

# Vorlesung 08: Vergleichen und Schranken

Peter Thiemann

Universität Freiburg, Germany

SS 2010

# Ziel

- ▶ Werkzeuge fürs Arbeiten mit Mengen und Abbildungen

# Inhalt

Gleichheit

Comparable

Maximum einer Collection

Fruchtbare Beispiele

Einschub: Innere und anonyme Klassen

Andere Ordnungen

Aufzählungstypen (Enumerated Types)

# Vergleichen von Objekten

# Statischer Typ vs dynamischer Typ

- ▶ Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- ▶ Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

# Statischer Typ vs dynamischer Typ

- ▶ Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- ▶ Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

## Beispiele

- ▶ Angenommen A **extends** B (Klassentypen).

```
A a = new A (); // rhs: Typ A, dynamischer Typ A
B b = new B (); // rhs: Typ B, dynamischer Typ B
B x = new A (); // rhs: Typ A, dynamischer Typ A
// für x gilt: Typ B, dynamischer Typ A
```

- ▶ Bei einem Interfacetyp ist der dynamische Typ **immer** ein Subtyp.
- ▶ Im Rumpf einer Methode definiert in der Klasse C hat `this` den statischen Typ C. Der dynamische Typ kann ein Subtyp von C sein, falls die Methode vererbt worden ist.

# Regeln für die Bestimmung des statischen Typs

- ▶ Falls Variable (Feld, Parameter)  $x$  durch  $\text{t}tt\ x$  deklariert ist, so ist der Typ von  $x$  genau  $\text{t}tt$ .
- ▶ Der Ausdruck `new C(...)` hat den Typ  $C$ .
- ▶ Wenn  $e$  ein Ausdruck vom Typ  $C$  ist und  $C$  eine Klasse mit Feld  $f$  vom Typ  $\text{t}tt$  ist, dann hat  $e.f$  den Typ  $\text{t}tt$ .
- ▶ Wenn  $e$  ein Ausdruck vom Typ  $C$  ist und  $C$  eine Klasse oder Interface mit Methode  $m$  vom Rückgabetyt  $\text{t}tt$  ist, dann hat  $e.m(...)$  den Typ  $\text{t}tt$ .
- ▶ Beim Aufruf eines Konstruktors oder einer Funktion müssen die Typen der Argumente jeweils Subtypen der Parametertypen sein.
- ▶ Bei einer Zuweisung muss der Typ des Audrucks auf der rechten Seiten ein Subtyp des Typs der Variable (Feld) sein.

# Vergleichen von Objekten

## Beispiel: Daten

```
class DateComparison {  
    Date d1 = new Date(27,3,1941);  
    Date d2 = new Date(8,5,1945);  
    Date d3 = new Date(8,5,1945);  
  
    boolean testD1D2 = d1 == d2; // Operator == auf Objekten  
    boolean testD2D3 = d2 == d3;  
    boolean testD3D3 = d3 == d3;  
}
```



# Vergleichen von Objekten

## Beispiel: Daten

```
class DateComparison {  
    Date d1 = new Date(27,3,1941);  
    Date d2 = new Date(8,5,1945);  
    Date d3 = new Date(8,5,1945);  
  
    boolean testD1D2 = d1 == d2; // Operator == auf Objekten  
    boolean testD2D3 = d2 == d3;  
    boolean testD3D3 = d3 == d3;  
}
```

## Ergebnis

```
DateComparison(  
    d1 = Date(...), d2 = Date(...), d3 = Date(...),  
    testD1D2 = false,  
    testD2D3 = false,  
    testD3D3 = true)
```

# Verschiedene Gleichheitsoperationen

- ▶ Der Gleichheitsoperator `==` ist auch auf Objekte anwendbar, aber liefert nicht das erwartete(?) Ergebnis!
- ▶ Er testet, ob beide Argument *dasselbe* Objekt bezeichnen.
- ▶ Oft ist **komponentenweise** Gleichheit gewünscht (*extensionale Gleichheit*).
- ▶ Muss programmiert werden (`equals()` Methode, vgl. `String`).

