Objektvergleich
Wie, wann und warum implementiert man die equals()-Methode?

Teil 1: Die Prinzipien der Implementierung von equals()

JavaSPEKTRUM, Januar 2002
Klaus Kreft & Angelika Langer

Vorbemerkung

Mit diesem Artikel beginnen wir unter dem Titel "Effective Java" eine Kolumne, die sich mit der Programmiersprache Java auseinandersetzen wird. Wir haben dabei bewusst den Titel "Effective Java" gewählt, um an die Tradition von Scott Meyers anzuknüpfen, der den Begriff "Effective..." durch seine "Effective C++"-Bücher populär gemacht hat. Betrachtungen unter dem Motto "Effective" wenden sich typischerweise der Tücke des Objekts zu und so werden wir uns in dieser Kolumne den mehr oder weniger offensichtlichen Fallstricken der Programmiersprache Java widmen.

Nun werden unter dem Begriff "Java" unzählige Aspekte subsummiert, von Realtime-Programmierung unter speziellen virtuellen Maschinen über GUI-Programmierung bis hin zur Applikationsentwicklung auf Basis von EJB und JSP. Es wäre vermessen, über "Java" in dieser Gesamtheit schreiben zu wollen. Wir beschränken uns daher bewusst auf den Kern von Java: die Programmiersprache selbst, wesentliche Aspekte der virtuellen Maschine und einige grundlegende APIs aus den Bibliotheken der Java 2 Standard Edition (J2SE). Unser Ziel ist es, über genau den Teil von Java zu schreiben, der jeden Java-Programmierer angeht, ganz egal in welcher Domain er oder sie programmiert. Dabei wollen wir uns die weniger offensichtlichen und bisweilen überraschenden Effekte in Java ansehen.

Objekt-Infrastruktur in Java

Beginnen wir mit scheinbaren Trivialitäten wie "Kopieren von Objekten" und "Vergleichen von Objekten". Gemeint sind die Methoden clone(), equals() und einige andere, die zusammen so etwas wie die "Infrastruktur" eines Objekts ausmachen. Was meinen wir mit "Infrastruktur"?

Alle Klassen in Java sind implizit von der Klasse Object abgeleitet und erben daher alle Methoden aus Object. Zu diesen geerbten Methoden gehören die public Methoden equals() und hashCode(). equals() vergleicht zwei Objekte miteinander, während hashCode() einen integralen Wert (den sogenannten Hashcode) berechnet. Mit den Details dieser Methoden werden wir uns in diesem und den folgenden Artikeln noch eingehend beschäftigen. An dieser Stelle nur soviel: beide Methoden werden u.a. gebraucht, um Java-Objekte in hash-basierten Containern wie beispielsweise HashSet ablegen zu können.

Wegen der automatischen Ableitung von der Superklasse Object sind equals() und hashCode() Teil der Schnittstelle einer jeder Java-Klasse, d.h. man kann auf allen Objekten in Java equals() und hashCode() aufrufen. Es gibt auch immer eine Implementierung dieser Methoden, nämlich entweder die aus Object geerbte Default-Implementierung oder eine klassenspezifische Implementierung, wenn die betreffende Klasse die geerbte Methode überschrieben hat.

Methoden wie equals() und hashCode() stellen Basisfunktionalität zur Verfügung, die man von allen Objekten in Java erwartet. Die Menge der Basisfunktionalitäten bezeichnet man bisweilen als "Infrastruktur" eines Objekts. Zur Infrastruktur gehören nicht nur equals() und hashCode(), sondern auch Funktionalität für Initialisierung und Aufräumen von Objekten sowie für Kopieren und Vergleichen von Objekten. Initialisierung geschieht üblicherweise mittels Konstruktoren, Aufräumen mittels finalize()-Methode, Kopieren mittels clone()-Methode, Vergleichen mittels equals() und compareTo()-Methode. Die Liste erhebt keinen Anspruch auf Vollständigkeit. Zur Infrastruktur gehören in gewissem Sinne auch die Methoden für die Serialisierung von Objekten, nämlich readObject() und writeObject(), weil sie ebenfalls so etwas wie Konstruieren und Kopieren von Objekten definieren. Die von einer Klasse geforderte Infrastruktur kann also variieren abhängig vom Kontext, in dem die Klasse verwendet werden soll.

Wie wir bereits gesehen haben, werden equals() und hashCode() von der Superklasse Object geerbt. Beides sind public Methoden in Object, d.h. equals() und hashCode() gehören immer zur Schnittstelle einer Klasse. Das ist anders bei clone() und finalize(). Diese beiden Methoden sind ebenfalls in der Superklasse Object definiert, aber sie sind dort als protected deklariert. Damit werden sie zwar geerbt, sind aber nicht automatisch Bestandteil der Schnittstelle der Subklasse. Nur wenn die Subklasse Funktionalität für das Kopieren oder Aufräumen unterstützen will, dann wird sie diese geerbten Methoden aus Object überschreiben und als eigene public Methoden zur Verfügung stellen. (Im Falle von clone() kommt noch hinzu, dass die Subklasse zusätzlich das Cloneable-Interface implementieren muss, damit die clone()-Methode funktioniert.)

Andere Teile der Infrastruktur haben gar nichts mit der Superklasse Object zu tun, sondern man implementiert gewisse Interfaces, um die entsprechende Infrastruktur zur Verfügung zu stellen. In diese Kategorie fallen die Methoden compareTo() aus dem Comparable-Interface und readObject() und writeObject(), aus dem Serializable-Interface. Diese Teile der Infrastruktur wird eine Klasse nur dann zur Verfügung stellen, wenn das sinnvoll erscheint, was allerdings häufig der Fall ist: wenn Objekte in baum-basierten Containern wie TreeSet abgelegt werden sollen, dann macht es sehr viel Sinn, dass die Klasse eine compareTo()-Methode bekommt. Analog, wenn Objekte serialisiert werden sollen, dann müssen readObject() und writeObject() implementiert werden.

Damit haben wir nun eine Liste von Basisfunktionalität, die jede Java-Klasse zur Verfügung stellen kann. Beim Design einer neuen Klasse muss entschieden werden, welche Teile der Infrastruktur unterstützt werden sollen. Gewisse Methoden, nämlich equals() und hashCode(), können gar nicht vermieden werden. Wenn eine Klasse diese Methoden nicht überschreibt, dann steht automatisch die Default-Funktionalität aus der Superklasse Object zur Verfügung. Für diese Methoden ist die entscheidende Frage nicht "Unterstützen? Ja oder Nein?", sondern

man muss entscheiden: "Ist das Default-Verhalten korrekt? Ja oder Nein?". Die Entscheidungen, die der Autor einer Klasse an dieser Stelle trifft, haben weitreichende Auswirkungen für die Benutzung und Benutzbarkeit der Klasse. Das gilt ganz besonders, wenn die Klasse eine potentielle Superklasse ist, und jede Klasse in Java, die nicht als final erklärt ist, ist eine potentielle Superklasse.

In dieser und den nachfolgenden Ausgaben der Kolumne wollen wir uns einige Teile dieser Infrastruktur näher ansehen. Dabei wird sich herausstellen, dass korrekte Implementierungen der Infrastruktur keineswegs immer trivial sind. Was theoretisch so harmlos aussieht, kann in der Praxis tückisch sein. Landläufig herrscht die Meinung: "Es ist doch kein Problem, clone() oder equals() zu implementieren. Da muss man doch nur alle Felder kopieren bzw. miteinander vergleichen und das war's dann schon. Oder nicht?" Oder doch? Wir werden sehen!

Schauen wir uns diesmal den Objektvergleich mittels equals() an.

Objektvergleich in Java

In Java gibt es zwei Möglichkeiten, Variablen zu vergleichen: die eine ist der Vergleich über den == Operator, die andere Möglichkeit ist der Vergleich mit Hilfe der equals()-Methode.

Beispiel:

```
int x = 100;
int y = 100;
...
if (x==y) ...
```

Hier werden zwei int-Variablen miteinander verglichen. Für den Vergleich gibt es nur den == Operator, weil der Typ int keine equals()-Methode hat. Generell unterscheidet man in Java zwischen Variablen vom primitiven Typ und Referenz-Variablen.

Primitive Typen sind in der Sprache vordefinierte Typen wie int, long, boolean, etc.. Für Variablen vom primitivem Typ gibt es nur den Vergleich über den == Operator und der liefert true, wenn beide Variablen den gleichen Wert enthalten, wie das in obigem Beispiel der Fall ist.

Nicht-primitiven Typen sind Klassen und Interfaces. Alle Variablen diesen Typs sind in Java Referenzvariablen. Sie verweisen lediglich auf Objekte, enthalten diese Objekte aber nicht.

Beispiel:

```
String s1 = new String("Hello World !");
String s2 = new String("Hello World !");
...
if (s1 == s2) ...      // yields false
...
if (s1.equals(s2)) ...    // yields true
```

Hier werden zwei String-Variablen verglichen. String ist eine Klasse und deshalb sind die beiden Variablen s1 und s2 Referenzvariablen. Für Referenzvariablen gibt es neben dem Vergleich per == Operator den Vergleich mit Hilfe der equals()-Methode. Die beiden Vergleiche haben nicht nur unterschiedliche Syntax, sondern auch unterschiedliche Semantik.
Der Vergleich per == Operator ist die Prüfung auf Identität der beiden referenzierten Objekte. In unserem Beispiel haben wir zwei Referenzen s1 und s2 auf zwei String -Objekte, die an verschiedenen Stellen auf dem Heap angelegt wurden und den gleichen Inhalt haben. Die beiden referenzierten String-Objekte sind zwar gleich in dem Sinne, dass sie den gleichen Inhalt, nämlich "Hello World !", haben, aber sie sind nicht identisch, da sie an verschiedenen Stellen im Speicher angelegt sind.
Das Beispiel zeigt den Unterschied zwischen dem == Operator und der equals()-Methode: Der Vergleich mittels == Operator prüft auf Identität der referenzierten Objekte, während der Vergleich mittels equals()-Methode im Falle von String auf Gleichheit des Inhalts der referenzierten Objekte prüft. In unserem Beispiel liefert der erste Vergleich false (d.h. "nicht identisch") und der zweite Vergleich true (d.h. "inhaltlich gleich").

