

Dynamische Speicherverwaltung

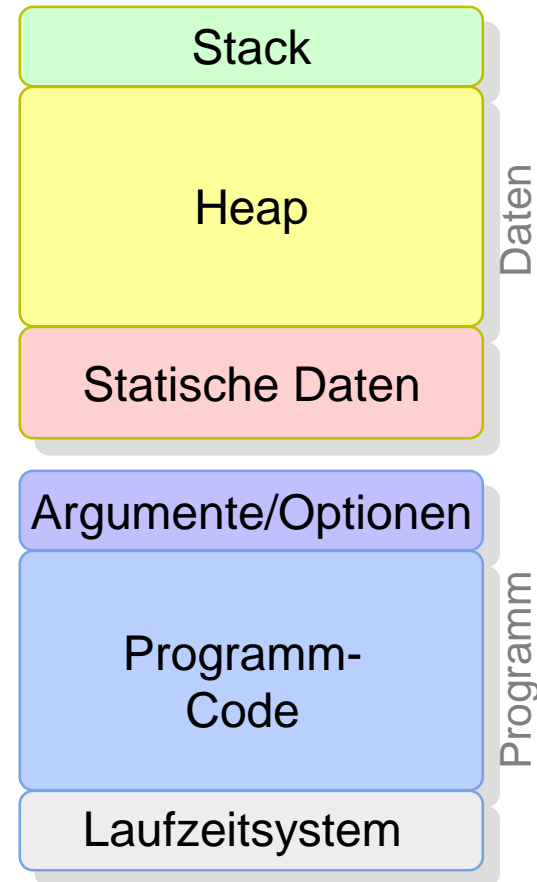


- Sie wissen, wie der Speicher in Programmen verwendet wird
- Sie wissen, was ein Heap (eine Halde) ist
- Sie kennen die Aufgaben des Speicherverwalters
- Sie wissen, wie die automatische Speicherfreigabe funktioniert
- Sie kennen mehrere Verfahren und deren Vor- und Nachteile
- Sie wissen, was Weak References und Finalizer sind und können damit umgehen

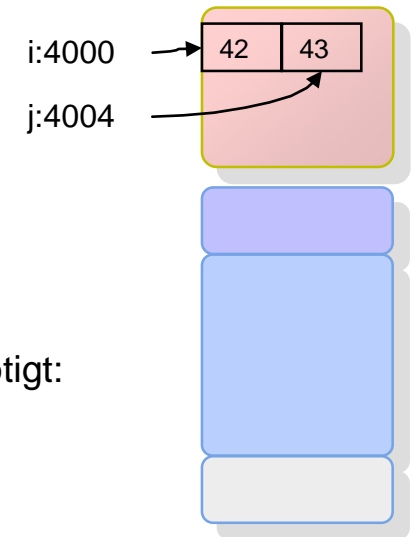
Umgang mit dem Hauptspeicher

- Der Umgang mit dem dynamischen Speicher (Heap) hat früher bis zu **40% der Entwicklungszeit** verursacht (inkl. Fehlersuche).
- Der Umgang mit dem dynamischen Speicher ist für die **Effizienz des Programms** oft ausschlaggebend.
- **Fatale Programmfehler** (Abstürze) sind meist auf einen Fehler im Gebrauch mit dem Speicher zurückzuführen.
- Fehler beim Umgang mit dem dynamischen Speicher sind **schwer zu entdecken**, weil sie erst zur Laufzeit auftreten.
- In objektorientierten Sprachen werden Objekte kreuz und quer referenziert, so dass **schwierig festzustellen ist, ob ein Objekt noch verwendet wird**.
- Java kennt deshalb die **automatische Speicherfreigabe**: Garbage-Collection (Müllsammelner)

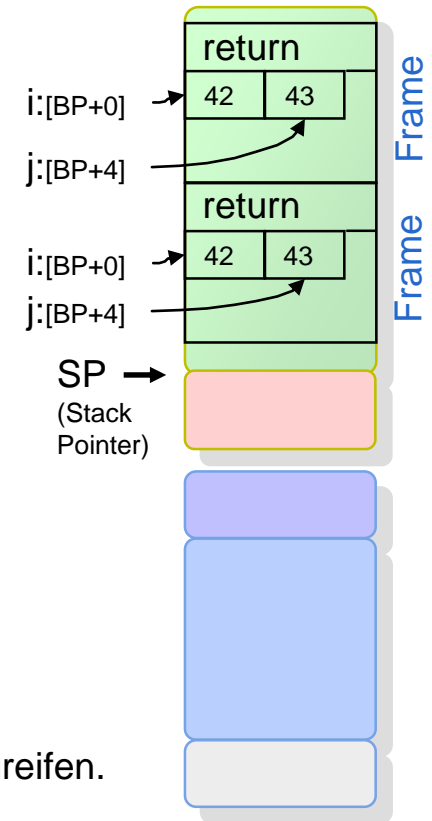
- **Stack:**
 - Rücksprung-Adressen
 - lokale Variablen
- **Heap:**
 - dynamische Daten (Objekte)
- **Statische Daten**
 - `static`
- **Argumente/Optionen**
 - beim Aufruf mitgegebene Werte
- **Programm Code**
 - das ausführbare Programm
- **Laufzeitsystem**
 - z.B. Kopie von Datei-Puffern



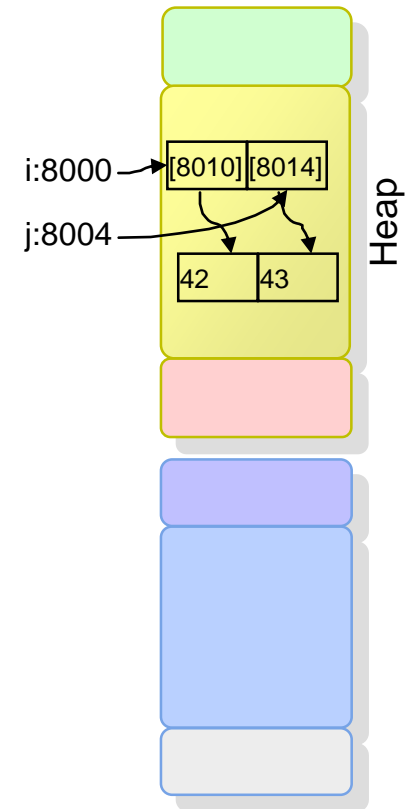
- Einfachste Zuteilungsstrategie
 - Der Compiler legt die Zuordnung Variablenname zu Speicheradressen bei der Übersetzung fest.
 - Schon in den ersten Programmiersprachen (FORTRAN) unterstützt.
 - Speicher wird beim Start des Programms angefordert und beim Verlassen wieder freigegeben.
 - Nutzung in Java: **statische Typen** (static)
- + Vorteile:
- + **einfach**
 - + beim Start kann schon gesagt werden, ob Speicher ausreicht
- Nachteile:
- die **Größen** aller Datenstrukturen müssen **zur Übersetzungszeit** bekannt sein.
 - es muss so viel **Speicher angefordert** werden, wie das Programm **maximal** benötigt: u.U. zu viel.
 - **keine rekursiven** Programmaufrufe möglich.
 - **keine dynamischen Datenstrukturen** wie Listen und Bäume möglich.



