Objekt-Vergleich mittels Comparatoren
Wie, wann und warum implementiert man die compareTo()-Methode?

JavaSPEKTRUM, Juli 2002
Klaus Kreft & Angelika Langer

Vorbemerkung

In dieser Artikelserie haben wir uns bisher mit Basisfunktionalität beschäftigt, die jede Klasse in Java zur Verfügung stellt.Dabei haben wir uns eingehend mit der Implementierung der Methode equals() befasst. In dieser Ausgabe wollen wir dort anknüpfen und uns die compareTo() Methode näher ansehen.Die Methode compareTo() hängt eng mit der Methode equals() zusammen, da beide Methoden Aussagen zur Gleichheit von Objekten liefern und deshalb konsistent zueinander implementiert werden sollten. In dieser Ausgabe werden wir uns ansehen, was diese Konsistenz-Forderung genau bedeutet, wofür man compareTo() überhaupt braucht, und worauf man achten muss, wenn man es korrekt implementieren will.

Wofür braucht man compareTo() ?

Die Methode compareTo() ist nützlich, wenn Objekte eines Typs in einem baum-basierten Container wie java.util.TreeSet oder java.util.TreeMap abgelegt werden sollen oder wenn Sequenzen solcher Objekte (mit Hilfe von Sortiermethoden wie Arrays.sort()) sortiert werden sollen. Für beide Anwendungsfälle braucht man eine Vergleichsfunktion, die eine Sortierreihenfolge liefert. Beim Sortieren ist das offensichtlich. Bei den Containern liegt es daran, dass baum-basierte Container ihre Objekte immer in sortierter Reihefolge halten.Die compareTo()-Methode definiert das benötigte Sortierkriterium: sie liefert die Aussage, ob das this-Objekt grösser, kleiner oder gleich einem anderen Objekt ist.compareTo()ist also eine Vergleichsfunktion, die insbesondere auch den Fall den Gleichheit abdeckt und deshalb in enger Beziehung zur equals()-Methode steht, die ebenfalls auf Gleichheit zweier Objekte prüft.

Wir haben uns in vorangegangenen Artikeln mit den Methoden equals() und hashCode() beschäftigt. Diese beiden Methoden werden gebraucht, wenn Objekte in einem hash-basierten Container verwaltet werden sollen. Baum-basierte Container sind intern ganz anders organisiert als hash-basierte Container, aber auch sie stellen gewisse Anforderungen an die Objekte, die sie verwalten können. Für die hash-basierten Container war die Anforderung: equals() und hashCode() müssen zur Verfügung stehen, und zwar in konsistenter und korrekter Form. Die genauen Anforderungen sind formal im equals()-Contract und hashCode()-Contract spezifiziert. Für die baum-basierten Container ist das ganz ähnlich.

Die Anforderung an Objekte, die in einem baum-basierten abgelegt werden können, besteht darin, dass für die Objekte eine Ordnung definiert sein muss. Diese Ordnung kann auf zwei Wegen zur Verfügung gestellt werden:

- Die im Container zu speichernden Objekte müssen das Interface Comparable implementieren und eine Methode compareTo() zur Verfügung stellen.Hier ist Interface Comparable:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- Es gibt einen Comparator, der Objekte, die im Container abgelegt werden sollen, vergleichen kann. Ein Comparator ist Instanz einer Klasse, welche das Comparator-Interface implementiert und Methoden equals() und compare() zur Verfügung stellt.Hier ist Interface Comparator:

```
public interface Comparator {
    int compare(Object o1, Object o2);
    boolean equals(Object o);
} i
```

Im ersten Fall ist die Vergleichsfunktionalität Bestandteil der Funktionalität der Klasse des Objekts, im zweiten Fall ist der Vergleich nicht in der Klasse, sondern extern im Comparator implementiert. Logisch sind diese beiden Varianten aber gleichwertig.Wir werden im Folgenden nur noch die Implementierung der compareTo()-Methode besprechen; alle Gesagte gilt dann analog auch für die compare()-Methode eines entsprechenden Comparators.

Wie man schon sehen kann, ist die compareTo()-Methode einer Klasse im Vergleich zur equals()- Methode von einer etwas anderen Qualität. equals() ist bereits in der Superklasse aller Klassen, nämlich Object, definiert und implementiert. Damit haben alle Objekte in Java eine equals()-Methode und man muss sich immer Gedanken darüber machen, ob das von Object geerbte equals() korrekt ist oder man es für eine neue Klasse nicht besser überschreiben sollte. Das ist für compareTo() deutlich anders. compareTo() ist nicht in Object definiert, sondern im Comparable-Interface.Das bedeutet, das nicht alle Java-Objekte automatisch "comparable" sind.Wenn eine Klasse das Comparable-Interface nicht implementiert, dann ist auch kein Malheur. Man kann immer noch einen entsprechenden Comparator definieren, der die Funktionalität des Objekt-Vergleichs in seiner compare()-Methode implementiert..Wenn es auch den nicht gibt, dann kann man solche Objekte eben nicht in baum-basierten Containern ablegen. Ausserdem kann man Sequenzen solcher Objekte nicht sortieren.Objekte ohne compareTo() und ohne Comparator sind per definitionem nicht vergleichbar und damit weder sortierbar noch in einem baum-basierten Container ablegbar.

Das kann von der Semantik der Klasse her völlig korrekt sein; nicht alle Objekte per se sind sortierbar. Es kann auch vorkommen, dass die Objekte zwar sortierbar wären, aber diese Funktionalität einfach nicht gebraucht wird in einer Applikation. Anders als bei equals(), wo man sich auf jeden Fall Gedanken machen muss, kann man die Problematik "compareTo()" bei Bedarf vernachlässigen. Es gibt ohne compareTo() gewisse Einschränkungen, aber keine fiesen

Fehler. Wenn man compareTo() aber implementieren will, dann sollte man es richtig machen.Wie das geht und worauf man achten muss, sehen wir uns im Folgenden näher an.

