www.javaspektrum.de

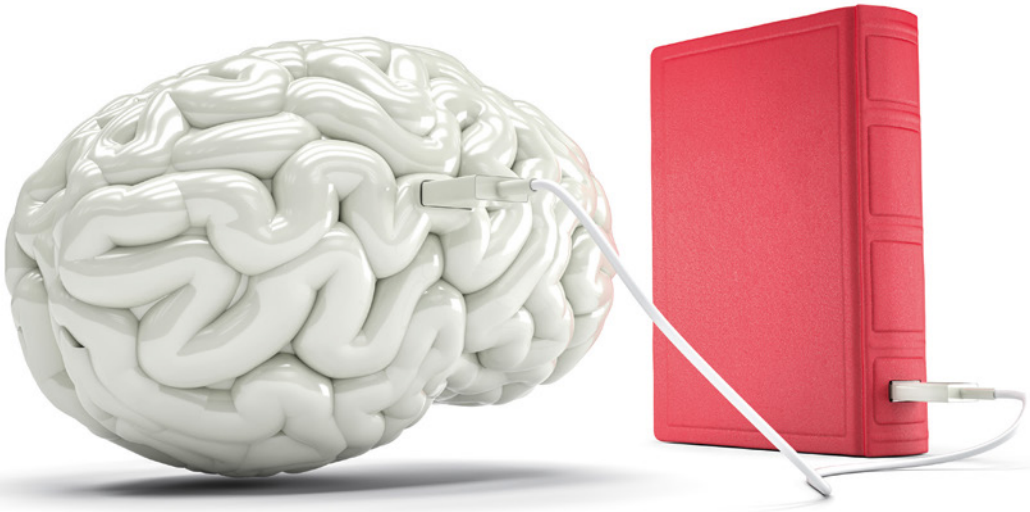
Ausgabe 6/2018
D € 7,90 · A € 8,90 · SFR 14,30

4 194156 107907 06

# JavaSPEKTRUM

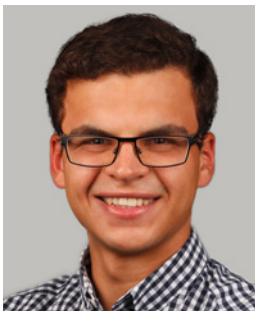
Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen

# Cognitive Computing
## Mit dem Computer auf Augenhöhe?



**Interview**

Die Data Scientists Marc Hilbert und Yury Dzerin über KI-Forschungsprojekte für den Rennsport im VW Data Lab