- Beim jedem Aufruf einer Prozedur (Methode) wird ein Speicherbereich reserviert: **Activation Record** oder **Frame**.
 - Diese Frames werden als Stack organisiert: LIFO
 - Die lokalen Variablen einer Methode werden relativ zum Framepointer (FP, Intel: BP=Base Pointer) angesprochen, z.B. [BP+4].
 - Der Offset zum FP muss vom Compiler festgelegt werden.
 - Nutzung in Java: für **lokale Variablen** von einfachen Typen (double, int, usw.) in Methoden
- + Vorteile:
- + **rekursive Aufrufe sind möglich.**
 - + **effiziente Zuteilung/Freigabe von Speicher (SP, FP).**
- Nachteile:
- Grösse der einzelnen Datenstrukturen muss bekannt sein.
 - Der Aufrufer kann nicht auf die Werte des Aufgerufenen nach dessen Rückkehr zugreifen.
 - D.h. die Werte auf dem Stack sind nach dem Verlassen der Methode verloren.
 - Stack-Overflow möglich.



- Variablen enthalten nicht den Wert sondern eine Adresse in den (Heap-) Speicherbereich.
 - Solche Variablen werden als Pointer oder Referenzen bezeichnet.
 - Referenz-Variablen selber können statisch, auf dem Stack oder im Heap gespeichert sein.
 - Der Speicher muss vom Programm mittels `new` angefordert und mittels `delete` wieder freigegeben werden.
 - Java: **alle Variablen von nicht eingebauten Typen** (Klassen), Strings und Arrays
- + Vorteile:
- + **rekursive Aufrufe sind möglich.**
 - + **die Grösse der einzelnen Datenstrukturen kann zur Laufzeit festgelegt werden.**
 - + **dynamische Datenstrukturen, z.B. Listen, sind möglich.**
- Nachteile:
- Zuteilung/Freigabe der Daten ist relativ rechenintensiv und kompliziert.
 - Speicher muss explizit vom Programm angefordert und wieder freigegeben werden
-> Programmierfehler möglich.
 - Speicherüberlauf (Heap-Overflow) möglich.



Einfache Speicherverwaltung

Der Speicherverwalter (Storage-Manager)

- Hat Schnittstelle um Speicher anzufordern und wieder freizugeben.
- Der einfache Speicherverwalter gibt direkt die Adresse des Speicherbereichs zurück.

```
class Storage {  
    long malloc (int size); // memory allocate  
    void free(long addr);  
}
```

- Bei objektorientierten Sprachen unpraktisch
 - die Grösse der Objekte ist dem System bekannt also Grössengabe redundant
 - es muss zusätzlich ein/der Konstruktor aufgerufen werden.

```
class Storage {  
    Object new (Class, Object[] args);  
    void delete (Object obj);  
}
```

- in Java Teil der Sprache: `MyObject obj = new MyObject("hallo");`

- Der Speicherverwalter unterhält zwei Listen:
 - **Belegt-Liste:** Liste der belegten Speicherbereiche.
 - **Frei-Liste:** Liste der freien Speicherbereiche.
- Speicher wird angefordert
 - es wird ein Block der angeforderten Grösse aus dem freien Bereich (definiert durch Frei-Liste) als belegt markiert.
 - er wird in eine Belegt-Liste eingetragen.
 - Rest des Bereichs (Verschnitt) wird in die Frei-Liste eingetragen.
 - eine **Referenz** auf den Bereich wird zurückgegeben.
- Speicher wird freigegeben
 - der Speicher wird aus der Belegt-Liste entfernt und in die **Frei-Liste** eingetragen.
- Problem: (externe) Fragmentierung (später)
 - im Hauptspeicher bilden sich mit der Zeit Löcher.
- Lösung
 - der Speicher wird periodisch kompaktiert (die Löcher werden zusammengeschoben).

- Vergessen den Speicher anzufordern
 - Konsequenz:
 - eine Referenz-Variable enthält einen zufälligen Wert -> es wird eine beliebige u.U. benutzte Speicherstelle zugegriffen und verändert -> "komische" Werte evtl. Programmabbruch
 - Abhilfe:
 - alle Referenz-Variablen automatisch mit einem 0-Wert (null) initialisieren, in Java -> Exception (NullPointerException)

- Zuwenig Speicher angefordert, z.B bei Arrays, bei Objekten
 - Konsequenz:
 - es wird benachbarter Speicher überschrieben, siehe oben.
 - Abhilfe:
 - Überprüfen ob Zugriff im gültigen Bereich, z.B. Array-Index (Java).
 - Die (Speicher-) Grösse von Objekten wird automatisch bestimmt (Java).
 - Nur sichere Cast werden erlaubt (Java).

- Vergessen den Speicher freizugeben: **Memory Leak**
 - Konsequenz:
 - das Programm benötigt immer mehr Speicher.
 - Bei virtuellem Speicher (Teil des Speichers wird auf die Disk ausgelagert -> Vorlesung BS): es muss immer mehr auf Disk ausgelagert werden -> Programm wird langsamer, später **Programmabbruch**.
 - Abhilfe
 - der **Speicherverwalter (Storage-Manager)** gibt den Speicher automatisch frei (Java).
- Der Speicher wird freigegeben obwohl er noch verwendet wird: **Dangling Pointer**
 - Konsequenz
 - zuerst passiert nichts, aber wenn der Speicherbereich vom Speicherverwalter wieder neu vergeben wird, dann zeigt die alte Variable immer in diesen Bereich.
 - es wird eine anderweitig benutzte Speicherstelle zugegriffen und verändert -> **"komische" sich plötzlich verändernde Werte oft/meist Programmabbruch**.
 - Abhilfe
 - der Speicherverwalter gibt den Speicher automatisch frei (Java).