## Gleichheit für einfache Klassen

- ▶ Einfache Klassen enthalten Felder von primitivem Typ.
- ▶ Ihre Werte können mit `==` verglichen werden (bzw. mit `equals()` für `String`).
- ▶ Beispiel: Vergleichsmethode `boolean same(Date that)` für die `Date`-Klasse.

```
// is this date the same as that date?  
boolean same (Date that) {  
    return (this.day == that.day) &&  
           (this.month == that.month) &&  
           (this.year == that.year);  
}
```

## Gleichheit für Mengen

Betrachte eine Klasse Set2, bei der jedes Objekt eine Menge von int mit *genau zwei Elementen* repräsentiert.

```
// Mengen von genau zwei Zahlen
class Set2 {
    private int one;
    private int two;

    public Set2 (int one, int two) { ... }

    // Elementtest
    public boolean contains (int x) {
        return (x == this.one) || (x == this.two);
    }
}
```

# Gleichheit für Mengen

## Implementierung

- ▶ Komponentenweise Gleichheit nicht angemessen.
- ▶ Zwei Mengen sind gleich, wenn sie sich gegenseitig enthalten

$$\{16, 42\} = \{42, 16\}$$

- ▶ `same()` Methode in `Set2`:

```
// Gleichheitstest  
public boolean same (Set2 that) {  
    return (this.contains (that.one))  
        && (this.contains (that.two))  
        && (that.contains (this.one))  
        && (that.contains (this.two));  
}
```

## Gleichheit und Vererbung

```
class SpecialDate extends Date {  
    private int rating;  
  
    SpecialDate (int day, int month, int year, int rating) {  
        super (day, month, year);  
        this.rating = rating;  
    }  
}
```

- ▶ Spezielle Daten könnten mit einer Bewertung versehen sein.
- ▶ Die `same()` Methode aus der Klasse `Date` ist anwendbar, allerdings liefert sie nicht die erwarteten Ergebnisse.

```
class DateTest {  
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);  
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);  
  
    boolean testss1 = s1.same (s1); // ==> true !!!  
    boolean testss2 = s1.same (s2); // ==> true ???  
}
```

# Gleichheit und Vererbung

## same-Methode in der Subklasse

- Eine spezialisierte Version der `same()` Methoden in der Subklasse ist erforderlich

```
boolean same (SpecialDate that) {  
    return super.same (that) && (this.rating == that.rating);  
}
```

- Damit funktioniert das Beispiel

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate  
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);  
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);  
  
    boolean testss1 = s1.same (s1); // ==> true !!!  
    boolean testss2 = s1.same (s2); // ==> false !!!  
}
```

# Gleichheit und Vererbung

## Weitere Probleme

- ▶ Andere Beispiele funktionieren nicht wie erwartet.
- ▶ `same`-Gleichheit ist nicht transitiv!

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);
    Date d2 = new Date (12,8,2001);

    boolean testsd = s1.same (d2); // ==> true ???
    boolean testds = d2.same (s2); // ==> true ???
    boolean testss = s1.same (s2); // ==> false !!!
}
```



# Gleichheit und Vererbung

## Weitere Probleme

- ▶ Andere Beispiele funktionieren nicht wie erwartet.
- ▶ `same`-Gleichheit ist nicht transitiv!