Cognitive Services von Microsoft

Handschriftenerkennung mit neuronalen Netzen

**Fachthemen**

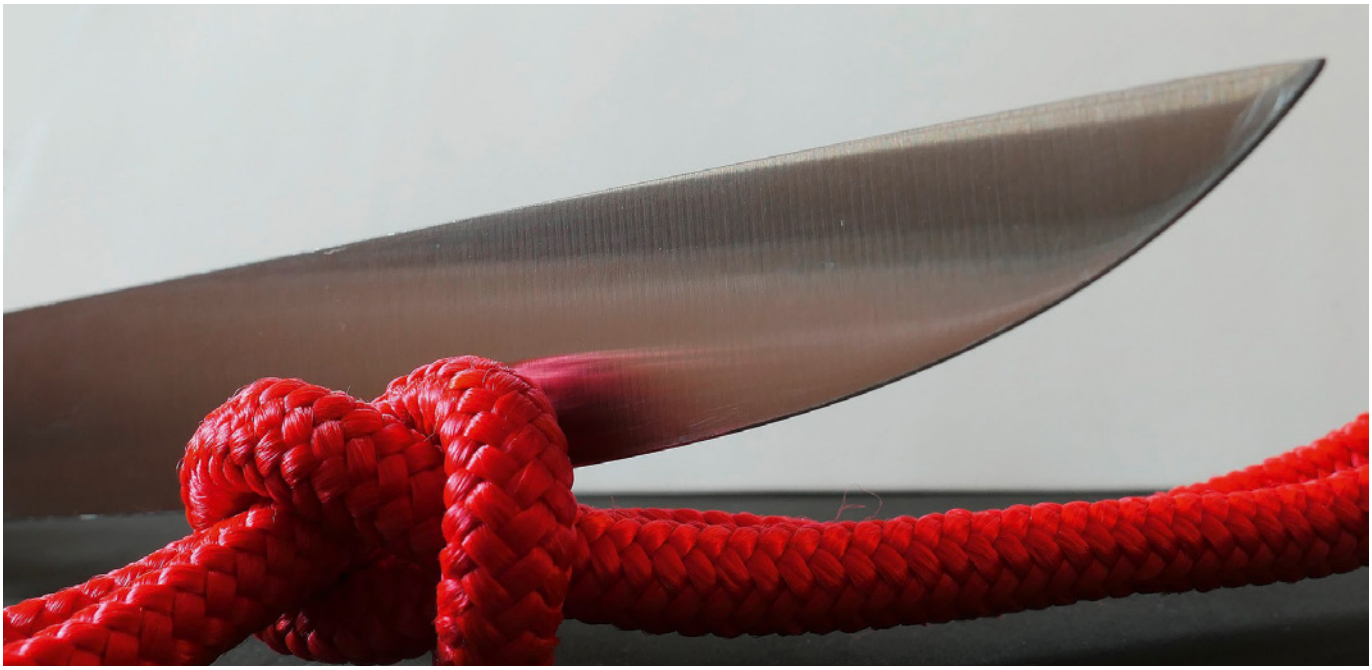
Objektreferenzen in Java

Immerwährende Immunität für Java-Code

zum Inhalt

Cognitive Computing – Mit dem Computer auf Augenhöhe?

G30325



## Garbage-Collector unter der Lupe

# Objektreferenzen in Java

**Christian Heitzmann**

**Trotz Garbage-Collector ist auch Java nicht vor Speicherlecks gefeit. Unfreiwillige Referenzen auf nicht mehr benötigte Objekte unterbinden deren Speicherbereinigung. Dabei verfügt Java – den meisten Programmierern unbekannt – schon seit Version 1.2 über schwächere Formen von Objektreferenzen sowie eine darauf basierende Datenstruktur. So können Objekte automatisch speicherbereinigt werden, obwohl sie noch referenziert werden – oder umgekehrt.**

Um das Zusammenspiel der verschiedenen Objektreferenztypen mit dem Garbage-Collector zu demonstrieren, benötigen wir zuerst große Datenobjekte, die den Speicher füllen. Listing 1 beschreibt eine solche Klasse, die in allen weiteren Codebeispielen verwendet wird.

Im Kern besteht ein `LargeObject` aus einem mit zufälligen Werten gefüllten Byte-Array von 1 MiB Größe. Jede neue Instanz dieser Klasse erhöht die atomaren statischen Zähler `createdObjectCount` und `unfinalizedObjectCount` um 1. Wird das Objekt später finalisiert, so wird `unfinalizedObjectCount` wieder um 1 verringert. Auf diese Weise lässt sich ganz einfach über die insgesamt erzeugten und noch existierenden (unfinalisierten) Objekte Buch führen, ohne dass groß mit Memory-Profilern oder dergleichen gearbeitet werden muss. Die so auf der Konsole ausgegebenen Wertepaare wurden zum Erstellen der Plots in diesem Artikel verwendet und lassen sich vom Leser jederzeit einfach rekonstruieren.

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public final class LargeObject {
  private static final int ARRAY_SIZE
    = 1024 * 1024; /* 1 MiB */
  private static AtomicIntege%r createdObjectCount
    = new AtomicInteger();
  private static AtomicInteger unfinalizedObjectCount
    = new AtomicInteger();
  private byte[] byteArray;

