Unveränderliche Typen in Java

Teil 1: Wie implementiert man unveränderliche Typen, lesende Zugriffsmethoden, Immutability-Adaptoren und "duale Klassen"?

JavaSPEKTRUM, März 2003
Klaus Kreft & Angelika Langer

Weil die Unveränderlichkeit (engl. immutability) von Typen ein wichtiges Thema in Java ist, widmen wir ihm diesen und den nächsten Artikel. Wir diskutieren die Implementierung von lesenden Methoden und die Eigenschaften von unveränderlichen Typen. In diesem Zusammenhang besprechen wir Read-Only-Adaptoren und sogenannte "duale Klassen". Im nächsten Artikel sehen wir uns die Immutability-Adaptoren in Collection-Framework des JDK an und diskutieren den Sinn und Zweck des Schlüsselworts final.

Wofür werden unveränderlichen Typen gebraucht?

Immutability spielt immer dann eine Rolle, wenn Objekte gemeinsam verwendet werden (engl. object sharing). Diese Situation entsteht in Java, wenn Referenzen einander zugewiesen oder von und an Methoden übergeben werden. Es verweisen dann mehrere Referenzen auf ein Objekt und die Besitzer dieser Referenzen können abwechselnd lesend und/oder schreibend auf das gemeinsam verwendete Objekt zugreifen.

Manchmal ist es unerwünscht, dass das gemeinsam verwendete Objekt von allen Beteiligten nach Belieben verändert werden kann. Dann kann man auf das Sharing verzichtet und Kopien anlegen, so dass jede Referenz auf ihre eigene Kopie des Objekts verweist.(Das Kopieren von Objekten haben wir ausführlich in /KRE3/ besprochen.) Es geht aber unter Umständen auch ohne das Kopieren. Den Aufwand für das Kopieren kann man vermeiden, wenn das gemeinsam verwendete Objekt gar nicht verändert werden kann. Und hier kommen unveränderliche Typen ins Spiel; sie helfen, den Performance-Overhead für das Kopieren von Objekten zu eliminieren.

Es gibt eine weitere Situation, in der unveränderliche Typen nützlich sind. In Programmen mit mehreren parallelen Threads können Objekte gemeinsam von mehreren Threads verwendet werden. Dann gibt es ebenfalls mehrere Referenzen auf ein gemeinsam verwendetes Objekt. Die Referenzen werden in verschiedenen parallel ablaufenden Threads gehalten und in solchen Fällen ist das Object Sharing ausdrücklich erwünscht: es soll eine Kommunikation zwischen den Threads über das gemeinsam verwendete Objekt stattfinden.

Wenn auf das gemeinsam verwendete Objekt schreibend zugegriffen werden kann, dann müssen die Zugriffe auf das gemeinsam verwendete Objekt synchronisiert werden, damit sie nacheinander und nicht ineinander verschränkt ablaufen. Sonst könnte es beispielsweise passieren, dass ein lesender Thread ein halb geschriebenes Objekt zu sehen bekommt, weil der schreibende Thread noch gar nicht fertig war, als er unterbrochen wurde. Hier helfen unveränderlichen Typen, den Synchronisationsaufwand zu vermeiden. Wenn man weiß, dass das Objekt unveränderlich ist, dann kann man auf die Synchronisation gänzlich verzichten.

Im Folgenden werden wir den Fall von Multithread-Anwendungen nicht weiter vertiefen, sondern wir werden den Nutzen von unveränderlichen Typen am Beispiel des Object Sharing in Single-Thread-Programmen betrachten. Alle Techniken und Vor- und Nachteile, die wir diskutieren werden, gelten aber für die Nutzung von veränderlichen und unveränderlichen Typen in Multithread-Umgebungen ganz genauso.
Unser Ziel in diesem Artikel ist es, unveränderliche Typen zu implementieren. Eine Eigenschaft von unveränderlichen Typen ist, dass alle ihre Methoden read-only-Funktionalität haben, d.h. sie verändern das this-Objekt nicht, sondern "lesen" es nur. Fangen wir also damit an, dass wir uns überlegen, wie man eigentlich eine lesende Methode implementiert.

Lesende Zugriffsmethoden

Nehmen wir einmal an, wir wollen eine Zugriffsmethode implementieren, die nur lesenden Zugriff auf private Daten eines Objekts gibt. Betrachten wir dazu das folgende Beispiel:

```
public final class Widget
{
  private Stamp lastModification = null;

  ... other methods and fields ...

  public Stamp getLastModification();

}
```

Es handelt sich um eine Klasse, die in einem Feld vom Typ Stamp Informationen über die letzte erfolgte Modifikation des Objekts festhält. Der Typ Stamp ist hier nicht näher ausgeführt, aber man stelle sich vor, er enthält Informationen wie den Zeitpunkt der Veränderung, den Urheber der Veränderung, etc. Das Feld wird an geeigneter Stelle initialisiert und in jeder verändernden Methode mit neuen Werten belegt. Wie das genau gemacht wird, ist an dieser Stelle ohne Belang. Uns interessiert vielmehr die Zugriffsmethode getLastModification(). Sie gibt Zugriff auf das private Feld. Dabei soll sie sicher keinen Schreibzugriff gestatten, sondern nur Lesezugriff. Sonst könnte "von Außen" der Zeitstempel manipuliert werden; er soll aber ausschließlich "von Innen", d.. von den Methoden der Klasse Widget, verändert werden. Wie kann man diese nur lesende Zugriffsmethode implementieren?

Also, so ist es sicher falsch:

```
public final class Widget
{
  private Stamp lastModification;

  ... other methods and fields ...

  public Stamp getLastModification()
  {
    return lastModification ;   // don't do this !!!
  }
}
```

Hier kann jeder nach dem Aufruf von getLastModification() auf das private Feld der Klasse Widget zugreifen und den Eintrag ändern. Zum Beispiel so:

```
Widget w = new Widget();
...
Stamp log = w.getLastModification();
log.setDate(new Date());
log.setAuthor("Molly Malicious");
```

Mit der Veränderung des Felds wäre dann die logische Konsistenz des Widget Objekts zerstört. Das sollte eigentlich nicht passieren. Das Problem rührt daher, dass Stamp ein veränderlicher Typ ist. Hat man erst einmal Referenz auf ein Objekt des Typs Stamp, dann hat man nicht nur Lese- sondern auch Schreibzugriff auf das Objekt. Man kann das Problem lösen, indem man das Feld kopiert, damit dem Aufrufer von getLastModification() eine eigene unabhängige Kopie zur Verfügung steht, die er nach Belieben ändern kann, ohne dass es Auswirkungen auf die Klasse Widget hat.

```
public final class Widget
{
  private Stamp lastModification;

  ... other methods and fields ...

  public Stamp getLastModification()
  {
    return (Stamp)lastModification.clone();
  }
}
```

Das Kopieren von Objekten kann u.U. relativ teuer sein, abhängig von der inneren Struktur und Größe des zu kopierenden Objekts. Generell wird man versuchen, den Overhead des Kopierens zu vermeiden, wann immer es geht, schon allein, weil es das neu erzeugte Objekt den Garbage Collector mit zusätzlicher Arbeit belastet. Und in diesem Fall geht es. Man kann ohne Kopie auskommen, nämlich mit Hilfe von unveränderlichen Typen.