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);
    Date d2 = new Date (12,8,2001);

    boolean testsd = s1.same (d2); // ==> true ???
    boolean testds = d2.same (s2); // ==> true ???
    boolean testss = s1.same (s2); // ==> false !!!
}
```

## Überraschung

- ▶ In den ersten beiden Fällen wird die Methode `same` der Klasse `Date` aufgerufen!
- ▶ Ursache: *Überladung* von Methoden.

# Überladung von Methoden

- ▶ Überladung: in einer Klasse gibt es mehrere Methoden mit gleichem Namen, die sich nur in Anzahl oder Typ der Parameter unterscheiden.
- ▶ Die Auswahl der tatsächlich aufgerufenen Methode erfolgt durch Java aufgrund des ermittelten Argumenttyps.

# Überladung von Methoden

- ▶ Überladung: in einer Klasse gibt es mehrere Methoden mit gleichem Namen, die sich nur in Anzahl oder Typ der Parameter unterscheiden.
- ▶ Die Auswahl der tatsächlich aufgerufenen Methode erfolgt durch Java aufgrund des ermittelten Argumenttyps.

## Beispiel

- ▶ In der Klasse `SpecialDate` gibt es **zwei** Methoden mit Namen `same`, **die sich nur im Parametertyp unterscheiden**:
  1. `boolean same (Date that)` (geerbt von `Date`)
  2. `boolean same (SpecialDate that)` (selbst definiert)
- ▶ In `testsd` wird #1, die geerbte Methode, aufgerufen, da `d2` den Typ `Date` hat.
- ▶ In `testds` wird auch #1 aufgerufen, da das Empfängerobjekt den Typ `Date` hat.

## Transitive Gleichheit

- ▶ Zufriedenstellende Implementierung benötigt **zwei** Methoden! (*double dispatch*)
- ▶ Schwierigkeit: Feststellen, ob das Argumentobjekt den gleichen *dynamischen Typ* wie das Empfängerobjekt hat.
- ▶ Die Methode `same (Date that)` muss in `Date` definiert sein und in allen Subklassen von `Date` überschrieben werden.
- ▶ Sie stellt lediglich fest, welchen *dynamischen Typ* das Empfängerobjekt zur Laufzeit hat.
- ▶ Dann testet sie mit dem **instanceof**-Operator, ob das Argumentobjekt zu einer Subklasse dieses dynamischen Typs gehört.
- ▶ Die Hilfsmethode `reallysame (Date that)` führt denselben Test in **umgekehrter** Richtung aus, wobei schon sichergestellt ist, dass das Argumentobjekt zu einer Superklasse des Empfängertyps gehört.
- ▶ Nun sind die dynamischen Typen gleich und die Felder können verglichen werden. Die Felder von `that` müssen zunächst durch einen *Typcast* sichtbar gemacht werden.

# Transitive Gleichheit (Implementierung)

## Basisfall

```
class Date {  
    private int day;  
    private int month;  
    private int year;  
  
    protected boolean reallysame (Date that) {  
        return (this.day == that.day) &&  
            (this.month == that.month) &&  
            (this.year == that.year);  
    }  
  
    public boolean same (Date that) {  
        return that.reallysame (this);  
    }  
}
```

# Code für Subklassen

```

class SpecialDate extends Date {
    private int rating;

    SpecialDate (int day, int month, int year, int rating) {
        super (day, month, year);
        this.rating = rating;
    }
    // dynamic type of that is a supertype of type of this
    protected boolean reallysame (Date that) {
        return (that instanceof SpecialDate)
            && super.reallysame (that)
            && (this.rating == ((SpecialDate)that).rating);
    }

    public boolean same (Date that) {
        return (that instanceof SpecialDate)
            && that.reallysame (this);
    }
}

```

# 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

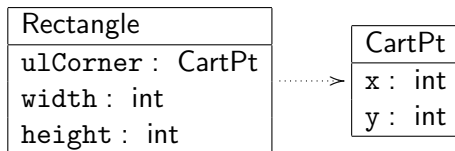
(B)d // statisch ok, dynamischer Fehler

(A)d // statisch ok, dynamisch ok



# Gleichheit für zusammengesetzte Objekte

## Beispiel



