Hash-Code-Berechnung
Wie, wann und warum implementiert man die hashCode()-Methode?

JavaSPEKTRUM, Mai 2002
Klaus Kreft & Angelika Langer

In den letzten beiden Artikeln dieser Kolumne (/ KRE1 / und / KRE2 /) haben wir uns mit der Methode equals() befasst. In dieser Ausgabe wollen wir und ansehen, wie und warum man die Methode hashCode() implementieren muss. equals() und hashCode() hängen eng zusammen und müssen konsistent zueinander implementiert werden. Immer dann, wenn man equals() implementiert hat, muss man auch hashCode() implementieren. Worin besteht der Zusammenhang? Was genau ist die Konsistenzanforderung? Wie implementiert man hashCode()?

Hash-basierte Container in Java

Die Methode hashCode() berechnet zu dem Objekt, auf dem sie gerufen wird, einen Hash-Code. Der Hash-Code ist ein integraler Wert, der verwendet wird, um Objekte in einem hash-basierten Container abzulegen oder sie in einem solchen Container zu finden. Die hash-basierten Container in Java sind java.util.Hashtable, java.util.HashMap, java.util.HashSet und deren Subklassen.

Hash-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 hash-basierten Containers zeigt.

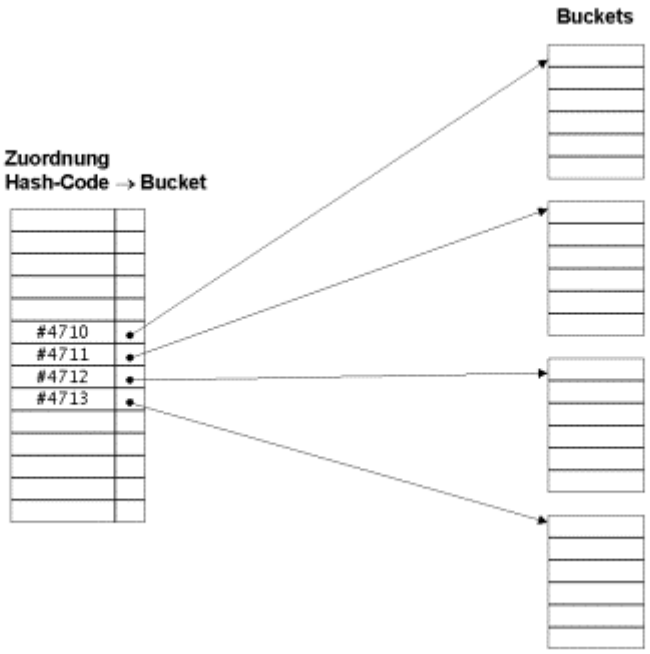

Abbildung 1: logischer Aufbau eines hash-basierten Containers

Ein hash-basierter Container ist so organisiert, dass er verschiedene Sektionen anlegt (sogenannte "buckets"), in denen die zu speichernden Objekte sequentiell abgelegt werden. Bei den hash-basierten Containern in Java ist zu beachten, dass genau genommen nicht die Objekte, sondern lediglich Referenzen auf die Objekte gespeichert werden. Einen Bucket kann man sich daher als Array oder Liste von Object-Referenzen vorstellen. Der Zugriff auf die verschiedenen Buckets erfolgt über einen integralen Index und ist damit hochperformant; er erfolgt in konstanter Zeit, d.h. der Zugriff auf den Bucket dauert immer gleich lang unabhängig von der Zahl der Elemente im Container. Innerhalb eines Buckets ist der Zugriff auf die Elemente allerdings langsam, weil er nämlich sequentiell erfolgt. Die Abhängigkeit ist hier linear, d.h. es dauert umso länger, je größer der Bucket ist. Deshalb ist ein hash-basierter Container mit vielen kleinen Buckets günstiger als einer mit wenigen großen Buckets.

Der integrale Index, der für das Auffinden des Buckets verwendet wird, bestimmt sich über den Hash-Code, den die Methode hashCode() berechnet. Das bedeutet, dass alle Objekte mit demselben Hash-Code im selben Bucket abgelegt sind. Schauen wir uns das noch einmal im Detail an. Betrachten wir als Beispiel das Einfügen eines Objekts in einen HashSet; das geht über die Methode add(Object o). Da wird zunächst der Hash-Code des Objekts o berechnet. Nehmen wir mal an, o.hashCode() liefert den Hash-Code 4711. Damit ist der Bucket identifiziert, in den das Objekt gehört. Dann wird im Bucket Nr. 4711 nachgesehen, ob es das Objekt o dort schon gibt; das ist nötig, weil im HashSet keine Duplikate erlaubt sind. Wie das Vorhandensein eines Objekts im Bucket festgestellt wird, sehen wir uns gleich noch genauer an. Wenn das Objekt im Bucket noch nicht vorhanden ist, dann wird eine Referenz auf das Objekt o im Bucket Nr. 4711 am Ende hinzugefügt. Andernfalls, wird ein Fehler gemeldet, indem der Returnwert false zurück gegeben wird.

