TRIONA
INFORMATION UND TECHNOLOGIE

Triona Workshop

# JVM-SPEICHERMANAGEMENT

# Motivation

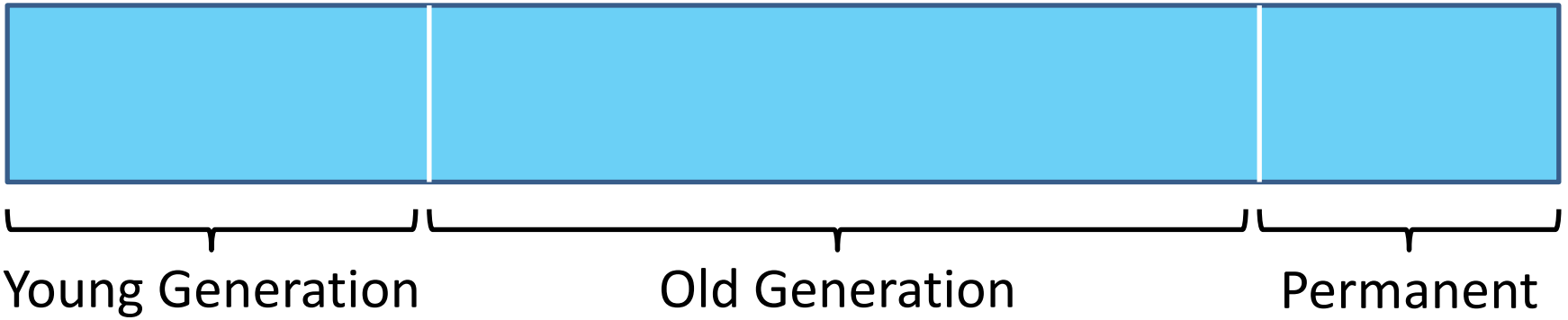
# Agenda

- Speicherbereiche der JVM
- Garbage Collection
- Monitoring und Tuning
- Beispiel
- It‘s your turn

# Speicherbereiche der JVM Historie

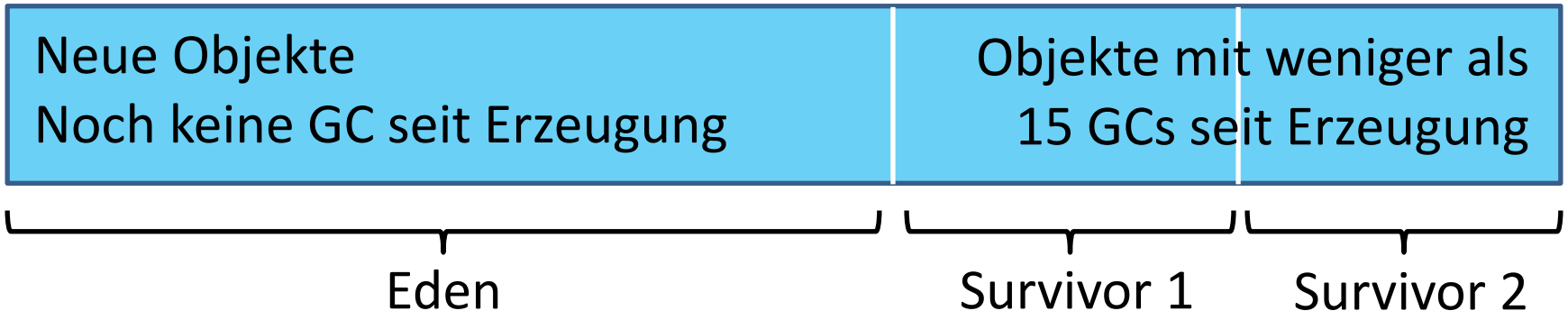
Anfangs: Continuous Heap

Seit JDK 1.3: Generational Heap



# Speicherbereiche der JVM
# Oracle HotSpot JVM

Young Generation (junge Objekte)



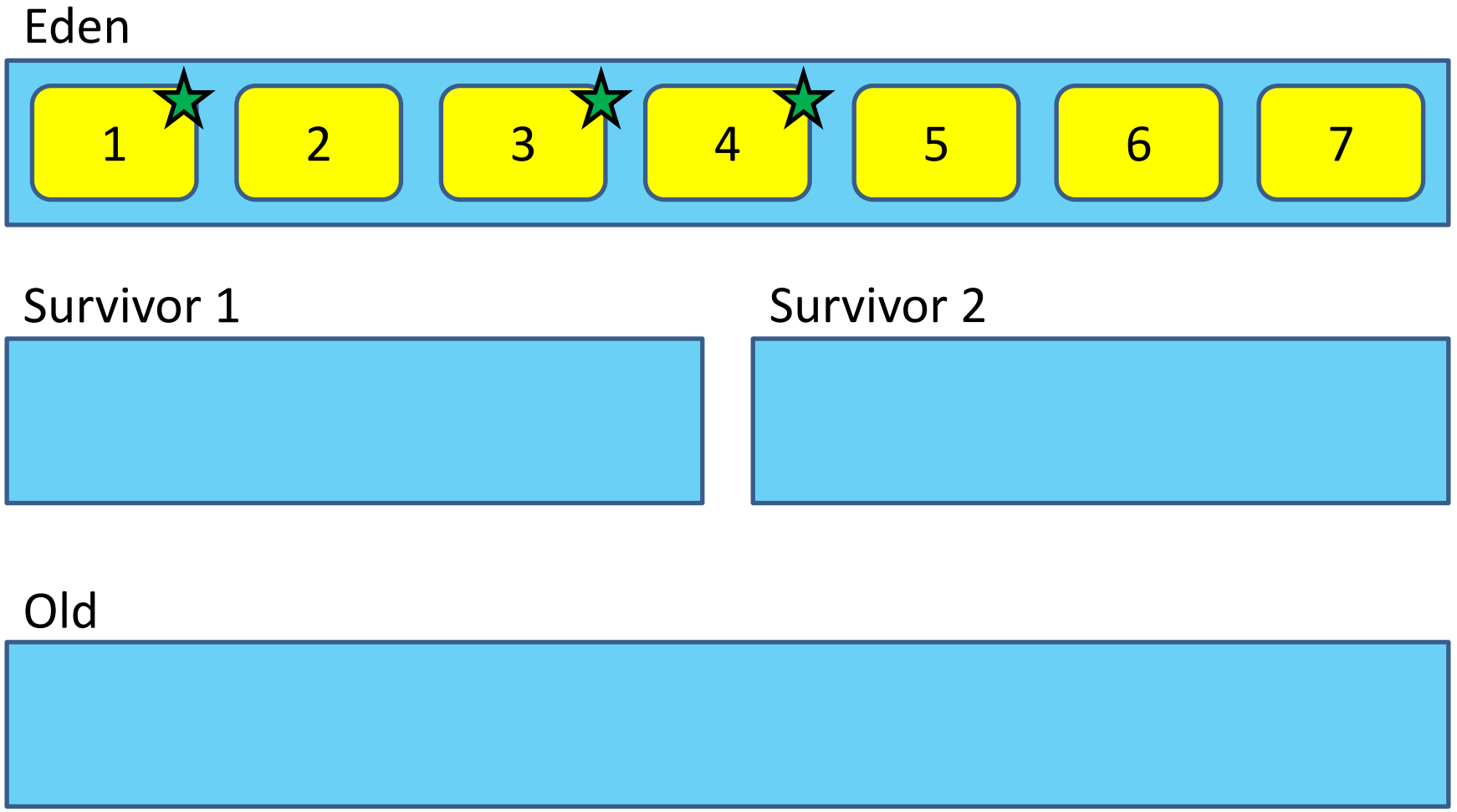

Old Generation (alte Objekte)

