Klonen von Objekten in Java

Recherche und Erstellung des Dokuments: Stefan Träger

Als Vorbereitung für Brownbag-Lunch

Inhalt

[1. Problemstellung 3](#_Toc469315113)

[2. Flaches Klonen oder Tiefes Klonen 3](#_Toc469315114)

[2.1 Flaches Klonen 3](#_Toc469315115)

[2.2 Tiefes Klonen 3](#_Toc469315116)

[3. Verschiedene Möglichkeiten des Klonens 4](#_Toc469315117)

[3.1 Serialisierung (Tiefe Kopie) 4](#_Toc469315118)

[3.2 Reflection (Tiefe Kopie) 4](#_Toc469315119)

[3.3 Copy-Constructor (Flache bzw. Tiefe Kopie) 4](#_Toc469315120)

[3.4 Cloneable-Interface und java.lang.clone() (Flache bzw. Tiefe Kopie) 4](#_Toc469315121)

[3.5 Bean Utils (Flache Kopie) 4](#_Toc469315122)

[3.6 Defensives Kopieren 4](#_Toc469315123)

[4. Implementierung 5](#_Toc469315124)

[4.1 Serialisierung 5](#_Toc469315125)

[4.2 Reflection per Hand 6](#_Toc469315126)

[4.3 Copy-Constructor 6](#_Toc469315127)

[4.3.1 Flache Kopie mit Copy-Constructor 6](#_Toc469315128)

[4.3.2 Tiefe Kopie mit Copy-Constructor 7](#_Toc469315129)

[4.4 Cloneable-Interface und java.lang.clone() 7](#_Toc469315130)

[4.4.1 Flache Kopie mit Object.clone() 7](#_Toc469315131)

[4.4.2 Tiefe Kopie mit Object.clone() 8](#_Toc469315132)

[4.5 Bean Utils 9](#_Toc469315133)

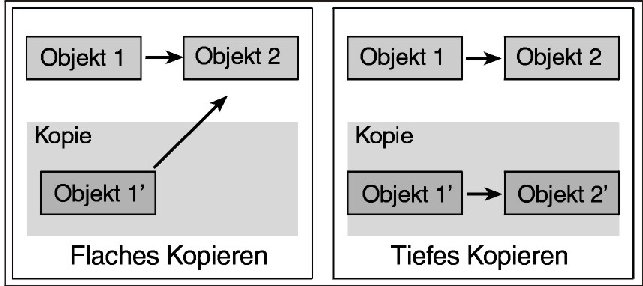
[4.6 Defensives Kopieren 9](#_Toc469315134)

[5. Vergleich bezüglich der Ausführungszeit 10](#_Toc469315135)

# 1. Problemstellung

In Projekten gibt es immer wieder Fälle, in denen eine Kopie eines Objektes benötigt wird. Eine einheitliche Lösung ist meistens nicht vorhanden, so dass jeder Entwickler das auf seine eigene Art und Weise löst. Deshalb soll allgemein eine Lösung gefunden werden, die auch nicht zu stark auf Kosten der Performance geht.

# 2. Flaches Klonen oder Tiefes Klonen



## 2.1 Flaches Klonen

Beim flachen Kopieren wird eine bitweise Kopie des Objekts erstellt. Referenzen auf andere Objekte bleiben dabei erhalten.

## 2.2 Tiefes Klonen

Beim tiefen Klonen werden zusätzlich rekursive Kopien der referenzierten Objekte erzeugt. Tiefe Kopien gehen aber aufgrund der rekursiven Kopien sehr zu Lasten der Performance. Deshalb sollte genau überlegt und geprüft werden, ob eine tiefe Kopie benötigt wird, oder auch eine flache Kopie reichen würde.

# 3. Verschiedene Möglichkeiten des Klonens

Im folgenden Abschnitt möchte ich einige Möglichkeiten vorstellen, wie man flache bzw. tiefe Kopien erstellen kann. Details zu den jeweiligen Implementierungen werden in Kapitel 4 vorgestellt.

## 3.1 Serialisierung (Tiefe Kopie)

Serialisierung bezeichnet die Umwandlung in eine sequentielle Datenstruktur. In Java wird hierbei das zu kopierende Objekt beispielsweise in ein byte-Array umgewandelt. Aus diesem byte-Array wird anschließend ein neues Objekt erzeugt.

## 3.2 Reflection (Tiefe Kopie)

Mittels Reflection können die einzelnen Felder eines Objekts ausgelesen werden und anschließend das neue Objekt mit diesen Werten initialisiert werden. Es gibt auch Frameworks (Beispielsweise „Dozer“), die beim Kopieren mittels Reflection behilflich sein können.

## 3.3 Copy-Constructor (Flache bzw. Tiefe Kopie)

Ein sogenannter „Copy-Constructor“ ist ein Konstruktor, der ein typgleiches Objekt als Parameter besitzt und aus diesem Objekt einen Klon erstellt. Je nach Implementierung dieses Konstruktors kann eine flache oder auch eine tiefe Kopie erstellt werden.

## 3.4 Cloneable-Interface und java.lang.clone() (Flache bzw. Tiefe Kopie)

Java bietet bereits im JDK einen Mechanismus, um Objekte zu kopieren. Hierbei wird die Methode clone() aus dem java.lang-package verwendet.

Die standardmäßige Object.clone() Methode kreiert eine flache Kopie. Wird jedoch das Cloneable-Interface nicht implementiert wirft diese Methode eine „CloneNotSupportedException“.

Wenn man jedoch das Cloneable-Interface implementiert, kann/sollte man auch die clone() Methode überschreiben. Hierbei kann man sie so überschreiben, dass eine tiefe Kopie erzeugt wird.

## 3.5 Bean Utils (Flache Kopie)

Im Apache Commons gibt es die BeanUtils, die via Reflection flache Kopien von Objekten erzeugen.

## 3.6 Defensives Kopieren

Attribute, die mutable, also veränderbar sind, sollte man am besten nicht direkt zurückgeben, sondern ein neues Objekt anhand des Attributes erstellen.

# 4. Implementierung

Implementiert wird das Projekt als Maven-Projekt in Eclipse Neon mit Java 8.

## 4.1 Serialisierung

Das Objekt und auch die darin enthaltenen rekursiven Objekte müssen alle das Serializable Interface implementieren.

Nachfolgend der Code für die Serialisierung bzw. die Deserialisierung eines Objekts:



## 4.2 Reflection per Hand





## 4.3 Copy-Constructor

### 4.3.1 Flache Kopie mit Copy-Constructor



### 4.3.2 Tiefe Kopie mit Copy-Constructor



## 4.4 Cloneable-Interface und java.lang.clone()

### 4.4.1 Flache Kopie mit Object.clone()

Wir lediglich die clone() Methode der Superklasse Object überschrieben, wird nur eine flache Kopie des Objekts erstellt.



### 4.4.2 Tiefe Kopie mit Object.clone()

Um eine tiefe Kopie eines Objekts zu erstellen, muss die Object.clone() Methode entsprechend überschrieben werden. Im folgenden Beispiel wird intern eine Serialisierung für die Kopie verwendet:



## 4.5 Bean Utils



## 4.6 Defensives Kopieren

# 5. Vergleich bezüglich der Ausführungszeit

Um die Performance der einzelnen Möglichkeiten zu vergleichen wird mittels JUnit eine bestimmte Menge an Objekten erzeugt und diese dann anschließend geklont. Dabei wird die Zeit, die die einzelnen Kopiervorgänge gebraucht haben mithilfe der StopWatch aus dem apache-commons-package gemessen.