Das Auffinden des möglichen Duplikats erfolgt wie gesagt sequentiell; es werden alle Objekte im Bucket nacheinander überprüft. Wie oben schon erwähnt, liegen in einem Bucket alle

Objekte, deren Hash-Code gleich ist. Das heißt aber nicht, dass deshalb alle Objekte im Bucket gleich sind. Es könnte beispielsweise sein, dass 2 Objekte a und b voneinander verschieden sind, aber die hashCode()-Methode berechnet denselben Hash-Code für die beiden Objekte a und b. Dann landen zwar beide Objekte im gleichen Bucket, sind aber deshalb nicht gleich. Mit Gleichheit ist hier im übrigen die Gleichheit im Sinne von equals() gemeint. In der Tat ruft die Methode add(Object o) auf jedem Element im Bucket die Methode equals() auf. Die übrigen Operationen auf einem hash-basierten funktionieren ganz analog. Es wird immer diese zweistufige Kombination von hashCode() und equals() verwendet, um auf Elemente im Container zuzugreifen.

Aus dieser Implementierung der hash-basierten Container in Java ergibt sich eine enge Beziehung zwischen den beiden Methoden equals() und hashCode(). Die beiden Methoden müssen zueinander konsistent sein. Wenn diese Konsistenz nicht gegeben ist, dann passieren seltsame Dinge, die man in erster Näherung mit "Der Hash-Container funktioniert nicht." beschreiben könnte. Details sehen wir uns später noch an.

Aus der geschilderten Organisation der hash-basierten Container ergeben sich zwei grundsätzliche Anforderung an den Algorithmus zur Hash-Code-Berechnung.

- Performanz der Hash-Code-Berechnung. Die Berechnung des Hash-Codes sollte schnell gehen. Sinn und Zweck der Hash-Code-Berechnung ist der performante index-basierte Zugriff auf den Bucket, in dem das gesuchte Objekt liegt. Wenn die Hash-Code-Berechnung aufwendig ist und lange dauert, dann ist der "schnelle" Zugriff nicht mehr schnell und alle Operationen auf dem hash-basierten Container werden schlechte Zugriffszeiten aufweisen.

- Verteilung der berechneten Hash-Codes. Die Suche innerhalb eines Buckets erfolgt sequentiell und dauert damit umso länger je größer der Bucket ist. Dieser langsame sequentielle Zugriff wirkt sich negativ auf die Performance aller Operationen auf dem hash-basierten Container aus. Um diesen negativen Effekt zu vermeiden, muss man die Buckets so klein wir möglich halten. Die Hash-Code-Berechnung sollte deshalb so sein, dass möglichst wenige Objekte im selben Bucket abgelegt werden. Idealerweise könnte für jedes Objekt ein anderer Hash-Code berechnet werden; dann gibt es sehr viele winzige Buckets mit jeweils genau einem Element. Die schlechteste Performance ergibt sich, wenn für alle Objekte der gleiche Hash-Code berechnet wird; dann gibt es nur einen riesigen Bucket, in dem alle Elemente enthalten sind.

Ein guter Hash-Code-Algorithmus muss also versuchen, eine möglichst gute Verteilung der Hash-Codes mit möglichst geringem Aufwand zu erreichen. Neben diesen grundsätzlichen Anforderungen an eine Implementierung der hashCode()-Methode muss noch auf die schon erwähnte Konsistenz zur equals()-Methode geachtet werden. Was das bedeutet, sehen wir uns im Folgenden an.

Der sogenannte hashCode-Contract

Was genau ist die Anforderung an eine Implementierung von hashCode(), die konsistent zu equals() ist? Die Anforderungen an hashCode() sind im sogenannten hashCode-Contract beschrieben. Den findet man in der JavaDoc der Java 2 Standard Edition (J2SE) unter dem Eintrag Object.hashCode. Hier ist der Originaltext:

public int hashCode()

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Das bedeutet das Folgende:

- Es ist egal, wie oft man hashCode() aufruft; es kommt immer der gleiche Wert heraus, es sei denn, der Inhalt des Objekts hat sich geändert. Das gilt aber nur für einen Programmlauf; beim nächsten Ablauf des Programms darf hashCode() einen anderen Wert produzieren.
- Wenn zwei Objekte gleich sind, dann müssen sie den gleichen Hash-Code haben.
- Wenn zwei Objekte verschieden sind, dann müssen sie deshalb keine unterschiedlichen Hash-Codes haben. Es wäre aber besser für die Performance, wenn die Hash-Codes von ungleichen Objekten verschieden wären.

