Fachhochschule Köln
Cologne University of Applied Sciences

Garbage Collection

Wahlpflichtfach Compilerbau

WS 2009/2010

Student:

Christian Funk, 11052106, ChristianFunk84@web.de

Inhaltsverzeichnis

1.0 Einleitung:

Diese Dokumentation wurde im Rahmen des Wahlpflichtfaches Compilerbau im Wintersemester 2009/2010 angfertigt.

Im Folgenden soll das Verfahren der Garbage Collection näher erläutert werden. Garbage Collection bedeutet dabei wörtlich übersetzt soviel wie "Abfall Ansammlung". Gemeint ist damit die Ansammlung von Daten im Speicher, welche nicht mehr benötigt werden und daher gelöscht werden können, um wieder Platz für neue Daten zu schaffen.

Gargabe Collection stellt dafür einige Algorithmen zur Verfügung, welche diese nicht mehr benötigten Datenblöcke automatisch auffindet, entfernt und den Speicher wieder freigibt. Es gibt allerdings auch viele Programmiersprachen, wie C/ C++, wo der Programmierer sich manuell um die Speicherbereinigung kümmern muss.

Ohne manuelle oder automatische Speicherbereinigung würde der Speicherbedarf durch jede allokierte Variable immer weiter ansteigen. Da der Speicherplatz aber durch eine endliche Anzahl von Bytes begrenzt ist, würde er theoretisch irgendwann voll sein und es könnten keine neuen Daten mehr zwischengespeichert werden.

Da es für das Verständnis der Garbage Collection wichtig ist zumindest grundlegend zu verstehen, wie der Speicher überhaupt aufgebaut und benutzt wird, behandelt der erste Abschnitt erst einmal kurz den Speicheraufbau, um daraufhin in die Algorithmen der Garbage Collection einzusteigen.

Ab Kapitel 5.0 wird dann darauf eingegangen, wie bei Java die Garbage Collection funktioniert und welche Besonderheiten es dabei gibt.

2.0 Speicherallokation:

Im folgenden Abschnitt soll ein kurzer Überblick über den Speicher gegeben werden.

Bei der Speicherzuweisung wird hauptsächlich zwischen zwei Bereichen unterschieden:

- **Statischer Bereich:** Code und statische Daten
- **Dynamischer Bereich:** Stack und Heap

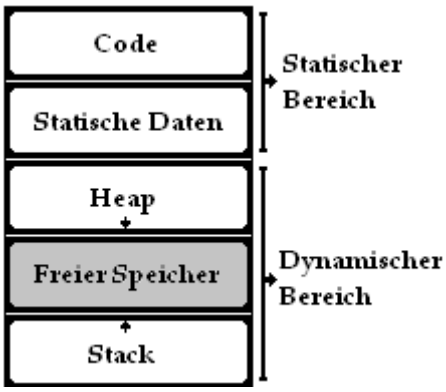


Abbildung 1: Aufbau des Speichers

2.1 Statischer Bereich:

Der statische Bereich zeichnet sich dadurch aus, dass Daten die hier abgelegt werden noch zur Kompilierungszeit bestimmt werden können, indem der Compiler sich nur den Programmcode anschaut. Dazu gehört zum einen der Code des Quellprogramms und auch statische Daten, wie beispielsweise globale Variabeln, wo die Größe schon zur Kompilierungszeit ermittelt werden kann.

Das Speichern von möglichst vielen statischen Daten zur Kompilierungszeit hat den Vorteil, dass der Speicher für diese Variabeln nicht mehr zur Laufzeit alloziert werden muss und dadurch rechenzeit eingespart werden kann.

2.2 Dynamischer Bereich:

Beim dynamischen Bereich unterscheidet man zwischen dem Stack- und dem Heap-Speicher. Beide Speicherbereich liegen gewöhnlich am entgegengestzten Ende des Speichers und wachsen, wie in Abbildung 1 dargestellt, aufeinander zu. Der Unterschied zum statischen Speicher ist, dass Daten, welche hier abgelegt werden, können erst zur Laufzeit bestimmt werden.

Stackspeicher:

Der Stackspeicher dient zur Speicherung von lokalen Daten und einigen Zusatz-informationen wie Rücksprungadressen, welche temporär von Prozeduren benutzt werden. Der benötigte Speicherbereich für die lokalen Daten wird beim Aufruf einer Prozedur belegt und beim Verlassen der Prozedur wieder freigegeben. Zudem werden beispielsweise noch Daten wie Registerwerte in den Stack gerettet, damit das Programm nach dem Verlassen einer Prozedur wieder korrekt weiter arbeiten kann.

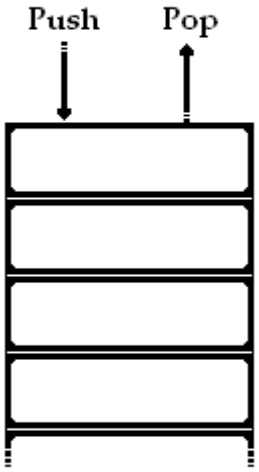


Abbildung 2: Arbeitsweise eines Stacks

Ein Stack arbeitet nach dem "Last In – First Out" (LIFO) Prinzip. Das heisst einfach, dass Daten immer ganz oben auf den Stack abgelegt werden und auch nur oben wieder weg genommen werden können. Man kann sich das wie eine Art Stapel von Tellern vorstellen. Ein Stack bietet dafür normalerweise die drei Prozeduren peek, push und pop. Mit peek wird auf ein beliebiges Element im Stack geschaut, ohne dieses zu Verändern. Mit push und pop können Daten auf den Stack abgelegt und heruntergenommen werden.

