In diesem Kapitel:

- *Garbage Collection*
- *Ein einfaches Modell*
- *Fiinalization*
  - *Wiederaktivierung von Objekten während der Finalisation*
- *Interaktion mit dem Garbage Collector*
- *Erreichbarkeit(reachability) und* Reference *Objekte*
  - *Die* Reference *Klasse*
  - *Stärken dieses Referenzkonzepts und der Reachability*
  - *Reference Queues*

Garbage Collection und Memory

Civilization is a limitless multiplication of unnecessary necessaries.
- Mark Twain

Die Virtuelle Maschine benutzt eine Technik, bekannt als *Garbage Collection*, um sicherzustellen, dass alle benötigten Objekte im Speicher bleiben und bestimmt jene Objekte, welche nicht weiter benötigt werden und deren besetzter Speicher somit freigegeben werden kann. Wir befassen uns mit Techniken, wie man den Garbage Collector gezielt einsetzen kann, gehen kurz auf seine Funktionsweise ein und mit Hilfe spezieller Referenzobjekte der Garbage Collection Prozess beeinflusst werden kann.

1.1. Garbage Collection

Objekte werden mit Hilfe des `new` Operators kreiert. Aber es existiert kein `delete` Operator, um den Speicher frei zu geben, falls er nicht mehr benötigt wird. Falls Sie ein Objekt nicht mehr benötigen, referenzieren sie es einfach nicht mehr - referenzieren Sie einfach ein anderes Objekt oder auf `null`. Falls Sie einen Methodenaufruf abschliessen werden auch die lokalen Variablen der Methode nicht meht benötigt und referenzieren somit nichts mehr. Objekte, welche nicht mehr referenziert werden, werden als `Garbage`, Abfall, bezeichnet. Der Prozess, diese Objekte zu finden und den Speicherplatz, den diese Objekte besetzen wieder freizugeben, bezeichnet man als *Garbage Collection*.

Die Java Virtual Machine benutzt Garbage Collection, um sicherzustellen, dass alle referenzierten Objekte im Speicher bleiben, und der Speicherplatz, der von nicht mehr erreichbaren Objekten belegt wird, die im Programm nicht mehr benötigt werden, frei zu geben. Es werden nur jene Objekte deallociert, welche aus dem Programm über Referenzen nicht mehr erreichbar sind. Dies ist ein starker Hinweis darauf, dass ein Objekt nicht mehr benötigt wird. Ein Objekt wird also nicht entfernt, solange es noch auf irgend eine Art und Weise, genauer eine sogenannte *Referenzkette*, erreichbar ist. Als Wurzelreferenzen (*roots*) für diese Referenzketten werden die aktuell benutzten Programmvariablen benutzt.

Einfach ausgedrückt ist ein Objekt falls es nicht mehr erreichbar ist überflüssig und sein besetzter Speicherplatz kann neu vergeben werden. Der Speicherplatz kann, muss aber nicht neu vergeben werden. Der Garbage Collector wird nach unterschiedlichen Kriterien aktiv. Daher kann es sein, dass nicht mehr benötigte Objekte noch eine Weile im Speicher

vorhanden sind. Es kann auch sehr wohl sein, dass ein Programm beendet wird, ohne je dem Garbage Collector aktiviert zu haben. Der Garbage Collector wird in der Regel nur dann aktiv, wenn er mehr Speicher benötigt oder falls die Gefahr besteht, dass nicht genug Speicherplatz zur Verfügung steht.

Der Garbage Collector muss also dafür sorgen, dass man sich keine Sorgen über nicht mehr benötigte Referenzen machen muss. In vielen Systemen haben Sie die Möglichkeit Objektreferenzen explizit zu löschen. Dies ist sogar dann möglich, wenn auf dieses Objekt immer noch referenziert wird. Im schlimmsten Fall wird nun ein gelöschtes Objekt speichreplatzmässig durch ein neues Objekt ersetzt. Dies hätte zur Folge, dass nun die alten Referenzen auf ein völlig andersartiges Objekt zeigen. Ein grösseres Desaster ist damit vorprogrammiert. Falls der Garbage Collector diese Aufräumarbeiten für Sie erledigt, besteht dies Gefahr der falschen Ferenzierung kaum mehr.