Hier sieht man deutlich die enge Beziehung zwischen equals() und hashCode():

- Die "Gleichheit zweier Objekte" im hashCode-Contract ist die Gleichheit, die durch equals() definiert ist. Das heißt, aus x.equals(y) muss sich (x.hashCode() == y.hashCode()) ergeben.

- Mit "Inhalt des Objekts" ist im hashCode -Contract der Inhalt des Objekts gemeint, der für den Vergleich mittels equals() relevant ist. Was nicht zum Ergebnis von equals() beiträgt, kann auch für hashCode()ignoriert werden.

Daraus ergibt sich, dass man für eine Klasse hashCode() immer dann implementieren muss, wenn man auch equals() implementiert. Vieles von dem, was wir in den letzten beiden Artikeln für equals() besprochen haben, gilt ganz analog auch für hashCode().

Probleme mit inkorrekten Implementierungen von hashCode()

Bevor wir uns in die Details einer Implementierung von hashCode() vertiefen, widmen wir uns zunächst erst einmal der Frage: was passiert eigentlich, wenn man hashCode() nicht implementiert, oder falsch implementiert?

Ebenso wie die equals()-Methode ist auch die hashCode()-Methode bereits in der Superklasse aller Klassen, nämlich Object, definiert. Wenn man hashCode() nicht implementiert, dann erbt die Klasse die Default-Implementierung aus der Superklasse Object. Das hat zur Folge, dass man für alle Java-Objekte einen Hash-Code berechnen kann. Das heißt aber auch, dass man sich für jede Klasse überlegen muss, ob die Implementierung von hashCode() für diese Klasse korrekt ist. Was tut die geerbte Default-Implementierung denn eigentlich?

Was Object.hashCode() macht, ist nicht so genau definiert. Die JavaDoc-Beschreibung sagt dazu:

> *As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)*

Das heißt, typischerweise basiert die Default-Hash-Code-Berechnung auf der Adresse des Objekts. Das macht Sinn, weil es die Anforderungen des hashCode-Contracts erfüllt, solange auch die Default-Implementerung von equals() nicht überschrieben wird. Zur Erinnerung: die Default-Implementierung von equals() in der Klasse Object vergleicht die Adressen der beiden Objekte. Die oben spezifizierte Implementierung von hashCode() ist dazu konsistent: es geht in beide Methoden lediglich die Adresse des Objekts ein. Damit ist gewährleistet, dass gleiche Objekte (d.h. solche mit gleicher Adresse) gleiche Hash-Codes haben (nämlich solche, die sich aus der Adresse berechnen lassen). Sobald man sich entschließt, für eine Klasse die equals() -Methode zu überschreiben, dann muss man auch die hashCode()-Methode überschreiben, weil sonst mit größter Wahrscheinlichkeit der hashCode-Contract verletzt ist. Welche Auswirkungen haben solche Verletzungen des hashCode-Contracts?

Betrachten wir das Beispiel einer Klasse PhoneNumber, die zwar equals() überschreibt, aber hashCode() nicht. Die equals()-Methode wird dann den Inhalt der PhoneNumber-Objekte vergleichen, also beispielsweise Vorwahl und Anschlussnummer. Die geerbte hashCode()-Methode berechnet aber den Hash-Code aus der Adresse des Objekts. Dann passiert das Folgende, wenn man beispielsweise die Telefonnummer als Schlüssel in einer HashMap

verwenden will: man kann zwar Key-Value-Paare in die HashMap eintragen, also beispielsweise den Namen zu einer bestimmten Telefonnummer, aber man wird sie u.U. nie mehr wiederfinden.

```
Map createPhoneDirectory()
{
  Map m = new HashMap();
  ...
  m.put(new PhoneNumber(501,4375493), new String("Bodo Ballermann"));
  ...
  return m;
}

void someSomething()
{
  ...
  // find phone directory entry
  m.get(new PhoneNumber(501,4375493));  // returns false !!!
  ...
}
```

