Das Kopieren von Objekten in Java

Teil 1: Was ist clone()? Wofür braucht man es? Warum sollte man es implementieren?

JavaSPEKTRUM, September 2002
Klaus Kreft & Angelika Langer

Mit diesem Artikel wollen wir die Serie über die Infrastruktur von Objekten in Java fortsetzen. Nachdem wir uns in den vergangenen Artikel ausführlich mit der Thematik "Objekt-Vergleich" befasst haben, wollen wir uns nun einem anderen Grundlagenthema zuwenden - dem Kopieren von Objekten. Wann und warum braucht man überhaupt Kopien von Objekten? Wie erzeugt man eine Kopie? Welche Infrastruktur muss für das Kopieren zur Verfügung gestellt werden? In diesem Kontext spielen das Cloneable-Interface und die clone()-Methode eine Rolle. Wie spielen sie zusammen? In welcher Beziehung stehen Object.clone(), das Cloneable-Interface und die clone()-Methode der eigenen Klasse? Braucht man clone() überhaupt oder gibt es Alternativen? Wie kopiert man Objekte mit und ohne clone()? Was sind die Vor-und Nachteile der verschiedenen Techniken? Diese Fragen diskutieren wir in der vorliegenden Ausgabe unserer Kolumne. In der nächsten Ausgabe werden wir dann die Implementierung von clone()diskutieren

Wofür braucht man clone() ?

In Java unterscheidet man zwischen Variablen vom primitivem Typ (wie char, short, int, double, etc.) und Referenzvariablen (von einem class oder interface-Typ). Variablen von primitivem Typ enthalten einen Wert des entsprechenden Typ und sowohl beim Zuweisen und Vergleichen als auch bei der Übergabe an und Rückgabe von Methoden wird immer der enthaltene Wert übergeben bzw. verglichen. Das ist bei Referenzvariablen anders. Das Objekt, auf das eine Referenzvariable verweist, wird per Referenz verwaltet und herumgereicht. Beim Zuweisen und Vergleichen per Zuweisungs- und Vergleichsoperator werden lediglich die Adressen der referenzierten Objekte zugewiesen bzw. verglichen; die referenzierten Objekte spielen gar keine Rolle. Ebenso wird bei der Übergabe an oder Rückgabe von Methoden nur die Adresse von Objekten übergeben, nicht jedoch das referenzierte Objekt selbst.

Die Referenzsemantik in Java spart Overhead, den das Kopieren der Objekte verursachen würde, führt aber andererseits zu manchmal unerwünschten Beziehungsverflechtungen. Bisweilen ist es in Java schwer, den Überblick darüber zu behalten, wer wann zu welchem Zweck eine Referenz auf ein bestimmtes Objekt hält und was der Betreffende mit der Referenz anstellt. Da es in der Sprache auch kein Konzept und Sprachmittel zum Schutz der referenzierten Objekte vor Modifikationen gibt, kann im Prinzip jeder, der eine Referenz auf ein Objekt hält, das referenzierte Objekt ändern. Das kann zu überraschenden Effekten führen.

Sehen wir uns ein typisches Beispiel für die Schwierigkeiten mit der Referenzsemantik an:

```
class ColoredPoint {
  private Point p;
  private int color;
  public ColoredPoint (Point newP, int newColor)
  { p = newP; color = newColor; }                          // (1)
}
...
Point[] createLine(int len, int m, int c) {
  Point nextPoint = new Point(0,c);                        // (2)
  Point [] line = new Point[len];
  for (int i=0; i<len; i++) {
    line[i] = new ColoredPoint(nextPoint , 0xFF00FF);      // (3)
    nextPoint.x += 1;
    nextPoint.y += m; }
  return line;
}
```

Wir haben eine Klasse ColoredPoint, die einen Punkt mit zwei Koordinaten und eine Farbe enthält. Der Konstruktor der Klasse ColoredPoint bekommt Initialwerte für diese beiden Daten und merkt sie sich in entsprechenden privaten Feldern. Die Methode createLine() erzeugt ein Array von Points, das eine Linie beschreibt. Wie sieht die Linie aus, die von createLine() erzeugt wird? Nicht ganz so, wie sich der Autor das gedacht hat. Wo ist das Problem?

Das Problem liegt in der Referenz-Semantik der Variablen in Java. Die Methode createLine() berechnet für jeden Punkt in dem Array, das die Linie beschreiben soll, die jeweiligen Koordinaten. Diese Koordinaten werden in dem Point-Objekt nextPoint abgelegt (siehe Codezeile (2)). Dieses Point-Objekt wird benutzt, um jeweils einen neuen ColoredPoint zu erzeugen, dessen Referenz schließlich im Array abgelegt wird (siehe Codezeile (3)).

Das Missverständnis besteht darin, dass createLine() offensichtlich davon ausgeht, dass der Konstruktor von ColoredPoint sich den Point, der als Konstruktor-Argument übergeben wird, merkt, indem er sich eine Kopie davon anlegt. Tatsächlich merkt sich der ColoredPoint-Konstruktor aber nur die Adresse des übergebenen Point-Objekts; die Zuweisung p = newP; (siehe Code-Zeile (1)) ist eine Zuweisung von Referenzvariablen und das ist in Java lediglich die Zuweisung der Objekt-Adresse, nicht des Objekt-Inhalts. Die Linie wird also aus len-vielen Punkten bestehen, die alle die zuletzt berechneten Koordinaten enthalten, weil sie alle auf das eine Point-Objekt nextPoint verweisen, das am Anfang der Methode createLine() erzeugt wurde.

Object Sharing

Das Beispiel demonstriert eine in Java typische Situation, die häufig dann auftritt, wenn Argumente an Konstruktoren übergeben werden, die der Konstruktor sich dann in Feldern der Klasse merken will. In solchen Fällen will die Klasse oft keine Referenz auf das übergebene Objekt halten, sondern will ihre eigene Kopie davon haben. Die Kopie hat den Vorteil, dass sie nicht mit anderen Objekten geteilt werden muss. In obigem Beispiel ist genau das Gegenteil

eingetreten: mehrere ColoredPoint-Konstruktoren haben sich Referenzen auf ein einziges Point-Objekt gemerkt und damit dieses Point-Objekt zum Gemeinschaftsgut gemacht. Eine solche Situation bezeichnet mal als Object Sharing und sie kann zu Problemen führen, wie in obigem Beispiel: das Object Sharing hat sich später in der createLine()-Methode negativ bemerkbar gemacht, weil das gemeinsam verwendete Point-Objekt verändert wurde. In unserem Beispiel wäre es sicher besser gewesen, wenn der Konstruktor eine Kopie angelegt hätte und sich eine Referenz auf seine eigene Kopie des Point-Objekts gemerkt hätte. Wie man sieht, führt das Object-Sharing leicht zu Problemen und kann durch das Anlegen von Kopien vermieden werden.

Unerwünschtes Object-Sharing tritt nicht nur im Zusammenhang mit Konstruktoren auf. Eine ähnliche Situation liegt beispielsweise vor, wenn Objekte von Methoden zurückgegeben werden. Wenn etwa die Methode einer Klasse eine Referenz auf ein Feld der Klasse zurückliefert, dann bekommen alle Empfänger des Returnwerts eine Referenz auf ein gemeinsam verwendetes Objekt. Auch das ist oft unerwünscht und der Empfänger will eigentlich seine eigene Kopie des zurück gelieferten Objekts haben. Um das zu erreichen, könnte die betreffende Methode jedes Mal eine Kopie anlegen und eine Referenz auf die jeweilige Kopie zurückgeben.

Woher weiß man eigentlich, ob bei der Übergabe von Referenzen an und von Methoden die Gefahr eines unerwünschten Object-Sharings besteht? Woran kann man erkennen, ob eine Referenz, die von einer Methode zurückgegeben wird, auf das Original verweist oder auf eine Kopie? Oder, betrachten wir unser Beispiel: woran hätte der Autor der createLine()-Methode erkennen können, wie der ColoredPoint-Konstruktor mit der übergebenen Point-Referenz umgeht? Ansehen kann man das einer Methode in Java nicht. Solche Details müssen in der JavaDoc-Beschreibung der Methode dokumentiert sein. Deshalb sollten generell alle Methoden, die Referenzen bekommen oder zurückgeben, in der JavaDoc klare Aussagen über die Benutzung der Referenz machen. Bei der Rückgabe von Referenzen muss klar sein, ob die gelieferte Referenz aufs Original-Objekt verweist und zu Objekt-Sharing führt, oder ob die Methode bereits von sich aus eine Kopie angelegt hat und eine Referenz auf diese Kopie zurückliefert. Bei der Übergabe von Referenzen an eine Methode muss ebenfalls geklärt sein, ob die Methode intern eine Kopie des referenzierten Objekts anlegt und verwendet, oder ob die Methode mit dem referenzierten Original-Objekt arbeitet. Im letzteren Fall muss ggf. der Aufrufer vor dem Aufruf der Methode bereits eine Kopie anlegen, wenn ein Objekt-Sharing verhindert werden soll. Der Aufrufer kann zur Sicherheit immer eine Kopie anlegen, ganz egal was die gerufene Methode macht, aber das ist natürlich nicht die effizienteste Lösung, weil unter Umständen unnötig oft kopiert wird. In jedem Fall muss die Arbeitsteilung zwischen Aufrufer und Methode geklärt und in der JavaDoc dokumentiert sein. Ohne klare Beschreibung in der JavaDoc kann kein Benutzer einer Methode wissen, ob er zur Vermeidung von Objekt-Sharing vor oder nach dem Aufruf der Methode Kopien angelegen muss oder nicht.

Im Zusammenhang mit der Übergabe von Referenzen an und von Methoden kommt es nicht automatisch immer zu Object-Sharing-Situationen. Solche Situationen treten nur auf, wenn beide (Aufrufer und gerufene Methode bzw. Klasse) das referenzierte Objekt nach dem Aufruf noch weiter verwenden wollen, wie etwa in unserem Beispiel mit createLine(): wenn

createLine() darauf verzichtet hätte, das Point-Objekt, das an den ColoredPoint-Konstruktor übergeben wurde, weiter zu verwenden, dann wäre überhaupt kein problematisches Object Sharing entstanden. Analog bei der Rückgabe von Referenzen: wenn eine Methode eine Referenz auf ein Objekt zurückgibt, dass sie gerade eben mit new angelegt hat, dann kann auch nichts passieren. Das Problem tritt nur auf, wenn die Methode eine Referenz zurück gibt, die auch später noch der Methode (oder anderen Methoden der Klasse) zugänglich ist, etwa weil die Referenz auf das zurück gegebene Objekt in einem Feld der Klasse abgelegt ist. Dann hat sowohl der Aufrufer über die zurückgegebene Referenz Zugriff auf das Objekt als auch die Klasse mit all ihren Methoden. Wenn aber die zurückgegebene Referenz nirgendwo hinterlegt wurde, dann hat nur der Aufrufer Zugriff aufs Objekt und ein problematisches Object-Sharing kommt überhaupt nicht zustande.

Das Anlegen von Kopien ist im Übrigen nicht die einzige Antwort auf unerwünschtes Object-Sharing. Die oben geschilderten Probleme lassen sich unter Umständen auch ohne Kopien lösen, zum Beispiel mit Immutability-Adaptoren. Wenn das gemeinsam verwendete Objekt nämlich unveränderlich (immutable) ist, dann stört das Object Sharing nicht, und dann ist auch nicht nötig, Kopien anzulegen. Immutability wollen wir aber in dieser Ausgabe der Kolumne nicht besprechen. Wir wollen uns stattdessen ansehen, wie man Kopien von Objekten erzeugt, wenn man solche Kopien braucht.