Heap-Speicher:

Der Heap-Speicher ist der zweite dynamische Speicherbereich. Hier werden Daten abgelegt, welche nicht nur für eine Prozedur, sondern beliebig lange, gültig sein können. Bei Java beispielsweise werden diese Objekte mit dem Schlüsselwörtern *new, anewarry, newarry* und *multianewarry* erstellt.

Da es bei Objekten auf dem Heap nicht so einfach ist vorherzusagen, wann ein Objekt nicht mehr gebraucht wird und der Speicher wieder freigegeben werden kann, müssen diese Objekte entweder bei einigen Programmiersprachen wie C/ C++ manuell gelöscht werden oder wie bei beispielsweise bei Java durch Algorithmen der Garbage Collection.

3.0 Warum Garbage Collection?

Wie bereits kurz in der Einleitung erwähnt, ist die Garbage Collection für die automatische Speicherbereinigung zuständig. Dafür muss der Garbage Collector im wesentlichen zwei Aufgaben erfüllen:

- Erstens: Auffinden von Garbage
- Zweitens: Freigeben des Speicherplatzes, damit er wieder zur Verfügung steht

Erstens: Auffinden von Garbage:

Um "Garbage" aufzufinden, gibt es viele verschiedene Algorithmen, wovon einige im Abschnitt 4.0 näher erläutert werden. Die Algorithmen definieren "Garbage" alle nach dem Prinzip:

"An object is considered garbage when it can no longer be reached from any pointer in the running program." (http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html)

Objekte, die nicht mehr erreichbar sind, können auch nicht mehr vom Programm angesprochen werden. Der Garbage Collector muss solche Speicherbereiche erkennen können.

Das Programm verwaltet selbst eine Reihe von Wurzeln (Rootset), auf die das Programm immer Zugriff hat. Jedes erstellte Objekt wird an eine der Wurzeln drangehangen und kann selbst auch wieder Referenzen auf weitere Objekte haben. Nicht mehr erreichbare Objekte sind dann solche, welche quasi nicht mehr "Rückwärts" bis zu ihren Wurzelknoten traversieren können. Wurzeln bei Java beispielsweise sind Daten auf dem Stack und globale Variabeln.

Zweitens: Freigeben des Speicherplatzes, damit er wieder zur Verfügung steht:

Nach dem Auffinden von nicht mehr referenenzierten Speicher, soll dieser natürlich wieder freigegeben werden. Auch das wird durch unterschiedliche Algorithmen durchgeführt. Dabei kommt aber noch ein weiterer Aspekt der Speicherbereinigung ins Spiel: Die Fragmentierung des Speichers.

Wenn der Heap noch nicht benutzt wurde besteht der Speicher aus einem zusammenhängenden freien Bereich. Unterschiedliche Datentypen benötigen aber eine unterschiedlich große Anzahl an Bytes im Speicher. Einfache Algorithmen, die immer nur einzelne Objekte aus dem Speicher heraus löschen würden, würden Lücken entstehen lassen.

Es ist daher wünschenswert, dass dieser fragmentierte Speicher wieder defragmentiert wird, weil dadurch die Lokalität verbessert wird und die Allokation von Speicher effektiver ausführbar ist. Bei defragmentiertem Speicher kann ein Objekt normalerweise sofort ans Ende alloziert werden, ohne erst den kompletten Speicher nach Lücken absuchen zu müssen.

3.1 Garbage Collection vs. Manuelle Speicherfreigabe:

Der Garbage Collector befreit den Programmierer von der Arbeit, sich selbst um die manuelle Speicherbereinigung kümmern zu müssen. Das hat den Vorteil, dass mehr Zeit zur Verfügung steht, um sich auf die eigentliche Programmieraufgabe zu konzentrieren.

Ein weiterer Vorteil der automatischen Speicherebereinigung ist, dass die manuelle Speicherfreigabe fehleranfällig sein kann. Es wird hierbei zwischen zwei Fehler-quellen unterschieden:

- Zum einen könnte ein Speicherbereich zu früh freigegeben werden, obwohl auf diesen noch Referenzen existieren. Das hat zur Folge, dass dieser Speicherbereich beispielsweise wieder mit neuen Daten belegt werden kann. Wenn das Programm nun wieder über die Referenzen auf diesen Speicherbereich zugreift, ist das Ergebnis nicht vorhersagbar. Solch ein Zeiger auf freigegebenen Speicher wird als **Dangling Pointer** (Hängender Zeiger) bezeichnet.
- Der zweite Fehler wäre es einen Speicherbereich gar nicht freizugeben, beispielsweise weil sich der Programmierer nicht sicher ist, ob ein Objekt später noch gebraucht wird. Das kann zu **Memory Leaks** (Speicherlücken) führen. Speicherlücken sind Bereiche, welche weder gelöscht noch dereferenziert werden können. Bei kleineren Anwendungen und häufigem Neustarten der Hardware, kann dieses Problem zwar oft harmlos sein, weil bei einem Neustart der flüchtige Speicher seine Ladung verliert und damit automatisch gelöscht wird. Problematisch kann dieses Problem aber beispielsweise bei Servern im Dauereinsatz, welche riesige Datenmengen verarbeiten werden.

