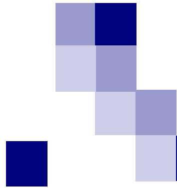


Java Garbage Collection

Kurzfassung

In diesem Seminarbericht möchte ich zuerst auf die Frage eingehen, was Garbage Collection ist und welche Vor- und Nachteile es in Bezug auf eine konventionelle Heapverwaltung mit sich bringt. Des weiteren stelle ich unterschiedliche Ansätze für Garbage Collection vor und gehe speziell auf die Java Garbage Collection Technik ein. Zu guter letzt zeige ich, wie Default-Einstellungen in der Java Garbage Collection verändert werden können, um die Java Garbage Collection an die eigene Anwendung anzupassen.

Was ist Garbage Collection?

In Programmiersprachen wie C, C++, die keine Garbage Collection verwenden, muss dynamisch allokierter Speicher explizit freigegeben werden. Bei Systemen mit Garbage Collection übernimmt das Garbage Collection System die Freigabe von nicht mehr referenzierten Objekten. Diese werden automatisch erkannt und freigegeben. Somit steht dieser freigegebene Speicher im weiteren Programmverlauf wieder zur Verfügung. Eine Definition von Garbage Collection könnte daher lauten: „Garbage Collection ist der Prozess des automatischen Freigebens von Objekten, die vom Programm nicht länger referenziert werden".

Vorteile von Garbage Collection

Ein Vorteil liegt auf der Hand: die Last des expliziten Freigebens von Speicher wird dem Programmierer abgenommen. Somit werden die beiden Fehlerquellen – eine zu frühe Freigabe (das heißt eine Freigabe eines Objekts, obwohl dieses noch referenziert wird und ein daraus entstehender Dangling Pointer) oder gar keine Freigabe – beseitigt.

Nachteile von Garbage Collection

Ein Nachteil liegt darin, dass das automatische Einsammeln von „Garbage" vom Laufzeitsystem gemacht werden muss. Das heißt es entsteht Overhead, der durch das explizite Anfordern und Freigeben von Speicher nicht anfällt. Dieser zu bewältigende Overhead mag in „normalen" Anwendungen keine allzu große Rolle spielen (obwohl dies natürlich immer von der Größe der Anwendung und der Anwendung selbst abhängt), jedoch in Echtzeitsystemen kann dies zu einem echten Problem werden.

Garbage Collection Algorithmen

Unabhängig von einem bestimmten Verfahren, müssen alle Garbage Collection Algorithmen zwei grundsätzliche Aufgaben erfüllen:

- zum einen müssen sie „Garbage"-Objeke erkennen und
- zum zweiten muss der Speicherplatz, der durch das Freigeben des „Garbage"-Objektes entsteht, dem Programm wieder zugänglich gemacht werden.

Widmen wir uns in einem ersten Schritt dem Erkennen von nicht mehr referenzierten Objekten. Ein Objekt kann freigegeben werden, wenn es vom Programm nicht mehr erreicht werden kann und somit den weiteren Programmablauf nicht mehr beeinflussen kann. Das heißt, wir müssen uns die Frage stellen, wann ein Objekt nicht mehr erreichbar ist bzw. welche Objekte erreichbar sind. Das Programm kann auf ein Objekt zugreifen, wenn es entweder

- direkt einen Zeiger darauf besitzt oder
- über eine oder mehrere Indirektionsstufen einen Zeiger erlangen kann (d. h. ein Objekt, auf das das Programm direkten Zugriff hat, besitzt einen Zeiger auf ein oder mehrere weitere Objekte (z. B. der Zeiger auf das erste Element einer verketteten Liste)).

Das Programm besitzt eine Menge von Zeigern

- in lokalen Variablen am Stack
- in globalen (statischen) Variablen und
- in Registern.

Diese Menge von Zeigern nennt man Wurzelzeiger. Alle über Wurzelzeiger direkt oder indirekt erreichbaren Objekte sind lebendig. Alle anderen können vom Garbage Collector freigegeben werden, da sie den weiteren Programmablauf nicht mehr beeinflussen können. Um die nicht mehr erreichbaren Objekte zu erkennen, gibt es mehrere unterschiedliche Ansätze, auf die ich im folgenden näher eingehen werde.

1) Reference Counting

Das Reference Counting ist das älteste Verfahren und ich möchte es nur der Vollständigkeit halber kurz erwähnen, da es nur mehr in Spezialfällen verwendet wird. In Java wird nicht mit dem Reference Counting Verfahren gearbeitet, da es einige gravierende Nachteile hat.

Die Grundidee des Reference Counting besteht darin, einfach alle Zeiger, die auf ein Objekt verweisen, zu zählen. Hierfür verwendet man einen Zähler für jedes Objekt. Hat ein Objekt keinen Zeiger mehr, der darauf

verweist, so hat der Zähler den Wert null und der Garbage Collector weiß, dass er dieses Objekt freigeben kann.

Nun muss man sich bei diesem Verfahren zwei grundsätzliche Fragen stellen:

1) Wie können Referenzen auf ein Objekt hinzukommen?
2) Wie können Referenzen von einem Objekt entfernt werden?

Diese beiden Fragen sind zentral, denn es wird beim Reference Counting „Buch geführt", wie viele Zeiger auf ein Objekt verweisen.

1) Referenzen können in folgenden Fällen hinzukommen:

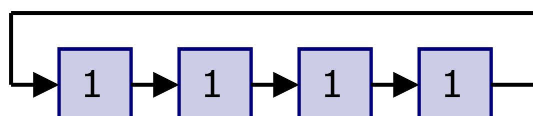
- Objekterzeugung: A a = new A(); //Zähler wird mit 1 initialisiert
- Zuweisungen: a = b; //Zähler von b wird um 1 erhöht
- Parameterübergabe: foo(a); //Zeiger wird kopiert

2) Referenzen können in folgenden Fällen wegfallen:

- Zuweisungen: a = b; //Zähler von a wird um 1 erniedrigt
- Freigabe lokaler Zeiger am Ende einer Methode
- Freigabe eines Objekts, das Zeiger enthält //Dekrementieren kann sich also fortpflanzen

Die Vorteile des Reference Counting liegen darin, dass sich der entstehende Aufwand der Garbage Collection auf die gesamte Laufzeit gleichmäßig verteilt. Das heißt es kommt zu keinem längeren Programmstopp. Ein weiterer Vorteil liegt darin, dass der Block sofort wieder freigegeben wird, sobald er nicht mehr referenziert wird. Somit steht dieser Speicherplatz dem weiteren Programmverlauf sofort wieder zur Verfügung.

Der Compiler muss also bei bestimmten Anweisungen zusätzlichen Code erzeugen, der das Inkrementieren bzw. Dekrementieren übernimmt. Das Inkrementieren und Dekrementieren stellt einen zusätzlichen Aufwand (Overhead) dar, der beispielsweise Zeigerzuweisungen ineffizient macht. Ein weiterer Nachteil des Reference Counting liegt darin, dass die Zähler Speicherplatz benötigen, der bei anderen Verfahren nicht benötigt wird. Des weiteren behandelt das Reference Counting zyklische Datenstrukturen nicht korrekt, da die Zähler einer zyklischen Datenstruktur niemals auf null gehen.