Unveränderliche Typen

Referenzvariablen von einem unveränderlichen Typ zeigen entweder auf Objekte, die sich tatsächlich nicht verändern, oder sie lassen veränderliche Objekte zumindest so aussehen, als seien sie unveränderlich. Man unterscheidet zwischen Read-Only-Sichten und "echten" unveränderlichen Typen. Im Folgenden sehen wir uns Adaptoren an, die diese beiden Arten von unveränderlichen Typen erzeugen.

Read-Only-Adaptoren

Sehen wir uns einen Read-Only-Adaptor am Beispiel unserer Stamp Klasse an, die vermutlich die folgenden Methoden haben wird:

```
public final class Stamp
{
```

```
  private Date date;
  private String author;

  ... other methods and fields ...

  public Date getDate()
  {
    return (Date)date.clone();
  }
  public void setDate(Date d)
  {
    date = (Date)d.clone();
  }

  public String getAuthor()
  {
    return author;
  }
  public void setAuthor(String a)
  {
    author = a;
  }
}
```

Wenn wir ein Interface definieren, das nur die lesenden Methoden der Klasse Stamp enthält, dann haben wir eine Read-Only-Sicht auf Objekte des Typs Stamp:

```
public interface ImmutableStamp
{
  public Date getDate();
  public String getAuthor();
}
```

Die Klasse Stamp kann nun dieses Interface implementieren:

```
public final class Stamp implements ImmutableStamp
{
  private Date date;
  private String author;

  ... other methods and fields ...

  public Date getDate();
  {
    return (Date)date.clone();
  }
  public void setDate(Date d)
  {
    date = (Date)d.clone();
  }
  public String getAuthor()
  {
    return author;
  }
```

```
  public void setAuthor(String a)
  {
    author = a;
  }
}
```

Mit dem Read-Only-Adaptor wollten wir erreichen, dass wir in der lesenden Zugriffsmethode getLastModification() möglichst ohne Kopieren auskommen. Nun schaffen wir es nicht, das Kopieren von Objekten gänzlich zu vermeiden, weil ja bereits in den Methoden der Stamp-Klasse Kopien erzeugt werden, aber wir können es doch deutlich reduzieren. Man könnte nämlich nun die Methode getLastModification() so ändern, dass sie anstelle einer Referenz auf ein Stamp-Objekt eine Referenz vom Typ ImmutableStamp zurück gibt. Das sähe dann so aus:

```
public final class Widget
{
  private Stamp lastModification;

  ... other methods and fields ...

  public ImmutableStamp getLastModification()
  {
    return lastModification;
  }
}
```

Durch die Rückgabe einer Referenz vom Typ ImmutableStamp auf das existierende Stamp-Feld des Widget-Objekts haben wir das Kopieren des Stamp-Objekts vermieden.Gleichzeitig haben wir aber mit Hilfe des Read-Only-Interfaces aber auch erreicht, dass der Aufrufer nur noch lesend auf das Stamp-Feld zugreifen kann. Das sieht man im nachfolgenden Beispiel:

```
Widget w = new Widget();
...

ImutableStamp log = w.getLastModification();
log.setDate(new Date());                  // does not compile
log.setAuthor("Molly Malicious");         // does not compile
```

Bereits zur Compilezeit bekommt man hier eine Fehlermeldung, weil über das Interface ImmutableStamp nur noch die lesenden Methoden getDate() und getAuthor() sichtbar sind.

Read-Only-Adaptoren haben einen gravierenden Haken. Das Stamp-Objekt sieht, durch die Brille des ImmutableStamp-Interfaces gesehen, nur so aus, als sei es unveränderlich. In Wahrheit kann das Stamp-Objekt natürlich immer noch geändert werden. Wir haben lediglich eine Art Absichtserklärung erreicht: die getLastModification()-Methode gibt zu erkennen, dass sie keinen schreibenden Zugriff auf das Stamp-Objekt geben möchte. Und in der Tat kann man auch versehentlich über das ImmutableStamp-Interface keine Veränderungen vornehmen. Aber das Interface gibt keine Garantie, dass das Stamp-Objekt tatsächlich unverändert bleibt. Es könnte ja an anderer Stelle über eine Stamp-Referenz verändert werden. Und natürlich kann man die ImmutableStamp-Referenz mit einem expliziten Cast in

eine Stamp-Referenz verwandeln, und dann kann man sogar selber verändernd auf das referenzierte Stamp-Objekt zugreifen. Ein Read-Only-Adapter gibt also keine Garantie, dass das referenierte Objekt unverändert bleibt.

Ob eine Read-Only-Sicht auf ein veränderliches Objekt nun gut oder schlecht ist, hängt ganz von den Umständen und der Erwartungshaltung ab. Es kann durchaus erwünscht sein, dass man selbst keine Veränderungen am Shared Object vornehmen will (und dies durch die Read-Only-Sicht zum Ausdruck bringt), man aber die Veränderungen am Shared Object, die von anderen herbeigeführt werden, sehen möchte. Dann ist eine Read-Only-Sicht auf ein veränderliches Objekt völlig in Ordnung. Es kann aber auch sein, dass man sich auf die Unveränderlichkeit des Objekts verlassen will. Das ist zum Beispiel bei Shared Objects in Multithread-Programmen der Fall. Die Synchronisation der Zugriffe auf das von mehreren Threads gemeinsam verwendete Objekt kann nur dann entfallen, wenn das Objekt sich tatsächlich nicht ändern kann. Bei Shared Objects in Multithread-Programmen wäre es ein fataler Fehler, wegen der Read-Only-Sicht auf die Synchronisation zu verzichten, weil das referenzierte Shared Objekt durch den Read-Only-Adapter keineswegs for Veränderungen geschützt ist.

Im Beipiel unserer getLastModification()-Methode kann man darüber streiten, ob die Read-Only-Sicht auf das veränderliche Stamp-Objekt gut oder schlecht ist. In jedem Falle sollte aber sorgfältig dokumentiert sein, was genau die Methode zurück gibt.

Ganz allgemein muss man sich darüber klar sein, was ein Read-Only-Interface tatsächlich bedeutet: es ist reine Read-Only-Sicht auf etwas möglicherweise Veränderliches. Die Gefahr liegt darin, dass u.U. nicht jedem auf Anhieb klar ist, dass etwas, das unveränderlich aussieht, dennoch verändert werden kann. Ein Read-Only-Interface könnte zu Missverständnissen führen. Deshalb ist es nicht ganz unproblematisch. Das gleiche gilt übrigens auch für Superklassen, die Immutability versprechen, dann aber auf Subklassen verweisen können, die gar nicht unveränderlich sind.

