

# **Application Performance Management**

## **Frühling 2022**

### **Garbage Collection**

*Zoltán Majó*

# Vorstellung

## Berufserfahrung

- Seit 2017: Senior Software Engineer / Technical Consultant / Projekt Manager  
Ergon Informatik AG, Zürich
- 2014-2017: Compileringenieur Java Virtuelle Maschine  
Oracle Corporation

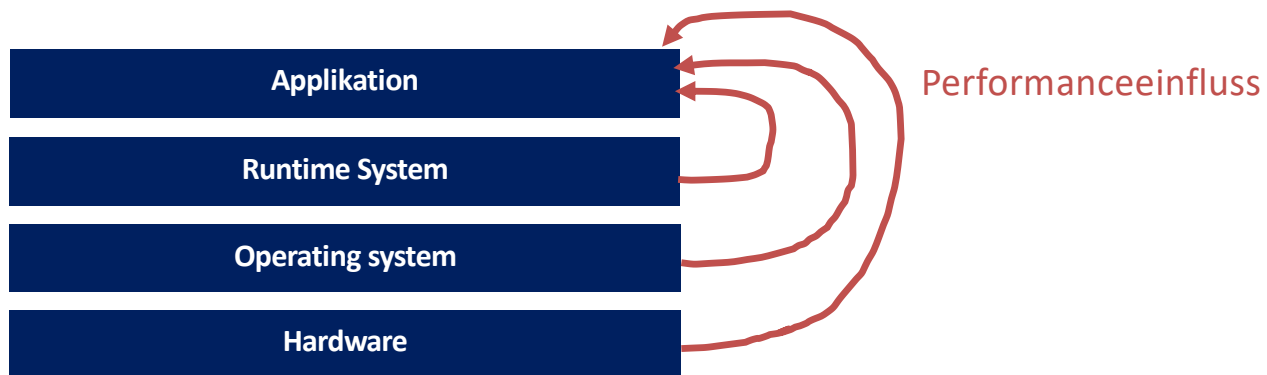
## Studium

- 2008-2014: Doktorat an der ETH Zürich
- 2002-2007: Informatikstudium, TU Cluj, Rumänien

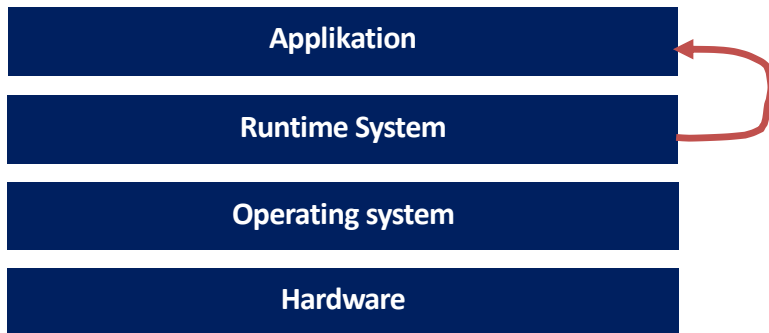
## Lehre

- 2017: Vorlesung Compilerbau an der ETH Zürich
- Seit 2018: APM an der FHNW

# Aufbau einer typischen Applikation



# Unser Fokus



Performanceeinfluss

# Einflussfaktoren durch einen Managed Runtime

- |                                    |  |
|------------------------------------|--|
| • <b>Garbage Collection</b>        | Wird am heutigen APM-Anlass betrachtet |
| • <b>Just-in-Time Kompilierung</b> | Wird im nächsten APM-Anlass betrachtet |
- **Class Loading**
  - **Optimierungstechniken für Managed Runtimes**
    - z.B. Inline Caching
  - ...

# Basis für praxisnahe Diskussion: Java Hotspot VM

- Konkretes Beispiel eines Managed Runtime
- Breite Verwendung weltweit
- Prinzipien gelten für andere Systeme auch

# Garbage Collection

- **Was ist GC?**
- **Was beeinflusst die Performanz von GC?**
  - Komplexität des GC-Algorithmus
  - Implementierung des Algorithmus
    - Generational GC
    - Serielle, parallele und nebenläufige GC
- **Wie beeinflusst GC die Performanz von Applikationen**
  - Wichtige Performanzmerkmale
  - Performanzerhöhung durch GC Tuning

Danke an Prof. Dr. Thomas R. Gross (ETH Zürich) für Teil der Folien

# Speicherverwaltung

## **Viele moderne Programmiersprachen unterstützen dynamische Speicherallokation**

- Programme können z.B. Records, Arrays und Objekte zur Laufzeit allozieren

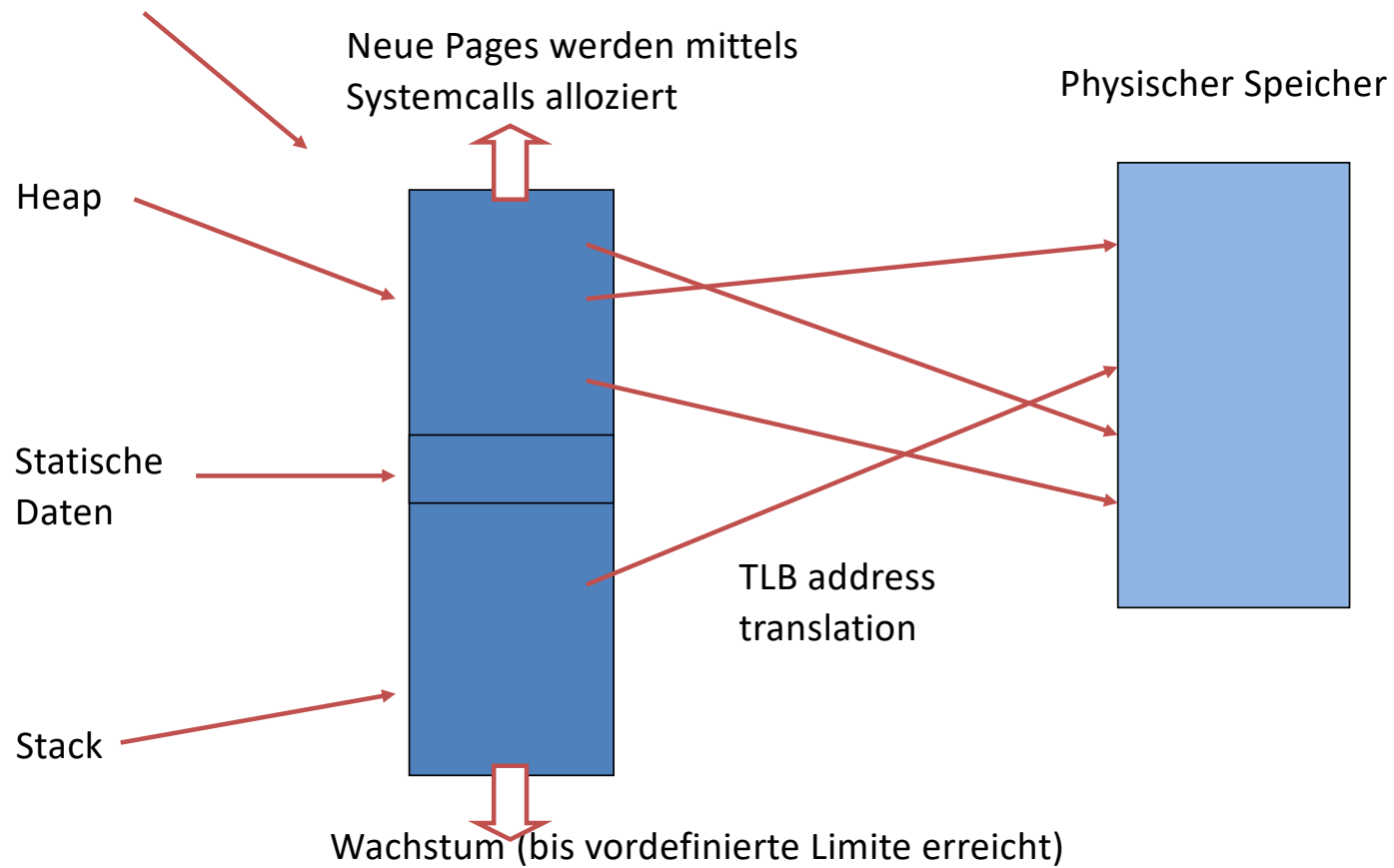
## **Die Programmiersprache muss mit der Zurückforderung und Recycling allozierter Speicher umgehen können**

- Teil der Spezifikation der Programmiersprache
- Aufgabe dem Runtime System (z.B., der Java Virtuelle Maschine) überlassen



# Speicherlayout

Virtueller Speicher  
(per Prozess)



# Speicherplatz

## Virtueller Speicher ist unbegrenzt

- Zumindest konzeptuell

## Physischer Speicher ist begrenzt

- Limite kann vom Betriebssystem gesetzt werden
- Limite kann beim Prozessstart gesetzt werden (wegen anderen Prozessen)
- Adressraum ist limitiert

## Performanz ist wichtig

- Festplatten sind weniger performant als Hauptspeicher

## Unbenutzte Daten müssen entfernt werden

→ «Garbage Collection» (GC)

# GC

## Was ist «Garbage» (dt. Müll)?

- Ein Objekt im Programm ist Müll, wenn das Objekt von keiner Berechnung wieder verwendet wird.

## Ist es einfach festzustellen, welche Objekte Müll sind?

- Nein. Es ist unentscheidbar. Zum Beispiel:

```
v = new Object();  
if (long-and-tricky-computation) {  
    use v  
} else {  
    don't use v  
}
```

# GC (Forts.)

Da es schwierig ist festzustellen, welche Objekte Müll sind, unterstützen Programmiersprachen unterschiedliche Ansätze

**Ansatz 1:** Der Programmierer muss sich darum kümmern

- Explizite Allokation/Deallokation

**Ansatz 2:** Das Laufzeitsystem muss sich darum kümmern

- Automatisch
- Viele Algorithmen

# Ansatz 1: Explizite Speicherverwaltung

Verwaltung des Speichers mittels einer Bibliothek

Programmierer entscheidet wann und wo Speicher alloziert/dealloziert wird

```
void* malloc(long n)  
void free(void *addr)
```

**Wenn nötig, die Bibliothek beantragt mehr Pages vom Betriebssystem**

- Mittels Systemcalls

# Vorteile/Nachteile der expliziten Speicherverwaltung

## Vorteile:

- Programmierer sind schlau
- Programmierer entscheidet, wann der Mehraufwand der Allokierung akzeptabel ist

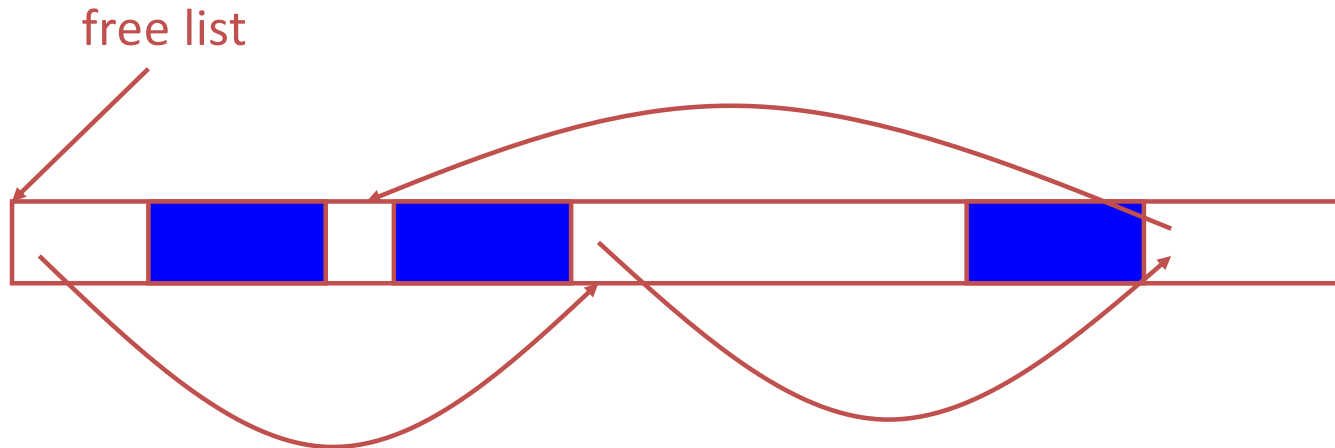
## Nachteile:

- Auch schlaue Programmierer machen Fehler
- Programmierer möchten sich nicht unbedingt mit solchen Details beschäftigen
- Automatische Speicherverwaltung kann günstig sein

# Explizite Speicherverwaltung: Details

## Wie funktioniert malloc/free?

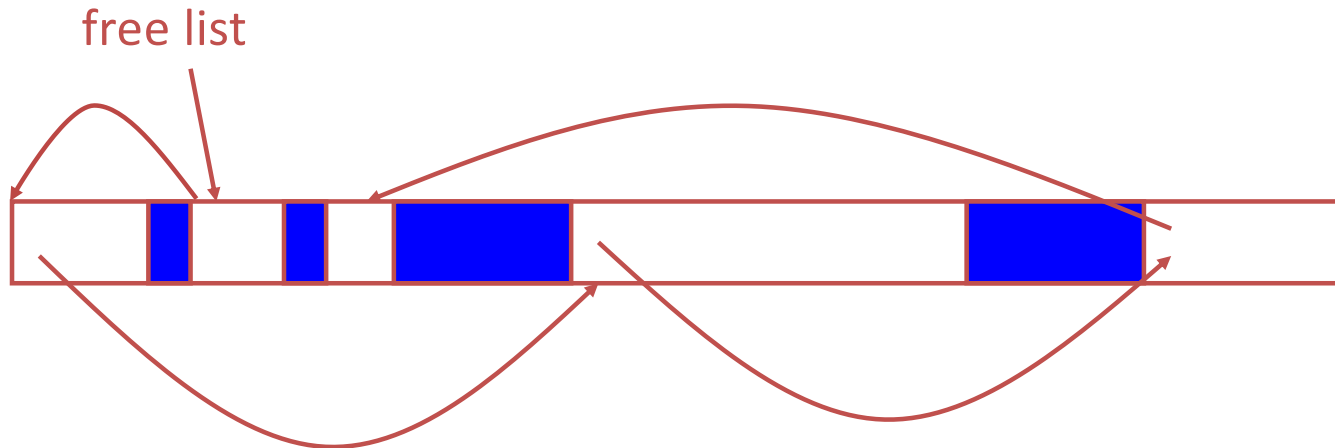
- Nicht (mehr) verwendete Speicherblöcke befinden sich in der “free list”
- `malloc`: sucht in der free list nach einem genug grossen Speicherblock
- `free`: platziert Speicherblock zum Anfang der free list



# Explizite Speicherverwaltung: Details

## Wie funktioniert malloc/free?

- Nicht (mehr) verwendete Speicherblöcke befinden sich in der “free list”
- `malloc`: sucht in der free list nach einem genug grossen Speicherblock
- `free`: plaziert Speicherblock zum Anfang der free list

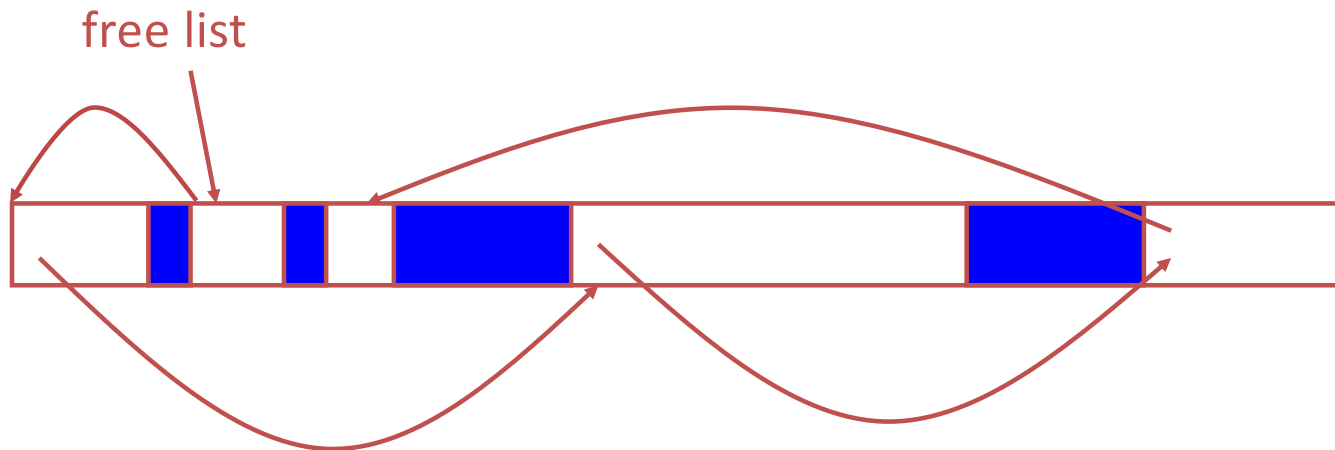




# Explizite Speicherverwaltung: Details

## Nachteile

- `malloc` is nicht für umsonst: Aufwand der Suche nach einem Block, der gross genug ist, kann signifikant sein
- Der Heap wird fragmentiert, während das Programm ausgeführt wird



# Mögliche Lösungen

## Mehrere free lists

- Eine free list für jede gegebene Blockgrösse
- Malloc und free sind beide  $O(1)$
- Mögliches Problem: Liste mit Blöcken der Grösse 4 ist verbraucht, auch wenn Blöcke der Grösse 2 und 6 verfügbar sind

## Blöcke Zweierpotenzen-Grössen

- Blöcke werden aufgeteilt um richtige Grösse zu erreichen
- Bei Freigabe werden angrenzende Blöcke zusammengeschmolzen

## Fragmentierung in jedem Fall vorhanden

- Verschwendeter Speicherplatz
- «No magic bullet»: Speicherverwaltung kostet immer was

# Automatische Speicherverwaltung – wieso?

**Programmieren mit expliziter Speicherverwaltung viel schwieriger ist als mit automatischer Speicherverwaltung**

- Konstante Sorge wegen «Dangling Pointers»
  - Instabilität, Maintenance
- Es ist unmöglich ein sicheres System zu entwickeln
  - System gibt keine Garantien
- Programmieren mit Sprachen, die automatische Speicherverwaltung unterstützen, ist einfacher
- Unterliegendes Laufzeitsystem kann den Speicher immer noch explizit verwalten

# Ansatz 2: Automatische Speicherverwaltung

**Zentrale Frage:**

**Wie wird entschieden, welche Objekte Müll sind?**

- (Ein Objekt im Programm ist Müll, wenn keine Berechnung im Programm dieses Objekt wieder verwendet.)