```
boolean same (Rectangle that) {  
    return (this.x == that.x) && (this.y == that.y)  
        && (this.ulCorner.same (that.ulCorner));  
}
```

- Rufe die Gleichheit auf den untergeordneten Objekten auf.

# Gleichheit für Vereinigungen von Klassen

- ▶ Definiere die `same()` Methode im Interface.
- ▶ Verwende die Vorgehensweise für Vererbung.
- ▶ Für abstrakte Klassen reicht es, die Methoden `same` und `reallysame` abstrakt zu belassen (da niemals Objekte existieren können, die diesen Klassentyp als Laufzeittyp besitzen).

# Alternative Lösung

Ohne Verwendung von **instanceof** und Typcast

Am Beispiel von IShape:

- ▶ Voraussetzung: alle Varianten sind bekannt.
- ▶ Erweitere das Interface um Methoden, die die jeweilige Variante erkennen und ggf. ein IShape Objekt in ein Objekt vom spezifischen Typ umwandeln. Die Methoden liefern `null`, falls die Umwandlung nicht möglich ist.

```
interface IShape {  
    Dot toDot();  
    Square toSquare();  
    Circle toCircle();  
  
    boolean same (IShape that);  
}
```

# Alternative Lösung

## Die abstrakte Klasse

```
abstract class AShape implements IShape {  
    Dot toDot () { return null; }  
    Square toSquare () { return null; }  
    Circle toCircle () { return null; }  
  
    abstract boolean same (IShape that);  
}
```

# Alternative Lösung

## Implementierung für Dot

```
class Dot implements IShape {  
    Dot toDot () { return this; }  
  
    boolean same (IShape that) {  
        Dot thatDot = that.toDot();  
        return (thatDot != null)  
            && (this.loc.same (thatDot.loc));  
    }  
}
```

# Intensionale Gleichheit

- ▶ **Extensionale Gleichheit** testet ob zwei Objekte gleich sind und sich gleich verhalten.
- ▶ Diese Aufgabe hat in Java die `equals` Methode.
- ▶ **Intensionale Gleichheit** testet ob ihre Argumente dasselbe Objekt bezeichnen, in dem Sinn, dass eine Änderung am einen Argument immer dieselbe Änderung am anderen Argument bewirkt.
- ▶ Diese Aufgabe hat in Java der `==` Operator. (Er testet die Gleichheit von Referenzen.)

## equals

- ▶ In Java vordefiniert:

```
class Object {  
    public boolean equals (Object other) { return this == other; }  
}
```

- ▶ Jede Klasse ist automatisch Subklasse von Object und erbt diese (meist so nicht gewünschte) Implementierung.
- ▶ equals muss überschrieben werden.
- ▶ Die Implementierung von equals muss eine Äquivalenzrelation sein.
- ▶ Beispiel:

```
class A {  
    public boolean equals (Object other) {  
        if (!other instanceof A) return false;  
        A aother = (A)other;  
        // recursively compare fields  
    }  
}
```

# Vergleichen

```
interface Comparable<T> {  
    // compares this object with the other  
    public int compareTo (T other);  
}
```

- ▶ Ergebnis:
  - ▶  $< 0$ , falls **this** kleiner als that
  - ▶  $= 0$ , falls **this** gleich that
  - ▶  $> 0$ , falls **this** größer als that
- ▶ Diese Ordnung ist die **natürliche Ordnung** auf T
- ▶ Muss eine totale Ordnung implementieren



# Beispiel: Vergleichen

- ▶ Vordefiniert: Integer **implements** Comparable<Integer>

```
Integer int0 = 0;  
Integer int1 = 1;  
assert int0.compareTo (int1) < 0;
```

- ▶ Vordefiniert: String **implements** Comparable<String>

```
String str0 = "zero";  
String str1 = "one";  
assert str0.compareTo (str1) > 0;
```

## Beispiele: Nicht vergleichbar

- ▶ Sinnlose Vergleiche werden abgefangen