Die automatische Speicherbereinigung hat aber auch einige Nachteile. Es wird zum einen ein gewisser Overhead erzeugt, welcher die Performance des Programms verringern kann und zum anderen hat der Programmierer keine Kontrolle über die Ausführungszeiten der Garbage Collection.

Es kann bei Java beispielsweise weder gesagt werden, wann die Garbage Collection startet, noch wann sie fertig ist. Es gibt zwar die Methode System.gc(), diese stellt aber eher eine Empfehlung an die Garbage Colletion dar.

4.0 Algorithmen:

4.1 Reference Counting:

Beim Reference Counting werden, wie der Name bereits vermuten lässt die Referenzen, die ein Objekt auf dem Heap besitzt gezählt. Wenn ein Objekt erstellt wird und durch eine Instanz referenziert wird, wird der Zähler für dieses Objekt mit dem Wert eins initialisiert.

Inkrementiert wird der Zähler beispielsweise durch einen Prozeduraufruf mit dem Objekt als Parameter oder durch eine weitere Referenzzuweisung. Für jede Referenz, welche weg fällt, weil beispielsweise der Variable eine neue Referenz zugewiesen wird, wird der Zähler um eins dekrementiert.

Erreicht der Zähler für ein Objekt den Wert null, bedeutet dies, dass es keine Referenzen mehr auf dieses Objekt gibt. Es ist also nicht mehr erreichbar und kann vom Garbage Collector entfernt werden.

Der Vorteil dieser Methode ist, dass der Garbage Collector keine Pausen einlegen muss, weil Objekte, wo der Referenzzähler auf null fällt, sofort erkannt und gelöscht werden können. Es wird also inkrementell gearbeitet.

Das Verfahren hat aber leider einige Nachteile. Zum einen ist es recht Overhead lastig, weil permanent für jedes Objekt mitgezählt werden muss und ein weiterer Nachteil ist zudem, dass dieses Verfahren nicht in der Lage ist Zykel zu erkennen.

Abbildung 3 zeigt ein Beispiel für solch einen Zykel. Wobei der erste Knoten eine Referenz über seinen Enkel hat. Dadurch hat der erste Knoten mindestens einen Zählerwert von 1 und wird nicht als Garbage erkannt, obwohl er vom Programm nicht mehr erreichbar ist.

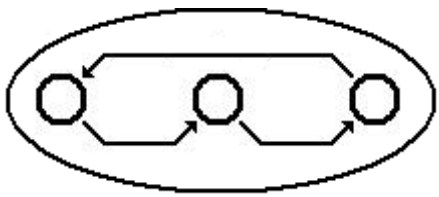


Abbildung 3: Beispiel eines einfachen Zykels.

4.2 Mark and Sweep:

Der Mark and Sweep Algorithmus gehört zu den ältesten Garbage Collection Algorithmen und zählt zu den so genannten Tracing Algorithmen. Im Gegensatz zum Reference Counting wird hierbei nicht für jede Variable mitgezählt, wie häufig diese noch referenziert wird. Dadurch entfällt der Overhead, welcher bei Reference Counting durch das Speichern der Variabeln und ständige mitzählen entsteht.

- Wenn festgestellt wird, dass nicht mehr genügend Speicher zum Allozieren neuer Datenobjekte zur Verfügung steht wird der Algorithmus gestartet.
- Solange der Algorithmus läuft wird das komplette Programm angehalten, bis der Algorithmus vollständig durchgelaufen ist. Das kann leider gerade bei Echtzeitanwendungen problematisch sein.

Der Algorithmus arbeitet in zwei Phasen:

Erstens: Die "Mark Phase":

In der Mark Phase werden von den Wurzelknoten des Programms aus alle erreichbaren Knoten (Daten) traversiert. Dabei wird jeder besuchte Knoten als "live" (lebendig) markiert. Alle anderen Knoten bleiben unmarkiert. Im Gegensatz zum Reference Counting entstehen hierbei keine Probleme mit zykulären Datenstrukturen.

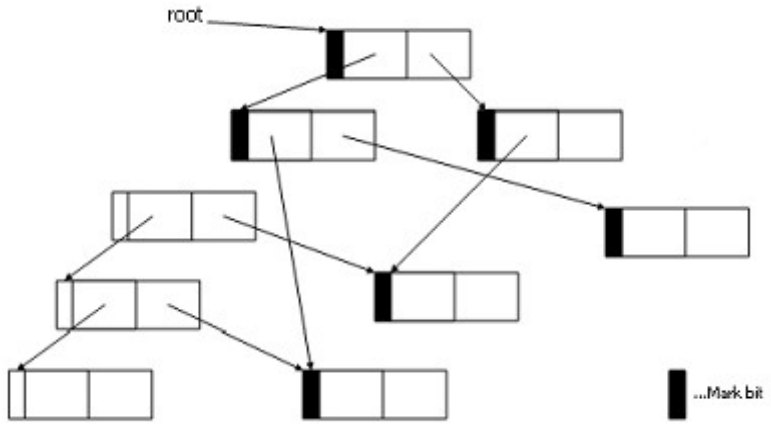


Abbildung 4: Mark Phase

In der ursprünglichen Version des Algorithmus geschah dieses markieren durch rekursive Tiefensuche. Dies hatte den Nachteil, dass für die Rekursionen wieder der Stack benutzt werden musste und es somit zu einem Stack Overflow kommen konnte.
Später wurden noch einige modifizierte Algorithmen, wie beispielsweise der "Schorr-