**Übliche Lösung: Ein Objekt ist Müll, wenn es von den "Roots" aus nicht mehr erreichbar ist**

- Roots = Register, Stack, globale statische Daten
  - Falls es vom Root aus zu einem Objekt keinen Pfad gibt, das Objekt kann nicht mehr im Programm verwendet werden und kann daher zurückgefordert werden.
- Zurückhaltende Approximation
  - Engl. «conservative approximation»

## **Ansatz 2: Automatische Speicherverwaltung (Forts.)**

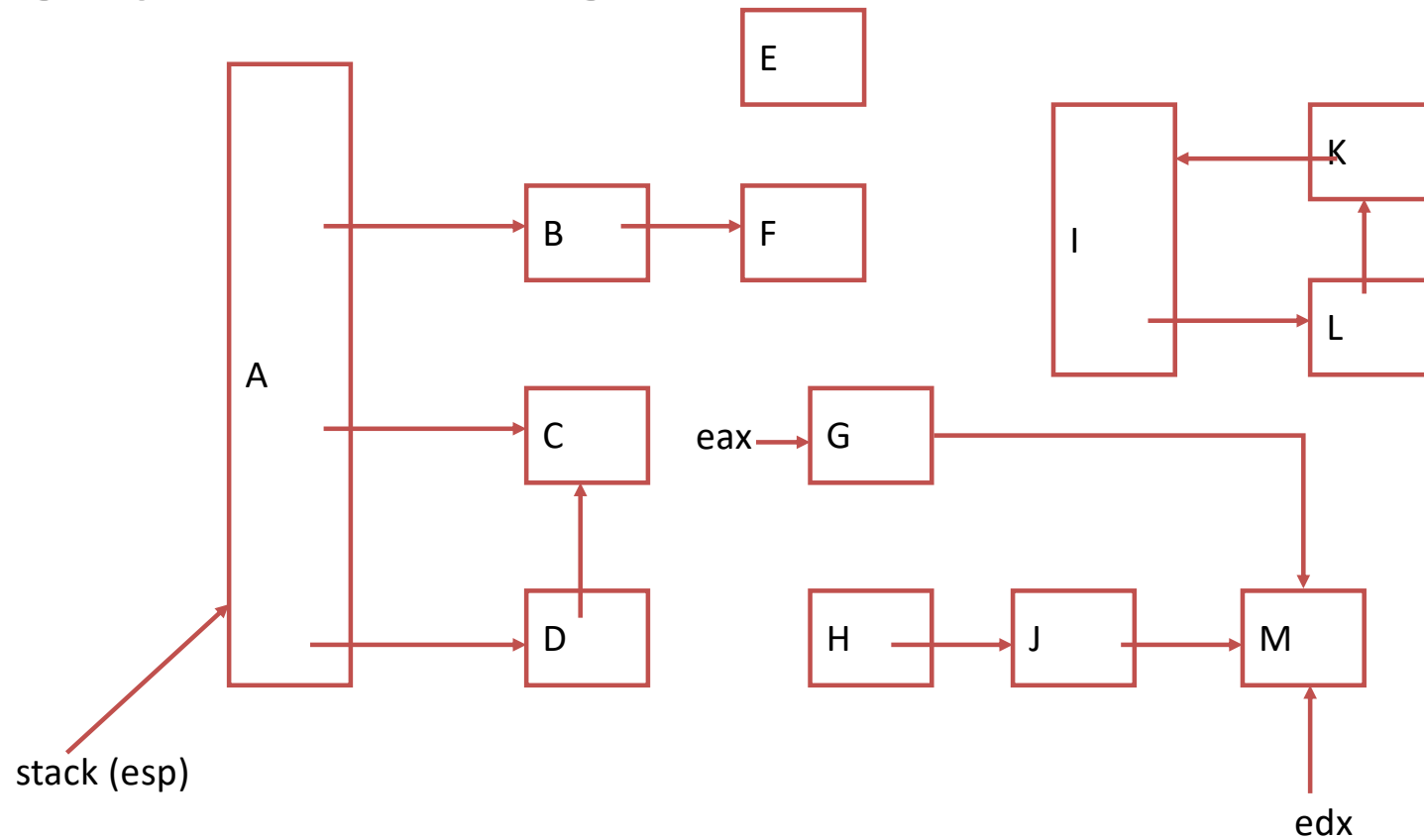
**Es gibt verschiedene Ansätze um automatische Speicherverwaltung zu realisieren**

**Die meisten Differenzen sind bezüglich**

- Wie wird entschieden, welche Objekte nicht erreichbar sind
- Wie werden (nicht) erreichbare Objekte behandelt

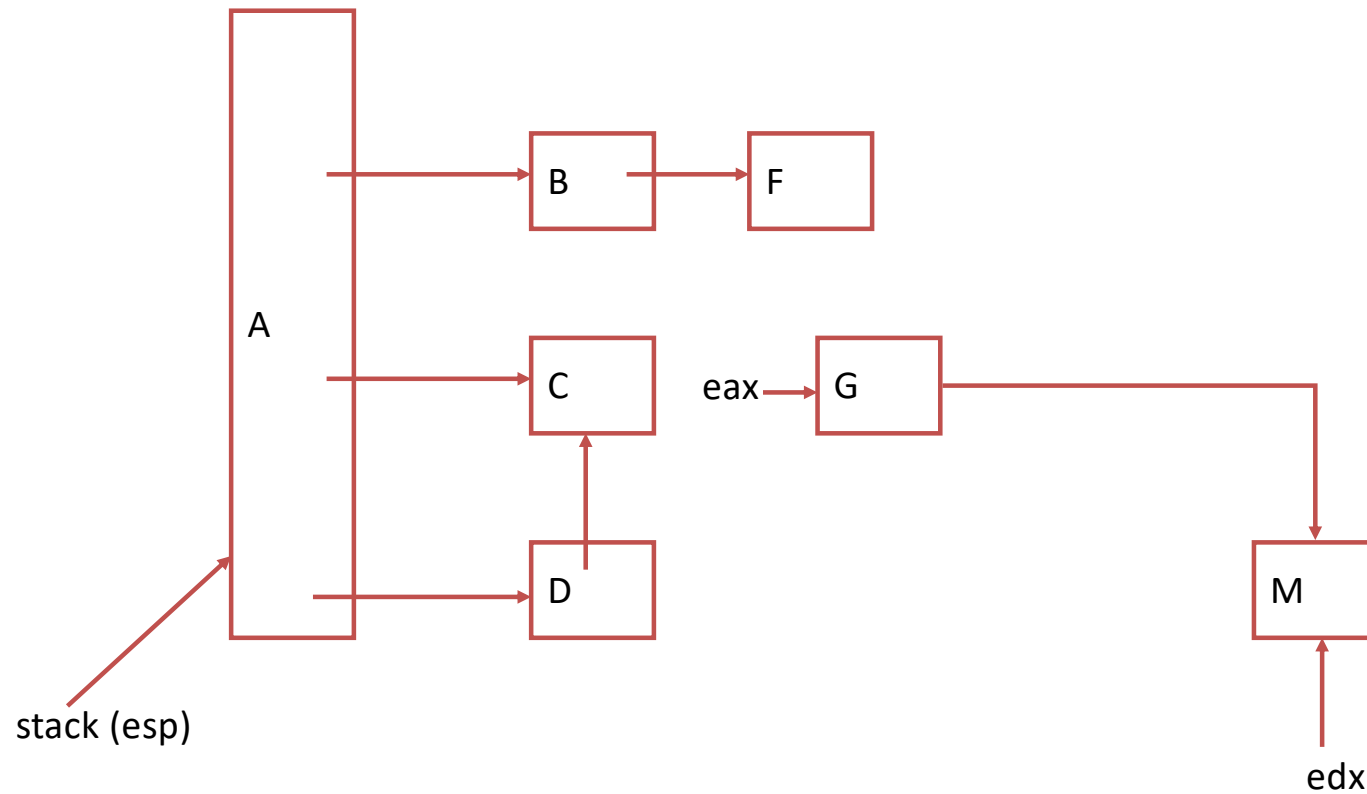
**Fokus des heutigen Anlasses: (automatische) Garbage Collection**

# Objektgraph (eines Programmes)



**Welche Objekte sind erreichbar (von den Roots aus)?**

# Objektgraph (eines Programmes) – nach GC



**Welche Objekte sind erreichbar (von den Roots aus)?**

# Diskussion: Wann kann GC passieren?

## Beispielprogramm

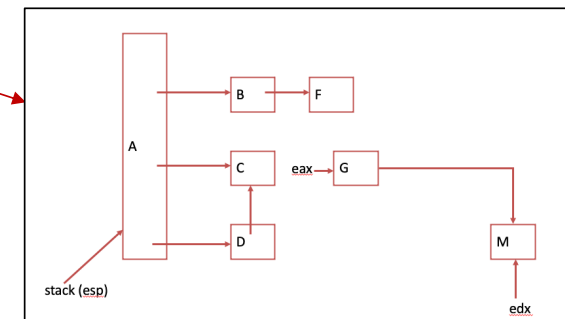
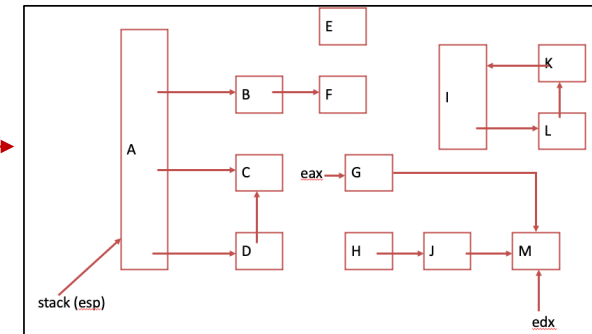
```
class Foo {  
    public static void main(String args[]) {  
        Object v;  
        v = new Object();  
        System.out.println(v);  
        System.gc();  
    }  
}
```



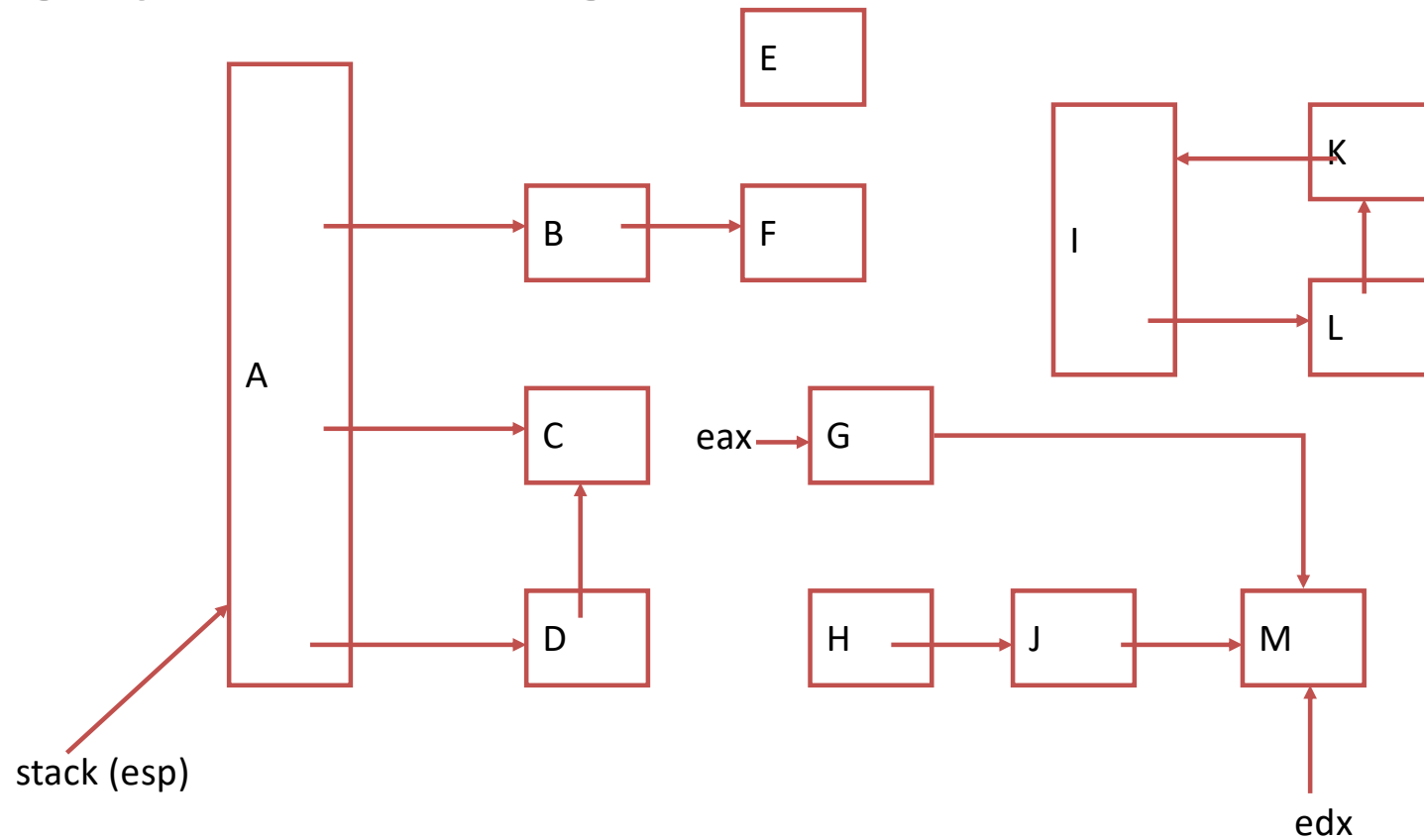
# Diskussion: Wann kann GC passieren?

