Threads, dynamische Speicherverwaltung

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

- Sie können Algorithmen in Java mittels Parallelisierung optimieren (Fortsetzung).
- Sie wissen, wie der Speicher in Programmen verwendet wird.
- Sie kennen die Aufgaben des Speicherverwalters.
- Sie wissen, wie die automatische Speicherfreigabe funktioniert.
- Sie wissen, was Weak References und Finalizer sind und können damit umgehen.

Basiert auf Material von:

Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

Parallelisierung (Fortsetzung) - Thread-Pool

Parallelisierung: Threads und Quick-Sort

```
class NaiveParallelQuicksort2 extends Thread {
   ...
   private final int SPLIT_THRESHOLD = 100000;

   public void run() {
      int mid = 0;
      Thread t1 = null;
      Thread t2 = null;

      if (left < right) {
         mid = partition(arr, left, right);
         if (mid - left > SPLIT_THRESHOLD) {
            t1 = new NaiveParallelQuicksort2(arr, left, l-1);
            t1.start();
         } else {
            Quicksort.quickSort(arr, left, l-1);
         }
         if (right - mid > SPLIT_THRESHOLD) {
            t2 = new NaiveParallelQuicksort2(arr, mid, right);
            t2.start();
         } else {
            Quicksort.quickSort(arr, mid, right);
         }
      if (t1 != null) t1.join();
      if (t2 != null) t2.join();
   }
}
```

Letzte Folie vorhergehende Lektion.

Nur falls Task genügend gross ist parallel ausführen.

Sequentielle Ausführung des Tasks.

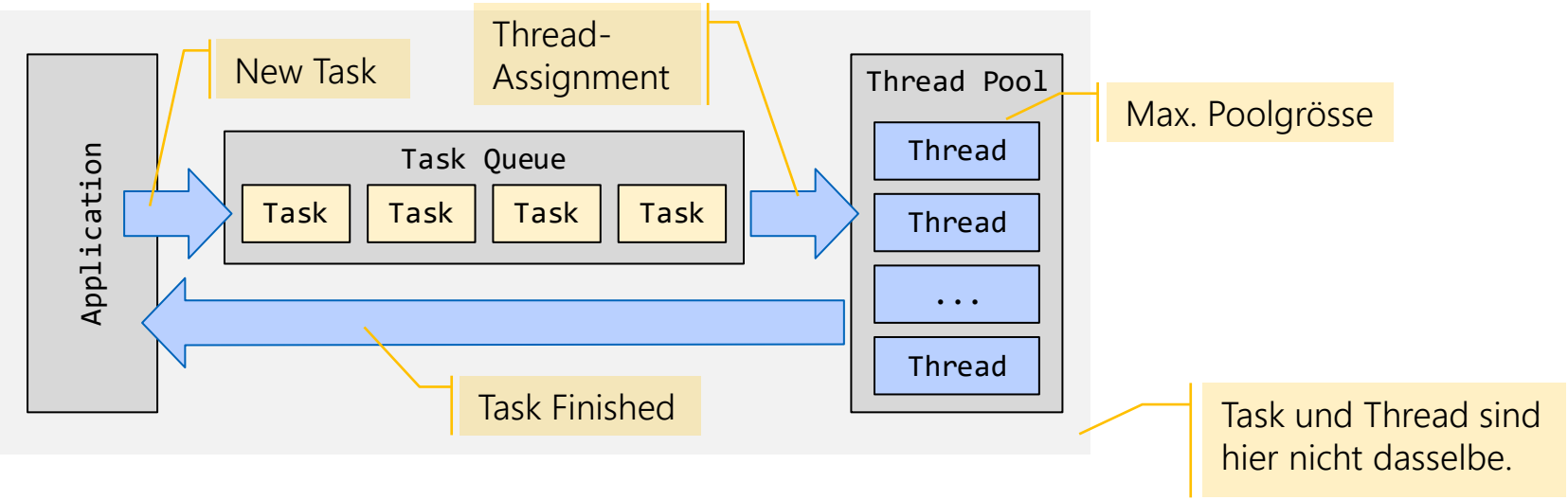
Thread-Pool: Threads und Tasks

- Es wird einer gewissen Anzahl von einmalig gestarteten Threads (Pool von Threads) immer wieder eine neue Aufgabe (Task) übergeben.
- Pool-Grösse entspricht in etwa der Anzahl Rechnerkerne:

```
java.lang.Runtime.getRuntime().availableProcessors()
```

  Bei I/O-intensiven Tasks mehr, da immer ein Teil wartend/blockiert ist.

- Die Reihenfolge der Abarbeitung der Tasks wird durch eine Ausführungs-strategie bestimmt (Thread-Pool-Typ).



Thread-Pool: Ausführungsstrategien

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Bessere Kontrolle über Ressourcen durch Einhaltung einer Ausführungsstrategie:

- In welcher Reihenfolge werden die Tasks in der Queue ausgeführt (LIFO, FIFO, priorisiert, ...).
- Anzahl der Tasks die gleichzeitig, parallel ausgeführt werden dürfen.
- Anzahl der Tasks die auf Ausführung warten dürfen.
- Periodische Threads (zeitgesteuert).

Auswahl effektivster Policy hängt vom Anwendungsfall ab.

Thread-Pool: Java-Implementation

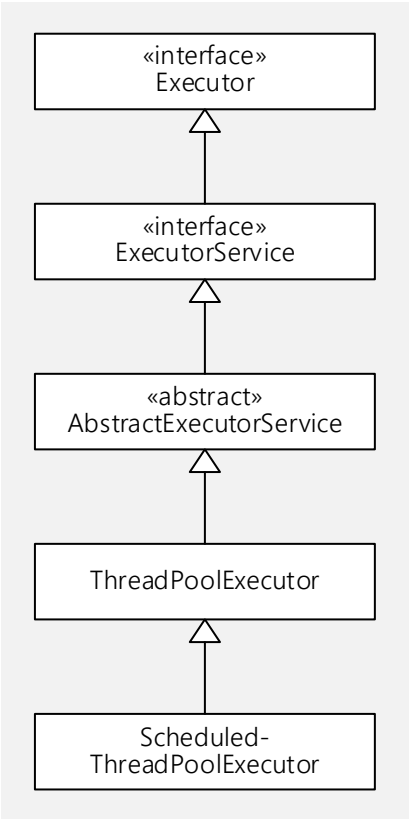
- Interface Executor (java.util.concurrent)
  stellt Framework zur Thread-Ausführung bereit,
  das verschiedene Ausführungsstrategien realisiert.
- Interface ExecutorService für Thread-Pools.
- Verschiedene Varianten des ThreadPoolExecutor
  werden durch Factory-Methoden in der Klasse
  (nicht Interface) Executors bereitgestellt:
  - FixedThreadPool:
    Erstellt einen Thread-Pool, der eine feste Anzahl von Threads
    wiederverwendet, die von einer gemeinsam genutzten, unbegrenzten
    Warteschlange aus operieren.

    ```
    static ExecutorService newFixedThreadPool(int nThreads)
    ```

  - CachedThreadPool:
    Erstellt einen Thread-Pool, der bei Bedarf neue Threads erstellt, zuvor
    erstellte Threads jedoch wiederverwendet, sobald sie verfügbar sind.
    Threads, die sechzig Sekunden lang nicht benutzt wurden, werden
    beendet und aus dem Cache entfernt.

    ```
    static ExecutorService newCachedThreadPool()
    ```

Factory-Methoden des Executors.



