

# References API: Soft-, Weak- und Phantom-Referenzen

Dominik Gruntz

Fachhochschule Aargau/Nordwestschweiz

[d.gruntz@fh-aargau.ch](mailto:d.gruntz@fh-aargau.ch)

Seit Java 2 gibt es das Paket `java.lang.ref`, in welchem *Soft*-, *Weak*- und *Phantom*-Referenzen definiert sind. Wie die Schnittstellen dieser schwachen Referenzen aussehen, ist bereits in anderen Artikeln beschrieben worden [Paw98,Hag02] (oder kann in der API-Dokumentation [API] nachgelesen werden). In diesem Artikel wird gezeigt, wozu diese Referenzen verwendet werden können. Für jeden der drei Referenztypen wird eine typische Anwendung gezeigt.

Java-Objekte werden auf dem Heap erzeugt und werden, sobald sie nicht mehr benötigt werden, vom Garbage Collector aus dem Speicher entfernt. Ein Objekt wird nicht mehr benötigt, wenn es ausgehend von statischen Feldern und lokalen Variablen der aktiven Methoden aller Threads nicht über eine Folge von Referenzen erreichbar ist (vgl. [Dau01]). Seit dem JDK 1.2 ist es möglich, eine Referenz auf ein Objekt zu definieren, die den Garbage Collector *nicht* daran hindert, das Objekt aus dem Speicher zu entfernen. Diese Referenzen nennt man schwache Referenzen. Damit ist es möglich, dass Java-Programme bis zu einem gewissen Grad mit dem Garbage Collector kooperieren. Der Begriff der Erreichbarkeit muss auf Grund dieser neuen Referenztypen angepasst werden.

Ein Objekt ist stark erreichbar, falls es von einem Thread aus über einen Pfad von gewöhnlichen (starken) Referenzen erreichbar ist. Ein Objekt ist schwach erreichbar, falls es nicht stark erreichbar ist, aber über eine Folge von Referenzen erreicht werden kann, welche mindestens eine schwache Referenz enthält. In Abbildung 1 ist ein Objektgraph dargestellt. Schwache Referenzen sind darin gestrichelt (rot) dargestellt. In diesem Graphen sind die Objekte E und F schwach erreichbar, alle anderen Objekte (insbesondere auch D und G) sind stark erreichbar. Der Garbage Collector könnte also die Objekte E und F bei Bedarf wegräumen.

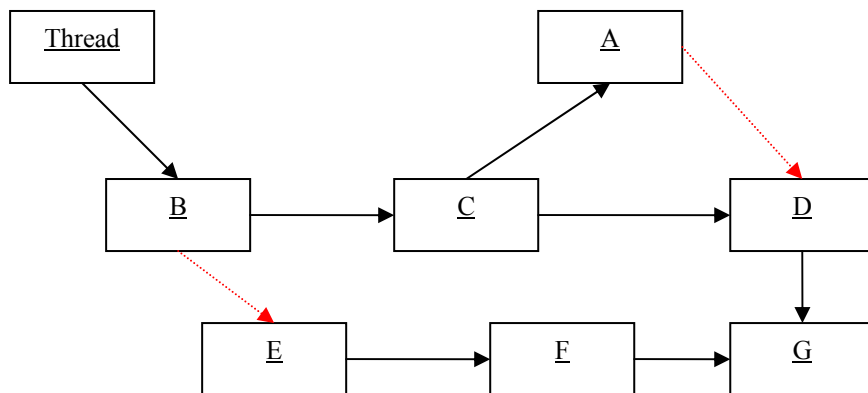


Abbildung 1: Objektgraph mit starken und schwachen Referenzen

Eine schwache Referenz ist ein Objekt vom Typ `Reference`. Diese Klasse enthält einen Verweis auf das schwach angebundene Objekt. Diese Referenz wird vom Garbage Collector speziell behandelt. Mit der Methode `get` kann eine starke Referenz auf das schwach referenzierte Objekt angefordert werden. Diese Methode gibt `null` zurück, falls das referenzierte Objekt nicht mehr verfügbar ist.

Beim Erzeugen einer schwachen Referenz kann eine `ReferenceQueue` mitgegeben werden. In diese Queue wird die Referenz eingefügt, sobald das referenzierte Objekt vom Garbage Collector freigegeben wurde. Eine Queue wird typischerweise für mehrere Referenzen verwendet. Mit den Methoden `poll` (nicht-blockierend) oder `remove` (blockierend) können die Referenzen wieder aus der Queue entfernt werden.