Mehr als 15 GCs seit Erzeugung

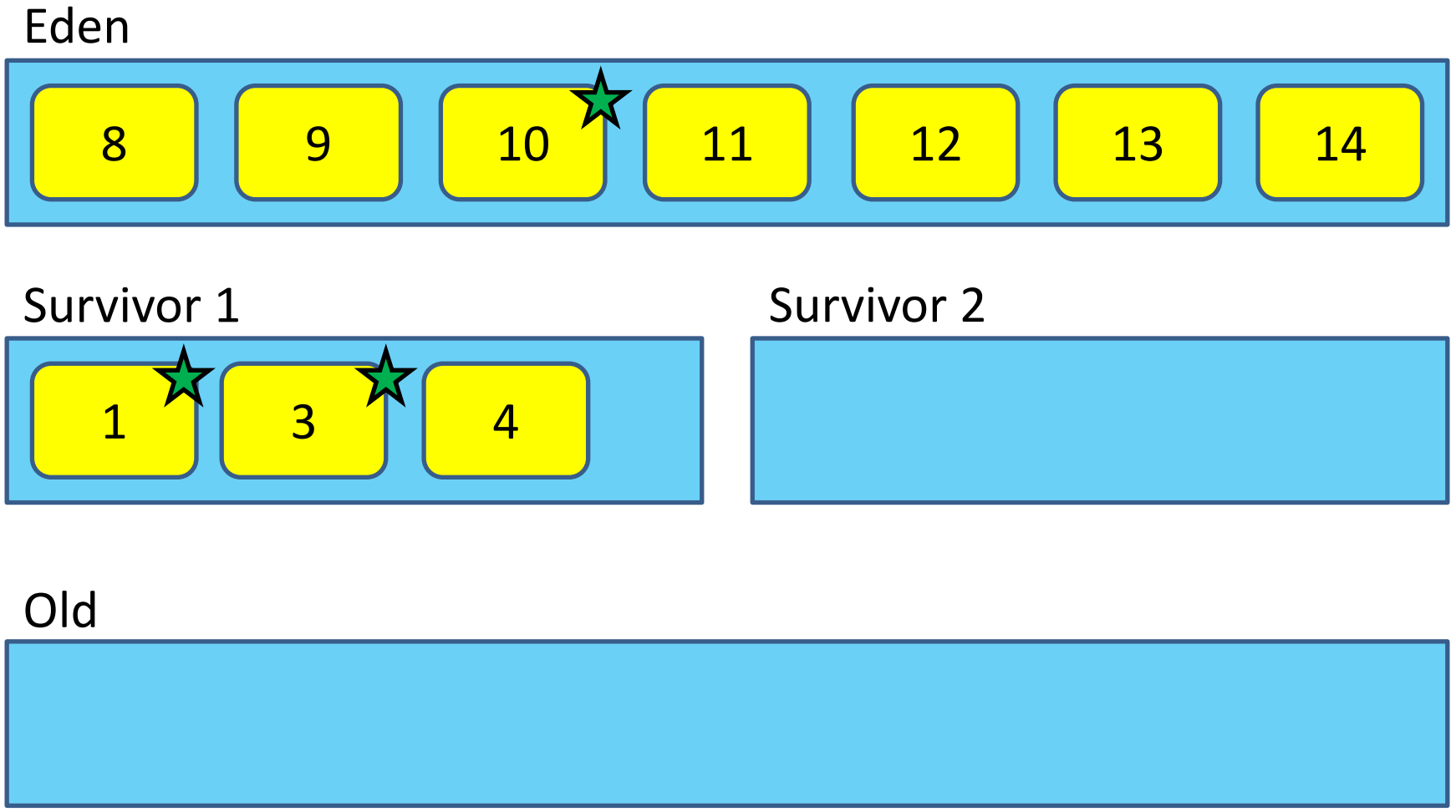
Permanent (statische Elemente)

Geladene Klassen, Infos für JIT-Compiler, …

# Speicherbereiche der JVM
## Oracle HotSpot JVM

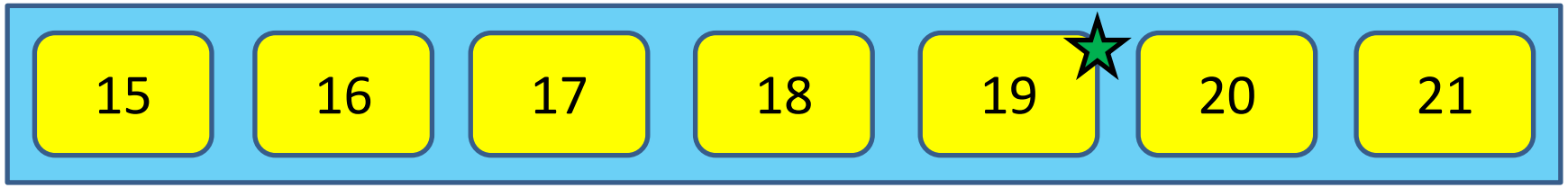


# Speicherbereiche der JVM
## Oracle HotSpot JVM



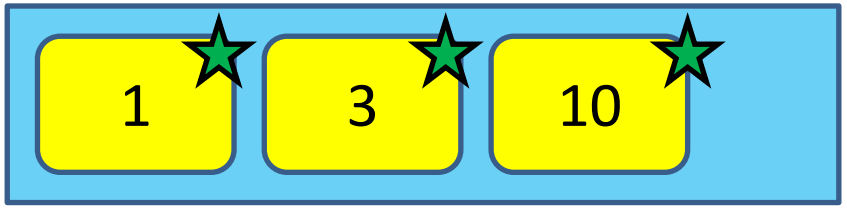
# Speicherbereiche der JVM
# Oracle HotSpot JVM

Eden




Survivor 1

Survivor 2




Old

# Speicherbereiche der JVM

## Warum Generational Heap?

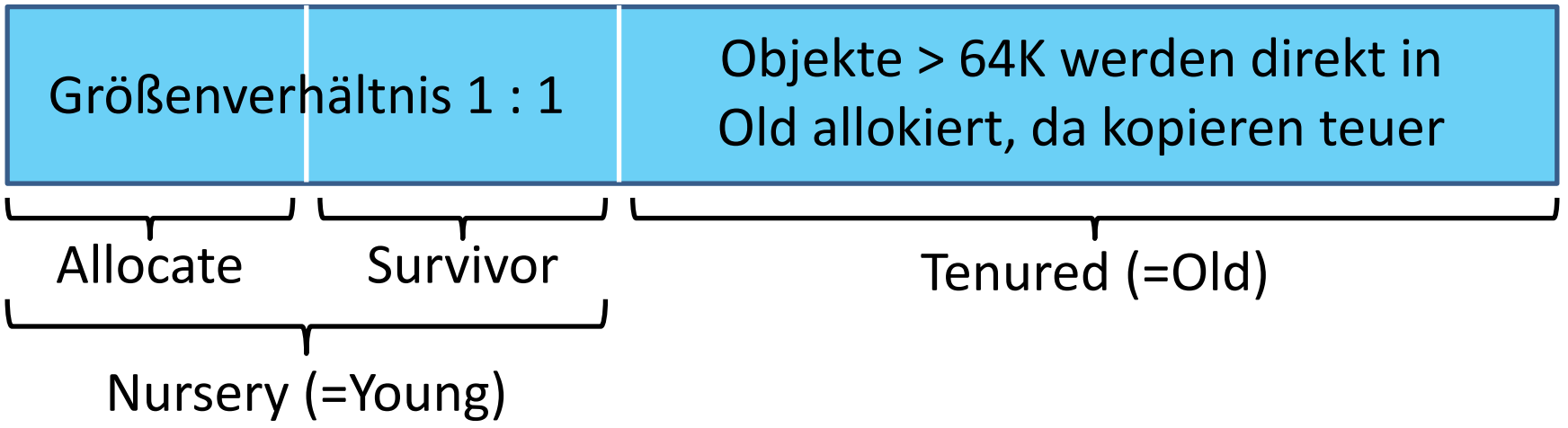
+ effizienter
+ bei GC nicht kompletten Speicher durchsuchen
+ GC teilweise parallel zur laufenden Applikation
+ unterschiedliche GC-Algorithmen einsetzbar