In diesem Beispiel wird als Schlüssel für das Eintragen und das Finden die gleiche Telefonnummer benutzt, nämlich (501) 437 5493. Der betreffende Eintrag zu dieser Nummer wird aber nicht gefunden, obwohl er in der HashMap vorhanden ist. Das liegt daran, dass "gleiche" PhoneNumber-Objekte nicht die gleichen Hash-Codes haben. In obigem Beispiel haben wir zwei PhoneNumber-Objekte mit gleichem Inhalt, aber mit unterschiedlichen Adressen verwendet. Die Methode HashMap.put() legt daher den Eintrag in einem Bucket ab, dessen Identifikation sich aus der Adresse des ersten PhoneNumber-Objekts ergibt, und die Methode hashMap.get() sucht den Eintrag in einem ganz anderen Bucket, dessen Identifikation sich aus der Adresse des zweiten PhoneNumber-Objekts ergibt. Dort ist der Eintrag aber nicht zu finden. Es wäre Zufall, wenn unter diesen Umständen überhaupt noch Einträge gefunden würden.

Man sieht an diesem Beispiel, dass die Verletzung der Forderung "Gleiche Objekte müssen gleiche Hash-Codes haben" dazu führt, dass die hash-basierten Container nicht funktionieren. Die Konsistenz zwischen hashCode() und equals() ist von fundamentaler Bedeutung für das Arbeiten mit hash-basierten Containern in Java. Wichtig für das Funktionieren der hash-basierten Container sind dabei die ersten beiden Anforderungen ("gleicher Hash-Code bei wiederholten Aufrufen" und "gleiche Hash-Codes für gleiche Objekte"). Die dritte Anforderung ist eigentlich keine Forderung, sondern lediglich ein Hinweis, dass die Performance besser ist, wenn die berechneten Hash-Codes für verschiedene Objekte nach Möglichkeit verschieden sind.

Hash-Codes und Persistenz

Die erste Anforderung des hashCode-Contracts enthält den Zusatz, dass die Hash-Codes bei verschiedenen Programmläufen durchaus verschieden sein dürfen. Diese Aussage erklärt sich durch die Tatsache, dass die Default-Implementierung von hashCode() in Object

typischerweise auf den Adressen der Objekte basiert. Die Adressen können natürlich bei jedem Programmlauf anders sein. Nebeneffekt dieser Tatsache ist, dass zumindest die Default-Implementierung von hashCode() nicht für Persistenz taugt.

Man könnte ja auf die Idee kommen, eine HashMap wie unser Telefonbuch aus dem obigen Beispiel persistent zu machen, indem man die Einträge in eine Datenbank schreibt oder sonstwie serialisiert und speichert. Dabei würden die Hash-Codes aus einem Programmlauf für die persistente Speicherung verwendet und beim nächsten Programmlauf wieder eingelesen. Die eingelesenen Hash-Codes sind aber unbrauchbar, weil für das gleiche Objekt diesmal ein ganz anderer Hash-Code berechnet wird.

Wenn man Hash-Codes persistent machen will, dann muss man die hashCode()-Methode entsprechend implementieren, nämlich so, dass tatsächlich immer für gleiche Objekte der gleiche Hash-Code berechnet wird. Die Default-Implementierung leistet dies nicht, und es ist auch von anderen Implementierungen der hashCode()-Methode nicht verlangt. Das heißt insbesondere, dass man das auch nicht von der hashCode()-Implementierung anderer Klassen erwarten darf.

Hash-Codes und Objekt-Referenzen

Selbst wenn man alle oben geschilderten Probleme vermieden hat und hashCode() korrekt implementiert hat, gibt es immer noch Überraschungen und Fehlerquellen, die mit der Referenz-Semantik in Java zu tun haben. In Java enthalten alle Container des JDK, inklusiver der hash-basierten Container, grundsätzlich Referenzen auf Objekte und niemals Kopien der "enthaltenen" Objekte. Infolgedessen erfolgt der Zugriff auf Elemente, die in einem hash-basierten Container abgelegt sind, immer über Referenzen. Wenn diese Referenzen verwendet werden, um das referenzierte Objekt zu modifizieren, dann ist es wahrscheinlich, dass der hash-basierte Container zerstört wird. Hier ist ein Beispiel:

```
AbstractSet mySet = new HashSet();

... do something with the set ...

if (!mySet.isEmpty()) {
  Iterator iter = mySet.iterator();
  while (iter.hasNext()) {
    Point element = (Point)iter.next();
    element.setLocation(0,0);
  }
}
```