Thread-Pool: Java-Implementation

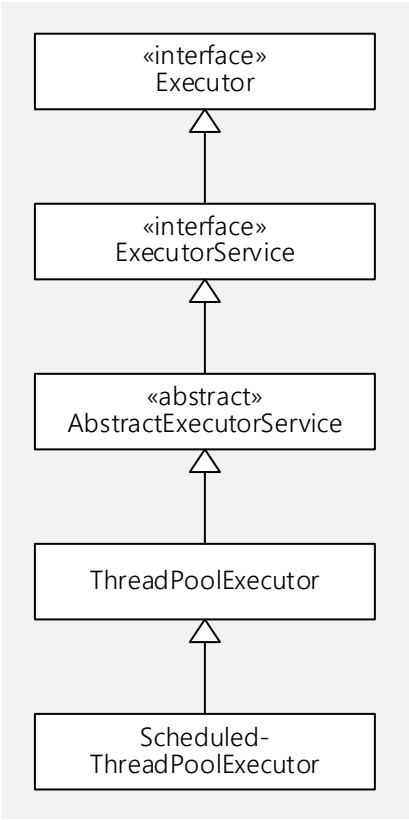
- (Fortsetzung)
  Verschiedene Varianten des ThreadPoolExecutors werden durch Factory-Methoden in der Klasse (nicht Interface) Executors bereitgestellt:
  - SingleThreadExecutor:
    Einzelner Thread, der eingereichte Tasks nacheinander (gemäß FIFO- oder LIFO-Prinzip oder nach Priorität) abarbeitet.

```
public static ExecutorService newSingleThreadExecutor()
```

  - ScheduledThreadPool:
    Pool mit fester Menge von Threads, die zur zeitgesteuerten bzw. wiederkehrenden Ausführung eingereichter Tasks eingesetzt werden kann.

```
public static ScheduledExecutorService
   newScheduledThreadPool(int corePoolSize)
```

  - Und andere...



Thread-Pool: Lebenszyklus

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Drei Zustände des Thread-Pools:

1. running:
   Threadpool nimmt Tasks entgegen und führt sie aus, sobald Threads verfügbar sind.

2. shutting down, Varianten:
   - Graceful Shutdown:
     Threadpool führt laufende und bereits angenommene, aber nicht begonnene Tasks noch aus, nimmt jedoch keine neuen Tasks mehr an.

     ```
     void shutdown()
     ```

   - Abrupt Shutdown:
     Versucht, alle aktiven Task zu stoppen, stoppt die Verarbeitung von wartenden Tasks und gibt eine Liste der Tasks zurück, die auf ihre Ausführung warteten.

     ```
     List<Runnable> shutdownNow()
     ```

3. terminated:
   Keine Tasks werden mehr ausgeführt oder angenommen.

Thread-Pool: Wie kommen Threads in den Pool?

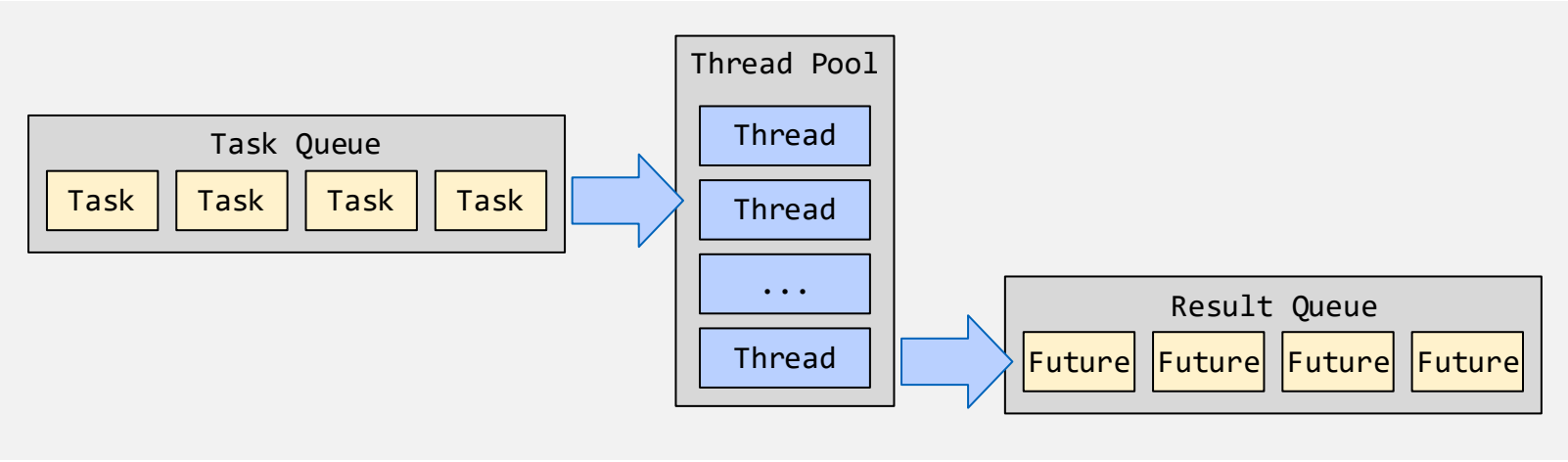
- Einstellen eines Tasks in einen Threadpool:

```
Future<?> submit(Runnable task)
```

Ohne Rückgabewert.

```
<T> Future<T> submit(Callable<T> task)
```

Mit Rückgabewert.



- Verwendung von Callable-Interface wenn Rückgabewerte/Resultate benötigt werden (Runnable liefert NULL-Wert bei Erfolg).
- Ein Future (Interface) stellt das Ergebnis einer asynchronen Berechnung dar.

Thread-Pool: Resultat der Ausführung

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Submit in Thread-Pool liefert **`Future`**:

- Mit **`get()`**-Methode zur Abfrage des Rückgabewerts vom Typ V:
  - Wartet, bis Rückgabewert feststeht.
  - Ggf. nur solange, bis angegebener Timeout erreicht.
- mit **`cancel()`**-Methode zur Stornierung der Aufgabe:
  - unmittelbare Streichung, wenn Bearbeitung noch nicht begonnen.
  - Abbruchversuch, wenn schon in Bearbeitung.

```
public interface Future<V> {
   V get();
   V get(long timeout, TimeUnit unit);
   boolean cancel(boolean mayInterruptIfRunning);
   boolean isCancelled();
   boolean isDone();
}
```

Wartet auf Beendigung des Tasks

Thread-Pool: Resultat der Ausführung

Beispiel einer Realisierung eines Tasks mit Rückgabetyp V in der call()-Methode (Implementierung des Interfaces Callable):

```
public interface Callable<V> {
   V call() throws Exception;
}
```

```
class Foo implements Callable<Integer> {
   ...

   public Integer call() {
      int result = ...;
      return result;
   }

}
```

Task mit Integer als Rückgabewert.

Thread-Pool: Thread-Safe-Collections

Im Paket java.util.concurrent gibt es einige Thread-Safe-Collections. Im Fall von erwarteten Race-Conditions (oder ähnlichem) sollten diese Klassen verwendet werden:

- ConcurrentHashMap<K,V>
- ConcurrentLinkedQueue<E>
- ConcurrentLinkedDequeue<E>
- SynchronousQueue<E>
- Und weitere…

Thread-Pool: Beispiel Quick-Sort-Task

```
public class QuicksortTask implements Runnable {
   static ExecutorService threadPool;
   static ConcurrentLinkedQueue<Future> futureList;

   public void sort(int[] a) {
      int parallelism = java.lang.Runtime.getRuntime().availableProcessors()*2;
      threadPool = Executors.newFixedThreadPool(parallelism);

      futureList = new ConcurrentLinkedQueue<Future>();
      QuicksortTask rootTask = new QuicksortTask(a, 0, a.length - 1);
      futureList.add(threadPool.submit(rootTask));
      while (!futureList.isEmpty()) {
         futureList.poll().get();
      }
      threadPool.shutdown();
   }

   public void run() {
      int l;
      if (left < right) {
         l = partition(a, left, right);
         if (l - left > SPLIT_THRESHOLD) {
            futureList.add(threadPool.submit(new QuicksortTask(a, left, l-1)));
            ...
```

Wir erwarten von den Tasks keine Rückgabewerte.

Initialisiere Thread-Pool.

Starte ersten Task.

Warte auf Beendigung aller Threads.

Terminiere den Thread-Pool.

Übergebe neuen Task an Thread-Pool.

