Aktivitäten im Juni/Juli 2003

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

Alternative: Generische Methode

```
public class Filter {
    // generic method
    public static <X> Collection<X> filter (Collection<X> source, ISelect<X> pred) {
        Collection<X> result = new ArrayList<X>();
        for (X elem : source) {
            if (pred.selected(elem)) {
                result.add(elem);
            }
        }
        return result;
    }
}
```

- Gewöhnliche Klasse mit generischer Methode
- Einführen von Typvariablen durch <X> vor dem Ergebnistyp der Methode

Generische Suche

Destruktiv

Bisherige Methode: funktional

- Collection im Argument unverändert
- Ergebnis ist neue Struktur mit Elementen aus der ursprünglichen Collection

Generische Suche

Destruktiv

Bisherige Methode: funktional

- Collection im Argument unverändert
- Ergebnis ist neue Struktur mit Elementen aus der ursprünglichen Collection

Alternative: destruktiv

- Abändern der ursprünglichen Collection

Destruktiver generischer Filter

```
public class DFilter {
    public static <X> void filter(Collection<X> source, ISelect<X> pred) {
        Iterator<X> iter = source.iterator();
        while(iter.hasNext()) {
            X elem = iter.next();
            if (pred.selected(elem)) {
                iter.remove();
            }
        }
    }
}
```

- ... verwendet java.util.Iterator<X> zum Durchlaufen der Collection

Das Iterator Interface (abgekürzt)

```
public interface Iterator<E> {
    /**
     * @return {@code true} if the iteration has more elements
     */
    boolean hasNext();

    /**
     * @return the next element in the iteration
     */
    E next();

    /**
     * Removes from the underlying collection the last element returned
     * by this iterator (optional operation). This method can be called
     * only once per call to {@link #next}.
     */
    void remove();
}
```

Iterable

- Ein Iterator kann aus jedem Referenztyp gewonnen werden, der das Interface java.lang.Iterable<X> implementiert.
- Jede Collection implementiert Iterable.
- Ein Array [T] implementiert Iterable<T>
- Iterable wird auch für `foreach`-Schleifen benötigt.

```
public interface Iterable<T> {

    /**
     * @return an Iterator over a set of elements of type T.
     */
    Iterator<T> iterator();
}
```

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 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 allgemeines Änderungsinterface

Änderungsinterface

```
// change something
interface ITransform<X,Y> {
    public Y transform (X x);
}
```

Collections transformieren

Funktional

```
class Transform {
    public static <X,Y> Collection<Y> map(Collection<X> source,
                                          ITransform<X,Y> fun) {
        Collection<Y> result = new ArrayList<Y>();
        for (X item : source) {
            result.add(fun.transform(item));
        }
        return result;
    }
}
```

- Ursprüngliche Collection bleibt unverändert
- Ergebnis in neuer Collection
- Transformation kann den Typ der Elemente ändern

Km in Meilen umwandeln

```
class ChangeKmToMiles implements ITransform<Entry,Entry> {
    public ChangeKmToMiles () {}
    // Umrechnungsformel
    private static double kmToMiles (double km) {
        return km * 0.6214;
    }
    // Transformation
    public Entry transform (Entry e) {
        return new Entry (e.getDate(),
                          kmToMiles(e.getDistance()),
                          e.getDuration(),
                          e.getComment());
    }
}
```

Km in Meilen umwandeln

```
class ChangeKmToMiles implements ITransform<Entry,Entry> {
    public ChangeKmToMiles () {}
    // Umrechnungsformel
    private static double kmToMiles (double km) {
        return km * 0.6214;
    }
    // Transformation
    public Entry transform (Entry e) {
        return new Entry (e.getDate(),
                          kmToMiles(e.getDistance()),
                          e.getDuration(),
                          e.getComment());
    }
}
```

Verwendung

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

Collections transformieren

Destruktiv

```
interface IDTransform<X> {
    public void dtransform (X item);
}
```

- Transformation muss den Typ der Elemente erhalten

```
    public static <X> void forall (Collection<X> source,
                                   IDTransform<X> fun) {
        for (X item : source) {
            fun.dtransform(item);
        }
    }
```

Collections destruktiv transformieren

```
class ChangeKmToMilesDestructively implements IDTransform<Entry> {
    public ChangeKmToMiles () {}
    // Umrechnungsformel
    private static double kmToMiles (double km) {
        return km * 0.6214;
    }
    // Transformation
    public void dtransform (Entry e) {
        e.setDistance (kmToMiles(e.getDistance()));
    }
}
```

Verwendung

```
Collection<Entry> logInKm = ...;
IDTransform<Entry> kmToMiles = new ChangeKmToMilesDestructively ();
Transform.forall (logInKm, kmToMiles);
```

Intermezzo: Vergleichen

Die Klasse Object

Jede Klasse erbt von der Klasse Object, die in Java vordefiniert ist. Dort sind einige Methoden definiert, die für Objektvergleiche relevant sind:

```
public class Object {
  public boolean equals(Object obj) {
    return this == obj;
  }
  public int hashCode() { ... }
  public final Class<?> getClass() { ... }
  ...
}
```

- *Die Methoden* `equals` *und* `hashCode` *sollten im Normalfall überschrieben werden!*
- `getClass` *kann nicht überschrieben werden*, da mit **final** definiert.

Die `equals` Methode

```
public boolean equals(Object obj) { ... }
```

Die `equals` Methode testet, ob `this` "gleich" `obj` ist.
Sie muss eine *Äquivalenzrelation* auf Objekten $\neq$ `null` implementieren.
D.h. für alle Objekte x, y und z, die nicht null sind, gilt:

- `equals` muss *reflexiv* sein:
  Es gilt immer x.equals(x).
- `equals` muss *symmetrisch* sein:
  Falls x.equals(y), dann auch y.equals(x).
- `equals` muss *transitiv* sein:
  Falls x.equals(y) und y.equals(z), dann auch x.equals(z).