Das Kopieren von Objekten

Für das Erzeugen von Kopien von Objekten gibt es in Java mehrere Möglichkeiten. Eine Klasse, die es ermöglichen will, dass Kopien von ihren Objekten erzeugt werden, kann eine clone()-Methode implementieren und/oder einen sogenannten Copy-Konstruktor zur Verfügung stellen. Es gibt auch noch andere Beispiele für Kopierfunktionalität, die aber ebenfalls auf Konstruktoren beruhen.

Klonen per clone()-Methode

Wenn eine Klasse eine clone()-Methode hat, dann können Kopie mit Hilfe dieser Methode erzeugt werden. clone() erzeugt ein neues Objekt vom gleichen Typ mit gleichem Inhalt und gibt eine Referenz auf das neue Objekt als Ergebnis zurück. Klassen, die eine clone()-Methode implementieren, müssen zusätzlich das Cloneable-Interface implementieren. Das Cloneable-Interface ist ein reines Marker-Interface, d.h. es ist leer, und definiert nicht etwa die clone()-Methode, wie man erwarten könnte. Es wird lediglich verwendet, um klonbare (cloneable) Klassen von nicht-klonbaren Klassen zu unterscheiden. Wofür das gebraucht wird, sehen wir uns später noch im Detail an. Schauen wir erst einmal ein Beispiel für eine cloneable Klasse an. Die JDK-Klasse java.util.Date ist ein Beispiel:

```
public class Date implements Cloneable {
   ...
   public Object clone() { ...  }
   ...
}
```

Kopieren per Copy-Konstruktor

Wenn eine Klasse einen Copy-Konstruktor hat, dann kann man diesen Konstruktor verwenden, um Kopien zu erzeugen. Das ist eine Alternative zur clone()-Methode. Der Begriff "Copy-Konstruktor" stammt aus C++. Man bezeichnet damit einen Konstruktor, der ein Objekt vom eigenen Typ als Argument akzeptiert und ein neues Objekt vom gleichen Typ mit gleichem Inhalt - nämlich die Kopie - erzeugt. Die JDK-Klasse java.lang.String ist ein Beispiel für eine solche Klasse:

```
public final class String {
  ...
  public String(String original) { ... }
  ...
}
```

Andere Formen des Kopierens

Daneben gibt es Klassen, die werden copy-konstruierbar noch cloneable sind. Die JKD-Klasse java.lang.StringBuffer ist ein solches Beispiel:

```
public final class StringBuffer {
  public StringBuffer(String str) { ... }
  public String toString() { ... }
}
```

Man kann eine Kopie eines StringBuffer erzeugen, indem man das Original in einen String konvertiert und aus diesem String einen neuen StringBuffer konstruiert:

```
StringBuffer copy = new StringBuffer(original.toString());
```

Auf die Vor- und Nachteile der verschiedenen Techniken gehen wir nächsten Artikel genauer ein. Es wird sich herausstellen, dass clone() die für das Kopieren zu empfehlende Technik ist. Hier wollen wir uns zunächst ansehen, was von einer Implementierung der clone()-Methode genau erwartet wird.

Der clone()-Contract

Die Anforderungen an die clone()-Methode einer Klasse sind im sogenannte clone()-Contract beschrieben, den man in der JavaDoc unter Object.clone() findet. Hier ist der Original-Wortlaut:

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object.
The general intent is that, for any object x, the expression:
x.clone() != x
will be true, and that the expression:
x.clone().getClass() == x.getClass()
will be true, but these are not absolute requirements. While it is typically the case that:
x.clone().equals(x)
will be true, this is not an absolute requirement.
Copying an object will typically entail creating a new instance of its class, but it also may require copying of internal data structures as well. No constructors are called.

Das bedeutet das Folgende:

- Klon und Original sind verschiedene Objekte, d.h. sie sind an verschiedenen Stellen im Speicher angelegt.
- Klon und Original sind vom selben Typ.
- Klon und Original sollten gleich sein im Sinne von equals(), d.h. sie sollten den gleichen Inhalt haben.

Die Methode Object.clone()

Wenn man eine Klasse cloneable machen will, dann muss die Klasse das Cloneable-Interface implementieren und eine clone()-Methode, typischerweise mit der Signatur public Object clone(), definieren. Im einfachsten Fall implementiert man die clone()-Methode, indem man super.clone() aufruft.

```
class MyClass implements Cloneable {
  ...
  public Object clone() {
    try { return super.clone(); } catch (CloneNotSupportedException e)
{}
  }
  ...
}
```

In diesem einfachen Fall hat man einfach nur die geerbte Methode Object.clone() als public-Methode zugänglich gemacht. Die Superklasse Object hat nämlich eine clone()-Methode, aber die ist protected und steht damit im public Interface ihrer Subklassen nicht automatisch zur Verfügung. Deshalb sind Java-Klassen zunächst einmal nicht cloneable; man muss die clone()-Methode erst einmal zugänglich machen.

Dazu genügt es nicht, eine public clone()-Methode zur Verfügung zu stellen, sondern die Klasse muss zusätzlich das Cloneable-Interface implementieren, sonst gibt es eine CloneNotSupportedException. Das Implementieren des Cloneable-Interface ist nötig, weil

Object.clone() prüft, ob das this-Object von einem Typ ist, der das Cloneable-Interface implementiert. Falls man clone() auf einem Objekt aufruft, das nicht cloneable ist, dann wirft Object.clone() eine CloneNotSupportedException. Das Zugänglich-Machen der clone()-Methode reicht also noch nicht; die Klasse muss außerdem immer auch das Cloneable-Interface implementieren.

Bei dieser Prüfung wird deutlich, dass das Cloneable-Interface als Marker-Interface dient: die Methode Object.clone() verwendet es zur Unterscheidung zwischen klonbaren und nicht-klonbaren Objekten.

- Die nicht-klonbaren Objekte dürfen nicht geklont werden; deshalb wird eine CloneNotSupportedException geworfen.
- Für die klonbaren Objekte erzeugt Object.clone() einen Klon. Dazu alloziert Object.clone() den benötigten Speicher und kopiert alle Felder des Objekts "as if by assignment", wie es in der Spezifikation heißt. Das bedeutet, dass Object.clone()alle Felder von this an die korrespondierenden Felder des neu erzeugten Objekts zuweist, was bei Referenzen heißt, dass nur die Referenz, nicht aber das referenzierte Objekt kopiert wird. Solche Kopien bezeichnet man als "flache Kopie" (shallow copy), im Gegensatz zur "tiefen Kopie" (deep copy), bei der alle Referenzen rekursiv verfolgt und auch die referenzierten Objekte kopiert werden.

Arrays werden im Übrigen implizit als cloneable angesehen und haben eine clone()-Methode, die eine "shallow copy" des Arrays anlegt: es werden alle Felder des Arrays kopiert; wenn die Felder Referenzen sind, werden nur die Referenzen, nicht aber die referenzierten Objekte kopiert.

Object.clone() ist als native Methode implementiert, d.h. sie ist nicht in Java, sondern in einer anderen Programmiersprache implementiert. Im Prinzip kann man sich die Implementierung der Methode Object.clone() so vorstellen, dass sie erst prüft, ob das this-Objekt cloneable ist. Wenn ja, dann wird Speicher in ausreichender Menge besorgt und der Inhalt von this wird bitweise kopiert. Das ergibt dann genau den Effekt einer "shallow copy".

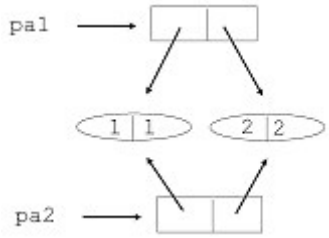
Shallow Copy vs. Deep Copy

In unserem Beispiel einer ersten einfachen Implementierung von MyClass.clone() (siehe oben) haben wir eine clone()-Methode implementiert, die eine flache Kopie des Originals erzeugt. Nun ist die Frage: ist diese Implementierung korrekt? Oder anders gesagt, wann sind flache Kopien ausreichend bzw. unzureichend? Sehen wir uns das einmal am Beispiel der clone()-Methode von Arrays an, die ja ebenfalls eine flache Kopie des Arrays erzeugt.

```
Point[] pa1 = { new Point(1,1), new Point(2,2) };
Point[] pa2 = null;

try {
 pa2 = (Point[]) pa1.clone();
} catch (CloneNotSupportedException e) { ... }
```

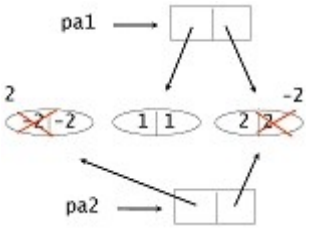
Hier wird ein Point-Array geklont per Aufruf der clone()-Methode für Arrays. Danach sieht die Situation wie folgt aus:



Was passiert, wenn man eines der beiden Point-Arrays manipuliert? Hier ist ein Beispiel mit ein paar Modifikationen des Klons pa2:

```
pa2[0] = new Point(-2,-2);
pa2[0].x =  2;
pa2[1].y = -2;
```

Auf den ersten Blick würde man annehmen, dass nur der Klon pa2 sich ändert, weil alle Zuweisungen im gezeigten Code sich auf pa2 beziehen. Aber so einfach ist das nicht. Da der Klon eine flache Kopie des Originals ist, wirken sich einige der Modifikationen auch auf das Original pa1 aus:

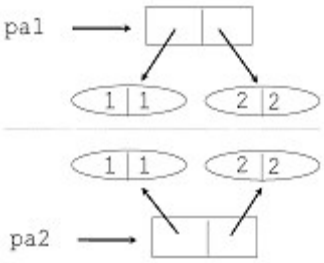


Das ist nicht ganz das, was man sich unter einem Klon vorstellt. Die Idee des Klonens oder Kopierens ist, dass Original und Kopie voneinander unabhängig sind, d.h. Veränderungen des einen Objekts sollen keine Auswirkungen auf das andere Objekt haben. Das ist hier ganz offensichtlich nicht erreicht worden; das geklonte Array erfüllt die Unabhängigkeitsanforderung nicht. Um eine Unabhängigkeit von Original und Klon zu erreichen, müssten wir hier eine tiefe Kopie machen. Das könnte man wie folgt implementieren:

```
Point[] pa1 = { new Point(1,1), new Point(2,2) };
Point[] pa2 = null;

try {
  pa2 = pa1.clone();
  pa2[0] = (Point) pa2[0].clone();
  pa2[1] = (Point) pa2[1].clone();
} catch (CloneNotSupportedException e){ ... }
```

Hier wird nicht nur das Array, sondern es werden auch alle Array-Elemente kopiert. Jetzt haben Original und Klon wirklich nichts mehr miteinander zu tun und Veränderungen des einen betreffen den anderen nicht.



Wann ist eine Kopie "tief genug"?