Fork/Join

Fork/Join: Idee

- Macht grundsätzlich dasselbe wie Threadpools.
- Threads manchmal immer noch zu schwergewichtig.
- "Teile und Herrsche"-Prinzip rekursiv auf Parallelität angewandt.

Für rekursive Tasks.

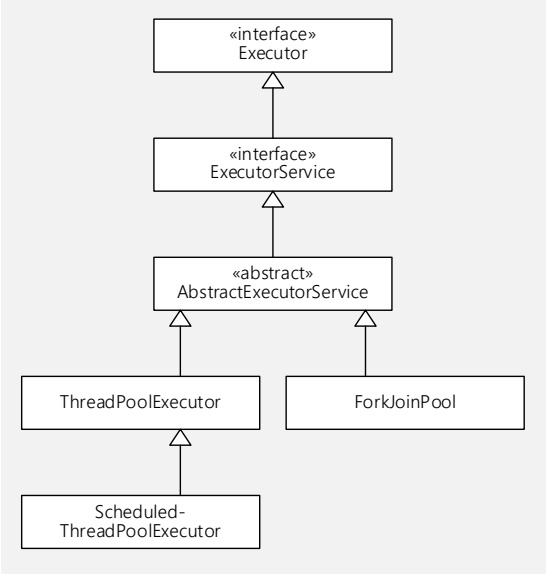
```
solve(Problem problem) {
   if (problem is small)
      directly solve problem
   else {
      split problem into independent parts
      fork new subtasks to solve each part
      join all subtasks
      compose result from subresults
   }
}
```

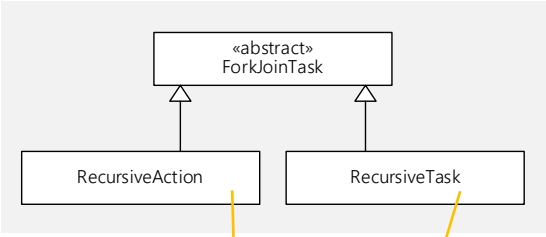
Fork/Join: So wird es gemacht

1. Definiere ein ForkJoinPool-Objekt.
2. Definiere ein Task-Objekt das von der Klasse RecursiveTask (mit Resultat) erbt.
3. Instanziere den ForkJoinPool.
4. Instanziiere (Root-)Task-Objekt und rufe **`invoke()`** auf.

   Analog run()-Methode bei Threads.
5. In der **`compute()`**-Methode (innerhalb Task-Objekt) Code mit der Aufteilung des Problems:
   1. Löse Problem direkt falls klein/einfach genug.
   2. Sonst:
      Teile Problem auf: Rufe `invokeAll(task1, task2, …)` auf; startet alle Tasks und wartet bis alle beendet sind.
      Oder:
      Starte Tasks mit `fork()` mit abschliessenden `join()` und mit `invoke()` beim Letzten.

      Äquivalent zu fork() mit anschliessendem join(), versucht aber immer, die Ausführung im aktuellen Thread zu beginnen.





ohne Rückgabewert

mit Rückgabewert

Fork/Join: Beispiel Quick-Sort-Task (invokeAll)

```
public class QuicksortForkJoin extends RecursiveAction {

   public static void sort(int[] arr) {
      int parallelism = java.lang.Runtime.getRuntime().availableProcessors();
      forkJoinPool = new ForkJoinPool(parallelism);
      QuicksortForkJoin rootTask = new QuicksortForkJoin(arr, 0, arr.length - 1);
      forkJoinPool.invoke(rootTask);
   }

   public void compute() {
      int mid;
      if (left < right) {
         mid = partition(arr, left, right);
         ForkJoinTask t1, t2;
         if (mid - left  > SPLIT_THRESHOLD && right - mid > SPLIT_THRESHOLD) {
            t1 = new QuicksortForkJoin(arr, left, mid - 1);
            t2 = new QuicksortForkJoin(arr, mid , right);
            invokeAll(t1, t2);
         } else {
            Quicksort.sort(arr, left, mid - 1);
            Quicksort.sort(arr, mid , right);
         }
      }
   }

}
```

Übergibt den Task an den Pool, der die compute()-Methode aufruft, und wartet auf Ergebnis.

Beide oder keiner in einem Thread ausgeführt.

Übergebe Tasks t1 und t2 dem Pool und warte auf Terminierung beider Tasks.

Fork/Join: Beispiel Quick-Sort-Task (invoke)

```
public class QuicksortForkJoin2 extends RecursiveAction {

   public static void sort(int[] arr) {
      int parallelism = java.lang.Runtime.getRuntime().availableProcessors();
      forkJoinPool = new ForkJoinPool(parallelism);
      QuicksortForkJoin2 rootTask = new QuicksortForkJoin2(arr, 0,arr.length-1);
      forkJoinPool.invoke(rootTask);
   }

   public void compute() {
      int mid;
      ForkJoinTask t1 = null;

      if (left < right) {
         mid = partition(arr, left, right);
         if (mid - left  > SPLIT_THRESHOLD) {
            t1 = new QuicksortForkJoin2(arr, left, mid-1).fork();
         } else Quicksort.sort(arr, left, mid-1);
         if (right - mid > SPLIT_THRESHOLD) {
            new QuicksortForkJoin2(arr, mid , right).invoke();
         } else Quicksort.sort(arr, mid , right);
         if (t1 != null) {
            t1.join();
         }
      }
   }
```

Übergebe Task t1 dem Pool und fahre mit der Ausführung fort.

Übergebe Task t2 dem Pool und warte auf Ergebnis.

Warte bis Tasks T1 abgearbeitet ist.

Fork/Join: Threads vs. Fork/Join

Besser Runnable Interface verwenden.

Threads	Fork/Join
subclass Thread	**subclass** RecursiveTask/Action<V>
override run	**override** compute
call start	**call** invoke, invokeAll, fork
call join (je Thread)	**call** join which returns answer or **call** invokeAll on multiple tasks

Umgang mit dem Hauptspeicher

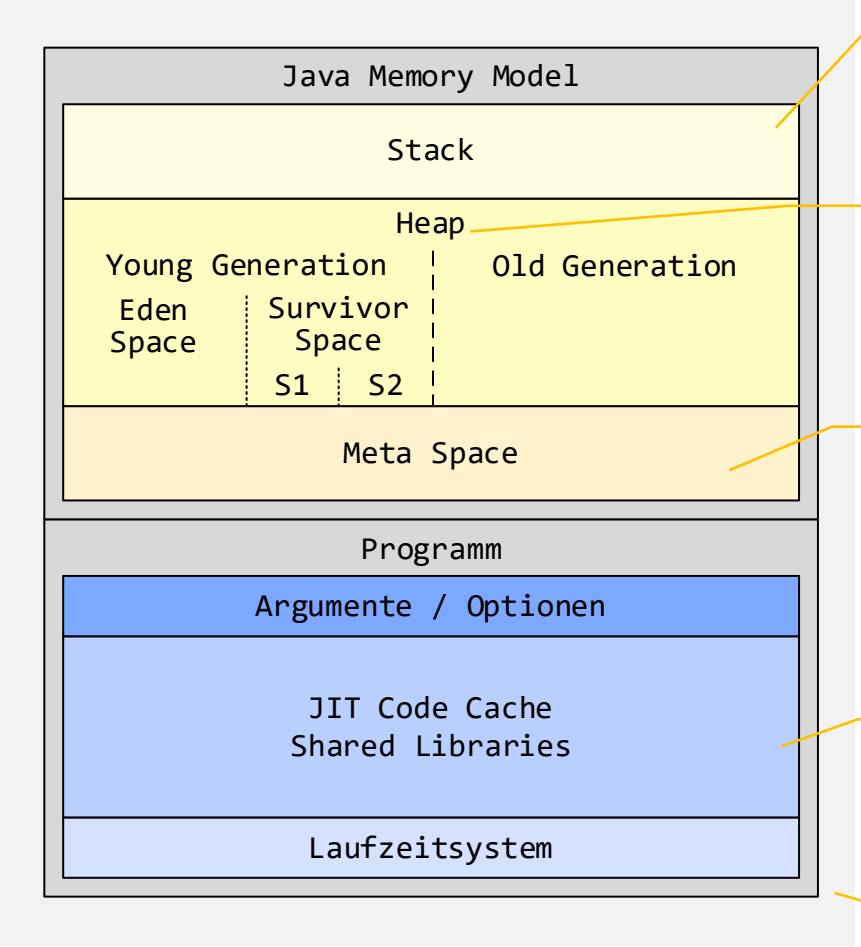
Dynamische Speicherverwaltung

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

- Der Umgang mit dem dynamischen Speicher (Heap) hat früher bis zu 40% der Entwicklungszeit verursacht (inkl. Fehlersuche).
- Fatale Programmfehler (Abstürze) sind/waren 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.
- Der Umgang mit dem dynamischen Speicher ist für die Effizienz des Programms oft ausschlaggebend.
- 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 (GC, Müllsammler), bzw. Gargabe-Collector.