In einem HashSet sind Point-Objekte abgelegt, oder genauer gesagt, Referenzen auf Point-Objekte. Die hasNext()-Methode des Iterators liefert sukzessive Referenzen auf alle enthaltenen Objekte. Diese Referenzen werden in dem Beispielcode verwendet, um die modifizierende Methode setLocation() der Klasse Point zu rufen, die den Inhalt des Point-Objekts verändert. Mit dem Inhalt des Objekts ändert sich aber auch der Hash-Code des Objekts und eigentlich müsste das veränderte Point-Objekt in einem anderen Bucket abgelegt sein. Das ist aber nicht der Fall; das betreffende Point-Objekt bleibt, wo es ist, und befindet

sich nach der Modifikation im falschen Bucket. Damit ist der ganze Container zerstört. Das kann sich darin äußern, dass das veränderte Point-Objekt weder im Container gefunden noch aus dem Container entfernt werden kann. Es können sich aber auch andere unerwünschte Effekte ergeben; das Verhalten von Operationen auf einem zerstörten Container ist generell undefiniert.

Die Ursache des Problems liegt in der Referenz-Semantik von Java. Der Benutzer bekommt über die Referenzen Schreib-Zugriff auf die Objekte im Container und kann die dort abgelegten Objekte an Ort und Stelle verändern. Jede Veränderung der Objekte, die das Ergebnis von hashCode()beeinflusst, zerstört aber den Container, weil das Objekt nach der Veränderung eigentlich in einem anderen Bucket abgelegt sein müsste. Das ist ein generelles Problem bei allen Containern, deren Organisation in irgendeiner Form auf dem Inhalt der gespeicherten Objekten beruht und bei denen Schreib-Zugriff auf die gespeicherten Objekte möglich ist. In diese Kategorie fallen alle hash-basierten, aber auch alle baum-basierten Container in Java. Schützen kann man sich vor dieser Falle nur durch Programmierdisziplin, indem man es unterlässt, enthaltene Objekte an Ort und Stelle im Container zu ändern. Man muss statt dessen das alte Objekte aus dem Container entfernen und das neue "modifizierte" Objekt in den Container einfügen.

Implementierung von hashCode()

Worauf muss man nun achten, wenn man hashCode() implementiert? Wichtig sind die beiden folgenden Aspekte:

- Konsistenz zu equals(). Gleiche Objekte müssen gleiche Hash-Codes haben, sonst ist das Arbeiten mit hash-basierten Containern nicht möglich, wie wir am Beispiel gesehen haben.
- Performanz. Die Hash-Code-Berechnung sollte so sein, dass die Zugriffe auf hash-basierte Container möglichst effizient sind. Das heißt zum einen, dass die Hash-Code-Berechnung selbst schnell und einfach sein muss. Zum anderen heißt es aber auch, dass die berechneten Hash-Code gleichmäßig verteilt sein sollten, damit wir einen Container mit vielen kleinen Buckets (statt wenigen großen Buckets) bekommen. Es sollen also möglichst viele verschiedene Hash-Codes auf möglichst effizientem Wege bei der Berechnung herauskommen.

Konsistenz zwischen hashCode() und equals()

Wie stellt man sicher, dass hashCode() und equals() konsistent zueinander sind? Man muss dafür sorgen, dass in die Berechnung des Hash-Codes nur die Informationen eingehen, die auch für die Implementierung von equals() berücksichtigt werden.

In obigem Beispiel der Klasse PhoneNumber mit überschriebenem equals() und geerbtem hashCode() ist dies verletzt. Die geerbte Default-.Implementierung von hashCode() basiert auf der Adresse des Objekts. Die Adresse spielt aber für die Implementierung des überschriebenen equals() keine Rolle, da ein korrektes equals() die Felder des Objekts vergleicht und sich für die

Adresse des Objekts überhaupt nicht interessiert. Unter diesen Umständen ist es nicht gewährleistet, dass gleiche Objekte gleiche Hash-Codes haben.

Normalerweise wird man also all die Felder in die Hash-Code-Berechnung einbeziehen, die in equals() miteinander verglichen werden und man wird in hashCode() alles ignorieren, was in equals() nicht vorkommt.

Das heißt zum Beispiel, dass man die Adresse nicht zur Hash-Code-Berechnung heranziehen darf, wenn die Adresse nicht zur Gleichheit beiträgt. Es heißt aber auch, dass man transiente Felder nicht für die Hash-Code-Berechnung berücksichtigen darf. Transiente Felder tragen zum logischen Inhalt eines Objekts nichts bei und werden deshalb in Implementierungen von equals() ignoriert (siehe /KRE/). Aus diesem Grunde müssen sie auch für die Implementierung von hashCode() ignoriert werden.