Wie das Beispiel zeigt, erreicht man mit einer tiefen Kopie das angestrebte Ziel nämlich, dass Original und Klon voneinander unabhängig sind. Der Aufwand für die tiefe Kopie ist aber nicht in allen Fällen erforderlich. Man unterscheidet 3 Fälle:

- Arrays von primitivem Typ
- Arrays von Referenzen auf unveränderliche Objekte
- Arrays von Referenzen auf veränderliche Objekte

Arrays von primitivem Typ. Nehmen wir als Bespiel ein Array von int-Werten, das wir klonen wollen. Die flache Kopie, die von clone() für Arrays erzeugt wird, ist bereits tief genug. Das liegt daran, dass Variablen von primitivem Typ ihre Werte enthalten und nicht auf sie verweisen. Deshalb ist die bitweise Kopie des Array-Elements, die von clone() erzeugt wird, tatsächlich eine Kopie des int-Werts selbst und es ist über den Aufruf von clone() für das Arrray hinaus nichts weiter zu tun, um einen echten Klon eines int-Arrays zu erzeugen.

```
int[] ia1 = { 1, 2 };
int[] ia2 = null;

try {
   ia2 = (int[]) ia1.clone();

} catch (CloneNotSupportedException e) { ... }
```

Abbildung 1: Klonen eines Arrays von primitivem Typ

Arrays von Referenzen auf unveränderliche Objekte. Für ein Array von Referenzen auf unveränderliche Objekte ist die flache Kopie, die von clone() für Arrays erzeugt wird, bereits tief genug. Das Ergebnis der flachen Kopie sind zwei Arrays, die die gleichen Adressen enthalten und damit auf dieselben Objekte verweisen. Da die referenzierten Objekte aber nicht verändert werden können, ist das Object-Sharing unproblematisch. Betrachten wir als Beispiel ein Array von Strings:

```
String[] sa1 = { "one", "two" };
String[] sa2 = null;

try {
 sa2 = (String[]) sa1.clone();
} catch (CloneNotSupportedException e) { ... }
```

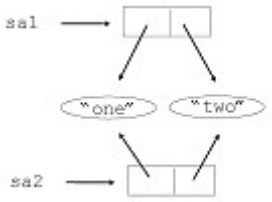


Abbildung 2: Klonen eines Arrays von Referenzen auf unveränderliche Objekte

Warum sind Original und Kopie in diesem Beispiel voneinander unabhängig, obwohl sie alle Array-Elemente gemeinsam referenzieren? Sehen wir uns an, welche Modifikation überhaupt auftreten können. Veränderungen des Originals und der Kopie betreffen jeweils nur die Arrays selbst, aber niemals die gemeinsam verwendeten String-Objekte. Die beiden gemeinsam referenzierten Strings "one" und "two" können nicht modifiziert werden, weil die Klasse String keine modifizierenden Methoden zur Verfügung stellt. Ein Aufruf wie sa1[1].concat(" steps") zum Beispiel sieht zwar so aus, als verändere er den String "two", aber in Wirklichkeit erzeugt dieser Aufruf einen neuen String mit Inhalt "two steps". Dieser neue String kann dann den alten ersetzen, z.B. durch sa1[1] = sa1[1].concat(" steps"), aber davon ist das andere Array sa2 nicht betroffen.

Arrays von Referenzen auf veränderliche Objekte. Das ist der Fall, den wir am Beispiel des Point-Arrays bereits ausführlich diskutiert haben. Hier reicht die flache Kopie nicht und es müssen neben dem Array auch alle referenzierten veränderlichen Array-Elemente kopiert werden, damit Original und Klon voneinander unabhängig sind.

```
Point[] pa1 = { new Point(1,1), new Point(2,2) };
Point[] pa2 = null;

try {
  pa2 = pa1.clone();
  pa2[0] = (Point) pa2[0].clone();
  pa2[1] = (Point) pa2[1].clone();
} catch (CloneNotSupportedException e){ ... }
```

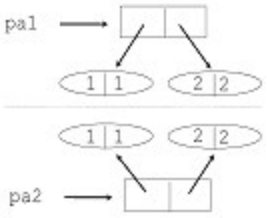


Abbildung 3: Klonen eines Arrays von Referenzen auf veränderliche Objekte

Was wir hier am Beispiel von Arrays beschrieben haben, gilt ganz analog auch für Klassen.

Arrays haben *Elemente*, die entweder von primitivem Typ sind oder aber Referenzen auf veränderliche oder unveränderliche Objekte. Die clone()-Methode für Arrays kopiert sämtliche Elemente bitweise. Je nach Art der Elemente genügt das oder es muss eine tiefe Kopie gemacht werden, wie oben beschrieben.

Objekte, d.h. Instanzen von Klassen, haben *Felder*, die entweder von primitivem Typ sind oder aber Referenzen auf veränderliche oder unveränderliche Objekte. Wenn man die clone()-Methode für eine solche Klasse implementieren will, dann wird man alle nicht-statischen Felder kopieren, so wie das Array-clone() sämtliche Array-Elemente kopiert.

Ganz analog zum Array stellt sich auch hier die Frage: genügt eine bitweise Kopie der Felder oder müssen Referenzen verfolgt werden und tiefe Kopien angelegt werden? Die Antwort ist

dieselbe wie für Arrays: wenn die Felder von primitiven Typ sind oder Referenzen auf unveränderliche Objekte, dann genügt normalerweise die bitweise Kopie (die übrigens von Object.clone() bereits erzeugt wird). Wenn die Felder Referenzen auf veränderliche Objekte sind, dann muss eine tiefe Kopie angelegt werden.

Allgemein kann man die Regel für die Tiefe der beim Klonen zu erzeugenden Kopie wie folgt formulieren: das Original-Objekt (oder -Array) und sein Klon müssen so unabhängig voneinander sein, dass keine Operation auf dem Original den Klon betrifft und umgekehrt. Alle Implementierungen von clone() sollten dieser Anforderung genügen. In der Praxis findet man manchmal clone()-Methoden, die keine ausreichend tiefe Kopie liefern; clone() für Arrays ist ein Beispiel, wie wir oben gesehen haben. Solche Implementierungen sollte man bei eigenen Klassen vermeiden. Es kann zwar vorkommen, dass man keine hinreichend tiefe Kopie erzeugen kann (wir werden im nächsten Artikel sehen warum), aber in solchen Fällen sollte man dann lieber gar kein clone() als ein inkorrektes clone() zur Verfügung stellen.

Und noch ein Hinweis: Wenn eine Klasse keine clone()-Methode definiert, dann sollte sie auch nicht das Cloneable-Interface implementieren. Dann klingt zwar fast wie ein Witz, kann aber vorkommen, weil das Cloneable -Interface leer ist. Man kann in der Tat (absichtlich oder versehentlich) eine Klasse definieren, die das Cloneable-Interface implementiert, aber keine clone()-Methode hat. Das ist zwar von der Logik her widersinnig, aber syntaktisch völlig in Ordnung. Der Compiler lässt das durchgehen, weil das Cloneable -Interface keine einzige Methode vorschreibt, auch keine clone()-Methode.

clone() und Generische Collections

Das leere Cloneable-Interface macht auch sonst noch Schwierigkeiten, beispielsweise beim Kopieren von generischen Collections. Unter generischen Collections versteht man heterogene Container, die Elemente verschiedenen Typs enthalten. Das einfachste Beispiel ist ein Array von Objects. Jedes Array-Element ist eine Referenz auf ein Objekt eines beliebigen Klassen- oder Interface-Typs in Java. Wie kann man so ein Object-Array klonen oder kopieren?

```
public class MyClass implements Cloneable {
  private Object oa[];

  public Object clone() {
    Object[] tmp = oa.clone();

    for (int i=0; i<oa.length; i++)
       ... clone each array element ...
    return tmp;
  }
}
```

Für solche Situationen gibt es die clone()-Methode. Im Prinzip ist es so gedacht, dass man für jedes Array-Element die clone()-Methode aufruft, vorausgesetzt das Array-Element ist überhaupt cloneable. clone() ist eine non-final Methode und so würde dann für jedes Element, egal welchen Typs es zur Laufzeit ist, die clone()-Methode dieses Typs angestoßen. Das würde dann so aussehen:

```
...
for (int i=0; i<oa.length; i++)
  if (oa[i] instanceof Cloneable) {
     tmp[i] = ((Cloneable) oa[i]).clone();
  }
...
```

Leider beschwert sich aber der Compiler über diesen wohlgemeinten Versuch, die clone()-Methode aufzurufen - und zu recht. Wir haben zwar ordnungsgemäß von Object nach Cloneable gecastet, um clone() nur dann aufzurufen, wenn das Objekt cloneable ist, und um die CloneNotSupportedException zu vermeiden. Aber da das Cloneable-Interface leer ist, gibt uns der Cast keinen Zugriff auf die clone()-Methode. So geht's also nicht; per Cast haben wir keine Chance clone() aufzurufen, solange wir den echten Typ des Objekts nicht kennen. Da bleibt dann nur eine Lösung: man muss sich zur Laufzeit Information darüber beschaffen, ob das Objekt von einem Typ ist, der die clone()-Methode implementiert und wenn ja, dann muss man diese clone()-Methode aufrufen. Für solche Aufgaben gibt es Reflection in Java.

Aufruf von clone() über Reflection

Im Package java.lang.reflect (zum Teil auch im Package java.lang) liefert der JDK Funktionalität, mit der man zur Laufzeit Meta-Information über Java-Typen beschaffen und benutzen kann. Man kann sich mit Hilfe der Methode getClass(), die bereits in Object definiert ist, ein Objekt vom Typ Class geben lassen, welches den Typ des Objekts repräsentiert, auf dem die getClass()-Methode aufgerufen wurde. Mit diesem Class-Objekt kann u.a. Information über Felder und Methoden der Klasse besorgt werden. In unserem Fall interessieren wir uns für eine bestimmte Methode, nämlich die clone()-Methode, die wir aufrufen wollen. Das sieht wie folgt aus:

```
...
for (int i=0; i<oa.length; i++)
    if (oa[i] instanceof Cloneable) {
      try {
        tmp[i] = oa[i].getClass()
                 .getMethod("clone", null)
                 .invoke(oa[i], null);
      } catch (Exception e) {
        throw new CloneNotSupportedException();
      }
    }
 ...
```

Auf die Details von Reflection wollen wir an dieser Stelle nicht weiter eingehen. Es sei aber angemerkt, dass Methoden-Aufrufe über Reflection nicht nur umständlicher und wesentlich unleserlicher sind als normale Aufrufe, sie sind auch deutlich aufwendiger und langsamer. Laufzeit-Unterschiede in der Größenordnung von 1:100 (je nach Virtueller Maschine und System-Kontext) sind nicht unrealistisch. Außerdem können bei Benutzung von Reflection zur Laufzeit wesentlich mehr Fehler auftreten als beim normalen statischen Aufruf. Beispiel: wenn man sich beim Methoden-Namen vertippt hat, dann merkt das normalerweise

der Compiler; beim Aufruf der Methode über Reflection wird dieser Fehler erst beim Programmablauf bemerkt und führt zu einer Exception, auf die das Programm sinnvoll reagieren muss. Die Nachteile des Methoden-Aufrufs über Reflection sind verglichen mit dem statischen Methodenaufruf gravierend. Man wird deshalb normalerweise immer den statischen Aufruf vorziehen. Beim Klonen von generischen Collections kann man die Nachteile durch die Reflection-Nutzung aber leider nicht vermeiden, weil wegen dem leeren Cloneable-Interface der statische Aufruf überhaupt nicht möglich ist.