Damit haben wir nun ein erstes intuitives Verständnis von equals(): es prüft auf inhaltliche Gleichheit im Gegensatz zum == Operator, der auf Identität prüft (equality vs. identity).

Leider ist es nicht immer so, dass equals() und der == Operator diese unterschiedlichen Eigenschaften haben. Man findet schon in den Java-Bibliotheksklassen Beispiele für abweichendes Verhalten.

Beispiel:

```
String init = "Hello World !";

StringBuffer sb1 = new StringBuffer(init);
StringBuffer sb2 = new StringBuffer(init);
...
if (sb1 == sb2) ...      // yields false
...
if (sb1.equals(sb2)) ...    // yields false (!!!)
```

Offenbar sind StringBuffer-Objekte selbst bei gleichem Inhalt nicht gleich; jedenfalls ist dies das Ergebnis des Vergleichs mittels equals(). Wie kann das sein?

Nun, das liegt daran, dass jede Klasse die equals()-Methode von der Superklasse Object erbt. Eine Klasse wie StringBuffer, die die geerbte equals()-Methode nicht überschreibt, stellt damit automatisch die Default-Implementierung von equals()aus Object zur Verfügung. Die Default-Implementierung ist aber identisch mit dem Verhalten des == Operators: es wird auf Identität der referenzierten Objekte geprüft.

Dieses Defaultverhalten von equals()aus Object erklärt sich dadurch, dass in der Klasse Object über die Struktur und den Inhalt von Subklassen nichts bekannt ist. Eine universelle Implementierung von equals(), die für jede beliebige Subklasse "das Richtige" tut, nämlich den Inhalt vergleichen, wäre zwar machbar gewesen (mit Hilfe von dynamischer Typinformation), aber aufwendig. Die Designer der Klasse Object haben sich für eine einfachere Lösung

entschieden und deshalb wird in Object.equals() nur auf Identität und nicht auf inhaltliche Gleichheit geprüft.

Dieses Default-Verhalten von Object.equals() und die Tatsache, dass die Klasse StringBuffer die geerbte equals()-Methode nicht überschreibt, erklären, warum in obigem Beispiel in beiden Vergleichen false als Ergebnis geliefert wird: die StringBuffer-Objekte haben zwar gleichen Inhalt, sind aber nicht identisch.

Ob das Ergebnis des Vergleichs von StringBuffer-Objekten mittels equals() das ist, was man erwartet, kann man sicher kontrovers diskutieren. Zumindest wirft es Fragen auf ... wann muss eine Klasse die Default-Implementierung von equals() überschreiben, und wann nicht? Und wenn ja, wie? Damit wollen wir uns im Folgenden beschäftigen.

Value vs. Entity-Types

Typen lassen sich in zwei Kategorien einteilen: man unterscheidet zwischen sogenannten Value- und Entity-Typen.

- **Value-Typen** . Alle primitiven Typen in Java sind Value-Typen. Sie enthalten einen Wert und dieser Wert ist das Wesentliche. Klassen können ebenfalls Value-Typen sein. Bei solchen Klassen ist der Inhalt der Objekte ganz wesentlich. Der Inhalt repräsentiert den Wert des Objekts und bestimmt das Verhalten der Objekte fast vollständig. Beispiele solcher Value-Klassen sind die Standard-Klassen BigDecimal, String, Date, Point, etc.

- **Entity-Typen** . Darunter versteht man Klassen, bei denen der Inhalt nicht das Wesentliche ist. Sie werden nicht als "Werte" betrachten und auch nicht als "Wert" herumgereicht. Das sind Typen, die hauptsächlich Dienste anbieten, oder Typen, die Referenzen auf andere unterliegende Objekte darstellen. Beispiele sind die Standardklassen Thread, Socket, oder FileOutputStream.

Betrachten wir zur Illustration ein Thread-Objekt und ein String-Objekt. Ein String-Objekt ist im Wesentlichen durch seinen Inhalt, nämlich die enthaltene Zeichenkette, bestimmt. Davon kann man Kopien anlegen und man kann sie vergleichen. Das ist bei einem Thread-Objekt ganz anders. Natürlich hat auch ein Thread -Objekt Inhalt; ein Thread hat einen Namen und einen Zustand (runnable, blocked, dead, usw.) und eine Priorität und er verwendet ein Runnable-Objekt, dessen Code er ausführt. Aber all diese Eigenschaften ergeben in ihrer Gesamtheit keinen "Wert", den man vergleichen oder kopieren möchte. Wann sind zwei Threads gleich? Wenn sie denselben Namen haben? Oder denselben Code ausführen? Das macht logisch keinen Sinn. Was soll man sich unter der Kopie eines Threads vorstellen? Auch das macht nicht so recht Sinn. In solchen Fällen spricht man von Entity-Typen, wobei die Grenze zwischen Value- und Entity-Typen oftmals schwer zu ziehen ist.

Was bedeutet die Unterscheidung zwischen Value- und Entity-Typen für die Implementierung von equals()?

Entity-Typen überschreiben selten die equals()-Methode. Da sie keine Werte darstellen, ist der Vergleich des Inhalts praktisch bedeutungslos und aus diesem Grunde ist es völlig in Ordnung, wenn zwei Entity-Objekte genau dann "gleich" sind, wenn sie identisch sind.

Das ist bei Value-Typen ganz anders. Der Inhalt ist das Wesentliche des Objekts und deshalb sind zwei Value-Objekte genau dann gleich, wenn sie den gleichen Inhalt haben. In solchen Fällen muss equals() überschrieben werden, denn die Default-Implementierung ist unbrauchbar für solche Value-Typen.

Was schließen wir daraus? Eine der ersten Entscheidungen, die beim Design einer neuen Klasse gefällt werden muss, ist die Entscheidung, ob die Klasse Value- oder Entity-Objekte beschreiben soll. Im Falle von Entity-Verhalten kann man sich die Arbeit mit equals() sparen; im Falle von Value-Verhalten muss man es implementieren.

In der Praxis

Wie ist das nun in der Praxis?

"Habe ich was falsch gemacht, wenn ich eine Klasse ohne equals() geschrieben habe?"

Das kommt darauf an. Wenn es ein Entity-Typ ist, also eine reine Service-Klasse ist oder einen Verweis auf irgendwas darstellt, dann nicht. Wenn es aber ein Value-Typ ist, dann ist die geerbte equals()-Methode normalerweise inkorrekt.

"Aber ich weiß genau, dass equals() überhaupt nicht aufgerufen, nirgendwo in der gesamten Applikation. Wozu soll ich mir all die ganze Arbeit machen, wenn das sowieso keiner braucht?"

Das ist ein überzeugendes Argument! Aber ... wer kann schon mit Bestimmtheit sagen, dass eine Methode, die heute nicht gebraucht wird, morgen ebenfalls nicht gebraucht werden wird? Das Gefährliche an equals() ist, dass es immer definiert ist, weil es bereits in der Superklasse Object implementiert ist. Wenn morgen jemand MyClass.equals() ruft, dann lässt sich das klaglos übersetzen und es läuft ... aber leider falsch. Die dann einsetzende Fehlersuche erinnert fatal an die Suche nach Pointer-Problemen in C oder C++ - und das glaubte man doch in Java hinter sich gelassen zu haben. Sobald man sich halbwegs darüber klar geworden ist, dass man mit seiner Klasse einen Value-Typen implementiert, dann sollte man auf jeden Fall equals() korrekt implementieren. Alles andere ist fahrlässig.

Erschwerend kommt hinzu, dass equals() nicht immer sichtbar benutzt wird, sondern bereits implizit von gewissen JDK-Klassen verwendet wird. Der wichtigste Vertreter dieser equals()-benutzenden JDK-Klassen sind die hash-basierten Container wie Hashtable, HashMap und HashSet. Aber auch andere Klassen benutzen equals(). Häufig ist dies nicht einmal explizit in der JavaDoc ausgewiesen; eine korrekte equals()-Implementierung wird deshalb von jeder Klasse erwartet. Man kann also gar nicht mit Gewissheit sagen, dass equals() nicht gebraucht wird, weil es nicht benutzt wird.

Das heißt, der Autor einer Klasse muss in jedem Fall entscheiden, welche Semantik (Entity- oder Value-Typ) die Klasse haben soll. Daraus ergibt sich dann die Semantik für die equals()-

Methode der neuen Klasse. Anders als bei anderen Teilen der Objekt-Infrastruktur kann man sich bei equals() um die Entscheidung nicht drücken. Wenn man sich nicht entscheidet, ist die Klasse mit ihrer geerbten Default-Implementierung von equals() u.U. inkorrekt.

Der sogenannte equals()-Contract

Wenn man nun equals() implementieren will, was muss man tun? Was wird von equals() erwartet? Intuitiv ist klar, dass es den Inhalt zweier Objekte vergleichen soll. Aber was bedeutet das genau?

Der Vergleich zweier Objekte sollte gewissen Regeln folgen, die man mehr oder weniger intuitiv von einem Vergleich erwartet. Diese zusätzlichen Eigenschaften einer Implementierung von equals() sind formal beschrieben im sogenannten "equals()-Contract". Den equals()-Contract findet man in der JDK JavaDoc unter Object.equals(). Hier ist die Originalbeschreibung aus der API Spezifikation der JavaTM 2 Platform, Standard Edition:

```
public boolean equals(Object obj)
```

> *Indicates whether some other object is "equal to" this one.*
>
> *The equals method implements an equivalence relation:*
>
> - *It is reflexive: for any reference value x, x.equals(x) should return true.*
> - *It is symmetric: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.*
> - *It is transitive: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.*
> - *It is consistent: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.*
> - *For any non-null reference value x, x.equals(null) should return false.*

Das bedeutet das Folgende:

- Jedes Objekt liefert beim Vergleich mit sich selbst true.
- Es ist egal, ob man x mit y vergleicht, oder y mit x; das Ergebnis ist dasselbe.
- Wenn x gleich y ist und y gleich z, dann sind auch x und z gleich.
- Man kann zwei Objekte beliebig oft miteinander vergleichen; es kommt immer dasselbe heraus, solange sich die Objekte nicht verändern.
- Alle Objekte sind von null verschieden.