Garbage Collection geschieht zwar auatomatisch, benötigt aber seine Zeit. Falls Sie zeitkritische Applikationen haben, macht es wenig Sinn in einem kritischen Moment dem Garbage Collector Zeit zum Aufräumen zu geben. Eine Methode dies zu vermeiden, ist ein sauberes Design, bei dem Sie möglichst wenig Objekte verwenden, und damit werden Sie auch möglichst wenig Abfall produzieren.

Garbage Collection ist keine Garantie dafür, dass immer Speicherplatz für neue Objekte zur Verfügung steht. Sie könnten eventuell unbeabsichtigerweise Listen mit Objekten füllen bis aller Speicherplatz verbraucht ist. Sie könnten aber auch Löcher im Speicher generieren (*leaks*) beispielsweise indem Sie eine Liste von Objekten generieren, die alle auf ein Objekt zeigen, welches bereits gelöscht wurde, oder welches Sie dann auf null setzen. Garbage Collection wird einige dieser Probleme lösen, aber nicht alle.

1.2. Ein einfaches Modell

Garbage Collection ist einfacher zu verstehen wenn man ein Modell zuhilfe nimmt. Deswegen beschreiben wir hier ein einfaches, in der Praxis aber in dieser Form nicht eingesetztes Beispiel für den Garbage Collection Prozess. Der Garbage Collection Prozess kann in zwei Phasen aufgeteilt werden:

- Aufteilung der Objekte in lebende und tote
- wieder verfügbar machen der durch die toten Objekte besetzten Speichers

Die lebenden Objekte sind jene, welche von den aktuell verwendeten Objekten aus erreichbar sind - also Objekte, die in irgend einer Form immer noch einsetzbar sind.

Tote Objekte sind Abfall; deren Speicherplatz kann also freigemacht werden.

Eines der üblichen Modell für Garbage Collectoren besteht im Zählen von Referenzen, dem *reference counting*:
falls Objekt X ein Objekt Y referenziert, wird der Referenzzähler bei Y um eins erhöht. Falls X seine Referenz auf Y löscht, reduziert das System den Referenzzähler bei Y um eins. Falls der Referenzzähler auf Null ist, braucht niemand mehr Y. Y kann somit aus dem Speicher entfernt werden. Falls Y entfernt wird, werden auch bei allen andern Objekten, auf die Y eventuell gezeigt hat, die jeweiligen Referenzzähler um eins reduziert.

Diese Referenzzähler versagen, falls Objekt X auf Objekt Y und Objekt Y auf Objekt X verweist, also *zirkuläre* Referenzketten auftreten. In diesem Fall werden weder X noch Y noch Referenzen, auf die die beiden verweisen, je gelöscht. Daher verwenden die meisten Garbage Collector Algorithmen kein einfaches Referenzenzählverfahren als Basis.

Das einfachste Modell ohne dieses Problem ist der sogenannte *mark-and-sweep* Algorithmus. Der Name deutet darauf hin, dass dabei zwei Phasen durchlaufen werden:

- zuerst werden, in der Markierungsphase, jene Objekte markiert, welche sicher nicht gelöscht werden dürfen. Dies geschieht durch Referenzauflösung, ausgehend von den Wurzelobjekten, also jenen Objekten, welche im Programm direkt verwendet werden. Es wird geschaut, auf welche Objekte diese verweisen.
  Als zweites werden diese referenzierten Objekte analysiert: es wird wieder untersucht, auf welche Objekte diese Objekte verweisen und diese werden ebenfalls markiert, falls sie nicht bereits markiert sind. Dieses Verfahren iteriert man, bis man keine neuen unmarkierten Objekte mehr findet.
  Damit hat man dann alle erreichbaren Objekte markiert.
- im zweiten Schritt, dem sweep, werden werden alle Objekte, welche nicht markiert sind, gelöscht.

