Johannes Kepler Universität Linz

Institut für Informatik
Abteilung für Systemsoftware

Seminar Inside Java und .NET
Garbage Collection unter .NET

Leitung: o. Univ.-Prof. Dr. Hanspeter Mössenböck

KAINEDER Benjamin, 0057212

Linz, 05. Februar 2004

1 Einleitung

„Garbage Collection“ bezeichnet den Vorgang, bei dem nicht benutzter, aber noch reservierter Speicher wieder für andere Programme benutzbar gemacht – freigegeben – wird. Während der belegte Speicher bei objektorientierten Systemen normalerweise sofort mit der expliziten Zerstörung eines Objektes freigegeben wird, geschieht dies bei Systemen mit Garbage Collection periodisch oder bei bestimmten Ereignissen (z.B. Speicherknappheit, oder expliziter Aufruf). Dabei werden alle unbenutzten Objekte auf einmal aus dem Speicher entfernt.

Garbage Collection findet man in sogenannten „managed code“-Umgebungen (wie Java und .NET), die eine Virtual Machine oder einen Interpreter zur Ausführung des Codes benutzen. Nur in diesen ist es möglich, den Lebenszyklus der erstellten Objekte zu verfolgen, nicht benutzte Objekte herauszufinden, sowie die Codeausführung während der Collection anzuhalten, und Objektzeiger zu ändern, wenn sich der Speicherort des Objektes auf dem Heap verändert hat.

1.1 Vorteile von Garbage Collection

- Es ist kein explizites Freigeben des Speichers mehr erforderlich: Die Laufzeitumgebung stellt fest, welche Objekte nicht mehr nutzbar für das Programm sind, und gibt diese frei. Dadurch werden gleich zwei häufige Fehlerquellen vermieden: Es kann nicht vorkommen, dass unbenutzter Speicher reserviert bleibt und dadurch für andere Programme verloren geht (sog. „memory leak“), und es kann nicht zur zu frühen Speicherfreigabe kommen, wenn ein Objekt zerstört wird, auf das später noch zugegriffen werden soll (sog. „dangling pointer“)

- Der Heap wird nicht fragmentiert: Während der Collection wird nach der Freigabe der Objekte der Speicher kompaktiert, sodass keine Speicherlücken zwischen den Objekte auftreten – der Speicher wird somit effizienter genutzt.

- Performance-Vorteil: Objekte werden am Heap nacheinander angelegt, dadurch ergibt sich ein Geschwindigkeitsvorteil für die meisten Anwendungen, da nacheinander erzeugte Objekte mit hoher Wahrscheinlichkeit aufeinander

verweisen. Durch die Nähe ihrer Speicherorte können sie gemeinsam in den Cache geladen werden. Im Gegensatz dazu werden Objekte aus „unmanaged code“ dort angelegt, wo sich der nächste freie Platz am Heap findet. Außerdem rücken durch die nachfolgende Verdichtung die langlebigen Objekte zusammen, woraus sich ebenfalls eine effizientere Cache-Nutzung ergeben kann.

2 Das Verfahren von .NET

Der von .NET verwendete Algorithmus wird als „Mark & Compact“ bezeichnet. Dabei werden in der ersten Phase alle erreichbaren Objekte gekennzeichnet, in der zweiten Phase alle nicht gekennzeichneten Objekte freigegeben, und schließlich wird der Heap verdichtet.

2.1 Mark-Phase

„Alle erreichbaren Objekte“ bezeichnet alle Objekte, die von globalen oder lokalen Referenzen bzw. von Referenzen die bereits in die CPU Register geladen wurden erreichbar sind, also auch alle indirekt zu erreichenden Objekte. In Abbildung 1 sind dies die Objekte A, C, D, F (jeweils direkt zu erreichen) und H (über D zu erreichen).