Baum-basierte Container in Java

Wie sehen baum-basierte Container aus und wofür brauchen sie die compareTo()-Methode?

Baum-basierte Container sind Datenstrukturen, die ihre Elemente in Knoten ablegen. All diese Knoten verweisen aufeinander, so dass der Container aus einem Geflecht von miteinander verbundenen Knoten besteht. Die Verweise sind so organisiert, dass ein Binär-Baum entsteht: in einem Binär-Buam verweist jeder Knoten auf seinen übergeordneten Knoten (parent) und auf zwei untergeordnete Knoten (children) (deshalb "Binär"-Baum). Die beiden Kind-Knoten sind so angelegt, dass das Element im linken Kind-Knoten kleiner und das Element im rechten Kind-Knoten grösser als das Element im eigenen Knoten sind. Auf diese Weise entsteht eine Sortierreihenfolge; es gibt einen Navigationsalgorithmus, der vom kleinsten Knoten über den jeweils nächst-grösseren Knoten bis zu grössten Knoten alle Elemente des Containers zugänglich macht. Damit sind alle Elemente in einem baum-basierten Container in sortierter Reihenfolge zugreifbar.

Baum-basierte Container gehören zu den Standard-Datenstrukturen in der Informatik und sind in der entsprechenden Standardliteratur über Datenstrukturen und Algorithmen beschrieben (siehe zum Beispiel /KNU/ oder /SED/). Hier ein kurzer Abriss über die wesentlichen Elemente; siehe auch Abbildung 1, welche den logischen Aufbau eines baum-basierten Containers zeigt.

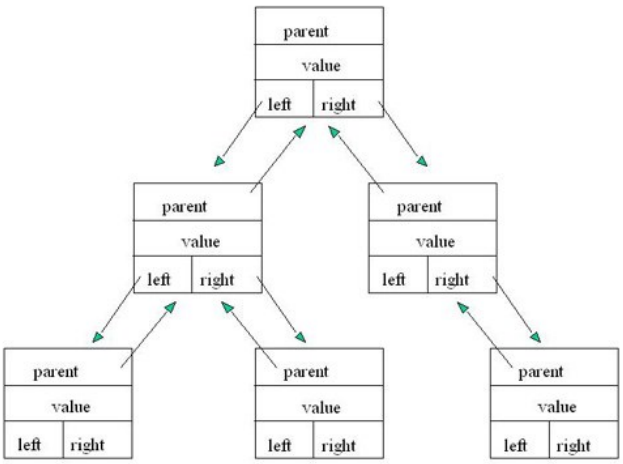

Abbildung 1: Interne Binär-Baum-Struktur eines baum-basierten Containers

Die Sortierreihenfolge ist eines der charakterisierenden Merkmale eines baum-basierten Containers.Wenn man daran interessiert ist, dass die Elemente in einer Sequenz immer in sortierter Reihenfolge verfügbar sind, dann ist ein baum-basierter Container ideal. Der Container sorgt selbständig dafür, dass neue Elemente immer an der "richtigen" Stelle in den Baum eingefügt werden, d.h.an der Stelle, an die das neue Element gemäss seines Inhalts in der Sortierreihenfolge gehört.Und hier kommt die compareTo()-Methode ins Spiel: für die

Entscheidung über die richtige Position eines Elements braucht ein baum-basierter Java-Container entweder die compareTo()-Methode des Elements oder einen äquivalenten Comparator.

Baum-basierte Container haben, über die Sortierreihenfolge hinaus, die Eigenschaft, dass der Zugriff auf die Elemente im Container in logarithmischer Zeit erfolgt. Zumindest ist das so, wenn der Binär-Baum balanciert ist, das heisst, wenn alle Äste in etwa gleich lang sind. Das ist normalerweise der Fall; baum-basierte Container balancieren ihre Binär-Bäume automatisch aus, wenn diese aus dem Gleichgewicht geraten sind. Verglichen mit einem hash-basierten Container sind die logarithmischen Zugriffszeiten sehr zuverlässig. Bei einem hash-basierten Container variiert die Güte der Zugriffszeiten mit der Güte der Hash-Code-Berechnung: sie kann bei guter Verteilung fast konstant sein, was viel besser als der logarithmische Zugriff ist, aber auch linear bei ungünstiger Verteilung, was sehr viel schlechter als der logarithmische Zugriff ist.

Man sieht also, dass die baum-basierten Java-Container sowohl für das Navigieren im Baum als auch für die Suche nach vorhandenen Elementen im Container oder das Einfügen von neuen Elementen in den Container entweder die compareTo()-Methode des Elements oder einen äquivalenten Comparator brauchen. Nehmen wir also einmal an, dass wir eine Klasse "comparable" machen wollen, damit wir Instanzen dieser Klasse in baum-basierten Container ablegen können. Was genau muss eine Implementierung von compareTo() leisten, damit der baum-basierte Container funktioniert? Die geforderten Eigenschaften sind im sogenannten Comparator-Contract festgelegt.

Der Comparator-Contract

Die Anforderungen an einen Comparator bzw.eine compareTo()-Methode findet man in der JavaDoc der Java 2 Standard Edition (J2SE) unter dem Eintrag Comparable.compareTo oder Comparator.compare.Hier ist der Originaltext:

```
public int compareTo(Object o)
```

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.

- In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.The implementer must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y.(This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)
- The implementer must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.
- Finally, the implementer must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

- It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)).Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact.The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Parameters: o - the Object to be compared.

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

Throws: ClassCastException - if the specified object's type prevents it from being compared to this Object.