Java Memory Model

Speicherverwendung in Java seit Version 8:



Rücksprungadressen, lokale Variablen.

Objekte (Objekt-Variablen). Die Aufteilung des Heaps wird auf den folgenden Folien erläutert.

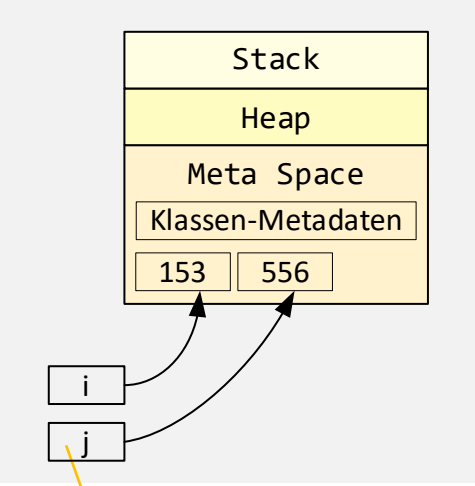
Metadaten und statische Variablen zu den Klassen, sowie weitere Metadaten (Non Heap Space, Java-eigener = native Heap). Bis Java 7 als Permanent Generation bezeichnet.

Häufig abgerufene Codeblöcke werden vom JIT zu nativem Code kompiliert und im Code-Cache gespeichert.

Vorsicht – abhängig von konkreter JVM.

Java Memory Model: Meta Space

- Einfachste Zuteilungsstrategie: Der Compiler legt die Zuordnung von Variablenname zu relativen Speicheradressen fest.
- Speicher wird beim Start des Programms angefordert und beim Beenden wieder freigegeben.
- Meta Space enthält Klassendefinitionen (Class Object) inklusive deren statischer Variablen.

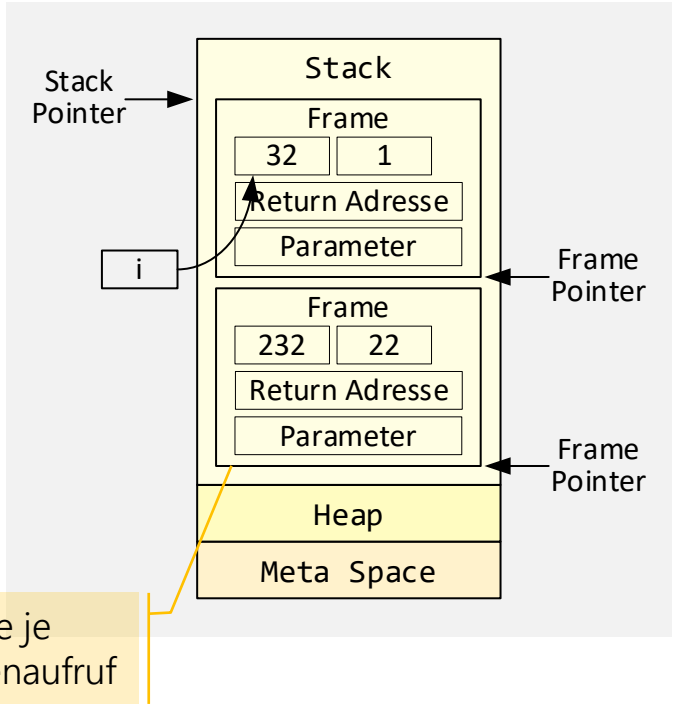


\+ Vorteile:
- \+ Einfach
- \+ Beim Start kann schon gesagt werden, ob Speicher ausreicht (bei Java nicht mehr seit Java 8).

\- Nachteile:
- – Die Grössen aller Datenstrukturen müssen zur Übersetzungszeit bekannt sein.
- – Keine rekursiven Programmaufrufe möglich.
- – Keine dynamischen Datenstrukturen wie Listen und Bäume möglich.

Java Memory Model: Stack

- Bei jedem Aufruf einer Methode wird ein Speicherbereich reserviert: Frame.
- Frames werden als Stack organisiert: LIFO.
- Die lokalen Variablen der Methoden werden relativ zum Framepointer angesprochen.

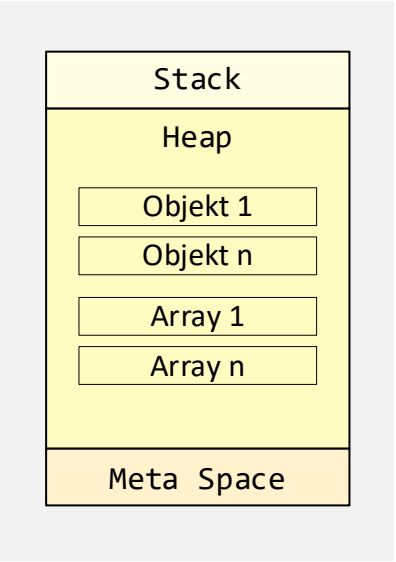


\+ Vorteile:
- \+ rekursive Aufrufe sind möglich.
- \+ effiziente Zuteilung/Freigabe von Speicher.

\- 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.

Java Memory Model: Heap

- Der Heap verwaltet die Instanzen von Klassen (Objekte) und Arrays.
- Der Speicher muss vom Programm mittels new angefordert und, je nach Sprache, mittels delete explizit wieder freigegeben werden.

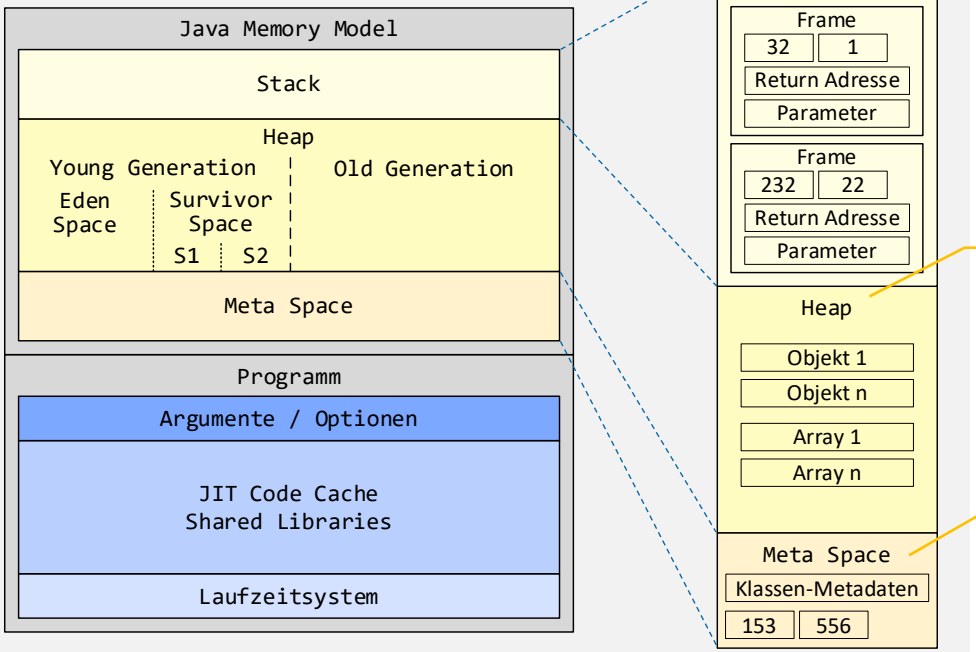


\+ Vorteile:
- \+ Die Grösse der einzelnen Datenstrukturen kann zur Laufzeit festgelegt werden.
- \+ Dynamische Datenstrukturen, z.B. Listen, sind möglich.