− Kann auch mal ineffizienter sein
− komplizierter
− Höherer Rechenaufwand

# Speicherbereiche der JVM
# IBM JVM

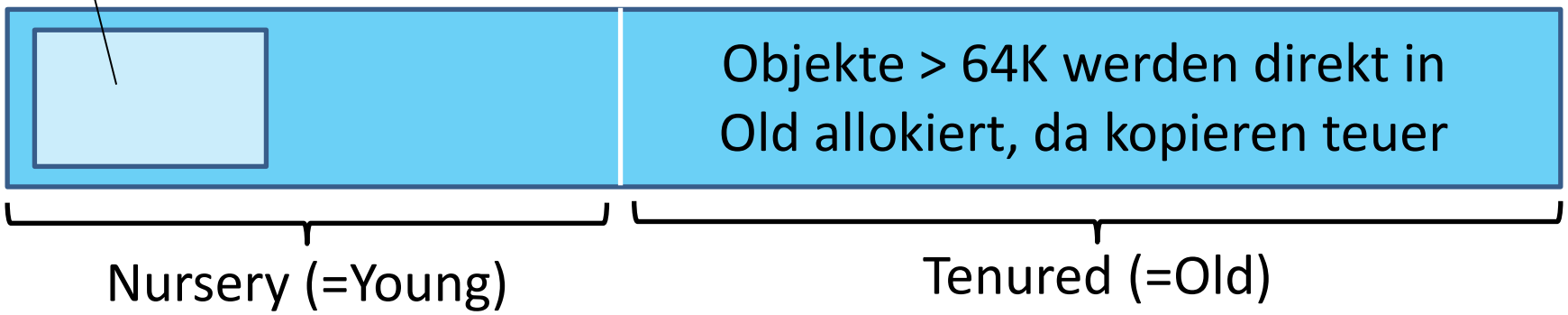
Einstellbar, ob continuous oder generational Heap

Kein Permspace. Statische Elemente im „normalen“ Heap



# Speicherbereiche der JVM
# Oracle JRockit JVM

Kein Permspace. Statische Elemente im „normalen“ Heap



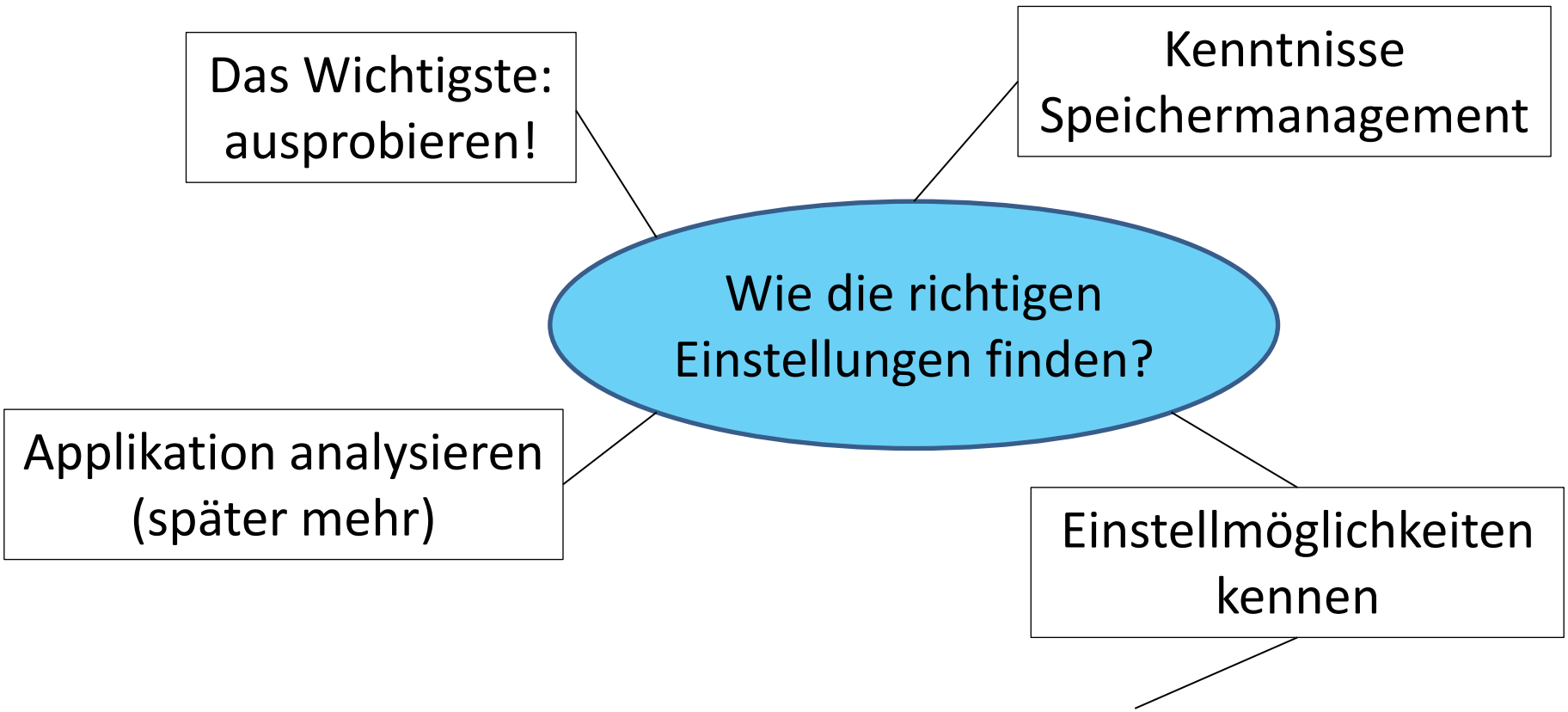

\+ Nur ein Teil des Nursery-Bereichs muss bei GC überprüft werden

\+ Jedes Objekt wird maximal 1 Mal kopiert

\- Nursery muss größer sein, da es ansonsten zu unnötigen Kopierungen in Old kommt

\- GC ist immer stop-the-world Event

# Speicherbereiche der JVM Konfiguration




http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html
http://blog.ragozin.info/2011/07/hotspot-jvm-garbage-collection-options.html
http://docs.oracle.com/cd/E18930_01/html/821-2431/abeic.html#abeij

# Agenda

- Speicherbereiche der JVM
- Garbage Collection
- Monitoring und Tuning
- Beispiel
- It‘s your turn