Das bedeutet das Folgende:

1. compareTo() muss eine der folgenden Antworten geben:
    a. negativer Returnwert falls this kleiner als other
    b. 0 falls this gleich other
    c. positiver Returnwert falls this grösser als other
    d. ClassClastException falls this und other nicht vergleichbar sind
2. Symmetrie: Wenn x kleiner als y ist, dann muss y grösser als x sein und umgekehrt.
3. Transitivität: Wenn x grösser als y und y grösser als z, dann muss auch x grösser als z sein.
4. Wenn x und y gleich sind, dann liefert der Vergleich von z mit x dasselbe Ergebnis wie der Vergleich von z mit y, d.h.wenn z grösser als x ist, dann ist es auch grösser als y, usw.
5. Konsistenz zu equals(): es wird empfohlen, dass Objekte, die gemäss equals()gleich sind auch gemäss compareTo() gleich sein sollten, und umgekehrt.

Hier sieht man deutlich die enge Beziehung zwischen equals() und compareTo().Es ist zwar nicht verlangt, dass equals() und compareTo() dieselbe Gleichheitsbeziehung liefern, aber es wird dringend empfohlen. Sehen wir uns einmal an, was passiert, wenn diese Forderung verletzt ist, und unter welchen Umständen sie überhaupt verletzt wird.

Konsistenz zwischen compareTo() und equals()

Ein Beispiel für eine Ordnungsrelation, die nicht konsistent zu equals() ist, ist die compareToIgnoreCase()-Methode der String-Klasse.Diese Methode liefert Gleichheit für Strings, die sich allein in der Gross-/Kleinschreibung unterscheiden. Beispiel:

```
String a = new String("abc");
String b = new String("ABC");


a.compareToIgnoreCase(b));    // yields: 0
a.equals(b));                 // yields: false
```

Ein Comparator auf der Basis von compareToIgnoreCase() wäre inkonsistent zu equals(), weil die Strings "abc" und "ABC" bzgl.equals() verschieden sind, aber bzgl.des Comparators aber gleich sind. (Die String-Klasse hat auch eine compareTo()-Methode, die konsistent zu equals() ist, aber lassen wir im Moment mal ausser Acht; wir wollen ja gerade den inkonsistenten Fall betrachten.)

Was passiert, wenn eine solche inkonsistente Vergleichsfunktionalität verwendet? Man wird zum Beispiel beobachten, dass man in einem TreeSet nur einen der beiden Strings "abc" und "ABC" ablegen kann, weil ein TreeSet keine Duplikate erlaubt und "abc" bzgl.compareToIgnoreCase() ein Duplikat von "ABC" wäre. Ausserdem würde man auf die Frage, ob "ABC" im Container enthalten ist, die Antwort "ja" bekommen, auch wenn "ABC" gar nicht enthalten ist, sondern nur eines seiner Duplikate wie "abc" oder "aBc" oder "abC". Beispiel:

```
class CaseInsensitiveStringComparator implements Comparator {
     public int compare(Object o1, Object o2) {
        return ((String)o1).compareToIgnoreCase((String)o2);
     }
     ...
}

Set s = new TreeSet(new Test.CaseInsensitiveStringComparator());
s.add(new String("abc"));               // yields: true
s.add(new String("ABC"));               // yields: false
Iterator i = s.iterator();
while (i.hasNext())
    System.out.println(i.next());  // set contains only "abc"
s.contains(new String("ABC"));         // yields: true
```

Die Effekte der Inkonsistenz zwischen compareTo() (bzw.Comparator) und equals() sind vielleicht verwirrend, aber es ist dennoch garantiert, dass die baum-basierten Container trotz Inkonsistenz funktionieren. Deshalb ist die Konsistenz empfohlen, aber nicht verlangt.

Ein anderes Beispiel für einen Vergleich, der inkonsistent zu equals() wäre das Folgende: Betrachten wir eine Klasse Name, bestehend aus Vor- und Zuname. Wenn in equals() sowohl Vor- als auch Zuname in den Vergleich eingehen, aber bei compareTo() nur der Nachname berücksichtigt wird, dann sind compareTo() und equals() inkonsistent. Dann wäre nämlich Peter Müller und Thomas Müller ungleich bzgl.equals(), aber gleich bzgl.compareTo(). Für die Konsistenz ist es erforderlich, dass in die Implementierung von compareTo() genau die Information eingeht, die auch zum Ergebnis von equals() beiträgt, und umgekehrt.

Wenn Klassen eine zu equals() inkonsistente Implementierung des Vergleichs liefern, dann sollte man das unbedingt in der Dokumentation der Klasse beschreiben. Der JDK schlägt sogar einen Wortlauf vor, der in der JavaDoc-Beschreibung der Klasse zu verwenden sei: "Note: this class has a natural ordering that is inconsistent with equals." (zu finden in /JDOC/ unter Comparable.compareTo() und Comparator.compare()).

Implementierung von compareTo()

Worauf muss man nun achten, wenn man compareTo() implementiert? Eigentlich muss man ziemlich genau dasselbe beachten wie für die Implementierung von equals(). Man wird im Prinzip Feld für Feld vergleichen und daraus das Gesamtergebnis bestimmen.

Signatur

Zur Signatur von compareTo() ist eigentlich nicht viel zu sagen. Sie ist im Interface Comparable festgelegt. Objekte, die man in einem TreeSet oder einer TreeMap ablegen will müssen das Comparable-Interface implementieren.Andernfalls gibt es eine ClassCastException zur Laufzeit. Also wird man die compareTo()-Methode mit der Signatur implementieren, die im Comparable-Interface verlangt wird:

```
public int compareTo(Object o)
```

Die Alternative zum Comparable-Interface ist der Comparator. Wenn die Element nicht das Comparable-Interface implementieren, dann gibt man bei der Konstruktion des Containers einen Comparator mit, dessen compare()-Methode dann statt der compareTo()-Methode der Elemente verwendet wird.