## Beispielprogramm

```
class Foo {  
    public static void main(String args[]) {  
        Object v;  
        v = new Object();  
        System.out.println(v);  
        System.gc();  
    }  
}
```



# Objektgraph (eines Programmes)



**Wie können wir feststellen, welche Objekte erreichbar sind?**

# Heute diskutierte Algorithmen

**Mark & Sweep GC**

**Mark & Copy GC**

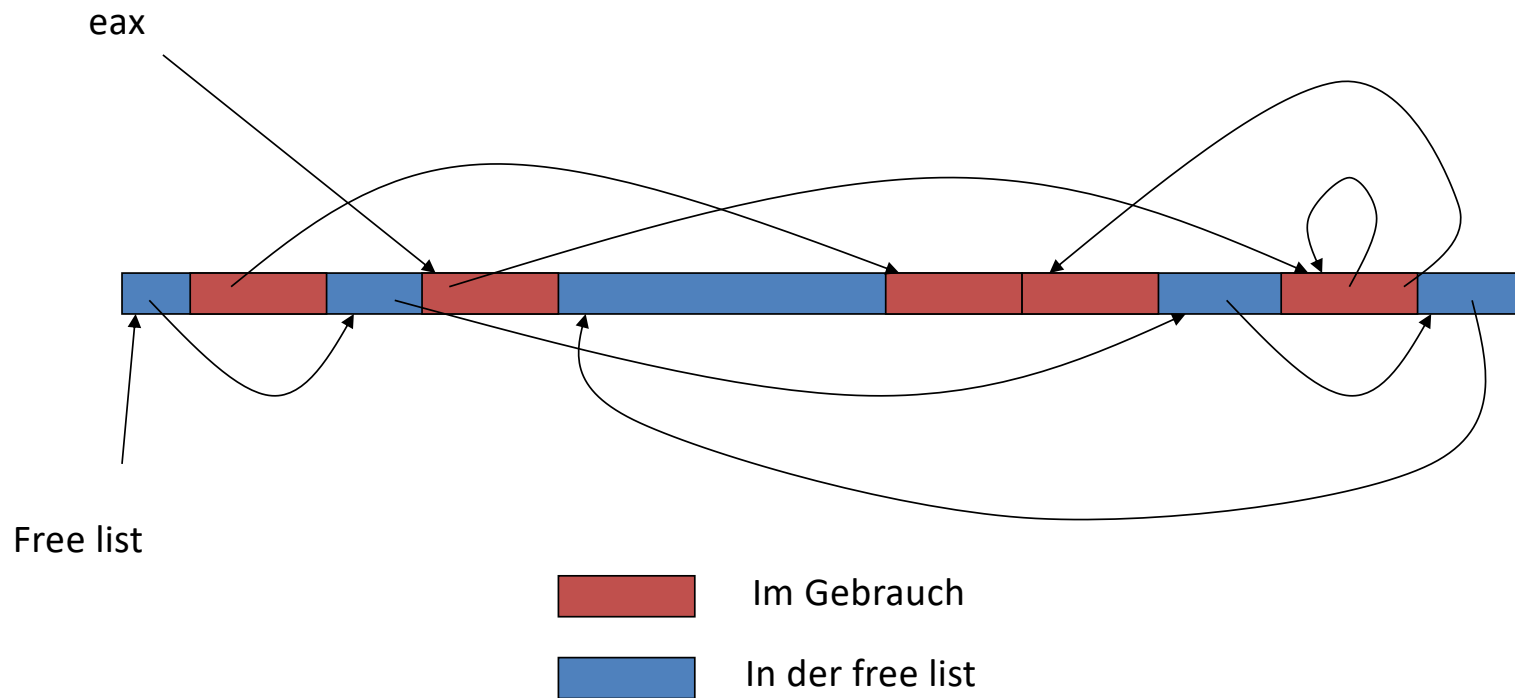
**Mark & Compact GC**

# Mark & Sweep GC

## Algorithmus besteht aus zwei Phasen

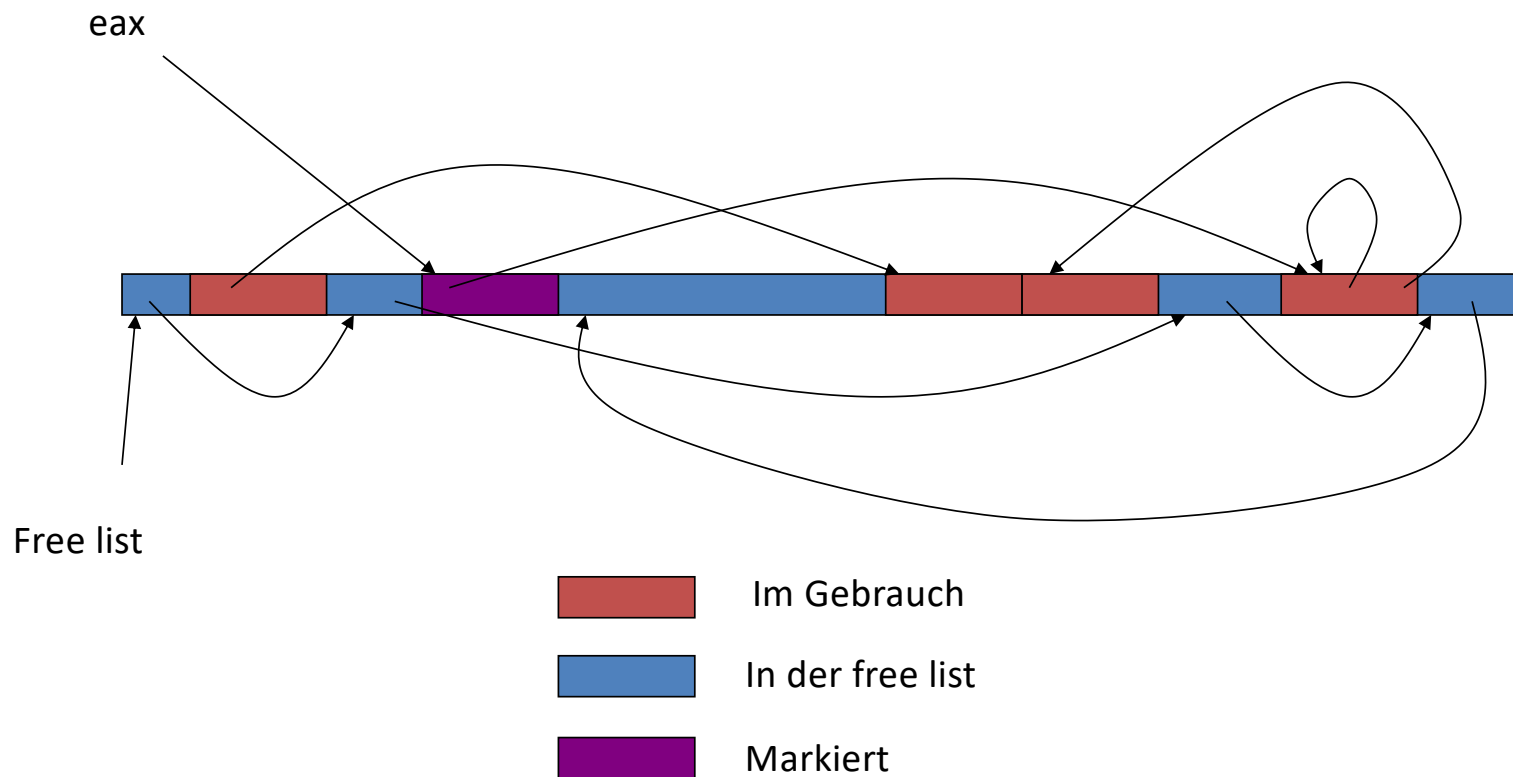
- **Mark:** Objektgraph wird "depth-first" durchquert und vom Root aus erreichbare Objekte werden markiert
- **Sweep:** Der ganze Heap wird durchquert, nicht markierte Objekte werden der free list zugefügt, die Markierung aller Objekte wird gelöscht

# Mark & Sweep: Zeitlupe



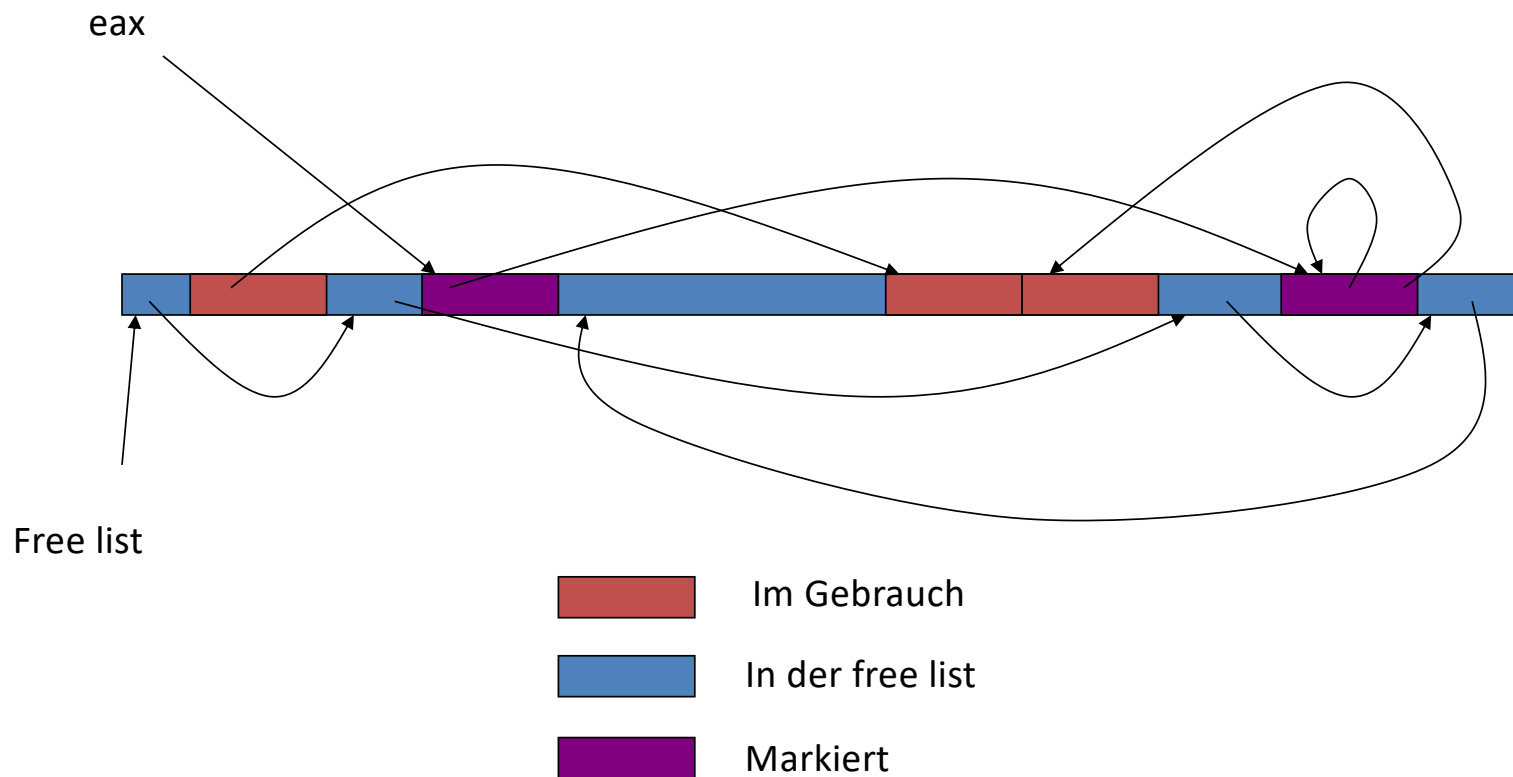
# Mark & Sweep: Zeitlupe

Mark Phase: vom Root aus erreichbare Objekte werden markiert



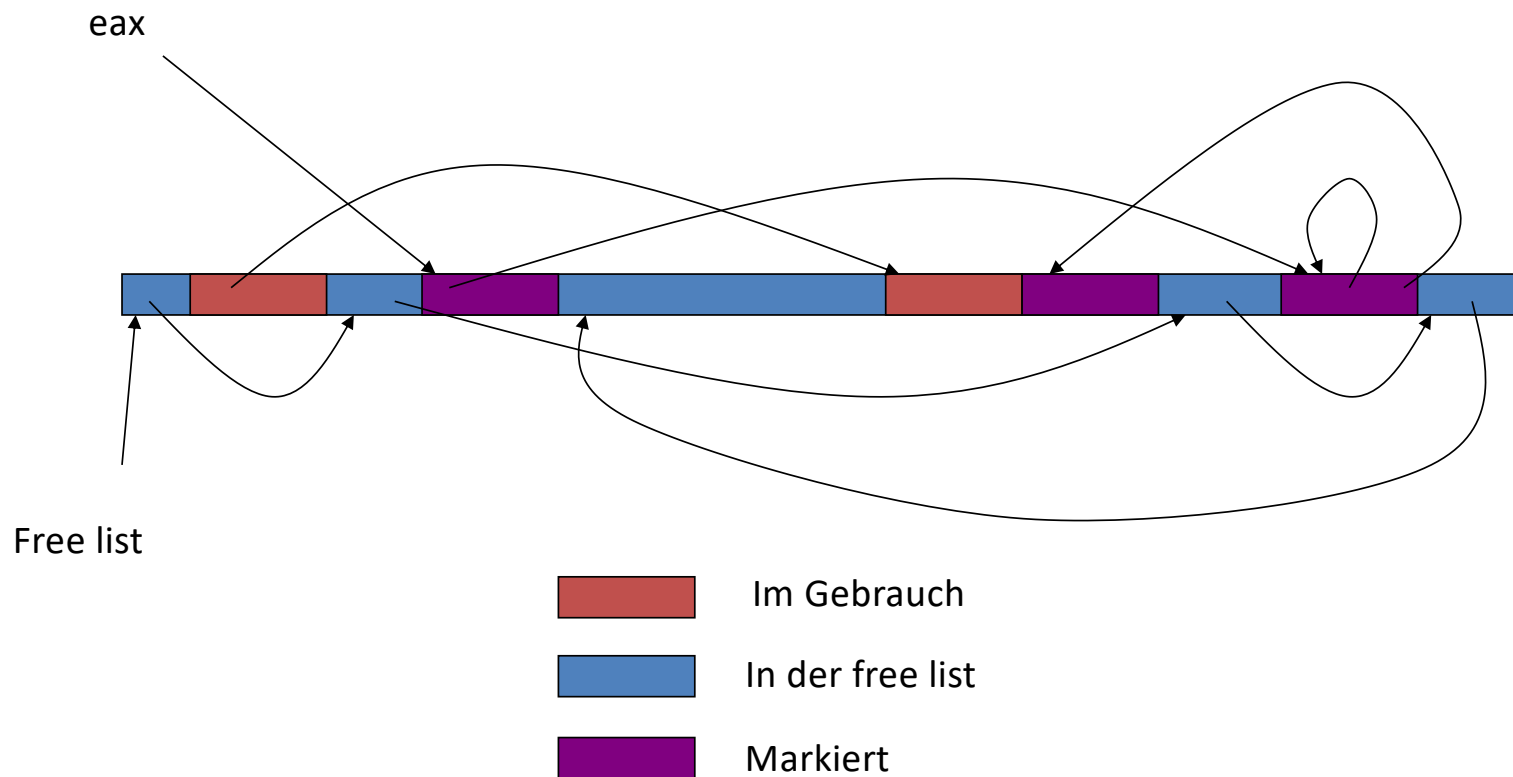
# Mark & Sweep: Zeitlupe

Mark Phase: vom Root aus erreichbare Objekte werden markiert



# Mark & Sweep: Zeitlupe

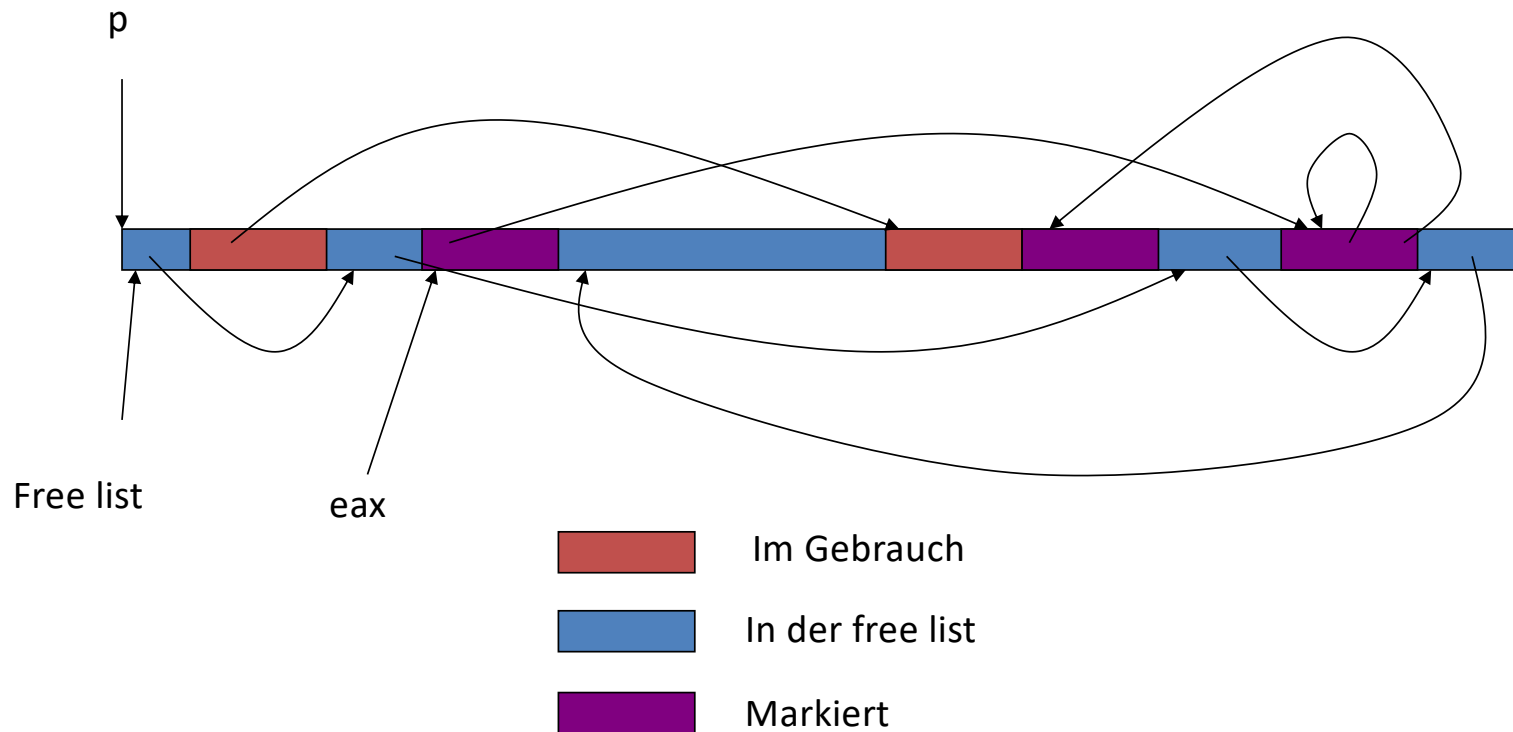
Mark Phase: vom Root aus erreichbare Objekte werden markiert