```
Integer i = 0;  
String s = "one";  
assert i.compareTo (s) < 0; // type error
```

- ▶ Manche Vergleiche werden nicht unterstützt

```
Number m = new Integer(2);  
Number n = new Double (3.14);  
assert m.compareTo (n) < 0; // type error
```

# Anforderungen an compareTo()

## Konsistenz mit equals()

Für zwei Objekte  $x$  und  $y$  muss gelten

*$x.equals(y)$  genau dann, wenn  $x.compareTo(y) == 0$*

- ▶ Verwendung mit SortedMap oder SortedSet: Wenn zwei Objekte, die nach compareTo gleich sind, in SortedSet / SortedMap eingefügt werden, so ist nachher nur eins der Objekte enthalten.
- ▶ Konsistenz ist in den Standardbibliotheksklassen gesichert.
- ▶ Ausnahme: Vergleiche von BigDecimal ignorieren die Zahl der Stellen
- ▶ Vergleich mit null
  - ▶  $x.equals(null) == false$
  - ▶  $x.compareTo(null) \Rightarrow NullPointerException$

# Anforderungen an compareTo()

## Eigenschaften

- ▶ `java.lang.Integer.signum(x)` liefert  $-1$ ,  $0$  oder  $1$  je nachdem, ob  $x$  negativ, null oder positiv ist
- ▶ `compareTo` ist antisymmetrisch

$$\text{signum}(x.\text{compareTo}(y)) == -\text{signum}(y.\text{compareTo}(x))$$

- ▶ Vergleichen ist transitiv

*Falls  $x.\text{compareTo}(y) \leq 0$  und  $y.\text{compareTo}(z) \leq 0$ , dann ist  $x.\text{compareTo}(z) \leq 0$*

- ▶ Vergleichen ist eine Kongruenz

*Falls  $x.\text{compareTo}(y) == 0$ , dann ist  $\text{signum}(x.\text{compareTo}(z)) == \text{signum}(y.\text{compareTo}(z))$*

- ▶ Vergleichen ist reflexiv

$$x.\text{compareTo}(x) == 0$$

## `equals()` und `hashCode()`

- ▶ Methoden aus `java.lang.Object`
- ▶ **public** `int hashCode()`  
muss eine Zahl liefern, die
  - ▶ gleich bleibt, solange keine Felder verändert werden, von denen `equals()` abhängt
  - ▶ gleich ist für Objekte, die `equal()` sind, d.h.

*Falls `x.equal(y)`, dann `x.hashCode() == y.hashCode()`*

- ▶ Die Umkehrung ist nicht erforderlich (meist auch nicht möglich), kann aber die Effizienz von Hashtabellen verbessern.

⇒ Wenn `equals()` überschrieben wird, dann muss auch `hashCode()` überschrieben werden.

# Maximum einer Collection

# Erste Implementierung

```
// maximum of a non-empty collection  
public static <T extends Comparable<T>> T max(Collection<T> coll) {  
    T candidate = coll.iterator().next();  
    for (T elem : coll) {  
        if (candidate.compareTo(elem) < 0) {  
            candidate = elem;  
        }  
    }  
    return candidate;  
}
```

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

# Beispiele

## ► Integer

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

## ► String

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

## ► **Nicht** für Number

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



## Verbesserte Signatur

- ▶ Ausgangspunkt

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

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

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

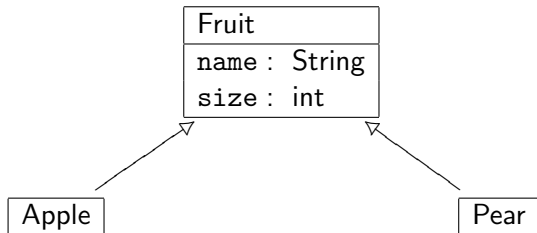
- ▶ Die Vergleichsoperation könnte auch auf einem Supertyp von T definiert sein.

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

- ▶ Definition in Java-Bibliothek 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

# Vergleich von Äpfel und Birnen erlaubt