Der Markierungsschritt darf eigentlich nicht unterbrochen werden, da Sie sonst nie genau wissen, ob Sie neue Objekte hinzubekommen haben, oder ob alte Objekte neu benutzt werden. Das heisst aber, dass in der Markierungsphase im Extremfall das Programm gestoppt werden muss.

Der Algorithmus hat noch einige andere Probleme. Aber das gesamte Thema Garbage Collection ist sehr komplex und immer noch im Fluss. Wir wollen nicht alle Sonderfälle kennen lernen. Wichtig ist es lediglich, das Sie das Grundprinzip verstanden haben und vorallem auch sehen, das Garbage Collection sie einiges an Performance kosten kann.

Jede VM implementiert den Garbage Collector auf ihre Art und Weise. In derVM Spezifikation ist dieser Algorithmus nicht im Detail beschrieben, es wird viel eher einfach eine Lösungskategorie beschrieben.Sie sollten also das Mark-and-Sweep Modell mehr als gedankliches Modell verstehen - nicht als die Implementation der aktuellenVM.

1.3. Finalization

In der Regel werden Sie nicht merken, wenn ein verwaistes Objekt aus dem Speicher entfernt wird- es funktioniert einfach. Aber eine Klasse kann eine `finalize()` Methode implementieren, welche ausgeführt wird, bevor der Objektraum frei gegeben wird. Diese Methode gibt Ihnen die Möglichkeit noch bestimmte Aktivitäten auszuführen, die sonst zu Problemen führen könnten.

Die `finalize()` Methode ist bereits auf der Stufe der Object Klasse, also der Wurzel aller Klassen, definiert:

`protected void finalize() throws Throwable`

diese Methode wird vom Garbage Collector aufgerufen, falls dieser festgestellt hat, dass das dazugehörige Objekt vernichtet werden soll.
Diese Methode wird einmal und nur einmal aufgerufen. Falls Sie aus irgend einem Grund das Objekt beim Methodenaufruf reaktivieren, werden Sie diese Methode nicht mehr einsetzen können, ausser Sie rufen sie direkt auf. Diese Methode kann irgend eine Ausnahme werfen. Diese wird aber vom Garbage Collector ignoriert. Es ist auch nicht bestimmt, welcher Systemthread diese Methode ausführen wird. Aber dieser Thread wird keine benutzergesteuerten Locks akzeptieren.

In der Regel benötigen Sie keine finalize() Methoden. Falls Sie eine definieren, sollten Sie sehr vorsichtig sein, da Sie sonst böse Überraschungen erleben könnten, beispielsweise könnten Objekte, auf die Sie referenzieren, bereits gelöscht sein.

Der Garbage Collector kümmert sich nur um den Speicher. Sollten Sie andere Ressourcen benützen, beispielsweise Dateien, dann könnte der Einsatz des Finalizers sinnvoll sein. In dieser Methode könnten Sie beispielsweise sicherstellen, dass alle Dateien geschlossen werden, die Sie dämlicherweise nicht bereits geschlossen haben. Besser wäre es, die Dateien schon vorher zu schliessen!

Schauen wir uns ein Beispiel für eine Klasse an, welche mit Dateien arbeitet, und die den Finalizer sehr vorsichtig definiert:

```
public class ProcessFile {
      private FileReader file;

      public ProcessFile(String path) throws FileNotFoundException {
            file = new FileReader(path);
      }

      // ...

      public synchronized void close() throws IOException {
            if (file != null) {
                  file.close();
                  file=null;
            }
      }
```

```
	protected void finalize() throws Throwable {
		try {
			close();
		} finally {
			super.finalize();
		}
	}
}
```

Das Schliessen der Datei berücksichtigt einige mögliche Fälle: es könnte sein, dass die Datei bereits geschlossen wurde oder dass mehrere Threads versuchen die Datei zu schliessen.