# Garbage Collection Überblick

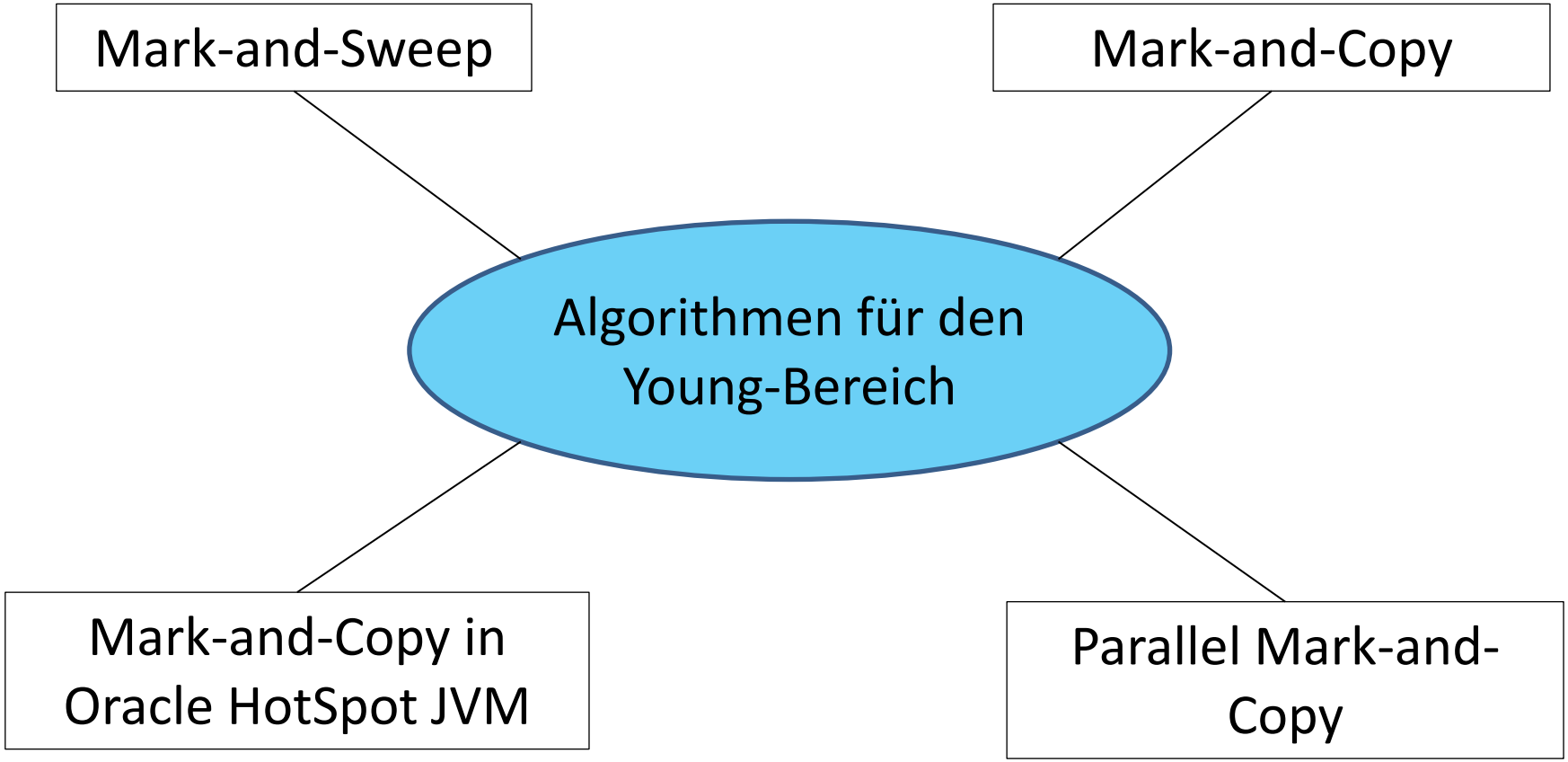
Identifiziert noch referenzierte Objekte
und gibt den übrigen Speicher frei

Probleme bei GC:
- Programm darf nicht weiterlaufen
- Je größer der Heap, desto länger dauert GC

Abb.-
GC

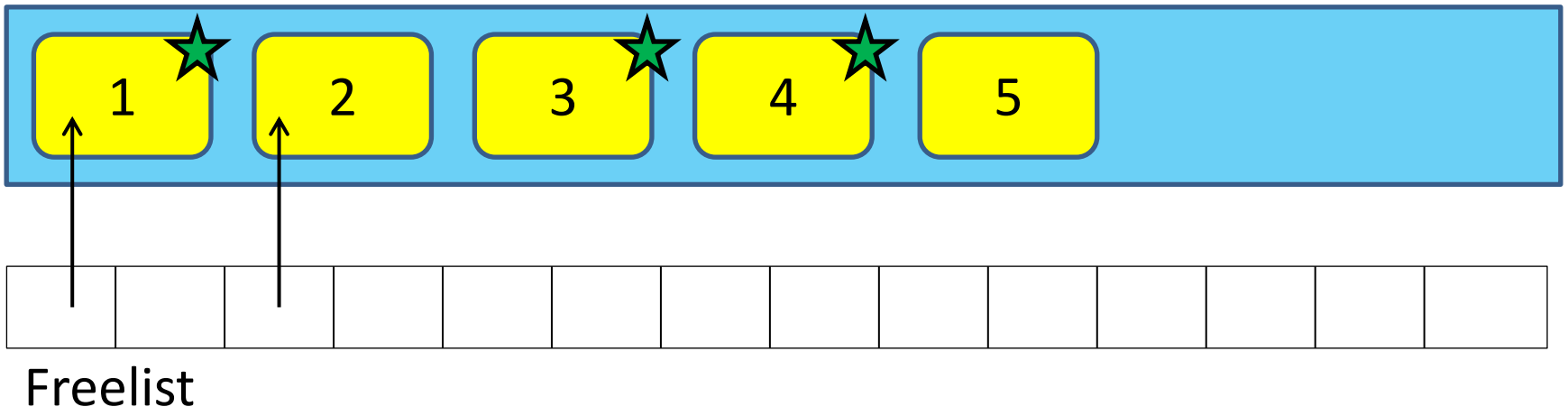
Lösungsansatz: Generational Heap
- durchsuchen des kompletten Heap teuer
- typischerweise viele kurzlebige Objekte
- deshalb häufig Young aufräumen (Minor GC)
- seltener Old aufräumen (Major GC)

# Garbage Collection
# GC-Algorithmen



# Garbage Collection
# GC-Algorithmen

## Mark-and-Sweep




Fragmentierung:
- aufwändig passenden Speicherbereich zu finden
- möglicherweise kein ausreichend großer Bereich frei

# Garbage Collection
# GC-Algorithmen

## Mark-and-Copy

(bereits bei IBM JVM erklärt mit Allocate und Survivor)

\+ Allokieren immer in kompaktem Bereich. Keine Freelist mehr notwendig

\+ Keine Fragmentierung möglich

\- Kopieren kostet Zeit

\- Höherer Speicherverbrauch, da From- und ToSpace immer gleich groß sein müssen, To aber nie voll werden wird

# Garbage Collection
# GC-Algorithmen

## Mark-and-Copy in Oracle HotSpot JVM
(bereits bei Erklärung Generational Heap demonstriert)

\+ Geringerer Speicherverbrauch durch Survivor, die bei Bedarf vergrößert bzw. aufgeräumt werden

\- Häufiges Kopieren ist teuer

\- Stop-the-world Algorithmus

Hat aber noch einen Haken:

Es könnten Referenzen von Old nach Young existieren!

# Fall: Promotion

Objekt in Young welches Referenz auf Objekt in Young enthält wird in Old kopiert. Referenziertes Objekt aber nicht.