\- Nachteile:
- – Zuteilung/Freigabe der Daten ist relativ rechenintensiv und kompliziert.
- – Speicher muss explizit vom Programm angefordert und wieder freigegeben werden
  → Programmierfehler möglich (Memory Leak).
- – Speicherüberlauf (Heap-Overflow) möglich.

Java Memory Model: Gesamtsicht

Welche Algorithmen werden zur Speicherverwaltung eingesetzt und wie können wir diese nutzen?



Beim Rücksprung aus der Methode wird benutzter Speicher automatisch zurückgegeben.

Hier können Objekte liegen bleiben, die nicht mehr benutzt werden und unnötig Speicher belegen.

Auch im Meta-Space liegen allenfalls Metadaten die nicht mehr benötigt werden.

Einfache Speicherverwaltung (Do-it-yourself)

Einfache Speicherverwaltung: Freie Zuteilung

- 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 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:
  - Im Hauptspeicher bilden sich mit der Zeit Löcher, (externe) Fragmentierung.
- Lösung:
  - Der Speicher wird periodisch kompaktiert.

Einfache Speicherverwaltung: Freie Zuteilung

- Speichermanager:
  - 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);
}
```

- Objektorientierte Sprachen:
  - Die Grösse der Objekte ist dem System bekannt, also Grössenangabe überflüssig.
  - Es wird zusätzlich ein Konstruktor aufgerufen.

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

```
MyObject obj = new MyObject("hallo");
```

In Java gibt es die Klasse Storage nicht, das Anlegen von Speicher ist Teil der Sprache (new).

Einfache Speicherverwaltung: Probleme

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

1. Vergessen Speicher anzufordern (z.B. mit new):
   - Konsequenz: Referenz-Variable enthält einen zufälligen Wert → es wird auf eine beliebige, u.U. benutzte Speicherstelle zugegriffen und verändert → «komische» Werte evtl. Programmabbruch.
   - Abhilfe in Java: Referenz-Variablen mit einem Null-Wert initialisieren → NullPointerException.

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

3. Vergessen den Speicher freizugeben: Memory-Leak
   - Konsequenz:
     - Das Programm benötigt immer mehr Speicher.
     - Bei virtuellem Speicher immer mehr auf Disk ausgelagert → langsamer, später Abbruch.
   - Abhilfe in Java: Der automatische Speicherverwalter gibt den Speicher frei.

4. Der Speicher wird freigegeben obwohl noch verwendet: Dangling Pointer
   - Konsequenz: Es wird auf eine anderweitig benutzte Speicherstelle zugegriffen und verändert → «komische» sich plötzlich verändernde Werte oft/meist Programmabbruch.
   - Abhilfe in Java: Nur der automatische Speicherverwalter gibt den Speicher frei.

Automatische Speicherverwaltung

Automatische Speicherverwaltung

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Hauptaufgabe der automatischen Speicherverwaltung ist die Freigabe des nicht mehr benötigten Speichers.

Einfache Algorithmen (keine Laufzeitinformation notwendig):

1. Referenzzählung
2. Smart-Pointer

Für Java nicht geeignet.

Vollautomatische Algorithmen / Garbage-Collection (Laufzeitinform. notwendig):

1. Mark-Sweep-GC
2. Mark-Compact-GC
3. Copying-GC
4. Generational-GC

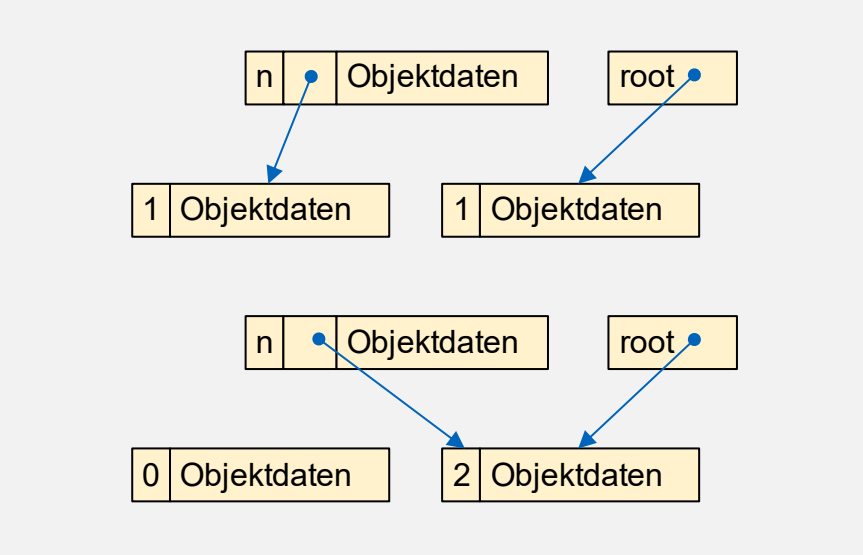
Konkrete Implementation(en) in Java abhängig von jeweiliger JVM.

Alle Verfahren haben Vor- und Nachteile. Performance-Einbusse ca. 5-10%.

Einfache Algorithmen: Referenzzählung

- Es wird gezählt, wie viele Referenzen auf ein Objekt verweisen.
- Wenn keine Referenz mehr vorhanden ist, dann kann Objekt gelöscht werden.
- Operationen des Referenzzähler
  - Bei einer Zuweisung wird der Referenzzähler um 1 erhöht.
  - Bei Wegnahme einer Referenz wird der Referenzzähler um 1 erniedrigt.

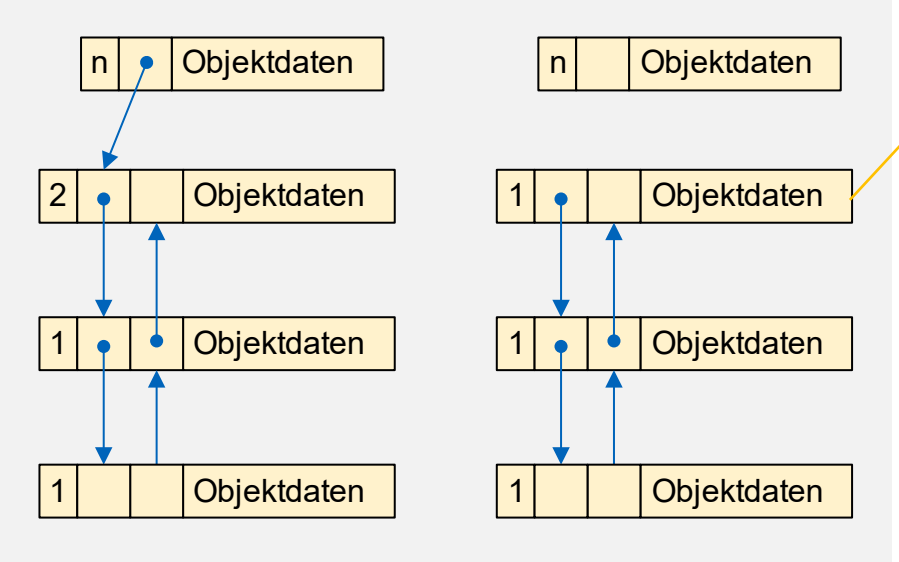
```
void addRef() {
   referenceCount++
}

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




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

Einfache Algorithmen: Referenzzählung

- Vorteile der Referenzzählung
  - Einfach, geringer Verwaltungsaufwand.
  - Speicher wird zum frühestmöglichen 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.