Fehler bei der Freigabe von Speicher sind wesentlich schwieriger in den Griff zu bekommen als die bei der Anforderung

Automatische Speicherverwaltung

- Hauptaufgabe der automatischen Speicherverwaltung ist die Freigabe des nicht mehr benötigten Speichers.
- einfache Verfahren (keine Laufzeitinformation notwendig):
 1. Referenzzählung
 2. Smart-Pointer
- vollautomatische Speicherfreigabe: **Garbage Collector** (Laufzeitinformation notwendig):
 3. Mark-Sweep GC
 4. Copying GC
 5. Generational GC
- Alle Verfahren haben Vor- und Nachteile: **there is no silver bullet**
- Performance Einbusse je nach Verfahren (ca. 5-10%).
- in Java;
 - Speicherverwalter sucht periodisch nach freiem Speicher.
 - `System.gc()` veranlasst den Speicherverwalter nach freiem Speicher zu suchen - sobald er Zeit hat.

Automatische Speicherverwaltung

Einfache Verfahren

1. Referenzzählung (reference counting)

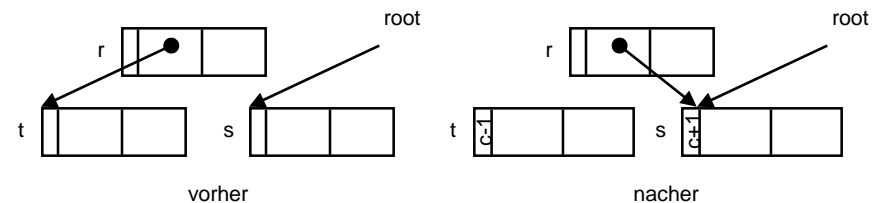
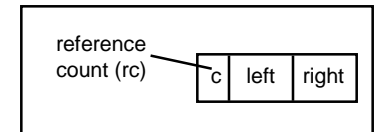
- Es wird gezählt, wie viele Referenzen auf ein Objekt verweisen.
- Wenn keine Referenz mehr vorhanden ist (Referenz-Zähler = 0), dann kann Objekt gelöscht werden.
- Operationen des (Referenz-)Zähler
 - Bei einer Zuweisung wird der (Referenz-)Zähler um 1 erhöht.
 - Bei Wegnahme einer Referenz wird der Zähler um 1 erniedrigt.
- als Methoden (Bsp. unten für Wertzuweisung):

```
int rc;
```

```
void addRef() {  
    rc ++;  
}
```

```
int release() {  
    if (--rc == 0) {  
        delete(this);  
    }  
    return rc;  
}
```

```
if (r.left != null)r.left.release();  
r.left = s;  
if (r.left != null)r.left.addRef();
```



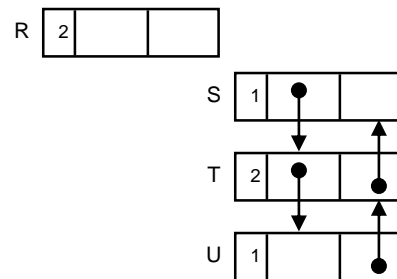
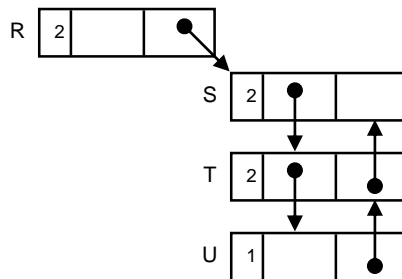
1. Eigenschaften der Referenzzählung

■ Vorteile der Referenzzählung

- einfach, geringer Verwaltungsaufwand.
- Speicher wird zum frühesten möglichen Zeitpunkt freigegeben.

■ Nachteile

- muss vom Programmierer durchgeführt werden -> Fehler möglich.
- zusätzliche Operationen (addRef, release) bei jeder Pointer-Zuweisung.
- **zyklische Datenstrukturen können nicht freigegeben werden.**
- häufiger als angenommen: z.B. doppelt verkettete Liste.



2. Smart Pointers zur Referenzzählung

- Objekt-Referenzen sind nicht einfach "dumme" Adressen sondern "smarte" Objekte.
- Smart Pointer merken selber
 - wenn ihnen ein neuer Wert zugewiesen wird.
 - wenn Sie nicht mehr zugreifbar sind (out of scope gehen).
- In Java mit einer assign-Methode implementiert
- In C++ mit Operator Overloading; kann wie normaler Zeiger verwendet werden.
- Vorteil: keine Fehler beim Erhöhen / Erniedrigen des Referenz-Zählers, da automatisch.
- Nachteile: wie Referenzzählung (keine zyklischen Datenstrukturen)

```
class MySmartRef {  
    MyObject ref;  
  
    void assign (MyClass obj) {  
        if (ref != null) ref.release();  
        ref = obj;  
        if (ref != null) ref.addRef();  
    }  
  
    MyObject get() {  
        return ref;  
    }  
}
```

```
class Test {  
    MySmartRef left = new MySmartRef();  
  
    void foo() {  
        left.assign(new MyClass());  
        left.get().foo2();  
        left.assign(null);  
    }  
}
```

Automatische Speicherverwaltung Vollautomatische Speicherfreigabe

- System kann selbständig feststellen, ob Speicher noch benötigt wird.
- In C/C++ praktisch unmöglich:
 - es kann mit Pointern gerechnet werden.
 - es sind unsichere Casts möglich.
 - Unions: Mehrfachbelegung von Speicher.
- In Java verboten -> automatische Speicherverwaltung möglich.

Speicher kann freigegeben werden, wenn er nicht mehr **direkt oder indirect referenziert** (i.e. angesprochen) werden kann.

- Der Speicherverwalter muss für diesen Zweck die **Referenzketten** traversieren.
 - Alle Wurzelobjekte:
 - alle **statischen Variablen**.
 - alle Variablen die im Moment des Aufrufs des Speicherverwalters sich auf dem **Stack** befinden.
 - weiterverfolgen der Kette:
 - innerhalb der Objekte alle **Referenzen** auf weitere Objekte kennen.
- Informationen über die Objekte selber werden als **Laufzeitinformationen** bezeichnet
 - in Java Zugriff über Klassen von: `java.lang.reflect.*`

3. Mark-Sweep-GC (Garbage Collection)

- Speicher wird nicht sofort freigegeben sondern erst bei Bedarf.
- Suche nach Blöcken, die freigegeben werden können, in zwei Phasen:

1. Mark

- von den Wurzelobjekten ausgehend.
- alle erreichbaren Blöcke werden markiert

2. Sweep

- sequentiell durch den Heap gehend
- alle nicht markierten Blöcke werden freigegeben.
- die Markierung von markierten Blöcken wird gelöscht .