Duale Klassen

In folgenden wollen wir eine Lösung vorstellen, die beide Aspekte von Immutability abdeckt: die Read-Only-Sicht auf etwas möglicherweise Veränderliches und die Referenz auf ein echt inveränderliches Objekt. Das kann man mit sogenannten dualen Klassen erreichen. Bei diesem Ansatz ist die Grundidee, dass man zwei verschiedene Klassen, eine für veränderliche und eine für unveränderliche Objekte, hat. Im Beispiel unserer Stamp-Abstraktion würde es neben der Stamp -Klasse noch eine zweite Klasse ImmutableStamp geben, die wie folgt aussähe:

```
public final class ImmutableStamp
{
  private Stamp stamp;

  public ImmutableStamp(Stamp s) { stamp = (Stamp)s.clone(); }

  public Date getDate()
  {
    return stamp.getDate();
  }
```

```
  public String getAuthor()
  {
    return stamp.getAuthor();
  }
}
```

Objekte vom Typ ImmutableStamp sind im Prinzip Kopien der korrespondierenden Stamp-Objekte. Sie haben, genau wie unser Read-Only-Interface zuvor, nur die lesenden Methoden getDate() und getAuthor(). Die Zugriffsmethode getLastModification() unserer Widget-Klasse würde dann wie folgt aussehen:

```
public final class Widget
{
  private Stamp lastModification;

  ... other methods and fields ...

  public ImmutableStamp getLastModification()
  {
    return new ImmutableStamp(lastModification);
  }
}
```

Mit dieser Lösung hat man ebenfalls erreicht, dass die Methode getLastModification() nur noch lesenden Zugriff auf den Zeitstempel gibt. Dieses Mal verweist die zurückgelieferte Referenz auf ein Objekt, das wirklich unveränderlich ist, weil es ein ImmutableStamp-Objekt ist, das überhaupt keine verändernden Methoden hat. Anders als in der zuvor besprochenen Adapter-Lösung mit dem ImmutableStamp-Interface, wo wir eine Referenz auf das Original-Stamp-Objekt zurückgeliefert hatten, welches nach wie .vor veränderlich ist.

Nun kann man sich fragen, was man mit dieser Lösung an Kopieraufwand gespart hat. Zunächst einmal nichts. In unserer Original-Implementierung der Widget-Klasse hatte die Methode getLastModification() einen Klon erzeut. Jetzt passiert genau dasselbe, allerdings implizit im Konstruktor des unveränderlichen Typs. Von nun an spart man aber Kopieraufwände ein, weil man mit dem ImmutableStamp-Objekt ein unverändliches Objekt hat, das man nie mehr kopieren muss und das man immer per Referenz weiterreichen kann. Wenn die ImmutableStamp-Klasse nicht existiert, dann gibt es keine Möglichkeit sicher zu stellen, dass in einem bestimmten Kontext Schreibzugriffe ausgeschlossen sind. Im Zweifelsfall muss man dann Kopien von Stamp-Objekten erzeugen, um sich gegen Veränderungen an gemeinsam verwendeten Stamp-Objekten zu schützen, so wie wir das in unserer allerersten Lösung gemacht hatten.

Klassen, die wie unser Stamp/ImmutableStamp-Paar in zwei Ausprägungen daher kommen, bezeichnet man als duale Klassen. Das wohl bekannteste Beipiel für ein solches Paar ist die String-Abstraktion im JDK, die in Form der beiden Klassen String und StringBuffer implementiert ist.

Duale Klassen im Detail

Mit der dualen Klasse haben wir eine Möglichkeit gefunden, sowohl veränderliche als auch unveränderliche Ausprägungen einer Abstraktion zu verwenden. Fehlt uns also noch die Read-Only-Sicht für all die Situationen, in denen wir durch die Read-Only-Brille auf ein veränderliches Objekt blicken wollen.

Die Read-Only-Sicht auf eine duale Klasse drückt man aus durch eine gemeinsame Superklasse oder ein gemeinsames Super-Interface, das es erlaubt, die beiden Typen von Objekten austauschbar zu verwenden. Das ist nützlich, wenn man Schnittstellen hat, denen es egal ist, ob die Objekte veränderlich oder unveränderlich sind. Diese Schnittstellen würden dann so deklariert, dass sie mit Superklassen- oder Super-Interface-Referenzen arbeiten, hinter denen sich beide Ausprägungen der Abstraktion verbergen können. Diese gemeinsame Superklasse oder das gemeinsame Super-Interface ist dann die Read-Only-Sicht auf beiden Arten von Objekten.

Im Beispiel unserer Stamp-Abstraktion sähe das so aus, wenn man ein gemeinsames Interface definiert:

```
public interface StampBase
{
 public Date getDate();
 public String getAuthor();
}


public final class Stamp implements StampBase
{
  private Date date;
  private String author;

  ... other fields and methods ...

  public Stamp(Date d, String a)
  {
    date = (Date)d.clone():
    author = a;
  }
  public Date getDate()
  {
    return (Date)date.clone();
  }
  public void setDate(Date d)
  {
    date = (Date)d.clone();
  }

  public String getAuthor()
  {
    return author;
  }
  public void setAuthor(String a)
```

```
  {
    author = a;
  }
}

public final class ImmutableStamp implements StampBase
{
  private Stamp stamp;

  public ImmutableStamp(Stamp s) { stamp = (Stamp)s.clone(); }

  public Date getDate()
  {
    return stamp.getDate();
  }
  public String getAuthor()
  {
    return stamp.getAuthor();
  }
}
```

Man kann auch noch einen Schritt weiter gehen und Gemeinsamkeiten der beiden Stamp-Klassen in eine gemeinsame Superklasse herausziehen. Das sieht dann wie folgt aus:

```
public class StampBase
{
 protected Date date;
 protected String author;

 ... other common fields ...

 public StampBase(Date d, String a)
 {
   date = (Date)d.clone():
   author = a;
 }

 public Date getDate()
  {
    return (Date)date.clone();
  }
  public String getAuthor()
  {
    return author;
  }

  ... other common read-only methods ...
}


public final class Stamp extends StampBase
{
  public Stamp(Date d, String a)
  {
```

```
    super(d,a);
  }
  public Stamp(StampBase s)
  {
    super(s.date,s.author);
  }

  public void setDate(Date d)
  {
    date = (Date)d.clone();
  }

  public void setAuthor(String a)
  {
    author = a;
  }

  ... other mutating methods ...
}

public final class ImmutableStamp extends StampBase
{
  public ImmutableStamp(Date d, String a)
  {
    super(d,a);
  }
  public ImmutableStamp(StampBase s)
  {
    super(s.date,s.author);
  }
}
```

In beiden Fällen sollte man aber nicht etwa auf die Idee kommen, Stamp von ImmutableStamp abzuleiten. Diese Idee ist ziemlich naheliegend; schließlich würde man dann die lesenden Methoden erben und müßte sie nicht re-implementieren. Wenn man diese Ableitung macht, dann ist die Semantik der ImmutableStamp-Klasse radikal anders: sie gibt keine Immutability-Garantie mehr. Die ImmutableStamp-Klasse degeneriert dann zu einer Read-Only-Sicht. Das liegt daran, dass eine Referenzvariable vom Typ ImmutableStamp wegen der Vererbungsbeziehung auf ein veränderliches Stamp-Objekt verweisen kann. Das ist nicht die Idee einer dualen Klasse.