Waite-Deutsch" Algorithmus entwickelt. Dieser Algorithmus arbeitet iterativ. Dabei kommt ein "Reversal Pointer" Verfahren zum Einsatz. Die Idee dahinter ist, dass der Algorithmus sich möglichst einfach merken soll, wie er von seinen Kindknoten wieder zu seinen Elternknoten kommen kann. Hierzu werden die Referenzen von den Vater auf die Kindknoten temporär beim traversieren umgedreht.

Somit ist es später einfach möglich, wieder zu den Wurzelknoten zurück zu traversieren. Diese Version funktionierte aber nur mit binären Bäumen und wurde auf eine Laufzeit von O(n+1) berechnet. Wodurch sie leider nicht sehr effizient arbeitet. Es gibt daher auch noch einige andere Verfahren, auf die im Dokument aber nicht genauer eingegangen werden.

Zweitens: Die "Sweep Phase":

In der Sweep Phase wird der Heap Speicher linear Zelle für Zelle durchlaufen. Dabei wird der Speicherplatz der Objekte, welche nicht markiert sind wieder zur Verfügung gestellt. Zudem werden die Markierungen der lebendigen Objekte wieder entfernt.

4.3 Mark and Compact:

Der beschriebene Mark and Sweep Algorithmus hat noch einen weiteren Nachteil. Durch das Löschen vereinzelter Daten auf dem Heap entsteht eine Fragmentierung des Speichers.

Daher gibt es noch eine ähnliche Variante des Algorithmus, welcher in drei Phasen arbeitet:

1. Markierungsphase, wie bei Mark and Sweep, wobei die lebendigen Objekte markiert werden.
2. Kompaktierung: Schliessen der Lücken im Heap durch Verschieben der einzelnen Objekte.
3. Zeiger-Anpassung: Anpassen der Referenzen aus der Wurzelmenge und dem Heap selbst und gleichzeitiges Zurücksetzen der Markierungen.

Im Gegensatz zum Mark and Sweep Algorithmus werden bei diesem Verfahren die erreichbaren Objekte an eine neue Position kopiert und damit wird der Speicher wieder defragmentiert.

4.4 Stop and Copy:

Der Stop and Copy Algorithmus gehört ebenfalls zu den Tracing Algorithmen. Der Heap Speicher ist bei diesem Verfahren in eine jeweils freie und eine benutzte Seite eingeteilt. Dem so genannten "from space" und "to space".

Wenn nicht mehr genügend Speicher zur Verfügung steht, beginnt der Algorithmus seine Arbeit und kopiert nun überlebende Daten vom "from space" zum "to space". Dazu müssen auch wieder die Referenzen auf die neuen Adressen angepasst werden.

Objekte die nach diesem Kopiervorgang im from space zurückgeblieben sind, gelten nun als Garbage und können freigegeben werden. Bei jedem weiteren Durchlauf des Algorithmus wird einfach die Seite des from und to space vertauscht.

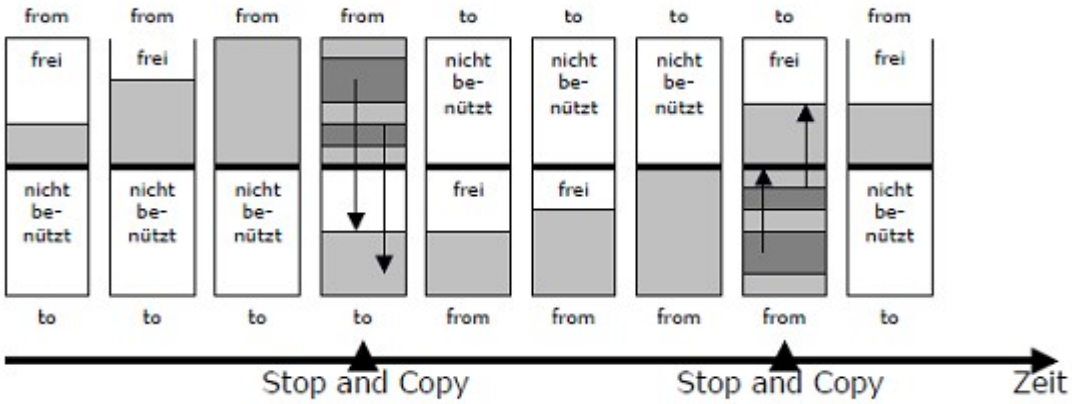


Abbildung 6: Stop and Copy

Der Vorteil des Stop and Copy Algorithmus ist, dass er der Speicherfragmentierung entgegen wirkt. Leider hält aber auch dieser Algorithmus, dass Programm während der Ausführung an. Außerdem wird bei diesem Verfahren doppelt so viel Heap Speicher benutzt, da immer eine Hälfte als to space zum Kopieren bereit stehen muss.