Eigentlich sind die Forderungen im equals()-Contract naheliegend und intuitiv verständlich. Das ist genau das, was jeder von einer Gleichheitsrelation erwartet. Man sollte also stets darauf achten, dass equals() konform zu diesen Regeln implementiert wird. Wenn eine Implementierung davon abweicht, dann sind Probleme unvermeidbar, weil sich alle Benutzer von equals() intuitiv auf die Eigenschaften verlassen, die der equals()-Contract formal beschreibt.

Anleitung zum Implementieren von equals()

Im Folgenden werden wir equals() Zeile für Zeile implementieren.

Signatur

Eine Implementierung von equals() beginnt damit, dass man sich die Signatur (d.h. Anzahl und Type der Argumente) der equals()-Methode überlegen muss. Üblicherweise will man die Version von equals(), die in der Klasse Object definiert ist, überschreiben. Aus diesem Grunde ist es klar, dass die equals()-Methode der eigenen Klasse exakt dieselbe Signatur haben muss, wie Object.equals(), nämlich

```
public boolean equals(Object other)
```

Es gibt alternative Ansätze, bei denen Überschreiben und Überladen kombiniert wird, aber diese Technik ist ungewöhnlich und wir wollen sie deshalb zunächst nicht betrachten.

Alias-Prüfung

Man kann die Implementierung von equals() sofort beenden, wenn die beiden zu vergleichenden Objekte identisch sind. In diesem Falle müssen sie den gleichen Inhalt haben und man kann sich den gesamten Vergleich des Inhalts sparen. Aus Optimierungsgründen kann man daher als erstes auf Identität von this und other prüfen.

```
public boolean equals(Object other) {
 if (this == other)
   return true;
 ...
}
```

Aufgabenteilung in einer Klassenhierarchie

Die nächsten Schritte der Implementierung sind abhängig davon, ob die Klasse, deren equals()-Methode wir implementieren wollen, eine direkte Subklasse von Object ist, oder ob es sich um eine in der Klassenhierarchie weiter unten liegende Subklasse handelt. Wir unterscheiden daher im Folgenden zwischen direkten Subklassen von Object und indirekten Subklassen von Object. (Eigentlich muss man den Begriff "direkte Subklasse von Object" noch etwas präziser fassen. Als "direkte Subklasse von Object" betrachten wir hier die erste Subklasse, die equals() implementiert. Es kann also durchaus vorkommen, dass Object eine Subklasse hat, die aber kein equals() implementiert, beispielsweise weil sie keine Felder hat. Wenn es eine Sub-Subklasse gibt, die equals() implementiert, dann ist diese Sub-Subklasse die "direkte Subklasse von Object" im Sinne unserer Definition.) Die Unterscheidung zwischen direkten und

indirekten Subklassen ist bedeutsam, weil gewisse Aufgaben in einer Klassenhierarchie nur einmal erledigt werden müssen. Und diese Aufgaben werden von direkten Subklassen übernommen.

Generell ist es so, dass eine Subklasse den Vergleich ihrer Felder durchführt und den Vergleich der von den Superklassen geerbten Felder an ihre direkte Superklasse delegiert. Ähnlich wie bei Konstruktoren delegiert dabei jede Klasse an ihre direkte Superklasse, so dass rekursiv die equals()-Methoden der gesamten Hierarchie aufgerufen werden und damit das gesamte Objekt verglichen wird. Die Rekursion endet bei der Klasse in der Hierarchie, die direkt von Object abgeleitet ist. Sie ruft die equals()-Methode ihrer Superklasse, nämlich Object.equals(), nicht auf. Stattdessen übernimmt sie gewisse Sonderaufgaben.

Dazu gehört die Prüfung auf Vergleichbarkeit, d.h. die Prüfung, ob der Inhalt von this überhaupt mit dem Inhalt von other verglichen werden kann[1]. Der Vergleich ist beispielsweise nicht möglich, wenn other eine null-Referenz ist, also keinen Inhalt hat. Der Vergleich ist auch dann nicht möglich, wenn other auf ein Objekt verweist, dass von einem gänzlich inkompatiblen Typ ist. Schließlich lassen sich "Äpfel und Birnen" nicht miteinander vergleichen. Hingegen ist der Vergleich immer dann möglich, wenn this und other von genau dem gleichen Typ sind.

Indirekte Subklassen von Object

Sehen wir uns zunächst die Klassen an, die nicht direkt von Object abgeleitet werden. In der equals()-Implementierung solcher Klassen wird nach dem bereits gezeigten Alias-Check an die Superklasse delegiert und super.equals() gerufen.

```
boolean equals(Object other) {
  ...
  if (!super.equals(other))
   return false;
  ...
}
```

Direkte Subklassen von Object

In einer Klasse, die direkt von Object abgeleitet ist, wird nicht super.equals() aufgerufen. Statt dessen wird der Fall einer null-Referenz behandelt und auf Vergleichbarkeit geprüft.

Der Aufruf von super.equals() ist nicht nur überflüssig, sondern wäre ernsthaft falsch; man sollte ihn also nicht etwa versehentlich machen. super.equals() ist im Falle einer direkten Subklasse von Object genau Object.equals(). Die Implementierung von Object.equals()liefert aber auch false, wenn this und other zwar denselben Inhalt haben, aber als Duplikate im Speicher an verschiedenen Stellen angelegt sind, also nicht identisch sind. Die Information, die Object.equals()liefert ist daher unbrauchbar für die Implementierung der equals()-Methode einer Subklasse.

[1] Dieser Vergleich ist nicht zu verwechseln mit dem Alias-Check, bei dem geprüft wird, ob this und other identisch sind.

Zu den Sonderaufgaben einer direkten Subklasse von Object:

Test auf null

Nach dem Alias-Check wird geprüft, ob other eine null-Referenz ist.

```
public boolean equals(Object other) {
  ...
  if (other == null)
    return false;
  ...
}
```

Diesen Test kann man im Prinzip auch in jeder indirekten Subklasse machen, aber es genügt, ihn genau einmal in der obersten Klasse durchzuführen. Wenn man sich an das Koch-Rezept hält, d.h. wenn jede Subklasse nach dem Alias-Check als erstes an die Superklasse delegiert und die oberste Klasse als erstes auf null abprüft, dann erfolgt die Prüfung garantiert, bevor irgendwelche Zugriffe auf Felder von other erfolgen. Damit ist sichergestellt, dass es nicht zu einer NullPointerException kommt, denn diese würde den equals()-Contract verletzen. Der equals()-Contract verlangt, dass der Vergleich mit null-Referenzen das Ergebnis false liefert; das Werfen einer NullPointerException ist daher kein konformes Verhalten.

Ganz allgemein sollte man es vermeiden, equals() mit einer NullPointerException zu beenden. Als Ergebnis von equals() wird true oder false erwartet. Wenn der Vergleich aus irgendwelchen Gründen nicht gemacht werden kann, dann sollte keine Exception geworfen werden, sonders es sollte false als Ergebnis geliefert werden.

Test auf Vergleichbarkeit

Nachdem getestet ist, dass other keine null-Referenz ist, wird auf Vergleichbarkeit geprüft. Interessanterweise ist das das kontroversesten Themen im Zusammenhang mit equals() überhaupt. Wir werden den Test auf Vergleichbarkeit in der nächsten Kolumne noch näher beleuchten. Hier nur ein erster Einblick in die Problematik.

Der Vergleichbarkeitstest ist nötig, weil equals() ein Argument vom Typ "Referenz auf Object" akzeptiert. Eine solche Referenz kann daher auf jede Art von Objekt zeigen und es ist keineswegs sicher gestellt, dass other auf ein Objekt desselben Typs wie this zeigt oder dass die referenzierten Objekte wenigstens in irgendeiner Form vergleichbar sind. Die Vergleichbarkeit muss daher durch einen expliziten Test feststellen.

Das heißt, man muss sich vor der Implementierung von equals(), genau genommen schon beim Design der Klasse, überlegen, mit welcher Art von Objekten ein Vergleich überhaupt möglich und sinnvoll ist. Im einfachsten Fall ist der Vergleich nur zwischen Objekten gleichen Typs erlaubt. Das sieht dann wie folgt aus:

```
public boolean equals(Object other) {
  ...
  if (other.getClass() != getClass())
   return false;
  ...
}
```

Daneben gibt es zahlreiche Techniken, bei denen versucht wird, den Vergleich zwischen Sub- und Superobjekten zu erlauben. Immerhin haben Sub- und Superobjekte einen gemeinsamen Superklassenanteil, den man miteinander vergleichen kann. Solche Implementierungen sind aber meistens inkorrekt, weil sie nicht transitiv und oft nicht einmal symmetrisch sind, und damit nicht den Anforderungen aus dem equals()-Contract entsprechen. Diese häufig fragwürdigen Implementierungen sind leider so populär, dass wir ihnen die nächste Ausgabe der Kolumne widmen werden.

Vergleich der Felder

Nach dem Delegieren an die Superklasse bzw. den Tests auf null und auf Vergleichbarkeit folgt in direkten wie indirekten Subklassen der eigentliche Vergleich der Felder.

Im Prinzip müssen alle Felder von this mit den korrespondierenden Feldern von other verglichen werden. Wie das im Einzelnen geschieht, hängt vom Typ der Felder ab. Man unterscheidet zwischen

- transienten Feldern,
- Feldern von primitivem Typ,
- Feldern, die Referenzen sind und den Wert null haben können,
- Feldern, die Referenzen auf Objekte sind die keine korrekte Implementierung von equals() haben, und
- allen übrigen Feldern.

Transiente Felder

Transiente Felder werden ignoriert. Sie tragen nichts zum Zustand des Objekts bei und gehören logisch nicht zum Inhalt des Objekts. Daher werden sie beim Vergleich nicht berücksichtigt.