Das Paket `java.lang.ref` definiert drei verschiedene Typen von schwachen Referenzen: *Soft*-, *Weak*- und *Phantom*-Referenzen. Diese drei Klassen sind von der abstrakten Basisklasse `Reference` abgeleitet. Im Folgenden sollen die Unterschiede dieser drei Referenztypen erklärt und mögliche Anwendungen aufgezeigt werden.

## SoftReference

Softreferenzen werden in der Klasse *SoftReference* implementiert. Das Besondere an Softreferenzen ist, dass die darüber referenzierten Objekte erst dann vom Garbage Collector weggeräumt werden, wenn der Speicher knapp wird. Die Spezifikation verlangt nur, dass alle über Softreferenzen angebotenen Objekte vom Garbage Collector weggeräumt worden sind, bevor ein *OutOfMemoryError* geworfen wird; aber die Idee ist, dass der Garbage Collector bei Speicherknappheit zuerst jene Objekte freigibt, die am längsten nicht mehr verwendet worden sind.

Softreferenzen werden typischerweise verwendet, um Daten im Speicher zu cachen, die aufwändig zu erzeugen sind, auf die jedoch verzichtet werden kann, wenn der Speicherplatz knapp wird. Falls z.B. Bilder geladen werden oder Daten aus einer Datenbank oder von einem Web Server gelesen werden, so kann es nützlich sein, diese Daten in einem Cache abzulegen, solange genügend Speicher vorhanden ist. Falls diese Daten später wieder benötigt werden, kann die Anwendung im Cache nachschauen, ob diese dort noch vorhanden sind. Ist dies der Fall, so können sie unverzüglich verwendet werden, andernfalls müssen die Daten neu bereitgestellt werden.

Im Beispiel in Listing 1 wird das Bild *image.gif* mit einer Softreferenz verankert. Die Methode *getImage* prüft jeweils, ob das Bild noch vorhanden ist, andernfalls wird es neu geladen.

```
static SoftReference ref = null;
static Image getImage() {
    Image img = null;
    if (ref != null) img = (Image) ref.get();
    if (img == null) {
        img = new ImageIcon("image.gif").getImage();
        ref = new SoftReference(img);
    }
    return img;
}
```

Listing 1: Cache für ein Image

Falls mehrere Objekte in einem Cache bereitgestellt werden müssen, lohnt es sich, eine Map zu implementieren, in welcher die Werte mit Softreferenzen verankert werden. Die Methode *put* legt anstelle des Objektes eine Softreferenz darauf ab und *get* liefert eine (starke) Referenz auf das gecachte Objekt, sofern dieses nicht bereits aus dem Speicher entfernt worden ist. Sobald der Garbage Collector ein in dieser Map abgelegtes Objekt freigibt, soll der Eintrag auch aus der Map entfernt werden. Dazu wird bei allen Mutator-Methoden (*put*, *remove* und *clear*) die den Softreferenzen zugeordnete Referenzqueue abgefragt und die dort gefundenen Elemente werden aus der Map gelöscht. Damit diese Elemente gelöscht werden können, muss deren Schlüssel bekannt sein. Dieser wird am einfachsten in einer Erweiterung der Klasse *SoftReference* abgelegt.

Die Klasse *SoftHashMap* in Listing 2 implementiert so eine Map. Diese Klasse verwendet intern eine normale *HashMap* um die Objekte zu verwalten. Die Erweiterung von *SoftReference* heisst dort *SoftEntry*. Die Reference-Queue wird in der Methode *processQueue* abgearbeitet.

```
import java.lang.ref.*;
import java.util.*;

public class SoftHashMap extends AbstractMap {
    private Map m = null;
    private ReferenceQueue q = new ReferenceQueue();

    public SoftHashMap() { m = new HashMap(); }
    public SoftHashMap(int initialCapacity) { m = new HashMap(initialCapacity); }
    public SoftHashMap(int initialCapacity, float loadFactor) {
        m = new HashMap(initialCapacity, loadFactor);
    }

    public Object get(Object key) {
        Object res = m.get(key);
        return res == null ? null : ((Reference) res).get();
    }
}
```

```

public Object put(Object key, Object value) {
    processQueue();
    Reference ref = new SoftEntry(key, value, q);
    Object res = m.put(key, ref);
    return res == null ? null : ((Reference) res).get();
}

public Object remove(Object key) {
    processQueue();
    Object res = m.remove(key);
    return res == null ? null : ((Reference) res).get();
}

public void clear() {
    processQueue();
    m.clear();
}

private void processQueue() {
    Reference r;
    while ((r = q.poll()) != null) {
        SoftEntry e = (SoftEntry) r;
        m.remove(e.key);
    }
}

private static class SoftEntry extends SoftReference {
    private Object key; // necessary so that freed objects can be removed
    private SoftEntry(Object key, Object value, ReferenceQueue q) {
        super(value, q);
        this.key = key;
    }
}
}