-> auf jedes Objekt verweist genau ein Zeiger, obwohl die Objekte vom Programm aus nicht mehr erreichbar sind

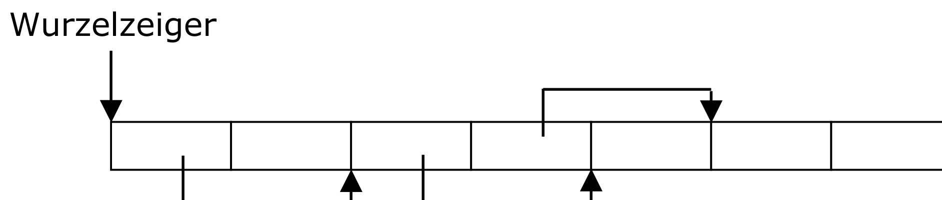
2) Mark and Sweep

Der Mark and Sweep Algorithmus verfolgt einen gänzlich anderen Ansatz als der Reference Counting Algorithmus. Hierbei wird die Erreichbarkeit eines Objekts nicht über einen Zähler definiert, sondern berechnet. Aus diesem Grund besteht dieses Verfahren auch aus zwei Phasen:

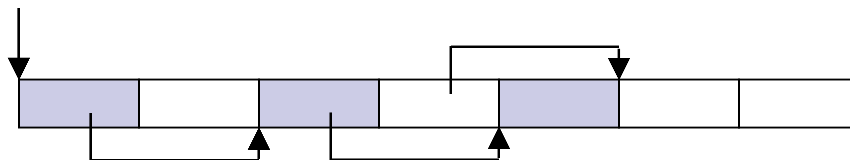
- zum einen aus der Mark-Phase und
- zum anderen aus der Sweep-Phase

In der Mark-Phase werden abhängig von den Wurzelzeigern alle erreichbaren Objekte berechnet (bzw. markiert). In der anschließenden Sweep-Phase werden die nicht markierten Objekte - also jene, die von den Wurzelzeigern aus nicht erreichbar sind - freigegeben.

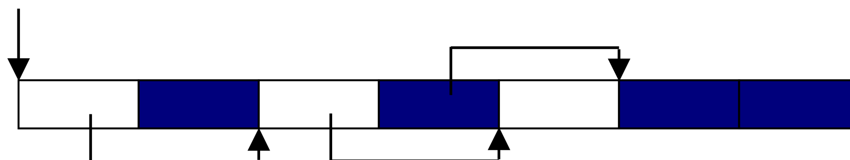
Beispiel:



In der Mark-Phase werden alle erreichbaren (lebendigen) Blöcke markiert:



In der Sweep-Phase werden die nicht markierten Blöcke freigegeben und die Markierung der erreichbaren Blöcke zurückgesetzt:



Im Gegensatz zum Reference Counting Verfahren werden die Objekte, die nicht mehr referenziert werden, nicht sofort freigegeben, sondern erst bei dem Lauf des Garbage Collectors. Bei jedem Garbage Collection Aufruf kommt es zu einer längeren Programmunterbrechung - der Aufwand für die Garbage Collection verteilt sich also nicht gleichmäßig auf das gesamte Programm. Ein weiterer Nachteil des Mark and Sweep liegt darin, dass die Sweep-Phase von der Größe des Heaps abhängig ist, daher ist auch die gesamte Laufzeit eines Garbage Collection Laufs von der Größe des Heaps abhängig.

Ein Vorteil des Mark and Sweep liegt darin, dass zyklische Datenstrukturen korrekt behandelt werden. Diese wurden ja beim Reference Counting nicht richtig verarbeitet. Des weiteren benötigt man für jeden Block nur ein Markierungsbit anstelle eines gesamten Zählers. Ein weiterer wichtiger Vorteil liegt darin, dass bei Zeigerzuweisungen (oder sonstigen Zeigermanipulationen) kein Overhead entsteht.

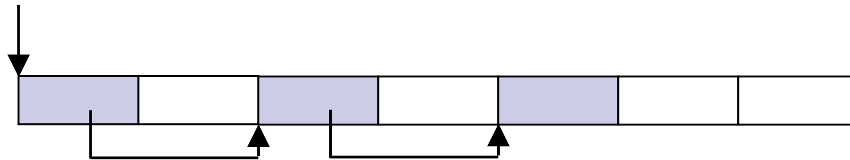
Der stark abstrahierte Pseudocode des Mark and Sweep Verfahrens könnte folgendermaßen aussehen:

```
void gc(){
  foreach (root variable r){
    mark(r);
  }
  sweep ();
}

void mark (Object p){
  if (!p.marked){
    p.marked = true; //Objekt markieren
    foreach (Object q referenced by p){
      mark(q); //rekursiver Aufruf!
    }
  }
}

void sweep (){
  foreach (Object p in the heap){ //Heapdurchlauf
    if (p.marked){
      p.marked = false; //Markierung zurücksetzen
    }
    else{
      heap.release(p); //Objekt freigeben
    }
  }
}
```

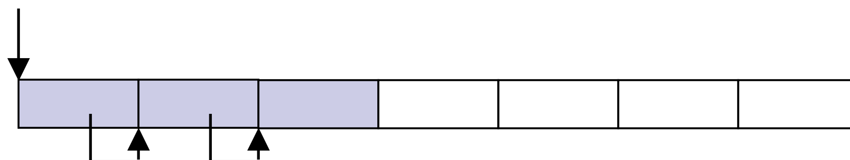
Obwohl der grundsätzliche Lösungsansatz dieses Algorithmus korrekt ist, hat er dennoch einen gravierenden Fehler, nämlich die rekursiven Aufrufe. Jeder rekursive Aufruf benötigt Speicher (am Stack). Ist dieser voll, so läuft dieser in den Heap – dies ist hier aber nicht möglich, da der Heap ja bereits voll ist. Es wird also ein Algorithmus benötigt, der ohne zusätzlichen Speicherbedarf auskommt, also ein iterativer Durchlauf der Objekte. Ein solches Verfahren ist der „Deutsch-Schorr-Waite"-Algorithmus, das ich hier der Vollständigkeit halber erwähnen möchte. Hierbei wird der Rückweg in den Zeigern selbst gespeichert und die Datenstruktur iterativ durchlaufen.

3) Mark and Compact

Ein Problem, welches wir bisher noch außer Acht gelassen haben, ist die Fragmentierung. Nach einem längeren Programmlauf und mehreren Garbage Collection Läufen verteilen sich die aktuell lebenden Objekten auf den gesamten Heap, das heißt zwischen lebendigen Objekten entstehen kleinere und größere freie Bereiche.



Es wäre wünschenswert, dass diese kleineren und größeren freien Bereiche zu einem einzigen großen freien Bereich werden würden.



Genau diese Aufgabe erfüllt der Mark and Compact Algorithmus. Hierbei handelt es sich nämlich um eine verdichtende (also defragmentierende) Variante des Mark and Sweep. Um Defragmentierung zu erreichen, sind allerdings 3 Sweep-Läufe notwendig.