Interessanterweise findet man im JDK eine Reihe von Klassen, die nicht nur die vom Comparable-Interface verlangte compareTo()-Methode haben, sondern noch eine oder mehrere überladene Versionen von compareTo() haben. Die Klasse java.util.Date ist ein Beispiel dafür.

```
public class Date implements java.io.Serializable, Cloneable,
Comparable {
    ...
  public int compareTo(Date anotherDate) {
    long thisTime = this.getTime();
    long anotherTime = anotherDate.getTime();
    return (thisTime<anotherTime ? -1 : (thisTime==anotherTime ? 0 :
1));
  }
  public int compareTo(Object o) {
    return compareTo((Date)o);
  }
}
```

Diese Implementierungstechnik hat den Vorteil, dass man sich Implementierung von int compareTo(Date) den Downcast von Object auf Date sparen kann, weil diese Methode nur gerufen wird, wenn das Argument ein Date oder eine Subklasse von Date ist. Und bei der Subklasse liegt der Hase im Pfeffer. Die Date-Klasse ist nicht als final deklariert und ist damit eine potentielle Superklasse. Wenn der Autor einer Subklasse von Date nun seinerseits die Subklassen-Objekte Comparable machen will, dann genügt es nicht mehr, nur die vom Comparable-Interface verlangte Version von int compareTo(Object) zu implementieren, sondern dann muss man sämtliche in der Superklasse vorhandenen Versionen von compareTo() überschreiben. Andernfalls kann es leicht Übersetzungsfehler wegen "ambiguous method calls" geben.

Sehen wir uns einmal eine Subklasse NamedDate von Date an:

```
public class NamedDate extends java.util.Date {
   private String name;
   NamedDate(String n) {name = n;}
   NamedDate(String n, int year, int month, int date)  {
      super(year,month,date); name=n;
   }
   public int compareTo(Object o) {
     if (super.compareTo(o)==0) {
         return name.compareTo(((NamedDate)o).name);
     }
     return super.compareTo(o);
   }
}
```

Die Subklasse redefiniert die im Comparable-Interface definierte compareTo()-Methode. Der Aufruf von compareTo() zum Zwecke des Vergleichs zweier NamedDate-Objekte führt dann aber zu einem Übersetzungsfehler:

```
NamedDate a = new NamedDate("New Year's Eve",2000,12,31);
NamedDate b = new NamedDate("today");
a.compareTo(b);  // does not compile
```

Die Fehlermeldung lautet: reference to compareTo is ambiguous, both method compareTo(java.util.Date) in java.util.Date and method compareTo(java.lang.Object) in NamedDate match

Der Fehler liesse sich zwar in diesem Beispiel durch einen Cast des Arguments nach Date vermeiden, also durch den Aufruf a.compareTo((Date)b); aber man kann dem Benutzer der Klasse NamedDate nicht zumuten, dass er weiss, zu welchem Typ er die Argumente einer Methode konvertieren muss, damit der Aufruf überhaupt geht. Die oben gezeigte Implementierung ist daher nicht zu empfehlen.

Die Fehlermeldung des Compilers wegen des zweideutigen Methodenaufrufs ist angesichts der Overload-Resolution-Regeln in Java nicht überraschend.Der Java-Compiler muss für den Aufruf von a.compareTo(b) die richtige Version von compareTo() finden. Dazu sucht er zunächst alle Kandidaten zusammen. Als Kandidaten kommen alle compareTo()-Methoden aus der Klasse NamedDate sowie alle zugänglichen compareTo()-Methoden aus sämtlichen Superklassen von NamedDate in Frage. Es gibt also 3 Kandidaten in unserem Beispiel:

- NamedDate.compareTo(Object)
- Date.compareTo(Object) und
- Date.compareTo(Date)

Aus dieser Kandidatenmenge bestimmt der Compiler dann den "besten" Kandidaten. In dieser Bewertung der Kandidaten gehen zwei Kriterien ein:

(1) die Konvertierungen, die für die Argumente der Methode nötig sind

(2) die Konvertierungen, die für das Objekt nötig sind, auf dem die Methode aufgerufen wird

Die genauen Regeln für den Methoden-Aufruf sind in der Sprachbeschreibung (siehe / GOS /) beschrieben und sind schwer zu verstehen. An dieser Stelle daher nur ein ungefährer Einblick:

In unserem Beispiel muss für den ersten Kandidaten NamedDate.compareTo(Object) eine Konvertierung von NamedDate, dem Typ des übergebenen Arguments, nach Object, dem deklarierten Argumenttyp, gemacht werden. Das ist eine Konvertierung des Arguments über zwei Stufen in der Klassenhierarchie hinweg; das Objekt selbst muss nicht konvertiert werden. Für den zweiten Kandidaten Date.compareTo(Object) muss dieselbe Konvertierung für das Argument gemacht werden und der Kandidat ist dazu noch in einem fremden Scope, nämlich der Superklasse Date definiert. Dieser Kandidat ist deshalb schlechter als der erste Kandidat. Der dritte Kandidat Date.compareTo(Date) erfordert eine Typkonvertierung fürs Argument über eine Stufe hinweg, von NamedDate nach Date, und ist in der Superklasse definiert, was eine Konvertierung des Objekts erfordert. Damit ist keiner der Kandidaten 1 und 3 eindeutig besser und der Compiler meldet den zweideutigen Aufruf als Fehler.

Weil zweideutige Methodenaufrufe in Java so schnell zustande kommen, vermeidet man es eigentlich generell, in Klassenhierarchien Overloading (mehrere Versionen derselben Methoden in derselben Klasse) und Overriding (Redefinition von Methoden in abgeleiteten Klassen) zu mischen. Wenn die Superklasse Date nur eine Version von compareTo() hätte, nämlich die vom Comparable-Interface verlangte Version Date.compareTo(Object), dann gäbe es keine Zweideutigkeiten: die Subklassen-Version NamedDate.compareTo(Object) wäre dann in unserem Beispiel eindeutig der beste Kandidat. Um die Zweideutigkeiten zu vermeiden, werden wir nur eine compareTo()-Methode pro Klasse implementieren.