```

abstract class Fruit implements Comparable<Fruit> {
    protected String name;
    protected int size;
    protected Fruit (String name, int size) {
        this.name = name; this.size = size;
    }
    public boolean equals (Object o) {
        if (o instanceof Fruit) {
            Fruit that = (Fruit) o;
            return this.name.equals (that.name) && this.size == that.size;
        } else { return false; }
    }
    public int hashCode() {
        return name.hashCode()*29 + size;
    }
    public int compareTo (Fruit that) {
        return this.size < that.size ? -1 :
            this.size > that.size ? 1 : 0;
    }
}

```

# Äpfel und Birnen

```
class Apple extends Fruit {  
    public Apple (int size) {  
        super ("Apple", size);  
    }  
}
```

```
class Pear extends Fruit {  
    public Pear (int size) {  
        super ("Pear", size);  
    }  
}
```

# Test mit Vergleichen

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

## Einschub: Signatur von `max`

- ▶ Die allgemeine Signatur von `max` war

```
public static <T extends Comparable<? super T>>  
    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>`

# Vergleich von Äpfeln mit Birnen nicht erlaubt

```
abstract class Fruit1 {  
    protected String name;  
    protected int size;  
    protected Fruit1 (String name, int size) {  
        this.name = name; this.size = size;  
    }  
    public boolean equals (Object o) {  
        if (o instanceof Fruit1) {  
            Fruit1 that = (Fruit1) o;  
            return this.name.equals (that.name) && this.size == that.size;  
        } else { return false; }  
    }  
    public int hashCode() {  
        return name.hashCode()*29 + size;  
    }  
    protected int compareTo (Fruit1 that) {  
        return this.size < that.size ? -1 :  
            this.size > that.size ? 1 : 0;  
    }  
}
```



# Äpfel und Birnen

```
class Apple1 extends Fruit1 implements Comparable<Apple1> {  
    public Apple1 (int size) {  
        super ("Apple", size);  
    }  
    public int compareTo (Apple1 a) {  
        return super.compareTo(a);  
    }  
}
```

```
class Pear1 extends Fruit1 implements Comparable<Pear1> {  
    public Pear1 (int size) {  
        super ("Pear", size);  
    }  
    public int compareTo (Pear1 that) {  
        return super.compareTo (that);  
    }  
}
```

# Test mit Vergleichen

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

# Einschub: Innere und anonyme Klassen

# Innere Klassen

- ▶ Eine innere Klasse wird innerhalb einer anderen Klasse deklariert.
- ▶ Anwendung:
  - ▶ Strukturierung von Code
  - ▶ Kapselung von Hilfsklassen
- ▶ Innere Klassen können selbst mit Sichtbarkeitsattributen versehen werden. (Oft **private**)
- ▶ Innere Klassen können auch erben, Interfaces implementieren und generisch sein.
- ▶ **Achtung:** eine innere Klasse übernimmt **nicht** die Superklassen und Interfaces der umschließenden Klasse!

# Statische innere Klassen

```
class DeeplyNested {  
    private static int foo = 42;  
    private static class A {  
        private static int a;  
        private static void testA() {  
            a = foo;  
        }  
    }  
    private static class OuterB {  
        private static class InnerB {  
            private static int testB() {  
                return A.a;  
            }  
        }  
    }  
    public int testDeeplyNested () {  
        A .testA();  
        return OuterB.InnerB.testB();  
    }  
}
```

# Statische innere Klassen

- ▶ Statische innere Klassen können nur auf statische Elemente anderer Klassen zugreifen.
- ▶ Sie können selbst aber auch *nicht-statische* Elemente enthalten.
- ▶ Sichtbarkeit
  - ▶ Äußere Klassen können auf private Elemente in inneren Klassen zugreifen.
  - ▶ Innere Klassen können auf private Elemente in äußeren Klassen zugreifen.
  - ▶ Innere Klassen können auf private Elemente anderer innerer Klassen mit gleicher äußerer Klasse zugreifen.