Die Laufzeit des Algorithmus hängt von der Anzahl der zu kopierenden Objekte ab. Es kann daher recht zeitaufwändig sein große Datenmengen immer von der einen zur anderen Seite zu kopieren.

4.5 Einleitung: Generationelle Garbage Collection:

Bisher haben alle vorgestellten Algorithmen darauf basiert den kompletten Speicher zu durchsuchen.

Es wurde aber empirisch festgestellt, dass über 80% - 98% der angelegten Objekte sehr jung sterben und nur sehr wenige Objekte lange überleben. *(Weak Generational Hypothesis)* Die meisten Objekte sterben in dieser Zeit, bevor wieder ein freier Megabyte Speicher auf dem Heap alloziert werden konnte.

Ein Beispiel dafür sind Iteratoren, welche nur kurz benutzt werden und dann wieder gelöscht werden können. Dagegen gibt es einige Objekte, welche beispielsweise beim Programmstart initialisiert werden und dann bis zum Ende des Programms bestehen bleiben müssen.

Es ist daher Sinnvoll nicht immer den kompletten Speicher zu durchsuchen, sondern den Speicher in junge und ältere Generationen einzuteilen. Genauer wird dieses Verfahren im folgenden Abschnitt 5.0 Java: Garbage Collection erläutert.

5.0 Java: Garbage Collection:

Die Java HotSpot Virtual Machine beinhaltet vier Garbage Collectoren, welche alle Generationell arbeiten. (Stand: J2SE 5.0 Update 6). Die vier Collectoren heissen "Serial Collector", "Parallel Collector", "Parallel Compacting Collector" und "Concurrent Mark-Sweep (CMS) Collector". Im Folgenden wird auf den Serial und Parallel Collector eingegangen.

Der Speicher wird von der JVM in drei Generationen "Young Generation", "Old Generation" und "Permanent Generation" eingeteilt. Die Permanent Generation beinhaltet Objekte, welche von der JVM geeignet erscheinen, um den Garbage Collector diese verwalten zu lassen. Das sind Beispielsweise Objekte, welche Klassen und Methoden beschreiben, sowie die Klassen und Methoden selbst.

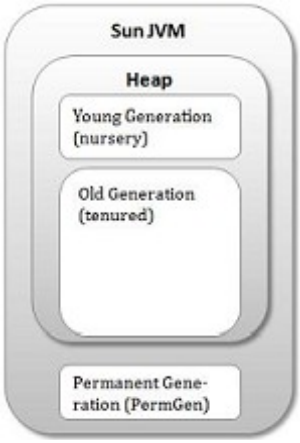


Abbildung 7: Interner Heap Aufbau

Die meisten Objekte werden zuerst in der Young Generation erstellt. In Abbildung 8 ist dieser Schritt am Pfeil "Allocation" zu sehen. Dieser Bereich ist üblicherweise kleiner als die Old Generation, da Objekte in diesem Bereich schnell wieder sterben und der Speicher dann wieder vom Garbage Collector frei gegeben werden kann.

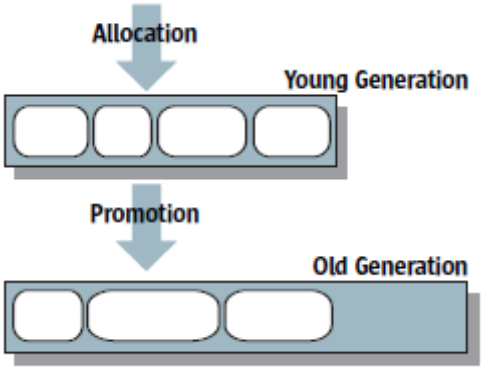


Abbildung 8: Young and Old Generation

Die Young Generation wird relativ häufig nach unreferenzierten Objekten durchsucht, weil es sehr wahrscheinlich ist, dass sich hier drin schnell wieder neuer Garbage gesammelt hat. Dazu wird ein möglichst effizienter und schneller Algorithmus benutzt.

Objekte, welche einige Zeit lang in der Young Generation überleben, haben eine Chance in die Old Generation übernommen zu werden. Dieser Schritt ist am Pfeil "Promotion" in der Abbildung 8 zu sehen.

Die Old Generation wird dagegen wesentlich seltener durchsucht, weil die Objekte, welche sich hier drin befinden eine höhere Chance haben noch referenziert zu sein. Daher wird hier auch ein anderer Algorithmus als in der Young Generation verwendet, welcher vor allem Speichereffizient sein sollte.