Das bedeutet aber nicht, dass alle Informationen, die in equals() berücksichtigt werden, auch in hashCode() berücksichtigt werden müssen. Man kann einen Teil der Information, also beispielsweise einen Teil der Felder, für die Hash-Code-Berechnung ignorieren. Das führt zwar dazu, dass ungleiche Objekte gleiche Hash-Codes haben, aber das ist nach der dritten Regel im hashCode-Contract ausdrücklich erlaubt, und manchmal ist es auch durchaus sinnvoll in Hinblick auf die Performanz.

Performanz von hashCode()

Der Zugriff auf Elemente im einem hash-basierten Container geschieht über eine rasche Identifikation des Bucket mittels Index (= Hash-Code) gefolgt von der relativ langsamen sequentiellen Suche innerhalb des hoffentlich kleinen Bucket. Der Vorteil der hash-basierten Container ist daher der schnelle Zugriff auf den Bucket mittels Index (= Hash-Code). Wenn nun die Berechnung des Hash-Codes ausgesprochen lange dauert, dann ist der Performance-Gewinn durch den schnellen Zugriff per Hash-Code im Handumdrehen zunichte gemacht. Deshalb sollten HashCode-Berechnungen möglichst effizient sein.

Die einfachste und schnellste Lösung wäre es, gar keine Berechnungen anzustellen und für alle Objekte immer denselben Integer-Wert zurück zu liefern. Das ist durchaus erlaubt und führt dazu, dass alle Objekte im selben Bucket landen. Dieser Bucket wird riesengroß sein und die sequentielle Suche darin wird reichlich lange dauern. Unter diesen Randbedingungen macht der hash-basierte Container keinen Sinn mehr; da kann man gleich eine verkettete Liste verwenden.

Die Hash-Code-Berechnung soll also nicht nur schnell sein, sondern auch zu einem Container mit vielen kleinen Buckets führen. Ziel ist eine möglichst performante Implementierung, die eine möglichst gleichmäßige Verteilung der berechneten Hash-Codes im Interval der möglichen Integer-Werte von -2147483648 bis 2147483647 erreicht.

Nun haben wir oben gesagt, dass man wegen der Konsistenz mit equals() alle Felder in die Hash-Code-Berechnung einbeziehen soll, die in equals() miteinander verglichen werden. Das ist auch durchaus praktikabel, solange das Objekt nicht allzu viele Felder hat.

Wenn eine Klasse z. B. aber ein größeres Array von Objekten enthält, dann werden zwar alle Array-Elemente zur Bestimmung der Gleichheit mittels equals() beitragen, aber für eine Implementierung von hashCode() wäre die Berücksichtigung sämtlicher Array-Elemente zu aufwendig. Daher würde man bei einem großen Array vielleicht nur jedes n-te Element in der Hash-Code-Berechnung berücksichtigen.

Analog kann man auch Felder weglassen, die sowieso bei den meisten Objekten den gleichen Werten haben werden. Solche Felder tragen nichts Relevantes zur Produktion unterschiedlicher Hash-Codes bei, erhöhen aber den Aufwand für die Hash-Code-Berechnung, wenn sie berücksichtigt werden.

Man muss auch nicht unbedingt in Klassenhierarchien auf jedem Level jeweils eine neue Version von hashCode() implementieren, um die Subklassen-.spezifischen Felder zu berücksichtigen. Wenn die Superklasse eine korrekte Implementierung von hashCode() zur Verfügung stellt, dann kann man es u.U. dabei belassen.

Bevor man also zur eigentlichen Berechnung des Hash-Codes ansetzt, muss man zunächst (für die Konsistenz mit equals()) die Felder identifizieren, die potentiell in die Berechnung eingehen können; das sind genau die Felder, die in equals() miteinander verglichen werden. Aus diesen Feldern wählt man dann diejenigen aus, die man berücksichtigen oder eben ignorieren will, damit eine Lösung entsteht, die einerseits eine vernünftige Verteilung der Hash-Codes erreicht, aber anderseits auch hinreichend effizient ist.

Anleitung zur Implementierung von hashCode()

Wenn man alle Felder identifiziert hat, die zur Berechnung des Hash-Codes beitragen sollen, wie stellt man dann die Hash-Code-Berechnung an? Die nachfolgend vorgeschlagene Lösung ist keine optimale Implementierung. Zur Berechnung von Hash-Codes gibt es reichlich Information in Fachbüchern. Was wir hier vorstellen wollen, ist ein Rezept für den Hausgebrauch. Wenn man sich daran hält, erzielt man voraussichtlich ein brauchbares, aber nicht notwendig optimales Ergebnis.