Der Algorithmus arbeitet folgendermaßen:

<u>1. Schritt:</u>
markiere alle erreichbaren Blöcke -> normale Mark-Phase

<u>2. Schritt:</u>
berechne für jeden markierten Block die Zieladresse nach Verdichtung (1. Sweep)

<u>3. Schritt:</u>
biege Wurzelzeiger und die Zeiger in den Blöcken auf die neue Zieladresse um (2. Sweep)

<u>4. Schritt:</u>
verschiebe die Blöcke auf die neue Zieladresse

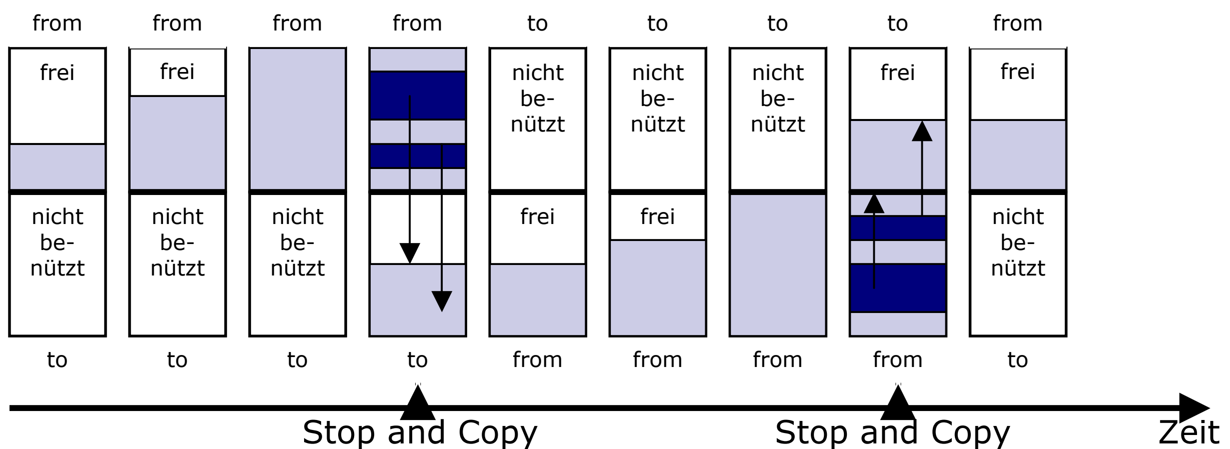
Ein Nachteil, der beim Lesen dieses Algorithmus sofort auffällt, ist, dass hierbei 3 Sweep-Läufe benötigt werden und dadurch das Verfahren langsam ist. Des weiteren muss zusätzlicher Speicher für die Zieladressen aufgewendet werden. Ein weiterer Nachteil liegt darin, dass die Blöcke kopiert werden müssen.

Nimmt man diese Nachteile in Kauf, so erhält man einen defragmentierten Heap, mit dem das Anfordern von Speicher besonders einfach und effizient wird (der freie Bereich ist ja immer ein großer Block).

4) Stop and Copy

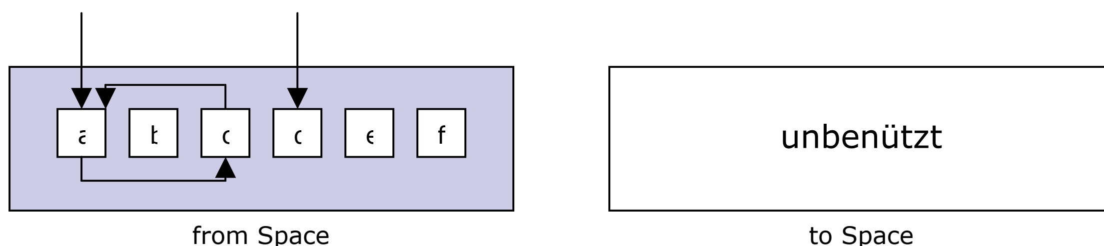
Kopierende Garbage Kollektoren wie der Stop and Copy Garbage Kollektor kopieren alle lebendigen Objekte in einen neuen Speicherbereich. Hierfür wird der Heap in zwei Teile (from Space und to Space) geteilt. Einer der beiden Teile ist zu einem gewissen Zeitpunkt jeweils unbenützt. Neue Objekte werden immer im from Space angelegt. Ist dieser voll, so beginnt die Arbeit des Garbage Kollektors. Ausgehend von den Wurzelzeigern werden die lebendigen (erreichbaren) Objekte durchwandert und in den to Space kopiert. Das Originalobjekt im from Space wird als kopiert markiert und in diesem Objekt wird ein forwarding Pointer auf die Kopie installiert. Sind alle erreichbaren Objekte kopiert, so befindet sich im from Space nur mehr Garbage (also nicht mehr erreichbare Objekte). Jetzt werden from Space und to Space einfach vertauscht und das Programm kann weiterlaufen.

Die folgende Graphik verdeutlicht den Ablauf:



Im ersten „Heap-Schnappschuss" ist nur ein Teil des Heaps belegt. Im zweiten Bild ist für weitere Objekte Speicher allokiert worden. Im dritten Bild schließlich ist der Speicher erschöpft und das Stop and Copy Verfahren beginnt zu laufen. Alle erreichbaren Objekte werden in den to Space kopiert und anschließend werden from und to Space vertauscht und das Spiel beginnt von neuem.

Gehen wir nun von dieser sehr groben Ebene auf eine feinere. Angenommen wir haben folgende Situation:



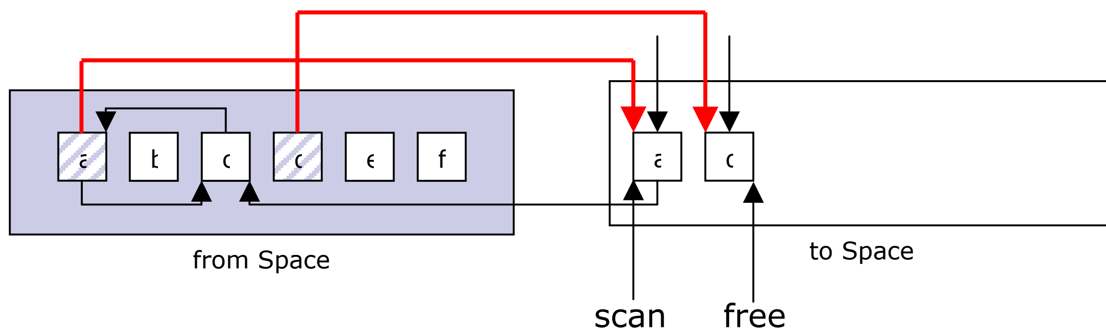
-> from Space ist erschöpft -> Stop and Copy beginnt

1. Schritt:

- alle von Wurzelzeigern aus erreichbaren Objekte werden von from nach to kopiert und als kopiert markiert
- des weiteren müssen die Wurzelzeiger auf die neuen Objekte gesetzt werden
- außerdem müssen von den Originalen forwarding Pointer auf die Kopien gesetzt werden (dies ist nötig, um später zu wissen, wo sich die Kopie im to Space befindet)

Sind diese Aufgaben erledigt, wird ein scan-Zeiger an den Beginn des to Spaces gesetzt sowie ein free-Zeiger an das Ende des to Spaces (genauer: an den Beginn des freien Platzes im to Space -> ab dem free-Zeiger können neue Objekte angelegt werden).