Trivial: Werden vom GC gemerkt und bei nächster GC berücksichtigt

# Garbage Collection
# GC-Algorithmen

## Fall: Zuweisung

Ein Objekt in Old erhält eine Referenz auf ein Objekt in Young

↓

Speicherbereich in Old wird als „dirty“ markiert (Bit in card-table gesetzt), bei nächster GC separat überprüft und „dirty“-Bit wieder gelöscht

↓

− Zusätzlicher Speicher für card-table benötigt

↓

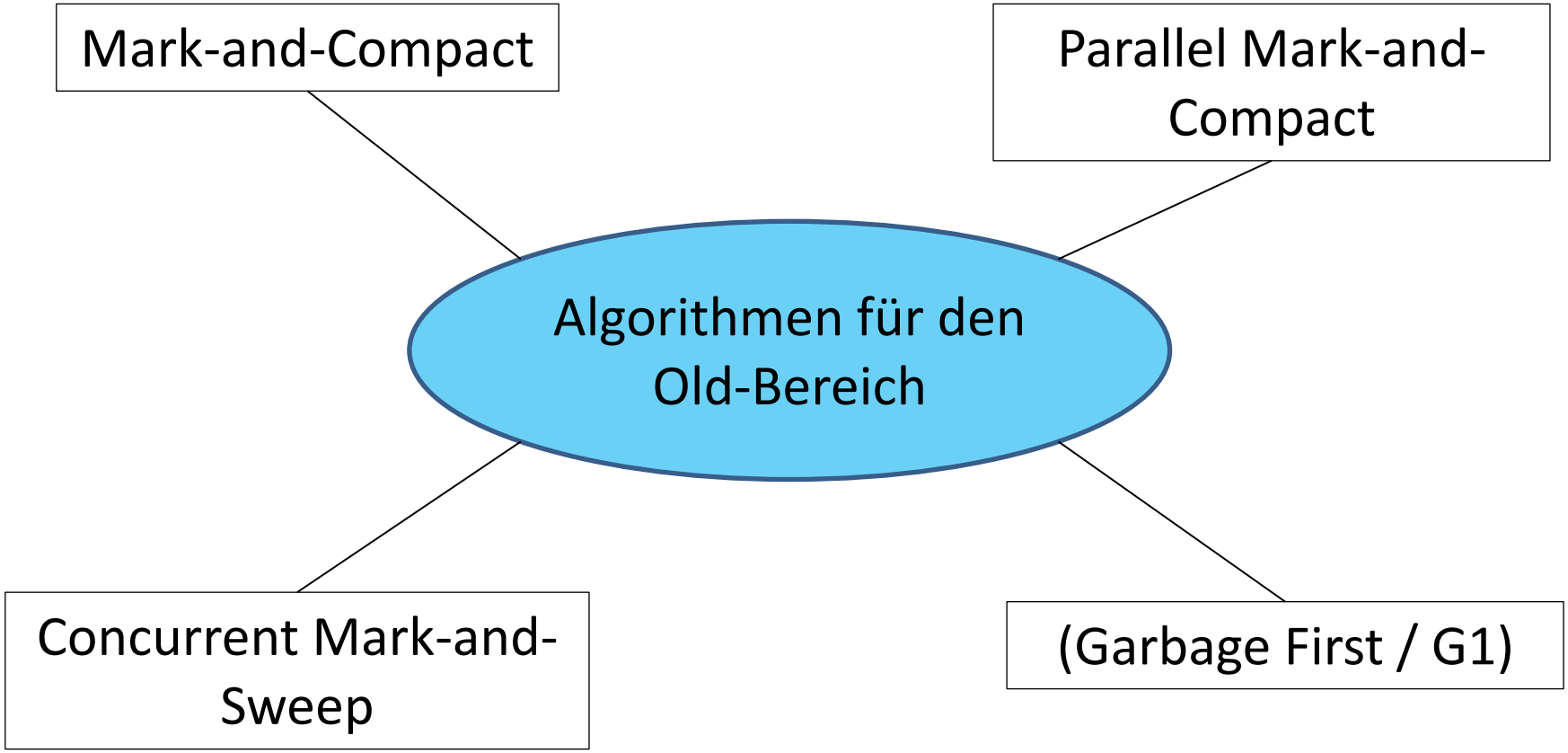
− Zusatzaufwand für setzen der / suchen nach „dirty“-Bits

# Garbage Collection
# GC-Algorithmen

## Zwischenfazit

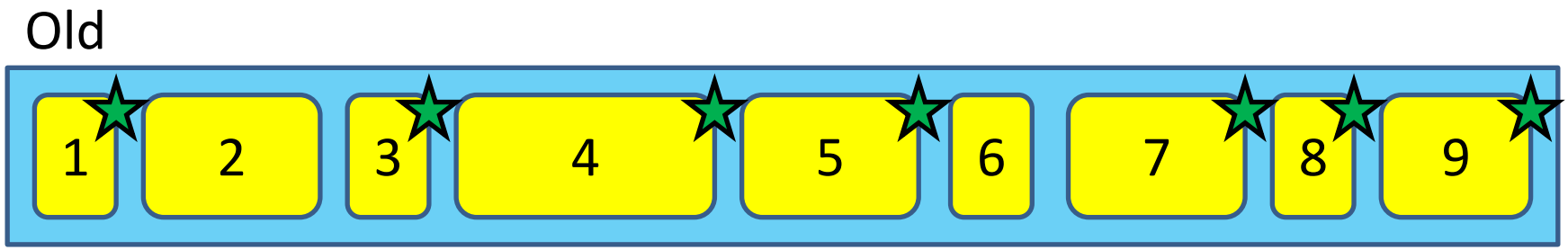
- Generational Heap stark falls
  - viele kurzlebige Objekte
  - wenige Referenzen von Old in Young
  - meistens der Fall
- Noch ein Nachteil offen
  - Mark-and-Copy ist stop-the-world
  - Reduzierung der Auswirkungen durch Parallelisierung