Non-Cloneable Objekte in Generischen Collections

Beim Kopieren von generischen Collections hat man nicht nur Probleme mit dem leeren Cloneable-Interface, sondern der Container könnte auch Referenzen auf Objekte enthalten, die tatsächlich gar nicht cloneable sind. Da hilft dann auch Reflection nichts mehr und man muss zu anderen Kopier-Techniken greifen. Dazu muss man aber sämtliche non-cloneable Typen per Fallunterscheidung identifizieren und die für den jeweiligen Typ passende Kopiertechnik kennen. Hier sind ein paar Beispiele:

```
for (int i=0; i<oa.length; i++)
    if (oa[i] instanceof Cloneable) {
      ... call clone via reflection ...
    }
    else if (oa[i] instanceof StringBuffer)
      tmp[i] = new StringBuffer (oa[i].toString());

    else if (oa[i] instanceof String)
      tmp[i] = oa[i];
    ...
    // all other types
    ...
    else throw new CloneNotSupportedException();
```

Diese Fallunterscheidung ist natürlich ein Albtraum was Erweiterbarkeit und Pflege der Software angeht. Deshalb ist anzuraten, dass alle Klassen, die Value-Typen (siehe unten) repräsentieren, clone() unterstützen sollten, auch wenn sie alternative Kopiertechniken, z.B. per Copy-Konstruktor, anbieten. Diese Erkenntnis hat sich in der Java-Community erst langsam durchgesetzt, wie man an der Geschichte des JDK sehen kann. In frühen Versionen des JDK (1.0 und 1.1) waren viele Klasse nicht cloneable, die man später cloneable gemacht hat; ein Beispiel ist die Klasse java.util.Date. Offenbar hat es sich als reales Problem herausgestellt, wenn jede Klasse ihre eigene Technik für das Erzeugen von Kopien entwickelt.

Die Unterscheidung zwischen Value- und Entity-Typen hatten wir bereits in einem der vorangegangenen Artikel erläutert (siehe / KRE /), als wir überlegt haben, welche Klassen equals() implementieren müssen (Value-Typen) und für welche Typen das nicht nötig ist (Entity-Typen). equals(), hashCode(), compareTo() und auch clone() sind Methoden, die nur für Value-Typen von Bedeutung sind, weil sich die Semantik dieser Methoden um den Inhalt des Objekts dreht. Bei Entity-Typen ist der Inhalt des Objekt nicht von so herausragender Bedeutung, so dass Entity-Typen meistens keine dieser Methoden implementieren (oder nur eine sehr simple auf der Adresse des Objekts basierende Implementierung haben).

Wenn man sich den clone()-Contract ansieht, kann man auch sehen, warum Entity-Typen keine clone()-Methode haben. Der clone()-Contract verlangt, dass x.clone() != x ist und dass x.clone().equals(x) ist, d.h. Klon und Original müssen verschiedene Objekte mit gleichem Inhalt sein. Nun ist es aber für Entity-Typen so, dass der ==-Operator und die equals()-Methode dieselbe Semantik haben: beide prüfen auf Identität der zu vergleichenden Objekte. Das liegt daran, dass für Entity-Typen die equals()-Methode nicht implementiert wird; dann gibt es nur die von Object geerbte equals()-Methode und die vergleicht die Adressen der Objekte, genau wie das der ==-Operator macht. Unter diesen Umständen kann ein Entity-Typ keine clone()-Methode haben, die dem clone()-Contract genügt: wenn Klon und Original verschiedene Objekte sind (d..h. x.clone() != x), dann liefert x.clone().equals(x) das Ergebnis false und der equals()-Contract wäre verletzt.

Das Klonen von unveränderlichen Objekten

Grundsätzlich sollte man clone() implementieren für alle Klassen mit Value-Semantik, es sei denn, es gibt gute Gründe, es nicht zu tun. Ein guter Grund liegt vor, wenn die Klasse unveränderliche (immutable) Objekte beschreibt, also keine modifizierenden Methoden anbietet. Objekte eines solchen Typs können niemals verändert werden. Man kann argumentieren, dass unveränderliche Objekte niemals kopiert werden müssen, weil man problemlos Referenzen darauf halten kann und das resultierende Object-Sharing bei unveränderlichen Objekten einfach kein Problem ist.

Dieser Logik folgend müssten dann alle veränderlichen Value-Typen cloneable sein und alle unveränderlichen non-cloneable. Leider ist das in der Praxis nicht so. Man kann sich keineswegs darauf verlassen, dass eine non-cloneable Klasse genau deshalb kein clone() hat, weil man keine Kopien braucht und die Objekte problemlos gemeinsam referenzieren kann. Die Klasse java.lang.String beispielsweise folgt dieser Regel; sie ist unveränderlich und non-cloneable. Aber bei der Klasse java.lang.StringBuffer stimmt es schon nicht mehr; sie ist non-cloneable, aber trotzdem veränderlich und keineswegs problemlos beim Object-Sharing. Aus der Tatsache, dass eine Klasse nicht cloneable ist, kann man daher nicht ableiten, dass keine Kopien von Instanzen dieser Klasse gebraucht werden. Der umgekehrte Schluss ist auch nicht möglich: aus der Tatsache, dass eine Klasse cloneable ist, kann man nicht ableiten, dass Kopien gebraucht werden.

Für eigene Klassen ist es empfehlenswert, sich eine klare Strategie zu überlegen, nämlich die 1:1-Beziehung zwischen "Für Instanzen dieser Klasse ist das Object-Sharing problematisch." und "Die Klasse ist cloneable." Dann kommt man automatisch dazu, dass alle veränderlichen Value-Typen cloneable sind und alle unveränderlichen Value-Typen non-cloneable sind und alle Entity-Typen ebenfalls non-cloneable sind.

Zusammenfassung und Ausblick

In diesem Artikel haben wir uns angesehen, warum das Kopieren in Java überhaupt eine Rolle spielt. Wir haben verschiedene Techniken dafür gesehen (im wesentlichen Klonen und Copy-Konstruktion) und festgestellt, dass es empfehlenswert ist, zumindest für veränderliche Value-

Typen die clone()-Methode zu implementieren. Wir haben die Anforderung an clone() (den sogenannten clone()-Contract) gesehen und uns überlegt, wie tief eine Kopie sinnvollerweise sein sollte. Und schließlich haben wir uns mit einigen Eigenarten des leeren Cloneable-Interface befasst.

Worauf man achten muss, wenn man clone()implementiert, werden wir in der nächsten Ausgabe der Kolumne untersuchen (siehe / CLON /). Dabei wird u.a. die besondere Rolle von Object.clone() deutlich werden, die wir bislang kaum erwähnt haben. Wir werden sehen, wo die Copy-Konstruktion als Alternative zum Klonen ihre Grenzen hat. Im übernächsten Artikel werden wir dann noch die CloneNotSupportedException diskutieren.

Das Kopieren von Objekten in Java

Teil 2: Wie implementiert man die clone() Methode?

JavaSPEKTRUM, November 2002
Klaus Kreft & Angelika Langer

Im letzten Artikel / KRE1 / dieser Serie haben wir uns angesehen, warum das Kopieren in Java überhaupt eine Rolle spielt. Wir haben verschiedene Kopier-Techniken gesehen (im wesentlichen Klonen und Copy-Konstruktion) und festgestellt, dass es empfehlenswert ist, zumindest für veränderliche Value-Typen die clone()-Methode immer zu implementieren. Wir haben die Anforderung an clone() (den sogenannten clone()-Contract) gesehen und uns überlegt, wie tief eine Kopie sinnvollerweise sein sollte.

Worauf man achten muss, wenn man clone()implementiert, werden wir in der dieser Ausgabe der Kolumne untersuchen.
Insbesondere in Klassenhierarchien ergeben sich weitere Anforderungen und Komplikationen, die wir uns im Detail ansehen werden. Dabei werden wir die besondere Rolle untersuchen, die Object.clone() bei der Implementierung von clone() spielt. Wir werden außerdem sehen, wo die Copy-Konstruktion als Alternative zum Klonen ihre Grenzen hat. Und schließlich werden wir noch zeigen, dass final-Felder beim Klonen besondere Probleme bereiten.

Prinzipien der Implementierung von clone()

Versuchen wir uns an einer Implementierung von clone(). Beginnen wir mit der Signatur.

Die Signatur von clone() und die CloneNotSupportedException

Ein Klasse, die die clone()-Methode implementieren will, muss normalerweise das Cloneable-Interface implementieren. Das Cloneable-Interface ist ein leeres Marker-Interface, das dazu verwendet wird, um klonbare von nicht-klonbaren Objekten zu unterscheiden. Da das Cloneable-Interface leer ist, gibt es keine zwingende Vorschrift, was die Signatur der clone()-Methode einer Klasse angeht. Aus diesem Grunde verwenden viele Programmierer die Signatur von Object.clone(), nämlich

```
Object clone() throws CloneNotSupportedException;
```

Den Return- und den Argumenttyp betreffend gibt es keine Diskussionen. Man will die protected Version der Superklasse Object mit einer public Version überschreiben, also müssen Returntyp und Argumentenliste exakt dieselben sein wie in Object.clone(). Aber über die Exception-Liste kann man geteilter Meinung sein.

Es ist ein offensichtlicher Widerspruch, in einer Klasse eine public-Methode clone() zu implementieren, gerade mit dem Ziel, das Klonen zu unterstützen, und dann gleichzeitig zu sagen: diese Klasse unterstützt das Klonen eigentlich gar nicht und wird unter Umständen eine CloneNotSupportedException werfen. Das ist unlogisch und aus diesem Grunde deklariert man

die clone()-Methode typischerweise als Methode, die keine checked Exceptions wirft (und damit insbesondere keine CloneNotSupportedException)[1].

In der Praxis findet man trotz des offensichtlichen Widerspruchs Implementierungen von clone(), die zwar deklarieren, dass sie eine CloneNotSupportedException werfen könnten, aber dann niemals eine solche werfen. Wegen dieser verwirrenden Situation werden wir den nächsten Beitrag in dieser Kolumne ausschließlich der CloneNotSupportedException widmen. In diesem Artikel wollen wir clone() implementieren und wir werden es aus oben geschilderten Gründen ohne throws-Klausel deklarieren und auch so implementieren.

Die Funktionalität von clone()

Die Anforderungen an die Funktionalität von clone() haben wir uns im letzten Artikel bereits angesehen. Im Wesentlichen bestehen sie darin, dass clone() ein neues Objekt anlegen muss, das inhaltlich gleich dem Original, aber vom Original unabhängig ist.

Dabei bedeutet "unabhängig", dass die Kopie so tief sein muss, dass jedwede Manipulation des Klons das Original nicht betrifft und umgekehrt. Das läuft darauf hinaus, dass alle Referenzen verfolgt und die referenzierten Objekte kopiert werden müssen, es sei denn, sie sind unveränderlich. Im Falle von unveränderlichen Feldern reicht es, nur die Referenzen, nicht aber die referenzierten Objekte zu kopieren.

Die geforderte "inhaltliche Gleichheit" bedeutet "Gleichheit im Sinne von equals()", d.h. der Vergleich von Klon und Original mit Hilfe von equals()muss true liefern.

Um die geforderte Funktionalität zu liefern, müssen folgende Aufgaben von der Implementierung einer clone()-Methode erledigt werden:

- Die Methode muss Speicher beschaffen für den Klon.
- Sie muss alle relevanten Felder in ausreichender Tiefe kopieren.

Speicherbeschaffung

Für die Speicherbeschaffung ist Object.clone() zuständig. Deshalb muss jede Implementierung von clone() als erstes super.clone() aufrufen, damit rekursiv am Ende Object.clone() angestoßen wird. Object.clone() beschafft den Speicher fürs gesamte Objekt (abhängig vom Laufzeittyp) und füllt diesen Speicher mit einer bitweise Kopie des Originals. Die Details der Speicherbeschaffung und -initialisierung sehen wir uns später noch an.

Wir haben im letzten Artikel schon festgestellt, dass diese bitweise Kopie nur in wenigsten Fällen eine ausreichend tiefe Kopie ist. Immer wenn die Klasse Felder hat, die Referenzen auf modifizierbare Objekte sind, dann müssen sich die clone()-Methoden der Subklassen anschließend noch um das Herstellen der hinreichend tiefen Kopie kümmern.

---

[1] Zu den Vor- und Nachteilen von unchecked und checked Exceptions und deren Verwendung aus Design-Sicht findet man interessante Überlegungen in Buch von Barbara Liskov (siehe / LIS /).