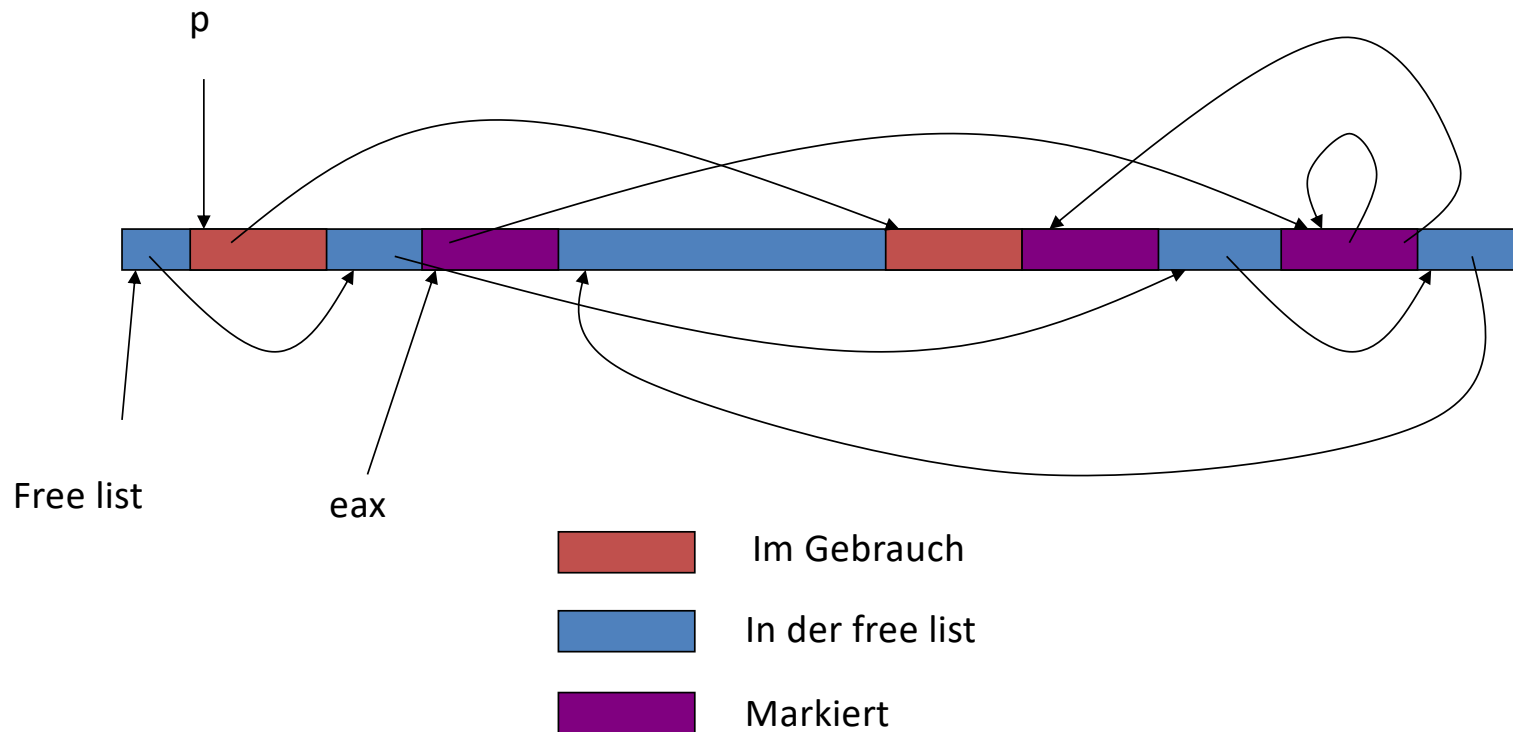
# Mark & Sweep: Zeitlupe

Sweep Phase: Sweep Pointer  $p$  aufsetzen; Sweep starten



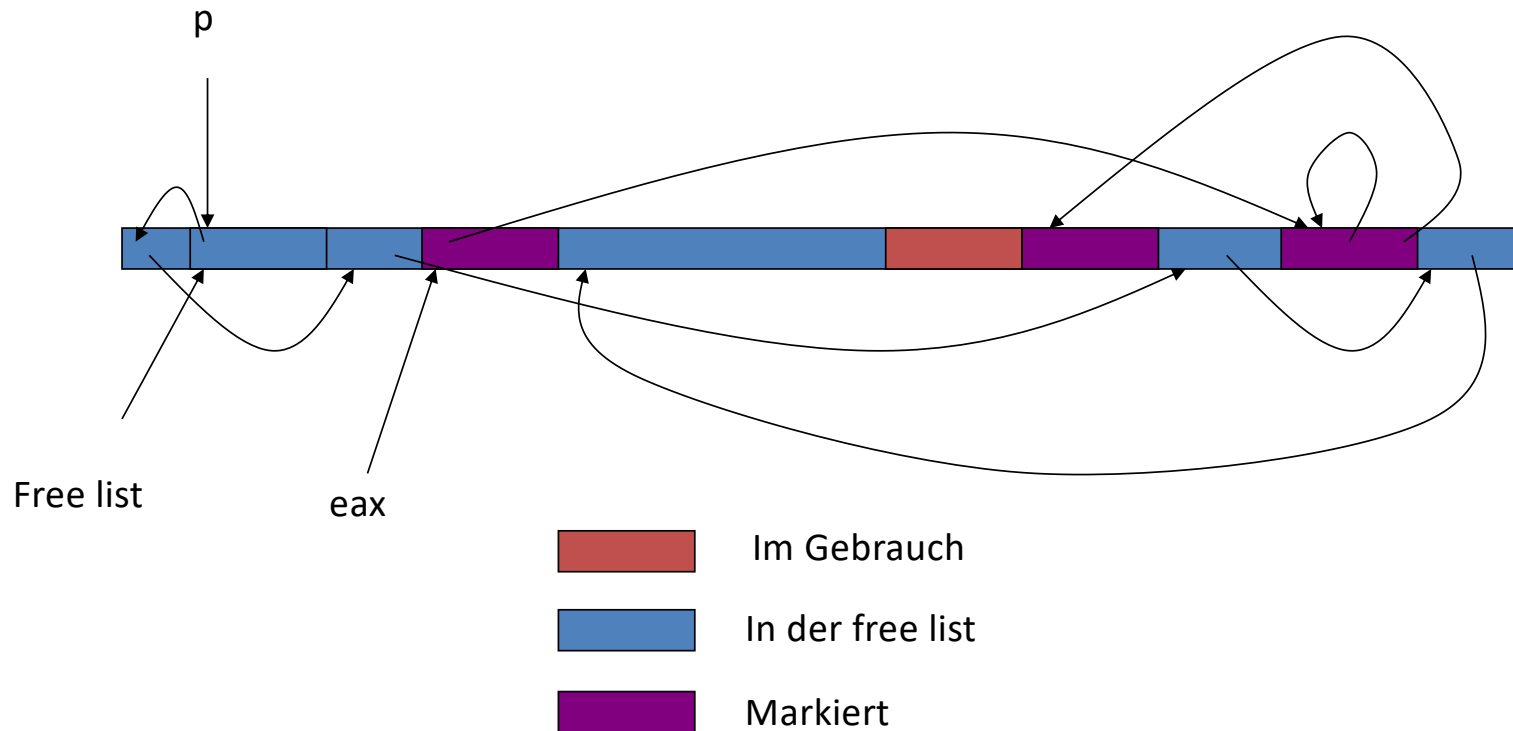
# Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



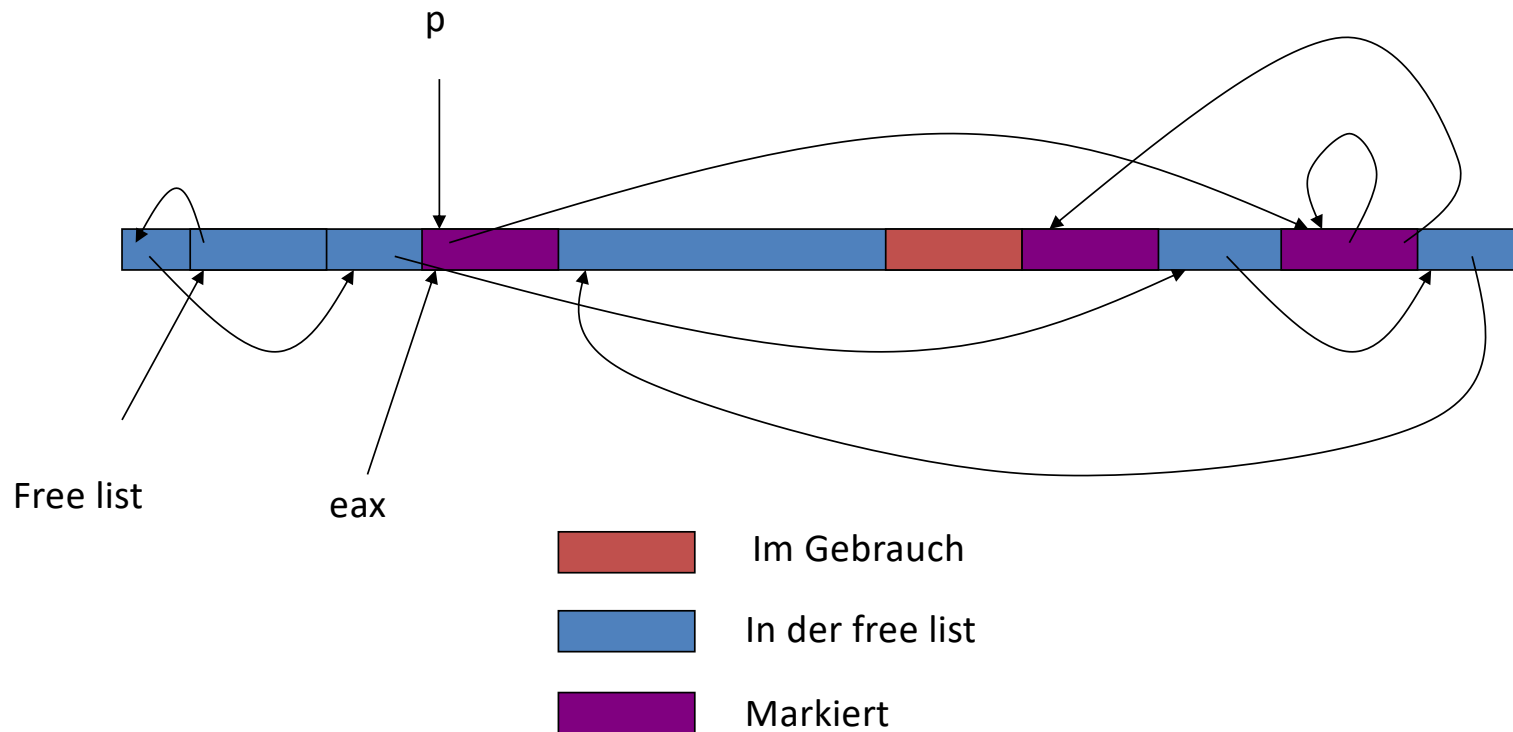
# Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



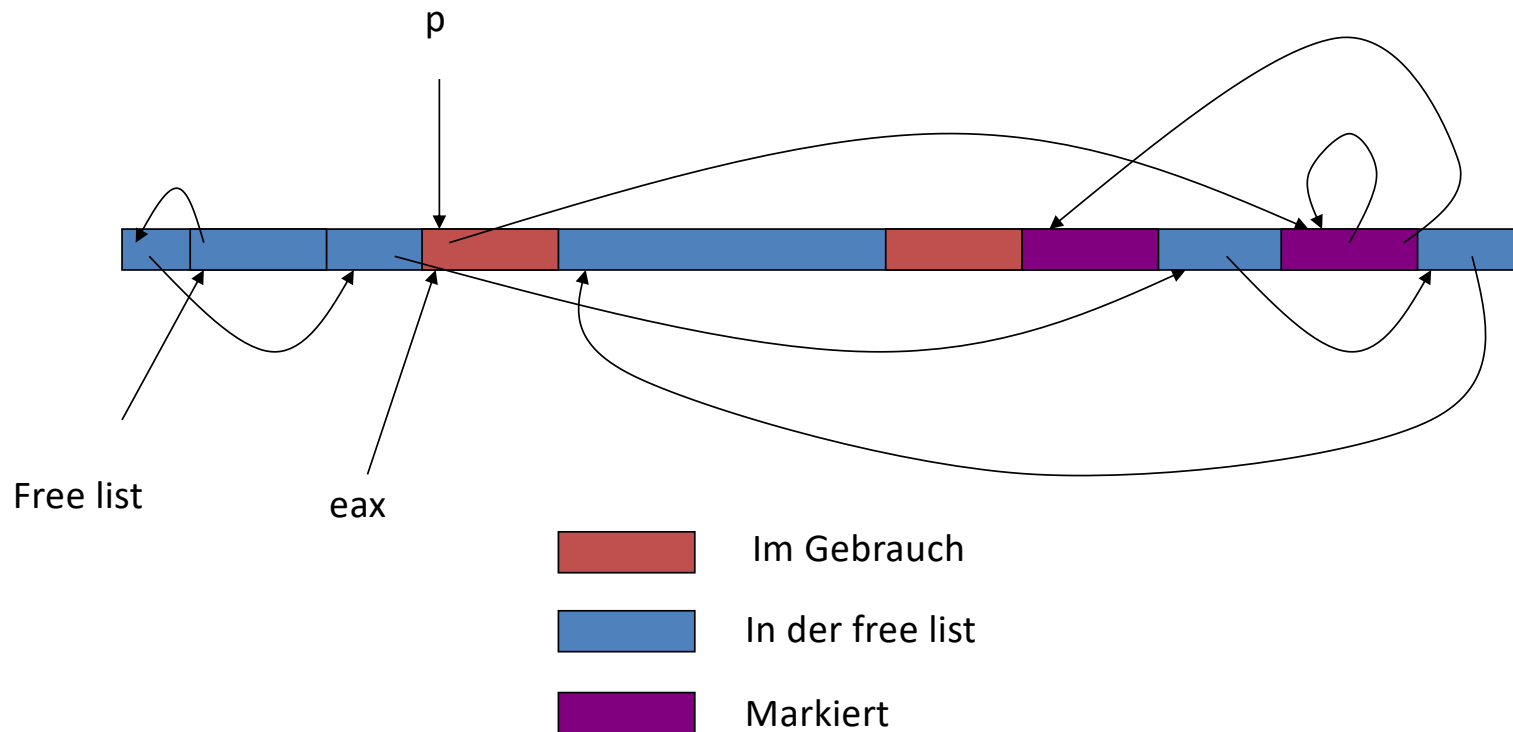
# Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