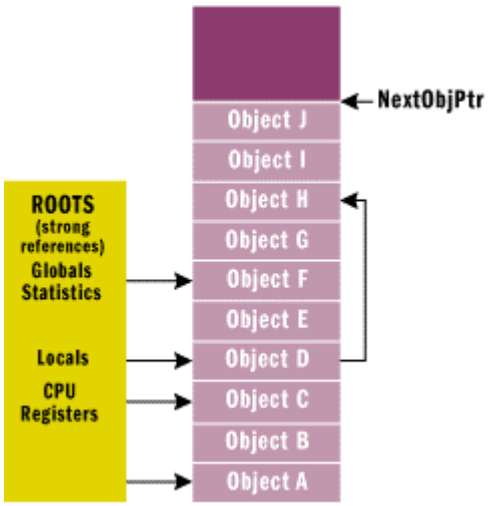

Abbildung 1: Heap vor Collection

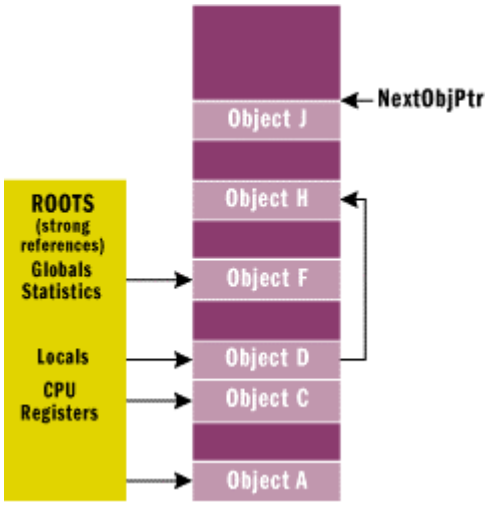

Abbildung 2: Alle nicht erreichbaren Objekte freigegeben

Zuerst werden alle Objekte als g*arbage* betrachtet (markiert), dann wird mit der Traversierung begonnen. Alle durch diese Traversierung erreichten Objekte werden als „*not garbage*“ markiert. Die übrig gebliebenen, durch das von der Laufzeit-

umgebung ausgeführte Programm nicht erreichbaren Objekte werden freigegeben (Abbildung 2).

2.2 Compact-Phase

Der Speicher hat nun Löcher, die eine Fragmentierung bewirken können, außerdem reserviert .NET nur Speicher am Ende des Objekt-Heap. Dies ist eine Vereinfachung und Performanzvorteil, da beim Neuanlegen des Objekts lediglich der Zeiger *NextObjPtr* verschoben werden muss. Deshalb werden in dieser Phase alle Objekte zusammengeschoben.

Dazu wird zunächst der neue Platz des Objektes berechnet, und alle Zeiger, die auf es verweisen, geändert. Dies erfordert detaillierte Kenntnis über die Objektstrukturen – ein gewichtiger Grund, weshalb Garbage Collection nur in *managed code*-Umgebungen möglich ist. Wenn die Neuberechnung aller Referenzen abgeschlossen ist, werden die Speicherblöcke umkopiert, sodass die Löcher verschwinden.

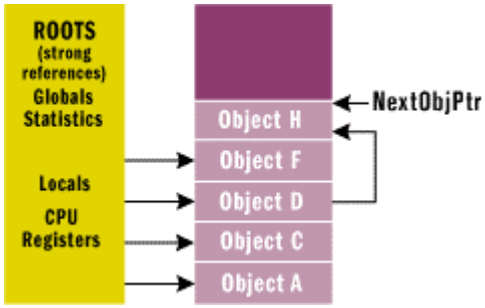

Abbildung 3: Heap nach Verdichtung

2.3 Optimierung durch Generationen

Die Garbage Collection selbst ist ein zeitaufwändiger Vorgang, insbesondere muss das Programm währenddessen angehalten werden, damit alle Referenzen bestimmt und im Folgenden angepasst werden können. Deshalb versucht man, den gesamten Vorgang möglichst kurz zu halten.

Untersuchungen an verschiedenen Programmen zeigen, dass meist die folgenden Regeln gelten: [Bur02]

- Je jünger (neuer) ein Objekt ist, desto kürzer ist seine wahrscheinliche Lebenserwartung, und je älter ein Objekt bereits ist, desto länger ist seine weitere wahrscheinliche Lebenserwartung.
- Sehr junge Objekte werden häufiger verwendet als alte Objekte, und haben sehr starke Beziehungen untereinander.