Kopieren der relevanten Felder

Jede clone()-Methode muss also super.clone() aufrufen und sich danach um die Felder der eigenen Klasse kümmern. Für die Felder von primitiven Typ ist nichts zu tun, weil Object.clone() diese Felder bereits mit einer Kopie der Felder des Originals gefüllt hat, was für primitive Felder ausreichend ist. Bei den Feldern, die Referenzen sind, können diejenigen ignoriert werden, die auf unveränderliche Objekte zeigen. Sie sind bereits durch Object.clone() in ausreichender Tiefe kopiert worden, indem die Adresse kopiert wurde. Dadurch kommt es zwar zu einem Object-Sharing zwischen Klon und Original, weil beide auf dasselbe unveränderliche Objekt verweisen. Aber das ist unproblematisch, weil das gemeinsam referenzierte Objekt sich niemals ändern kann. Es bleiben also nur noch die Felder übrig, die auf veränderliche Objekte verweisen. Diese Felder müssen nach dem Aufruf von super.clone() in ausreichender Tiefe kopiert werden.

Die besonderen Eigenschaften von Object.clone()

Schauen wir uns die Speicherbeschaffung durch Object.clone() noch einmal an. Kann man den Speicher nicht auch einfach per new beschaffen statt Object.clone() zu rufen? Das funktioniert in der Tat bei Klassen, die als final deklariert sind, macht aber bei einer non-final Klasse keinen Sinn. Sehen wir uns das fehlerhafte Beispiel einer non-final Klasse an:

```
class MyClass {
   private int aField;
   ...
   public Object clone() {
     MyClass tmp = new MyClass();
     tmp.aField = aField;
     ... copy all remaining fields to tmp ...
     return tmp;
   }
}
```

Soweit ist das noch in Ordnung, aber sobald eine Subklasse abgeleitet wird, gibt es Probleme:

```
class MySubClass extends MyClass {
   private Object anotherField;
   ...
   public Object clone() {
     MySubClass tmp = ... allocate memory ...
     ... copy all fields to tmp ...
     return tmp;
   }
}
```

Diese Subklasse muss Speicher für den Klon beschaffen und dann noch alle Felder kopiere, um den Klon zur Kopie des Originals zu machen. Dazu müssen nicht nur die eigenen Felder kopiert werden, sondern auch die von der Superklasse geerbten Felder. Auf die geerbten Felder, die in der Superklasse als private deklariert sind, hat die Subklasse keinen Zugriff, deshalb muss sie an die entsprechende Methode der Superklasse delegieren. Man würde also super.clone() rufen wollen, insbesondere da super.clone() sowieso Speicher beschafft. Hier ist der Versuch eines Aufrufs von super.clone():

```
class MySubClass extends MyClass {
   private Object anotherField;
   ...
   public Object clone() {
     MySubClass tmp = (MySubClass) super.clone();
     ... copy all fields to tmp ...
     return tmp;
   }
}
```

Diese Implementierung wird natürlich eine ClassCastException auslösen, weil super.clone() zwar eine Object-Referenz zurück gibt, die aber auf ein Objekt vom Supertyp MyClass verweist. Die Methode MyClass.clone() beschafft zwar Speicher, aber nicht genug; sie erzeugt ein kleineres Objekt, nämlich ein Superklassen-Objekt. Was wir aber brauchen, ist die Beschaffung von Speicher für das größere Subklassen-Objekt. Das kann die Superklassen-Methode MyClass.clone() aber nicht leisten, weil die Superklasse keine Kenntnis vom der abgeleiteten Subklasse hat.

Die Methode Object.clone() hingegen hat die besondere Eigenschaft, dass sie Speicher in der richtigen Menge abhängig vom Laufzeittyp des Objekts beschafft und das ganze Objekt bereits als bitweise Kopie des Originals füllt. Das Ergebnis von Object.clone() ist ein Objekt vom richtigen Typ in der richtigen Größe mit teilweise nützlichem Inhalt. Weil Object.clone() diese besondere Funktionalität hat, sollte man dafür sorgen, dass diese Methode auch aufgerufen wird, damit sie ihre Aufgaben übernehmen kann. Man kann sich sicher auch alternative Implementierungen von clone() überlegen, die ohne Object.clone() auskommen und Speicher mit new beschaffen, vielleicht unter Verwendung von Reflection, und irgendwelche Füll-Methoden aus den Superklassen aufrufen. Verglichen mit dem eleganten Mechanismus über Object.clone() dürfte das allerdings eher ineffizient und umständlich zu warten sein.

Die Regel ist daher: jede potentielle Superklasse, d.h. jede non-final Klasse, muss in ihrer Implementierung von clone() dafür sorgen, dass Speicher für das gesamte Objekt, das von this referenziert wird, beschafft wird, also nicht nur für ein Teil-Objekt vom eigenen Typ. Und das ist in effizienter Weise nur mit Hilfe von Object.clone() möglich. Deshalb sollte jede Implementierung von clone() als erstes super.clone() aufrufen.

Die non-final Klasse aus unserem obigen Beispiel darf also nicht so aussehen:

```
class MyClass {
   private int aField;
   ...
   public Object clone() {
     MyClass tmp = new MyClass();
     tmp.aField = aField;
     ... copy all remaining fields to tmp ...
     return tmp;
   }
}
```

Sondern sie sollte so aussehen:

```
class MyClass {
   private int aField;
   ...
   public Object clone() {
     try {  MyClass tmp = (MyClass) super.clone(); } catch
(CloneNotSupportedException e) {}
     tmp.aField = aField;
     ... copy all remaining fields to tmp ...
     return tmp;
   }
}
```

Mit dieser Implementierung von clone() lässt sich dann auch die Subklasse übersetzen, weil super.clone() in diesem Falle tatsächlich ein Objekt vom Subklassen-Typ liefert:

```
class MySubClass extends MyClass {
   private Object anotherField;
   ...
   public Object clone() {
     MySubClass tmp = (MySubClass) super.clone();
     ... copy all fields to tmp ...
     return tmp;
   }
}
```

Wie man an diesem Beispiel wieder einmal sieht, fällt potentiellen Superklassen, d.h. non-final Klassen, die Verantwortung für die gesamte Subhierarchie zu. Wenn in einer non-final Klasse clone() falsch implementiert ist, dann haben die Subklassen praktisch keine Chance mehr, eine korrekte Implementierung von clone() zu liefern. In unserem Beispiel hatte die Superklasse den Speicher für den Klon falsch beschafft, nämlich über new und nicht über Object.clone(). Das ist ein Fehler und führt zu Problemen in den Subklassen-Implementierungen von clone(). Aber es könnte ja auch noch schlimmer sein: was passiert, wenn eine potentielle Superklasse nicht cloneable ist und die Methode clone() überhaupt nicht implementiert?

Non-Cloneable Superklassen

Wenn eine non-final Klasse clone() nicht implementiert, dann ist das u.U. eine gravierende Einschränkung für die Subklassen. Betrachten wir ein Beispiel aus dem JDK: dort gibt es die Klasse java.util.Observable. Bei einem Observable-Objekt können sich Observer-Objekte registrieren lassen, damit sie später unter bestimmten Umständen notifiziert werden. Ein Observable-Objekt enthält zum Zwecke der Notifizierung ein Array von Observer-Objekten.

Die Klasse Observable ist explizit als Superklasse entworfen, aber sie ist nicht cloneable und sie hat auch keine Implementierung der clone()-Methode. Observable ist also eine non-cloneable Superklasse. Stellen wir uns nun vor, wir wollen von Observable ableiten und die Subklasse soll cloneable sein.

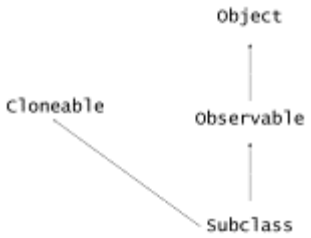


Abbildung 1: Vererbungsbeziehungen für eine cloneable and observable Klasse

Eine Implementierung einer solchen Subklasse könnte wie folgt aussehen:

```
class Subclass extends java.util.Observable implements Cloneable {
  private Object aField;

  public Object clone() {
    try {
      Subclass tmp = (Subclass) super.clone();
      tmp.aField = aField.clone();
    } catch (CloneNotSupportedException) { ... }
  }
}
```

Es wird korrekterweise super.clone() gerufen, um den Speicher zu beschaffen und die Superklassen-Anteile in angemessener Tiefe zu kopieren. Da die Superklasse Observable aber keine Implementierung von clone() hat, ist super.clone() direkt Object.clone(). Es wird nun zwar Speicher für ein Subclass-Objekt beschafft und dieser Speicher wird mit einer bit-weisen Kopie des Original-Subclass-Objekts gefüllt, aber das reicht nicht aus. Die Superklasse Observable enthält ein privates Array von Observer-Objekten und dieses Array wird natürlich nicht kopiert. Die bit-weise Kopie kopiert lediglich die Adresse des Arrays, so das Original und Klon auf dasselbe Array verweisen. Das ist keine genügend tiefe Kopie und die Subklasse hat auch keine Chance, eine genügend tiefe Kopie zu erzeugen, weil sie keinen Zugriff auf das private Observer-Array der Superklasse Observable hat.

Das Beispiel zeigt, dass eine Superklasse, die keine clone()-Methode anbietet, ihre Subklassen gravierend einschränkt: die Subklassen können dann u.U. ebenfalls keine clone()-Methode implementieren.

Unter gewissen Umständen muss es nicht zu Problemen führen, wenn eine Superklasse keine clone()-Methode hat. Wenn etwa alle Felder der Superklasse von primitivem Typ sind, dann ist die von Object.clone() erzeugte bit-weise Kopie bereits genügend tief. Das gleiche gilt, wenn die Superklasse nur Felder hat, die auf unveränderliche Objekte verweisen. Aber in allen anderen Fällen wirkt sich das Fehlen einer clone()-Methode auf die Subklasse aus: sie kann dann auch nicht cloneable sein.

Fazit: Alle non-final Klassen, die Value-Type[2] repräsentieren und nicht-statische Felder haben, welche Referenzen auf veränderliche Objekte oder Arrays von veränderlichen Objekte sind, sollten eine Implementierung von clone() haben, welche super.clone() ruft und alle ihre Felder in genügender Tiefe kopiert. Diese Implementierung von clone() muss nicht einmal als public Methode angeboten werden und die non-final Klasse muss auch nicht cloneable sein. Es genügt, wenn eine korrekte Implementierung von clone() als protected Methode zur Verfügung steht, damit Subklassen ihrerseits cloneable werden können und ihre clone()-Methode mit Hilfe der protected clone()-Methode der Superklasse implementieren können.

Verwendung von non-final Methoden in clone()

Noch ein Hinweis zur Implementierung von clone() in non-final Klassen: die clone()-Methode sollte keine non-final Methoden rufen. Solche Methoden könnten in Subklassen redefiniert werden und auf subklassen-spezifische Felder zugreifen, die zum Zeitpunkt des Aufrufs noch gar keine sinnvollen Wert haben.