+ Vorteile:

- + **keine zusätzlichen Operationen bei Pointer-Zuweisungen nötig.**
- + **zyklische Datenstrukturen können aufgelöst werden.**

- Nachteil:

- Aufwand
- das Programm muss während der Mark-Sweep Phase gestoppt werden.
- es entstehen Löcher (Fragmentierung).

3. Mark-Sweep GC Algorithmus

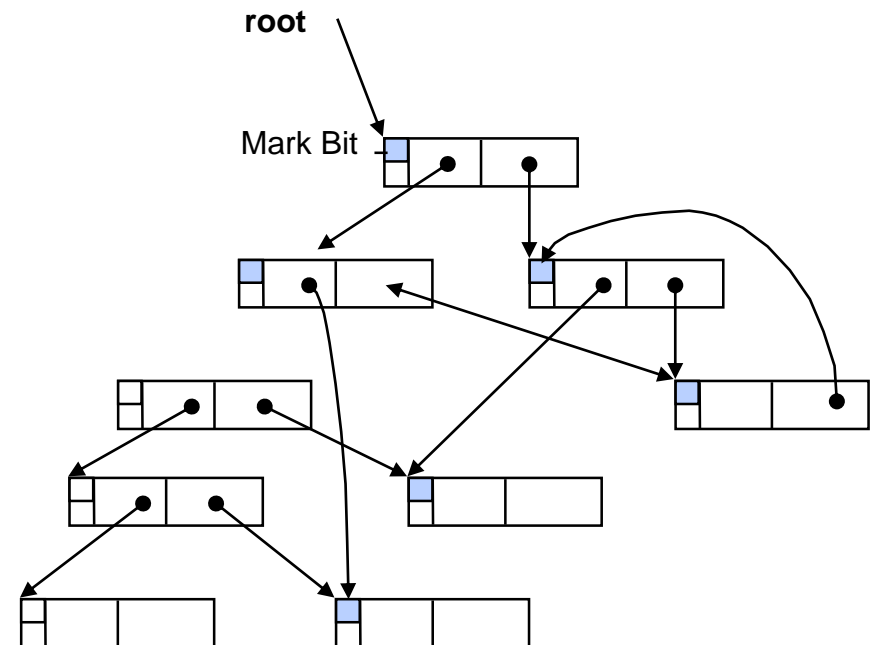
Mark-Sweep Tiefensuche:

```
gc()  
  for all N in root  
    mark (N)  
  sweep()
```

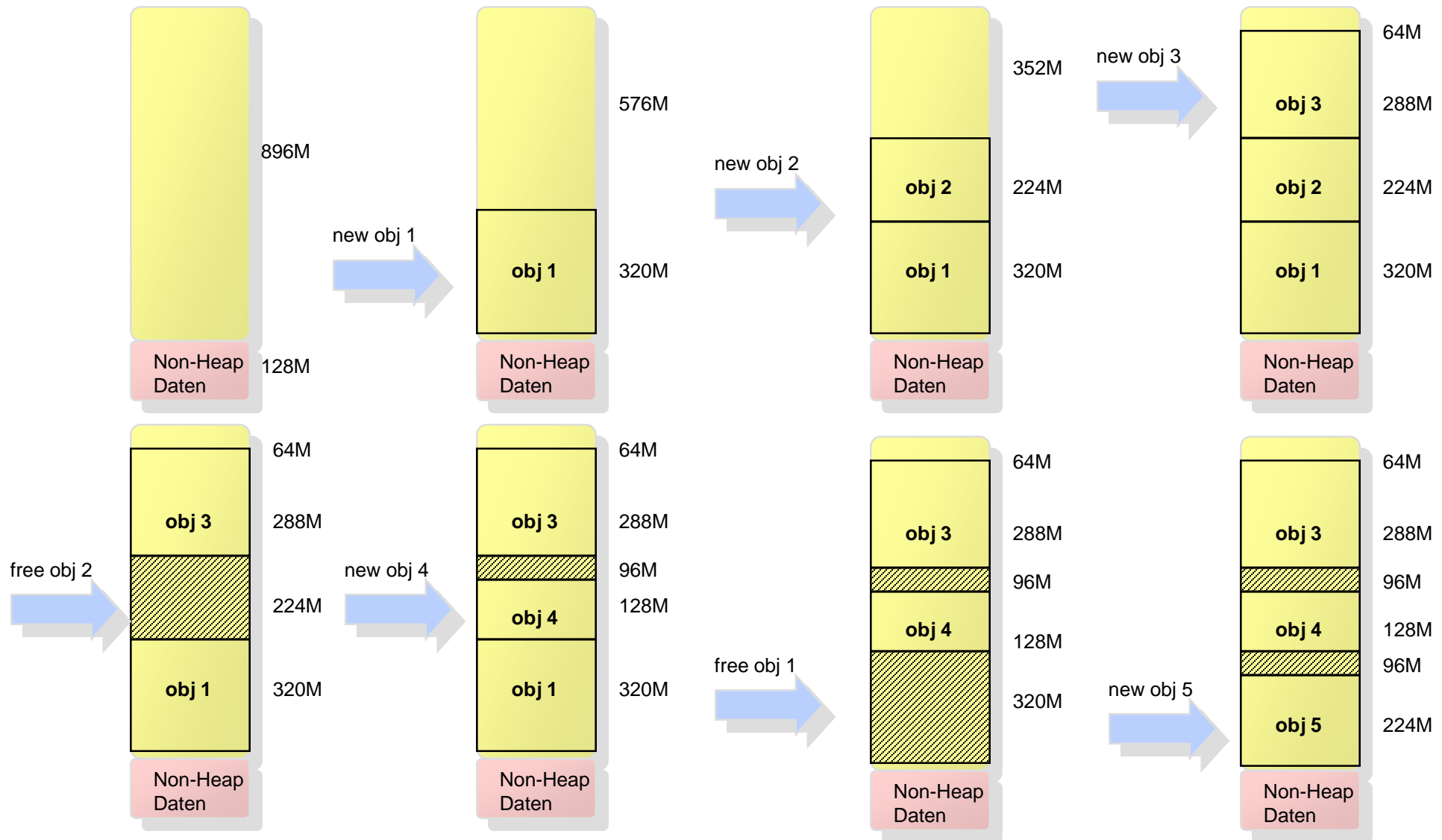
```
mark(N) =  
  if mark_bit(N) == unmarked  
    mark_bit(N) = marked  
  for all M in Children(N)  
    mark(M)
```

```
sweep()  
  for all N in heap  
    if mark_bit(N) == unmarked  
      free(N)  
    else  
      mark_bit(N) = unmarked
```