Einfache Algorithmen: Smart-Pointers

- Objekt-Referenzen sind nicht einfach «dumme» Adressen sondern «smarte» Objekte mit Referenz-Zähler.
- Smart-Pointer merken selber:
  - Wenn ihnen ein neuer Wert zugewiesen wird.
  - Wenn Sie nicht mehr zugreifbar sind (out-of-scope gehen).
- In C++ mit Operator-Overloading, kann wie normaler Zeiger (Raw-Pointer) verwendet werden. C++ kennt kein Garbage-Collection, es wird mit Smart-Pointern in Kombination mit Destruktoren gearbeitet.
- Vorteil: keine Fehler beim Erhöhen und Erniedrigen des Referenz-Zählers.
- Nachteile: wie Referenzzählung, zyklischen Datenstrukturen werden nicht erkannt.

Vollautomatische Algorithmen

Vollautomatisch heisst: 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.

- In Java: Speicher kann freigegeben werden, wenn er nicht mehr direkt oder indirekt referenziert wird. Der Speicherverwalter muss hierfür alle Referenzketten traversieren:

  1. Startpunkt der Referenzketten → alle Wurzelobjekte:
     - Alle statischen Variablen der Klassen.
     - Alle lokalen Variablen und «Referenz»-Konstanten, die sich im Moment der Traversierung auf dem Stack befinden.
  2. Weiterverfolgen der Kette:
     - Innerhalb der Objekte allen Referenzen auf weitere Objekte folgen.

Vollautomatische Algorithmen: Mark-Sweep-GC

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

- Speicher wird nicht sofort freigegeben sondern erst bei «Bedarf».
- Suche nach Blöcken, die freigegeben werden können, in zwei Phasen:
  1. Mark:
     - Traversierung aller erreichbarer Objekte.
     - Alle erreichbaren Objekte werden markiert
  2. Sweep:
     - Sequentiell durch den Heap gehend, Speicher aller nicht markierten Objekte freigeben.
     - Die Markierung von markierten Objekte wird gelöscht.

\+ Vorteile:
- \+ Keine zusätzlichen Operationen bei Pointer-Zuweisungen nötig.
- \+ Zyklische Datenstrukturen können aufgelöst werden.

\- Nachteil:
- – Aufwand.
- – Das Programm muss während Mark-Sweep-Phase gestoppt werden (Stop-The-World-Mechanismus).
- – Es entstehen Löcher (Memory-Fragmentierung).

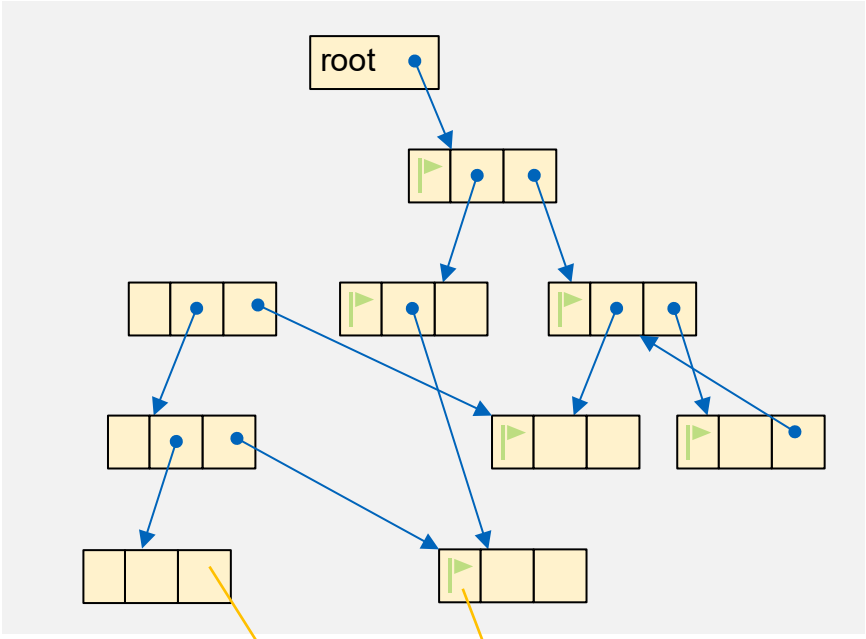
Vollautomatische Algorithmen: Mark-Sweep-GC

Mark-Sweep-Tiefensuche:

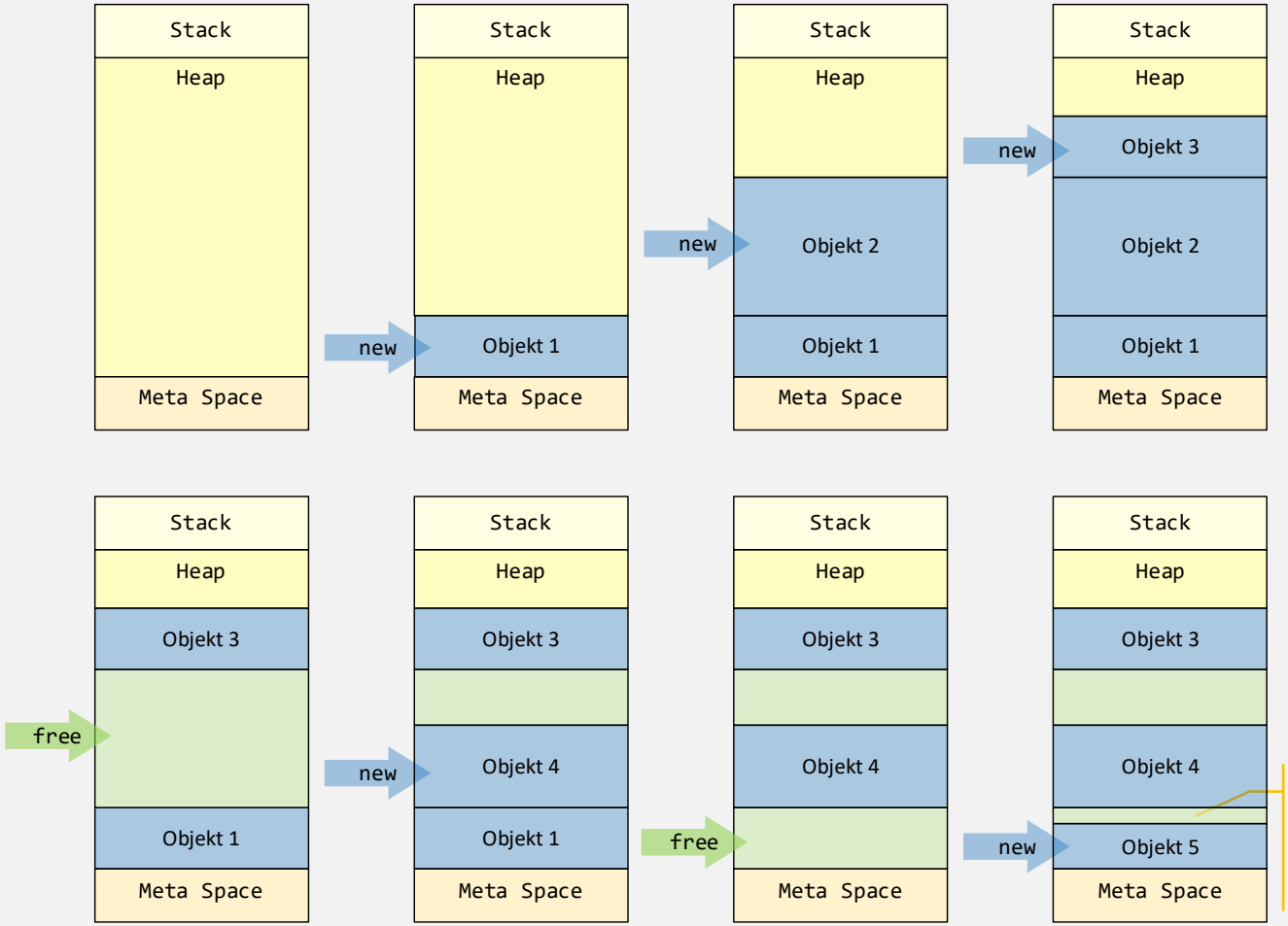
```
gc()
   for all n in root
      mark (n)
   sweep()
```

```
mark(N)
   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
```



Vollautomatische Algorithmen: Mark-Sweep-GC-Problem → Heap-Fragmentierung



Lücken im Heap, zu klein um wiederverwendet zu werden.