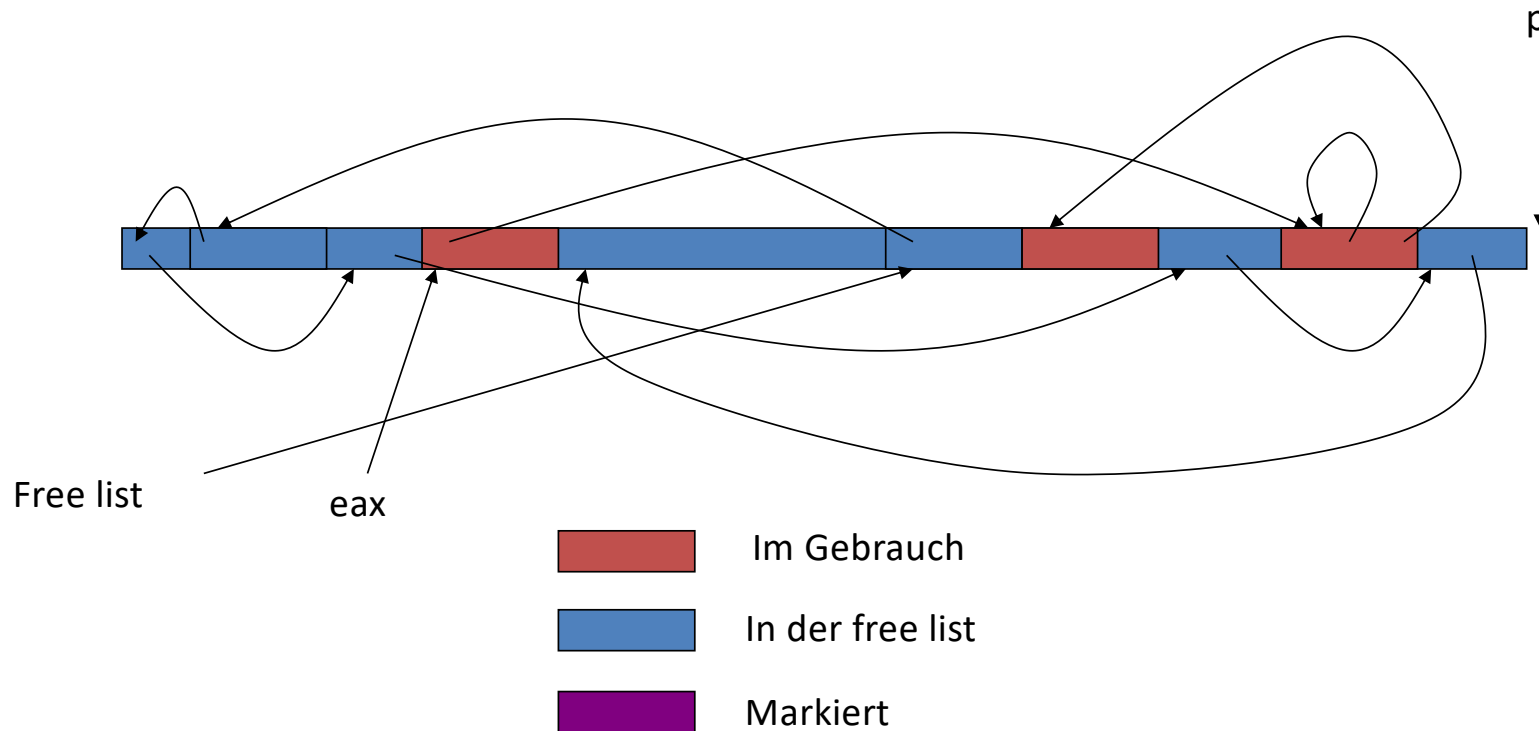
# Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



## Mark & Sweep: Zeitlupe

**Sweep Phase: GC fertig, wenn Ende des Heaps erreicht wird; Ausführung des Programm kann wieder aufgenommen werden**



# Bemerkungen: Mark & Sweep GC

## Vorteile

- GC wird «in situ» durchgeführt
- Kein Extra Speicherplatz nötig

## Nachteile

- Fragmentierung kann ein Problem sein
- Programm muss während des GCs gestoppt werden
- Allokation kann langsam sein: Passender Block muss in der free list gesucht werden
- Sweep-Phase muss den ganzen Heap überqueren
  - Algorithmus kann weiter optimiert werden (wird heute nicht diskutiert)

# Mark & Copy

## Idee: 2 Heaps werden verwendet

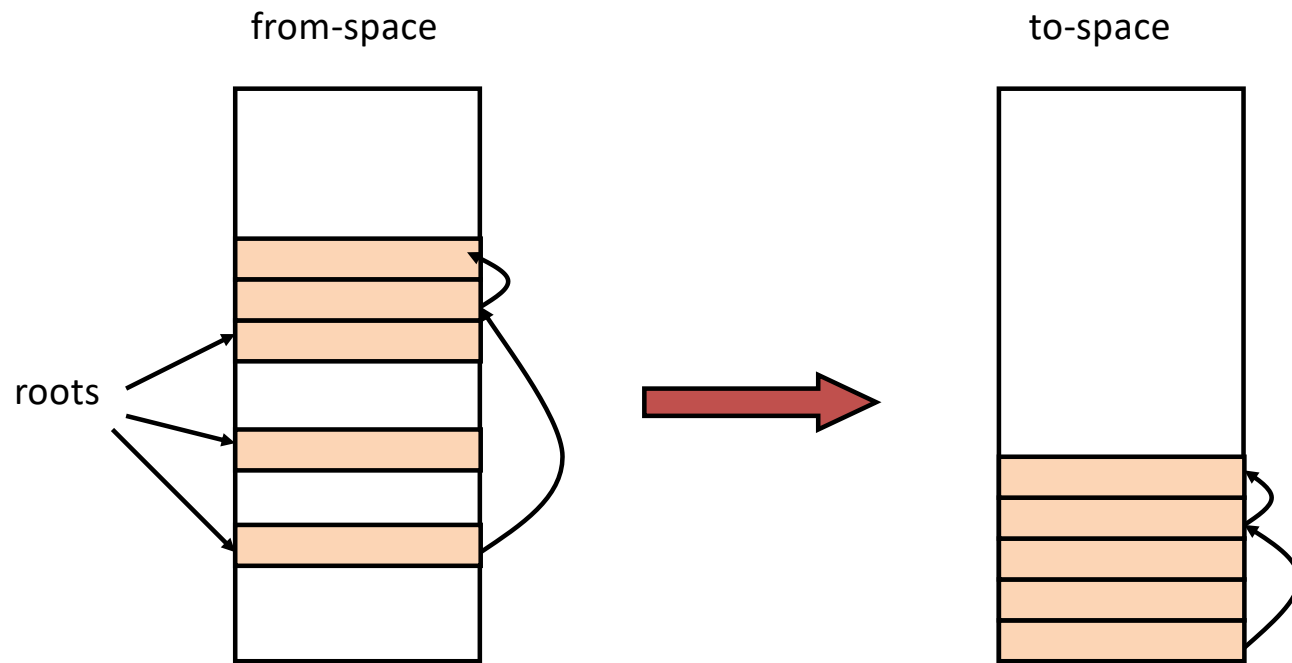
- Ein Heap (sog. from-space) wird vom Programm verwendet
- Der andere Heap (sog. to-space) nicht verwendet bis GC startet

## GC:

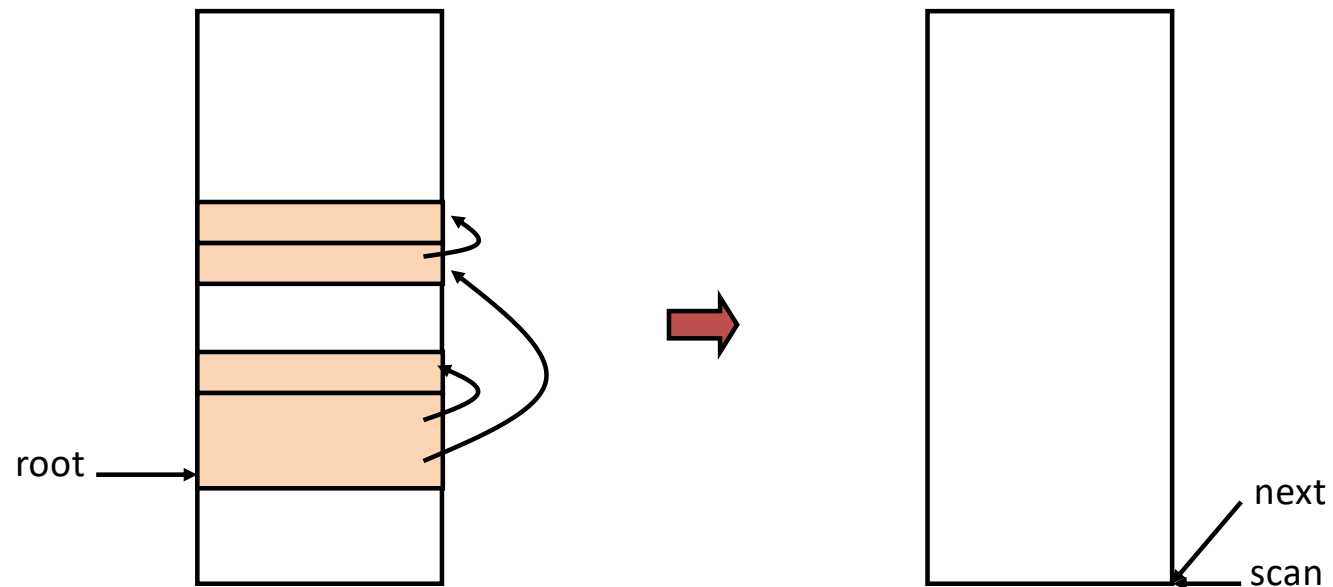
- Startet beim Root Set und traversiert den Objektgraphen
- Erreichbare Objekte werden vom from-space ins to-space kopiert
- Unerreichbare Objekte sind im from-space hinterlassen
- Die Rolle der Heaps wird gewechselt