Betrachten wir ein Beispiel: eine non-final Klasse, die in ihrer Implementierung von clone() eine non-final Methode sanityCheck() ruft. Die Methode sanityCheck() prüft die logische Konsistenz des Objekts und wirft im Fehlerfall die unchecked Exception IllegalStateException, die einfach von clone() einfach durchgelassen wird; aber das ist hier ohne Bedeutung.

```
class SuperClass implements Cloneable {
  ... private fields ...

  protected void sanityCheck() throws IllegalStateException { ... }

  public Object clone() {
    try { SuperClass tmp = (SuperClass) super.clone(); } catch
(CloneNotSupportedException e) {}
    tmp.sanityCheck();
    ... create deep-enough copies of fields ...
    return tmp;
  }
}
```

Ein Subklasse dieser Klasse wird unter Umständen die non-final Methode sanityCheck() überschreiben und kann dabei auf eigene Felder zugreifen:

```
class SubClass extend SuperClass {
  private Object aReference;

  ... constructors and stuff ...
```

[2] Die Unterscheidung zwischen Value- und Entity-Typen haben wir bereits in einem vorangegangenen Artikel beschrieben (siehe /KRE2/), als wir überlegt haben, welche Klassen equals() implementieren müssen (Value-Typen) und für welche Typen das nicht nötig ist (Entity-Typen). equals(), hashCode(), compareTo() und auch clone() sind Methoden, die nur für Value-Typen von Bedeutung sind, weil sich die Semantik dieser Methoden um den Inhalt des Objekts dreht. Bei Entity-Typen ist der Inhalt des Objekt nicht von so herausragender Bedeutung, so dass Entity-Typen meistens keine dieser Methoden implementieren.

```
  protected void sanityCheck() throws IllegalStateException {
    ... access fields of aReference...
  }
  public Object clone() {
    SubClass tmp = (SubClass) super.clone();
    tmp.sanityCheck();
    tmp.aReference = aReference.clone();
  }
}
```

Wenn nun SubClass.clone() gerufen wird, dann wird zuerst super.clone(), d.h. SuperClass.clone(), gerufen, welches tmp.sanityCheck() anstößt. Das Objekt tmp wurde von Object.clone() erzeugt und ist korrekterweise ein SubClass-Objekt. Es wird daher die Methode SubClass.sanityCheck() gerufen. Diese Methode greift auf Felder zu, die per aReference zu erreichen sind. Allerdings hat aReference noch nicht seinen endgültigen Wert zugewiesen bekommen; das geschieht erst im nachfolgenden Statement. Es ist zu vermuten, dass die Methode sanityCheck() daher ein fehlerhaftes Ergebnis liefern wird.

Ganz allgemein kann es vorkommen, dass non-final Methoden, die in clone() gerufen werden, den Klon in halbfertigem Zustand antreffen, weil subklassen-spezifische Felder noch nicht ihren endgültigen Wert haben. Das ist ein Problem, welches auch bei Konstruktoren auftritt (und das wir hier nicht weiter vertiefen wollen). Bei Konstruktoren gibt es die Empfehlung, den Aufruf von non-final Methoden zu vermeiden, und das gleiche gilt auch hier.

Fazit: In der Implementierung von clone() vermeide man den Aufruf von non-final Methoden auf dem halbfertigen Klon.

Copy-Konstruktion vs. clone()

Im letzten Artikel dieser Kolumne hatten wir neben clone() andere Methoden zur Erzeugung von Kopien von Objekten in Java erwähnt und gesagt, dass die clone())-Methode die empfohlene Technik sei. Sehen wir uns das noch einmal genauer an.

Eine der populärsten dieser alternativen Kopier-Mechanismen ist die Copy-Konstruktion. Klassen mit Copy-Konstruktor haben eine Konstruktor, der ein Objekt vom eigenen Typ als Argument akzeptiert und ein neues Objekt mit gleichem Inhalt erzeugt. Das sieht auf den ersten Blick genauso aus wie die Funktionalität von clone(). Welchen Nachteil hat das gegenüber clone()? Hier ist eine solche Klasse Person, die nicht cloneable ist, aber einen Copy.-Konstruktor hat:

```
class Person {
  private String name;
  private Date birthday;
  public Person(Person other)
  { name = other.name;
    birthday = (Date)other.birthday.clone();
  }
}
```

Zunächst einmal ist das völlig in Ordnung. Die Probleme tauchen auf, wenn Subklassen von Person ins Spiel kommen. Betrachten wir eine Subklasse Employee:

```
class Employee extends Person {
  private float salary;
  public Employee(Person p, float s)
  { super(p); salary = s; }
  public Employee(Employee other)
  { super(other); salary = other.salary; }
  ...
}
```

Nehmen wir außerdem an, es gibt eine Methode copy(), welche Person-Objekte kopiert:

```
Person copy(Person p)
{ return new Person(p); }
```

Kein Problem, solange nur Person-Objekte kopiert werden. Die copy()-Methode kann aber auch mit einem Employee als Argument aufgerufen werden. Dann würde man erwarten, dass als Ergebnis eine Person-Referenz auf ein Employee-Objekt zurück kommt. Das ist aber nicht der Fall: es wird nur eine Kopie des Person-Anteils des Employee-Objekts geliefert. Diesen Vorgang der Verstümmelung bezeichnet man als "object slicing" und i.a. allgemeinen ist das kein erwünschter Effekt. Man erwartet statt dessen, dass copy() ein Objekt von dem Typ zurück liefert, der auch hineingesteckt wurde.

Das Problem des Object-Slicing ließe sich vermeiden, wenn die copy()-Methode abhängig vom Typ des Arguments unterschiedliche Konstruktoren aufrufen würde. Das könnte dann so aussehen:

```
Person copy(Person p)
{
  if (p.getClass() == Person.class)
     return new Person(p);
  if (p.getClass() == Employee.class)
     return new Employee(p) ;
  ...
}
```

Allerdings ist das der klassische Wartungsalbtraum: jedes Mal, wenn eine neuen Subklasse entsteht, muss die copy()-Methode um einen weiteren if-Zweig erweitert werden. So sollte man es auf gar keinen Fall machen. Für typ-abhängige Methodenaufrufe gibt es in der Objekt-Orientierung den Mechanismus des Polymorphismus. Das Problem ließe sich elegant lösen, indem statt der Kopie per Copy-Konstruktion einen Klon per clone()-Methode erzeugen würde. Eine bessere Implementierung würde also so aussehen:

```
Person copy(Person p)
{ return p.clone(); }
```

Die clone()-Methode ist eine polymorphe Methode, die zur Laufzeit abhängig vom Typ des Objekts, auf dem sie gerufen wird, ausgewählt wird. Wenn also ein Employee als Argument

übergeben wird, dann wird automatisch Employee.clone() gerufen, und es wird ein Employee -Objekt und nicht nur ein Person-Objekt erzeugt. Was man für die Implementierung von Methoden wie copy() braucht, ist eine polymorphe Kopier-Methode, und genau das kann ein Konstruktor nicht leisten. Konstruktoren sind grundsätzlich nicht-polymorphe Methoden. Das polymorphe Kopieren ist nur über clone() zu erreichen. Aus diesem Grunde ist clone() die bevorzugte Technik für das Kopieren von Objekten[3].

Außerdem hat die gezeigte Person-Klasse mit ihrem Copy-Konstruktor die oben schon am Beispiel von Observable diskutierten Schwächen einer non-final Klasse, die nicht cloneable ist und damit ihre Subklassen einschränkt: was auch immer man versucht, es gelingt nicht, die non-final Klasse Employee mit einer korrekten clone()-Methode auszustatten.

clone() und final Felder

Es gibt noch ein Problem bei der Implementierung von clone(): Klassen, die Felder haben, die als final deklariert sind, machen Schwierigkeiten. Betrachten wir das Beispiel einer Klasse, die ein Feld vom Typ java.util.Date hat, welches ein Zeitstempel enthält. Der Stempel markiert den Zeitpunkt der letzten Veränderung des Objekts und wird in jeder modifizierenden Methode neu gesetzt. Aus Optimierungsgründen soll beim Update des Zeitstempels kein neues Date-Objekt erzeugt werden, sondern es soll das vorhandene Date-Objekt geändert werden, um den neuen Zeitstempel abzulegen. Diese Nutzungsweise des Date-Feldes kann in Java sichergestellt werden, indem das Feld als final deklariert wird.

Felder oder Variablen, die als final deklariert sind, können nicht verändert werden. Das bedeutet bei Variablen von primitivem Typ, dass sich der Wert der Variablen niemals ändert. Bei Referenzvariablen bedeutet es, dass sich die Referenz niemals ändern kann, d.h. die final Referenzvariable wird immer auf dasselbe Objekt verweisen. Es bedeutet aber nicht, dass das referenzierte Objekt nicht geändert werden kann.

In unserem Beispiel haben wir ein Feld, dass auf ein Date-Objekt verweist. Das Date-Objekt, d.h. der Zeitstempel, kann geändert werden, aber die Referenz selbst soll aus den oben geschilderten Optimierungsgründen bestehen bleiben. In einer solchen Situation ist sinnvoll und korrekt, das Referenzfeld als final zu deklarieren, um die gemachte Designentscheidung in der Implementierung zu erzwingen.

Was passiert nun, wenn unsere Klasse mit ihrem Zeitstempel geklont werden soll? Hier ist der Versuch einer Implementierung von clone() für diese Klasse:

```
public class MyClass implements Cloneable {
   final private Date cTime = new Date();
   private Integer cInt;
```

[3] Es gibt eine einzige Situation, in der ein Copy-Konstruktor als Ersatz für eine clone()-Methode vertretbar ist: bei Klassen, die als final deklariert sind. Eine final Klasse kann keine Subklassen haben; ohne Klassenhierarchie spielt Polymorphie keine Rolle und dann ist auch ein Copy-Konstruktor akzeptabel.

```
   public Object clone() {
     try { MyClass tmp = (MyClass)super.clone(); }
     catch (CloneNotSupportedException e) {}
     tmp.cTime = new Date();   // error: cannot assign to final field
     tmp.cInt = new Integer(cInt.intValue());
     return tmp;
   }
}
```

Der Klon soll natürlich seinen eigenen Zeitstempel haben, der zum Zeitpunkt der Erzeugung des Klons mit dem augenblicklichen Zeitpunkt initialisiert wird. Das wird per Zuweisung versucht, aber leider weist der Compiler die Zuweisung als Fehler zurück, weil final Felder nicht verändert werden können. Dieses Problem tritt immer auf, wenn Objekte geklont werden sollen, die final Felder haben. Das Problem liegt darin, dass super.clone() bereits ein fertiges Objekt liefert, in dem die final Felder bereits als bit-weise Kopie der entsprechenden Felder des Originals initialisiert sind. final Felder können daher nur den Inhalt haben, den super.clone() ihnen gibt; spätere Änderungen, wie hier die Zuweisung einer Kopie des Date-Objekts, sind nicht möglich.

Für das Problem gibt es nur eine Reihe von denkbaren Lösungen, die aber alle unbefriedigend sind:

1. Man verzichtet auf die Deklaration von final Felder in Klassen, die cloneable sein sollen. Damit verzichtet man darauf, seine Designentscheidungen klar und deutlich mit Mitteln der Sprache auszudrücken und man macht außerdem etwaige Optimierungen, die der Compiler für final Felder machen könnte, unmöglich.

2. Man umgeht die final Deklaration mithilfe von Reflection (siehe java.lang.reflect.Field.setAccessible()). Das funktioniert aber nur so lange, wie kein Security Manager aktiviert ist, den diesen Hack verhindert. [1]

3. Man greift auf Konstruktoren zurück, um clone() zu implementieren. Damit handelt man sich aber die oben diskutierten Object-Slicing-Probleme ein. Eine solche Lösung führt immer dazu, dass die Klasse oder zumindest die clone()-Methode final sein muss.