Heap nach Schritt 1:

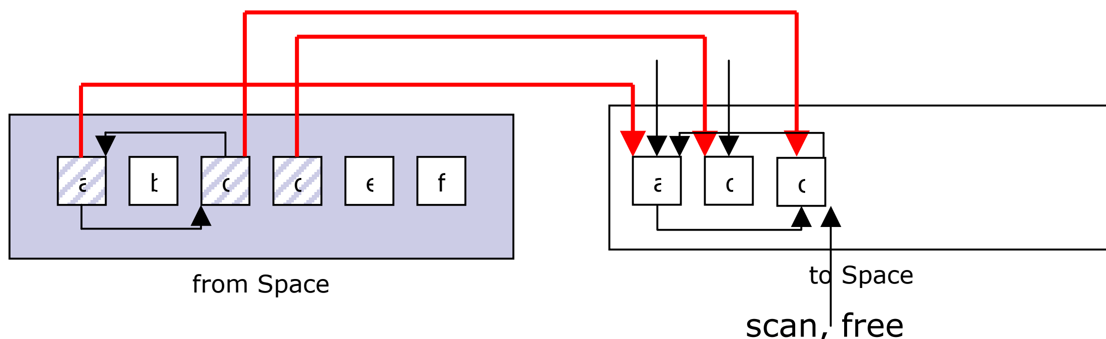


2. Schritt:

Nun muss der scan-Zeiger durch die kopierten Objekte bewegt werden und es muss nach Referenzen in diesen Objekten gesucht werden. Stößt der scan-Zeiger auf eine Referenz, so müssen folgende Schritte erledigt werden:

Ist das referenzierte Objekt noch nicht kopiert, so muss es in den to Space kopiert werden und als kopiert markiert werden, ein forwarding Pointer auf die Kopie gesetzt werden und es muss der Zeiger auf die Kopie „umgebogen" werden. Wurde das Objekt bereits vorher kopiert, so muss nur der Zeiger auf die Kopie „umgebogen" werden.

Heap nach Schritt 2:

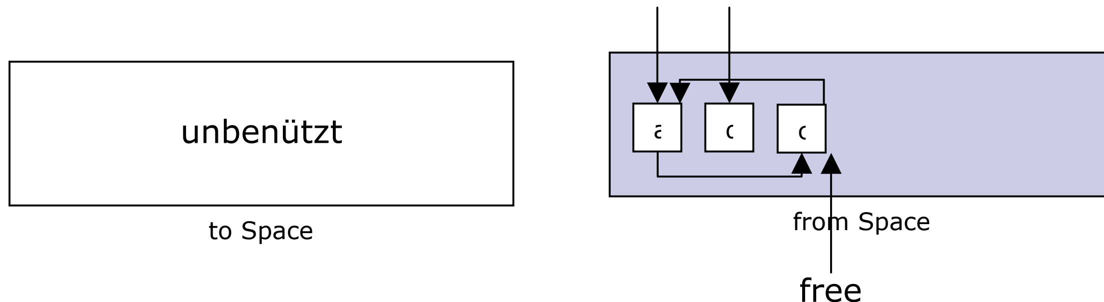


Das Verfahren terminiert, wenn scan == free.

<u>3. Schritt:</u>

from Space und to Space vertauschen

<u>Heap nach Schritt 3:</u>



Nun kann das Programm weiterlaufen und neue Objekte können ab dem free-Zeiger angelegt werden.

<u>Vorteile:</u>

Ein großer Vorteil des Stop and Copy Verfahrens liegt darin, dass es im Gegensatz zum Mark and Sweep Verfahren mit einem Pass auskommt. Zusätzlich verdichtet das Verfahren mit diesem einen Pass (vgl. Mark and Compact: Mark-Phase plus 3 Sweep-Phasen!). Durch dieses Verdichten wird das Allokieren von Speicher sehr einfach, da einfach die angeforderte Menge ab dem free-Zeiger zurückgegeben werden kann. Ein weiterer Vorteil gegenüber dem Mark and Sweep Verfahren liegt darin, dass die Laufzeit des Algorithmus unabhängig von der Heapgröße ist. Beim Stop and Copy Verfahren hängt die Laufzeit von der Anzahl der lebendigen Objekte ab.

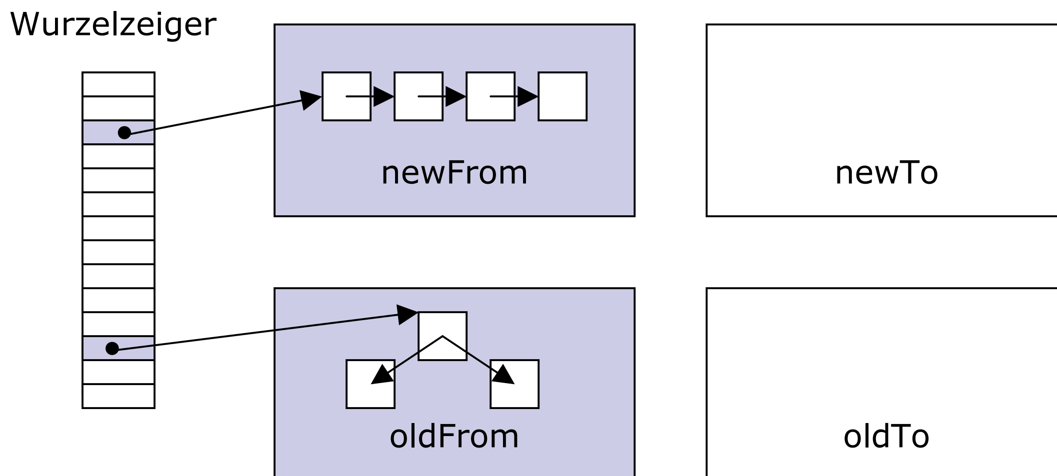
<u>Nachteile:</u>

Das Stop and Copy Verfahren hat zwei wesentliche Nachteile: zum einen kann nur der halbe Heap zu einem Zeitpunkt verwendet werden, das heißt man hat doppelten Speicherbedarf. Zum anderen kostet das Kopieren der Objekte Zeit.

4) Generation Scavenging

Das Generation Scavenging stellt eine Variante des Stop and Copy Verfahrens dar. Ein Problem, das bei dem Stop and Copy Verfahren auftritt, ist, dass langlebige Objekte bei jeder Garbage Collection erneut kopiert werden müssen. Daher unterscheidet das Generation Scavenging Verfahren zwischen kurzlebigen und langlebigen Objekten. Das heißt der Heap wird nicht nur in zwei Teile (from Space und to Space) geteilt, sondern in vier (newFrom, newTo, oldFrom, oldTo). Neue Objekte werden immer im newFrom Space angelegt. Nachdem ein Objekt mehrere Garbage Collecti-

on Läufe überlebt hat, kommt es in den old Space. Das heißt es muss für jedes Objekt ein Zähler mitgeführt werden, der das Alter eines Objekts enthält. Steigt das Alter eines Objekts über einen gewissen Wert, so wird es in den old Space kopiert.