Das heißt, dass bei der Verarbeitung von jungen Objekten eine hohe Wahrscheinlichkeit besteht, mehr freien Speicher zu bekommen als bei der Prozessierung alter Objekte. Dies erscheint logisch, wenn man den hohen Anteil von temporären, kurzlebigen Objekten bedenkt, der bei strikt objektorientierter Programmierweise zwangsläufig auftritt.

Die zweite Annahme bedeutet, dass die Verarbeitung junger Objekte effizienter ist, weil die inneren Referenzen nicht so weit verzweigt sind und dadurch schneller traversiert werden können.

Es ist also durchaus sinnvoll, das Alter von Objekten zu erfassen, und ältere nur bei Bedarf zu prozessieren, da hier das Kosten/Nutzen-Verhältnis geringer ist. Die Objekte werden in sogenannten Generationen zusammengefasst, wobei in Generation 0 gestartet wird. Überlebt das Objekt eine Garbage Collection, erreicht es die nächsthöhere Generation (Abbildung 5 und Abbildung 6, die rosa gekennzeichneten Objekte werden jeweils freigegeben).

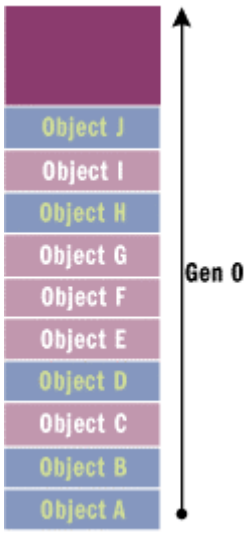

Abbildung 4: Vor der ersten Collection

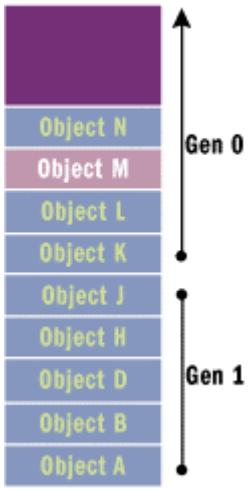

Abbildung 5: Nach der ersten Collection

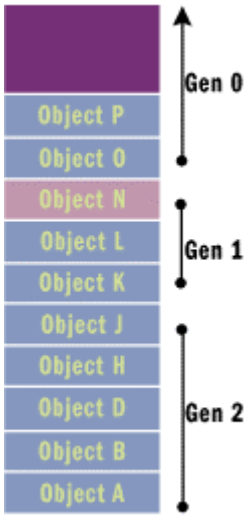

Abbildung 6: Nach der zweiten Collection

Nun kann der GC die Freigabe auf die junge Generation 0 beschränken, da hier der Wahrscheinlichkeit nach mehr Speicher freigegeben wird als in den älteren Generationen. Dazu wird die Traversierung der Objektzeiger in der Mark-Phase beschränkt auf neue Objekte in der Generation 0, und alle alten Objekte aus den anderen Generationen, auf die seit dem letzten GC-Durchlauf geschrieben wurde. Dies ist erforderlich, weil ein altes Objekt auf eines der neuen Objekte zeigen könnte, dieses darf dann natürlich nicht freigegeben werden. Unter Windows kann dies mit Kernel-Funktionen, die den jeweiligen Speicherbereich auf Schreibzugriffe überwachen, erreicht werden.

Daraus ergeben sich nun die folgenden Performance-Steigerungen:

- In der Mark-Phase müssen nicht alle Objekte traversiert und markiert werden
- Da aus den älteren Generationen (im Heap „unten“ liegend) keine Objekte freigegeben werden, müssen weniger Speicherinhalt umkopiert werden, dadurch müssen auch weniger Zeiger umgebogen werden

3 Multi-Threading

Wenn der Garbage Collector eine Collection starten will, müssen alle von der Laufzeitumgebung abhängigen Programme angehalten werden, damit die Zeiger auf die später im Speicher verschobenen Objekte umgebogen werden können. Würde das Programm währenddessen weiterlaufen, würden die im Stack und in den Registern vorhandenen Zeiger nach der Compact-Phase auf den falschen Speicherbereich zeigen.