Die `equals` Methode (Fortsetzung)

Weitere Anforderungen an `equals`:

- `equals` muss *konsistent* sein:
  Wenn Objekte x und y nicht `null` sind, dann sollen wiederholte Aufrufe von x.equals(y) immer das gleiche Ergebnis liefern, es sei denn, ein Gleichheits-relevanter Bestandteil von x oder y hat sich geändert.
- Wenn x nicht `null` ist, dann liefert x.equals(null) das Ergebnis false.

Wichtig

- Jede Implementierung von `equals` muss auf diese Anforderungen hin getestet werden. Grund: *Manche Operationen in java.util.Collection verlassen sich darauf!*
- Die Methode equals(Object other) muss überschrieben werden.
  Typischer Fehler:

```
public boolean equals (MyType other) { ... }
```

Typische Implementierung von `equals`

```
public class A {
  public boolean equals (Object other) {
    if (this == other) { return true; }
    if (!other instanceof A) { return false; }
    // use with caution:
    if (!this.getClass().equals (other.getClass())) { return false; }
    A aother = (A)other;
    // compare relevant fields...
  }
}
```

Neuheiten:

- **instanceof**-Operator
- Typcast (A)other
- getClass()

Der **instanceof**-Operator

- Der boolesche Ausdruck

  ausdruck **instanceof** *objekttyp*

  testet ob der dynamische Typ des Werts von *ausdruck* ein Subtyp von *objekttyp* ist.
- Angenommen A **extends** B (Klassentypen):

```
A a = new A();
B b = new B();
B c = new A(); // statischer Typ B, dynamischer Typ A

a instanceof A // ==> true
a instanceof B // ==> true
b instanceof A // ==> false
b instanceof B // ==> true
c instanceof A // ==> true (testet den dynamischen Typ)
c instanceof B // ==> true
```

Der Typcast-Operator

- Der Ausdruck (*Typcast*)

  (*objekttyp*) *ausdruck*

  hat den statischen Typ *objekttyp*, falls der statische Typ von *ausdruck* entweder ein Supertyp oder ein Subtyp von *objekttyp* ist.
- Zur Laufzeit testet der Typcast, ob der **dynamische Typ** des Werts von *ausdruck* ein Subtyp von *objekttyp* ist und bricht das Programm ab, falls das nicht zutrifft. (Vorher sicherstellen!)
- Angenommen A **extends** C und B **extends** C (Klassentypen), aber A und B stehen in keiner Beziehung zueinander:

```
A a = new A(); B b = new B(); C c = new C(); C d = new A();

(A)a // statisch ok, dynamisch ok
(B)a // Typfehler
(C)a // statisch ok, dynamisch ok
(B)d // statisch ok, dynamischer Fehler
(A)d // statisch ok, dynamisch ok
```

Die `getClass`-Methode

```
public final Class<?> getClass() { ... }
```

Liefert ein Objekt, das den Laufzeittyp des Empfängerobjekts repräsentiert. Für jeden Typ T definiert das Java-Laufzeitsystem genau ein Objekt vom Typ Class<T>. Die Methoden dieser Klasse erlauben (z.B.) den Zugriff auf die Namen von Feldern und Methoden, das Lesen und Schreiben von Feldern und den Aufruf von Methoden.

Implementierung von `equals` (Fortsetzung)

```
// compare relevant fields; beware of null
// int f1; // any non−float primitive type
if (this.f1 != other.f1) { return false; }
// double f2; // float or double types
if (Double.compare (this.f2, other.f2) != 0) { return false; }
// String f3; // any reference type
if ((this.f3 != other.f3) &&
    ((this.f3 == null) || !this.f3.equals(other.f3))) {
  return false;
}
// after all state−relevant fields processed:
return true;
```

- Double.compare: Beachte spezielles Verhalten auf NaN und -0.0

Die `hashCode`-Methode

Vertrag von hashCode

- Bei mehrfachem Aufruf auf demselben Objekt muss hashCode() immer das gleiche Ergebnis liefern, solange keine Felder geändert werden, die für equals() relevant sind.
- Wenn zwei Objekte equals() sind, dann muss hashCode() auf beiden Objekten den gleichen Wert liefern.
- Die Umkehrung hiervon gilt nicht.

Rezept für eine brauchbare hashCode Implementierung

1. Initialisiere `int result = 17`
2. Für jedes Feld *f*, das durch equals() verglichen wird:
   2.1 Berechne einen Hash Code c für das Feld *f*, je nach Datentyp
   - `boolean`: `(f ? 1 : 0)`
   - `byte`, `char`, `short`: `(int)f`
   - `long`: (f ^ (f >>> 32))
   - `float`: `Float.floatToIntBits(f)`
   - `double`: konvertiere nach `long` ...
   - `f` ist Objektreferenz und wird mit equals vergleichen: `f.hashCode()` oder 0, falls `f == null`
   - `f` ist Array: verwende `java.util.Arrays.hashCode(f)`

   2.2 `result = 31 * result + c`
3. `return result`

Vergleichen

```
package java.lang;
interface Comparable<T> {
  int compareTo (T that);
}
```

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

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

Vergleichbar machen

```
class Date implements Comparable<Date> {
  ...
  // Vergleich für Comparable<Date>
  public int compareTo (Date that) {
    if (this.year < that.year ||
        this.year == that.year && this.month < that.month ||
        this.year == that.year && this.month == that.month
          && this.day < that.day) {
      return -1;
    } else if (this.year == that.year && this.month == that.month
                  && this.day == that.day) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

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...