Was macht aber nun, wenn man von einer Superklasse wie Date mit ihren überladenen Versionen von compareTo() ableiten will oder muss? Da bleibt nur die Möglichkeit, sämtliche Versionen von compareTo() zu redefinieren. Wenn man dann ohnehin schon dabei ist, diverse Varianten der Methode zu definieren, dann liegt es nahe, im Stil der Superklasse auch noch eine subklassen-spezifische Variante NamedDate.compareTo(NamedDate) hinzuzufügen, die die eigentliche Funktionalität implementiert und an die die übrigen Versionen delegieren.Damit ist eine Inflation von compareTo()-Methoden vorprogrammiert, die sich natürlich fortsetzt, sobald die Klassenhierarchie wächst und weitere Sub-Subklassen hinzu kommen.

Insgesamt halten wir die Verwendung mehrerer überladener Versionen von compareTo(), wie man sie im JDK findet, für wenig nachahmenswert. Sie bietet keinen nennenswerten Vorteil und führt in Klassenhierarchien zu einer inflationären Vermehrung der überladenen Versionen. Bei final Klassen, wie zum Beispiel der String-Klasse, stört es nicht, aber bei non-final Klassen raten wir davon ab.

Die eigentlich Implementierung der compareTo()-Methode ähnelt der Implementierung von equals().

Alias-Prüfung

Man kann, wie bei der Implementierung von equals(), zwecks Optimierung als erstes prüfen, ob die beiden zu vergleichenden Objekte identisch sind. Identische Objekte sind insbesondere auch gleich, d.h.compareTo() muss 0 zurückgeben, und man kann sich allen weiteren Aufwand sparen.

```
public int compareTo(Object other) {
  if (this == other)
     return 0;
  ...
}
```

Test auf null

Die Prüfung, ob other eine null-Referenz ist, wird bei der Implementierung von compareTo() üblicherweise unterlassen. Statt dessen wird einfach auf other zugegriffen und ggf.eine NullPointerException provoziert.Das ist gängige Praxis so und der Comparator-Contract sagt auch nichts dazu, wie null-Referenzen behandelt werden müssen.Aus der Symmetrie-Anforderung des Comparator-Contracts kann man allerdings indirekt ableiten, dass der Vergleich mit null eine NullPointerException auslösen muss, weil der symmetrische Fall, nämlich null.compareTo(o) eine solche Exception hervorruft.

Bei der Implementierung von equals() ist das anders. Dort muss man den Sonderfall der null-Referenz als Argument von equals() explizit abfangen, weil der equals()-Contract verlangt, dass der Vergleich mit null immer false liefern muss.equals() ist deshalb in der Behandlung von null-Referenzen nicht symmetrisch: o.equals(null) muss false liefern, wohingegen null.equals(o) eine NullPointerException auslöst. compareTo() hingegen ist auch für null-Referenzen symmetrisch und wirft in beiden Fällen eine NullPointerException.

Test auf Vergleichbarkeit

Das Comparable-Interface verlangt, dass man compareTo() mit der Signatur public int compareTo(Object other) implementiert.Das bedeutet, dass other auf Objekte beliebigen Typs verweisen kann.Aus diesem Grunde muss man prüfen, ob other überhaupt mit this vergleichbar ist. Diesen Vergleichbarkeitstest haben wir im Falle von equals() ausführlich diskutiert (siehe / KRE /). Alles dort Gesagte gilt uneingeschränkt auch hier.

Aus Gründen der Konsistenz zu equals() wird man für den Vergleichbarkeitstest in compareTo() dieselbe Technik verwenden, die man auch in equals() verwendet hat. Wir hatten vorgeschlagen, diesen Test mit Hilfe von getClass() zu machen:

```
int compareTo(Object other) {
  ...
  if (other.getClass() != getClass())
    throw new ClassCastException();
  ...
}
```

Als Reaktion im Falle von Unvergleichbarkeit schreibt der Comparator-Contract eine ClassCastException vor. Das ist eine Reaktion, die deutlich verschieden ist von dem Ergebnis,

welches equals() in der gleichen Situation liefert. Die equals()-Methode liefert im Falle von Unvergleichbarkeit false zurück, wohingegen compareTo() in derselben Situation eine unchecked Exception wirft. Dieser kleine Unterschied hat grössere Auswirkung, insbesondere was das Arbeiten mit Containern betrifft.

In einem hash-basierten Java-Container können Objekte beliebigen Typs abgelegt werden. Das heisst, es können auch im selben Container verschiedene Typen von Objekten liegen. Solche heterogenen Elementsequenzen kann mit baum-basierten Java-Containern nicht erreichen. In einem TreeSet beispielsweise können nur Objekte abgelegt werden, die miteinander vergleichbar sind. Beim Versuch, einen "Fremdlings" im TreeSet einzufügen, wird die compareTo()-Methode gerufen, die in diesem Fall der Unvergleichbarkeit eine ClassCastException auslöst. Es wird also nicht gelingen, einen "Fremdling" einzufügen. Das ist beim HashSet anders. Zwar wird auch beim Einfügen des "Fremdlings" die equals()-Methode gerufen, aber sie wirft keine Exception, sondern sagt lediglich, dass der "Fremdling" verschieden ist von Elementen anderen Typs.Das ist kein Problem und der "Fremdling" wird eingefügt und kann später auch wieder im Container gefunden werden.

Das Verhalten von TreeSet und HashSet ist also ganz anders und dieser Unterschied ergibt sich als Nebeneffekt der ClassCastException, die von compareTo() ausgelöst wird.Interessanterweise implementieren sowohl TreeSet als auch HashSet ein gemeinsames Interface, nämlich das Set-Interface. Wenn man gegen dieses Set-Interface programmiert, dann weiss man nicht, ob sich hinter dem Interface ein TreeSet oder ein HashSet verbirgt und man weiss dann auch nicht, ob die add()-Methode mit einer unchecked Exception abbrechen kann oder nicht. Das ist einer der vielleicht nicht so erfreulichen Effekte der unchecked Exceptions in Java: die Methode Set.add() ist so deklariert, dass sie keine Exceptions wirft, zumindest keine checked Exceptions.Dennoch wird die semantisch identische Operation im Falle von HashSet gutartig verlaufen, während sie im Falle von TreeSet mit einem gravierenden Fehler, nämlich mit einer unchecked Exception, scheitert.