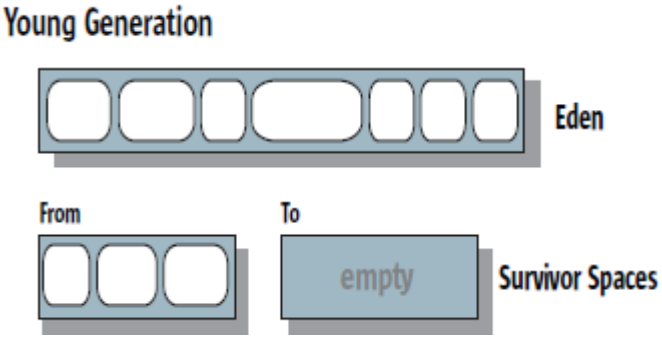


Abbildung 9: Young Generation

Die Young Generation wird selbst nochmal aufgeteilt in einen Bereich, welcher "Eden" genannt wird, sowie einen "From" und "To" Survivor Space. Neue Objekte werden normalerweise zuerst im Eden alloziert. Es sei denn sie sind zu groß. Dann werden sie direkt in der Old Generation angelegt.

5.1 Serial Collector:

Der Serial Collector wird für die J2SE Version standardmässig benutzt. Er benötigt sowohl für die Young als auch für die Old Generation nur eine CPU zur Garbage Collection.

5.1.1 Young Generation:

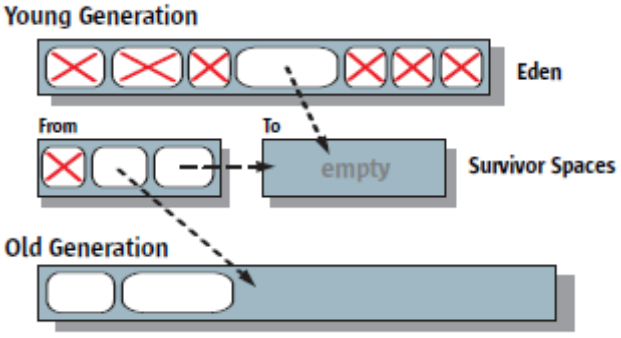


Abbildung 10: Young Generation Collection

Abbildung 10 zeigt den Ablauf der Garbage Collection:

- Die lebendigen Objekte des Eden Bereichs werden in den To Survivor Space kopiert. Es sei denn sie sind zu groß für diesen Bereich. Dann werden sie direkt in die Old Generation kopiert.
- Relativ junge Objekte des From Survivor Space werden auch erstmal noch in den To Survivor Space kopiert.
- Ältere Objekte des From Survivor Space dagegen werden in die Old Generation kopiert.
- Alle Objekte, welche im Eden und From Survivor Space übrig bleiben gelten nun als nicht erreichbar. (In Abbildung 10 durch ein X dargestellt)

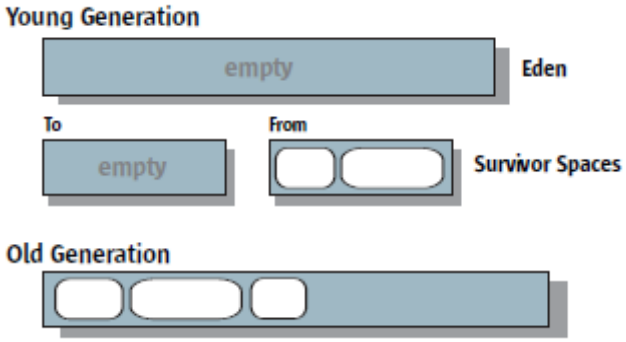


Abbildung 11: Nach der Young Generation Collection

Abbildung 11 zeigt den Speicher nach der Collection. Der Eden und frühere From Survivor Space, welcher in der Abbildung 10 noch links war sind nun leer (empty) und der From und To Survivor Space haben die Rollen getauscht.

5.1.2 Old Generation:

Die Garbage Collection in der Old Generation wird durch einen Mark, Sweep and Compact Algorithmus durchgeführt, wie in Abschnitt 4.2 und 4.3 erläutert. Abbildung 12 illustriert den Ablauf.

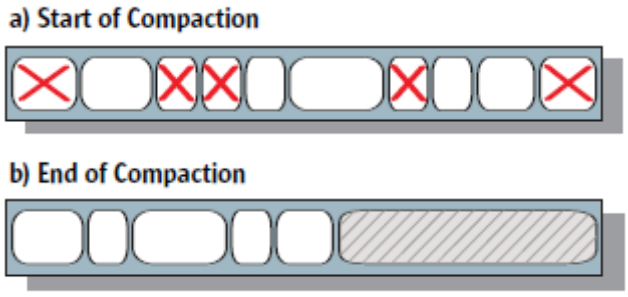


Abbildung 12: Old Generation Collection

5.2 Parallel Collector:

Der Parallel Collector hält wie auch der Serial Collector das Programm an und arbeitet nach einem Copy Verfahren. Der Unterschied ist aber, dass die Young Generation durch mehrere Prozessoren abgearbeitet wird und dadurch der Overhead durch die Collection reduziert wird.