Belegt-Liste



Problem: Fragmentierung des Heaps



Automatische Speicherverwaltung
Vollautomatische Speicherfreigabe
Kompaktierende Speicherverwaltung

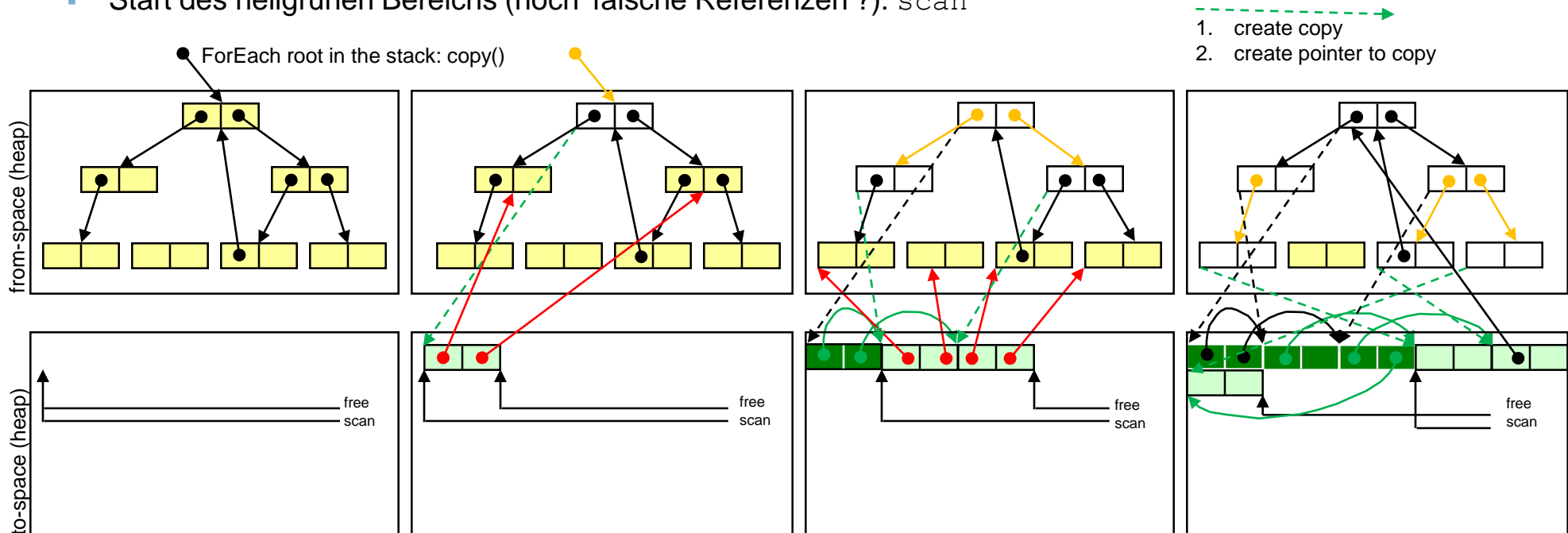
4. Copying GC

- Der Speicher wird in zwei gleiche Teile (Semi Spaces) aufgeteilt
 - der eine Semi-Space enthält die aktuellen Daten.
 - der andere Semi-Space enthält die obsoleten Daten.
- Neue Bereiche/Daten werden im aktuellen Semi-Space angelegt
- Wenn kein Platz mehr im aktuellen Semi-Space
 - es werden alle noch referenzierten Daten in den anderen Semi-Space kopiert.
 - die Rollen der Semi-Spaces wird vertauscht.
- + Vorteil:
 - + es entstehen keine Löcher.
 - + die Suche nach freien Blöcken entfällt (belegter Bereich ist kompakt).
- Nachteil:
 - es wird doppelt so viel Speicher benötigt.
 - -> virtuellen Speicher verwenden.

4. Copying GC Algorithmus

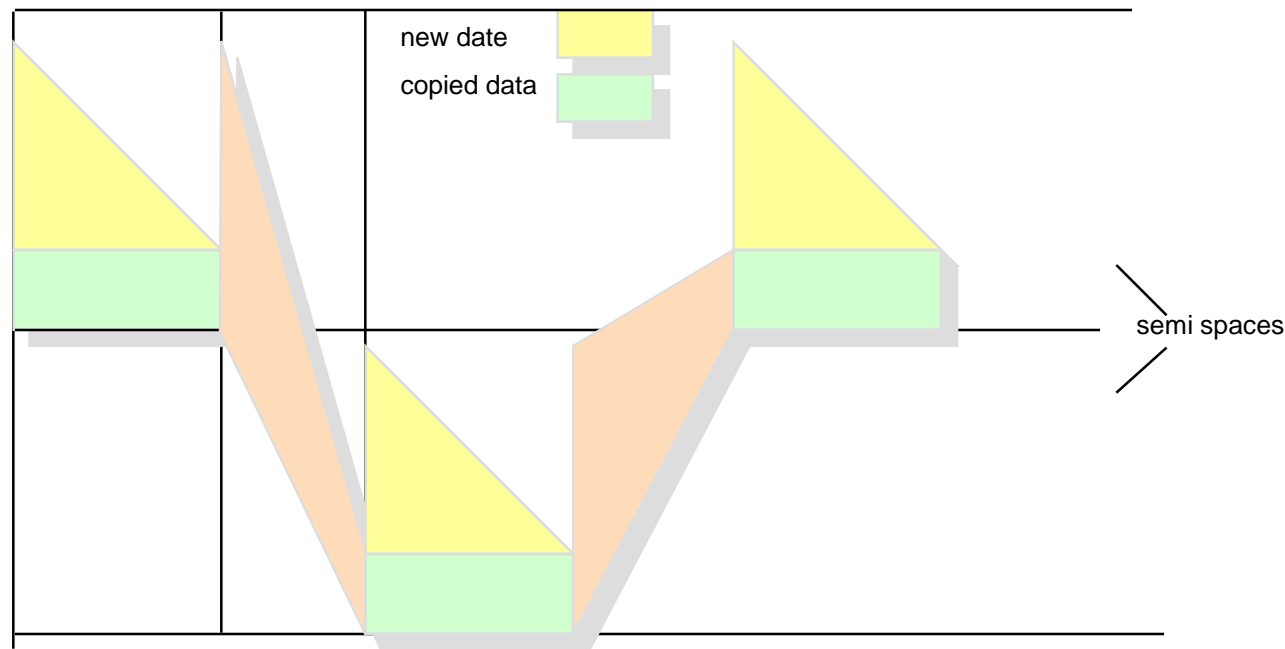
Cheney's Copying Algorithm (1970): Breitensuche