Primitive Typen

Felder von primitivem Typ werden mit Hilfe des == Operators verglichen. Beispiel:

```
class MyClass {
 private int size;
 ...
 public boolean equals(Object other) {
 ...
 if (size != ((MyClass)other).size)
   return false;
 ...
 }
}
```

Bei primitiven Typen vergleicht der == Operator den Inhalt und das ist genau das, was wir hier brauchen.

Referenz-Typen

Felder, die Referenzen sind, werden verglichen, indem die equals()-Methoden der referenzierten Objekte gerufen werden. Beispiel:

```
class MyClass {
 private String s;
 ...
 public boolean equals(Object other) {
 ...
 if (!(s.equals(((MyClass)other).s)))
  return false;
 ...
 }
}
```

Mögliche null-Referenzen

Obige Lösung ist natürlich nur korrekt, wenn von der Logik der Klasse her sichergestellt ist, dass das betreffende Feld keine null-Referenz sein kann. Wenn null ein möglicher Wert ist, dann muss zuvor auf null abgeprüft werden, um eine NullPointerException zu verhindern. Beispiel:

```
class MyClass {
 private String possNull;
 ...
 public boolean equals(Object other) {
 ...
 if (possNull == null)
 {if (((MyClass)other).possNull != null)
    return false;
 }
 else
 {if (!(possNull.equals(((MyClass)other).possNull)))
    return false;
 }
 ...
 }
}
```

Typen ohne korrektes equals()

Der Aufruf der equals()-Methode des referenzierten Objekts macht nur Sinn, wenn diese equals()-Methode korrekt, d.h. konform zu den Regeln des equals()-Contracts, implementiert ist. Andernfalls muss man Sonderlösungen und Umgehungen finden.

Ein Beispiel für einen solchen Typ, der eine Sonderbehandlung braucht, ist StringBuffer. Man kann StringBuffer-Objekte zwar per StringBuffer.equals() miteinander vergleichen, aber es wird auf Identität und nicht auf inhaltliche Gleichheit geprüft. Das ist nicht das, was hier gebraucht wird; wir wollen den Inhalt der StringBuffer-Objekte vergleichen und brauchen daher eine Umgehungslösung, die man mit Hilfe der Klasse String bauen kann. Für einen korrekten Vergleich des Inhalts von zwei StringBuffer-Objekten kann man beide Objekte in String-Objekte

konvertieren (per toString()-Methode) und dann die String-Objekte per String.equals() miteinander vergleichten

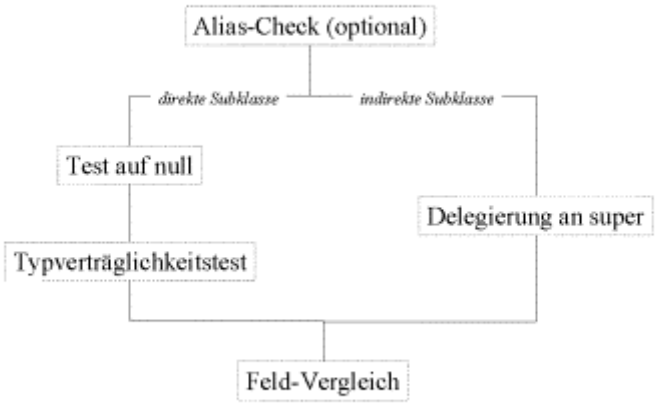
Ein anderes Beispiel für Felder, die eine Sonderbehandlung brauchen, sind Arrays. Zum Vergleich von zwei Arrays muss der Inhalt der Arrays elementweise verglichen werden, indem man für jedes Array-Element die equals()-Methode aufruft (oder bei Elementen von primitivem Typ den == Operator). Zur Arbeitserleichterung gibt bereits eine Hilfsklasse, nämlich java.util.Arrays. Diese Klasse hat eine statische Methode equals(), die genau das oben beschriebene tut.

Am Ende, wenn alle Prüfungen und Vergleiche erfolgreich durchgeführt worden sind, wird true zurückgegeben.

Rollenspiele

Klassen ohne korrekte Implementierung von equals() machen ihren Benutzern reichlich Probleme. Die Probleme sind insbesondere dann gravierend, wenn es sich bei der inkorrekten Klasse um eine Superklasse handelt. Die Subklasse hat dann kaum noch eine Chance, ihrerseits eine korrekte Implementierung von equals() zur Verfügung zu stellen, weil Zugriff auf die privaten Daten der Superklasse gar nicht möglich ist und u.U. die Kette von rekursiven Delegationen an super.equals() unterbrochen ist. Es ist daher bei der Implementierung von potentiellen Superklassen, d.h. von Klassen, die nicht als final deklariert sind, besonders wichtig, dass sie equals() korrekt implementieren und nach Möglichkeit dem vorgeschlagenen Rezept folgend ihre Sonderaufgaben übernehmen[2].

Hier nochmals in der Übersicht die Verantwortlichkeiten in einer Klassenhierarchie, sowie Beispielimplementierungen im Source-Code:



[2] Andere Vorgehensweisen sind denkbar, aber in jedem Falle sollten alle Klassen in einem Projekt oder zumindest in einer Klassenhierarchie demselben Konzept folgen. Alternative Implementierungen von equals() und deren Vor- und Nachteile besprechen wir im nächsten Artikel.

Implementierung von equals() in einer direkten Subklasse von Object

```
class MyClass {
 private String s;
 private int i;
 ...
 public boolean equals(Object other) {
 if (this == other)
  return true;
 if (other == null)
  return false;
 if (other.getClass() != getClass())
  return false;

 if (!(s.equals(((MyClass)other).s)))
  return false;
 if (i != ((MyClass)other).i)
  return false;
 ...
 return true;
 }
}
```

Implementierung von equals() in einer indirekten Subklasse von Object

```
class MySubclass extends MyClass {


 private String t;
 ...
 public boolean equals(Object other) {
 if (this == other)
  return true;
 if (!super.equals(other))
  return false;

 if (!(t.equals(((MySubClass)other).t)))
  return false;
 ...
 return true;
 }
}
```

Haben wir damit den equals()-Contract erfüllt? Sehen wir uns die 5 Anforderungen noch einmal an.

1. *Jedes Objekt liefert beim Vergleich mit sich selbst true.*

Das ist erfüllt, weil wir als erstes den Alias-Check ausführen.

2. *Es ist egal, ob man x mit y vergleicht, oder y mit x; das Ergebnis ist dasselbe.*

Wir vergleichen nur Objekte gleichen Typs miteinander; deshalb ist unsere Implementierung von equals() symmetrisch.
Verletzungen dieser Forderung können auftreten, wenn der Vergleich zwischen Objekten unterschiedlichen Typs erlaubt wird. Dann findet man bisweilen Implementierungen, die asymmetrisch sind, weil A.equals() den Vergleich mit Objekten des Typs B erlaubt, aber B.equals() den Vergleich mit A-Objekten nicht zulässt. Dazu mehr im nächsten Artikel.

3. *Wenn x gleich y ist und y gleich z, dann sind auch x und z gleich.*

Wir vergleichen nur Objekte gleichen Typs miteinander; deshalb ist unsere Implementierung von equals() transitiv.
Verletzungen dieser Forderung können auftreten, wenn der Vergleich zwischen Objekten unterschiedlichen Typs erlaubt wird. Dann findet man bisweilen Implementierungen, die intransitiv sind, weil der Vergleich zwischen einem A-Objekt und einem B-Objekt true liefern kann und genauso der Vergleich zwischen dem B-Objekt und einem anderen A-Objekt; aber das heißt noch nicht, dass deshalb die beiden A-Objekte gleich sind, obwohl das laut equals()-Contract so sein sollte. Dazu mehr im nächsten Artikel.

4. *Man kann zwei Objekte beliebig oft miteinander vergleichen; es kommt immer dasselbe heraus, solange sich die Objekte nicht verändern.*

In unserer Implementierung von equals() werden die Felder miteinander verglichen. Es geht keine weitere Information in die Produktion des Booleschen Ergebnisses ein. Deshalb ist das Ergebnis immer dasselbe, sofern sich die Objekte nicht ändern.
Fehler können nur auftreten, wenn beispielsweise statische Daten in die Ermittlung des Ergebnisses eingehen würden, was aber ganz ungewöhnlich wäre.

5. *Alle Objekte sind von null verschieden.*

Das haben wir durch die Prüfung auf null in der direkten Basisklasse von Object erreicht und dadurch, dass wir NullPointerExceptions sorgfältig vermieden haben.

Zusammenfassung und Ausblick

Jede Klasse in Java muss ein Minimum an Objekt-Infrastruktur implementieren, damit sie sinnvoll verwendbar ist. Zu diesen grundlegenden Methoden gehört u.a. auch die equals()-Methode. Dabei ist man nicht frei in der Wahl der Semantik, die man der equals()-Methode gibt. Jede Implementierung von equals()sollte die Regeln des sogenannten equals()-Contracts befolgen. Dafür ist es nötig, dass Klassen, die Value-Typen repräsentieren, die Defaultimplementierung aus Object.equals() überschreiben, weil diese auf Identität und nicht auf inhaltliche Gleichheit prüft, was für Value-Typen semantisch falsch ist. Wir haben das Prinzip solcher Implementierungen diskutiert, wobei eine Reihe von Details offen geblieben sind. Insbesondere ist offen, welche Arten von Objekten als "miteinander vergleichbar "gelten sollen. Wir haben nur den Vergleich von Objekten gleichen Typs erlaubt. Dazu gibt es aber Alternativen, die wir im nächsten Artikel diskutieren werden.

Objektvergleich
Wie, wann und warum implementiert man die equals()-Methode?