Der Finalizer führt in der Regel auch den Finalizer der Oberklasse aus. Dies garantiert, dass auch die übergeordneten Teile des Objekts vernichtet werden, also keine Leichen im Speicher enstehen. Damit wird auch garantiert, dass die Oberklasse zum Objekt aufgeräumt wird, selbst dann, wenn eine Ausnahme geworfen würde.

Die Reihenfolge, in der der Garbage Collector Objekte löscht ist nicht voraussehbar: es kann sein, dass ein Objekt nie gelöscht wird. Es hängt alleine von Garbage Collector ab, wann die Zeit für ein Objekt gekommen ist. Sie können den Garbage Collector entweder mit `System.gc()` oder `Runtime.gc()` von Hand starten (darauf kommen wir im nächsten Abschnitt zurück). Aber Sie haben keine Kontrolle darüber, ob und wann der Prozess ausgeführt wird.

Falls eine Applikation regulär beendet wird, wird der Garbage Collector nicht ausgeführt. Das heisst, dass in diesem Falle die Finalizer überhaupt nie ausgeführt werden. In der Regel ist dies kein Problem, da Sie ja nicht so programmmieren sollten, dass der Finalizer nötig ist. Es ist auch so, dass in der Regel (je nach Implementation der VM) beim Abschluss eines Programms alle Dateien und Socketverbindungen geschlossen werden. Temporäre Dateien können Sie zudem als "delete on exit" markieren. Dann werden diese automatisch beim Programmabschluss gelöscht.

1.3.1. Wiederbeleben eines Objekts im Finalizer

Im Prinzip könnten Sie bei der Ausführungdes Finalizers ein Objekt reaktivieren - beispielsweise indem Sie dieses Objekt in eine statische Liste eintragen. Allerdings ist dies fast das Dümmste was Sie machen können in Ihrer Programmiererkarriere. Es gibt mehrere Gründe dafür:

- die `finalize()` Methode wird nur einmal ausgeführt. Falls Sie in der `finalize()` Methode das Objekt zu neuem Leben erwecken, werden Sie diesen Schritt lediglich einmal machen können. Das ist aber vielleicht nicht ganeu das, was Sie eigentlich wollten.
- falls Sie wirklich ein Objekt wiederbeleben müssen, sollten Sie sich überlegen, ob es nicht gescheiter wäre, das Objekt zu klonen oder ein neues Objekt zu kreieren.
- falls Sie unbedingt daran festhalten wollen, ist vermutlich Ihr Design so schief, dass Sie sich darüber Gedanken machen sollten.

1.4. Interaktion mit dem Garbage Collector

Obschon Sie in Java keine Möglichkeit haben, ein Objekt aus dem Speicher zu löschen, bietet Ihnen der Garbage Collector die Möglichkeit unbenutzte Objekte zu suchen und zu eliminieren. Die Runtime Klasse bietet Ihnen zusammen mit einigen Hilfsmethoden in er System Klasse die Möglichkeit, den Garbage Collector aufzurufen und zu verlangen, dass die Finalizer bei allen Objekten ausgeführt wird, die nicht mehr benötigt werden (sofern Finalizer definiert sind):

`public void gc()`
verlangt, dass die VM sich ernsthaft überlegt, die nicht mehr benutzten Objekte zu löschen, so dass der Speicherplatz frei wird.

`public void runFinalization()`
verlangt, dass die VM sich ernsthaft überlegt, die Finalizer der Objekte auszuführen,welche nict mehr erreichbar sind, aber für die deren Finalizer noch nicht ausgeführt wurden.

`public long freeMemory()`
liefert eine Schätzung über den freien Speicher in Bytes

`public long totalMemory()`
liefert die Anzahl Bytes im Systemspeicher.