Bei der dualen Klasse hat man getrennte Typen für die veränderliche und die unveränderliche Ausprägung der Abstraktion. Die beiden Typen sind nicht zuweisungsverträglich, d.h. nicht voneinander abgeleitet. Variablen des einen Typs können nicht in Variablen des anderen Type gecastet werden. Allerdings sind "Konvertierungen" möglich, indem aus dem Objekt des einen Typs ein Objekt des anderen Type konstruktiert wird. Das sind aber keine Typkonvertierungen, sondern Objektkonvertierungen, die Kopieraufwände beinhalten.

Typkonvertierungen sind möglich zwischen dem dritten Supertyp (falls vorhanden) und den beiden Sybtypen. Das heißt, man kann eine Read-Only-Sicht (durch den gemeinsame Supertyp) auf beide Arten von Objekten haben, und man kann zwischen der Read-Only-Sicht

und der uneingeschränkten "echten" Sicht hin und her konvertieren. Bei dieser Art der Konvertierung sind werden keine Kopien gemacht, sondern nur Sichten verändert. Zwischen den beiden Subtypen, dem veränderlichen und unveränderlichen Typ, kann jedoch nicht konvertiert werden.

Mit dualen Klassen ist man nun sehr flexibel:

- Man hat die Möglichkeit, Methoden zu implementieren, die mit StampBase-Referenzen arbeiten. Solche Methoden greifen nur lesend auf die Stamp-Objekte zu, interessieren sich aber desweiteren nicht dafür, ob die Stamp-Objekte veränderlich oder unveränderlich sind. Man kann auch jederzeit einen Cast auf den veränderlichen oder unveränderlichen Subtyp machen, wenn das gebraucht wird.
- Wo man sicherstellen muss, dass sich Stamp-Objekte nicht ändern, kann man ImmutableStamp-Objekte verwenden. Über den Konstruktor ImmutableStamp(StampBase s) kann man jederzeit aus veränderlichen Stamp-Objekten unveränderliche ImmutableStamp-Objekte erzeugen.
- Und wo die gesamte Funktionalität der Stamp-Abstraktion inklusive der verändernden Methoden gebraucht wird, da kann man mit Stamp-Objekten arbeiten, die man ebenfalls per Konstruktion aus ImmutableStamp-Objekten erzeugen kann.

Marker-Interface für Immutability

Leider bietet die Sprache Java praktisch keine Unterstützung für die Immutability an. Es gibt ein winziges bischen an Unterstützung in Form des Schlüsselwortes final; darauf kommen wir in der nächsten Ausgabe der Kolumne zurück. Ansonsten ist man auf Konventionen und Programmierdisziplin angewiesen. Die Tatsache, dass eine Klasse unveränderlich ist, kann man nur im Namen der Klasse und/oder in der Dokumentation zur Klasse ausdrücken.

Wer es etwas deutlicher sagen will, kann sich ein leeres Marker-Interface Immutable definieren und alle unveränderlichen Klassen davon ableiten. Das hat den Vorteil, dass man zur Laufzeit ein Objekt mit Hilfe des instanceof-Operators fragen kann, ob es unveränderlich ist.

Aber auch das bietet keine absolute Sicherheit, insbesondere dann nicht, wenn Vererbung im Spiel ist. Wir haben in unserem Beispiel einer dualen Abstraktion bewußt die veränderliche und die unveränderliche Klasse als final Klassen deklariert. Bei einer non-final Klasse, die unveränderlich ist, ist es reine Disziplin und guter Wille, dass diese Semantik in den Subklassen auch beibehalten wird.

Zusammenfassung und Ausblick

In diesem Artikel haben wir uns angesehen, wie man unveränderliche Typen implementiert. Unveränderliche Typen sind nützlich, weil sie die Notwendigkeit, Kopien von Objekten zu erzeugen, reduzieren und weil sie den Synchronisationsaufwand in Multithread-Umgebungen vermindern.

Unveränderliche Typen haben nur lesende Methoden und sind idealerweise final Klassen oder haben nur Subklassen, die ebenfalls unveränderliche Typen sind. Unveränderliche Typen sollte man nicht mit Read-Only-Sichten auf veränderliche Typen verwechseln.
Als duale Klasse bezeichnet man Abstraktionen, die als Paar von einer veränderlichen und einer unveränderlichen Klasse implementiert sind. Duale Klassen haben häufig einen gemeinsamen Supertyp, der die Read-Only-Sicht auf beide Klassen repräsentiert.
In der nächsten Ausgabe dieser Kolumne sehen wir uns das Schlüsselwort final an und was es mit Immutability zu tun hat. Darüber hinaus untersuchen wir die Immuability-Adaptoren des JDK-Collection-Frameworks.

Unveränderliche Typen in Java

Teil 2: Wie die Immutability-Adaptoren im JDK-Collection-Framework funktionieren und was das Schlüsselwort final mit Immutability zu tun hat.

JavaSPEKTRUM, Juli 2003
Klaus Kreft & Angelika Langer

Im letzen Artikel dieser Kolumne (siehe /KRE1/) haben wir diskutiert, wie man unveränderliche Typen, duale Klassen und Read-Only-Adaptoren implementieren kann. Das wollen wir in diesem Artikel weiter verfolgen. Es gibt Beispiele für Read-Only-Adaptoren im JDK, nämlich im Collection-Framework. Diese "unmodifiable" Collections sehen wir uns in diesem Artikel näher an. Außerdem erklären wir, was das Schlüsselwort final mit Immutability zu tun hat. Und am Ende fassen wir zusammen, welche Arten von Immutability es gibt, was sie bedeuten und wie man damit umgeht.

Immutability-Adaptoren für Collections im JDK

Im Package java.util (siehe / JDK /) ist u.a. der Collection Framework definiert mit verschiedenen Container-Interfaces wie Set, List, und Map sowie verschiedenen Container-Implementierungen wie ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, usw. Die Collections sind erwartungsgemäß veränderliche Typen; sie haben z.B. Methoden wie add() und remove() für das Hinzufügen und Löschen von Elementen.