Teil 2: Der Vergleichbarkeitstest

JavaSPEKTRUM, März 2002
Klaus Kreft & Angelika Langer

In diesem Artikel knüpfen wir an den vorangegangenen Artikel (siehe /KRE1/) über equals()-Implementierungen an. Nach einen kurzen Wiederholung der Hauptaspekte des letzten Artikels konzentrieren wir uns auf den Vergleichbarkeitstest, den man im Rahmen einer equals()-Implementierung durchführen muss. Anhand von Beispielen aus Veröffentlichungen und dem JDK diskutieren wir die verschiedenen Aspekte des Vergleichbarkeitstest mittels getClass()-Methode und instanceof-Operator.

Rückblick

Objekte in Java kann man miteinander vergleichen, indem man die Methode equals() des einen Objekts aufruft und das andere Objekt als Argument mitgibt. Das Ergebnis ist ein boolscher Wert, der angibt, ob die beiden Objekte "gleich" sind. Ausnahmslos alle Klassen in Java definieren diese Methode. Die Implementierung der equals()-Methode ist entweder von der Superklasse Object geerbt oder wurde in der betreffenden Klasse überschrieben.

Genau damit haben wir uns in der letzten Ausgabe dieser Kolumne beschäftigt: wie implementiert man equals() in korrekter Art und Weise, wenn die Defaultimplementierung aus Object nicht das Richtige tut. Wenn man eine neue Klasse in Java entwirft, dann muss man neben all den anderen Design-Entscheidungen auch noch entscheiden, ob die Klasse equals() überschreiben muss oder nicht. Wir haben zwischen Value- und Entity-Typen unterschieden und festgestellt, dass Value-Typen üblicherweise die equals()-Methode überschreiben müssen, wohingegen für Entity-Typen die geerbte Default-Implementierung aus Object ausreichend ist. Das liegt daran, dass Value-Objekte durch ihren Inhalt charakterisiert sind und "Gleichheit" im Sinne von equals() bedeutet daher für einen Value-Typ "Gleichheit des Inhalts". Das ist bei Entity-Typen anders. Dort bedeutet "Gleichheit", dass zwei Objekte identisch sind, d.h. nicht nur den gleichen Inhalt haben, sondern ein und dasselbe Objekt sind.

In der Implementierung der equals()-Methode ist man nicht völlig frei, sondern die Implementierung muss den sogenannten "equals()-Contract" erfüllen. Den equals()-Contract findet man in der JDK JavaDoc unter Object.equals(). Er enthält neben der Forderung der eigentlichen Funktionalität, nämlich dass equals() auf Gleichheit zweier Objekte prüfen soll, folgende 5 Regeln:

- Jedes Objekt liefert beim Vergleich mit sich selbst true.

- Es ist egal, ob man x mit y vergleicht, oder y mit x; das Ergebnis ist dasselbe.
- Wenn x gleich y ist und y gleich z, dann sind auch x und z gleich.
- Man kann zwei Objekte beliebig oft miteinander vergleichen; es kommt immer dasselbe heraus, solange sich die Objekte nicht verändern.
- Alle Objekte sind von null verschieden.

Man sollte stets darauf achten, dass equals() konform zu diesen Regeln implementiert wird. Wenn eine Implementierung davon abweicht, dann sind Probleme unvermeidbar, weil sich alle Benutzer von equals() intuitiv auf die Eigenschaften verlassen, die der equals()-Contract formal beschreibt.

Wir haben im letzten Artikel eine Anleitung zum Implementieren von equals() gegeben. Hier noch einmal die wesentlichen Elemente:

- *Signatur* . Die equals()-Methode der eigenen Klasse sollte dieselbe Signatur haben wie Object.equals(), nämlich public boolean equals(Object other).
- *Alias-Prüfung.* Aus Optimierungsgründen kann man als erstes auf Identität von this und other prüfen.

```
public boolean equals(Object other) {
  if (this == other)
     return true;
  ...
}
```

- *Aufgabenteilung in Klassenhierarchien* . Direkte und indirekte Subklassen von Object übernehmen unterschiedliche Aufgaben:
- *Indirekte Subklassen: Delegation an super* . Die Prüfung, ob die geerbten Anteile des Objekts gleich sind, wird an die Superklasse delegiert.

```
boolean equals(Object other) {
  ...
  if (!super.equals(other))
    return false;
  ...
}
```

- *Direkte Subklassen: Test auf null* . Es wird geprüft, ob other eine null-Referenz ist, damit sichergestellt ist, dass es nicht zu einer NullPointerException kommt.

```
public boolean equals(Object other) {
  ...
  if (other == null)
     return false;
```

```
  ...
}
```

- *Direkte Subklassen: Test auf Vergleichbarkeit* . Es wird geprüft, ob this und other vergleichbar sind. Das ist der Fall, wenn sie vom selben Typ sind. Wir haben folgenden Test vorgeschlagen:

```
public boolean equals(Object other) {
  ...
  if (other.getClass() != getClass())
    return false;
  ...
}
```

Daneben gibt es andere Techniken, bei denen versucht wird, den Vergleich zwischen Sub- und Superobjekten zu erlauben. Diese anderen Techniken und ihre Vor- und Nachteile wollen wir in diesem Artikel im Detail betrachten.

- *Vergleich der Felder* . Nach den für direkte und indirekte Klassen unterschiedlichen Aufgaben folgt bei allen Klassen der eigentliche Vergleich der Felder.

Am Ende, wenn alle Test erfolgreich waren, wird true zurückgegeben.

Wenn man diesem Muster folgt, dann kann man sich beruhigt zurücklehnen; die resultierenden Implementierungen von equals() sind konform zum equals()-Contract und damit garantiert korrekt. Allerdings gibt es Alternativen zu einigen der oben aufgeführten Aufgaben. Besonders heikel ist der Vergleichbarkeitstests, den wir mit Hilfe der getClass()-Methode vorgeschlagen haben. Im Folgenden sehen wir uns diese Alternativen näher an, insbesondere weil alternative Lösungen sehr häufig vorkommen und leider fast ebenso häufig inkorrekt oder stark situationsabhängig und deshalb fragil sind.

Test auf Vergleichbarkeit

Warum haben wir den Test auf Vergleichbarkeit überhaupt gemacht? Die equals()-Methode akzeptiert ein Argument vom Typ Object, d.h. eine Referenz auf ein Objekt eines beliebigen Typs (ausgenommen sind natürlich primitiven Typen, da sie nicht von Objekt abgeleitet sind). Mit anderen Worten, es ist zunächst einmal nicht sicher gestellt, dass this und other vom selben Typ sind und überhaupt miteinander verglichen werden können. Deshalb macht man den Test auf Vergleichbarkeit.

Wir haben einen sehr strengen Test gemacht, nämlich den Test auf Typgleichheit per getClass():

```
public boolean equals(Object other) {
  ...
  if (other.getClass() != getClass())
    return false;
  ...
}
```

Mit dieser Implementierung von equals() sind nur Objekte vom gleichen Typ miteinander vergleichbar. Das ist insofern etwas rigide, weil es unter Umständen auch Sinn machen kann, Objekte unterschiedlichen Typs miteinander zu vergleichen. Beispielweise könnte man zulassen, dass Objekte von Sub- und Superklassen miteinander verglichen werden können. Immerhin haben Sub- und Superobjekte etwas gemeinsam, das man miteinander vergleichen könnte, nämlich die Felder der Superklasse.

Vergleich von Sub- und Superobjekten

Betrachten wir also eine Klassenhierarchie und nehmen wir an, der Vergleichbarkeitstest in equals() sei so implementiert, wie wir es vorgeschlagen hatten, nämlich per getClass():

```
class Super {
 public boolean equals(Object other) {
  ...
  if (other.getClass() != getClass())  // Vergleichbarkeitstest
    return false;
  ...
 }
}

class Sub extends Super {
 public boolean equals(Object other) {
  ...
    if (!super.equals(other))          // Delegation an Superklasse
    return false;
  ...
 }
}
```

Man beachte, dass der Vergleichbarkeitstest nur in der direkten Subklasse von Object gemacht wird, weil nach der vorgeschlagenen Anleitung dieser Test wegen des rekursiven Aufrufs von equals() nur einmal, nämlich in der obersten Superklasse, gemacht werden muss. Dann liefert der Vergleich von Sub- und Superklassenobjekte immer das gleiche Ergebnis, nämlich false:

```
Super s1 = new Super();
Sub   s2 = new Sub();
... super.equals(sub) ...  // immer false
```

Das liegt daran, dass beim Vergleich geprüft wird, ob sub.getClass() gleich super.getClass() ist, und das immer falsch, weil die beiden Objekte tatsächlich von verschiedenem Typ sind. (getClass() ist eine Methode, die in Object definiert ist. Sie liefert ein eindeutiges Objekt vom Typ Class zurück, welches den dynamischen Typ des Objekts (d.h. den Typ zur Laufzeit im Gegensatz zum statischen Typ zur Compilezeit) repräsentiert. Objekte von verschiedenem Typ haben verschiedene Class-Objekte.)

Um zu erreichen, dass der Vergleich true liefert, wenn der Superklassenanteil der beiden Objekte gleich ist, muss equals() anders implementiert werden. Man prüft dann nicht mit der getClass()-Methode, sondern mit dem instanceof-Operator. (Der instanceof-Operator prüft, ob

das Objekt auf der linken Seite vom gleichen oder von einem Subtyp des Typs auf der rechten Seite ist.) Hier ist ein typisches Beispiel:

```
class Super {
 public boolean equals(Object other) {
  ...
  if (!other instanceof Super)
    return false;
  ...
 }
}
class Sub {
 public boolean equals(Object other) {
  ...
  if (!other instanceof Sub)
    return false;
  ...
 }
}
```

Damit besteht die Chance, dass der Vergleich auch einmal true liefert, nämlich dann, wenn die gemeinsamen Felder gleich sind.

```
Super s1 = new Super();
Sub   s2 = new Sub();
... super.equals(sub) ...  // möglicherweise true
```

Der Vergleichbarkeitstest in equals() prüft auf sub instanceof Super, d.h. ob das Objekt sub von einem Typ ist, der gleich oder abgeleitet ist vom Typ Super. Das ist hier der Fall, und dann werden die übrigen Prüfungen durchgeführt, um schließlich zu entscheiden, ob sub und super "gleich" sind.