  public LargeObject() {
    /* Create and fill byte array with random data. */
    byteArray = new byte[ARRAY_SIZE];
    ThreadLocalRandom.current().nextBytes(byteArray);
    /* Update and print atomic counts. */
    int updatedCreatedObjectCount
      = createdObjectCount.incrementAndGet();
    int updatedUnfinalizedObjectCount
      = unfinalizedObjectCount.incrementAndGet();
    System.out.println(updatedCreatedObjectCount + "\t"
      + updatedUnfinalizedObjectCount);
  }
  @Override
  public int hashCode() {
    return Arrays.hashCode(byteArray);
  }
  @Override
  protected void finalize() {
    /* Update and print atomic counts. */
    int updatedUnfinalizedObjectCount
      = unfinalizedObjectCount.decrementAndGet();
    System.out.println(createdObjectCount.get() + "\t"
      + updatedUnfinalizedObjectCount);
  }
}
```

*Listing 1: LargeObject*

## Grundlagen der Garbage-Collection

Befasst man sich vertieft mit der Materie von Finalisierung und Garbage-Collection, so stellt man fest, dass es zwischen „Objekt

**Christian Heitzmann** ist Gründer und Geschäftsführer der Simplex Code AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung vor allem für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.
E-Mail: christian.heitzmann@simplexacode.ch

lebendig" und „Objekt tot" noch ein paar Zwischenabstufungen gibt. Hier die Kurzversion: Stellt der Garbage-Collector bei seinem Durchlauf fest, dass ein Objekt nicht mehr erreicht werden kann, so ruft er zuerst dessen `finalize`-Methode auf. Erst in einem zweiten oder gar noch späteren Durchgang wird der Speicher dieses Objektes dann auch wirklich freigegeben.

Der Grund, wieso der Garbage-Collector nicht alles gleich „in einem Wisch" machen kann, liegt darin, dass es der `finalize`-Methode theoretisch freisteht, eine Referenz ihres eigenen Objektes „herauszugeben" und sich so wiederum nicht mehr unerreichbar zu machen. Das wäre bildlich gesprochen so, wie wenn der Sensenmann an der Haustür klingelt, aber anstatt ihm zu öffnen, verlässt man das Haus durch die Hintertür und erfreut sich weiterhin des Lebens.

Der Garbage-Collector benötigt also (mindestens) einen zweiten Durchlauf, um festzustellen, ob das Objekt, welches bereits *finalisiert* wurde, nun auch wirklich unerreichbar ist beziehungsweise blieb. Zu betonen ist dabei, dass die `finalize`-Methode nur *ein einziges Mal* im Leben eines Objektes aufgerufen wird. Um beim leicht makaberen Bild zu bleiben: Beim nächsten Mal wird der Sensenmann seinen Besuch nicht mehr mit der Türklingel ankündigen, sondern direkt zur Tat schreiten.

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

public final class ObjectReferencesDemo {
  private static final int RING_BUFFER_SIZE
    = 20_000;
    // = 6000; or  = 3000; or  = 100;
  private static final int CREATED_OBJECT_COUNT_MAX
    = 20_000;

  public static void main(String[] args)
      throws InterruptedException {
    ObjectReferencesDemo demo = new ObjectReferencesDemo();
    demo.hardReferences();
    /* [...] Other method calls omitted. */
  }
  private void hardReferences() {
    LargeObject[] ringBuffer = new LargeObject[RING_BUFFER_SIZE];
    /* Fill ring buffer with HARD references to large objects. */
    for (int i = 0; i < CREATED_OBJECT_COUNT_MAX; i++) {
      ringBuffer[i % RING_BUFFER_SIZE] = new LargeObject();
    }
    /* Print hash code of random large object. */
    LargeObject randomLargeObject = ringBuffer[nextRandomIndex()];
    assert (randomLargeObject != null);
    System.out.println(randomLargeObject.hashCode());
  }
  /* [...] Other methods defined later. */
  private int nextRandomIndex() {
    return ThreadLocalRandom.current().nextInt(RING_BUFFER_SIZE);
  }
}
```

*Listing 2: Harte Referenzen*

## „Harte" Referenzen

Um den Garbage-Collector bei seiner Arbeit zu beobachten, lassen wir das Demoprogramm aus Listing 2 laufen. Es besteht im Wesentlichen aus einem `LargeObject`-Array der Größe `RING_BUFFER_SIZE`. Wie der Name bereits impliziert, wird dieses Array als Ringpuffer verwendet. Werden dem Array also mehr Objekte hinzugefügt, als es fassen kann, dann beginnt der Index einfach wieder von vorne.