Bei .NET-Code kann dies jederzeit erfolgen, allerdings ist für unmanaged Code (z.B. eine externe, nicht-.NET DLL, deren Objekte am .NET-Heap liegen), der von .NET ausgeführt wird, ein spezieller Mechanismus erforderlich. Dazu wird der Thread angehalten, während dieser Phase werden die Objektreferenzen durchlaufen und die freizugebenden Objekte bestimmt. Danach kann der Thread sofort weiterlaufen: Da es nicht möglich ist, auf den Speicherbereich des externen Threads zuzugreifen und damit die Zeiger anzupassen, dürfen vom Thread verwendete Objekte nicht im Speicher verschoben werden, diese werden „pinned objects“ genannt.

Die Compact-Phase wird somit stark eingeschränkt, da die *pinned objects* wie einen Block im Speicher bilden, und der Heap nicht mehr völlig fragmentfrei gehalten bleiben kann. Die von .NET verschobenen Objekte müssen um die *pinned objects* herum angeordnet werden.

4 Finalization

Finalization ist eine wichtige Ergänzung in *managed code*-Umgebungen wie .NET und auch Java. Es existiert kein echter Destructor, da der Speicher ja automatisch verwaltet wird. Manchmal ist es aber doch notwendig, Ressourcen manuell freizugeben, nämlich dann, wenn es sich um Ressourcen handelt, bei denen .NET nicht bekannt ist, wie diese korrekt abgebaut werden sollen, z.B. Netzwerkverbindungen, Dateihandles, oder Datenbankverbindungen. Durch eine Finalize-Methode wird das Objekt informiert, wenn der GC es freigeben will, und kann seinerseits die verwendeten Ressourcen freigeben.

4.1 Ablauf

Objekte, die eine Finalize-Methode implementieren, werden vom Garbage Collector unterschiedlich zu normalen Objekten behandelt: Bei der Erzeugung eines derartigen Objekts wird ein Zeiger auf dieses in die sogenannte Finalization-Queue gestellt (siehe Abbildung 7).

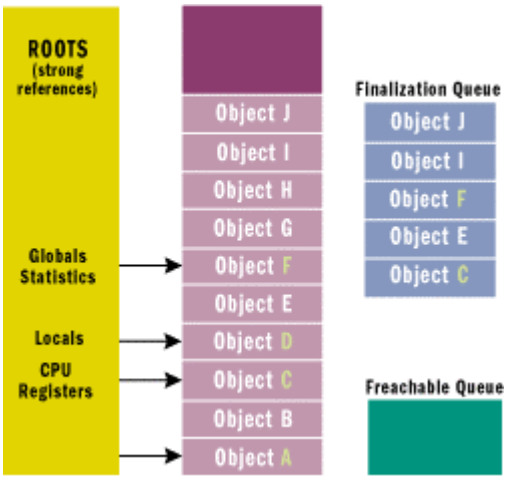

Abbildung 7: Objekte mit Finalize-Methode

Soll ein Objekt freigegeben werden, und befindet sich ein Zeiger darauf im Finalization-Queue, wird dieser in den F-reachable Queue verschoben (Abbildung

8). Objekte, deren Zeiger sich nun im F-reachable Queue befinden, können noch nicht freigegeben werden, da nun erst ihre Finalize-Methode aufgerufen werden muss.

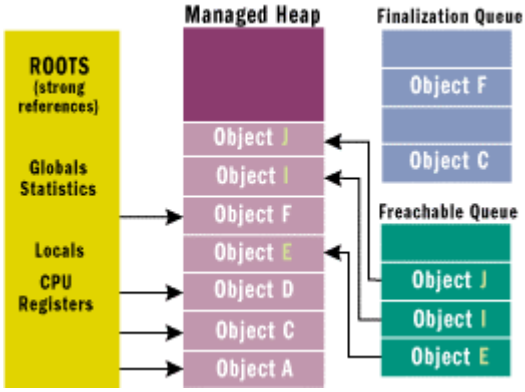

Abbildung 8: Freigeben von E, I und J

Alle anderen Objekte werden entfernt und der Speicher kompaktiert. Nun wird ein separater Thread gestartet, der die Aufgabe hat, den F-reachable Queue abzuarbeiten und die Finalize-Methoden der referenzierten Objekte aufzurufen.