# Verkettete Listen

```

class LinkedList {
    private ListElem header = new ListElem(null,null,null);
    private int size = 0;
    LinkedList() {
        header.next = header;
        header.prev = header;
    }
    ...
    private static class ListElem {
        private Entry entry;
        private ListElem next;
        private ListElem prev;
        ListElem(Entry entry,ListElem next,ListElem prev) {
            this.entry = entry;
            this.next = next;
            this.prev = prev;
        }
    }
}

```

## ... mit Iterator

```
class LinkedList {...  
    private int size = 0;  
    private static class ListElem {...}  
  
    private class ListIterator implements Iterator {  
        private int nextIndex = 0;  
        private ListElem next = header.next;  
        public boolean hasNext() {  
            return nextIndex != size;  
        }  
        public Entry next() {...}  
        public void remove() {...}  
    }  
  
    public Iterator iterator() {  
        return new ListIterator();  
    }  
}
```



## Nicht-statische innere Klassen

- ▶ Jede Instanz einer nicht-statischen inneren Klasse hält implizit eine Referenz auf das dazugehörige Objekt der äußeren Klasse.
- ▶ In der Methode `hasNext` wird auf die nicht statische Instanzvariable `size` zugegriffen. Deshalb kann die Klasse `ListIterator` nicht statisch sein.
- ▶ Auf das Objekt der äußeren Klasse `LinkedList` kann mit `LinkedList.this` aus der Klasse `ListIterator` heraus zugegriffen werden.
- ▶ Falls keine Namenskonflikte existieren, kann direkt auf Attribute und Methoden des äußeren Objekts zugegriffen werden.
- ▶ Ein `ListIterator`-Objekt kann für ein `LinkedList`-Objekt `myList` durch `myList.new ListIterator()` erzeugt werden. (hier muss die innere Klasse sichtbar sein!)

# Anonyme Klassen

- ▶ Anonyme Klasse = Klasse ohne Namen
- ▶ Abkürzung für innere Klasse, die nur einmal benötigt wird.
- ▶ Beispiele
  - ▶ Listener für Swing-Objekte
  - ▶ Filter für Listenfilterung
  - ▶ Vergleichsoperatoren
- ▶ Erzeugung eines Objekts durch den **Ausdruck**

```
new Interface () {  
    // Felder und Methoden zur Implementierung von Interface  
    // genau wie im Rumpf einer Klassendeklaration  
};
```

- ▶ Alternativ eine Instanz einer abstrakten Klasse

```
new AClass (arg1,..., argn) {  
    // Felder und Methoden  
};
```

# Andere Ordnungen

# Alternative Vergleiche

- ▶ Die natürliche Ordnung ist manchmal nicht die richtige.
- ▶ Beispiel:
  - ▶ Früchte dem Namen nach vergleichen
  - ▶ Strings der Länge nach vergleichen
- ▶ Java stellt dafür das `Comparator` Interface bereit.

# Comparator

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

- ▶ `compare (x,y)` liefert
  - ▶  $< 0$ , falls  $x$  kleiner als  $y$
  - ▶  $= 0$ , falls  $x$  gleich  $y$
  - ▶  $> 0$ , falls  $x$  größer als  $y$

## Beispiel: Strings zuerst der Länge nach vergleichen

```
class SizeOrder implements Comparator<String> {  
    public SizeOrder () {}  
    public int compare (String x, String y) {  
        return x.length() < y.length() ? -1 :  
            x.length() > y.length() ? 1 :  
            x.compareTo(y);  
    }  
}
```