- drei Farben von Knoten.
 - weiss: Kopierte Knoten im alten Semi-Space; am Schluss der GC Phase gelöscht.
 - hellgrün: in neuen Semi-Space kopiert, aber Referenzen auf Nachfolger noch in den alten Semi-Space.
 - dunkelgrün: in neuem Semi-Space kopiert, sowie direkte Nachkommen ebenfalls.
- nur zwei Zeiger
 - Start des freien Bereichs: `free`
 - Start des hellgrünen Bereichs (noch 'falsche Referenzen'?): `scan`



4. Performance, zeitliches Verhalten

- Es muss jeweils der ganze Speicher durchsucht werden
 - relativ langer Unterbruch des Programms notwendig.
- Es wird in einen neuen Speicherbereich umkopiert
 - Lokalität des RAM Zugriffs geht verloren -> schlechte Performance (Begründung: RAM Caches; auf Disk ausgelagerter Speicher / Virtual Memory).

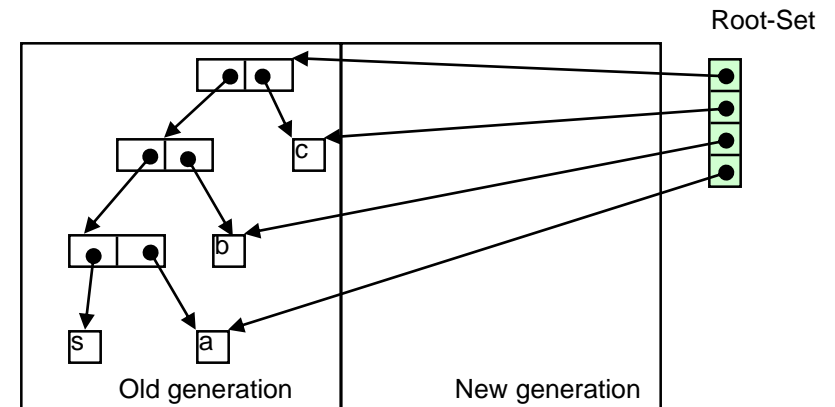
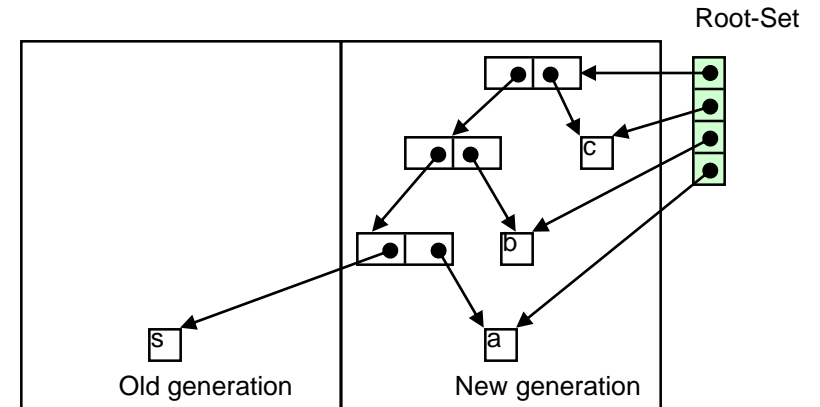


5. Generational GC

- Nachteile von Mark-Sweep und Copying
 - es muss immer der ganze Speicher durchlaufen werden.
 - Programm muss während dieser Zeit angehalten werden -> bei interaktiven Systemen störend.
- Beobachtung:
 - Lebensdauer von Objekten in Programmen ist sehr unterschiedlich.
- Schwache Generationshypothese
 - most objects die young
- Starke Generationshypothese (umstritten)
 - the older an object is the less likely it is to die
- Idee:
 - die Objekte in Generation unterteilen.
 - neue Generationen häufiger nach freizugebenden Objekten durchsuchen.

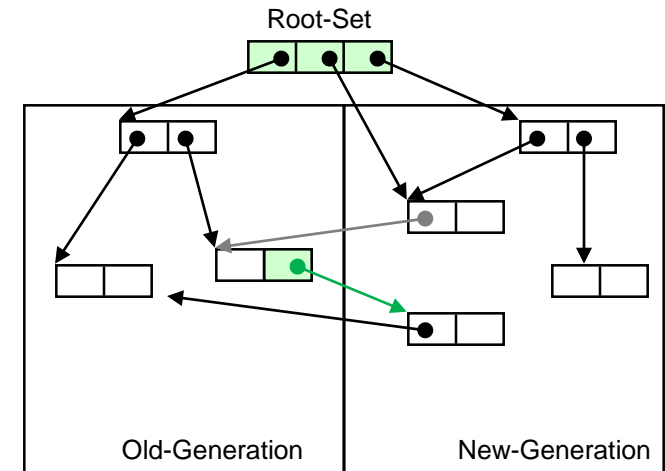
5. Generational-Copying GC Algorithmus

- neue Objekte entweder im New-Generation-Space angelegt. z.B.:
`root[0] = new R(a,b)`
- Alle noch referenzierten Elemente im New-Generation-Space werden in den Old-Generation-Space kopiert; die übrigen werden gelöscht.
- Ziel: Objekte im New-Generation-Space können unabhängig vom Old-Generation-Space eingesammelt werden.