Woran liegt es nun, dass man in compareTo() eine Exception auslöst statt eine gutartige Antwort zu liefern? Das liegt einfach an der Semantik des Vergleichs. Es gibt keinen Returnwert, der Unvergleichbarkeit ausdrückt. Man muss sich entscheiden zwischen grösser, kleiner und gleich.Ausserdem soll die Vergleichsrelation auch noch symmetrisch und transitiv sein.Wenn man etwa versuchen würde zu sagen "Alle fremden Objekte sind kleiner als meine Objekte.", dann würde man im Falle der Unvergleichbarkeit einen positiven Wert zurück geben. Mit dieser Strategie gerät man aber in Konflikt mit der Symmetrie-Anforderung. Aus `meinObjekt.compareTo(fremdesObjekt) > 0` müsste folgen `fremdesObjekt.compareTo(meinObjekt) < 0`. Und das stimmt natürlich nicht, wenn alle diese Strategie verfolgen. Deshalb ergeben sich "Inseln der Vergleichbarkeit", das heisst, Mengen von Objekte, die miteinander per compareTo() verglichen werden können. Und nur miteinander vergleichbare Objekte können gemeinsam im selben baum-basierten Container abgelegt werden.

In der oben vorgeschlagenen Implementierung des Vergleichbarkeitstest per getClass() besteht eine "Insel" aus allen Objekte desselben Typ.Ebenso wie bei equals() kann man überlegen, ob man die Vergleichbarkeit auf Sub- und Superklassen ausdehnen möchte. Solche

Implementierungen von compareTo() findet man im JDK. Wir haben bereits ein Beispiel gesehen, nämlich in der Klasse java.util.Date:

```
public class Date implements java.io.Serializable, Cloneable,
Comparable {
    ...
  public int compareTo(Date anotherDate) {
    long thisTime = this.getTime();
    long anotherTime = anotherDate.getTime();
    return (thisTime<anotherTime ? -1 : (thisTime==anotherTime ? 0 :
1));
  }
  public int compareTo(Object o) {
    return compareTo((Date)o);
  }
}
```

In der Implementierung von Date.compareTo(Object) wird ein Downcast von Object nach Date gemacht. Dieser Downcast stellt den Test auf Vergleichbarkeit dar. Er scheitert mit der geforderten ClassCastException, wenn das Argument kein Date ist und auch nicht von einem Typ, der von Date abgeleitet ist. Hier werden also alle Objekte als vergleichbar angesehen, die vom gleichen oder von einem Subtyp sind. Diese Art der Implementierung ähnelt der für equals() oft verwendeten, aber problematischen Technik des Vergleichbarkeitstest per instandeof-Operator. Bei equals() haben wir gesehen, dass Implementierungen von equals() in Klassenhierarchien, die die Vergleichbarkeit per instanceof-Test auf Subtypen ausdehnen, i.a.inkorrekt sind, weil sie intransitiv sind. Genau dasselbe Problem ergibt sich hier bei compareTo(), wenn der Vergleichbarkeitstest per Downcast gemacht wird.

Betrachten wir die oben schon verwendete Subklasse NamedDate:

```
public class NamedDate extends java.util.Date {
     private String name;
     NamedDate(String n) {name = n;}
     NamedDate(String n, int year, int month, int date) {
        super(year,month,date); name=n;
     }
     public int compareTo(Object other) {
       return compareTo((NamedDate)other);
     }
     public int compareTo(Date other) {
       return super.compareTo(other);
     }
     public int compareTo(NamedDate other) {
       if (this==other) return 0;
       int res = super.compareTo(other);
       if (res==0) return name.compareTo(((NamedDate)other).name);
       return res;
     }
}
```

Wir haben hier all die notwendigen überladenen Versionen von compareTo() im Stil der Superklasse Date implementiert. Diese Implementierung scheitert schon daran, dass sie die Anforderung (4) aus dem Comparator-Contract verletzt:

> *Finally, the implementer must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.*

Hier ist ein Beispiel:

```
Date  x = new Date();
NamedDate y = new NamedDate("SD deadline");
NamedDate z = new NamedDate("today");

x.compareTo(y);   // returns 0
x.compareTo(z);   // returns 0
y.compareTo(z);   // returns -33
```

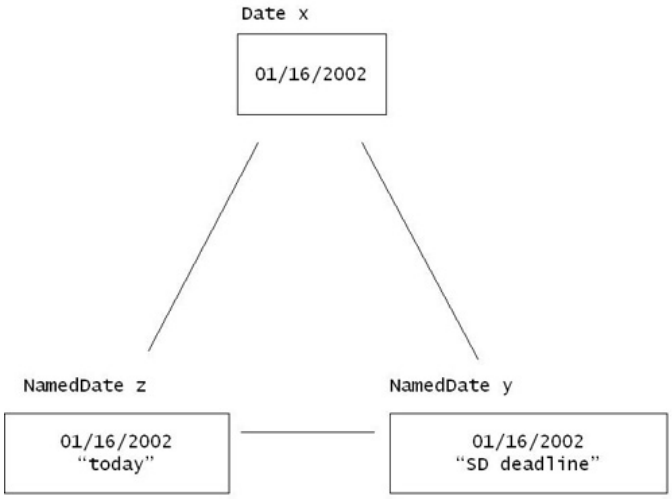

Abbildung 2: Beispiel einer inkorrekten (intransitiven) Implementierung von compareTo()

Der Aufruf x.compareTo(y) ruft die Methode Date.compareTo(Date), welche die Superklassen-Anteile der beiden Objekte vergleicht. Deshalb kommt hier Gleichheit heraus. Der Aufruf x.compareTo(z) ruft wiederum die Methode Date.compareTo(Date) und es kommt wieder Gleichheit heraus.Jetzt müsste auch für den Vergleich y.compareTo(z) Gleichheit herauskommen.Das ist aber nicht der Fall. Es wird die Methode NamedDate.compareTo(NamedDate) gerufen, welche den Subklassen-spezifischen Anteil, nämlich den String, einbezieht und dann eben keine Gleichheit mehr feststellt.