### Verwendung

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

## Beispiel nochmal mit anonymer Klasse

```
Comparator<String> sizeOrder =  
new Comparator<String> () {  
    public int compare (String x, String y) {  
        return x.length() < y.length() ? -1 :  
            x.length() > y.length() ? 1 :  
            x.compareTo(y);  
    }  
};
```

### Verwendung

```
assert "two".compareTo("three") > 0;  
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

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

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

- ▶ Analog für min



## Beispiele mit Comparator

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

# Maximum mit Comparator

```
public static <T>
T max(Collection <? extends T> coll, Comparator<? super T> comp) {
    Iterator<? extends T> iter = coll.iterator();
    T candidate = iter.next();
    while(iter.hasNext()) {
        T elem = iter.next();
        if (comp.compare(candidate, elem) < 0) {
            candidate = elem;
        }
    }
    return candidate;
}
```

# Natürliche Ordnung als Comparator

Mit anonymer Klasse

```
public static <T extends Comparable<? super T>>  
Comparator<T> naturalOrder() {  
    return  
    new Comparator<T> () {  
        public int compare(T t1, T t2) {  
            return t1.compareTo(t2);  
        }  
    };  
}
```

# Umdrehen der Ordnung

Mit anonymer Klasse

```
public static <T>
Comparator<T> reverseOrder(final Comparator<T> comp) {
    return
    new Comparator<T> () {
        public int compare(T t1, T t2) {
            return comp.compare(t2, t1);
            // alternativ:
            // return -comp.compare(t1, t2);
        }
    };
}
```

# Implementierung des Minimums

## ► Mit Comparable:

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

## ► Mit Comparator:

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

# Lexikographisches Vergleichen für Listen aus Elementvergleich

```

public static <T>
Comparator<List<T>> listComparator(final Comparator<T> comp) {
    return new Comparator<List<T>>() {
        public int compare(List<T> l1, List<T> l2) {
            int n1 = l1.size();
            int n2 = l2.size();
            for(int i = 0; i < Math.min(n1, n2); i++) {
                int k = comp.compare(l1.get(i), l2.get(i));
                if (k != 0) {
                    return k;
                }
            }
            return n1 < n2 ? -1 :
            n1 == n2 ? 0 : 1;
        }
    };
}

```

# Aufzählungstypen (Enumerated Types)

# Aufzählungstypen in Java 5

- ▶ Ein Aufzählungstyp enthält endlich viele benannte Elemente.
- ▶ Beispiel

```
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

```
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

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E> {  
    private final String name;  
    private final int ordinal;  
    protected Enum (String name, int ordinal) {  
        this.name = name; this.ordinal = ordinal;  
    }  
    public final String name() { return name; }  
    public final int ordinal() { return ordinal; }  
    public String toString() { return name; }  
    public final int compareTo(E o) {  
        return ordinal - o.ordinal;  
    }  
}
```

# Die Klasse Season

```
// corresponding to
// enum Season { WINTER, SPRING, SUMMER, AUTUMN }
final class Season extends Enum<Season> {
    private Season(String name, int ordinal) { super(name, ordinal); }
    public static final Season WINTER = new Season ("WINTER", 0);
    public static final Season SPRING = new Season ("SPRING", 1);
    public static final Season SUMMER = new Season ("SUMMER", 2);
    public static final Season AUTUMN = new Season ("AUTUMN", 3);
    private static final Season[] VALUES = {WINTER, SPRING, SUMMER, AUTUMN};
    public static Season[] values() { return VALUES.clone(); }
    public static Season valueOf (String name) {
        for (Season e : VALUES) {
            if (e.name().equals (name)) { return e; }
        }
        throw new IllegalArgumentException();
    }
}
```

# Erklärung für die Typschränken

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

```
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 Typschränken

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

```
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>
- ⇒ Elemente von Season miteinander vergleichbar, **aber nicht** mit Elementen von anderen Aufzählungstypen!

## So tut's nicht: zu einfach

- ▶ Ohne die Typschränken 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!**