Um diese Methoden starten zu können, benötigen Sie eine Referenz auf das aktuelle `Runtime` Objekt. Dieses erhalten Sie mittels der statischen Methode `Runtime.getRuntime()`. Die System Klasse unterstützt die statischen `gc()` und `runFinalization()` Methoden benfalls: in diesem Falle werden aber auf die Runtime Methoden des zugeordneten Laufzeitsystems zurückgegriffen:

`System.gc()` entspricht `Runtime.getRuntime().gc()`

Es kann aber sehr wohl sein, dass der Garbage Collector beim Starten keinen Speicher frei geben kann. Dies hätte zur Folge, dass keine Objekte gelöscht werden. Es kann aber, speziell in zeitkritischen Anwendungen, sinnvol sein vor dem Kreieren grösserer oder mehrerer Objekte den Garbage Collector zu starten. Der Vorteil eines solchen Vorgehens ist, dass Sie zum einen verfügbaren Speicher eventuell frei machen und auch dafür sorgen, dass anschliessen im zeitkritischen Teil Ihres Programms nicht plötzlich der Garbage Collector aktiv wird. Die folgende Methode schafft möglichst viel freien Speicher:

```
public static void fullGC() {
      Runtime rt = Runtime.getRuntime();
      long istFrei = rt.freeMemory();
      long warFrei;
      do {
            warFrei = istFrei;
            rt.runFinalization();
            rt.gc();
            istFrei = rt.freeMemory();
      } while(istFrei > warFrei);
}
```

Die Methode führt die while Schleife so lange aus, bis der freie Speicher auf dem Maximum ist. Dies geschieht durch wiederholten Aufruf der Methoden, die wir oben beschrieben haben.

In der Regel werden Sie runFinalization() nicht aufrufen müssen, weil finalize() Methoden asynchron von Garbage Collector aufgerufen werden. Unter Umständen ist es aber hilfreich, die Finalizer aufzurufen, da diese eventuell Ressourcen frei geben könnten.

Für viele Fälle ist die obige Methode zu absolut. Oft wird man sich mit einem einfachen Aufruf des Garbage Collectors begnügen. Die wiederholten Aufrufe des Garbage Collectors bringen bei jedem Aufruf weniger. Auf vielen Systemen sind sie sogar nutzlos.

```
package garbagecollector;

public class TotaleSpeicherbereinigung {

  public static void main(String[] args) {
    Runtime rt = Runtime.getRuntime();
    String[] tm = new String[10000];
    for (int i=0; i<10000; i++)
      tm[i]=i+" tes Objekt wurde kreiert";
    long istFrei = rt.freeMemory();
    long warFrei;
    System.out.println("Freier Speicher vor GC: "+istFrei);
    do {
      warFrei = istFrei;
      System.out.println("Finalization");
      rt.runFinalization();
      rt.gc();
      istFrei = rt.freeMemory();
      System.out.println("Freier Speicher : "+istFrei);
    } while (istFrei > warFrei);
    System.out.println("Freier Speicher : "+istFrei);
    for (int i=0; i< 10000; i++)
      tm[i]=null;
    System.out.println("Freier Speicher vor GC: "+istFrei);
    do {
      warFrei = istFrei;
      System.out.println("Finalization");
      rt.runFinalization();
      rt.gc();
      istFrei = rt.freeMemory();
      System.out.println("Freier Speicher : "+istFrei);
    } while (istFrei > warFrei);
    System.out.println("Freier Speicher : "+istFrei);
  }
}
```

Das Beispiel schafft zuerst möglichst viel Speicherplatz, kreiert dann eine Menge Objekte und löscht diese anschliessen wieder. Die zweite Speicherplatzbereinigung gibt einiges an Speicher wieder frei.

1.5. Erreichbarkeitszustand und Reference *Objekte*

Ein Objekt kann vom Garbage Collector lediglich gelöscht werden, falls keine Referenzen auf das Objekt verweisen. Es gibt aber Fälle, in denen Sie auch Objekte entfernen möchten, welche diese Bedingung nicht erfüllen.

Beispiel:
Sie schreiben einen Web Browser und benötigen Speicherplatz. Eines Ihrer Objekte ist ein Bild. Sie wissen genau, dass dieses noch im Cache ist, also jederzeit leicht wieder geladen werden kann. Daher ist es naheliegend, temporär dieses Objekt zu entfernen, obschon Sie es eigentlich noch benötigen.