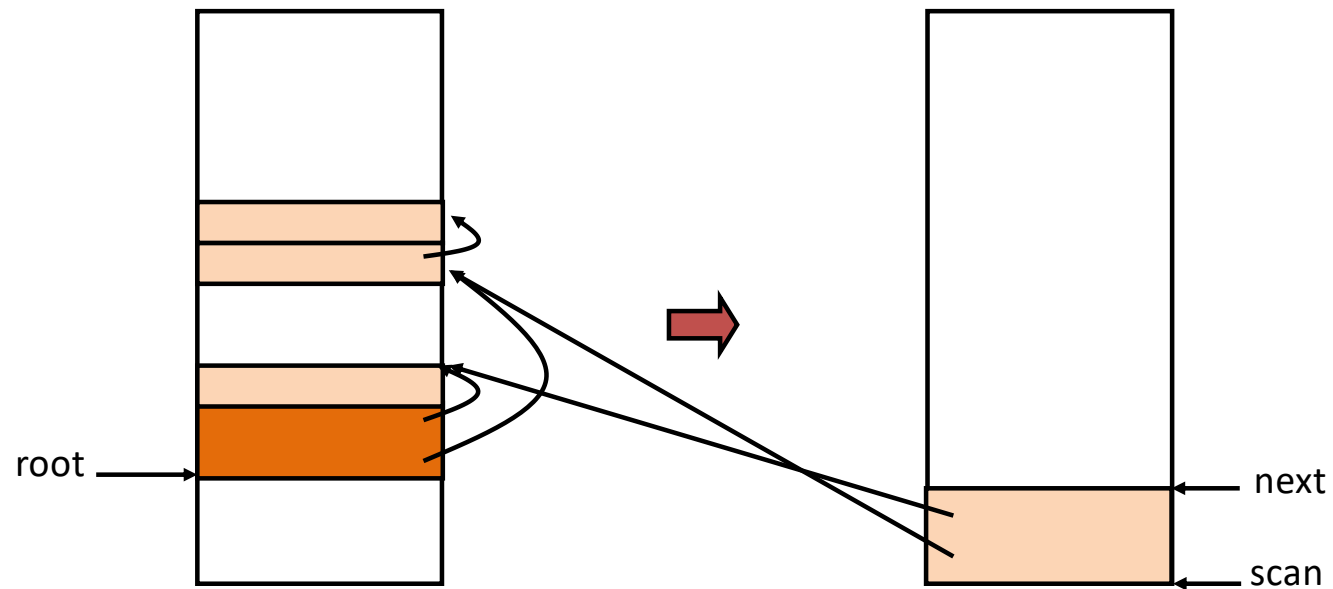
# Mark & Copy



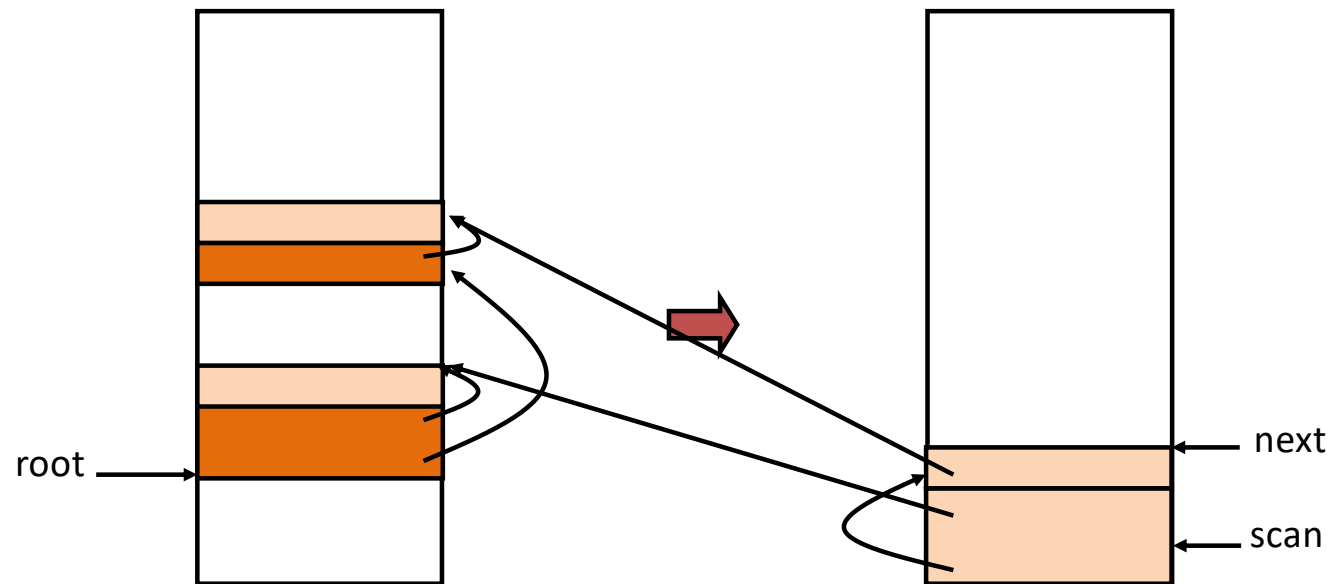
# Mark & Copy: Zeitlupe



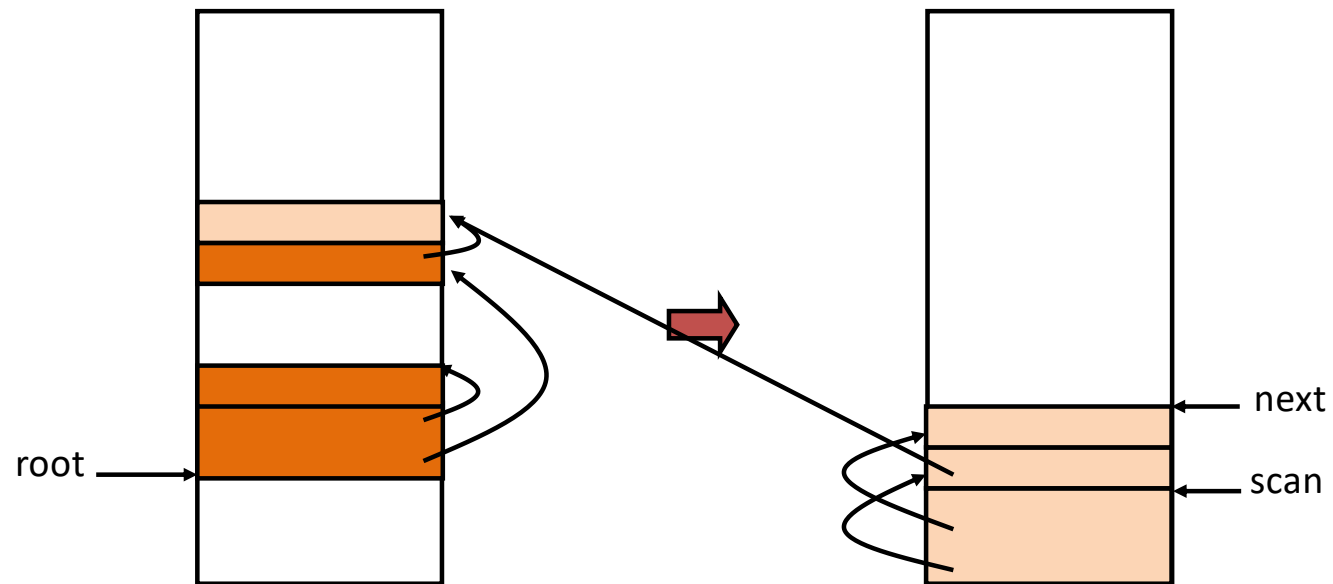
# Mark & Copy: Zeitlupe



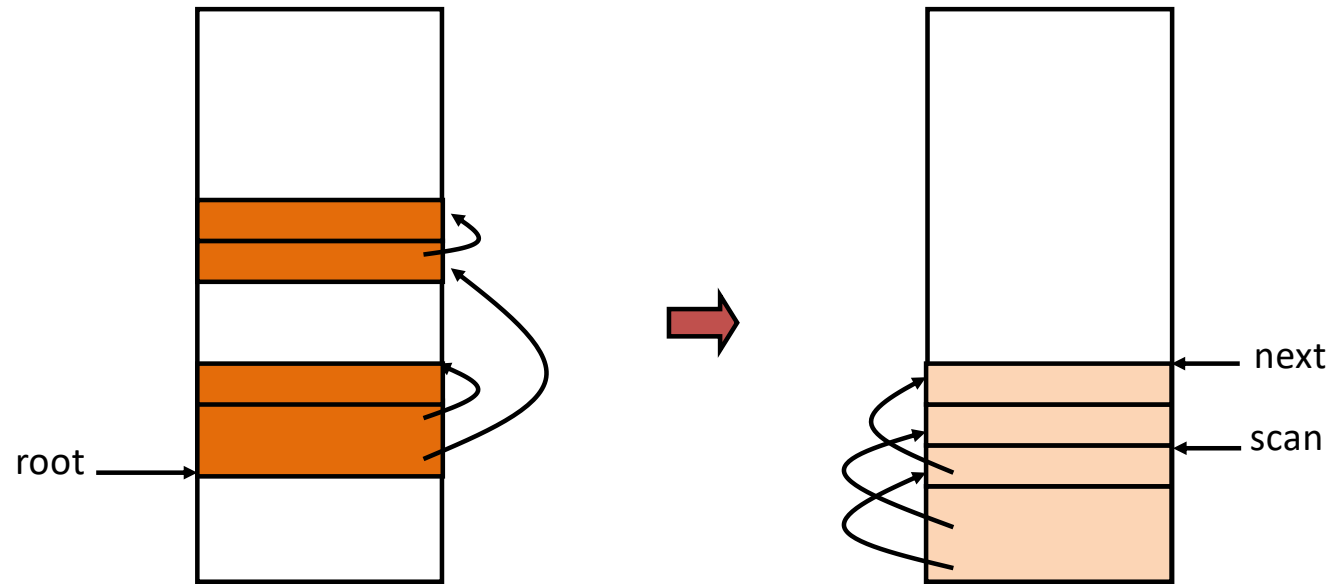
# Mark & Copy: Zeitlupe



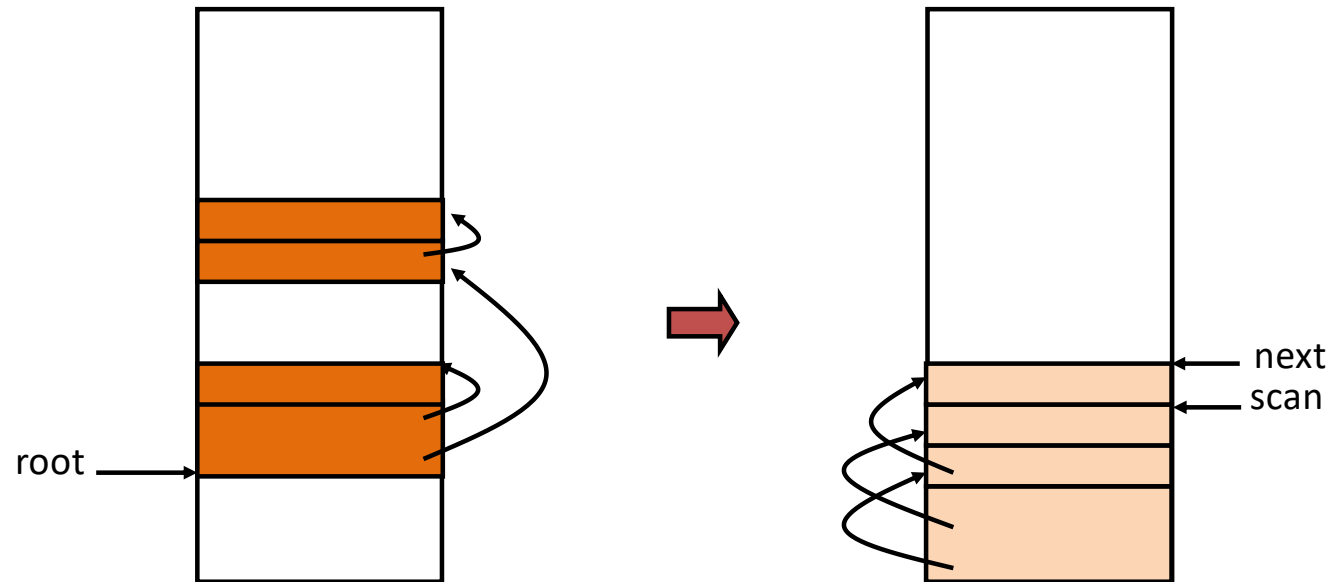
# Mark & Copy: Zeitlupe



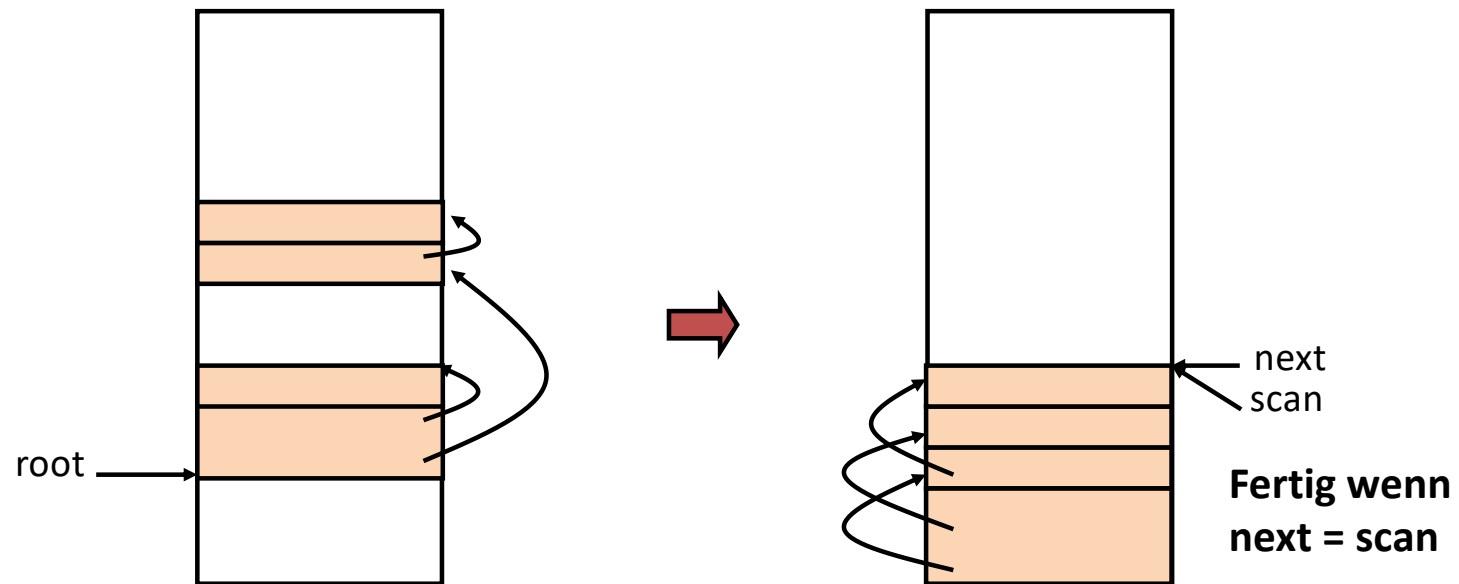
# Mark & Copy: Zeitlupe



# Mark & Copy: Zeitlupe

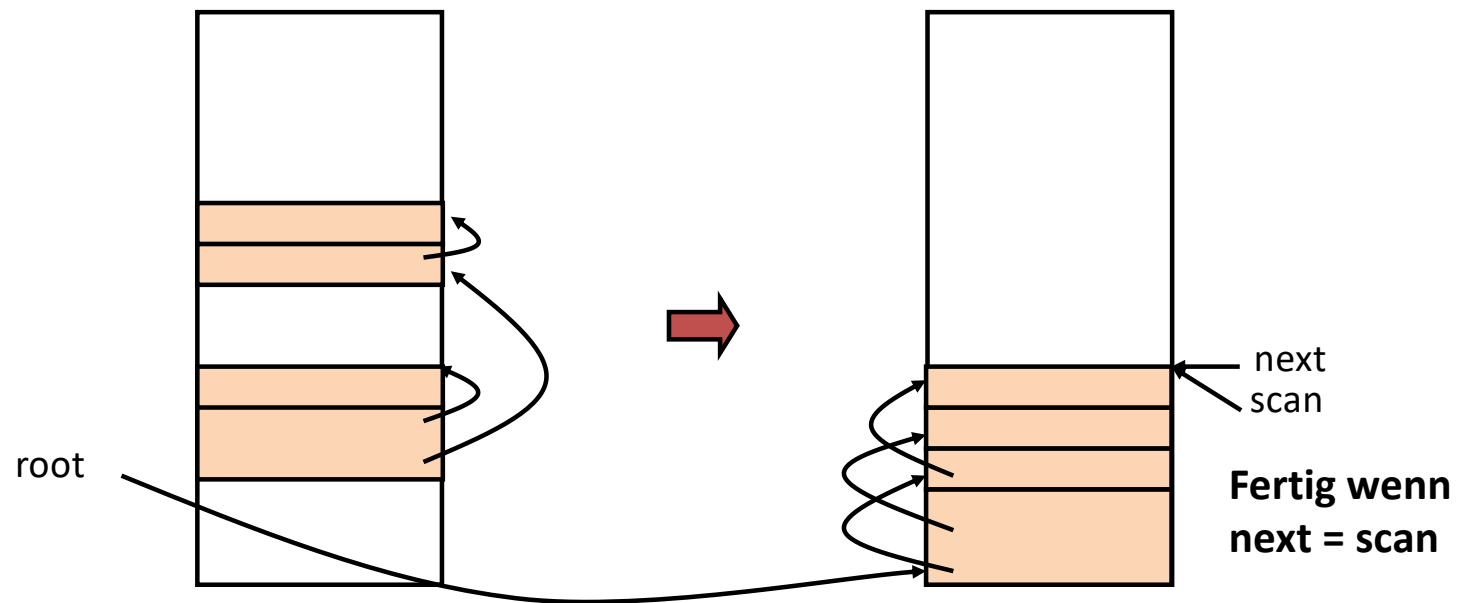


# Mark & Copy: Zeitlupe





# Mark & Copy: Zeitlupe



# Bemerkungen (Mark & Copy)

## Vorteile

- Einfach
- Eliminiert Fragmentierung
- Laufzeit proportional zur Anzahl erreichbaren Objekten
- Schnelle Allokation: Pointer wird mit Objektgröße inkrementiert

## Nachteile

- Zusätzlicher Speicherplatz nötig
- Programm muss während des GCs gestoppt werden

# Mark-Compact GC

Ähnlich wie Mark & Sweep

**Unterschied:** Nach der Mark Phase werden Objekte zum Anfang des Heaps umgelegt

## Vorteile

- Keine Fragmentierung
- In-situ: Kein zweiter Heap nötig


## Nachteile

- Zusätzliche Traversierung des Heaps nötig (3 Traversierungen insgesamt)
- Programm muss während des GCs gestoppt werden

# Diskussion:

## Vergleich Speicherverwaltungsmethoden

Relevante Kriterien?



| Algorithmus       |  |  |  |
|-------------------|--|--|--|
| Mark & Sweep GC   |  |  |  |
| Mark & Copy GC    |  |  |  |
| Mark & Compact GC |  |  |  |

# Diskussion:

## Vergleich Speicherverwaltungsmethoden

| Algorithmus       | Fragmentierung | Footprint | Stop-the-world | Geschwindigkeit Allokierung | Geschwindigkeit GC |
|-------------------|----------------|-----------|----------------|-----------------------------|--------------------|
| Mark & Sweep GC   | ✗              | ✓         | ✗              | ✗                           | ✗                  |
| Mark & Copy GC    | ✓              | ✗         | ✗              | ✓                           | ✓                  |
| Mark & Compact GC | ✓              | ✓         | ✗              | ✓                           | ✗                  |

# Diskussion:

## Vergleich Speicherverwaltungsmethoden

Bemerkung: Alle Algorithmen müssen die Applikation stoppen

| Algorithmus       | Fragmentierung | Footprint | Stop-the-world | Geschwindigkeit Allokierung | Geschwindigkeit GC |
|-------------------|----------------|-----------|----------------|-----------------------------|--------------------|
| Mark & Sweep GC   | ✗              | ✓         | ✗              | ✗                           | ✗                  |
| Mark & Copy GC    | ✓              | ✗         | ✗              | ✓                           | ✓                  |
| Mark & Compact GC | ✓              | ✓         | ✗              | ✓                           | ✗                  |

# Bemerkungen

**Kein Algorithmus hat gutes Resultat bei allen Kriterien**

- Stop-the-world ist problematisch bei allen

**Frage: Könnte man verschiedene Algorithmen kombinieren?**

- «Best-of-both-worlds» Lösung → **Generational GC**

# Garbage Collection

- Was ist GC?
- Was beeinflusst die Performanz von GC?
  - Komplexität des GC-Algorithmus
  - Implementierung des Algorithms
    - Generational GC
    - Serielle, parallele und nebenläufige GC
- **Wie beeinflusst GC die Performanz von Applikationen**
  - Wichtige Performanzmerkmale
  - Performanzerhöhung durch GC Tuning



# Generational GC

## Empirische Beobachtung 1

Wenn ein Objekt eine lange Zeit erreichbar war, wird es wahrscheinlich erreichbar bleiben

## Empirische Beobachtung 2

In vielen Programmen sterben die meisten Objekte jung

Idee 1: Arbeit wird erspart, wenn junge Objekte häufig und alte Objekte selten gescanned werden.

Idee 2: Für alte Objekte kann mehr Aufwand gewidmet werden.

# **Generational GC in der HotSpot JVM**

**Bemerkung: Bisher wurde GC unabhängig von einer konkreten Implementierung betrachtet**

**Ab jetzt diskutieren wir eine konkrete Implementierung (Java HotSpot 8)**

# Generational GC in der HotSpot JVM

Was fehlt bei der Young Generation?