Man beachte, dass es bei der Verwendung des instanceof-Operators nicht genügt, den Test nur einmal in der obersten Superklasse zu machen, weil hier "hard-coded" der Name der Klasse, für die das equals() jeweils implementiert wird, in die Prüfung eingeht.

Das Symmetrie-Problem der instanceof-Lösung

Leider ist diese Implementierung inkorrekt: sie verletzt die Symmetrie-Anforderung aus dem equals()-Contract. Wenn man nicht super mit sub vergleicht, sondern umgekehrt, dann kommt u.U. ein anderes Ergebnis heraus, was nicht sein darf.

```
Super s1 = new Super();
Sub   s2 = new Sub();
... super.equals(sub) ...  // möglicherweise true
... sub.equals(super) ...  // immer false
```

Der Vergleichbarkeitstest im ersten Aufruf von equals() prüft auf sub instanceof Super, d.h. ob das Objekt sub von einem Typ ist, der gleich oder abgeleitet ist vom Typ Super. Das ist der Fall. Der Vergleichbarkeitstest im zweiten Aufruf von equals() prüft auf super instanceof Sub, d.h.

ob das Objekt super von einem Typ ist, der gleich oder abgeleitet ist vom Typ Sub. Das ist natürlich falsch; deshalb kommt hier immer false heraus, ganz egal, ob die beiden Objekte einen gleichen Superklassenanteil haben, oder nicht. Das ist eine klare Verletzung der Symmetrieanforderung des equals()-Contract und diese Implementierung ist inkorrekt.

Wenn diese Lösung falsch ist, warum zeigen wir sie dann? Diese Art der Implementierung ist leider so weit verbreitet, dass wir sie in zahllosen Veröffentlichungen gesehen haben und auch in realem Source-Code, beispielsweise in frühen Versionen der JDK-Klasses. Heute ist dieses Symmetrie-Problem allgemein bekannt, aber leider gibt es immer noch unzählige Beispiele inkorrekter und problematischer Implementierungen von equals(), die zwar nicht alle asymmetrisch sind, sondern andere Probleme haben, aber man findet sie sowohl in der gängigen Literatur als auch in den Standard Bibliotheken des JDK. Schauen wir uns also einfach mal um.

Stöbern in Büchern und Source-Code

Wir haben verschiedene Bücher aus dem Regal genommen und geöffnet. Wir haben public-domain Java-Source-Code angesehen. Dabei haben wir auf Anhieb ein ganze Reihe von problematischen Implementierungen gefunden. Wir haben nicht etwa verzweifelt nach pathologische Fällen oder ganz besonders miserablen Veröffentlichen gesucht, sondern wir haben wahllos zugegriffen. Im Folgenden wollen wir eine Auswahl der gefundenen "Stilblüten" zeigen und daran die Problem von equals()-Implementierungen diskutieren. Das tun wir nicht, weil die genannten Bücher so besonders schlecht sind (das ist nicht der Fall; manche der Bücher sind sogar sehr, sehr gut und trotz der Fehler absolut empfehlenswert) oder weil wir natürlich immer alles besser wissen als James Gosling, sondern um zu zeigen, dass selbst Java-Gurus die Problematik korrekter equals()-Implementierungen und ihrer Folgen unterschätzen. Die Materie ist nicht-trivial und man muss ganz sorgfältig nachdenken, wenn man sich zukünftigen Ärger nach Möglichkeit ersparen will.

Wir betrachten Beispiele aus "Program Development in Java" von Barbara Liskov und John Guttag (siehe /LIS/), "Effective Java" von Joshua Bloch (siehe /BLO/), "Practical Java" von Peter Haggar (siehe /HAG/), und aus dem JDK 1.3 Sourcecode (siehe /JDK/ ; Autoren: James Gosling, Arthur van Hoff, Alan Liu). Listing 1 bis 4 zeigen die Beispiele.

Listing 1: Barbara Liskov, "Program Development in Java", Seite 182, siehe /LIS/

```
public class Point3 extends Point2 {
 private int z;
 ...
 public boolean equals(Object p) { // overriding definition
   if (p instanceof Point3) return equals((Point3)p);
   return super.equals();
 }
 public boolean equals(Point2 p) { // overriding definition
   if (p instanceof Point3) return equals((Point3)p);
   return super.equals();
 }
```

```
 public boolean equals(Point3 p) { // extra definition
   if (p==null || z!=p.z) return false;
   return super.equals();
 }
 ...
}
```

Listing 2: JDK 1.3, package java.util, class Date, Autoren: James Gosling, Arthur van Hoff, Alan Liu, siehe /JDK/

```
public class Date implements java.io.Serializable, Cloneable,
Comparable {
   private transient Calendar cal;
   private transient long fastTime;
   private static Calendar staticCal = null;
   // ... lots of static fields ...
   ...
   public long getTime() {
       return getTimeImpl();
   }
   private final long getTimeImpl() {
       return (cal == null) ? fastTime : cal.getTimeInMillis();
   }
   ...
   public boolean equals(Object obj) {
       return obj instanceof Date && getTime() == ((Date)
obj).getTime();
   }
}
```

Listing 3: Josh Bloch, "Effective Java", Item 7 und Item 8, siehe /BLO/

```
public final class PhoneNumber {
 private final short areaCode;
 private final short exchange;
 private final short extension;
 ...
 public boolean equals(Object o) {
   if (o==this)
      return true;
   if (!(o instanceof PhoneNumber))
      return false;
   PhoneNumber pn = (PhoneNumber)o;
   return pn.extensions == extension &&
          pn.exchange   == exchange  &&
          pn.areaCode   == areaCode;
 }
 ...
}
```

Listing 4: Peter Haggar, "Practical Java", Praxis 8 bis Praxis 14, siehe /HAG/

```
class Golfball {
  private String brand;
  private String make;
  private int compression;
  ...
  public String brand() {
    return brand;
  }
  ...
  public boolean equals(object obj) {
    if (this == obj)
       return true;
    if (obj!=null && getClass() == obj.getClass())
    {
       Golfball gb = (Golfball)obj; // Classes are equal, downcast.
       if (brand.equals(gb.brand())  &&  // Compare atrributes.
           make.equals(gb.make()) &&
           compression == gb.compression())
         return true;
    }
    return false;
  }
}
class MyGolfball extends Golfball {
 public final static byte TwoPiece = 0;
 public final static byte ThreePiece = 1;
 private byte ballConstruction;
 ...
 public byte constuction() {
   return ballConstruction;
 }
 ...
 public boolean equals(Object obj) {
   if (super.equals(obj))
   {
     MyGolfball bg = (MyGolfball)obj;  // Classes equal, downcast.
     if (ballConstruction == gb.construction())
       return true;
   }
   return false;
 }
}
```

Sehen wir uns die Beispiele einmal der Reihe nach an.

Listing 1: Eine inkorrekte Version von equals()

Schon auf den ersten Blick fällt auf, dass Barbara Liskov eine interessante Kombination von Überladen und Überschreiben verwendet. Da gibt es nicht nur eine Version von equals(),

sondern gleich mehrere, mit unterschiedlichen Signaturen. So kann man's auch machen. Das hier vorgeschlagen Verfahren ist aber ein relativ ungewöhnlicher Ansatz, der sowohl Vor- als auch Nachteile hat. Das wollen wir hier aber gar nicht diskutieren. Unser Interesse gilt im Moment dem Vergleichbarkeitstest.

Wenn man genau hinsieht, erkennt man, dass in dieser Implementierung equals() nicht transitiv und damit inkorrekt ist. Das Beispiel in Listing 1 zeigt eine Subklasse Point3, die von einer Superklasse Point2 abgeleitet ist. Die Klasse Point3 hat drei überladene Versionen von equals() zu bieten. Schauen wir mal, was passiert, wenn man diese equals()-Methoden benutzt. Nehmen wir an, wir haben Point2- und Point3-Objekte mit gleichem Superklassenanteil:

Wenn wir diese Objekte miteinander vergleichen, ergibt sich das Folgende:

```
System.out.println(p1.equals(origin));  // ruft Point3.equals(Point2)
System.out.println(origin.equals(p2));  // ruft Point2.equals(Point2)
System.out.println(p1.equals(p2));      // ruft Point3.equals(Point3)
```

Man würde erwarten, dass alle drei Objekte gleich sind, weil sie den gleichen Superklassenanteil enthalten, und daher folgende Ausgabe erwarten:

```
true
true
true
```

Statt dessen ergibt sich folgende Ausgabe:

```
true
true
false
```

Gemäß dieser equals()-Implementierung sind die Point3-Objekte p1 und p2 nicht gleich, obwohl sie beide gleich mit dem Point2-Objekt origin sind. Nach den Regeln der Transitivität müssten die Point3-Objekte p1 und p2 aber gleich sein, da der Vergleich mit einem dritten Objekt in beiden Fällen true liefert. Diese equals()-Implementierung ist nicht transitiv und verletzt den equals()-Contract und ist damit inkorrekt. Ist das ein Problem? Es ist doch offensichtlich, dass p1 und p2 nicht gleich sind. Vielleicht ist der ganze equals()-Contract Blödsinn ... ?!?!?

Der equals()-Contract ist keinesfalls blödsinnig; das Beispiel ist vielleicht etwas zu simpel. Man stelle sich eine geringfügig komplexere Situation vor, in der drei Superklassen-Referenzen miteinander verglichen werden, ohne dass man weiß, welche Art von Objekt da genau referenziert wird. Wenn man drei Point2-Referenzen miteinander vergleicht und zwei der Vergleiche true liefern, aber der dritte liefert plötzlich false, dann ist das schon sehr überraschend und kann leicht zu Fehlern führen.

Worin besteht also genau das Problem? Die Verletzung der Transitivitätsanforderung stammt daher, dass hier verschiedene Implementierungen von equals() gerufen werden und diese

verschiedenen Implementierungen unterschiedliche Semantik haben. Sehen wir uns das noch einmal im Detail an.