Dies lässt sich elegant mit der Modulo-Operation (Rest einer Division) `i % RING_BUFFER_SIZE` realisieren. Die früher an diesem Index erzeugten Objekte verlieren damit ihre (einzige) Referenz zum Programm. Wie Heliumballone, deren Schnur abgetrennt wurde, sind sie ab dann nicht mehr erreichbar und werden früher oder später vom Garbage-Collector entfernt. Genau diese Aufräumaktion möchten wir beobachten.

In diesem und allen folgenden Beispielen werden immer 20 000 `LargeObject`s erzeugt. Die Größe des Ringpuffers lässt sich einstellen. Auf meiner Maschine (Java 8 auf einem iMac mit 32 GiB Arbeitsspeicher) haben sich die in den Listings beziehungsweise den beiden Abbildungen aufgeführten Werte als anschaulich herausgestellt. Der Leser kann und soll dies natürlich für seine Gegebenheiten anpassen.

Im Anschluss an die füllende `for`-Schleife wird noch ein zufälliges Element herausgepickt und dessen Hash-Code angezeigt. Dies hat zwei Gründe: Zum einen sollen dem Compiler und der Virtual Machine alle Möglichkeiten verbaut werden, irgendwelche Optimierungen im Sinne von „diese Objekte werden ja später eh nie wieder gebraucht, also kann man da abkürzen" vorzunehmen. Zum anderen soll es vor allem für die weiter unten vorgestellten Objektreferenztypen den Zugriff auf das referenzierte Objekt demonstrieren.

Lässt man das Demoprogramm mit den Ringpuffergrößen (RPGs) 20 000, 6000, 3000 sowie 100 laufen und plottet jeweils die Wertepaare „erzeugte Objekte" und „unfinalisierte Objekte", dann ergibt sich (auf meiner Maschine) die Abbildung 1. Zu erkennen ist klar der `OutOfMemoryError` bei knapp 6900 Objekten (roter Graph), wenn dem Programm keine Möglichkeit gegeben wird, sich

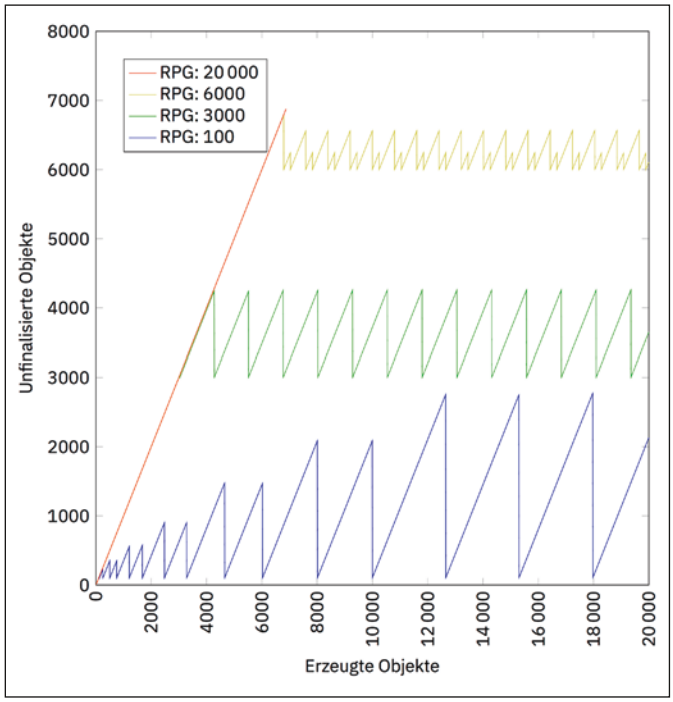



*Abb. 1: Harte Referenzen*

```
private void weakReferences() {
  WeakReference<?>[] ringBuffer
    = new WeakReference<?>[RING_BUFFER_SIZE];
  /* Fill ring buffer with WEAK references to large objects. */
  for (int i = 0; i < CREATED_OBJECT_COUNT_MAX; i++) {
    ringBuffer[i % RING_BUFFER_SIZE]
      = new WeakReference<LargeObject>(new LargeObject());
  }
  /* Print hash code of random large object, if possible. */
  LargeObject randomLargeObject
    = (LargeObject) ringBuffer[nextRandomIndex()].get();
  System.out.println(Objects.hashCode(randomLargeObject));
}
```

*Listing 3: Weak-Referenzen*

alter Objekte zu entledigen, was bei einer RPG von 20 000 zweifelsohne der Fall ist.