# Garbage Collection
# GC-Algorithmen



# Garbage Collection
# GC-Algorithmen

## Mark-and-Compact

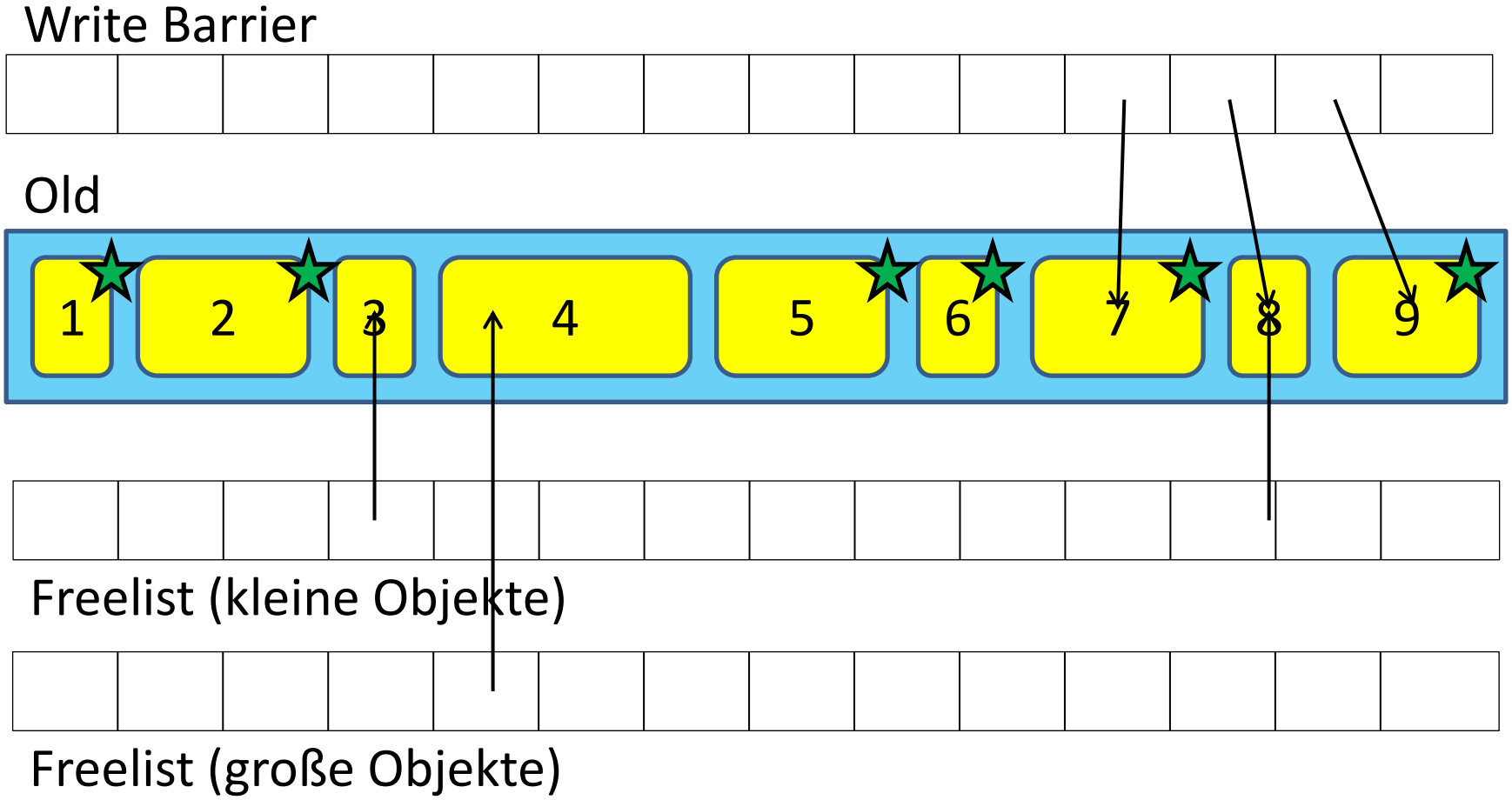



Leichte Fragmentierung möglich

Stop-the-world und durch großen Old-Bereich und aufwändigen Algorithmus lange Pausenzeiten. Wird aber deutlich seltener durchgeführt als Minor GC

# Garbage Collection
# GC-Algorithmen

## Concurrent Mark-and-Sweep (CMS)



# Garbage Collection
# GC-Algorithmen

## Concurrent Mark-and-Sweep (CMS)

\+ **Kurze stop-the-world-Pausen**

\- Erhöhter Rechenaufwand und längere Laufzeit → höhere Belastung des Prozessors

\- Längere Pausen bei Minor GC, da Allokation in Old länger dauert

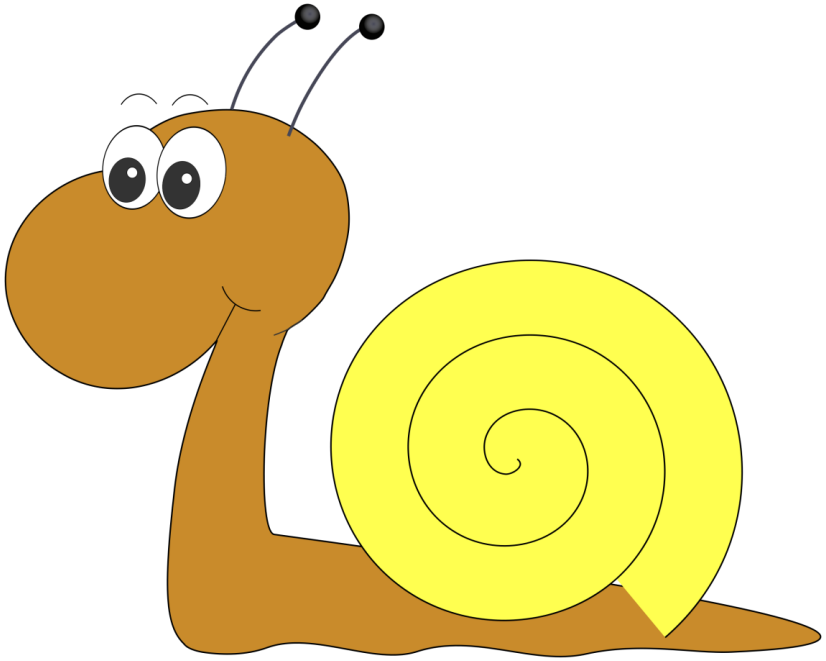
\- höherer Speicherverbrauch, da GC länger dauert und die Anwendung also länger Zeit hat, Objekte zu produzieren

\- Fragmentierung

# Agenda

- Konfiguration der JVM
- Speicherbereiche der JVM
- Garbage Collection
- Monitoring und Tuning
- Beispiel
- It‘s your turn

# Monitoring und Tuning

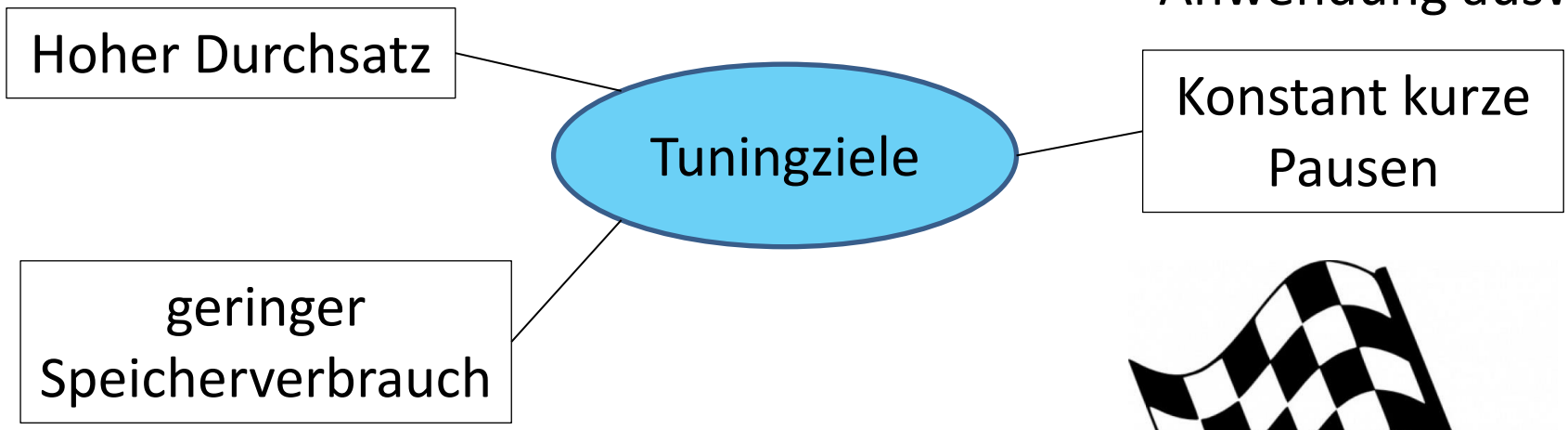


Vor GC- und Speichertuning zunächst nach Fehlern in der Anwendung suchen!