Sie möchten also eine spezielle Art von Referenzen:
die Referenz sollte nicht dazu führen, dass das Objekt im Speicher belassen wird. Solche Referenzen werden als *Referenzobjekte* bezeichnet.

Die Aufgabe eines Referenzobjekts ist es, eine Referenz auf ein anderes Objekt, den *Referenten*, aufrecht zu erhalten.

Damit kann man indirekte Beziehungen einführen:
anstelle einer Referenz auf ein Objekt benutzen Sie eine Referenz auf ein Referenzobjekt, welches den Zugang zum eigentlichen Objekt schafft.

Der Garbage Collector kann nun das eigentliche Objekt löschen, da er weiss, dass ein Referenzobjekt darauf verweist. Das Referenzobjekt muss wissen, wie es wieder zum Objekt gelangt.

1.5.1. Die Reference Klasse

Die Klassen für Referenzobjekte sind Teil des `java.lang.ref` Packages. Die Hauptklasse im Package ist die abstrakte Klasse `Reference`, die Oberklasse für alle Referenzklassen.

1.5.1.1. Klassenbeschreibung

Die Klasse liefert ein Reference Objekt. Diese unterstützen in begrenztem Masse eine Wechselwirkung mit dem Garbage Collector. Ein Programm kann ein Reference Objekt einsetzen, um eine Referenz auf ein anderes Objekt aufrecht zu erhalten und zwar so, dass dieses zweite Objekt sogar vom Garbage Collector gelöscht werden kann.

Ein Programm kann auch Reference Objekte einsetzen, um informiert zu werden, falls der Garbage Collector festgestellt hat, dass sich die Erreichbarkeit eines Objekts verändert hat.

1.5.1.2. Package Spezification

Ein *reference Objekt* kapselt eine Referenz auf ein anderes Objekt und zwar so, dass das Referenzobjekt selbst genau so manipuliert werden kann, wie jedes andere Objekt.

Es gibt drei Typen von Referenzobjekten:
- *soft,*
- *weak* und
- *phantom.*

Jeder Typ entspricht einem anderen Level von Erreichbarkeit, gemäss der Definition, die wir gleich geben werden.

Softreferenzen werden bei speichersensitiven Caches, Weakreferenzen werden oft bei umfangreichen Mappings und Phantomreferenzen werden für pre-mortem Aufräumarbeiten eingesetzt (sie erlauben mehr Flexibilität der der Handhabung der Objektvernichtung und Speicherzurückgewinnung) im Zusammenhang mit dem Java Finalization Mechanismus.

Jeder Referenztyp wird mittels einer Unterklasse der abstrakten Reference Klasse implementiert. Eine Instanz einer dieser Unterklassen kapselt eine einzige Referenz zu einem bestimmten Objekt, den Referenten. Jedes Referenzobjekt besitzt Methoden für den Zugriff auf, das Löschen und und das Bestimmen der Referenz. Referenzobjekte selbst können nicht verändert werden. Sie können die drei grundlegenden Unterklassen der Reference Klasse erweitern und eigene Klassen definieren und mit Methoden versehen.

1.5.1.3. Notifikation

Ein Programm kann verlangen, dass es informiert wird, falls sich die Erreichbarkeit eines (des referenzierten) Objekts ändert. Dazu muss ein Referenzobjekt beim Kreieren in eine Referenzwarteschlange eingetragen (*registered*) werden (*reference queue*). Nachdem der Garbage Collector festgestellt hat, dass sich die Erreichbarkeit der Referenz geändert hat, wird er diese Referenz in die entsprechende Warteschlange eintragen (*enqueue*). Diese Warteschlangen werden durch die `ReferenceQueue` Klasse implementiert.

1.5.1.4. Automatisch gelöschte Referenzobjekte

Soft und Weak Referenzen werden vom Garbage Collector automatisch gelöscht, bevor sie in eine Warteschlange eingeordnet werden (selbst wenn sie eingetragen sein sollten). Daher müssen Soft und Weak Referenzen bei keiner Warteschlange eingetragen werden. Sie können

auch so genutzt werden. Bei Phantomreferenzen muss eine Warteschlange eingesetzt werden damit diese Referenzobjekte sinnvoll nutzbar sind.