Vollautomatische Algorithmen: Mark-Compact

- Der Mark-Compact-Algorithmus hat den gleichen Markierungsprozess wie der Mark-Sweep-Algorithmus.
- Dieser Algorithmus bereinigt jedoch nicht direkt die Objekte, die im Garbage gesammelt werden können. Stattdessen verschiebt er alle referenzierten Objekte an den Anfang des Heaps, sodass keine Memory-Fragmentierung auftritt.

\+ Vorteile:
- \+ Analog Mark-Sweep-Algorithmus.
- \+ Keine Memory-Fragmentierung.

\- Nachteil:
- – Analog Mark-Sweep-Algorithmus.
- – Aufwand ist noch grösser → schlechte Performance.

Vollautomatische Algorithmen: Copying-GC

- Der Speicher wird in zwei gleiche Teile (Semi Spaces) aufgeteilt:
  - Der eine Semi-Space enthält die aktuellen Objekte.
  - Der andere Semi-Space enthält ausschliesslich obsolete Objekte.
- Neue Daten werden im aktuellen Semi-Space angelegt.
- Wenn kein Platz mehr im aktuellen Semi-Space:
  - Es werden alle noch referenzierten Objekte in den anderen Semi-Space kopiert. Nicht mehr referenzierte Objekte bleiben liegen.
  - Die Rollen der Semi-Spaces werden vertauscht.

\+ Vorteil:
- \+ es entstehen keine Löcher.
- \+ die Suche nach freien Blöcken entfällt (belegter Bereich ist kompakt).

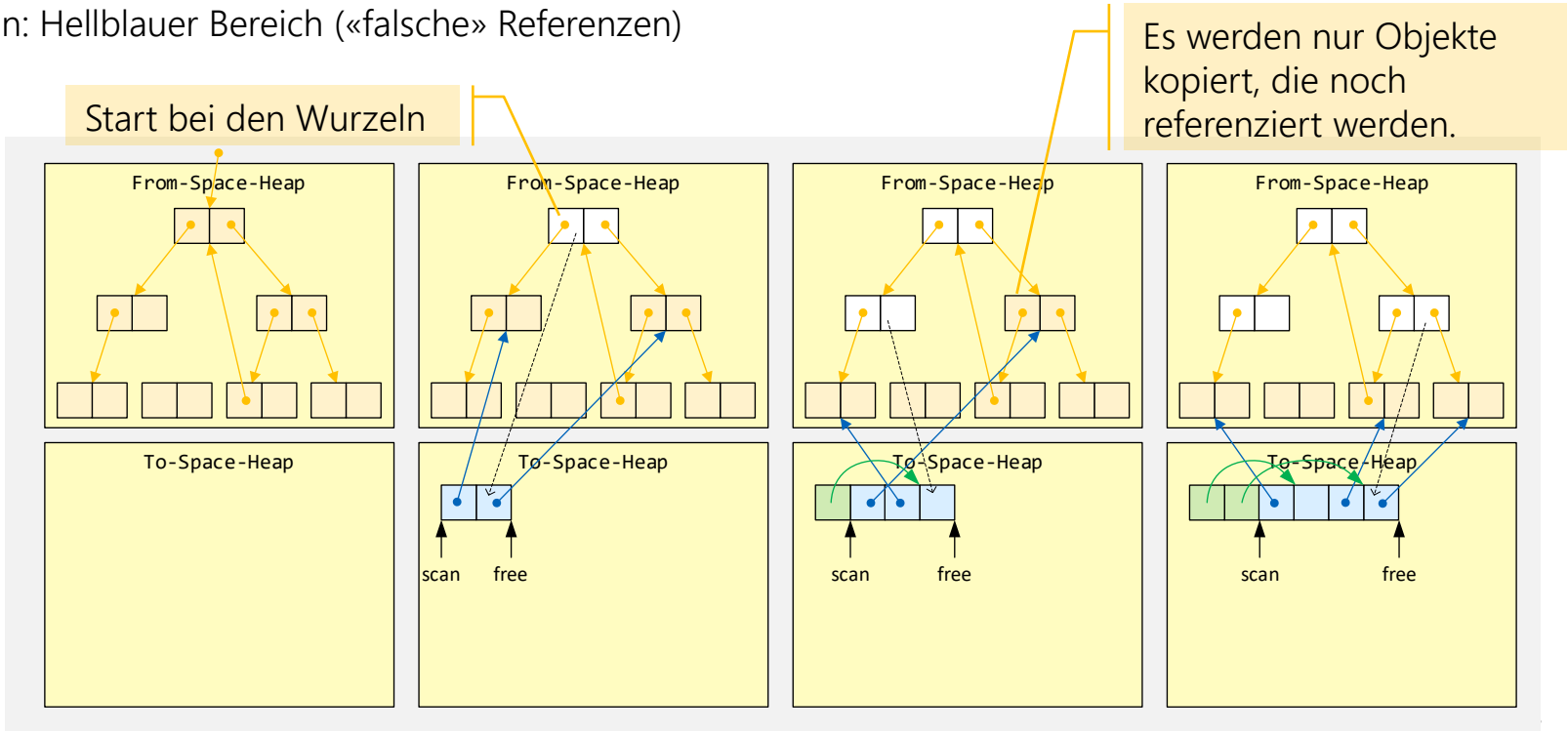
\- Nachteil:
- – Es wird doppelt so viel Speicher benötigt.
- – Aufwändiger Algorithmus.
- – Es muss immer der ganze Speicher durchlaufen werden.
- – Programm muss während dieser Zeit angehalten werden (Stop-The-World-Mechanismus).

Vollautomatische Algorithmen: Copying-GC

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Cheney's-Copying-Algorithm (1970): Breitensuche

- Drei Farben von Knoten:
  - Weiss: Kopierte Knoten im alten Semi-Space.
  - Blau: In neuen Semi-Space kopiert, Referenzen auf Nachfolger noch in den alten Semi-Space.
  - Grün: Kopiert mit korrekten Referenzen.
- Nur zwei Zeiger:
  - free: Freier Bereich
  - scan: Hellblauer Bereich («falsche» Referenzen)