Bei einer RPG von 6000 bleibt dem Garbage-Collector wenig Spielraum. Er muss (nachdem der Ringpuffer einmal komplett befüllt wurde) stets 6000 große Objekte im Speicher halten, sieht sich allerdings sehr oft mit der „oberen Schmerzgrenze" von ca. 6900 Objekten konfrontiert. Als Folge daraus werden sehr oft und in sehr kurzen Intervallen die Objekte speicherbereinigt, um sich die notwendige, wenn auch knappe „Atemluft" zu verschaffen.

Die Abbildungen für die RPGs 3000 und 100 sind selbsterklärend. Hervorzuheben ist hier, dass der Garbage-Collector den Speicher praktisch nie ganz volllaufen lässt, sondern bereits viel früher, als er eigentlich müsste, zur Speicherbereinigung schreitet. Im blauen Graphen (RPG 100) ist auch zu sehen, dass der Garbage-Collector über Heuristiken verfügt, die ihn anfangs schnell Kleinigkeiten beseitigen lassen, mit der Zeit aber bemerken lassen, dass immer weitere Daten nachkommen, für die sich größere Zyklen anbieten.

## Weak-Referenzen

Java stellt mit der Klasse `java.lang.ref.WeakReference<T>` eine Art „Wrapper" zur Verfügung, der aus einer klassischen „harten" Referenz eine „schwache" Referenz macht. Listing 3 zeigt, wie eine solche *Weak-Referenz* erzeugt und verwendet wird.

Der Garbage-Collector darf gemäß Spezifikation alle Objekte bereinigen, die nur noch über Weak-Referenzen erreichbar sind. In der Folge kann es also passieren, dass das Objekt „hinter" der Weak-Referenz verschwunden ist. Zugriff auf das Objekt erhält man über die `get`-Methode, die im erwähnten Fall `null` zurückgeben wird.

Abbildung 2, die aus dem Demoprogramm mit Weak-Referenzen entstanden ist, ist eindrücklich. Praktisch unabhängig von der Ringpuffergröße entscheidet sich der Garbage-Collector von Zeit zu Zeit, gleich *alle* schwachen Referenzen zu entfernen. Vor allem für die RPG 20 000 – die vorher zu einem Speicherüberlauf führte – besteht diese Gefahr nun nicht mehr. Im Gegenzug muss man nun damit rechnen, dass der Zugriff auf ein zufälliges Element des Arrays in dem Sinne „fehlschlägt", als dass man nach Aufruf der `get`-Methode der `WeakReference` nun „mit leeren Händen dasteht", sprich nur noch `null` zurückerhält.

Für was soll das nun gut sein? Ganz einfach, es kann Fälle geben, wo es „noch schön" wäre, ein (großes) Objekt im Speicher behalten und referenzieren zu können, zum Beispiel ein großes Bild. Wird dieses Objekt aber (zum Beispiel aufgrund von Speicherknappheit) doch vom Garbage-Collector entfernt, dann ist es „auch nicht weiter tragisch". Im Notfall lässt sich das Bild ja wieder von einem anderen Ort einlesen.