1.5.1.5. Erreichbarkeit

Die Reihenfolge der unterschiedlichen Erreichbarkeitslevel wiederspiegelt sich im Lebenszyklus der Objekte. Der stärkste Erreichbarkeitslevel ist die starke Erreichbarkeit, die Unerreichbarkeit der schwächste.

Die verschiedenen Level sind folgendermassen definiert:

- ein Objekt ist (*strongly reachable*) stark erreichbar, falls es von einem Thread erreicht werden kann ohne irgend ein Referenzobjekt benutzen zu müssen. Ein neu kreiertes Objekt ist stark erreichbar, vom Thread, der das Objekt kreiert hat.
- ein Objekt ist *softly reachable,* falls es zwar nicht direkt, aber mittels einer Softreferenz erreicht werden kann.
- ein Objekt ist *weakly reachable*, falls es weder direkt (strong) noch soft sondern lediglich über eine Weakreferenz erreicht werden kann. Falls die Weakreferenzen zu einem weakly reachable Objekt gelöscht werden, kann dieses Objekt bei der Finalization berücksichtigt werden.
- ein Objekt ist *phantom reachable*, falls es keine der obigen Bedingungen erfüllt, seine Finalization ausgeführt wurde und eine Phantomreferenz auf das Objekt zeigt.
- schliesslich ist ein Objekt *unreachable* und damit löschbar, wenn es keine der obigen Bedingungen erfüllt.

1.5.1.6. Methoden

Die folgenden Methoden sind für alle Referenzklassen definiert.

`public Object get()`
liefert das referenzierte Objekt dieses Referenzobjekts. Falls das Referenzobjekt gelöscht wurde, entweder durch den Garbage Collector oder durch das Programm,wird null zurück geliefert.

`public void clear()`
löscht das Referenzobjekt. Falls diese Methode aufgerufen wird, wird das Objekt nicht in eine Warteschlange eingetragen.

`public boolean isEnqueued()`
zeigt an, ob das Referenzobjekt bereits ineine Warteschlange eingetragen wurde, durch ein Programm oder durch den Garbage Collector. Falls das Referenzobjekt nicht in eine Warteschlange eingetragen wurde, liefert die Methode den Wert false.

`public boolean enqueue()`
trägt dieses Referenzobjekt in die Warteschlange ein, in die es eingetragen ist (falls überhaupt). Der Rückgabewert zeigtan, ob der Eintrag erfolgreich war (true) oder nicht (false), beispielsweise, weil es in der Warteschlange nicht eingetragen ist.

Die Unterklassen stellen Methoden zum Binden des Referenzobjekts an den Referenten zur Verfügung stellen. Dies geschieht in der Regel im Konstruktor. Beispiel

```
public SoftReference(Object referent, ReferenceQueue q)
```

1.5.1.7. Testprogramme

Sie finden auf dem Server / der CD mehrere Testprogramme, mit deren Hilfe Sie die Referenzmechanismen genauer kennen lernen können.

Neben dem einfachen Ausführen der Programme können Sie auch noch mit den VM Parametern spielen.

Testen Sie beispielsweise

```
-verbose:gc -Xms16M -Xmx80M -Xprof
```

oder

```
-verbose:gc -Xms16M -Xmx80M -Xprof - Xrunhprof
```

als VM Parameter, *nicht* als Applikationsparameter.

Der erste Parameter produziert zusätzliche Garbage Collector Informationen,
die zwei nächsten spezifizieren initialen und maximaler Memory Pool in Megabyte (M),
die zwei letzten Parameter lieferen Ihnen Profilinformationen zum Ausführen der Testprogramme.

Der erste und die zwei letzten Parameter sollten in produktiven Applikationen nicht eingesetzt werden.