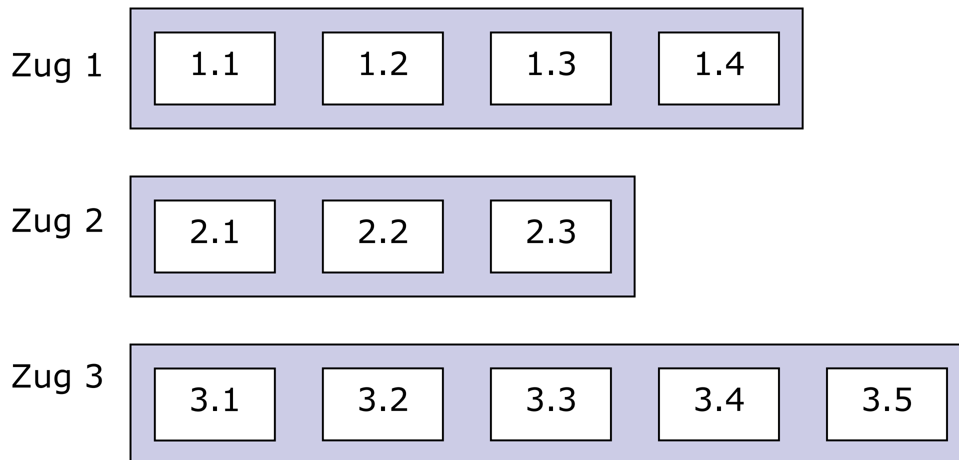
5) Train Algorithmus

Ein Problem, das alle bisherigen Verfahren gemeinsam haben, ist, dass sie die Programme (unter Umständen) für eine längere Zeit unterbrechen. Aus diesem Grund wünscht man sich ein Verfahren, das die Speicherbereinigung inkrementell erledigen kann. Somit wird bei jedem Aufruf des Garbage Kollektors nur ein kleiner Teil des Heaps bereinigt. Eine Garbage Collection dauert daher nie lange und der Aufwand für die Speicherbereinigung verteilt sich gleichmäßig.

Der Train Algorithmus ist ein Verfahren, der diesen Anforderungen gerecht wird. Hierbei stellt man sich den Heap wie einen großen Bahnhof vor. In einem Bahnhof gibt es mehrere Züge und jeder Zug besteht wiederum aus mehreren Waggons. Jeder Zug steht auf einem bestimmten Gleis, das eine bestimmte Nummer trägt. Die Waggons gehören jeweils zu einem bestimmten Zug, wobei sie innerhalb des Zugs eine bestimmte Position haben.

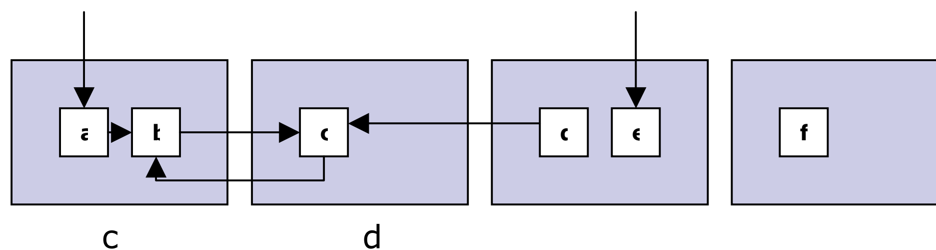
Übertragen auf den Heap kann man sich das nun folgendermaßen vorstellen: Der gesamte Heap wird in Blöcke gleicher Größe (Waggons) unterteilt. Bei einem Garbage Collection Lauf wird jeweils nur ein Block (Waggon) bereinigt. Diese Waggons sind nun zu einem bestimmten Zug zugeteilt und sie haben innerhalb des Zugs eine Ordnung. Bei dem Aufruf des Garbage Collectors wird jeweils der Waggon mit der niedrigsten Nummer bereinigt. Dies ist der erste Waggon in dem Zug mit der niedrigsten Nummer.

Der Heap könnte zu einem bestimmten Zeitpunkt folgendermaßen eingeteilt sein:



In diesem Beispiel besteht der erste Zug aus 4 Waggons, der zweite aus 3 und der dritte aus 5. Beim nächsten Aufruf des Garbage Kollektors wird nun der Waggon mit der niedrigsten Nummer bereinigt werden. In diesem Beispiel ist dies der Waggon mit der Nummer 1.1.

Der Train Algorithmus bereinigt die Waggons mit dem Stop and Copy Verfahren. Hierfür muss der Train Algorithmus zum einen alle Wurzelzeiger kennen. Zum anderen, da ja jeweils der erste Waggon bereinigt wird, alle Zeiger die aus nachfolgenden Waggons auf Objekte in diesen Waggon zeigen. Die Zeiger, die also von hinten in einen vorderen Waggon zeigen, werden in einem so genannten Remembered Set verwaltet.



In diesem Beispiel zeigt auf den ersten Waggon nur ein Zeiger von hinten nach vorne, nämlich aus dem c-Objekt. Genauso verhält es sich hier mit dem zweiten Waggon, hier zeigt ein Zeiger aus dem d-Objekt von hinten nach vorne. Bei den beiden letzten Waggons zeigt kein Zeiger von hinten nach vorne, somit sind die Remembered Sets von diesen beiden Waggons leer.

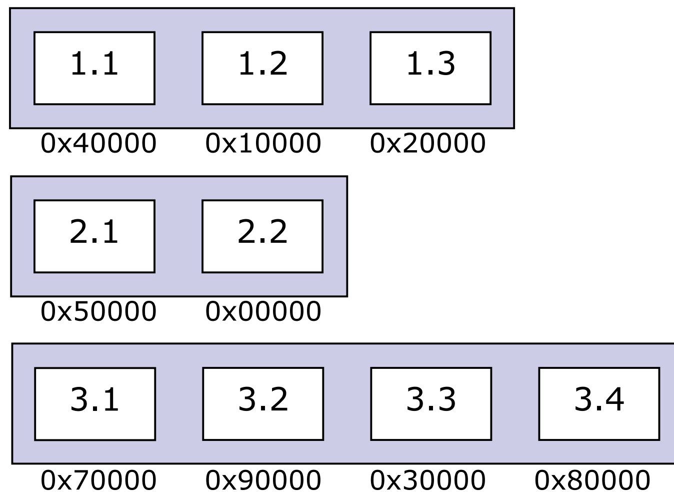
Eine Frage, die sich hier nun aufdrängt ist: Wie weiß man, wann ein Zeiger von hinten nach vorne zeigt? Die logische Einteilung 1.1, 1.2, … stimmt ja nicht mit der physischen (im Speicher) überein. Hierfür wird eine Tabelle mitgeführt, die die physischen Adressen auf die logische Reihenfolge abbildet.

Um den Aufwand für diese Tabelle gering zu halten, kann folgendes Schema verwendet werden:

Blockgröße (Waggongröße) = 2^k
somit ist die Blockadresse ein Vielfaches von 2^k
Blocknummer = Blockadresse um k Bits nach rechts geshiftet
Tabelle[Blocknummer] besagt, welcher Waggon dort steht

<u>Beispiel</u>

Block#	Zug	Waggon
0	2	2
1	1	2
2	1	3
3	3	3
4	1	1
5	2	1
6	-	-
7	3	1
8	3	4
9	3	2
10	-	-



Physisch im Speicher liegen sie also folgendermaßen:

2.2	1.2	1.3	3.3	1.1	2.1	leer	3.1	3.4	3.2	leer

Angenommen wir haben folgende Situation:
Zeiger 0x00B40 zeigt nach 0x40F10

Ist dies ein Zeiger von hinten nach vorne?
Auf 0x00B40 befindet sich laut Tabelle Waggon 2.2.
Auf 0x40F10 befindet sich laut Tabelle Waggon 1.1.
Daher: Zeiger aus 2.2 zeigt auf 1.1 = Zeiger von hinten nach vorne.