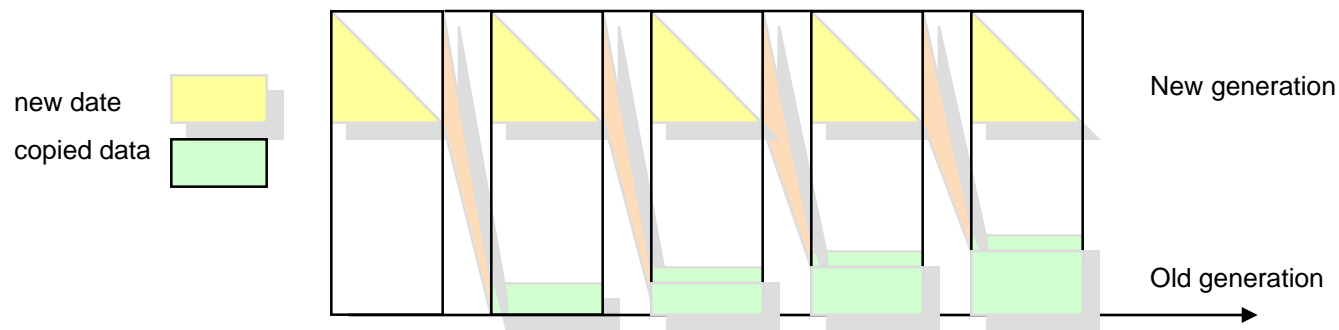
5. Generational-Copying GC Algorithmus

- **Referenzen von New Generation nach Old Generation**
 - werden nicht speziell behandelt.
- **Referenzen von Old Generation nach New Generation**
 - Referenzen aus dem Old-Generation-Space müssen zum **Root-Set** hinzugenommen werden (beim Copying-GC-Algorithmus waren alle zum Starten verwendeten Referenzen im Stack).
- Probleme mit dem Root-Set:
 1. ein Objekt wird durch den GC-Algorithmus in den Old-Generation-Space verschoben: Referenz wird gegebenenfalls in das Root-Set übernommen.
 2. eine Referenz eines Objekts im Old-Generation-Space wird auf ein Objekt im New-Generation-Space gewechselt: Root-Set muss allenfalls ergänzt werden.

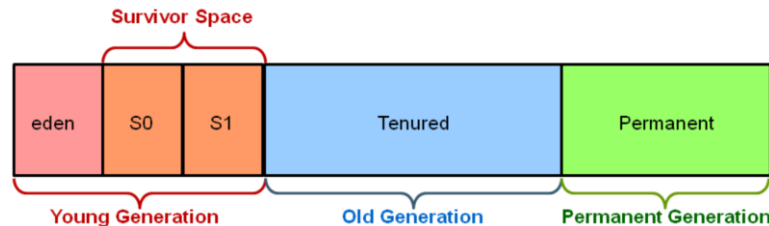


5. Performance, zeitliches Verhalten

- New-Generation-Space kann unabhängig von Old-Generation-Space behandelt werden: Minor Collection
- Beim Old-Generation-Space müssen alle Referenzen betrachtet werden: Major Collection
 - einfache Lösung , da i.d.R. wenige Objekte im New-Generation-Space: Alle Objekte des New-Generation-Space werden zum Root-Set des Old-Generation-Space hinzugefügt.
- Häufig Minor Collection, selten Major Collection durchführen
 - kürzere Unterbrüche Notwendig
 - bessere Lokalität der Zugriffe



Aufbau des Hotspot Heaps (Oracle Doku)



- Young Generation is where all new objects are allocated and aged.
 - When the young generation fills up, this causes a minor garbage collection.
 - Stop the World Event - All **minor garbage collections** are "Stop the World" events.
- The Old Generation is used to store long surviving objects.
 - Eventually the old generation needs to be collected, also a "Stop the World" event.
 - This event is called a **major garbage collection**.
- The Permanent generation contains
 - Metadata required by the JVM to describe the classes and methods used in the application.
 - The permanent generation is included in a **full garbage collection**.

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

Java Klassen

- Datenstrukturen, die Sammlungen (Hastable, List usw.) von andern Objekten beinhalten, benötigen interne Referenzen auf diese Objekte.
- Werden von GC gefunden und traversiert.
 - Objekte in Sammlungen können nicht freigegeben werden, sobald (ausser Sammlung) keine Referenzen mehr auf diese Objekte vorhanden sind – obwohl diese allenfalls nicht mehr benötigt werden.
 - Sehr viele Referenzen müssen traversiert werden.
- Lösung
 - Einführung von Referenzen, die nicht traversiert werden: Weak References
 - Objekt wird gelöscht, auch wenn noch Weak References auf dieses zeigen.
- + Vorteil:
 - + Objekte in Sammlungen können freigegeben werden.
- Nachteil
 - Problem mit Dangling Pointer

- Referenzen auf Objekte, die Java - wenn sie nicht anderweitig referenziert werden - freigeben kann. Klasse **WeakReference** in package `java.lang.ref`:

▪ <code>WeakReference(Object referent)</code>	Konstruktor
▪ <code>void clear()</code>	löscht Referenz auf das Objekt
▪ <code>Object get()</code>	retourniert referenziertes Objekt

- Falls Objekt gelöscht wird, dann wird `WeakReference` zu null gesetzt

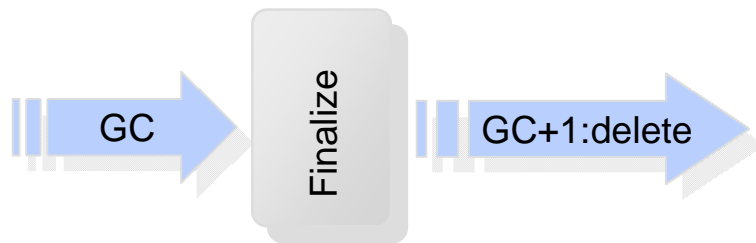
```
public class TestWR {  
    WeakReference next;  
  
    public static void main(String[] args) throws Exception {  
        TestWR a = new TestWR();  
        a.next = new WeakReference(new TestWR());  
        System.out.println(a.next.get());  
        System.gc(); // run garbage collector  
        Thread.sleep(1000);  
        System.out.println(a.next.get());  
    }  
}
```

Kombination von automatischer mit nicht-automatischer Speicherverwaltung: Finalizer

- Es gibt Ressourcen, die nicht automatisch verwaltet werden
 - Dateien, Fenster, usw. generell: **Betriebssystem-Ressourcen**
- Müssen explizit wieder freigegeben werden:
 - z.B. `BufferedReader br = new BufferedReader(new FileReader(path)); br.close();`
 - bei SWT (Eclipse GUI) müssen die Widgets mittels `dispose()` explizit freigegeben werden
- Spätestens aber wenn Objekt gelöscht wird, das die Ressource verwendet.
- Objekt wird über spezielle Methode benachrichtigt, bevor es freigegeben wird:
finalize()
 - z.B. `resource.close()` in Methode `finalize()` aufrufen, wird auch im Falle einer Exception ausgeführt.
- Probleme bei der Anwendung von Finalizer:
 - Aufruf erst später (bei statischen Objekt gar nicht)
 - Reihenfolge nicht bestimmt.
 - Finalizer-Routine sollten so kurz wie möglich sein (Programm wird während GC Phase gestoppt).
 - die Finalizer der Oberklasse muss am Schluss mit `super.finalize()` aufgerufen werden (nicht automatisch wie beim Konstruktor).