Im ersten Vergleich p1.equals(origin) wird die Methode Point3.equals(Point2) gerufen. Sie ist wie folgt implementiert:

```
public class Point3 extends Point2 {
  ...
  public boolean equals(Point2 p) { // overriding definition
    if (p instanceof Point3) return equals((Point3)p);
    return super.equals();
  }
  ...
}
```

Die Methode Point3.equals(Point2) prüft, ob das andere Objekt (also origin) von einem Typ ist der ein Subtyp von Point3 ist. Das ist falsch; es ist genau anders herum: der Typ von origin ist der Supertyp von this. Das Ergebnis des Test ist also false; dann wird super.equals() gerufen. Die Implementierung von super.equals(), welches in diesem Fall Point2.equals(Point2) ist, wird in Listing 1 nicht gezeigt, aber man kann annehmen, dass sie der gleichen Logik folgt. Die Methode Point2.equals(Point2) vergleicht Point2-Objekte miteinander, in diesem Falle origin mit dem Superklassenanteil von p1. Das Ergebnis ist erwartungsgemäß true. Dasselbe ergibt sich, wenn origin mit p2 verglichen wird. Beide Vergleiche sind gemischte Vergleiche zwischen Super- und Subobjekten. Die Semantik dieser "mixed-type"-Vergleiche ist der Vergleich der Superklassenanteile der beteiligten Objekte, was wir im Folgenden als "slice comparison" bezeichnen werden.

Für den dritten Vergleich p1.equals(p2) wird die Methode Point3.equals(Point3) gerufen. Dieser Vergleich müsste nun nach den Regeln der Transitivität true liefert. Das tut er aber nicht. Die beiden Objekte sind ja auch nicht gleich. Was stimmt hier also nicht? Die Methode Point3.equals(Point3) führt einen ganz anderen Vergleich durch als die Methode Point3.equals(Point2), die in den beiden "mixed-type"-Vergleichen verwendet wurde. Die Methode Point3.equals(Point3) macht einen "same-type"-Vergleich, d.h. sie vergleicht Objekte desselben Typs und vergleicht dabei die gesamten Objekte, nicht nur den Superklassenanteil davon. Semantisch ist das ein ganz anderer Vergleich.

Und genau hier liegt das Problem. Alle beteiligten Methoden heißen equals(), aber sie machen ganz unterschiedliche Dinge. Je nach Konstellation wird mal die Superklassen-Variante von equals() gerufen, die beim "mixed-type"-Vergleich nur Superklassenanteile vergleicht, und mal wird die Subklassen-Variante von equals() gerufen, die alle Felder in Betracht zieht und deshalb zu anderen, inkompatiblen Ergebnissen kommt. Es ist konzeptionell nicht möglich, einen korrekten, transitiven Vergleich von Sub- und Superklassenobjekte zu definieren, wenn mal nur die Superklassenanteile und mal die gesamten Subobjekte betrachtet werden. Der Widerspruch ist weder theoretisch noch praktisch aufzulösen. Das heißt, solche Implementierungen wie die aus Listing 1 sind immer inkorrekt, da sie nicht transitiv sind.

Wenn man "mixed-type"-Vergleiche in Klassenhierarchien zulassen will, dann müssen alle Vergleiche in der gesamte Hierarchie dieselbe Semantik haben. Man könnte beispielsweise

eine isEqualToPoint2()-Methode definieren, die in der Superklasse Point2 als final definiert ist und daher in Subklassen nicht überschrieben werden kann. Diese isEqualToPoint2()-Methode würde die Point2-Anteile von Point-Objekten jeder Art vergleicht. Das wäre auf allen Stufen der Hierarchie derselbe Vergleich. Dann wären auch p1 und p2 aus unserem Beispiel "gleich" im Sinne von isEqualToPoint2(), weil ja nur die Superklassenanteile verglichen würden. Eine solche Vergleichsmethode erfüllt alle Kriterien des equals()-Contract, mit der winzigen Einschränkung, dass sie einen vielleicht etwas eigenartigen Begriff von Gleichheit implementiert, nämlich die Gleichheit des Superklassenanteils aller Point-Objekte. Deshalb haben wir die Methode auch nicht equals() genannt.

Fazit: Symmetrische Implementierungen von equals(), die den Vergleich von Sub- und Superobjekten per instanceof-Operator zulassen und in den verschiedenen Stufen der Klassenhierarchie überschrieben werden, sind immer intransitiv und damit inkorrekt. Implementierungen von equals(), die nur den Vergleich von Objekten gleichen Typs per getClass()-Methode zulassen, sind dagegen unkritisch: sie haben das oben beschilderte Intransitivitätsproblem nicht.
Asymmetrische Implementierungen von equals() mit instanceof haben wir bis jetzt noch nicht betrachtet; das tun wir im nächsten Abschnitt. Asymmetrische Implementierungen können transitiv sein, aber sie sind natürlich auch nicht korrekt, wegen der fehlenden Symmetrie.

Listing 2: Eine weit verbreitete, aber dennoch fragwürdige Version von equals()

Nun, die Implementierung in Listing 1 war ohnehin etwas exotisch, schon allein durch den Mix von Überladen und Überschreiben. Vielleicht war das ja ein Ausreißer. Schauen wir uns mal was Reelles an und sehen uns Klassen aus den JDK-Bibliotheken an. Ein typisches Beispiel ist in Listing 2 gezeigt.

Es handelt sich um das Beispiel einer non-final Klasse die einen Value-Typ repräsentiert, nämlich die Klasse Date. Die Klasse Date mit ihrer Implementierung der equals()-Methode ist zunächst einmal korrekt, so wie sie ist. Probleme sind aber zu erwarten, sobald jemand von der Klasse Date ableitet. Man beachte, dass die Klasse Date offenbar bewusst als potentielle Superklasse deklariert ist; sie ist nämlich nicht als final deklariert. Dann leiten wir doch einmal eine Subklasse ab und sehen, was passiert.

Nehmen wir an, die Subklasse hat zusätzliche Felder und muss aus diesem Grunde die Implementierung von Date.equals() überschreiben, so dass auch die neuen Felder in den Vergleich eingehen. Hier ist eine vorstellbare Subklasse NamedDate:

```
public class NamedDate extends Date {
  private String name;
  public boolean equals(Object other) {
    if (other instanceof NamedDate &&
!name.equals(((NamedDate)other).name))
       return false;
    return super.equals(other));
  }
}
```

Natürlich kann man NamedDate.equals() auch anders implementieren, aber wir folgen hier dem Stil, den die Superklasse Date nahelegt. Zur Erinnerung, hier die Implementierung von Date.equals():

```
public class Date {
  ...
  public boolean equals(Object obj) {
    return obj instanceof Date && getTime() == ((Date) obj).getTime();
  }
}
```

Beide Versionen von equals() benutzen den instanceof-Operator und lassen den "mixed-type"-Vergleich zu. Betrachten wir ein Beispiel:

```
EndOfMillenium.equals(NewYearsEve)   // slice comparison: true
NewYearsEve.equals(TheEnd)           // slice comparison: true
EndOfMillenium.equals(TheEnd)        // whole-object comparison: false
```

Wir beobachten dasselbe Transitivitätsproblem wie in Listing 1. Das liegt daran, dass Date.equals() den Vergleich von Sub- mit Superobjekten erlaubt, aber jede Subklasse mit Wahrscheinlichkeit equals() überschreiben wird und damit einen semantisch anderen Vergleich implementieren wird. Und dann gibt es die oben schon ausführlich geschilderten Probleme.

Vielleicht war es ja falsch, die equals()-Methode der Subklasse NamedDate mit Hilfe des instanceof-Operators zu implementieren. Wie wäre es denn, wenn man NamedDate.equals() mit Hilfe von getClass() implementierte? Das wäre auch inkorrekt, denn dann wäre equals() nicht mehr symmetrisch. Man könnte Date-Objekte zwar mit NamedDate-Objekten per Date.equals() vergleichen, aber umgekehrt per NamedDate.equals() geht es nicht, oder genauer gesagt, es kommt immer false heraus, egal ob die beiden Objekte die gleichen Superklassenfelder haben oder nicht. Das ist eine Verletzung der Symmetrie-Anforderung aus dem equals()-Contract und damit also auch falsch.

Die Autoren dieser Date-Klasse hätten ihren Nutzern den Ärger mit inkorrekten Implementierungen von equals() in Subklassen ersparen können und die Date-Klasse oder zumindest die Methode Date.equals() als final deklarieren können. Dann wäre die Klasse Date keine Superklasse oder aber man könnte zwar ableiten, aber equals() nicht überschreiben. In beiden Fällen wäre das Transitivitätsproblem ausgeschlossen. Leider habe die Autoren das nicht getan. Also muss man als JDK-Benutzer aufpassen, dass man nicht in diese Falle tappt: man sollte von solchen Bibliotheksklassen wie Date, die in ihrer Implementierung von equals() den instanceof-Operator verwenden, nur mit allergrößter Vorsicht ableiten. Wenn man ableiten will, muss man auf jeden Fall erst einmal den Source-Code der anvisierten Superklasse studieren und feststellen, wie equals() dort überhaupt implementiert ist; in der JavaDoc steht dazu üblicherweise überhaupt nichts.

Wenn man dann weiß, dass das equals() der Klasse, von der man ableiten will, den instanceof-Operator verwendet, dann macht eine Subklasse nur Sinn, wenn sie keine zusätzlichen Felder

definiert und auch sonst nichts hinzufügt, was eine neue Version von equals() erfordern würde. Nur dann, wenn das Superklassen-equals() in der Subklasse nicht überschrieben werden muss, macht eine Subklasse überhaupt Sinn. (Tipp: als Autor einer solchen Subklasse sollte man mit Rücksicht auf seine eigenen Nutzer den Fehler nicht wiederholen und seine Subklasse als final deklarieren oder ein Dummy-equals() implementieren, welches final ist und nichts weiter tut, als an super.equals() zu delegieren.)