Zu diesen Collection-Typen sind Immutability-Adaptoren definiert, die man sich über statische Methoden der Klasse Collections beschaffen kann: man übergibt eine Referenz auf eine veränderliche Collection als Argument und erhält als Return eine Referenz auf eine unveränderliche Collection zurück. Genauer gesagt bekommt man keine Referenz auf eine unveränderliche Collection, sondern eine Referenz auf ein Objekt, das eine eingeschränkte Sicht auf die veränderliche Collection gibt, nämlich eine nur lesende Sicht. Wir werden im folgenden aber dennoch den Begriff "unveränderliche Collection" anstelle von "Lese-Sicht auf veränderliche Collection" verwenden, weil es einfach kürzer ist, und außerdem ist es die

direkte Übersetzung des entsprechenden Begriffs "unmodifiable collection", wie er in der JDK-API-Dokumentation verwendet wird.

Nehmen wir das Beipiel einer ArrayList. Zu einem ArrayList-Objekt kann man sich über den Aufruf von Collections.unmodfiableList() die Lese-Sicht auf das ArrayList-Objekt geben lassen.

List myList;
myList = new ArrayList();
myList = Collections.unmodifiableList(myList);

Von welchem Typ die Lese-Sicht auf das ArrayList-Objekt ist, bleibt unbekannt, weil die Methode Collections.unmodfiableList() lediglich eine Referenz vom Typ List zurückgibt. List ist ein Interface, das sowohl von ArrayList als auch von LinkedList und Vector implementiert wird. Das bedeutet, dass der "unveränderliche" Container die gesamte List-Funktionalität anbietet, was einigermaßen überraschend ist. Man würde vielmehr erwarten, dass modifizierende Methoden wie add() und remove() gar nicht erst zur Verfügung stehen. Die Designer des JDK-Collection-Frameworks haben aber eine andere Lösung gewählt: die verändernden Methoden stehen zur Verfügung und können auch aufgerufen werden, scheitern aber zur Laufzeit mit einer UnsupportedOperationException. Hier wird der Compilezeit-Check durch einen Laufzeit-Check ersetzt.

Immutability-Adaptoren mit Laufzeit-Check

Übertragen wir diese im JDK verwendete Technik einmal auf die Implementierung, die wir in der letzten Ausgabe dieser Kolumne (siehe /KRE1/) diskutiert haben. Wir hatten einen unveränderlichen Typ ImmutableStamp implementiert, der in einer Widget-Klasse verwendet wurde, um eine lesende Zugriffsmethode getLastModification() zu implementieren. Hier noch einmal zur Erinnerung die Widget-Klasse sowie die Stamp- und ImmutableStamp-Klasse:

```
public final class Stamp
{
  private Date date;
  private String author;

  ... other fields, constructors and methods ...

  public Date getDate()
  {
    return Date.clone();
  }
  public void setDate(Date d)
  {
    date = d.clone();
  }
  public String getAuthor()
  {
    return author;
  }
  public void setAuthor(String a)
  {
    author = a;
```

```
  }
}

public final class ImmutableStamp
{
  private Stamp stamp;

  public ImmutableStamp(Stamp s) { stamp = s.clone(); }

  public Date getDate()
  {
    return stamp.getDate();
  }
  public String getAuthor()
  {
    return stamp.getAuthor();
  }
}

public final class Widget
{
  private Stamp lastModification;

  ... other methods and fields ...

  public ImmutableStamp getLastModification()
  {
    return new ImmutableStamp(lastModification);
  }
}
```

Die Stamp-Abstraktion hatten wir als duale Klasse implementiert, die aus einer veränderlichen Klasse Stamp und einer unveränderlichen Klasse ImmutableStamp besteht. Man beachte, dass die ImmutableStamp-Klasse in unserer Implementierung ausschließlich lesende Methoden hat.

Wenn wir die JDK-Technik, die für die "unmodifiable collections" verwendet wird, auf unsere ImmutableStamp-Klasse übertragen, dann sieht die Klasse anders aus, nämlich so:

```
public final class ImmutableStamp
{
  private Stamp stamp;

  public ImmutableStamp(Stamp s) { stamp = s; }

  public Date getDate()
  {
    return stamp.getDate().clone();
  }
  public void setDate(Date d)
  {
    throw new UnsupportedOperationException();
  }
  public String getAuthor()
```

```
  {
    return stamp.getAuthor();
  }
  public void setAuthor(String s)
  {
    throw new UnsupportedOperationException();
  }
}
```

Ob man dieses Vorgehen für sinnvoll hält oder nicht, muss jeder für sich selbst entscheiden. Wir halten Immutability-Adaptoren, bei denen die modifizierenden Methoden zwar angeboten werden, aber immer und unbedingt eine Exception werfen, für wenig nachahmenswert und würden die verändernden Methoden lieber gleich weglassen.

Dass die JDK-Technik mit dem Laufzeit-Check nicht ganz so ideal ist, zeigt sich auch in der Dokumentation der Immutability-Adaptoren des JDK-Collection-Frameworks (siehe / JDK /). Wenn man die JavaDoc liest, ist eigentlich nicht auf Anhieb klar, welche Methoden ein "unmodifiable" Container nun unterstützt und welche nicht. Wenn man dagegen ein Interface hätte, in dem nur die lesenden Methoden vorhanden wären, dann könnte man sofort erkennen, welche Methoden von der "unmodifiable" Collection unterstützt werden und welche nicht. Und natürlich würde auch der Compiler bereits beim Übersetzen Fehler melden, wenn man eine Methode aufrufen will, die nicht erlaubt ist.

Die JDK-Designer haben aber für die Entscheidung gegen den Compiletime-Check einen guten Grund gehabt. Als Begründung findet man die Aussage, dass sie andernfalls unzählige Interfaces und Klasses hätten definieren müssen, was nun auch nicht gerade zur Klarheit beigetragen hätte (siehe / FAQ /). Das stimmt auch, weil es nicht nur den "unmodifiable" Adaptor gibt, sondern auch noch einen "synchronized" Adaptor, und man kann beide Adaptionen beliebig miteinander kombinieren. Dann hätte man gleich 3 Interfaces für jeden der 6 Collection-Typen definieren müssen. Statt der erforderlichen 18 Interfaces hat man gar nichts Neues definieren müssen; die unveränderlichen Listen sind immer noch Listen, weil sie alle Methoden des List-Interfaces unterstützen, obwohl sie einige Methoden eigentlich konzeptionell nicht unterstützen.

"Shallow" Immutability

Sehen wir uns einmal an, was man mit den Immutability-Adaptoren des JDK anfangen kann und wie sie benutzt werden. Als Beispiel verwenden wir wieder unsere Widget-Klasse.

Wenn die Klasse Widget nicht nur die letzte Änderung speichern würde, sondern die gesamte Historie aufzeichnen würde, dann müßte sie intern eine Collection (z.B. eine Liste) von Stamp-Objekten verwalten. Die Klasse Widget könnte dann so aussehen:

```
public final class Widget
{
  private List changeLog;

  ... other methods and fields ...
```

```
  public ImmutableStamp getLastModification();
  public List getChangeLog();
}
```

Dabei nehmen wir an, dass in der privaten Liste Stamp-Objekte abgelegt werden und dass die Stamp-Abstraktion wie im letzten Artikel empfohlen als duale Klasse bestehend aus 2 Klassen, ImmutableStamp und Stamp, implementiert ist. Wie müßten die beiden Zugriffsmethoden getLastModification() und getChangeLog() implementiert werden?