Eine Lösung per Konstruktor könnte so aussehen:

```
public class MyClass implements Cloneable {
   final private Date cTime;
   private Integer cInt;

   private MyClass(MyClass m)
   { cTime = new Date(); }

   final public Object clone() {
     MyClass tmp = new MyClass(this);
     tmp.cInt = new Integer(cInt.intValue());
     return tmp;
   }
}
```

In dieser Lösung verwenden wir ein sogenanntes "blank final" Feld, ein Sprachmittel, dass in den ersten Versionen der Sprache noch gar nicht existierte und erst später (in Version 1.1) eingeführt wurde. Normale final Felder müssen bereits bei der Definition mit ihrem unveränderlichen Wert versorgt werden. Für solche Felder verlangt der Compiler, dass bereits in der Definition auch der Wert spezifiziert wird, so wie wir das in unserer ersten versuchten Implementierung gemacht hatten. Für blank final Felder ist das etwas anders: sie müssen nicht schon in der Definition mit ihrem Wert versorgt werden, sondern das kann im Konstruktor geschehen. Das hat den Vorteil, dass in verschiedenen Konstruktoren verschiedene Werte zugewiesen werden können. Das war mit den normalen final Feldern nicht möglich; deshalb wurden die blank final Felder erfunden.

Konstruktor gesetzt wird. In diesem Konstruktor haben wir die Chance das Feld so zu setzen, wie es benötigt wird, nämlich so, dass es auf ein neues Date-Objekt verweist. Diesen privaten Konstruktor verwenden wir in der Implementierung von clone(), damit der Klon seinen eigenen Zeitstempel bekommt. Der unangenehme Nebeneffekt ist, dass nun der Speicher für den Klon per new und nicht mit Hilfe von super.clone() beschafft wird, was in Subklassen zu den schon beschriebenen Problemen führt. Deshalb haben wir die clone()-Methode als final deklariert.

Das geschilderte Problem tritt im Allgemeinen nur bei final Feldern auf, die Referenzen auf veränderliche Objekte sind. Bei final Feldern von primitiven Typ und bei final Feldern, die Referenzen auf unveränderliche Objekte sind, ist die von Object.clone() erzeugte bit-weise häufig Kopie bereits ausreichend und eine spätere Zuweisung eines anderen Werts in der clone()-Methode ist nicht nötig. Damit tritt auch das diskutierte Problem nicht auf.

Wenn das Problem aber auftritt, dann muss man sich zwischen den beiden skizzierten Lösungen entscheiden. Beide Lösungen sind unbefriedigend und man würde sich etwas mehr Unterstützung von der Sprache wünschen. Der Compiler könnte clone() als "besondere Methode" behandeln und das Setzen von blank final Feldern in clone() erlauben, so wie es auch in den Konstruktoren erlaubt ist. Aber da das geschilderte Problem in der Praxis nicht extrem häufig auftritt, ist wohl nicht damit zu rechnen, dass sich an der Sprache in dieser Hinsicht etwas ändern wird.

Zusammenfassung

Es gibt drei Gründe, warum Klassen eine korrekte Implementierung von clone() haben sollten:

1. um das polymorphe Kopieren zu ermöglichen. Copy-Konstruktoren leisten dies nicht; die Klasse muss das Cloneable-Interface implementieren und eine public clone()-Methode haben.

2. um cloneable Subklassen zu ermöglichen. Dazu braucht die Klasse nicht selbst cloneable zu sein; es genügt eine protected clone()-Methode.

3. um das Kopieren von generischen Collections zu erleichtern. Das hatten wir im letzen Artikel (siehe / KRE1 /) erwähnt. Für jede non.cloneable Klasse muss eine Sonderlösung gefunden werden; für cloneable Klassen ist das Kopieren wesentlich einfacher.

Die einzigen Klassen, die eine clone()-Methode nur aus Grund [3], also nicht unbedingt, brauchen, sind unveränderlichen Klassen wie z.B. String und Klassen, die Entity-Typen repräsentieren, weil man Instanzen von diesen Klassen im Prinzip überhaupt nicht kopieren muss. Sowohl die clone()-Methode als auch ein Copy-Konstruktor ist für solche Klassen optional.

Für Klassen, die als final deklariert sind, gilt Ähnliches: final Klassen müssen keine Rücksicht auf etwaige Subklassen oder Polymorphie-Anforderungen nehmen. Falls Kopien gebraucht werden, reicht ein Copy-Konstruktor aus. Die clone()-Methode würde nur aus Grund [3] gebraucht, was aber bereits Grund genug ist, um die Klasse cloneable zu machen.

Wir haben uns in diesem Artikel außerdem angesehen, worauf man bei der Implementierung von clone() achten muss:

- Man sollte clone() nicht so deklarieren, dass es eine CloneNotSupportedException wirft. (Dazu mehr im nächsten Artikel).

- Man sollte in non-final Klassen immer super.clone() aufrufen, um den Speicher für den Klon zu beschaffen.

- Man sollte in cloneable Klassen final Felder vermeiden, die Referenzen auf veränderliche Objekte sind.

Das Kopieren von Objekten in Java

Teil 3: Die CloneNotSupportedException - Sollte die clone()-Methode eine CloneNotSupportedException werfen?

JavaSPEKTRUM, Januar 2003
Klaus Kreft & Angelika Langer

In dieser Ausgabe unserer Kolumne wollen wir uns ansehen, ob eine clone()-Methode eine CloneNotSupportedException werfen sollte. Wir haben uns in den vorangegangenen zwei Artikeln (siehe / KRE1 /) bereits ausführlich mit clone() beschäftigt und dabei vorgeschlagen, dass clone() grundsätzlich keine CloneNotSupportedException werfen sollte. Dieses Thema wird aber in der Java-Community durchaus kontrovers diskutiert und wir wollen uns aus diesem Grunde den JDK genauer ansehen, um zu sehen, woher diese kontroverse Diskussion eigentlich stammt.

Die Idee von Cloneable, clone() und der CloneNotSupportedException

Ein Klasse, die die clone()-Methode implementieren will, muss normalerweise das Cloneable-Interface implementieren. Das Cloneable-Interface ist ein leeres Marker-Interface, das dazu verwendet wird, um klonbare von nicht-klonbaren Objekten zu unterscheiden. Da das Cloneable-Interface leer ist, gibt es keine zwingende Vorschrift, was die Signatur der clone()-Methode einer Klasse angeht. Aus diesem Grunde verwenden manche Programmierer für die clone()-Methoden ihrer eigenen Klassen die Signatur von Object.clone(), nämlich

```
Object clone() throws CloneNotSupportedException;
```

Nun ist es ein offensichtlicher Widerspruch, in einer Klasse eine public-Methode clone() zu implementieren, gerade mit dem Ziel, das Klonen zu unterstützen, und dann gleichzeitig zu sagen: diese Klasse unterstützt das Klonen eigentlich gar nicht und wird unter Umständen eine CloneNotSupportedException werfen. Das ist unlogisch und aus diesem Grunde haben wir empfohlen, die clone()-Methode immer als Methode zu deklarieren, die keine CloneNotSupportedException wirft.

Das ist auch im JDK gängige Praxis. Fast alle Klassen des JDK, die das Cloneable-Interface implementieren, haben eine clone()-Methode, die keine CloneNotSupportedException wirft. Die Verwirrung rührt im Wesentlichen von der JavaDoc-Beschreibung der Methode Object.clone() her. Hier ein Auszug aus der Original-Beschreibung:

> *Throws:*
>
> *CloneNotSupportedException - if the object's class does not support the Cloneable interface. Subclasses that override the clone method can also throw this exception to*

indicate that an instance cannot be cloned.
OutOfMemoryError - if there is not enough memory.

Manche Java-Programmierer haben das so verstanden, dass alle clone()-Methoden die CloneNotSupportedException in ihrer throws-Klausel deklarieren sollten, damit Subklassen die Möglichkeit haben, diese Exception zu werfen. Schließlich kann man ja als Autor einer Superklasse nicht wissen, ob Subklassen später überhaupt die clone()-Methode implementieren können oder wollen.

Diese Argumentation ist insoweit richtig, als die Deklaration von clone() ohne throws-Klausel in einer non-final Klasse tatsächlich bedeutet, dass eine Subklasse nicht die Freiheit hat, in ihrer Implementierung von clone() irgendwelche checked Exceptions zu werfen. Diese Einschränkung ist aber eigentlich auch in Ordnung. Bei einem sauberen objekt-orientierten Design repräsentiert die Ableitungsbeziehung zwischen Super- und Subklasse eine sogenannte "is-a"-Beziehung, d.h. ein Objekt der Subklasse ist vom Typ her kompatibel zu einem Superklassenobjekt und kann überall dort verwendet werden, wo ein Objekt der Superklasse verlangt wird. Dieses Prinzip ist als Liskov Substitution Principle (LSP) bekannt (siehe u.a. / LIS /).

Das bedeutet insbesondere, dass die Subklasse sämtliche Operationen unterstützen muss, die die Superklasse unterstützt. Nun kann das Werfen einer CloneNotSupportedException in der clone()-Methode kaum als "Unterstützen der clone()-Operation" bezeichnet werden; es ist vielmehr das Gegenteil. Unter solchen Umständen entstünde eine Klasse, die cloneable wäre (da sie das Cloneable-Interface von der Superklasse erbt) und eine clone()-Methode hätte, aber dann beim Aufruf eine CloneNotSupportedException werfen würde. Das ist gegen jede Intuition und etwa so unlogisch wie eine Klasse, die cloneable ist, aber keine clone()-Methode hat.

Nun kann es aber vorkommen, dass die Subklasse tatsächlich keinen Klon erzeugen kann. Das kann zum Beispiel passieren, wenn kein Speicher mehr vorhanden ist. Was macht man mit solchen oder anderen Fehlersituationen? Das wirft ganz allgemein die Frage auf: wie kann das Scheitern einer clone()-Methode zum Ausdruck gebracht werden, wenn die clone()-Methode so deklariert ist, dass sie keine Exception wirft? Nun, durch eine checked Exception geht es offensichtlich nicht. Das ist aber auch richtig so. Exceptions (sowohl checked als auch unchecked Exceptions) drücken in Java logische Fehler aus, die vorhersehbar und vermeidbar sind, im Gegensatz zu den Errors, die schwere Ausnahmezustände beschreiben, die auf Fehler in der Laufzeitumgebung (Virtuelle Maschine, Garbage Collector, AWT) zurückgehen.

Wenn bereits von der Logik her klar ist, dass in bestimmten Situationen kein Klon erzeugt werden kann, dann sollte die Klasse schon von vornherein das Cloneable-Interface gar nicht implementieren und auch keine clone()-Methode haben. Schließlich ist das Cloneable-Interface für jeden Benutzer der Klasse genau das Kennzeichen, an dem man erkennen kann, dass die Klasse cloneable ist, und dann sollte die Klasse auch nur dann cloneable sein, wenn sich die clone()-Methode sinnvoll implementieren lässt.

Unvorhersehbare Fehler können natürlich trotzdem auftreten. Das sind dann aber schwere Ausnahmefehler, die durch einen Error ausgedrückt werden, und keine "CloneNotSupported"-Situationen. Der Mangel an Speicherplatz ist ein Beispiel; in solchen Fällen wird ein OutOfMemoryError ausgelöst. Andere Fehlersituationen sind eigentlich kaum vorstellbar, wenn sich alle Klassen an die Regel halten, dass sie keine CloneNotSupportedException werfen, wenn sie cloneable sind. Das wird klar, wenn man sich ansieht, was eine kanonische Implementierung von clone() tut: sie ruft die clone()-Methoden für alle Felder und die Superklasse auf. Die einzige clone()-Methode, die eine CloneNotSupportedException werfen könnte, ist Object.clone(). Aber genau das kann nicht eintreten, weil die Klasse cloneable ist.

Fazit: Die clone()-Methode von Klassen, die das Cloneable-Interface implementieren, sollte keine Exception werfen. Alle denkbaren Fehlersituationen sind so schwere Fehler, dass sie angemessen über einen Error ausgedrückt werden.