Ganz genau wie bei equals() zeigt sich, dass man den Vergleich von Sub- und Superklassen-Objekten nicht korrekt implementieren kann, wenn die Subklassen eigene Felder hinzufügen, die zum Vergleich beitragen. Die Implementierungen von compareTo() in der Klasse java.util.Date sind zwar nicht falsch, aber eine Subklasse von Date, die zusätzliche nicht-transiente, nicht-statische Felder hat, kann keine korrekte Implementierung von compareTo() liefern. Deshalb sollten die Date.compareTo()-Methoden als final deklariert sein.

Für eigene Klassen und Klassenhierarchien empfehlen wir, wie bei equals(), den Vergleichbarkeitstest per getClass(). Der Vergleichbarkeitstest per Downcast ist nur bei final-Klassen unproblematisch, wo er sogar den Vorteil bietet, dass die ClassCastException als Seiteneffekt des Donwncasts automatisch erzeugt wird.

Warum der Test per Downcast im JDK verwendet wird und auch in der einschlägigen Java-Literatur (siehe zum Beispiel /BLO/) propagiert wird, ist relativ unklar. Der Test per Downcast verleitet zu den gezeigten Fehlern in Klassenhierarchien, wohingegen der Test per getClass() robust ist und praktisch keine Nachteile hat.Er führt lediglich zu der Einschränkung, dass nur noch Objekte gleich Typs in einem baum-basierten Container abgelegt werden können. Das lässt sich aber leicht umgehen, indem man zusätzliche Comparatoren implementiert. Nehmen wir einmal an, wir hätten Date und NamedDate mit getClass() implementiert statt mit dem Downcast und wir wollten trotzdem einen TreeSet anlegen, der Date und NamedDate-Objekte enthält. Dann kann man sich einen speziellen Date-Comparator schreiben, der in seiner compare()-Methode den Vergleich von Date und NamedDate-Objekten zulässt und dabei immer den Date-Anteil der Objekte vergleicht. Wenn man den TreeSet mit diesem Date-Comparator versorgt, dann verwendet der Container statt der compareTo()-Methoden der einzelnen Elemente die compare()-Methode des Date-Comparators. Damit kann man den gewünschten heterogenen Container problemlos angelegen und alle Elemente im Container werden bzgl.Sortierung wie Date-Objekte behandelt.

Die Einschränkung durch die getClass()-Technik ist im Falle von compareTo() deutlich geringer als bei equals(), weil die baum-basierten Container verglichen mit den hash-basierten Containern sehr flexibel sind.Ein hash-basierter Java-Container ist darauf fixiert, die hashCode() und die equals()-Methode der Elemente zu verwenden. Ein Pendant zum Comparator, also eine Alternative zu hashCode() und equals(), gibt es bei den hash-basierten Java-Containern nicht.

Vergleich der Felder

Für den eigentlichen Vergleich wird man die einzelnen Felder von this mit den korrespondierenden Feldern von other vergleichen.In welcher Reihenfolge man das tut, hängt von der Semantik der Klasse ab. Üblicherweise priorisiert man die Felder und prüft weniger priore Felder nur wenn die höher prioren Felder gleich sind. Also zum Beispiel bei einem Namen wird man erst die Familiennamen miteinander vergleichen, und nur wenn die Familiennamen gleich sind, vergleicht man auch die Vornamen.Diese Priorisierung der Felder determiniert die resultierende Sortierreihenfolge.

Delegation an die Superklasse

Bei Feldern, die von einer Superklasse geerbt werden, wird man den Vergleich der geerbten Felder per super.compareTo() an die Superklasse delegieren. Ob man das als erstes oder als letztes tut, hängt von der Semantik der Klasse und der gewünschten Sortierreihenfolge ab. Es kommt häufig vor, dass die subklassen-spezifischen Felder nur dann verglichen werden, wenn der Superklassen-Anteil gleich ist.Das sieht dann wie folgt aus:

```
public int compareTo(Object other) {
  ...
```

```
  int result = super.compareTo(other);
  if (result == 0) {
    ...compare subclass specific parts ...
    return ...;
  }
  else
    return result;
}
```

Es muss aber nicht in allen Fällen so sein, dass die Superklassen-Felder in der Sortierreichenfolge die höhere Priorität gegenüber den Subklassen-Feldern haben.In unserem Beispiel einer Klasse, die Vor- und Zunamen enthält, könnte eine Klasse abgeleitet werden, die zusätzlich das Geburtsdatum enthält. Wenn nun zuerst super.compareTo() gerufen wird, dann bestimmt der Name die Sortierreihenfolge, und das Geburtsdatum kommt nur ins Spiel, wenn die Namen gleich sind. Es könnte aber auch so sein, dass die Sortierung nach Alter bzw.Geburtsdatum erfolgen soll; dann würde super.compareTo() später gerufen, nämlich erst dann, wenn die Geburtsdaten gleich sind.Das heisst, anders als bei equals() delegiert man nicht generell als erstes an die Superklasse. Das kann so sein, muss aber nicht.Weglassen kann man die Delegation an die Superklasse, d.h.den Vergleich der Superklassen-Felder, allerdings nicht.In den Vergleich müssen alle Felder einbezogen werden, die auch für equals() berücksichtigt werden, wegen der Konsistenz zu equals(). Dazu gehören natürlich auch die geerbten Felder.

Vergleich der eigenen Felder

Für den eigentlichen Vergleich der Felder unterscheidet man zwischen

- transienten Feldern,
- Feldern von primitivem Typ,
- Feldern, die Referenzen sind,
- Feldern, die Referenzen auf Objekte sind, die nicht Comparable sind, und
- Feldern, die Referenzen auf Objekte sind, die bei equals() eine Sonderbehandlung erhalten haben.