# Monitoring und Tuning
# Tuningziele

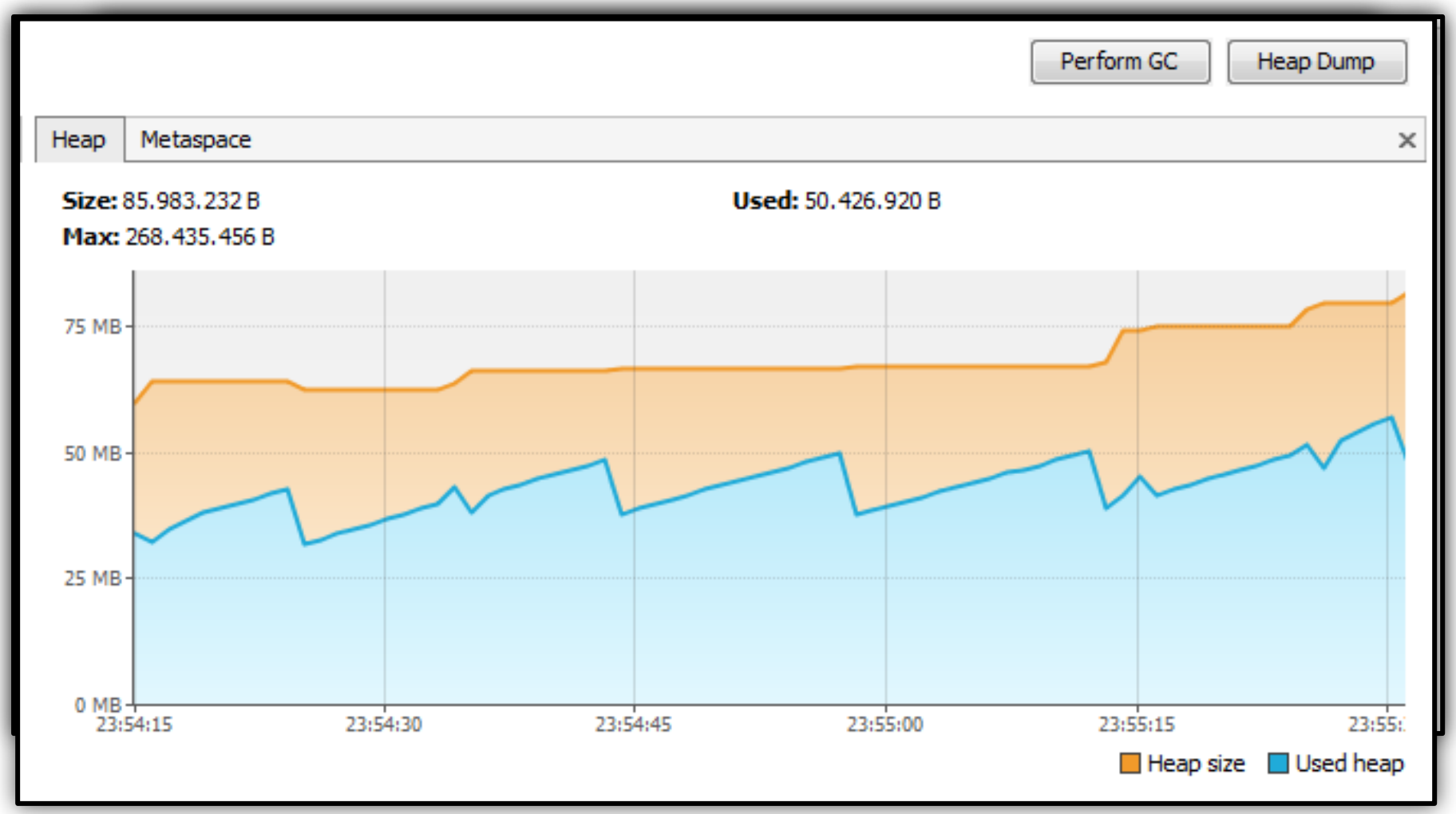
Zeit relativ zur Gesamtzeit, die die JVM der Anwendung zur Verfügung steht

keine langen Pausen, da diese sich auf die Performance der Anwendung auswirken

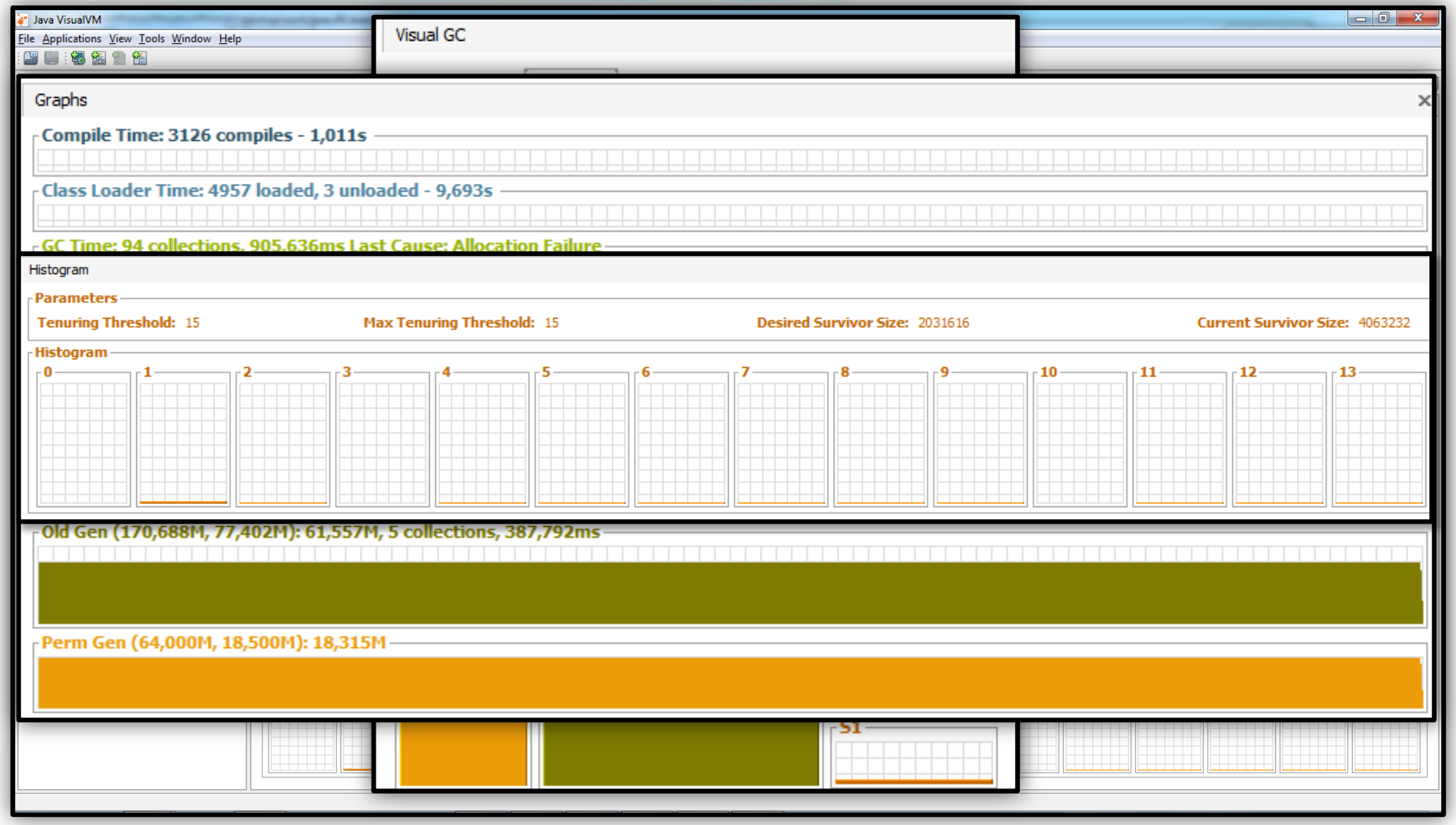



Häufig Fehler im Programmablauf und nicht unbedingt Probleme des GC.
Möglicherweise durch Einbau von mehr Speicher lösbar