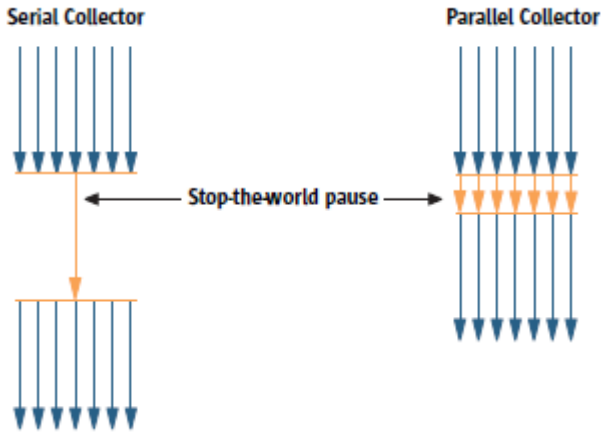
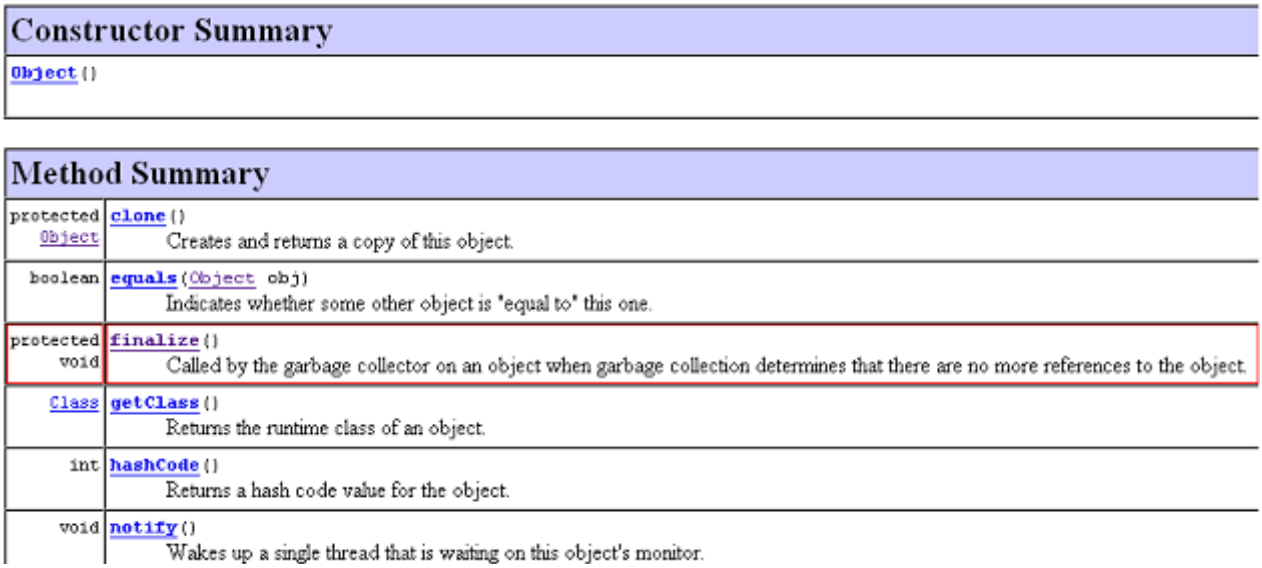


Abbildung 13: Vergleich Serial/ Parallel Collector

Die Old Generation wird jedoch nach wie vor durch einen Mark, Sweep and Compact Algorithmus abgearbeitet. Dabei kommt weiterhin nur ein einziger Prozessor zum Einsatz.

6.0 Java: Finalization:

Die Garbage Collection bei Java ist noch dadurch ein wenig verkompliziert, dass es die Methode *finalize()* gibt, welche von der Klasse Object implementiert wird.



Constructor Summary

`Object()`

Method Summary

protected Object	`clone()` Creates and returns a copy of this object.
boolean	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
protected void	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	`getClass()` Returns the runtime class of an object.
int	`hashCode()` Returns a hash code value for the object.
void	`notify()` Wakes up a single thread that is waiting on this object's monitor.

Abbildung 14: Methode *finalize()* von Object

Wenn eine Klasse die Methode *finalize()* überschreibt, muss diese zuerst vom Garbage Collector ausgeführt werden, bevor der Speicher für dieses Objekt freigegeben werden darf.

Die Methode ist eigentlich dazu gedacht, um noch letzte Aufräumaktionen durchzuführen oder um beispielsweise Eingabe-/ Ausgabeströme zu schliessen, sobald das Objekt entfernt wird. Sie wird erst dann ausgeführt, sobald ein Objekt nicht mehr referenziert wird. Da der Ausführungszeitpunkt der Garbage Collection allerdings nicht vorhersagbar ist, kann man sich nicht darauf verlassen, dass *finalize()* zu einem bestimmten Zeitpunkt aufgerufen wird.

Es ist aber auch möglich, ein Objekt durch diese Methode wieder zum Leben zu erwecken. Dazu muss in der überschriebenen *finalize()* Methode beispielsweise einfach die *this* Referenz gerettet werden.

```
class Testclass {

   protected void finalize() throws Throwable {
      otherClass.resurrectMe(this);
      super.finalize();
   }
}
```

Da es durch solch eine Konstruktion möglich ist ein Objekt, welches *finalize()* überschreibt wieder zu beleben, muss der Garbage Collector bei diesen Objekten zwei mal schauen, ob sie nun wirklich nicht mehr leben, bevor er den Speicher freigeben darf.

Aber selbst wenn ein Objekt sich dadurch wiederbelebt, wäre dies nur ein einziges mal möglich, weil die Methode *finalize()* nur ein einziges mal pro Objekt aufgerufen wird.