Transiente Felder werden wie bei equals() ignoriert.

Bei Feldern von primitiven Typ verwendet man die entsprechenden Operatoren >, <, und ==. Man könnte bei numerischen Feldern auf die Idee kommen, einfach die Differenz zu berechnen, weil die Differenz den verlangten Returnwert bereits sehr nahe kommt: die Differenz ist 0 ist bei Gleichheit und positiv oder negativ bei Ungleichheit. Hier ist ein Beispiel:

```
class someClass implements Comparable {
 private int size;
 ...
 int compareTo(Object other) {
  ...
  int diff = size - ((someClass)other).size;
  if (diff != 0)
     return diff;
  ...
 }
```

```
}
```

Das ist allerdings nicht zu empfehlen, weil die Differenz überlaufen könnte (d.h grösser als INTEGER.MAX_VALUE = (231-1) sein könnte) und dann ist das Ergebnis falsch.

Bei Feldern, die Referenzen sind, wird man meistens die compareTo()-Methode der entsprechenden Klasse rufen.

Bei Feldern, die Referenzen auf Typen sind, die das Comparable-Interface nicht implementieren, muss man auf einen geeigneten Comparator zurückgreifen, falls es einen gibt. Wenn es den auch nicht gibt und das Feld aber zur Implementierung von equals() beigetragen hat, dann kann man keine Konsistenz zu equals() mehr erreichen. Das sollte man dann entsprechend dokumentieren.

Feldern, die Referenzen auf Objekte sind, die bei equals() eine Sonderbehandlung erhalten haben, müssen wegen der Konsistenz zu equals() eine entsprechende Sonderbehandlung in der Implementierung von compareTo() erhalten. Ein Beispiel ist die StringBuffer-Klasse. Wenn man StringBuffer miteinander vergleichen will, muss man sie zunächst in Strings konvertieren und dann die Strings miteinander vergleichen. Für die Implementierung von compareTo() macht man das genauso. Das bietet sich auch schon deshalb an, weil die Klasse StringBuffer gar nicht Comparable ist, die Klasse String hingegen schon.

Zusammenfassung

In diesem Artikel haben wir uns angesehen wofür man die Methode compareTo() braucht, nämlich für das Sortieren von Objekten und das Speichern von Objekten in baum-basierten Containern.Wir haben uns die Anforderungen an die compareTo()-Methode, den sogenannten Comparator-Contract, angesehen und diskutiert, wie man eine korrekte Implementierung von compareTo() bewerkstelligt. Die wesentlichen Schwierigkeiten sind dabei die Konsistenz zu equals(), die empfohlen wird, aber nicht zwingend erforderlich ist. Die Konsistenz-Anforderung führt zu einer engen Abhängigkeit von der Implementierung der equals()-Methode einer Klasse. Eine weitere Schwierigkeit ist der Test auf Vergleichbarkeit von Objekten, den man entweder per getClass() oder per Downcast machen kann. Der Test per getClass() ist robuster und weniger fehleranfällig.Der Test per Downcast ist weit verbreitet sowohl im JDK als auch in der Java-Literatur, obwohl er zu Fehlern verleitet; er ist nur für final Klassen korrekt und hat dort den Vorteil, dass die notwenige Exception automatisch ausgelöst wird.Ausserdem haben wir gesehen, dass baum-basierte und hash-basierte Container trotz gemeinsamer Interfaces semantischrecht unterschiedlich sind, weil Operationen, die beim hash-basierten Container gutartig sind, beim baum-basierten Container zu Exceptions führen.

Literaturverweise

/KRE/ **Objekt-Vergleich per equals(), Teil 1 und 2**
Klaus Kreft & Angelika Langer
Java Spektrum, Januar 2002 und März 2002
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/01.Equals-Part1/01.Equals1.html
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/02.Equals-Part2/02.Equals2.html

/KRE2/ **Secrets of equals()**
Part 1: Not all implementations of equals() are equal
Part 2: How to implement a correct slice comparison in Java
Angelika Langer & Klaus Kreft
Java Solutions, April 2002 and August 2002
URL: http://www.AngelikaLanger.com/Articles/Java/SecretsOfEquals/Equals.html
URL: http://www.AngelikaLanger.com/Articles/Java/SecretsOfEquals/Equals-2.html

/GOS/ **The Java Language Specification, 2nd Ed.**, sec.15.12, p.345ff
James Gosling, Bill Joy, Guy Steele, Gilad Bracha
Addison-Wesley, June 2000
ISBN: 0201310082
URL: http://java.sun.com/docs/books/jls/

/BLO/ **Effective Java Programming Language Guide**
Josh Bloch
Addison-Wesley, June 2001
ISBN: 0201310058

/KNU/ **The Art of Computer Programming, vol.3: Sorting and Searching. ed.2.**
Donald E.Knuth
Addison Wesley, 1998
ISBN 0-201-89685-0

/SED/ **Algorithms. Second edition.**
Robert Sedgewick
Addison-Wesley, 1983, 1988, 1989 reprint with authors corrections
ISBN 0-201-06673-4

/JDK/ **Java 2 Platform, Standard Edition v1.3.1**
URL: http://java.sun.com/j2se/1.3/

/JDOC/ **Java 2 Platform, Standard Edition, v 1.3.1 - API Specification**
URL: http://java.sun.com/j2se/1.3/docs/api/index.html

Anmerkungen

[i] Diese equals()-Methode im Interface Comparator hat nichts mit dem Vergleich der Objekte zu tun, die der Comparator mittels compare() vergleicht. Comparator.equals() vergleicht Comparatoren miteinander; Comparator.compare() vergleicht zwei Objekte eines fremden Typs miteinander. Beispiel für einen Comparator:

```
final class CaseInsensitiveStringComparator implements Comparator {
  public int compare(Object o1,Object o2) {
    return ((String)o1).compareToIgnoreCase((String)o2);
  }
  public boolean equals(Object obj) {
    return (obj==null)?false:(getClass()==obj.getClass());
  }
}
```