# Monitoring und Tuning
# jVisualVM



# Monitoring und Tuning
# VisualGC



# Monitoring und Tuning
# Tuningmaßnahmen

## Auswahl des richtigen GC

1 bis 4 Kerne typischerweise serielle GC, da ansonsten Verwaltungsoverhead für die Synchronisierung zu hoch

bei 4 Kernen und mehr ist parallel GC typischerweise schneller

# Monitoring und Tuning
# Tuningmaßnahmen

Hoher Durchsatz: JVM bringt möglichst wenig Zeit mit GC zu

+ Ansatz: Objekte sterben möglichst jung (am Besten in Eden)

- Kopieren teuer. Je früher Objekte sterben, desto weniger wird kopiert.
- GC in Old ist teurer als in Young. Landen weniger Objekte in Old, seltener Major GC
- Eden bzw. Survivor können vergrößert werden
- TenuringThreshold erhöhen, damit Objekte später in Old landen

# Monitoring und Tuning
# Tuningmaßnahmen

## Konstant kurze Pausen

Ansatz: Kein Sterben in Old
→ keine GC → keine Pausen
→ keine Fragmentierung

Objektsterben in Old verringern durch Vergrößerung Eden / Survivor oder Erhöhung TenuringThreshold

Ansatz: GC-Häufigkeit

GC zu häufig: blockiert CPU

GC zu selten: schnell kein Speicher mehr

GC läuft wenn Old-Generation bestimmten Füllgrad. Einstellbar.

# Monitoring und Tuning
# Tuningmaßnahmen

## Fazit

- Hoher Durchsatz und konstant kurze Pausen konkurrieren miteinander
  - Selten GC: lange Pausen, hoher Durchsatz
  - Häufig GC: konstant kurze Pausen, geringerer Durchsatz
  - Entscheidung je nach Anwendungsfall

# Quellen


- http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-1401022.html
- http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html
- https://blog.codecentric.de/2010/01/java-outofmemoryerror-1-akt-das-java-memory-modell-stellt-sich-vor/
- https://blog.codecentric.de/en/2012/08/useful-jvm-flags-part-5-young-generation-garbage-collection/
- http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf
- http://www.angelikalanger.com/Articles/EffectiveJava/50.GC.YoungGenGC/50.GC.YoungGenGC.html
- http://www.angelikalanger.com/Articles/EffectiveJava/49.GC.GenerationalGC/49.GC.GenerationalGC.html
- http://apmblog.compuware.com/2011/05/11/how-garbage-collection-differs-in-the-three-big-jvms/
- http://java-blog.de/2010/09/12/jvm-speicher-heap-gc-und-co/

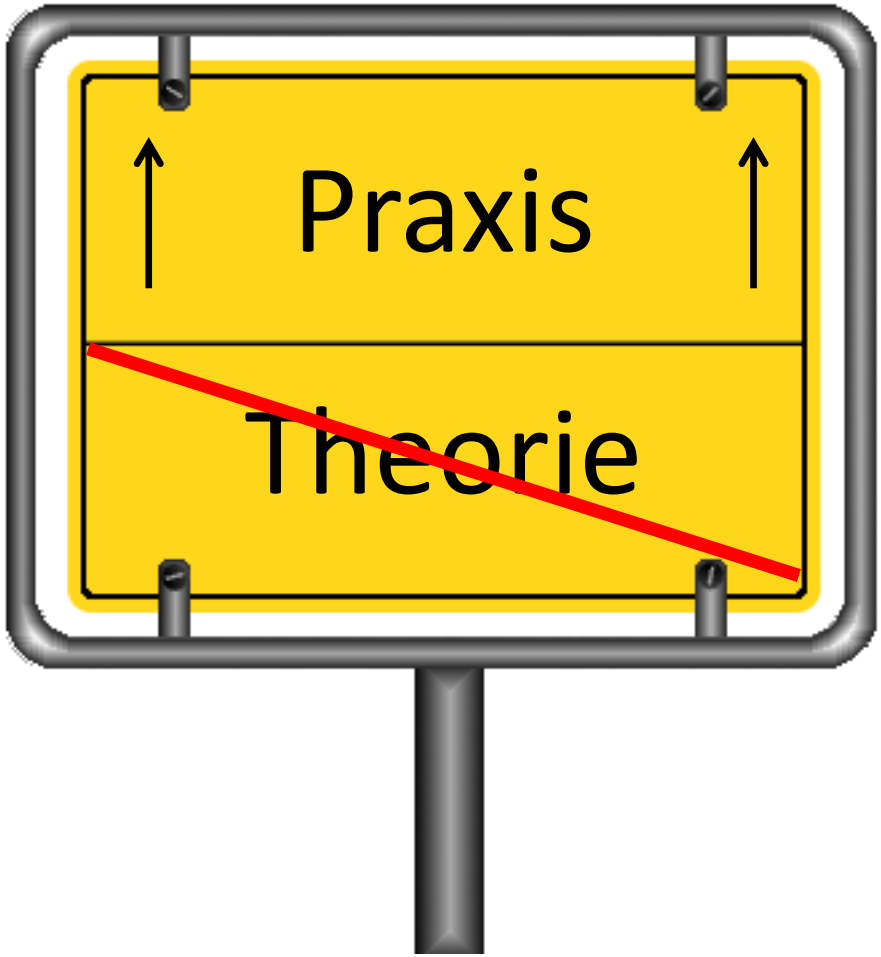
# Quellen


- https://www.codecentric.de/kompetenzen/publikationen/der-garbage-collector-das-unbekannte-wesen/
- http://javabook.compuware.com/content/memory/the-three-jvms.aspx
- http://www.angelikalanger.com/Articles/EffectiveJava/51.GC.OldGen.MarkCompact/51.GC.OldGen.MarkCompact.html
- http://www.angelikalanger.com/Articles/EffectiveJava/52.GC.OldGen.CMS/52.GC.OldGen.CMS.html
- http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html
- http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All
- http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html
- http://www.angelikalanger.com/Articles/EffectiveJava/55.GC.G1.Overview/55.GC.G1.Overview.html
- http://www.angelikalanger.com/Articles/EffectiveJava/56.GC.G1.Details/56.GC.G1.Details.html

# Quellen


- http://de.slideshare.net/MonicaBeckwith/con5497
- https://blogs.oracle.com/poonam/entry/understanding_g1_gc_logs
- http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html
- http://www.angelikalanger.com/Articles/EffectiveJava/53.GC.Tuning.Goals/53.GC.Tuning.Goals.html
- http://www.angelikalanger.com/Articles/EffectiveJava/54.GC.Tuning.Details/54.GC.Tuning.Details.html
- http://stackoverflow.com/questions/12130107/difference-between-sampling-and-profiling-in-jvisualvm
- http://www.oracle.com/technetwork/java/visualgc-136680.html

# JVM-Speichermanagement



# Agenda

- Konfiguration der JVM
- Speicherbereiche der JVM
- Garbage Collection
- Monitoring und Tuning
- Beispiel
- It‘s your turn