Die Idee besteht darin, dass man jedem Feld, dass berücksichtigt werden soll, einen Integer-Wert zuordnet und dann all diese Integer-Werte aufaddiert, wobei man noch einen geeigneten Multiplikator verwendet. Man berechnet also nach folgender Formel:

$$\text{hashcode}_N = \text{hashcode}_{N-1} * \text{multiplikator} + \text{feldwert}_N$$

Anfangswert und Multiplikator

Man beginnt mit einem Anfangswert für den Hash-Code (in der Formel: hashcode0). Der Anfangswert ist typischerweise von 0 verschieden ist. In unserem Beispiel (siehe unten) wählen wir 17 als Anfangswert, das ist aber rein willkürlich. Dazu wählt man sich einen Multiplikator. Der Multiplikator ist typischerweise eine nicht zu große Primzahl, in unserem Beispiel 59. Dann geht die Implementierung von hashCode() wie folgt los:

```
public int hashCode() {
    int hc = 17;
    int hashMultiplier = 59;
    ...
    return hc;
}
```

Feldwert

Jedem für die Berechnung relevanten Feld des Objekts muss ein Integer-Wert zugeordnet werden. Der zugeordnete Wert hängt vom Typ des Objekts ab. Tabelle 1 zeigt eine Übersicht.

Typ	zugeordnete Integer-Werte
`Boolean`	`(field ? 0 : 1)`
`byte, char, short, int`	`(int) field`
`long`	`(int)(field>>>32) und (int)(field & 0xFFFFFFFF)`
`float`	`((x==0.0F)?0:Float.floatToIntBits(field))`
`double`	`((x==0.0)?0L:Double.doubleToLongBits(field)) und anschliessende Behandlung wie bei long`
`Referenz`	`((field==null)?0:field.hashCode())`

Hier ein paar Erläuterungen zur Umrechnung in Integer-Werte.§ Felder vom Typ Boolean werden einfach auf 0 oder 1 umgewertet.

- Kleinere integrale Typen (byte, char, short, int) werden so verwendet, wie sie sind.
- Größere integrale Typen, also long, werden in zwei Integers zerlegt. Man kann sie auch zu einem einzigen Integer vereinen, z.B. durch ein bitweises exclusive-OR: `(int)(field^(field>>>32))`.
- Gleitkomma-Typen werden in ihre jeweiligen Bitlayouts verwandelt. Im Fall von double kommt ein long heraus, den man dann wie für long beschrieben zerlegt. Die Abfrage auf 0.0 ist notwendig, weil 0.0 und -0.0 gleich sind, wenn man sie in equals() per ==-Operator vergleicht. Die korrespondierenden Bit-Layouts von 0.0 und -0.0 sind aber verschieden. Deshalb wird der Fall gesondert behandelt.
- Für Felder, die Referenzen auf Objekte sind, ruft man die hashCode()-Methode des referenzierten Objekts. Die Abfrage auf null ist notwendig, damit hashCode() keine NullPointerException wirft.

Sonderfälle

Bei der Implementierung von equals() haben wir Klassen mit inkorrekter Implementierung von equals() gesondert behandeln müssen. Unser Beispiel war die Klasse StringBuffer, die die

equals()-Methode nicht überschreibt. Immer wenn Felder vom Typ StringBuffer miteinander verglichen werden müssen, haben wir die StringBuffer in String-Objekte umgewandelt und dann die String-Objekte per String.equals() miteinander verglichen.
Für die Implementierung von hashCode() müssen wir analog vorgehen, damit die Konsistenz zwischen hashCode() und equals() gewährleistet ist. Einem StringBuffer-Feld wird daher nicht field.hashCode() zugeordnet, sondern field.toString().hashCode().
Derartige Sonderbehandlungen für all diejenige Felder notwendig, wo sie auch in equals() nötig waren.

Arrays

Arrays werden gesondert behandelt. Bei größeren Arrays wird man wahrscheinlich nur eine Auswahl der Array-Elemente berücksichtigen wollen. Die einzelnen Array-Elemente behandelt man dann ihrem jeweiligen Type entsprechend wie oben beschrieben. Hier ist ein Beispiel, in dem nur die Array-Elemente berücksichtigt werden, deren Index eine Zweierpotenz ist:

```
class MyClass {
  private Object arrayField[];

  public int hashCode() {
    int hc = 17;
    int hashMultiplier = 59;
    ...
    hc = hc * hashMultiplier + arrayField.length();
    for (int i=0; i<arrayField.length(); i<<=1) {
       hc = hc * hashMultiplier + arrayField.hashCode();
    }
    ...
    return hc;
  }
}
```