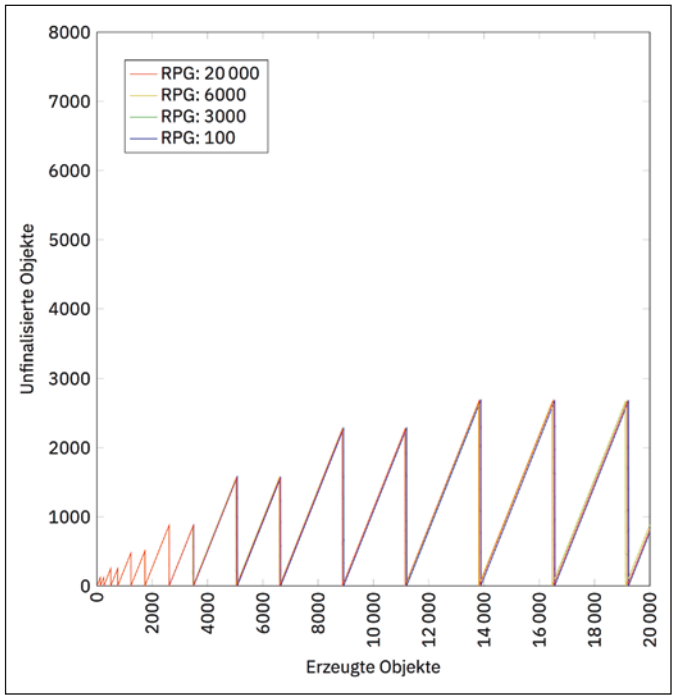



*Abb. 2: Weak-Referenzen*

## Soft-Referenzen

*Soft-Referenzen* verhalten sich theoretisch ähnlich wie Weak-Referenzen. Wenn man Listing 4 mit dem vorherigen Listing vergleicht, findet man außer beim Objektreferenztypen keinen Unterschied.

In der Dokumentation zur Klasse `SoftReference<T>` heißt es: *„All soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an* `OutOfMemoryError`*."* [JSE8Ref] Soft-Referenzen sind zwar „weich", aber doch ein kleines bisschen „härter" als die oben erwähnten Weak-Referenzen.

Leider stellt sich die zitierte Aussage in der Praxis als unwahr heraus. Wer das Demoprogramm mit einer RPG von 20 000 laufen lässt, wird eine `OutOfMemoryException` erhalten, obwohl dies laut Spezifikation nicht passieren dürfte.

Die Antwort auf dieses Problem findet sich tief vergraben in einer Oracle-FAQ zum Garbage-Collector. Gemäß dieser Antwort werden Soft-Referenzen speziell behandelt, nämlich indem sie erst dann entfernt werden, wenn sie schon ein gewisses Alter erreicht haben und gleichzeitig ein hoher Speicherdruck herrscht

```
private void softReferences() {
  SoftReference<?>[] ringBuffer
      = new SoftReference<?>[RING_BUFFER_SIZE];
  /* Fill ring buffer with SOFT references to large objects. */
  for (int i = 0; i < CREATED_OBJECT_COUNT_MAX; i++) {
    ringBuffer[i % RING_BUFFER_SIZE]
      = new SoftReference<LargeObject>(new LargeObject());
  }
  /* Print hash code of random large object, if possible. */
  LargeObject randomLargeObject
    = (LargeObject) ringBuffer[nextRandomIndex()].get();
  System.out.println(Objects.hashCode(randomLargeObject));
}
```

*Listing 4: Soft-Referenzen*

```
private void phantomReferences(boolean clearReferences) {
  PhantomReference<?>[] ringBuffer
    = new PhantomReference<?>[RING_BUFFER_SIZE];
  /* Set up reference queue and corresponding worker thread. */
  ReferenceQueue<LargeObject> referenceQueue
    = new ReferenceQueue<>();
  Thread queueWorkerThread
    = new Thread(new QueueWorkerRunnable(referenceQueue,
      clearReferences));
  queueWorkerThread.setDaemon(true);
  queueWorkerThread.start();
  /* Fill ring buffer with PHANTOM references to large objects. */
  for (int i = 0; i < CREATED_OBJECT_COUNT_MAX; i++) {
    ringBuffer[i % RING_BUFFER_SIZE]
      = new PhantomReference<LargeObject>(
        new LargeObject(), referenceQueue);
  }
  /* Obtaining reference to random large object will fail. */
  LargeObject randomLargeObject
    = (LargeObject) ringBuffer[nextRandomIndex()].get();
  assert (randomLargeObject == null);
}
```

*Listing 5: Phantom-Referenzen*

[FAQ]. Mit anderen Worten: Je älter die Soft-Referenz und je knapper der freie Speicher, desto eher wird sie bereinigt. Bezüglich des „Alters" der Soft-Referenz sprechen wir hier allerdings nicht von Millisekunden oder Sekunden, sondern eher von Minuten oder gar Stunden. Die Idee der Soft-Referenzen sollte es sein, damit (Least-Recently-Used-)Caches implementieren zu können. Die notwendige „Schräubchendreherei" macht es aber aus meiner Sicht viel zu fragil, um sich darauf verlassen zu können, geschweige denn Portabilität zu gewährleisten.