Dazu wird ein separater Thread gestartet, da das Abarbeiten der Methoden je nach verwendeten Ressourcen relativ lange dauern kann (z.B. Abbau einer Datenbankverbindung). Durch den eigenen Thread kann die Garbage Collection nach dem Verschieben der Referenzen beendet werden und die gestoppten Programme können wieder weiterlaufen.

4.2 Nachteile des Verfahrens

Das hat aber zur Folge, dass die Objekte – deren Finalization ja noch nicht abgeschlossen wurde – erst im nächsten GC-Durchlauf tatsächlich freigegeben werden können (Abbildung 9). D.h., dass Objekte mit Finalize-Methode auf jeden Fall um einen Durchlauf länger als erforderlich Speicher besetzt halten und dadurch mehr verbraucht wird.

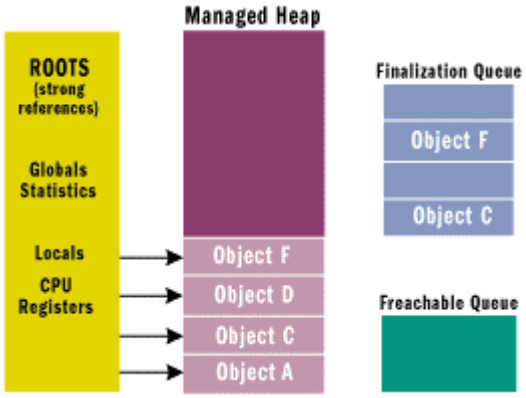

Abbildung 9: Heap nach dem nächsten GC-Durchlauf
(E, I und J wurden jetzt freigegeben)

Ein weiterer Nachteil ist, dass die Erzeugung und die Freigabe länger dauert, da in beiden Fällen Zeiger auf den beiden Queues angelegt oder kopiert werden müssen. Die Zeit, ein Objekt mit Finalize-Methode zu erzeugen, ist in etwa fünfmal so lang, wie ein Objekt ohne eine derartige Methode [Bur02].

Auch ist dieser Mechanismus kein vollwertiger Ersatz für einen Destruktor, wie er in anderen Programmiersprachen wie z.B. C++ und Object Pascal zum Einsatz kommt. Er kann nicht direkt aufgerufen werden, und die Reihenfolge der Abarbeitung durch den GC-Thread ist willkürlich, ebenso wie der genaue Zeitpunkt des Aufrufens der Methode. Es kann somit nicht garantiert werden, dass Objekt B tatsächlich erst nach Objekt A freigegeben wird. Man kann sich in der Finalize-Methode also nicht darauf verlassen, dass ein anderes Objekt – auch wenn es später erzeugt wurde oder es z.B. einer inneren Klasse des aktuellen Objekts angehört – noch existiert.

Außerdem wird die Ausführung der Finalize-Methode keinesfalls garantiert. Wird die Laufzeitumgebung beendet, wird auch der Thread für die Abarbeitung abgebrochen, unabhängig vom Status. Damit soll eine schnelle Aufräumphase garantiert werden, unter Umständen bleiben jedoch Ressourcen zurück.

Eine Finalize-Implementierung sollte also nur dann stattfinden, wenn nötig.

4.3 Dispose-Pattern

Die meisten Ressourcen sollten allerdings möglichst früh wieder freigegeben werden, und nicht erst zum nächsten Garbage Collector-Durchlauf. Einerseits weil z.B. Ressourcen kapp sind (z.B. Datenbankverbindungen) oder deren Aufrechterhaltung Rechenleistung und Bandbreite verbraucht (Datenbank- und

Netzwerkverbindungen). Auch können Probleme bei unteilbaren Ressourcen auftreten (z.B. exklusives Dateihandle), weil der tatsächliche Freigabezeitpunkt nicht vorhersagbar ist.