Man fragt sich, warum diese Fallen im JDK so zahlreich sind. Die Klasse Date ist lediglich ein wahllos herausgegriffenes Beispiel; es gibt viele davon im JDK 1.3. Einer der Java-Gurus bei Sun (nämlich Joshua Bloch, der Autor von "Effective Java" und Entwickler des Collection-Frameworks) hat dazu in einem privaten Email gesagt, man habe ja erst vor kurzem überhaupt erkannt, dass das ein Problem sei mit dem instanceof-Test in den non-final Klassen. Und außerdem sei das ja eigentlich auch gar kein Problem; es käme schließlich so gut wie nie vor, dass man beim Ableiten neue Felder hinzufügt. (Siehe dazu auch Item 14 aus seinem Buch /BLO/ .) So selten ist das Hinzufügen von Feldern beim Definieren von Subklassen unserer Erfahrung nach nicht, aber für gewisse Projekte mag Joshua Bloch's Einschätzung trotzdem richtig sein.

Wir würden aber dennoch empfehlen, den Implementierungsstil von JDK-Klassen wie Date für eigene Klassen nicht bedenkenlos nachzuahmen. Man vermeidet viel Aufwand und Kopfschmerzen, wenn man Klassen gar nicht erst als potentielle Superklassen zulässt, d.h. eine Klasse sollte normalerweise final sein, es sei denn, sie ist wirklich als Superklasse gemeint. Das Design von Superklassen ist deutlich aufwendiger und schwieriger als das Design "normaler" non-final Klassen. Man muss nicht nur bei der equals()-Methode von Superklassen besonders aufpassen; das gleiche gilt auch für die clone()-Methode eine Superklasse. Es muss generell fein säuberlich dokumentiert werden, wie und wann und unter welchen Umständen non-final Methoden in den Implementierungen anderer Methoden der Superklasse verwendet werden und was von diesen non-final Methoden erwartet wird; schließlich muss das der Autor einer Subklasse wissen und beachten, falls er die Methoden in seiner Subklasse überschreibt. Da in Java alle Klassen per Default non-final und damit potentielle Superklassen sind, passiert es relativ häufig, dass der Autor einer non-final Klasse überhaupt nicht darüber nachgedacht hat, ob seine Klasse als Superklasse taugt, sondern die Klasse ist nur "zufällig" eine potentielle Superklasse. Ob eine solche Klasse zum Ableiten geeignet ist, ist fraglich.

Zurück zu unseren Literaturstudien. Sehen wir uns die verbleibenden beiden Beispiele an.

Listing 3: Eine korrekte Version von equals()

Das ist das Beispiel einer PhoneNumber-Klasse, die man im Buch von Joshua Bloch findet. Die Implementierung von equals() verwendet zwar den instanceof-Test, aber der ist hier unproblematisch, weil die Klasse final ist. Joshua Bloch vermeidet so alle oben diskutierten Transitivitätsprobleme. Die Lösung ist korrekt, aber in ihrer Benutzbarkeit auf final-Klassen beschränkt.

Listing 4: Noch eine korrekte Version von equals()

Das ist ein Beispiel aus dem Buch von Peter Haggar. Wir sehen in Listing 4 das Beispiel einer Superklasse Golfball und ihrer Subklasse MyGolfball. Die Subklasse überschreibt equals(), weil sie Subklassen-spezifische Felder hat. Die Superklasse benutzt nicht den instanceof-Test, sondern verwendet getClass():

```
class Golfball {
   ...
   public boolean equals(object obj) {
     if (this == obj)
        return true;
     if (obj!=null && getClass() == obj.getClass())
     {
        Golfball gb = (Golfball)obj; // Classes are equal, downcast.
        if (brand.equals(gb.brand())  &&  // Compare atrributes.
            make.equals(gb.make()) &&
            compression == gb.compression())
          return true;
     }
     return false;
   }
}
class MyGolfball extends Golfball {
  ...
  public boolean equals(Object obj) {
    if (super.equals(obj))
    {
      MyGolfball gb = (MyGolfball)obj;  // Classes equal, downcast.
      if (ballConstruction == gb.construction())
        return true;
    }
    return false;
  }
}
```

Das ist genau die Art von Implementierung, die wir empfehlen würden. Der fundamentale Unterschied zu all den anderen betrachteten Beispielen ist, dass hier der Vergleich von Golfball- mit MyGolfball-Objekten grundsätzlich fehlschlägt. Hier ist ein Beispiel:

```
gb1.equals(original)  // mixed-type comparison: yields false
original.equals(gb2)  // mixed-type comparison: yields false
gb1.equals(gb2)       // same-type comparison:  yields false
```

Diese Art der Implementierung ist korrekt und erfüllt alle Anforderungen des equals()-Contract. In Klassenhierarchien, in denen equals() überschrieben werden kann und muss, ist das die sinnvollste Strategie.

Anmerkung: Man kann im übrigen die beiden Implementierungstechniken (instanceof vs. getClass()) nicht innerhalb einer Klassenhierarchie mischen. Würden wir beispielsweise versuchen, die NamedDate-Klasse (siehe Listing 2 und anschließende Diskussion) mit

getClass() zu implementieren, während die Superklasse Date den instanceof-Operator verwendet, dann produzieren wir ein asymmetrisches equals() für diese Klassenhierarchie: ein NamedDate ließe sich mit einem Date vergleichen, aber nicht umgekehrt. Der Designer der Superklasse trägt also große Verantwortung: er legt die Implementierungsstrategie für die equals()-Methoden in der gesamte Hierarchie fest.

Schlussfolgerung

Es gibt zwei Möglichkeiten, den Vergleichbarkeitstest in equals() zu machen: mit Hilfe der getClass()-Methode oder mit Hilfe des instanceof-Operators.

- Die getClass()-Technik ist robust und unproblematisch und daher zu empfehlen.
- Die instanceof-Technik ist weit verbreitet, macht aber nur bei Klassen Sinn, die final sind.

Man findet in der Literatur und auch in der Praxis viele non-final Klassen, die trotz der Transitivitätsprobleme den instanceof-Test in ihrer non-final Methode equals() machen. Das Ableiten von solchen Klassen ist äußerst fehleranfällig und sollte grundsätzlich vermieden werden. (Man kann in vielen Fällen bei der Definition neuer Klassen Ableitung durch Delegation ersetzen; es muss nicht immer Vererbung sein.)

Für die Implementierung eigener Klassen hält man sich am besten an die obige Regel: bei final-Klassen ist es egal, wie man den Vergleichbarkeitstest in equals() macht, und bei non-final Klassen nehme man getClass().

Ausblick

Wir haben in diesem und dem vorangegangenen Artikel einige Aspekte der Implementierung von equals() besprochen. Das Thema ist damit noch nicht erschöpfend behandelt. Wir haben zum Beispiel bislang kaum erwähnt, dass equals() nicht allein auf der Welt ist und Querbezüge zu anderen Infrastruktur-Methoden wie hashCode() und compareTo() hat. Die Implementierungen dieser Methoden müssen konsistent zu equals() sein. Mehr darüber beim nächsten Mal (/KRE3/).

Nachtrag

Viele Jahre später - dieser Zusatz wurde im Juli 2009 geschrieben, also 7 Jahre nach der Veröffentlichung des obigen Artikels - wird das Thema equals() noch immer diskutiert. Mittlerweile hat sich augenscheinlich herumgesprochen, dass die Implementierung von equals() vielleicht doch nicht ganz so simpel ist, wie sie in Joshua Bloch's "Effective Java" auf den ersten Blick erscheinen mag. Diesmal wird das Thema aus dem Blickwinkel von Scala aufgerollt, rückübersetzt nach Java. Dem interessierten Leser sei daher folgender Beitrag als Ergänzung empfohlen: "How to Write an Equality Method in Java" von Martin Odersky, Lex Spoon und Bill Venners vom 1. Juni 2009 (/OSV/).

Literaturverweise

/KRE1/ **Wie, wann und warum implementiert man die equals()-Methode?**
Teil 1: Die Prinzipien der Implementierung von equals()
Klaus Kreft & Angelika Langer
Java Spektrum, Januar 2002
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/01.Equals-Part1/01.Equals.html

/KRE2/ **Wie, wann und warum implementiert man die equals()-Methode?**
Teil 2: Der Vergleichbarkeitstest
Klaus Kreft & Angelika Langer
Java Spektrum, März 2002
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/02.Equals-Part2/02.Equals2.html

/OSV/ **How to Write an Equality Method in Java**
Martin Odersky, Lex Spoon und Bill Venners
artima developer, Juni 2009
URL: http://www.artima.com/lejava/articles/equality.html

/KRE3/ **Wie, wann und warum implementiert man die hashCode()-Methode?**
Klaus Kreft & Angelika Langer
Java Spektrum, Mai 2002
URL:
http://www.AngelikaLanger.com/Articles/EffectiveJava/03.HashCode/03.HashCode.html

/KRE4/ **Secrets of equals()**
Part 1: Not all implementations of equals() are equal
Angelika Langer & Klaus Kreft
Java Solutions, April 2002
URL:
http://www.AngelikaLanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html

/KRE5/ **Secrets of equals()**
Part 2: How to implement a correct slice comparison in Java
Angelika Langer & Klaus Kreft
Java Solutions, August 2002
URL:

http://www.AngelikaLanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals-2.html

/DAV/ **Durable Java: Liberté, Égalité, Fraternité**
Mark Davis
Java Report, January 2000
URL: http://www.macchiato.com/columns/Durable5.html

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

/BLO2/ Joshua Bloch's comment on instanceof versus getClass in equals methods:
A Conversation with Josh Bloch
by Bill Venners
URL: http://www.artima.com/intv/bloch17.html

/HAG/ **Practical Java: Programming Language Guide**
Peter Haggar
Addison-Wesley, March 2000
ISBN 0201616467

/LIS/ **Program Development in Java: Abstraction, Specification, and Object-Oriented Design**
Barbara Liskov with John Guttag
Addison-Wesley, January 2000
ISBN: 0201657686

/GOF/ **Design Patterns: Elements of Reusable Object-Oriented Software**
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Addison-Wesley, January 1995
ISBN: 0201633612

/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**