7.0 Java: Referenzstärken:

Eine weitere Besonderheit unter Java ist, dass es seit Java 1.2 möglich ist, Referenzen unterschiedlicher Stärke zu definieren. Die Stärke eines Objekts bezieht sich darauf, wie stark der Garbage Collector sich an die Referenz binden soll. Es gibt hierbei vier Abstufungen von Strong bis Phantom.

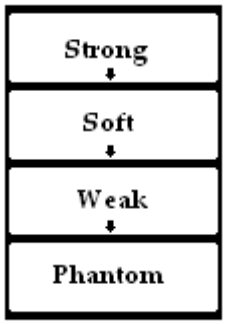


Abbildung 15: Referenz Abstufungen

Soft, Weak und Phantom References sind von der Klasse Reference abgeleitet. Um eine dieser Referenzen zu erzeugen, muss nur nur ein Objekt des entsprechenden Typs angelegt und dem Konstruktor eine Reference übergeben werden. Es ist zudem auch möglich Maps anzulegen.

Strong References:

Als Strong References werden die Referenzen bezeichnet, welche man intuitiv beim Programmieren benutzt:

```
Object x = new Object();
```

Dies erzeugt ein Datenobjekt vom Typ Object und speichert den Wert in der Variabeln x.

Strong References verhalten sich so, wie in den vorherigen Abschnitten über Algorithmen beschrieben. Sie können vom Garbage Collector erst dann eingesammelt werden, wenn es keine Referenz mehr auf diese Daten gibt.

Weak References:

Es gibt Fälle, in denen es aber nicht unbedingt wünschenswert ist, dass ein Datenobjekt erst dann gelöscht werden kann, wenn es gar nicht mehr benutzt wird. Ein Beispiel dafür wäre ein Programm, welches große Bilder im Speicher zwischenlädt damit diese nicht erst immer vom langsameren Festplattenspeicher aus geladen werden müssen.
Bei einer großen Datenbank mit Bildern, wo viele Benutzer drauf zugreifen, müsste nun theoretisch mit Strong References immer verfolgt werden, wann genau ein Bild nicht mehr benutzt wird. Dann müsste die Referenz gelöscht werden und der Garbage Collector könnte das Bild abräumen. Dieses Verfahren könnte aber auch schnell zu Speicher-problemen führen, was ein generelles Problem beim Caching von großen Daten sein kann.

Einfacher wäre es daher, dem Garbage Collector quasi autonom zu gestatten, bestimmte Referenzen auch dann abzuräumen, wenn der Speicher knapp wird. Dazu gibt es Weak bzw. Soft References, welche sich sehr ähnlich sind. Der Garbage Collector darf diese Daten auch dann abräumen und den Speicher wieder freigeben, wenn es noch Referenzen auf diese Daten gibt.

Bei der Referenzstärke gilt das Prinzip, dass ein Objekt immer so stark referenziert wird, wie die stärkste Referenz die darauf verweist. Ein Objekt, welches also sowohl über eine Strong als auch über eine Weak Reference erreichbar ist, gilt trotzdem als Strong Reference. Ein Objekt, welches aber über eine Kette von Referenzen erreichbar ist, ist nur so stark erreichbar wie die schwächste Referenze in der Kette. Beispiel:

Date ist in diesem Beispiel nur weakly reachable.

```
SoftReference sr = new SoftReference(new WeakReference(new Date()));
```

Soft Reference:

Vor Version 1.3.1 wurden Soft und Weak Referenzen fast gleich behandelt. Seitdem gibt es aber einige Unterschiede. Soft Referenzen dürfen seit Version 1.3.1 erst abgeräumt werden, wenn es keine Weak References mehr gibt. Zudem bleiben sie noch eine Sekunde für jeden freien Megabyte Speicher im Heap nach der letzten Referenzierung bestehen.

Phantom Reference:

Phantom Referenzen bilden die schwächste Form und unterscheiden sich stärker von den anderen Referentypen. Sie können nicht einmal durch die finalize() Methode wiederbelebt werden, da die get() Methode der Phantom Referenz immer null zurück liefert. Daher ist es auch nicht einmal möglich, auf das Objekt zuzugreifen.

Es gibt zwei Möglichkeiten zur Anwendung von Phantom References. Sie können erstens dazu verwendet werden, um zu bestimmen wann ein Objekt vom Speicher entfernt wurde und zweitens kann man sie als eine Art Alternative zu der Methode finalize() benutzen. Durch finalize() muss der Garbage Collector immer noch ein zweites mal schauen, ob das Objekt nun wirklich noch am Leben ist. Phantom Referenzen sind aber nicht wiederbelebbar und ersparen dadurch diesen zweiten Durchlauf.

8.0 Literaturverzeichnis:

Bücher:

[1] Compiler – Prinzipien, Techniken und Werkzeuge – 2. aktualisierte Auflage
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

[2] Java Performance Tuning - 2nd Edition
Jack Shirazi

Online:

[3] Inside the Java Virtual Machine
http://www.artima.com/insidejvm/ed2/gcP.html

[4] Memory Management in the Java HotSpot Virtual Machine
http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

[5] Documentation: Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine
http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

[6] GC FAQ -- draft
http://iecc.com/gclist/GC-faq.html