Die Lösung ist naheliegend: Es wird eine manuell aufrufbare Freigabe-Methode implementiert, die vom verwendenden Programmierer zur Freigabe der Ressourcen genutzt werden soll. Vergisst dieser darauf oder kommt es durch einen Fehler zur Nicht-Freigabe, springt die Finalize-Methode praktisch als Backup ein, indem dort die Freigabe auf jeden Fall durchgeführt wird.

Für diesen Ansatz gibt es eigens das *IDisposable*-Interface, das die *dispose()*-Methode definiert (oftmals wird auch *close()* als zusätzlicher Methodenname implementiert, insbesondere bei Dateiressourcen). Diese soll als Hauptfreigabe-methode dienen. Nach der erfolgten Freigabe der Ressourcen kann der Garbage Collector angewiesen werden, den Zeiger im Finalization-Queue zu entfernen – die Behandlung bei der Freigabe des Objekts verläuft somit wie bei Objekten ohne Finalize-Methode, diese wird nicht mehr aufgerufen, und es kommt demnach auch zu keiner erhöhten Speicherbelastung, weil das Objekt sofort entfernt werden kann.

Trotz dieser eleganten Lösung bleibt aber der Performanznachteil beim Erzeugen der Objekte, weshalb eine Finalize-Methode nach wie vor nur mit gutem Grund implementiert werden sollte.

4.4 Resurrection

Eine Besonderheit ist die Ressurection (Wiederauferstehung) von Objekten. Zeiger im F-Reachable Queue werden als Wurzelzeiger aufgefasst, d.h. nachdem das Objekt bereits als tot erkannt wurde, wird dessen Zeiger aus dem Finalization-Queue in den F-Reachable-Queue verschoben, lebt dadurch wieder, und ist erst nachdem die Finalize-Methode abgearbeitet und der Zeiger entfernt wurde, tatsächlich tot, und wird entfernt.

Das Objekt kann nun in seiner Finalize-Methode eine globale oder statische Referenz auf sich selbst setzen, sodass es auch nach dem Entfernen seines Zeigers aus dem Queue erreichbar bleibt. Folglich darf es nicht freigegeben werden – es wurde wiederbelebt. Allerdings wird Finalize das nächste Mal nicht mehr ausgeführt, da der Zeiger schon aus dem Finalization-Queue entfernt wurde, und nicht mehr automatisch hinzugefügt wird. Das Objekt kann sich jedoch mittels einer GC-Methode wieder explizit im Queue registrieren (siehe Referenz GC-Methoden)

Resurrection hat einen begrenzten Nutzen, insbesondere auch weil es von der Programmstruktur her schwer zu durchschauen sein kann. Für bestimmte Anwendungsgebiete eignet es sich jedoch sehr gut, z.B. für Objekte mit zeitintensiven Konstruktoren (z.B. wenn eine Datenbankverbindung geöffnet bleiben soll), oder einen Objektpool mit einer fixen Anzahl an Objekte, die sich während der Laufzeit des Programms nicht ändern soll. Mittels Resurrection kann also die Freigabe eines derartigen Objekts verhindert werden.

5 Schwache Referenzen

Direkte Referenzen auf Objekte durch eine Variable sind sogenannte „Starke Referenzen“ (*strong references*). Die Freigabe derartiger Objekte wird vom Programm erlaubt – wenn die Variable nicht mehr auf das Objekt verweist, und daher nicht mehr darauf zugegriffen werden kann, ist es Garbage und kann zerstört werden. Solange noch eine starke Referenz darauf existiert, muss es im Speicher bleiben.

Schwache Referenzen (*weak references*) hingegen verhalten sich genauso wie starke (direkte) Referenzen, mit dem Unterschied, dass hier die Laufzeitumgebung bestimmt, ob die angesprochenen Objekte freigegeben werden oder nicht.