Es gibt zwei Möglichkeiten, das Problem zu umgehen. Man kann einerseits das Virtual-Machine-Flag `-XX:SoftRefLRUPolicyMSPerMB` auf einen Wert setzen, der geringer als die standardmäßigen 1000 ms ist. So bedeutet zum Beispiel `-XX:SoftRefLRUPolicyMSPerMB=50`, dass die „Soft-Reference-Least-Recently-Used-Policy" ihren Schwellenwert bei 50 ms pro (freiem) Megabyte hat. Wer sich darunter nichts vorstellen kann, ist vermutlich nicht alleine. Für das Demoprogramm musste ich auf meiner Maschine den Wert auf 1 oder 0 heruntersetzen, um einen Speicherüberlauf zu vermeiden.

Man kann andererseits die großen Objekte auch einfach langsamer erzeugen. Auf meiner Maschine lief das Programm erst ohne Absturz, als ich pro Schleifendurchlauf eine Pause von 1000 ms eingefügt hatte.

## Phantom-Referenzen

Weak- und Soft-Referenzen bieten dem Programm stets eine (wenn auch instabile) Referenz zum dahinterliegenden Objekt an, gegenüber dem Garbage-Collector ist es aber jederzeit „zum Abschuss freigeben". Bei *Phantom-Referenzen* verhält es sich genau umgekehrt: Sie bieten dem Programm *keine* Referenz an, aber dem Garbage-Collector wird das Bereinigen des Objektes verboten. Wahrlich eine seltsame Konstellation, zu der mir spontan das Bild einer Pantomime einfällt, die verzweifelt versucht, durch eine imaginäre abgeschlossene Tür zu gehen. Aufschließen kann sie aber auch niemand, weil es die Tür in echt ja gar nicht gibt.

Phantom-Referenzen dienen dazu, Objekte (noch) im Speicher zu behalten, obwohl das Programm (schon lange) keinen Zugriff mehr darauf hat. Der Hauptanwendungsfall bestand ursprünglich darin, die Garbage-Collection großer Objekte zu verhindern beziehungsweise so lange aufzuschieben, bis sich ein günstigerer





zum Inhalt

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

final class QueueWorkerRunnable implements Runnable {
  private ReferenceQueue<LargeObject> referenceQueue;
  private boolean clearReference;

  public QueueWorkerRunnable(ReferenceQueue<LargeObject> referenceQueue,
      boolean clearReference) {
    this.referenceQueue = Objects.requireNonNull(referenceQueue);
    this.clearReference = clearReference;
  }
  @Override
  public void run() {
    try {
      while (true) {
        /* Block until next reference object is available. */
        Reference<? extends LargeObject> largeObjectReference
          = referenceQueue.remove();
        /* Clear reference object and let its memory reclaim. */
        if (clearReference) {
          largeObjectReference.clear();
        }
      }
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

*Listing 6: Hintergrundthread für eine ReferenceQueue*

Zeitpunkt dafür anbot. Listing 5 zeigt eine Anwendung der Phantom-Referenzen.

## ReferenceQueue

Im Codebeispiel wird neu die `ReferenceQueue` eingeführt, welche sich eigentlich für alle drei vorgestellten Objektreferenztypen verwenden lässt, aber nur für Phantom-Referenzen obligatorisch ist. Weak- und Soft-Referenzen werden dieser Warteschlange hinzugefügt, *bevor* der Garbage-Collector das dahinterliegende Objekt finalisiert. Phantom-Referenzen werden der Queue hinzugefügt, *nachdem* das Objekt finalisiert, aber noch nicht speicherbereinigt wurde.

Dieser letzte Punkt ist entscheidend: Das Objekt wurde bereits finalisiert, und sobald feststeht, dass man mit ihm nichts anderes mehr machen kann, als es aus dem Speicher zu entfernen, landet es in der `ReferenceQueue`. Über diese Queue lässt sich ein Zugang zur entsprechenden `PhantomReference`, nicht aber zum dahinterliegenden Objekt verschaffen. Der Aufruf von `PhantomReference#get` wird *immer* `null` zurückgeben, um zu verhindern, dass man ein Objekt, das in seinen letzten Atemzügen steckt, doch noch wiederbelebt.