Fangen wir mit der Methode getLastModification() an. Sie soll das letzte Element in der Liste der Stamp-Objekte als ImmutableStamp zurückgeben. Da muss man sich zunächst einmal fragen: was genau soll das bedeuten? Es gibt zwei Möglichkeiten: entweder es wird ein Snapshot dieses letzten Eintrags erzeugt und zurück gegeben oder es wird eine Read-Only-Sicht auf den letzten Eintrag zurück gegeben. Beides würde Sinn machen. Die Read-Only-Sicht wäre eine reine Sicht auf einen bestimmten Eintrag in der Stamp-Liste. Wenn sich dieser Eintrag ändert, dann würde man das sehen können. Der Snapshot dagegen ist unveränderlich. Er ist eine Kopie des Eintrags in der Stamp-Liste, so wie er zum Zeitpunkt des Aufrufs der Methode getLastModification() gerade eingetragen ist.

Hier ist die Implementierung der Methode getLastModification(), wenn sie eine Read-Only-Sicht zurückliefert:

```
public final class Widget
{
  private List changeLog;

  ... other methods and fields ...

  public ImmutableStamp getLastModification()
  {
    return changeLog.get(changeLog.size()-1);
  }
  public List getChangeLog();
}
```

Hier ist die Implementierung der Methode getLastModification(), wenn sie einen Snapshot zurückliefert:

```
public final class Widget
{
  private List changeLog;

  ... other methods and fields ...

  public ImmutableStamp getLastModification()
  {
    return new ImmutableStamp(changeLog.get(changeLog.size()-1));
  }
  public List getChangeLog();
}
```

Etwas komplizierter ist die Lage bei der Methode getChangeLog(). Auch hier muss man sich zunächst fragen: was soll denn genau die Semantik dieser Methode sein? Eines ist klar: sie soll keinen Schreibzugriff auf die private Stamp-Liste des Widget-Objekts zulassen. Das heißt, eine naive Implementierung wie die folgende wäre auf jeden Fall falsch:

```
public List getChangeLog()
  {
    return changeLog;  // don't do this !!!
  }
```

```
Hier wird voller (d.h. auch schreibender) Zugriff auf die private
Liste gibt. Aber wie wäre es damit?
```

```
  public List getChangeLog()
  {
    return Collections.unmodifiableList(changeLog);
  }
```

Hier wird der Immutability-Adaptor aus dem Collection-Framework verwendet, um eine unveränderliche Sicht auf die private Liste zurück zu geben. Das sieht so aus, als hätten wir damit eine Zugriffsmethode implementiert, die eine Read-Only-Sicht auf ein privates Feld eines Objekts gibt. Über eine Read-Only-Sicht kann man üblicherweise das Feld nicht ändern, aber man kann Veränderungen, die von anderen gemacht werden, sehen. Das wäre eine sinnvolle Semantik für eine solche Zugriffsmethode. Bleibt die Frage, haben wir tatsächlich eine Read-Only-Sicht implementiert?

Wenn man genauer hinsieht, stellt man fest, dass zwar Methoden wie add() und remove() auf der zurückgelieferten "unmodifiable" Liste nicht mehr funktionieren, genauer gesagt, sie werfen grundsätzlich eine UnsupportedOperationException. Das heißt, wir haben eine Read-Only-Sicht auf die Collection. Aber Methoden wie get() oder der Zugriff auf Elemente über den Iterator der Collection geben nach wie vor vollen Zugriff auf die enthaltenen Elemente. Die Elemente in der Liste sind Stamp-Objekte, die man dann natürlich nach Belieben modifizieren kann, wie das folgende Beispiel zeigt:

```
Widet w = new Widget();
...
List log = w.getChangeLog();
Iterator iter = log.iterator();
while (iter.hasNext())
{
  Stamp s = (Stamp)iter.next();
  s.setDate(new Date());
  s.setAuthor("Molly Malicious");
}
```

Das heißt, wir haben keine Read-Only-Sicht implementiert. Das Beispiel zeigt, dass der Immutability-Adaptor der Collection allein nicht ausreicht, um Veränderungen an den privaten Daten des Widget zu verhindern. Es genügt nicht, dass wir eine Read-Only-Sicht auf den Container zurück geben. Wir müssten außerdem eine Read-Only-Sicht auf die im Container enthaltenen Element geben.

Das könnten wir in diesem speziellen Beispiel dadurch erreichen, dass wir die gesamte Implementierung der Widget-Klasse umkrempeln und in der Liste nicht Stamp-Objekte, sondern ImmutableStamp-Objekte abgelegen. Das wäre durchaus eine Überlegung wert. Aber diese Lösungsmöglichkeit besteht ja nicht immer. Das geht beispielsweise dann nicht, wenn die Listenelemente nicht als duale Klassen implementiert sind und es den unveränderlichen Typ gar nicht gibt, oder wenn die Collection von der Semantik her tatsächlich veränderliche Element enthalten soll, oder wenn man eine existierende Klasse wie die Widget-Klasse nicht in größerem Stil ändern will.

In solchen Fällen bleibt einem nichts anderes übrig, als eine Kopie des gesamten Containers zu erzeugen. In unserem Beipiel sähe eine Implementierung mit Kopieren des Containers dann so aus:

```
public List getChangeLog()
{
  List copy = new ArrayList();
  Iterator iter = log.iterator();
  while (iter.hasNext())
  {
     copy.add(new ImmutableStamp(iter.next()));
  }
  return Collections.unmodifiableList(copy);
}
```

Jetzt haben wir die Methode getChangeLog() tüchtig umgekrempelt. Sie hat nicht mehr die Semantik, dass sie eine Read-Only-Sicht auf die private Liste der Stamp-Objekte gibt, sondern sie liefert jetzt eine Read-Only-Sicht auf einen Snapshot dieser privaten Liste.

Es sind andere Implementierungen denkbar, zum Beispiel eine Read-Only-Sicht auf eine Kopie einer private Collection, die Read-Only-Sichten auf die Elemente der Original-Collection enthält. Das wäre dann eine echte Read-Only-Sicht. In unserer Lösung geht das nicht, weil unsere duale Stamp-Abstraktion keine Read-Only-Sicht auf veränderliche Stamp-Objekte ermöglicht. In anderen Fällen wäre das aber eine sinnvolle Alternative.