## Heap Layout



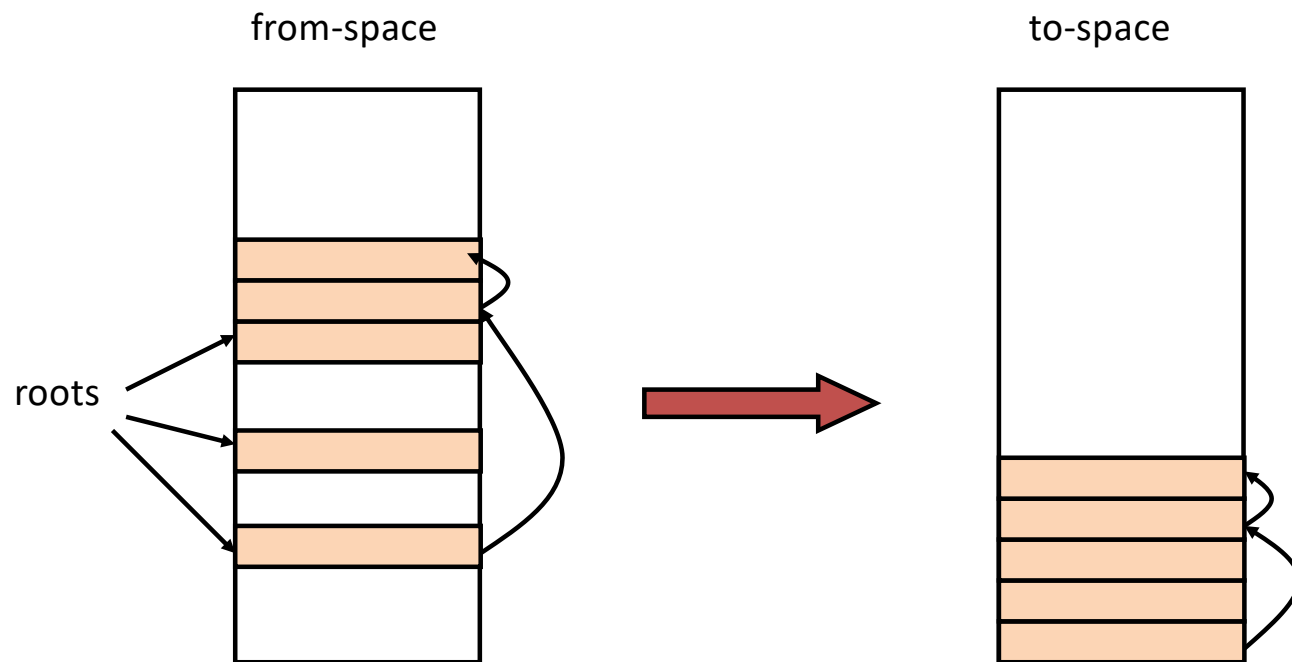
## Minor GC

- Nur young Generation wird collected
- Mark and Copy Algorithmus

## Major GC

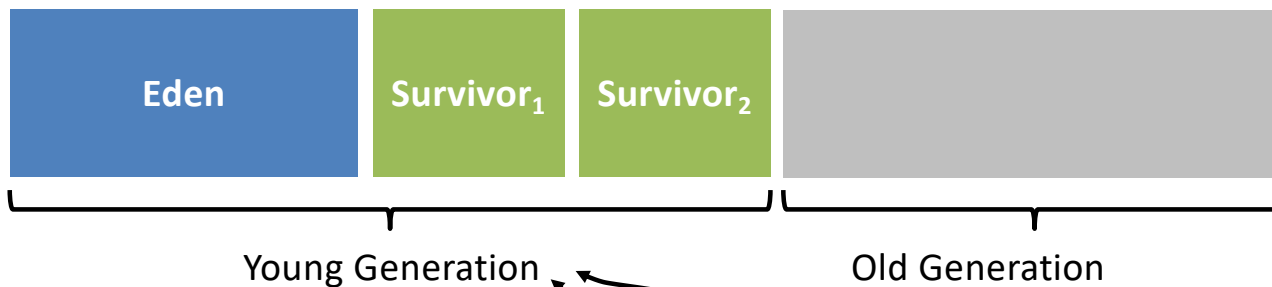
- Young und old Generation wird collected
- Mark and Compact Algorithmus (für Old Generation)

**Hint: Mark & Copy (wie vorhin gezeigt)**



# Generational GC in der HotSpot JVM

## Heap Layout



## Minor GC

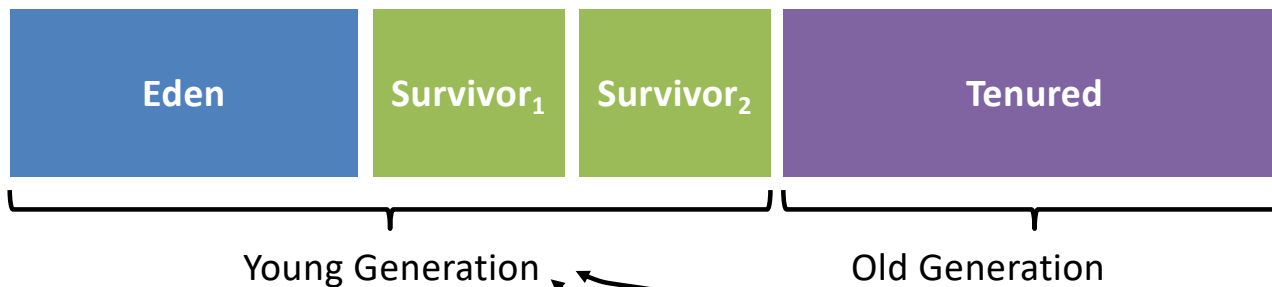
- Nur young Generation wird collected
- Mark and Copy Algorithmus

## Major GC

- Young und old Generation wird collected
- Mark and Compact Algorithmus

# Generational GC in der HotSpot JVM

## Heap Layout



## Minor GC

- Nur young Generation wird collected
- Mark and Copy Algorithmus

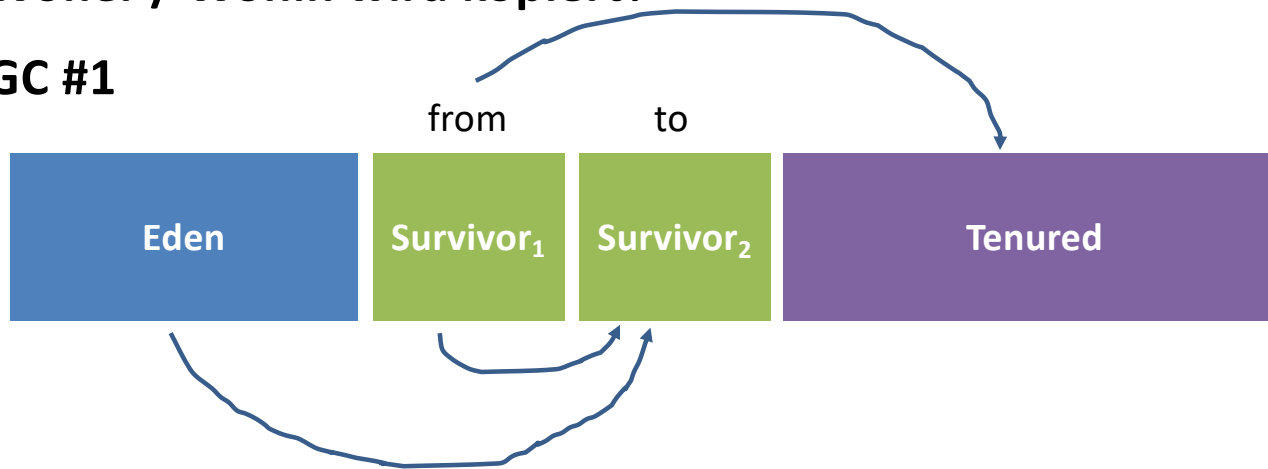
## Major GC

- Young und old Generation wird collected
- Mark and Compact Algorithmus

# Minor GC

Woher / Wohin wird kopiert?

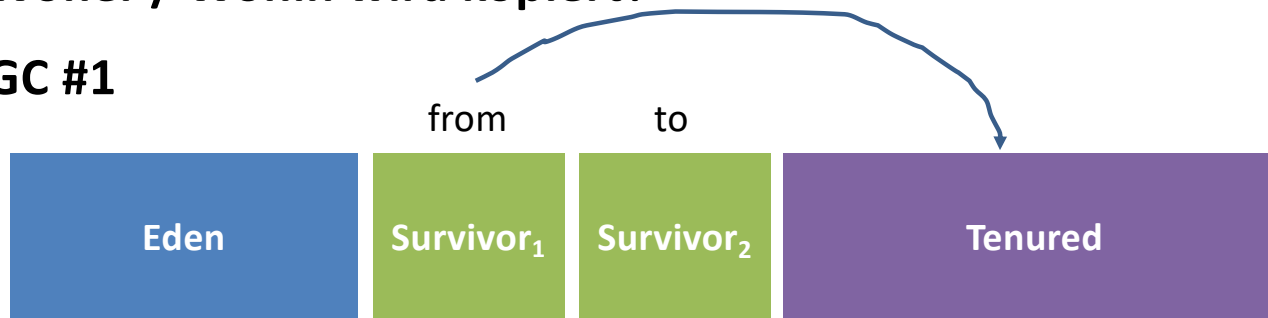
GC #1



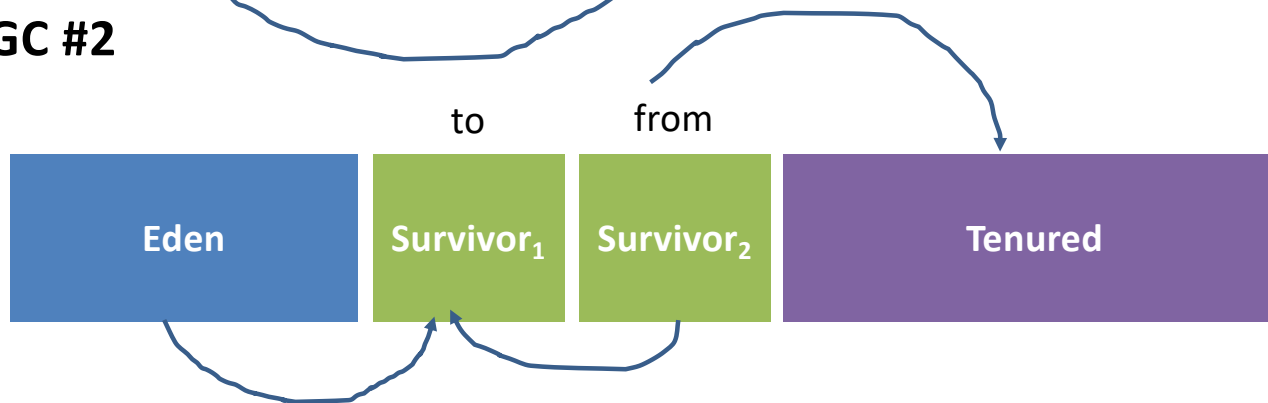
# Minor GC

Woher / Wohin wird kopiert?

GC #1



GC #2





# Verbesserung Performanz

**Was sind aus Applikationssicht wichtige Performanzmerkmale?**

- Durchsatz
- Reaktionsfähigkeit

# Ansätze

**Für Steigerung des Durchsatzes: Paralleler GC**

- GC Algorithmus parallelisiert

**Für Reduzierung der Pausenzeiten : Nebenläufiger GC**

- GC Algorithmus (oder Teile davon) läuft gleichzeitig mit dem Benutzerprogramm

# Annahmen für Beispiel

## Applikation

- Parallel mit zwei Threads
- Perfekt parallelisiert

## Hardware

- Zwei Prozessorkernen

# Serieller vs. Paralleler vs. Nebenläufiger GC

**Serieller GC**



**Paralleler GC**



**Nebenläufiger GC**



Zeit

# Fragen

**Bei welcher GC-Variante ist der Durchsatz der Applikation am besten?**

**Bei welcher GC-Variante sind die Pausenzeiten der Applikation am niedrigsten?**

# HotSpot GCs: Übersicht

## Serial GC

- Young Generation: Serieller Mark & Copy
- Old Generation: Serieller Mark & Compact

## Parallel GC

- Young Generation: Paralleler Mark & Copy
- Old Generation: Paralleler Mark & Compact

## Nebenläufiger GC: Concurrent Mark and Sweep (CMS)

- Young Generation: Paralleler Mark & Copy
- Old Generation: Mostly Concurrent Mark & Sweep

# GC Tuning

**Wiederholung: Was sind aus Applikationssicht wichtige Performanzmerkmale?**

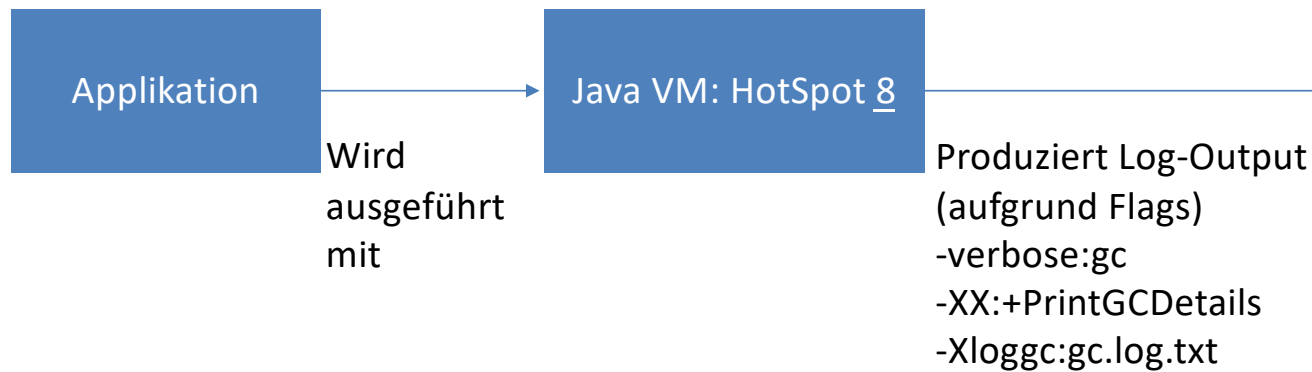
- Durchsatz
- Reaktionsfähigkeit

**Auch wenn der für das Ziel entsprechende GC-Implementierung eingeschaltet wurde, ist die Performanz manchmal nicht gut genug**

- Problem kann manchmal durch einen manuellen Eingriff gelöst werden = **GC Tuning**