Das Einzige, was man mit einer Phantom-Referenz aus der Warteschlange noch machen kann, ist der Aufruf ihrer `clear`-Methode. Bei allen Objektreferenztypen führt `clear` zur Freigabe gegenüber dem Garbage-Collector. Und hier sind wir beim Hauptzweck: Durch das Aufschieben des `clear`-Aufrufes hin zu einem günstigeren Zeitpunkt lässt sich die Speicherbereinigung manuell steuern. Listing 6 zeigt den Code eines Hintergrundthreads, der einer `ReferenceQueue` die Objektreferenzen entnimmt und (bei entsprechendem Flag) manuell deren Speicherbereinigung triggert.

Der Leser kann dies ausprobieren, indem er das Demoprogramm mit einer Ringpuffergröße von 20 000 laufen lässt und wahlweise das „Clearen" der Phantom-Referenzen erlaubt (`true`) oder unterbindet (`false`). Bei Letzterem wird es zu einer `OutOfMemoryException` kommen, obwohl die Objekte bereits finalisiert wurden.

## WeakHashMap

Mit der Klasse `java.util.WeakHashMap` stellt Java eine Hash-Tabelle zur Verfügung, deren Schlüssel als Weak-Referenzen implementiert sind. Solange der Benutzer dieser Hash-Tabelle die originalen Schlüssel „in der Hand behält", bleiben auch die entsprechenden Tabelleneinträge bestehen. Verschwindet ein originaler Schlüssel, das heißt, existieren zu ihm keine „harten" Referenzen mehr, dann steht es dem Garbage-Collector frei, in Bälde auch den entsprechenden Schlüssel aus der Tabelle zu entfernen. Intern verwendet die `WeakHashMap` das oben beschriebene Konzept der `ReferenceQueues`, um nach Entfernen des (Weak-)Schlüssels auch den dazugehörigen Wert in der Tabelle zu entfernen. Eine `WeakHashMap` räumt sich also stets selber auf, indem sie für die automatische Entfernung aller Tabelleneinträge sorgt, zu denen kein originaler Schlüssel mehr existiert.

## Zusammenfassung

Die Implementierung einer eigenen, vermeintlich „besseren" Datenstruktur ist immer ein zweischneidiges Schwert. Gleiches gilt für das „Tuning" des Garbage-Collectors. Die Versuchung ist oft groß, das Resultat jedoch selten besser. Eine fundierte Auseinandersetzung mit der eigenen Programmlogik und der Einsatz von Standard-Datenstrukturen aus bewährten Bibliotheken, allen voran das Collection-API, ist in der Regel nachhaltiger.

Dennoch stellen die hier vorgestellten Objektreferenztypen ein interessantes Konzept dar. Rein persönlich würde ich von den Soft- und Phantom-Referenzen absehen. Für Weak-Referenzen und vor allem für die `WeakHashMap` lassen sich aber durchaus berechtigte Anwendungen finden – jetzt, wo einem deren theoretische Grundlagen bekannt sind.

## Literatur und Links

**[Code]** Quelltexte zum Herunterladen, https://link.simplexacode.ch/qnge

**[FAQ]** Oracle, Frequently Asked Questions About the Java HotSpot VM, https://www.oracle.com/technetwork/java/hotspotfaq-138619.html#gc_softrefs

**[JSE8jls126]** Java Language Specification, 12.6 Finalization of Class Instances, https://docs.oracle.com/javase/specs/jls/se8/html/jls-12.html#jls-12.6

**[JSE8OF]** Java Platform, Standard Edition 8 API Specification, Object#finalize, https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#finalize--

**[JSE8Ref]** Java Platform, Standard Edition 8 API Specification, Package java.lang.ref, https://docs.oracle.com/javase/8/docs/api/java/lang/ref/package-summary.html

**[JSE8WHM]** Java Platform, Standard Edition 8 API Specification, Class WeakHashMap<K,V>, https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html