Eine weitere Frage, die noch ungeklärt ist, ist, wohin die Objekte bei der Bereinigung eines Waggons kopiert werden. Bei dem einfachen Stop and Copy Verfahren gibt es ja nur 2 Bereiche: from Space und to Space. Bei einer Garbage Collection werden alle erreichbaren Objekte vom from Space in den to Space kopiert und anschließend werden die beiden vertauscht. Beim Train Algorithmus hängt das Kopierziel davon ab, von wo aus die zu kopierenden Objekte erreicht werden. Hierbei gibt es folgende Möglichkeiten:

- entweder wird ein Objekt über einen Wurzelzeiger erreicht
- oder es wird über ein Objekt aus einem hinteren Waggon erreicht

Wird das Objekt über einen Wurzelzeiger erreicht, so wird es in den letzten Waggon des letzten Zugs kopiert. Wird es von einem hinteren Waggon aus erreicht, so unterscheidet man

- ob dieser Waggon demselben Zug angehört
- oder ob es ein Waggon aus einem anderen Zug ist

Gehört der Waggon demselben Zug an, so kommt das kopierte Objekt in den letzten Waggon desselben Zugs. Gehört es hingegen einem anderen Zug an, so kommt das Objekt in den letzten Zug des jeweiligen Objekts.

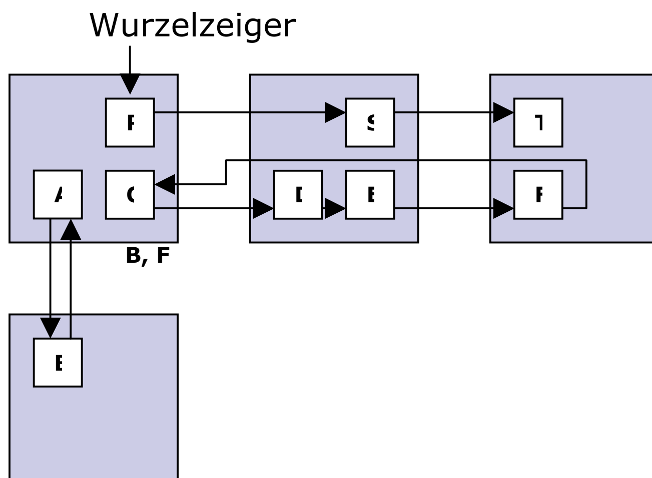
Diese Regel ist entscheidend, da sie sicherstellt, dass zyklische Datenstrukturen in einem Zug zusammenkommen. Ist eine zyklische Datenstruktur vom Programm aus nicht mehr erreichbar, so gibt es keinerlei zugfremde Zeiger mehr darauf und sie kann freigegeben werden.

<u>Train-Algorithmus in Pseudocode</u>

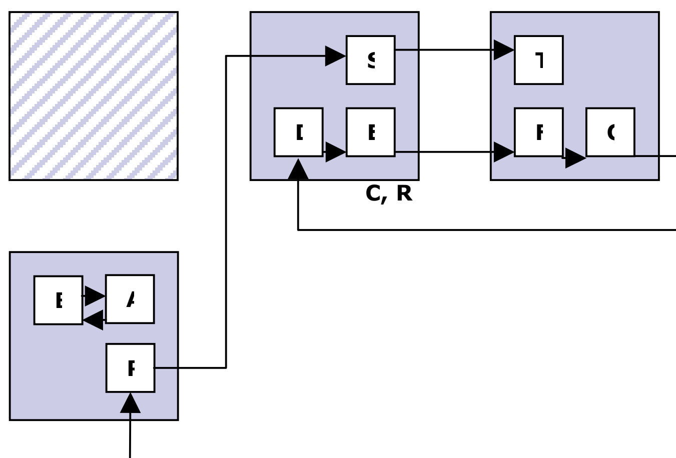
```
if (keine zugfremden Zeiger auf 1. Zug){
     gib ganzen 1. Zug frei;
}
else{
     car = Waggon mit niedrigster Nummer;

     foreach (p in RememberedSet(car)){
          kopiere p_obj in letzten Waggon von Train(p);
          wenn voll -> neuer Waggon in Train(p);
     }

     foreach (p in Wurzelzeiger auf car){
          kopiere p_obj in letzten Waggon des letzten Zugs
          (nicht 1. Zug), wenn voll -> neuer Zug
     }

     for (all pointers p in copied objects){//scan
          if (p zeigt nach car){
               kopiere p_obj in letzten Waggon von Train(p);
               wenn voll -> neuer Waggon im gleichen Zug
          }
          else if (p zeigt auf Waggon m, der vor Car(p) liegt){
               trage p im RememberedSet(m) ein;
          }
          else if (p zeigt auf junge Generation){
               trage p im RememberedSet der jungen Generation
               ein;
          }
     }
}
gib car frei;
```

<u>Beispiel</u>

Annahme: max. 3 Objekte finden in einem Waggon Platz

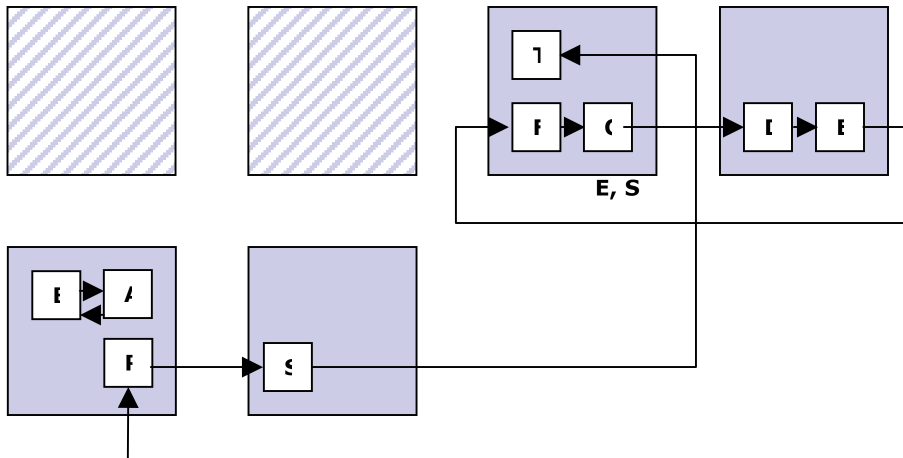


Im ersten Schritt (= 1 Aufruf des Garbage Kollektors) wird der erste Waggon bereinigt. Hierbei werden zuerst alle Objekte, die laut Remembered Set aus einem hinteren Waggon aus erreicht werden, kopiert. Aus diesem Grund kommt das Objekt A in den 1. Waggon des 2. Zugs und das Objekt C in den letzten Waggon des 1. Zugs. Des weiteren kommen alle von Wurzelzeigern aus erreichbaren Objekte in den letzten Waggon des letzten Zugs. In unserem Fall kommt daher das Objekt R in den 1. Waggon des 2. Zugs. Dann werden die Zeiger in den kopierten Objekten untersucht. Da C einen Zeiger auf den 2. Waggon des ersten Zugs besitzt, wird es in diesem Remembered Set eingetragen. Das gleiche gilt für das Objekt R. Nach dem ersten Schritt ergibt sich daher folgendes Bild:

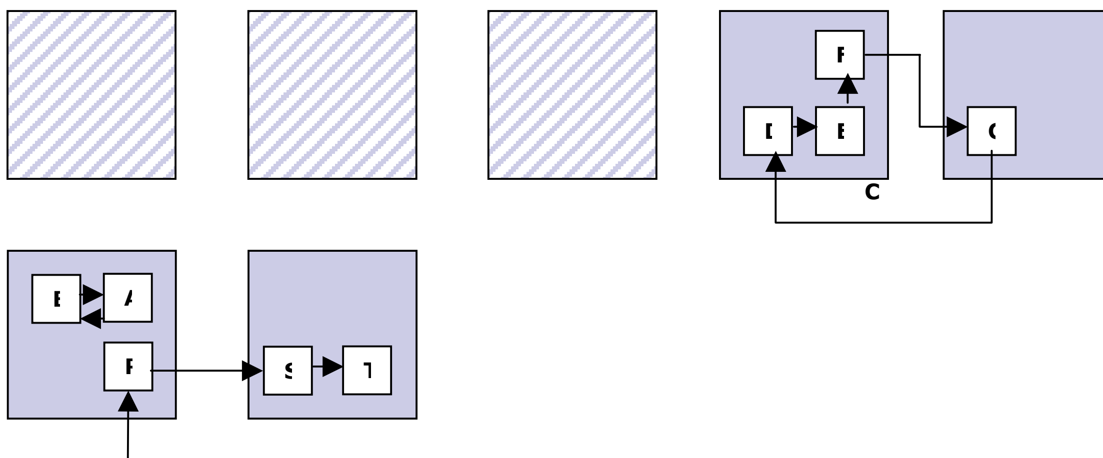


Im zweiten Schritt wird nun der zweite Waggon bereinigt. Hierbei werden wiederum zuerst jene Objekte kopiert, die laut Remembered Set von hinteren Waggons aus erreicht werden können. In unserem Fall sind dies die Objekte S und D. Da in den zweiten Waggon keine Wurzelzeiger zeigen, werden die kopierten Objekte nun nach Zeigern durchsucht. Das Objekt S hat einen Zeiger auf das Objekt T, darum wird es im Remembered Set des Waggons von T eingetragen. Das Objekt D hat einen Zeiger auf das Ob-

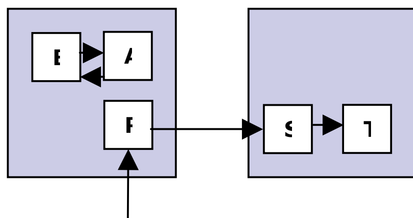
jekt E. Daher muss das Objekt E ebenfalls kopiert werden. Da das Objekt E nun auch zu den kopierten Objekten zählt, wird es auch nach Zeigern durchsucht. Das Objekt E hat einen Zeiger auf das Objekt F und muss daher im Remembered Set des Waggons von F eingetragen werden. Nach dem zweiten Schritt ergibt sich daher folgendes Bild:



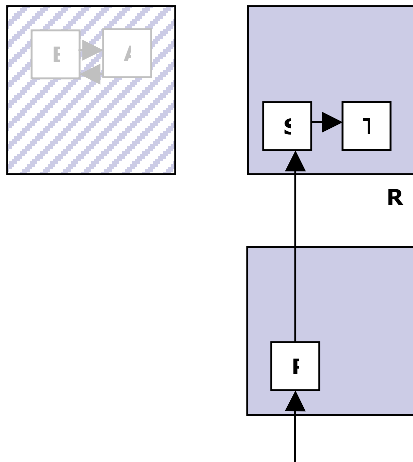
Nach Bearbeitung des dritten Waggons ergibt sich folgendes Bild:



Im nächsten Schritt ergibt sich die Situation, dass keine zugfremden Zeiger auf den ersten Zug zeigen, daher wird der gesamte 1. Zug freigegeben. Es ergibt sich daher folgendes Bild:

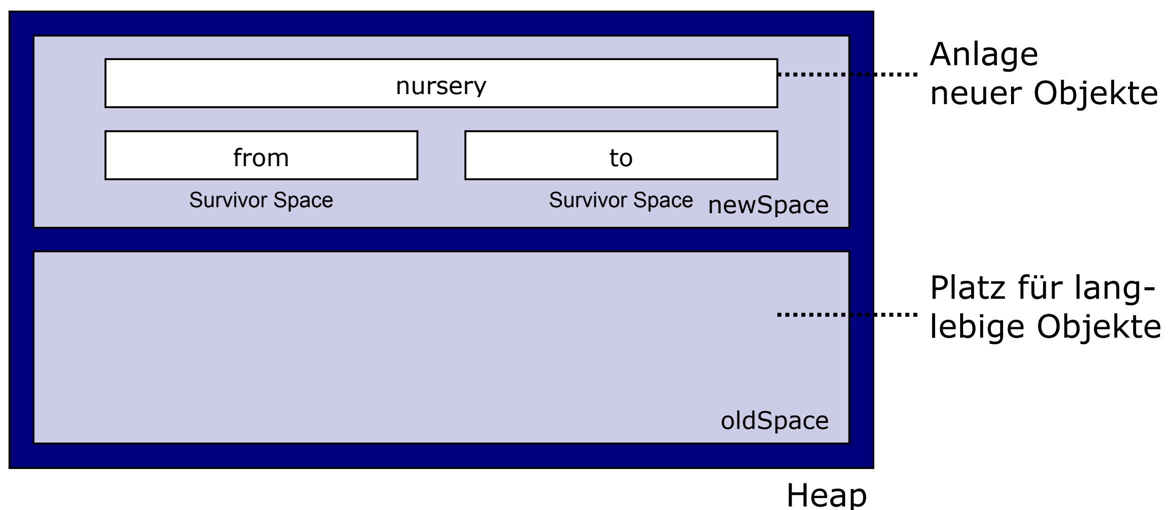


Im nächsten Schritt wird wiederum der erste Waggon bereinigt. Es ergibt sich folgendes Bild:



Garbage Collection in Java Hotspot VM 1.3.1

Nach der Beschreibung der Algorithmen möchte ich nun auf den Java Garbage Kollektor eingehen und auf die Art und Weise, wie er die oben beschriebenen Algorithmen einsetzt. Der Java Garbage Kollektor verwendet mehrere Verfahren. Zum einen verwendet der Java Garbage Kollektor zwei Generationen (einen newSpace und einen oldSpace) und zum anderen werden in diesen Generationen unterschiedliche Garbage Collection Verfahren eingesetzt. Der Aufbau des Heaps sieht folgendermaßen aus:



Der newSpace wird mit dem Stop and Copy Verfahren bereinigt. Die neuen Objekte werden hierbei immer in der nursery angelegt. Das heißt, in der nursery befinden sich immer die jüngsten Objekte. Ist die nursery voll belegt, so beginnt der Garbage Collector zu laufen. Hierbei werden die Objekte nach dem Stop and Copy Verfahren von der nursery und dem fromSpace in den toSpace kopiert. Anschließend werden from und to vertauscht und das Ganze beginnt von neuem. Nach einer gewissen Anzahl von Kopiervorgängen werden die Objekte in den oldSpace verschoben.

Die Verwaltung des oldSpace erfolgt entweder nach dem Mark and Compact Verfahren oder dem Train-Algorithmus. Das default-Verfahren ist der Mark and Compact Algorithmus. Möchte man, dass der Train-Algorithmus verwendet wird, so kann man dies durch Übergabe des Kommandozeilenparameters `–Xincgc` erreichen.

Anmerkung

„from" und „to" werden auch als „Survivor Space#1" und „Survivor Space#2" bezeichnet. Die „nursery" trägt auch noch den Namen „eden".