```

Listing 2: Implementierung einer SoftHashMap für Caches

Listing 2 zeigt nur die wichtigsten Methoden. Weggelassen ist die Methode *entrySet*, welche eine Menge von Entry-Objekten zurückgibt. Diese Menge enthält logisch nur jene Einträge der Map, welche noch auf aktive Objekte verweisen, d.h. die Methoden *iterator* und *size* müssen diesen Spezialfall berücksichtigen. Die vollständige Implementierung der Klasse *SoftHashMap* ist unter [Gru04] verfügbar. Eine Erweiterungsmöglichkeit wäre, die zuletzt verwendeten Objekte über starke Referenzen zu verankern, damit diese nicht vom Garbage Collector entfernt werden.

### WeakReference

Die Klasse *WeakReference* implementiert „normale“ schwache Referenzen. Das über eine Weakreferenz angebundene Objekt wird vom Garbage Collector entsorgt, sobald es nur noch über Folgen von Referenzen erreichbar ist, die alle mindestens eine Weakreferenz enthalten.

Weakreferenzen werden primär verwendet, um Zusatzinformation an Objekte anzufügen. Die zusätzlichen Attribute werden als Attributobjekt in einer Map abgelegt, wobei das attributierte Objekt als Schlüssel verwendet wird. Ein anderer Weg um ein Objekt mit zusätzlichen Attributen zu versehen wäre, seine Klasse zu erweitern. Diese Lösung ist jedoch nicht möglich, wenn es keine Klasse gibt (wie z.B. bei Arrays) oder wenn die Klasse nicht erweiterbar ist (finale Klassen). Zudem müsste beim Erweitern einer Klasse jede *new*-Anweisung geändert werden, was nur dann ohne grossen Aufwand möglich ist, wenn die Objekte über eine Factory erzeugt werden.

Wir hatten kürzlich dieses Problem, als wir an die Knoten eines von ANTLR (Syntaxparser) erzeugten Syntaxbaumes Zusatzinformation anfügen mussten. Den ANTLR-Code wollten wir nicht ändern, da dieser aus einer Beschreibung der Sprache automatisch generiert wird. Wir haben daher diese Zusatzinformationen in einer Map abgelegt, in welcher die Knoten als Schlüssel verwendet werden.

Falls nun jedoch ein Objekt verschwindet, so muss die damit assoziierte Zusatzinformation ebenfalls gelöscht werden. Dies kann erreicht werden, in dem der Schlüssel in der Map über eine Weakreferenz angebunden wird. Sobald die Map erkennt, dass das über die Weakreferenz angebundene

Schlüsselobjekt vom Garbage Collector weggeräumt worden ist, wird auch der zugehörige Eintrag gelöscht. Diese Funktionalität wird in Java in der Klasse *WeakHashMap* bereitgestellt und z.B. in der Klasse *ThreadLocal* verwendet, um Daten mit einem Thread zu assoziieren.

Eine Weakreferenz wird jedoch auch verwendet, um ein Singleton zu verankern. Ein Singleton ist ein Objekt, das nur einmal instanziiert sein darf. Typischerweise implementiert man dazu eine Klasse mit einem privaten Konstruktor und einer Methode *getInstance*, mit welcher auf die Singleton-Instanz zugegriffen werden kann. Um Speicher zu sparen, wird die Instanz in dieser Methode typischerweise verzögert erzeugt (*lazy initialization*). Sobald jedoch niemand mehr das Singleton verwendet, sollte dieses auch wieder vom Garbage Collector freigegeben werden. Dies ist nur möglich, wenn das Singleton in der Klasse über eine schwache Referenz referenziert wird. Listing 3 zeigt das typische Codemuster dazu.

```
public class Singleton {
    private static Reference ref;
    private Singleton ( ) { }
    public static synchronized Singleton getInstance ( ){
        Singleton instance = null;
        if(ref != null) instance = (Singleton)ref.get();
        if(instance == null){
            instance = new Singleton(); ref = new WeakReference(instance);
        }
        return instance;
    }
}
```