Sehen wir uns nach diesen Betrachtung jetzt einmal an, wie die Klassen aus dem JDK ihre clone()-Methoden implementieren.

Implementierungen von clone() im JDK

Praktisch alle clone()-Methoden im JDK folgen der oben beschriebenen Regel. Es gibt allerdings einen häufig auftretenden Fehler: viele clone()-Methoden sind in der JavaDoc so beschrieben, dass sie eine CloneNotSupportedException werfen. Wenn man dann die Implementierung dieser Methoden anschaut, stellt man fest: es stimmt gar nicht. Die Methoden haben korrekterweise keine throws-Klausel und werfen auch keine Exceptions. Da passen Implementierung und Dokumentation ganz offensichtlich nicht zusammen.

Das Phänomen erklärt sich dadurch, dass für diese Methoden keine JavaDoc-Kommentare geschrieben wurden. In solchen Fällen benutzt das JavaDoc-Tool automatisch die Beschreibung der Methode der Superklasse. Die Beschreibung aus der Superklasse ist in all diesen Fällen unpassend: es ist nämlich die Beschreibung von Object.clone(). Dieser offensichtliche Dokumentationsfehler sollte uns daran erinnern, dass man ihn für seine eigenen Klassen leicht vermeiden kann, indem man zu jeder Methode, die man implementiert, auch tatsächlich JavaDoc-Kommentare schreibt.

Von diesem Dokumentationsfehler mal abgesehen, sind aber praktisch alle clone()-Methoden so implementiert, dass sie keine Exception, und damit insbesondere keine CloneNotSupportedException, werfen. Die meisten dieser Implementierungen rufen super.clone()auf, und damit letztendlich Object.clone(), und müssen irgendwie mit der CloneNotSupportedException fertig werden, die für Object.clone() deklariert ist, aber gar nicht auftreten kann, weil die Klasse das Cloneable-Interface implementiert. Für den Umgang mit der CloneNotSupportedException, die gar nicht auftreten kann, findet man drei verschiedene Strategien im JDK: "völlig unterdrücken" oder "abbilden auf einen InternalError" oder "null zurückgeben". Sehen wir uns diese drei Strategien einmal anhand von Beispielen näher an.

CloneNotSupportedException unterdrücken

Ein Beispiel für diese Implementierungstechnik findet man zum Beispiel in java.util.Date:

```
public Object clone() {
        Date d = null;
        try {
            d = (Date)super.clone();
            if (d.cal != null) d.cal = (Calendar)d.cal.clone();
        } catch (CloneNotSupportedException e) {} // Won't happen
        return d;
    }
```

Die Klasse Date ist direkt von Object abgeleitet und implementiert das Cloneable-Interface, deshalb kann von super.clone() keine CloneNotSupportedException kommen. Calendar.clone() ist so deklariert, dass es keine Exception wirft; hier kann also auch keine CloneNotSupportedException auftreten. Deshalb wird die CloneNotSupportedException abgefangen und unterdrückt.

CloneNotSupportedException abbilden auf einen InternalError

Ein Beispiel für diese Implementierungstechnik findet man zum Beispiel in java.awt.geom.Point2D:

```
public Object clone() {
 try {
     return super.clone();
 } catch (CloneNotSupportedException e) {
     // this shouldn't happen, since we are Cloneable
     throw new InternalError();
 }
}
```

Das ist im Prinzip die gleiche Situation wie oben bei Date. Hier hat der Autor aber entschieden, dass die CloneNotSupported Exception, nicht völlig unterdrückt werden soll, sondern dass dies ein interner Fehler ist. Ist dieser InternalError gerechtfertigt? Irgendwie schon. Wenn von Object.clone() tatsächlich eine CloneNotSupportedException kommt, was eigentlich nicht sein kann, dann liegt in der Tat in der Laufzeitumgebung ein schweres Problem vor: vielleicht eine inkonsistente oder fehlerhafte virtuelle Maschine oder eine andere kaum vorstellbare Fehlersituation.

CloneNotSupportedException abbilden auf die Rückgabe einer null-Referenz

Das ist eine eher exotische Variante, die wir in der Klasse java.text.Format gefunden haben:

```
public Object clone() {
    try {
      Format other = (Format) super.clone();
      return other;
    } catch (CloneNotSupportedException e) {
      // will never happen
```

```
            return null;
        }
    }
```

So geht es natürlich auch. Hier wird der Returnwert der Methode verwendet, um die Fehlersituation zum Ausdruck zu bringen. Das ist ein schönes Beispiel, welches die Grauzone zwischen Returncodes und Exceptions demonstriert. Unter Umständen kann man dieselbe logische Information entweder über einen besonderen Fehler-Returncode oder über eine Exception ausdrücken kann. Man hätte auch die Methode Object.clone() so spezifizieren können, dass sie ganz ohne Exception auskommt. Object.clone() hat zwei mögliche Ergebnisse: die Referenz auf den erzeugten Klon, falls dieser erzeugt werden konnte, oder die Information, dass das Objekt nicht cloneable ist. Das letztere Ergebnis hätte sich in einer null-Referenz als Rückgabewert ausdrücken lassen. Das hat man allerdings anders gemacht; es wird statt dessen die CloneNotSupportedException geworfen. Und deshalb ist die oben gezeigte Variante einer clone()-Implementierung auch wernig empfehlenswert; eigentlich rechnet kein Benutzer mit einer null-Referenz als Ergebnis von clone().

Empfehlenswert sind die Varianten "Unterdrücken" und" InternalError". Welche von beiden Techniken man vorzieht, ist Geschmacksache. Man kann natürlich auch einen anderen Error oder gar eine unchecked Exception werfen. Beides ist aber unüblich. Es hat sich eingebürgert, dass man einen InternalError wirft, wenn man die CloneNotSupportedException nicht unterdrücken will.

Das leere Cloneable-Interface

Die ganze Verwirrung um die CloneNotSupportedException hätte sich von vornherein vermeiden lassen, wenn das Cloneable-Interface klare Vorgaben machen würde. Die Tatsache, dass Cloneable ein leeres Interface ist, hat allerlei Nachteile.

Wir haben schon im vorletzten Artikel gesehen, dass das leere Cloneable-Interface zum Beispiel beim Kopieren von generischen Collections Schwierigkeiten bereitet; es bleibt einem nichts anderes übrig, als die clone()-Methode per Reflection aufzurufen, weil der Cast auf Cloneable keinen Zugriff auf die clone()-Methode gibt. Außerdem kann es Klassen geben, die das Cloneable-Interface implementieren, aber keine clone()-Methode haben, was völlig widersinnig ist, aber nicht verhindert werden kann. Und im Zusammenhang mit der CloneNotSupportedException wäre es auch wünschenswert, dass das Cloneable-Interface sinnvolle Vorgaben über eine throws-Klausel für die clone()-Methode machte.

Besteht die Aussicht, dass das Cloneable-Interface vielleicht in Zukunft korrigiert wird? Wohl kaum. Egal wie man die clone()-Methode eines Cloneable-Interfaces definiert, die Korrektur würde existierenden Code brechen. Da es nie Vorgaben für die Signatur von clone() gegeben hat, existieren clone()-Methoden mit und ohne throws-Klausel. (Es gibt sogar clone()-Methoden mit throws(CloneNotSupportedException)-Klausel im JDK. Ein Beispiel ist die Klasse java.awt.datatransfer.DataFlavor.)

Ganz egal, wie man sich bei der Korrektur von Cloneable entscheidet, ein Teil des heute existierenden Java-Codes würde unübersetzbar werden. Wenn man das Cloneable-Interface mit einer clone()-Methode ohne throws-Klausel definiert, dann werden all die Klassen unbrauchbar, die eine clone()-Methode mit "throws CloneNotSupportedException "-Klausel haben. Wenn man umgekehrt das Cloneable-Interface mit einer clone()-Methode mit "throws CloneNotSupportedException"-Klausel definiert, dann blieben zwar alle cloneable Klassen gültig, aber die Benutzer dieser Klassen haben ein Problem: sie müssen plötzlich die CloneNotSupportedException behandeln, wenn sie nach eine Cast auf Cloneable die clone()-Methode aufrufen. Wie auch immer man das anstellt, die Änderung des heutigen leeren Cloneable-Interfaces würde in jedem Fall existierenden Code brechen. Solche Brüche hat Sun bislang vermieden; man ist dort sehr um Kompatibilität der JDK-Versionen bemüht. Deshalb ist nicht zu erwarten, dass das Cloneable-Interface jemals eine clone()-Methode haben wird.

Nun kann man das für eigene Projekte und Klassen natürlich anders und besser machen. Als wir das Für und Wider der CloneNotSupportedException auf der OOP-Konferenz im Januar 2002 dargestellt haben, kam folgender Vorschlag aus dem Auditorium: "Kann man nicht ein projekt-spezifisches Cloneable-Subinterface haben, dass eine clone()-Methode hat und diese Interface anstelle des Cloneable-Interfaces verwenden?" Das ist eine gute Idee, die natürlich voraussetzt, dass es Programmierrichtlinien gibt oder die Software-Entwickler anderweitig motiviert sind, dieses neue Interface auch zu verwenden. Das Interface könnte dann wie folgt aussehen:

```
/**
 * In contrast to the standard interface
<code>java.lang.Cloneable</code>
 * this interface has a <code>clone</code> method. It is supposed to
be
 * used in lieu of the standard <code>java.lang.Cloneable</code>
interface.
 *
 * @author  ...
 * @version ...
 * @see     java.lang.Cloneable
 */
public interface CloneableWithCloneMethod extends Cloneable {

  /**
  * Creates and returns a copy of this object.
  *
  * @return     a clone of this instance.
  * @exception  OutOfMemoryError            in case of not enough
memory.
  * @exception  InternalError               in case of an unexpected
CloneNotSupportedException.
  * @see        project.CloneableWithCloneMethod
  */
  public Object clone();
}
```

Damit ist man zwar für die eigenen Klassen einen Schritt weiter, aber beim Klonen von generischen Collections beispielsweise muss man sich immer noch mit existierenden third-party Klassen herumschlagen, die das Cloneable-Interface implementieren und von dem projektspezifischen CloneableWithCloneMethod-Interface nichts wissen.

Zusammenfassung

Die clone()-Methode sollte keine CloneNotSupportedexception werfen, sondern im Fehlerfall einen InternalError auslösen.

Literaturverweise

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

/KRE1/ **Das Kopieren von Objekten in Java (Teil 1)**
Klaus Kreft & Angelika Langer
JavaSpektrum, September 2002
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/05.Clone-Part1/05.Clone-Part1.html

/KRE3/ **Das Kopieren von Objekten in Java (Teil 3): Die CloneNotSupportedException**
Klaus Kreft & Angelika Langer
JavaSpektrum, Januar 2003
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/07.Clone-Part3/07.Clone-Part3.html

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

/CLON/ **Das Kopieren von Objekten in Java (Teil 2 + 3)**
Klaus Kreft & Angelika Langer
JavaSpektrum, November 2002 + Januar 2003
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/06.Clone-Part2/06.Clone-Part2.html
URL: http://www.AngelikaLanger.com/Articles/EffectiveJava/07.Clone-Part3/07.Clone-Part3.html

/HAG/ **Practical Java - Programming Language Guide, Praxis 64**
Peter Haggar
Addison-Wesley, 2000

ISBN: 0201616467

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

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

/LIS/ **Program Development in Java**
Abstraction, Specification, and Object-Oriented Design, Section 4.4
Barbara Liskov with John Guttag
Addison-Wesley, June 2000
ISBN: 0-201-65768-6

/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