Was zuerst unsinnig erscheint, hat einen nützlichen Anwendungsfall: Hat ein Programm sehr viele Daten zu lesen und anzuzeigen, wird viel Speicher verbraucht. Nur das gerade angesehene Objekt im Speicher zu behalten, führt allerdings zu einer langsameren Bedienung, da beim Anzeigen eines anderen Objektes dieses erst geladen (z.B. aus Datei oder Datenbank) werden muss. Ein Programm könnte nun mit Hilfe von schwachen Referenzen im Hintergrund weitere Objekte als *Weak References* laden, um den Wechsel der Ansicht schneller zu gestalten. Der hierbei verbrauchte Speicher kann nun durch die schwachen Referenzen vom System jederzeit zurückgefordert werden, wenn es diesen benötigt (z.B. für andere Programme). Vor dem Holen einer *Weak Reference* muß geprüft werden, ob das Ziel *null* ist, dann wurde das Objekt vom System gelöscht. In diesem Fall (worst case) muss das Objekt nochmals erstellt werden.

Schwache Referenzen eigenen sich also für Objekte, die man zwar später wieder braucht oder möglicherweise brauchen kann, die das System aber bei Speicherplatzmangel freigeben darf. Schwache Referenzen sind somit auch ein einfacher Weg, sich an die im System vorhandene Speichermenge anzupassen, also bei viel freiem Speicher diesen voll auszunutzen oder sich bei Speicherknappheit auf weniger zu beschränken.

Zusammenfassung

Trotz des sehr einfachen Verfahrens zur automatischen Speicherverwaltung unterstützt .NET bereits sehr viele Mechanismen, die die Mächtigkeit und Flexibilität erhöhen. Generationen sind ein äußerst einfaches Konzept, aber dabei mindestens genauso effizient in ihrer Anwendung.

Die gute Unterstützung von Objekten aus *unmanaged code* in der .NET Umgebung ist das vermutlich wichtigste Feature der Speicherverwaltung. Die postulierte Modularität und Unabhängigkeit von Programmiersprachen kann somit ohne weiteres angewendet werden, und die Integration existierender Komponenten stellt keine unüberwindliche Hürde dar.

Der Finalization-Mechanismus ist ein guter Ersatz für Destruktoren in Fällen, wo Ressourcen nicht automatisch gehandhabt werden können. Die Features Resurrection und WeakReferences (beides auch in Java existent) stellen interessante Funktionen dar, mit Hilfe derer sich die Speicherverwaltung für Spezialfälle flexibel ausnutzen lässt.

Referenz GC-Methoden

Im Folgenden eine Auflistung nützlicher Methoden des Garbage Collectors, die zur Steuerung des Verhaltens und Interaktion mit dem GC benützt werden können:

`System.GC.Collect();`

Stösst eine Garbage Collection an

`System.GC.Collect(int generation);`

Stösst eine Garbage Collection an, und sammelt nur Objekt bis zu dieser Generation

`System.GC.GetGeneration(object);`

Liefert die Generation zum angegebenen Objekt

`System.GC.GetTotalMemory();`

Liefert System-Speicher zurück

`System.GC.KeepAlive(object);`

Hat den Zweck, allein mittels des Methodenaufrufs eine Referenz auf das Objekt darzustellen, um zu verhindern, dass es vom GC freigegeben wird (z.B. wenn das betreffende Objekt noch von *unmanaged code* verwendet wird)

`System.GC.SuppressFinalize(object);`

Damit wird der spätere Aufruf der Finalize-Methode unterdrückt – zur Verwendung in Dispose-Methoden

`System.GC.ReRegisterForFinalize(object);`

Bei Resurrection eines Objekts kann es mittels dieser Methode erneut im Finalization-Queue eingetragen werden

`System.GC.WaitForPendingFinalizers();`

Stoppt die Ausführung des Programms solange, bis der Thread, der die Finalize-Methoden nach der Garbage Collection parallel abarbeitet, fertig ist

Literaturverzeichnis

[MSDNa] http://msdn.microsoft.com/library/default.asp?url= /msdnmag/issues/1100/gci/toc.asp
Dezember 2003

[MSDNb] http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/default.aspx
Dezember 2003

[Bur02] Kevin Burton: .NET Common Language Runtime Unleashed, Sams Publishing 2002

[SNS03] Dave Stutz, Ted Neward, Geoff Shilling: Shared Source CLI Essentials. O'Reilly 2003