Ein Problem, das Garbage Kollektoren haben, die Generation Scavenging einsetzen, sind die Zeiger zwischen den Generationen. Zum einen gibt es hier Zeiger von der jungen Generation in die alte und zum anderen Zeiger von der alten Generation in die junge. Diese Zeiger zwischen den Generationen müssen in der jeweils anderen Generation als zusätzliche Wurzelzeiger behandelt werden. Der Java Garbage Kollektor löst das Problem folgendermaßen:

- Um die Zeiger von der jungen Generation in die alte zu kennen, wird einfach vor jeder Garbage Collection in der alten Generation, eine Garbage Collection in der jungen Generation durchgeführt. Bei diesem Garbage Collection Lauf werden dann alle Zeiger von der jungen Generation in die alte gesammelt.
- Die Zeiger von der alten Generation in die junge sind nicht ganz so einfach zu handhaben. Um dieses Problem zu lösen, wird der gesamte oldSpace in Teilbereiche (so genannte Cards) zu 512 Bytes zerteilt. Für jede solche Karte verwaltet man ein Bit, das anzeigt, ob ein schreibender Zugriff in dieser Karte stattgefunden hat (diese schreibenden Zugriffe werden mit so genannten Write Barriers abgefangen). Findet ein solcher schreibender Zugriff statt, so wird das Bit für diese Karte gesetzt (auf dirty). Beim nächsten Lauf des Garbage Kollektors in der jungen Generation werden all jene Karten nach Zeigern in die junge Generation durchsucht, deren dirty Bit gesetzt ist. Dieses Verfahren wird als Card Marking Scheme bezeichent.

Garbage Collection ab Java Hotspot VM 1.4.1

Damit der Java Garbage Kollektor in parallelen Anwendungen nicht zum Bottleneck wird, gibt es seit der Java Hotspot VM 1.4.1 zwei zusätzliche parallele Garbage Collection Verfahren. Für die junge Generation gibt es hierfür den so genannten „Parallel Collector". Dieser Parallel Collector stellt ein paralleles Stop and Copy Verfahren dar. Für die alte Generation gibt es einen „Concurrent Mark and Compact Collector". Wie der Name bereits erahnen lässt, handelt es sich hierbei um ein paralleles Mark and Compact Verfahren. Um die parallelen Verfahren zu aktivieren, muss als Kommandozeilenparameter `-XX:+UseParallelGC` übergeben werden.

Anpassung/Optimierung des Heaps

Der Heap kann durch zahlreiche Kommandozeilenparameter an die eigenen Bedürfnisse angepasst werden. Gerade in großen Anwendungen kann die Optimierung des Heaps zu einem beträchtlichen Performance-Gewinn beitragen. Um optimieren zu können, ist es wichtig, das Verhalten des Garbage Kollektors zu kennen. Hierbei ist der Kommandozeilenparameter `-verbose:gc` hilfreich. Übergibt man diesen Parameter, so wird bei jeder Garbage Collection ein Output erzeugt, der Aufschluss darüber gibt, welche Garbage Collection durchgeführt wurde (im newSpace oder eine gesamte Garbage Collection), wie viel Platz vor und nach der Garbage Collection von lebendigen Objekten belegt wird und wie lange die Garbage Collection gedauert hat. Ein Output könnte hier folgendermaßen aussehen:

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628->83769K(776768K), 1.8479984 secs]
```

Hierbei haben zwei Garbage Collection Läufe im newSpace stattgefunden und eine gesamte Garbage Collection. Die Werte vor und nach den Pfeilen beschreiben, wie viel Platz vor und nach der Garbage Collection von lebendigen Objekten belegt wird. In Klammern wird die Gesamtgröße des Heaps ausgegeben, wobei dies die Gesamtgröße minus die Größe eines survivor spaces ist, da ja ein survivor space zu einem bestimmten Zeitpunkt jeweils unbenützt wird.

Mit dem Kommandozeilenparameter `-Xloggc:file` kann die Ausgabe des Garbage Kollektors in eine Datei erfolgen, die durch file spezifiziert wird.

Heap-Parameter

Der Heap kann durch eine Vielzahl an Parametern angepasst werden. Zum einen gibt es Parameter, die die gesamte Heapgröße betreffen. Zum anderen gibt es Parameter, die nur die junge Generation betreffen und die Aufteilung innerhalb der jungen Generation.

<u>Parameter, die die gesamte Heapgröße betreffen</u>

```
-Xms
-Xmx
-XX:MinHeapFreeRatio
-XX:MaxHeapFreeRatio
```

`-Xms` stellt die minimale Heapgröße (und damit auch gleichzeitig die Heapgröße zu Beginn) dar und `-Xmx` stellt die maximale Heapgröße dar.

Beispiele

```
-Xms6291456 //Angabe in Bytes
-Xms6144k //Angabe in KB
-Xms6m //Angabe in MB
```

Der Wert von `-Xms` muss ein Vielfaches von 1024 Bytes sein und größer als 1 MB. Der default-Wert beträgt 2 MB.

```
-Xmx83886080 //Angabe in Bytes
-Xmx81920k //Angabe in KB
-Xmx80m //Angabe in MB
```

Der Wert von `-Xmx` muss ebenfalls ein Vielfaches von 1024 Bytes sein und größer als 2 MB. Der default-Wert beträgt 64 MB.

Aus der Tatsache, dass es eine minimale und maximale Heapgröße gibt, ergibt sich, dass der Heap zur Laufzeit wachsen oder schrumpfen kann. Wie sich die Heapgröße verändert, bestimmen die beiden Kommandozeilenparameter `-XX:MinHeapFreeRatio` und `-XX:MaxHeapFreeRatio`. Die JVM versucht nämlich das Verhältnis von freiem Platz zu lebendigen Objekten innerhalb eines gewissen Rahmens zu halten. Dieser Rahmen wird durch die oben erwähnten Parameter gesetzt. Setzt man die minimale und maximale Heapgröße auf den gleichen Wert, so ist die Größe des Heaps fix und kann sich nicht verändern.

Anmerkung

Wenn man keine Probleme mit Programmunterbrechungen hat (die durch den Garbage Kollektor verursacht werden), sollte man möglichst viel Speicher dem Heap zur Verfügung stellen. Für große Anwendungen ist der default-Wert von 64 MB oft zu klein.

Parameter, die die junge Generation betreffen

```
-XX:NewRatio
-NewSize
-MaxNewSize
-XX:SurvivorRatio
```

`-XX:NewRatio` bestimmt das Größen-Verhältnis zwischen der jungen und der alten Generation. Stellt man `-XX:NewRatio` beispielsweise auf den Wert 3, so ist das Verhältnis zwischen junger und alter Generation mit 1:3 festgelegt (der jungen Generation wird also ¼ der gesamten Heapgröße zugewiesen).

Die Werte `NewSize` und `MaxNewSize` legen die untere bzw. obere Grenze des newSpace fest. Genauso wie bei den Parametern `-Xms` und `-Xmx` kann man die Größe der jungen Generation fix festlegen, indem man die Werte `NewSize` und `MaxNewSize` auf den gleichen Wert setzt.

Weiters kann man die Größe von to und from (also den beiden SurvivorSpaces) festlegen mit dem Parameter `-XX:SurvivorRatio`. Setzt man diesen Wert beispielsweise auf 6, so beträgt das Verhältnis zwischen der nursery und einem survivorSpace 1:6 (das heißt die nursery ist acht mal so groß wie ein survivorSpace - nicht sieben mal so groß, da es ja zwei survivorSpaces gibt).

Schlussbemerkung

Wie dieser Seminarbericht zeigt, gibt es eine Vielzahl an Verfahren für Garbage Collection mit unterschiedlichen Vor- und Nachteilen. Der Java Garbage Kollektor vereinigt mehrere Verfahren und damit auch die Vorteile dieser Verfahren.