"Stilbruch": SWT aus alter Smalltalk GUI Bibliothek entstanden

- Problem: `finalize()` kann wieder Referenz, z.B. auf sich selber, setzen
 - *Objekte dürfen erst in der nächsten GC Phase freigegeben werden.*
- Freigeben der Objekte mit Finalizer ist zweistufig:
 1. GC: Objekt nicht referenziert: `finalize()` aufrufen und in Finalized-Liste übertragen.
 2. GC+1: alle Objekte in Finalized-Liste, die nicht markiert wurden werden freigeben.



```
mark(N)
    if mark_bit(N) == unmarked
        mark_bit(N) = marked
        for all M in Children(N)
            mark(M)
```

```
sweep()
    for all N in Heap
        if mark_bit(N) == unmarked
            if (!N in Finalized)
                N.finalize()
                addToFinalized(N)
            else
                delete(N)

removeFromFinalized(N)
    else mark_bit(N) = unmarked
```

Setzen und Anzeige von GC Informationen

- XAuf der Kommandozeile kann angegeben werden:
 - Initiale Heapgrösse: -Xms (unabhängig vom GC)
 - Maximale Heapgröße -Xmx (unabhängig vom GC)
 - Protokollieren von Garbagekollektorläufen: -XX:+PrintGC oder -XX:+PrintGCDetails
- `java -Xms500m -Xmx800m -XX:+PrintGC Main`

```
[GC 38835K->38906K(63936K), 0.1601889 secs]
[GC 39175K(63936K), 0.0050223 secs]
[GC 52090K->52122K(65856K), 0.1452102 secs]
[GC 65306K->65266K(79040K), 0.1433074 secs]
```

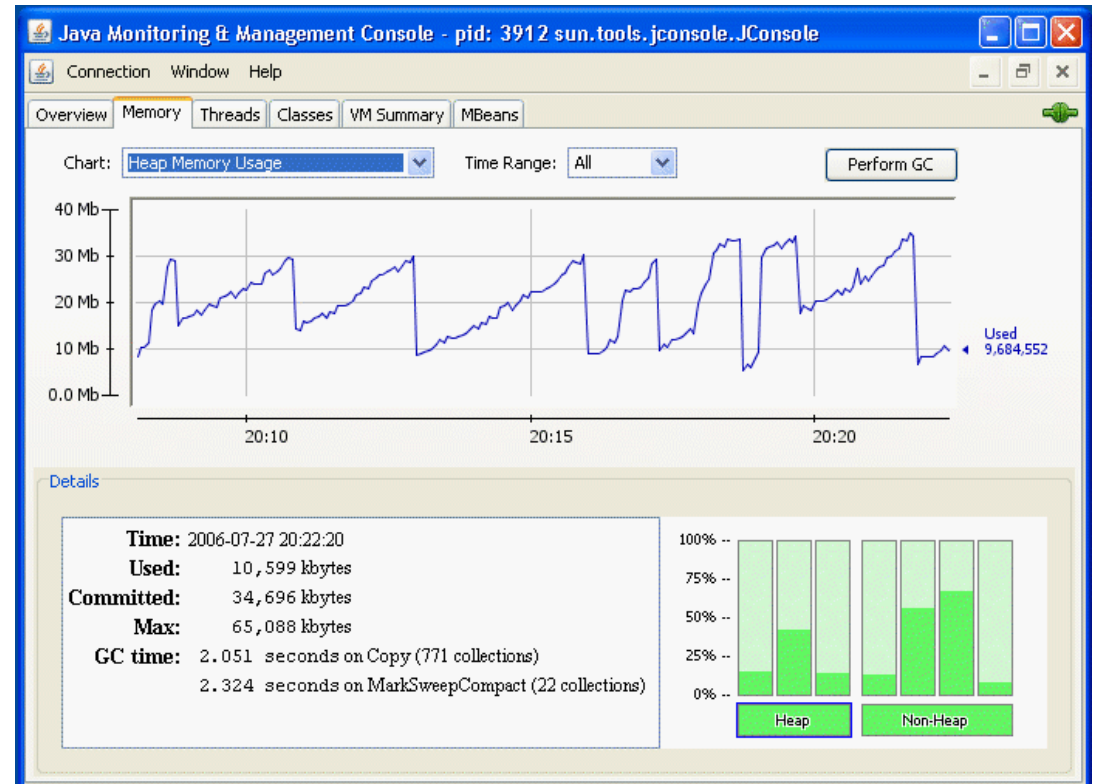
- `Runtime.getRuntime().totalMemory();` liefert den Gesamtspeicher
- `Runtime.getRuntime().freeMemory();` liefert den freien Speicher
- `System.gc();` ruft den G.C. bei nächster Gelegenheit auf

GC als Thread implementiert ->
dem System Zeit geben mit z.B.
`Thread.sleep`

- JConsole: das Werkzeug zur Java Systemüberwachung

- im <jdk>\bin Verzeichnis
- zur Anzeige von:
 - Memory
 - Threads
 - Prozessor-Auslastung
 - ...

- Auch auf entfernten Maschinen möglich



- Speicherverwaltung
 - Stack, Heap, Statische Daten
 - Argumente/Optionen, Programm Code, Laufzeitsystem
- Einfache Speicherverwaltung
 - Fehler bei Speicher-Anforderung / Freigabe
- Automatische Speicherverwaltung
 - Einfache Verfahren
 - Referenzzählung
 - Smart Pointer
 - Automatische Verfahren
 - Mark-Sweep
 - Kompaktierende
 - Copying
 - Generational
- Weak References
- Finalizer
- Werkzeuge