# GC Tuning

## Setup





# Beispieloutput

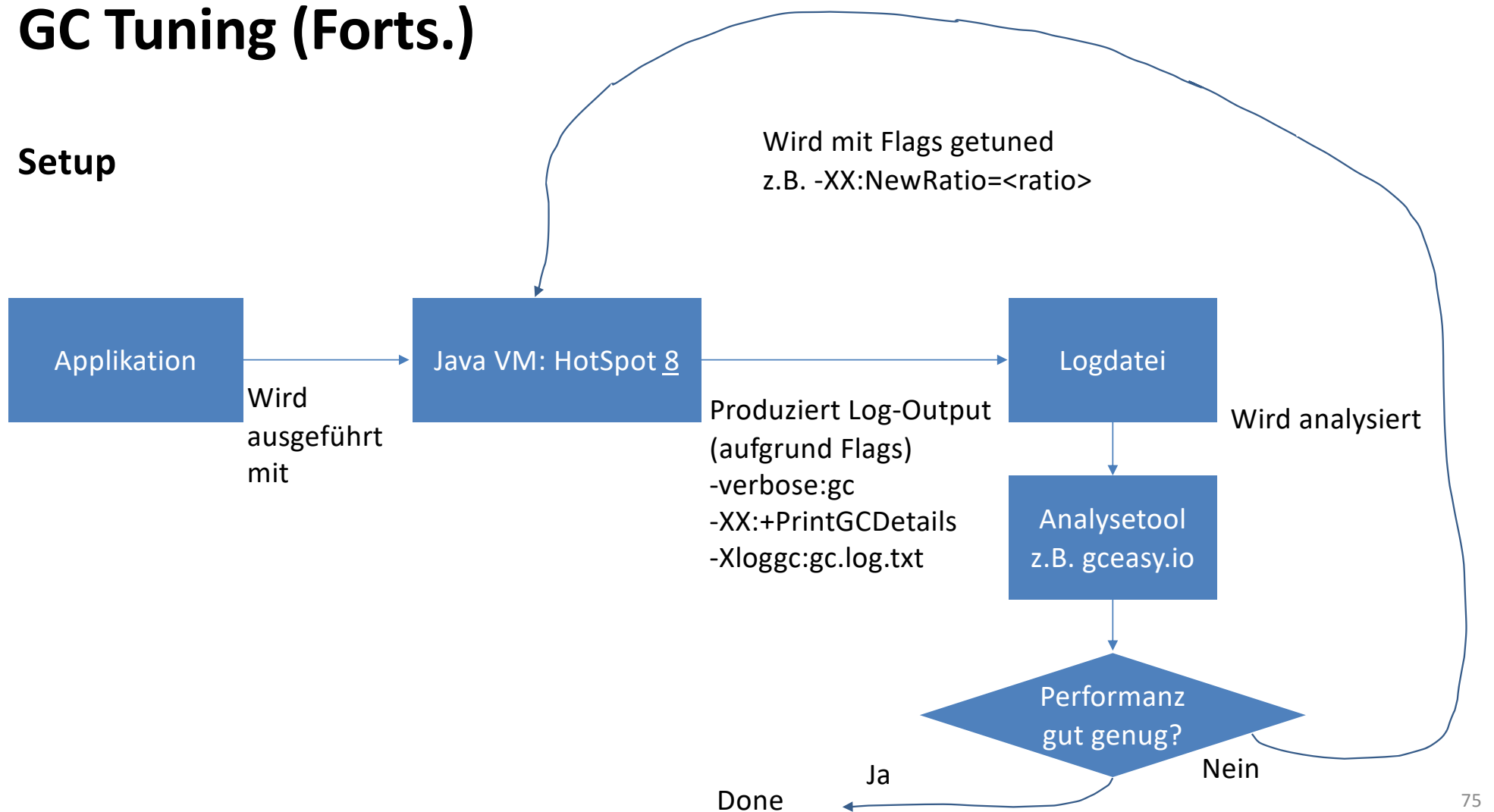
```
Java HotSpot(TM) 64-Bit Server VM (25.141-b15) for bsd-amd64 JRE (1.8.0_141-b15), built on Jul 12 2017 04:35:23 by "java_re" with gcc 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)
Memory: 4k page, physical 16777216k(1248276k free)
```

```
/proc/meminfo:
```

```
CommandLine flags: -XX:InitialHeapSize=33554432 -XX:MaxHeapSize=33554432 -XX:NewRatio=1 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:SurvivorRatio=1 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
0.120: [GC (Allocation Failure) [PSYoungGen: 6143K->5106K(11264K)] 6143K->5794K(27648K), 0.0030349 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.167: [GC (Allocation Failure) [PSYoungGen: 11239K->3729K(11264K)] 11927K->4425K(27648K), 0.0032820 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
0.171: [GC (Allocation Failure) [PSYoungGen: 9865K->1648K(11264K)] 10561K->2344K(27648K), 0.0006056 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
0.172: [GC (Allocation Failure) [PSYoungGen: 7786K->5090K(11264K)] 8482K->8475K(27648K), 0.0022283 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
0.218: [GC (Allocation Failure) [PSYoungGen: 11229K->5106K(11264K)] 14615K->9020K(27648K), 0.0012928 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.267: [GC (Allocation Failure) [PSYoungGen: 11246K->3553K(11264K)] 15160K->7467K(27648K), 0.0011604 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.270: [GC (Allocation Failure) [PSYoungGen: 9693K->1152K(11264K)] 13607K->5346K(27648K), 0.0009220 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
0.272: [GC (Allocation Failure) [PSYoungGen: 7293K->5090K(11264K)] 11487K->11469K(27648K), 0.0019212 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.316: [GC (Allocation Failure) [PSYoungGen: 11231K->5090K(11264K)] 17610K->11621K(27648K), 0.0013460 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
0.370: [GC (Allocation Failure) [PSYoungGen: 11216K->2080K(11264K)] 17747K->8628K(27648K), 0.0006940 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.372: [GC (Allocation Failure) [PSYoungGen: 8222K->5090K(11264K)] 14769K->14671K(27648K), 0.0026775 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.375: [GC (Allocation Failure) [PSYoungGen: 11231K->5090K(11264K)] 20812K->15663K(27648K), 0.0014235 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.419: [GC (Allocation Failure) [PSYoungGen: 11231K->4001K(11264K)] 21804K->14590K(27648K), 0.0009612 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.467: [GC (Allocation Failure) [PSYoungGen: 10143K->1920K(11264K)] 20732K->12525K(27648K), 0.0006870 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.468: [GC (Allocation Failure) [PSYoungGen: 8062K->5090K(11264K)] 18667K->18592K(27648K), 0.0026388 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
0.471: [Full GC (Ergonomics) [PSYoungGen: 5090K->0K(11264K)] [ParOldGen: 13502K->8264K(16384K)] 18592K->8264K(27648K), [Metaspace: 2682K->2682K(1056768K)], 0.0040828 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

# GC Tuning (Forts.)

## Setup



# Übung

## 1. Testprogramm kompilieren: `javac Test.java`

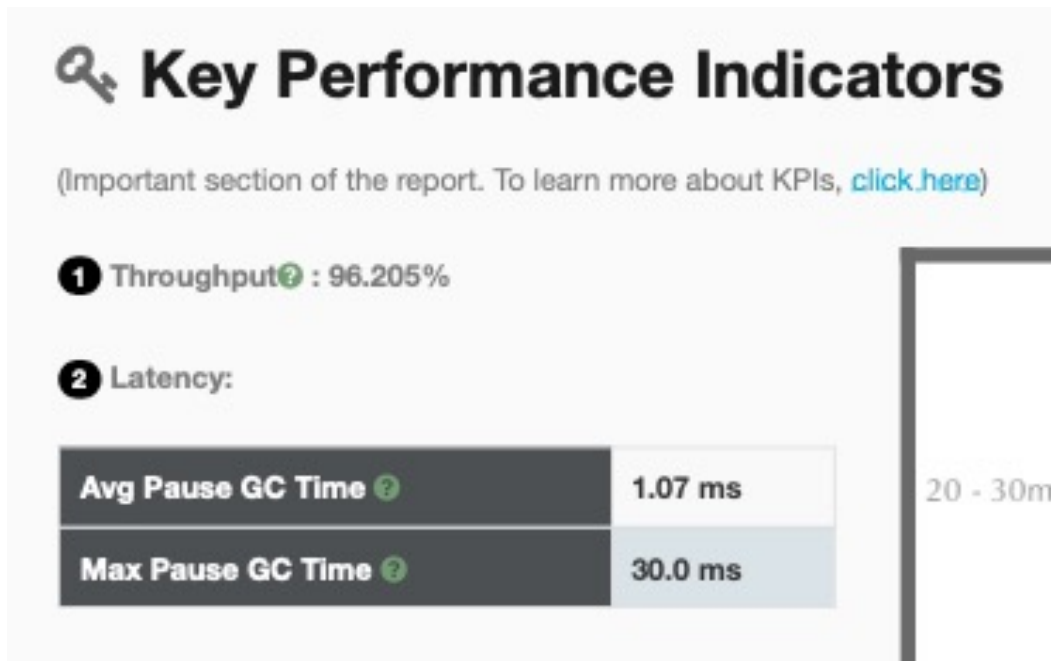
## 2. Testprogramm mit reduzierter Heapgrösse ausführen

- `java -Xmx64M -verbose:gc -XX:+PrintGCDetails -Xloggc:gc-64m-default.log.txt Test`
- Frage 1: Was ist der Durchsatz und die maximale Pausenlänge bei dieser Ausführung?

## 3. Testprogramm mit Custom-Flags ausführen

- `java -Xmx64M -verbose:gc -XX:+PrintGCDetails -XX:NewRatio=1 -XX:SurvivorRatio=8 -Xloggc:gc-64m-NR1-SR8.log.txt Test`
- Frage 2: Was bewirken die Flags `NewRatio` und `SurvivorRatio`? Hint: `JSGCT.pdf`
- Frage 3: Was ist der Durchsatz und die maximale Pausenlänge bei dieser Ausführung?
- Frage 4: Ist es durch die Verwendung der Flags `-XX:GCTimeRatio` und `-XX:MaxGCPauseMillis` einen höheren Durchsatz bzw. niedrigere maximale Pausenzeit als bei Frage 2 und 3 zu erreichen?

## Frage 1: Was ist der Durchsatz und die maximale Pausenlänge bei der Ausführung -Xmx64M?



## Frage 2: Was bewirken die Flags NewRatio und SurvivorRatio?

**-Xmx64M**




**-Xmx64M -XX:NewRatio=1 -XX:SurvivorRatio=8**



## Frage 3: Was ist der Durchsatz und die maximale Pausenlänge der Ausführung?

### Key Performance Indicators

(Important section of the report. To learn more about KPIs, [click here](#))

**1** Throughput  : 99.825%

**2** Latency:

|   |          |
|---|----------|
| Avg Pause GC Time  | 0.151 ms |
| Max Pause GC Time  | 10.0 ms  |

9 - 10ms

# Diskussion

**Ist GC Tuning einfach oder eher schwierig? (Denken Sie auch an reelle Applikationen)  
Wieso?**

# Kommandozeilen – Beispiele

```
java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g  
-XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0  
-XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSScavengeParallelRemarkEnabled  
-XX:+UseCMSInitiatingOccupancyOnly -XX:ParallelGCThreads=12  
-XX:LargePageSizeInBytes=256m ...
```

```
java -Xms8g -Xmx8g -Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M  
-XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2 -XX:-UseAdaptiveSizePolicy  
-XX:+UseConcMarkSweepGC -XX:+CMSConcurrentMTEnabled  
-XX:+CMSScavengeParallelRemarkEnabled -XX:+CMSScavengeParallelSurvivorRemarkEnabled  
-XX:CMSMaxAbortablePrecleanTime=10000 -XX:+UseCMSInitiatingOccupancyOnly  
-XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC -Xnoclassgc ...
```



# Flags

**java -XX:+PrintFlagsFinal | grep "GC\|CMS\|G1"**  
**169**

```
uintx AdaptiveSizeMajorGCDecayTimeScale      = 10           {product}
uintx AutoGCSelectPauseMillis                 = 5000          {product}
bool   BindGCTaskThreadsToCPUs                = false         {product}
bool   CMSAbortSemantics                      = false         {product}
uintx  CMSAbortablePrecleanMinWorkPerIteration = 100           {product}
intx   CMSAbortablePrecleanWaitMillis         = 100          {manageable}
uintx  CMSBitMapYieldQuantum                  = 10485760      {product}
uintx  CMSBootstrapOccupancy                  = 50            {product}
...
```

# Frage

## Können wir Applikationseigenschaften festhalten um GC zu lenken?

- Rate der Allokierungen (engl. allocation rate)
- Rate der Mutationen (engl. mutation rate)
- ...

## Ohne messbare Applikationseigenschaften: Experimentieren

If the heap grows to its maximum size and the throughput goal isn't being met, then the maximum heap size is too small for the throughput goal. Set the maximum heap size to a value that's close to the total physical memory on the platform, but doesn't cause swapping of the application. Execute the application again. If the throughput goal still isn't met, then the goal for the application time is too high for the available memory on the platform.

JSCT.pdf (HotSpot Virtual Machine Garbage Collection Tuning Guide)

# Praxis

**Manchmal muss man mit den zur Verfügung stehenden Werkzeugen was machen...**

## **Empfehlung(en) für Tuning**

- Sinnvolle und systematische Experimente
- Genug Zeit einplanen
- ...

## **Limiten der GC-Algorithmen gelten**

- Stop-the-world kann bei allen in GC-Implementierungen in der Oracle JVM vorkommen

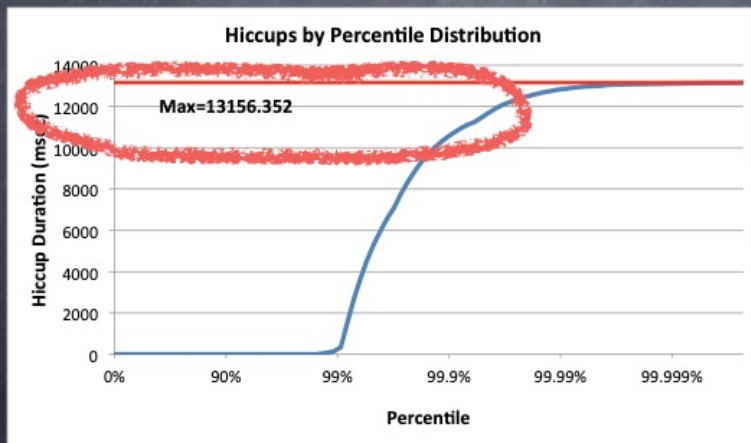
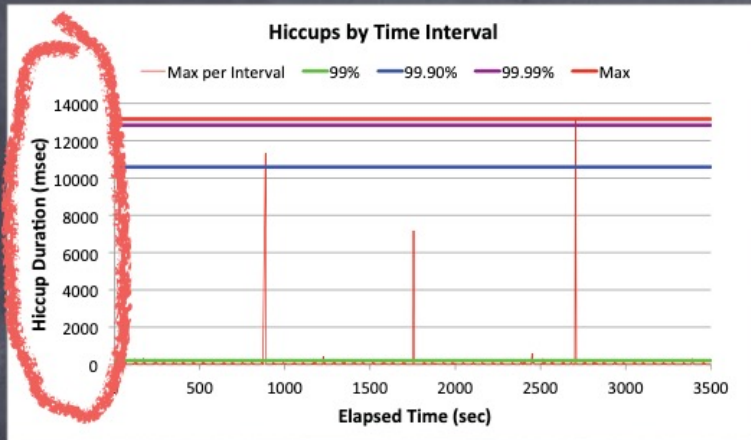
## **Neue GC-Implementierungen mit niedrigeren Pausenzeiten**

- Azul Zing VM C4 – Continuously Concurrent Compacting Collector
- Oracle JVM Shenandoah GC – ultra-low pause time garbage collector
- Oracle JVM ZGC – scalable low latency garbage collector

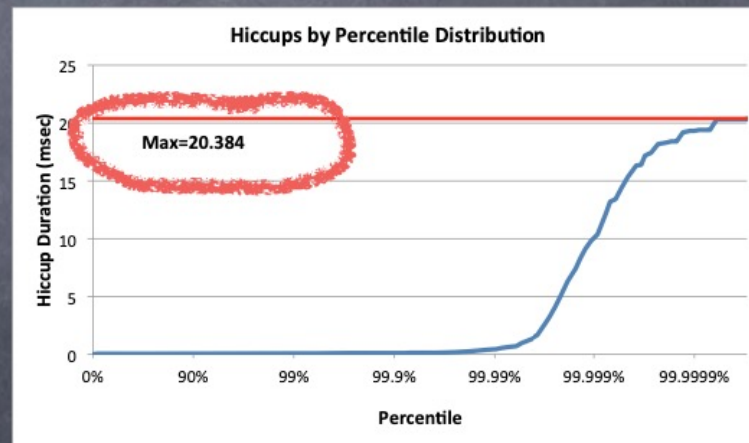
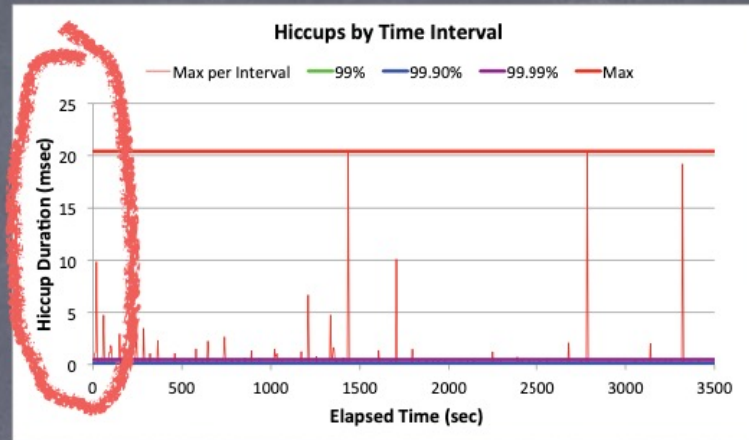
# **Azul C4**

**Die folgende Folie ist von Gil Tene (Azul) übernommen worden**

## Oracle HotSpot CMS, 1GB in an 8GB heap



## Zing 5, 1GB in an 8GB heap



# Oracle ZGC

## Empfohlene Bibliographie

- Gute Zusammenfassung: <https://www.youtube.com/watch?v=88E86quLmQA> (39 Minuten)
- Details: <https://wiki.openjdk.java.net/display/zgc/Main>

# Garbage Collection

- **Was ist GC?**
- **Was beeinflusst die Performanz von GC?**
  - Komplexität des GC-Algorithmus
  - Implementierung des Algorithms
    - Generational GC
    - Serielle, parallele und nebenläufige GC
- **Wie beeinflusst GC die Performanz von Applikationen**
  - Wichtige Performanzmerkmale
  - Performanzerhöhung durch GC Tuning