Und hier eine Beispielausgabe mit diesen Parametern wie oben gesetzt:

```
...
GC 75108K->75108K(77484K), 0.5884576 secs]
[Full GC 76670K->76670K(81664K), 122.7273878 secs]
[GC 77451K->77451K(81664K), 8.9485845 secs]
[GC 79014K->79014K(81664K), 0.1406164 secs]
[Full GC 80576K->80576K(81664K), 1.1432958 secs]
[Full GC 80576K->80576K(81664K), 0.0507517 secs]
```

[Main]eatMemory() **** OUT OF MEMORY ****
(Programmausgage)

```
[Full GC 80576K->1666K(81664K), 0.0747355 secs]

Flat profile of 157.90 secs (684 total ticks): main

  Interpreted + native   Method
 51.6%     2  +    77    weakreference.Main.eatMemory
  9.8%     0  +    15    java.lang.Runtime.gc
  5.2%     0  +     8    java.io.FileInputStream.readBytes
  4.6%     0  +     7    java.io.FileOutputStream.writeBytes
  2.0%     0  +     3    java.lang.System.gc
  1.3%     0  +     2    java.lang.System.arraycopy
  1.3%     2  +     0    java.util.LinkedList.addBefore
  1.3%     0  +     2    java.lang.ClassLoader.defineClass0
  1.3%     2  +     0    java.util.LinkedList$Entry.<init>
```

```
  0.7%     0  +     1    sun.net.www.URLConnection.<init>
  0.7%     0  +     1    java.lang.Class.getPrimitiveClass
  0.7%     0  +     1    java.security.AccessController.doPrivileged
  0.7%     0  +     1    java.io.Win32FileSystem.getBooleanAttributes
  0.7%     1  +     0    java.util.LinkedList.add
  0.7%     0  +     1    java.lang.String.replace
  0.7%     0  +     1    java.io.BufferedReader.readLine
  0.7%     0  +     1    java.util.Properties.load
  0.7%     0  +     1    sun.security.provider.Sun$1.run
 84.3%     7  +   122    Total interpreted

  Thread-local ticks:
 77.6%   531             Blocked (of total)
 10.5%    16             Class loader
  5.2%     8             Unknown code


Global summary of 158.55 seconds:
100.0% 13910             Received ticks
 94.4% 13127             Received GC ticks
  0.2%    33             Other VM operations
  7.6%  1051             Blocked ticks
  0.2%    30             Threads_lock blocks
  8.7%  1207             Delivered ticks
  8.7%  1207             All ticks
  0.1%    16             Class loader
  0.1%     8             Unknown code
```

Sie finden eine vollständige Liste aller Parameter plus eine detaillierte Beschreibung in der JDK Dokumentation unter `tooldocs\win32\java.html`.

Verschiedene Tools zeigen Ihnen diese Auswertungen grafisch an, beispielsweise JProbe.

1.5.2. Die ReferenceQueue Klasse

1.5.2.1. Definition der Klasse

public class **ReferenceQueue** extends Object

Der Garbage Collector stellt Objekte, die einen bestimmten Erreichbarkeitslevel erreichen, in die entsprechende Warteschlange. Sie könnten diese Information benutzen, um situativ Aktionen zu starten, speziell falls Sie speicherkritische Applikationen schreiben.

1.5.2.2. Der Konstruktor

`ReferenceQueue()`

konstruiert eine neue Referenzobjekt-Warteschlange.

1.5.2.3. Die Methoden

`Reference poll()`

prüft die Warteschlange, um festzustellen, ob ein Referenzobjekt verfügbar ist. Falls dies der Fall ist, wird gleich eines zurückgeliefert.

`Reference remove()`

entfernt das nächste Referenzobjekt indieser Warteschlange. Falls keines verfügbar ist, blockiert die Methode solange, bis ein Objekt vorhanden ist.

`Reference remove(long timeout)`

entfernt das nächste Referenzobjekt dieser Warteschlange. Falls keines vorliegt, blockiert die Methode bis eines vorhanden ist oder die engegebene Zeit überschritten wird.

Damit schliessen wir den eher komplexen, weil kaum gebrauchten Teil unserer Betrachtung des Garbage Collectors ab.

SPEICHERVERWALTUNG