Listing 3: Singleton

Mit einem Singleton erreicht man, dass von einer Klasse maximal eine Instanz existiert. Man kann diese Idee auch auf mehrere Instanzen verallgemeinern und sicherstellen, dass von gleichen Objekten jeweils nur eine Instanz existiert. Man spricht dann von Kanonisieren von Objekten. Sinnvoll ist dieser Ansatz jedoch nur, wenn die Objekte unveränderbar (immutable) sind. Der Vorteil ist, dass kanonische Objekte durch einen Vergleich der Referenzen verglichen werden können. Ein Aufruf der unter Umständen teuren Methode *equals* ist dann nicht mehr nötig. Ein guter Kandidat in Java wäre die Klasse *Boolean*. Von dieser Klasse müssten jeweils maximal zwei Instanzen existieren. Kanonische Objekte werden typischerweise auch in Computer-Algebra-Systemen verwendet, um Ausdrücke schnell vergleichen zu können.

In Java können Strings mit der Methode *intern* kanonisiert werden. Diese Methode liefert eine Referenz auf eine eindeutige Repräsentation des gegebenen Strings zurück. Diese Funktionalität kann auch für eigene Klassen mit einer *WeakHashMap* realisiert werden. In dieser Map werden die kanonischen Instanzen abgelegt. Will man ein neues Objekt kanonisieren, so sucht man in dieser Map, ob unter diesem Schlüssel bereits eine eindeutige Repräsentation abgelegt worden ist (dazu müssen in der Klasse der zu kanonisierenden Objekte die Methoden *equals* und *hashCode* überschrieben sein). Falls noch kein Objekt mit diesem Schlüssel vorhanden ist, wird dieses Objekt neu abgelegt (*map.put(x, x)*). Ein Problem dabei ist, dass die so in einer *WeakHashMap* abgelegten Objekte nie mehr aus dem Speicher entfernt werden, da die Werte über starke Referenzen in der Map verankert sind. Hier müssen also ebenfalls schwache Referenzen verwendet werden. Im Listing 4 ist eine Klasse *Canonicalizer* abgebildet, welche mit der Methode *intern* eindeutige Referenzen zurückliefert.

```
public class Canonicalizer {
    private Map m = new WeakHashMap();
    public synchronized Object intern(Object x){
        if(x == null) throw new IllegalArgumentException();
        Object ref = m.get(x);
        if(ref != null) ref = ((Reference)ref).get();
        if(ref != null){ return ref; }
        else {
            m.put(x, new WeakReference(x));
            return x;
        }
    }
}
```

Listing 4: Implementierung eines Kanonisierers

## PhantomReference

Phantomreferenzen sind in der Klasse *PhantomReference* definiert und stellen einen Sonderfall unter den schwachen Referenzen dar. Eine Phantomreferenz auf ein Objekt erlaubt nicht, auf das Objekt zuzugreifen, auch dann nicht, wenn das Objekt vom Garbage Collector noch nicht weggeräumt worden ist. Die Methode *get* liefert daher immer *null* zurück (was nicht ganz konform zur Spezifikation in der abstrakten Basisklasse *Reference* ist).

Phantomreferenzen dienen einzig dazu festzustellen, wann ein Objekt nicht mehr erreichbar ist und vom Garbage Collector weggeräumt wird. Diese Aufgabe wird auch von der *finalize*-Methode übernommen, aber

- in der *finalize*-Methode können das Objekt oder andere referenzierte Objekte wieder verankert werden (z.B. in einem statischen Feld) und
- die *finalize*-Methode kann nur für jene Objekte überschrieben werden, deren Klassen erweiterbar sind, also nicht für finale Klassen oder Arrays.

Phantomreferenzen sind da sicherer und flexibler. Sobald ein (finalisiertes) Objekt als unerreichbar erkannt worden ist und aus dem Speicher entfernt werden kann, wird es, falls eine (noch nicht aus dem Speicher entfernte) Phantomreferenz darauf verweist, in die zugehörige ReferenceQueue eingefügt. Mit *poll* oder *remove* kann es aus dieser Queue entfernt werden, und Aufräumarbeiten können ausgeführt werden. Speziell an den Phantomreferenzen ist, dass das Objekt erst dann aus dem Speicher entfernt wird, wenn auf allen auf dieses Objekt verweisenden Phantomreferenzen die Methode *clear* ausgeführt wird oder wenn diese Phantomreferenzen selber vom Garbage Collector eingesammelt worden sind.