Fazit: Wie schon beim Klonen (siehe /KRE3/) muss man auch bei der Immutability beachten, dass man die Unveränderlichkeit "tief genug" ansetzt. Die Read-Only-Sicht auf einen Container, der veränderliche Objekte enthält, genügt nicht, um sicher zu stellen, dass die Collection auch tatsächlich unverändert bleibt. Man muss außerdem noch dafür sorgen, dass auch die enthaltenen Element unveränderlich sind. Wobei man unterscheiden muss, ob man mit "unveränderlich" eine Read-Only-Sicht oder einen Snapshot meint. Für eine Read-Only-Sicht auf einen Container braucht man eine "unmodifiable collection", die Read-Only-Sichten auf die Element enthält. Für einen Snapshot eines Containers braucht man eine Kopie des Containers, der Kopien der Elemente enthält.

final Variablen

Nach all unseren Betrachtungen über unveränderliche Typen und Read-Only-Sichten auf veränderliche Objekte fragt man sich, welche Unterstützung die Sprache Java eigentlich in diesem Zusammenhang bietet. Wie man an den Beispielen in diesem und dem

vorangegangenen Artikel schon gesehen hat, bietet Java keinerlei Möglichkeiten, lesende Methoden von schreibenden zu unterscheiden, oder um veränderliche von unveränderlichen Typen zu unterscheiden, oder um zu sagen, ob eine Methode eine Referenz auf eine Kopie oder aufs Original liefert. All das muss man verbal in der Dokumentation aufschreiben und auch dort wieder nachlesen. Es gibt lediglich die Möglichkeit, veränderliche von unveränderlichen Variablen zu unterscheiden. Dazu kann man das Schlüsselwort final auf Variablen anwenden.
Der Inhalt einer Variablen, die als final erklärt ist, kann nicht geändert werden. Bei Variablen vom primitivem Typ, d.h. int, long, boolean, etc., bedeutet es, dass sich der Wert der Variablen nach der Initialisierung nicht mehr ändert. Bei Referenzvariablen bedeutet es, dass sich die in der Referenzvariablen gespeicherte Adresse nicht mehr ändert. Mit anderen Worten, eine final Referenzvariable verweist auf das Objekt, das ihr bei der Initialisierung zugewiesen wurde und kann niemals auf ein anderes Objekt verweisen. Es bedeutet aber nicht, dass das referenzierte Objekt vor Veränderungen geschützt ist.
Beispiele:

```
final int max = 256;
max = 0; // error: does not compile

final Date deadline = new Date();
deadline = new Date(100,0,1,0,0,0);     // error: does not compile
deadline.set(new Date(100,0,1,0,0,0));  // fine:  compiles
```

Mit anderen Worten, das Schlüsselwort final hilft im Zusammenhang mit Immutability nur in wenigen Situationen. Beispielsweise hilft es, wenn man einen unveränderlichen Typ bauen will, dessen Felder alle von primitivem Typ sind. Dann kann man alle Felder als final deklarieren und der Compiler kann dann dafür sorgen, dass man nicht versehentlich in irgendeiner Methode diese Felder verändert. Hier ist ein Beispiel:

```
public final class ImmutablePoint
{
  private final float x;
  private final float y;
  private final float z;

  ImmuablePoint(float a, float b, float c) { x=a; y=b; z=c; }

  ... other methods and fields ...

  public float getX() { return x; }
  public float getY() { return y; }
  public float getZ() { return z; }

  public void translate(ImmutablePoint p) // nonsensical method
  {
    x+=p.getX();  // error: does not compile
    y+=p.getY();  // error: does not compile
    z+=p.getZ();  // error: does not compile
  }
}
```

Wenn man aber Felder hat, die Referenzen sind, dann hilft es nicht, wenn man diese Felder als final deklariert, weil es ja nicht genügt, dass die Referenzbeziehung unveränderlich ist; man will eigentlich sagen, dass auch das referenzierte Objekt nicht geändert werden soll. Das kann man aber mit Sprachmitteln in Java gar nicht ausdrücken. Hier ist ein weiteres Beipiel, in dem man die fehlende Unterstützung bei final Referenzvariablen sieht. Wir werden daran zeigen, wie man diese Lücke mit Hilfe von unveränderlichen Typen schließen kann. Betrachten wir folgende Klasse und ihre final Felder:

```
public final class ImmutablePeriod
{
  private final Date begin;
  private final Date end;

  ImmuablePeriod(Date a, Date z) { begin=a; end=z; }

  ... other methods and fields ...

  public Date getBegin() { return begin; }
  public Date getEnd()   { return end; }

  public void stretch(int factor) // nonsensical method
  {
    end.setTime(end.getTime()*factor);  // oops!
  }
}
```

Man kann versehentlich die referenzierten Date-Objekte ändern, ohne dass der Compiler sich beschwert. Da die Sprache keine Sprachmittel bietet, um das zu verhindern, kann nur mit Immutability-Interfaces oder unveränderlichen Typen sagen, was man meint. Wenn man z.B. eine ImmutableDate-Adaptor-Klasse hat, dann kann man es so machen:

```
public final class ImmutableDate
{
  private Date date;

  public ImmutableDate(Date d) { date=d; }
  public long getTime()  { return date.getTime(); }
  public int getYear()   { return date.getYear(); }
  public int getMonth()  { return date.getDate(); }
  public int getDay()    { return date.getDay();
  public int getHour()   { return date.getHour(); }
  public int getMinute() { return date.getMinute(); }
  public int getSecond() { return date.getSecond();
}

public final class ImmutablePeriod
{
  private final ImmutableDate begin;
  private final ImmutableDate end;

  public ImmuablePeriod(Date a, Date z)
  {
    begin=new ImmutableDate(a);
```

```
    end=new ImmutableDate(z);
  }

  ... other methods and fields ...

  public ImmutableDate getBegin() { return begin; }
  public ImmutableDate getEnd()   { return en; }

  public void stretch(int factor)
  {
    end.setTime(end.getTime()*factor);  // error: does not compile
  }
```

Man merke sich, dass final Referenzvariablen, im Gegensatz zu final Variablen von primitivem Typ, nicht dazu führen, dass das referenzierte Objekt unverändert bleibt. Es kommt in der Praxis selten vor, dass man sagen will, dass die Referenzbeziehung zwischen Variable und Objekt unveränderlich ist. Viel häufiger möchte man sagen, dass die Referenzvariable und das referenzierte Objekt unveränderlich ist, aber das kann man in Java mit dem Sprachmittel final nicht ausdrücken.

Zur Vollständigkeit eine letzte Bemerkung zum Schlüsselwort final: Die Anwendung des Schlüsselworts final auf Klassen oder Methoden hat natürlich gar nichts mit Immutability von Variablen und Objekten zu tun, sondern hat Auswirkungen auf die Vererbung. Von final Klassen kann man nicht ableiten und final Methoden kann man nicht überschreiben.

Immutability – eine zusammenfassende Begriffsbestimmung

Wir haben uns in diesem und dem vorangegangenen Artikel ausführlich mit diversen Aspekten von unveränderlichen Typen befasst. Dabei haben wir verschiedene Arten von Immutability gesehen. Fassen wir abschließen noch einmal zusammen, was man eigentlich unter "unveränderlich" versteht:

- **Read-Only-Sicht.**
  Bei dieser Art von Immutability definiert man Interfaces oder Klassen, die ausschließlich lesende Methoden haben. Variablen von diesen Typen können auf veränderliche oder unveränderliche Objekte von Subtypen verweisen.