Superklassenanteile

Die Berücksichtigung von geerbten Feldern delegiert man an die Superklasse. Das heißt, man ruft super.hashCode().

```
public int hashCode() {
    ...
    hc = hc * hashMultiplier + super.hashCode();
    ...
    return hc;
}
```

Das darf man allerdings nur tun, wenn die Superklasse nicht Object ist, weil ja sonst die Adresse des Objekts berücksichtigt würde, und das führt dann zu den schon geschilderten Problemen. So ähnlich wie bei equals(), wird man zwischen direkten und indirekten Subklassen von Object unterscheiden. In den indirekten Subklassen wird man super.hashCode()rufen, während man es in den direkten Subklassen nicht tun wird.

hashCode() in Klassenhierarchien

Im letzten Artikel haben wir ausführlich diskutiert, ob man in Klassenhierarchien den Vergleich zwischen Super- und Subklassen-Objekten zulassen sollte. Das Ergebnis war im wesentlichen die Erkenntnis, dass man ihn i.a. nicht zulassen wird; dann ist man auf der sicheren Seite. Und wenn man den ihn doch zulassen will, dann sollte er als final-Methode in der Superklasse definiert sein.

Die Entscheidung, die man diesbezüglich für equals() getroffen hat, hat Auswirkungen auf die Implementierung von hashCode(). Wenn die equals()-Methode final ist, dann sollte auch die hashCode()-Methode final sein. Denn sonst könnte hashCode()überschrieben werden. Wenn die überschreibende Version von hashCode() Subklassen-spezifische Felder berücksichtigt, dann ist die Konsistenz zu equals() verletzt: zwei Subklassenobjekte könnten dann gleich sein, weil nur Ihre Superklassenanteile miteinander verglichen werden, aber sie hätten verschiedene Hash-Codes, da Subklassen-spezifische Information in die Hash-Code-Berechnung eingeht

Ähnliche Gefahren lauern, wenn der Super-Subklassen-Vergleich erlaubt ist, ohne dass equals() als final deklariert ist. (Davon haben wir abgeraten, aber man findet es in der Praxis.) Dann sollte hashCode() nur in der Superklasse (und dort am besten als final) definiert sein und in der Subklasse auf keinen Fall überschrieben werden. Wenn man hashCode() in der Subklasse überschreibt, dann ist wieder die Konsistenz zwischen equals() und hashCode()verloren: zwei Objekte, ein Super- und ein Subklassen-Objekt, könnten dann gleich sein, weil nur ihr Superklassenanteil verglichen wird. Wenn hashCode()aber in Super- und Subklasse in verschiedenen Versionen existiert, dann werden die Hash-Codes dieser beiden gleichen Objekte nicht unbedingt gleich sein.

Zusammenfassung

Wir haben uns in dieser Ausgabe mit der Methode hashCode() befasst, die man auf allen Objekte in Java aufrufen kann. Wir haben gesehen, wann man die Default-Implementierung von hashCode() überschreiben muss, nämlich immer dann wenn man dasselbe für equals() tut. Wir haben den sogenannten hashCode-Contract angesehen, der die Anforderungen an eine Implementierung von hashCode() spezifiziert. Wir haben die Konsequenzen bei Verletzung des hashCode-Contracts gesehen; die hash-basierten Container funktionieren dann nicht. Und schließlich haben wir die Prinzipien einer Implementierung betrachtet; wichtig für eine korrekte Implementierung ist die Konsistenz zu equals() und die Performanz der Hash-Code-Berechnung sowie die Güte der Hash-Code-Verteilung.

Wegen der engen Beziehung zwischen equals() und hashCode() an dieser Stelle nochmals die Empfehlung, equals() sorgfältig und wohl überlegt zu implementieren. equals() hat Auswirkungen auf andere Versionen von equals() in derselben Klassenhierarchie und auf die Hash-Code-Berechnung. Und das ist noch nicht alles; in der nächsten Ausgabe dieser Kolumne werden wir unsere Betrachtungen über die Objekt-Infrastruktur fortsetzen und uns die compareTo()-Methode ansehen, die für die Benutzung der baum-basierten Container von

Bedeutung ist. Auch für compareTo() gibt es eine Konsistenzanforderung in Bezug auf equals(). Dazu mehr beim nächsten Mal.

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.Equals1.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

/KRE3/ **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

/KRE4/ **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: Hashing and Cloning**
Mark Davis
Java Report, April 2000
URL: http://www.macchiato.com/columns/Durable6.html

/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