Ein weiterer Vorteil der Phantomreferenzen gegenüber der *finalize*-Methode ist, dass derselbe Aufräumcode für Objekte von unterschiedlichen Klassen verwendet werden kann. In einer Prototyp-Implementierung der Sun Virtual Machine sind Phantomreferenzen beispielsweise verwendet worden, um die Verwendungszeit von Objekten von allen Klassen zu überwachen. Mit dieser Information können später neue Objekte in einem für den Garbage Collector optimalen Bereich angelegt werden [AgGa00].

Als letztes Beispiel wollen wir eine *Finalizer*-Klasse bauen, in welcher für Objekte Aufräumcode registriert werden kann. Dazu wird auf dieses Objekt eine Phantomreferenz definiert. Zu dieser Phantomreferenz wird in einer Map der Aufräumcode (vom Typ *Runnable*) abgelegt. Ein eigener Thread wartet nun darauf, dass eine Phantomreferenz in der Referenzqueue abgelegt wird. Sobald dies der Fall ist, wird der zugehörige Aufräumcode ausgeführt. Dieser Code wird jedoch auch garantiert ausgeführt, wenn die Applikation terminiert. Dazu wird im System ein Shutdownhook registriert, welcher alle ausstehenden Phantomreferenzen in die Queue einfügt.

```
public class Finalizer {
    private static ReferenceQueue queue = new ReferenceQueue();
    private static Map references = new HashMap();
    private static volatile boolean running = true;

    /* This method registers an object to be cleaned up. The cleanup code is called when the object
    * is no longer reachable or when the program shuts down.
    */
    public static void register(Object x, Runnable r){
        synchronized(references){
            references.put(new PhantomReference(x, queue), r);
        }
    }

    static {
        Thread t = new Thread(){
            public void run(){
                while(running || references.size() > 0){
                    try {
                        Reference ref = queue.remove();
                        Runnable run;
                        synchronized(references){run = (Runnable)references.remove(ref);}
                        ref.clear();
                        run.run();
                    }
                }
            }
        };
    }
}
```

```

        }
        catch(Exception e){}
    }
    synchronized(references){references.notify();}
}
};
t.setDaemon(true);
t.start();

Thread hook = new Thread(){
    public void run(){
        synchronized(references){
            Iterator it = references.keySet().iterator();
            while(it.hasNext()){
                ((Reference)it.next()).enqueue();
            }
            running = false;
            try {references.wait();} catch(InterruptedException e){}
        }
    }
};
Runtime.getRuntime().addShutdownHook(hook);
}
}
}

```

Listing 5: Implementierung eines Finalizers

### Zusammenfassung

In diesem Artikel haben wir aufgezeigt, wie und wozu schwache Referenzen in Java verwendet werden können. Wir haben gesehen, dass die Klasse *WeakReference* verwendet wird, um Informationen mit Objekten zu assoziieren oder um Kanonischer zu bauen, und Referenzen vom Typ *SoftReference* für den Bau von in-memory Caches verwendet werden können. Mit der Klasse *PhantomReference* hat man eine flexible Alternative zur *finalize*-Methode. Schwache Referenzen sind sicher kein Werkzeug für den täglichen Gebrauch. Es kann jedoch sinnvoll sein, schwache Referenzen indirekt mit der Klasse *WeakHashMap* oder den in diesem Artikel entwickelten Klassen *SoftHashMap*, *Canonicalizer* oder *Finalizer* zu verwenden, und genau diese Anwendungen wollte dieser Artikel aufzeigen.

### Referenzen:

- [AgGa00] O. Agesen und A. Garthwaite, *Efficient Object Sampling Via Weak References*, in: Proceedings of the 2nd International Symposium on Memory Management, ISMM, p. 121-126, ACM Press, 2000.
- [Am02] J. Amsterdam, *Java References*, Dr. Dobb's Journal, Februar 2002.
- [Gru04] <http://www.gruntz.ch/papers/references/SoftHashMap.java>, 2004.
- [API] Programierschnittstelle, <http://java.sun.com/j2se/1.4.2/docs/guide/refobs/>
- [Dau01] R. Dauster, *Effiziente Speicherverwaltung*, JavaSPEKTRUM, 6/2001.
- [Hag02] P. Haggart, *Guidelines for using the Java2 reference classes*, Oktober 2002, <http://www-106.ibm.com/developerworks/library/j-refs/>
- [Paw98] M. Pawlan, *Reference Objects and Garbage Collection*, August 1998, <http://java.sun.com/developer/technicalArticles/ALT/RefObj/>