  Beispiele: Immutability-Interfaces oder auch die "unmodifiable" Adaptoren der JDK-Collections oder der Supertyp einer dualen Klasse wie in /KRE1/ beschrieben.

  Diese Art der Immutability kann leicht missverstanden werden. Eine Read-Only-Sicht bedeutet lediglich, dass man ohne Cast das referenzierte Objekt nicht ändern kann. Eine Read-Only-Sicht ist aber keine Garantie, dass das referenzierte Objekt tatsächlich unverändert bleibt. Es kann andere Referenzen auf dasselbe Objekt geben, über die das Objekt verändert werden kann. Man kann auch jederzeit einen Cast auf den "echten" Typ des Objekts machen. Falls der "echte" Typ Veränderungen zuläßt, dann kann man das referenzierte Objekt sogar selber verändern.

- **"Echte" Unveränderlichkeit.**
  Bei dieser Art von Immutability hat man ebenfalls Klassen, die ausschließlich lesende Methoden haben. Aber diesmal können Variablen von diesen Typen nur auf unveränderliche Objekte verweisen. Typischerweise sind solche Klassen als final deklariert, d.h. es gibt gar keine Subklassen.

  Beispiel: final Klassen, die nur lesende Methoden haben, z.B. String, oder der unveränderliche Teil einer dualen Klasse (siehe /KRE1/).

  Das ist die Art von Unveränderlichkeit, bei der man das Object Sharing bedenkenlos zulassen kann, weil das Objekt über keine Referenz jemals geändert werden kann. Man braucht Objekte diesen Typs nicht zu kopieren und man kann in Multithread-Programmen auf die Synchronisation der Zugriffe auf Objekte diesen Typs verzichten.

- **final Variable von primitivem Typ.**
  Solche Variablen sind ebenfalls "echt" unveränderlich.

  Das ist der einzige Fall, wo man mit Mitteln der Sprache, nämlich über die Qualifizierung mit final, ein Objekt als unveränderlich kenzeichnen kann.

- **final Referenzvariable.**
  Die Art von Immutability bringt man zum Ausdruck durch die Kennzeichnung einer Referenzvariablen als final Variable.

  Es bedeutet, dass eine unveränderliche Beziehung zwischen Referenzvariable und Objekt besteht. Es bedeutet aber nicht, dass das referenzierte Objekt unveränderlich ist

Tipps für den Umgang mit unveränderlichen Typen in der Praxis

Benutzung

Wenn man unveränderliche Typen verwendet, dann muss man sich immer genau überlegen, ob man einen echt unveränderlichen Typ oder nur eine Read-Only-Sicht auf etwas möglicherweise Veränderliches vor sich hat. Beides kann sinnvoll und nützlich sein, je nach Kontext. Man darf diese beiden Arten von Immutability nur nicht miteinander verwechseln.

Die Read-Only-Sicht sagt, dass man in diesem Kontext keine Veränderungen des referenzierten Objekts machen will. Eine Garantie ist das allerdings nicht. Nur die echt unveränderlichen Typen garantieren, dass die referenzierten Objekte sich nicht ändern. Und nur Objekte, die sich tatsächlich nicht ändern, kann man bedenkenlos als Shared Objects verwenden.

Implementierung

Wenn man unveränderliche Typen implementiert, muss man sich ebenfalls überlegen, was man eigentlich mit "unveränderlich" meint. Man muss es sorgfältig in der Dokumentation beschreiben und man muss es entsprechend implementieren. Insbesondere muss man

wissen, wie man lesende Zugriffsfunktionen implementiert; das haben wir im vorangegangenen Artikel (siehe /KRE1/) besprochen.

Das sicherste Beispiel einer Abstraktion, die alle Formen der Immuability unterstützt, ist die duale Klasse, wie wir sie in im vorangegangenen Artikel (siehe /KRE1/) beschrieben und in vereinfachter Form (siehe Stamp-Abstraktion) in diesem Artikel verwendet haben. Sie besteht aus einer Superklasse oder einem Super-Interface, dass die Read-Only-Sicht repräsentiert, und zwei final Subklassen, die den veränderlichen und den echt unveränderlichen Typ implementieren. Über Konstruktoren und Casts kann zwischen den verschiedenen Typen hin und her transformiert werden. Ob man allerdings für eine Abstraktion den Implementierungsaufwand einer dualen Klassen spendieren will, muss man im Einzelfall entscheiden. Das wird man sicher nicht für alle Abstraktionen tun.

Literaturverweise

/KRE1/ **Unveränderliche Typen in Java (Teil 1)**
Klaus Kreft & Angelika Langer
JavaSpektrum, März 2003
http://www.AngelikaLanger.com/Articles/EffectiveJava/08.Immutability-Part1/08.Immutability-Part1.html

/KRE2/ **Unveränderliche Typen in Java (Teil 2)**
Klaus Kreft & Angelika Langer
JavaSpektrum, Juli 2003
http://www.AngelikaLanger.com/Articles/EffectiveJava/09.Immutability-Part2/09.Immutability-Part2.html

/KRE3/ **Das Kopieren von Objekten in Java (Teil 1 - 3)**
Klaus Kreft & Angelika Langer
JavaSpektrum, September 2002 + November 2002 + Januar 2003
http://www.AngelikaLanger.com/Articles/EffectiveJava/05.Clone-Part1/05.Clone-Part1.html
http://www.AngelikaLanger.com/Articles/EffectiveJava/06.Clone-Part2/06.Clone-Part2.html
http://www.AngelikaLanger.com/Articles/EffectiveJava/07.Clone-Part3/07.Clone-Part3.html

/JDK/ **Java 2 Platform Std. Ed. v1.4 - Package java.util**
http://java.sun.com/j2se/1.4/docs/api/index.html

/FAQ/ **Java Collections API Design FAQ**
http://java.sun.com/j2se/1.3/docs/guide/collections/designfaq.html#1

/DAV1/ **Durable Java – Immutables**
Mark Davis
URL: http://www.macchiato.com/columns/Durable2.htm

/DAV2/ **Durable Java – Abstraction**
Mark Davis
URL: http://www.macchiato.com/columns/Durable3.htm

/GOE1/ **Java theory and practice: To mutate or not to mutate?**
Immutable objects can greatly simplify your life
Brian Goetz
IBM developerWorks, February 18, 2003
URL: http://www-106.ibm.com/developerworks/java/library/j-jtp02183.html

/GOE2/ **Java theory and practice: Is that your final answer?**
Guidelines for the effective use of the final keyword
Brian Goetz
IBM developerWorks, October 1, 2002
URL: http://www-106.ibm.com/developerworks/java/library/j-jtp1029.html