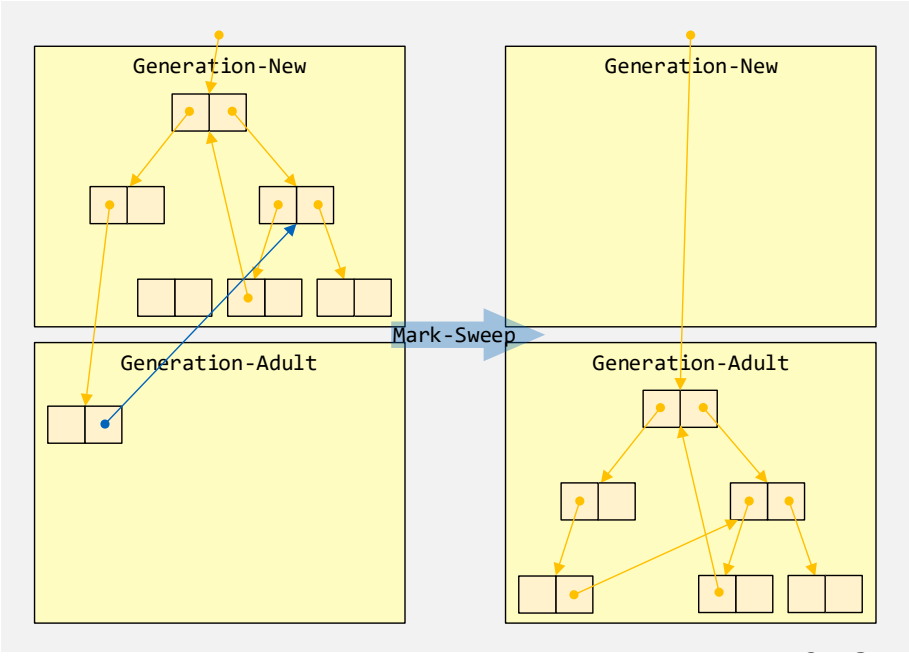


Vollautomatische Algorithmen: Generational GC

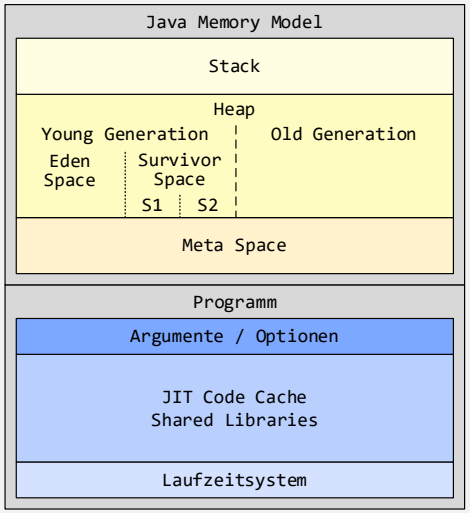
- Hypothese: Die meisten Objekte werden nur kurz gebraucht (z.B. das Iterator-Objekt).
- Idee: Die Objekte in (zwei, drei oder mehr) Generationen unterteilen. Neue Generationen werden häufiger nach freizugebenen Objekten durchsucht.
- Es handelt sich dabei nicht wirklich um einen neuen Algorithmus. Es können jetzt aber verschiedene Algorithmen zu unterschiedlichen Zeitpunkten je Generationstyp angewendet werden:
  - In der jungen Generation werden viele nicht referenzierte Objekte erwartet, der Coping-GC ist hier eine gute Lösung. Hierbei können die Objekte auch vom Bereich der jungen in den Bereich der alten Objekte kopiert werden.
  - In der alten Generation kann z.B. der Mark-Compact-Algorithmus eine gute Lösung sein.

Vollautomatische Algorithmen: Generational GC

- Ziel: Objekte im Generation-New-Space können unabhängig vom Generation-Adult-Space eingesammelt werden.
- Neue Objekte werden im Generation-New-Space angelegt.
- Alle noch referenzierten Elemente im Generation-New-Space werden z.B. mittels Copying-GC in den Generation-Adult-Space kopiert. Für die Referenzketten müssen die Referenzen des Generation-Adult- in den Generation-New-Space ebenfalls (als Root) beachtet werden.
- Für den Generation-Adult-Space muss z.B. ein Mark-Compact-GC eingesetzt werden.

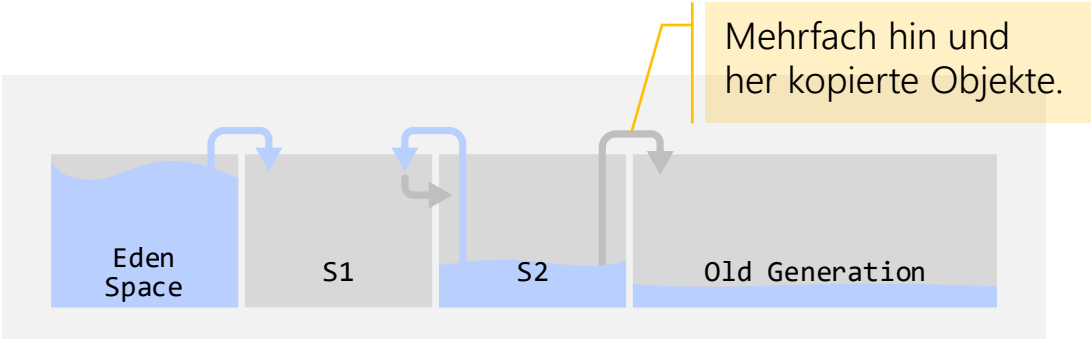


Generational GC in Java: Young Generation

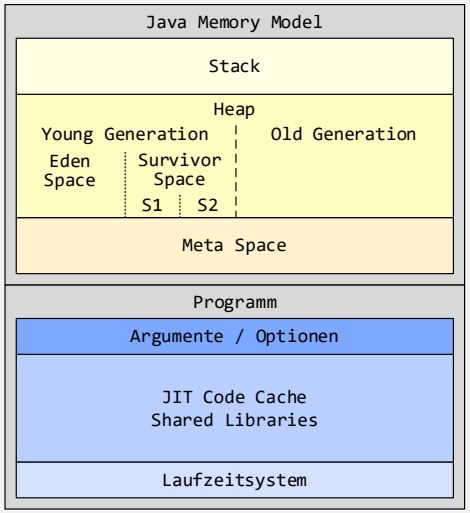


Vorsicht:
Die Darstellungen der Generationen geben die GCs Serial, Parallel und Garbage First (G1) der HotSpot VM wieder (siehe Folie 50).

- Neue Objekte werden im Eden-Space angelegt.
- Wenn der Young-Generation-Heap überläuft, wird ein z.B. der Mark-Compact-GC ausgeführt (abhängig vom konkreten GC). Dieser bereinigt den gesamten Bereich, oder Teile davon (Stop-the-world-Methode wird angewendet).
- Objekte im Eden-Space werden in den Survivor-Space kopiert. Objekte im S1- und S2-Bereich werden hin und her kopiert, dadurch wird Memory-Fragmentierung verhindert. Nach mehreren GC-Durchgängen wird ein Objekt aus dem Survivor- in den Old-Generation-Bereich kopiert.

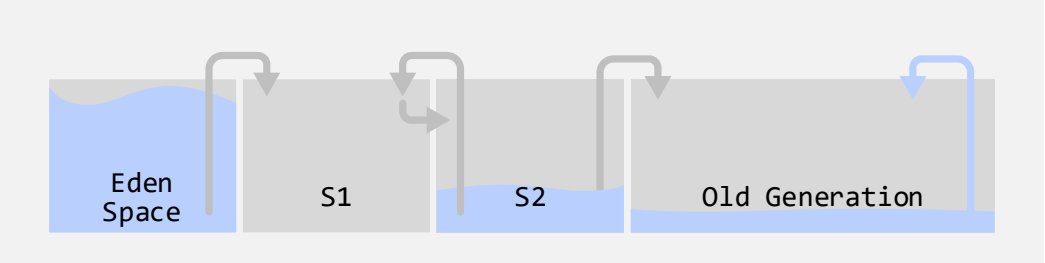


Generational GC in Java: Old Generation



Mit System.gc() kann die Garbage-Collection explizit ausgelöst werden.

- Der Old-Generation-Bereich ist grösser als der Young-Generation-Bereich, hat daher längere Laufzeit (Stop-the-world-Methode wird angewendet).
- Es wird z.B. der Mark-Compact-Algorithmus (abhängig vom konkreten GC) angewendet.
- Grosse Objekte werden direkt im Old-Generation-Bereich abgelegt. Vorsicht, falls diese häufig nur für kurze Zeit angelegt werden.
- Der Meta-Space wird unabhängig vom Heap bereinigt.



Tuning & Tools der Garbage Collection

GC in Java: Weak/Soft-References

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Anwendungsbeispiel von Weak/Soft-References:

- Datenstrukturen, die Sammlungen (Hastable, List usw.) von andern Objekten beinhalten, benötigen interne Referenzen auf diese Objekte. Diese werden von GC gefunden und traversiert.
- Problem:
  - Objekte in Sammlungen können nicht freigegeben werden, sobald (ausser Sammlung) keine Referenzen mehr auf diese Objekte vorhanden sind – obwohl diese eventuell nicht mehr benötigt werden.
  - Sehr viele Referenzen müssen traversiert werden.
- Lösung:
  - Einführung von Referenzen, die nicht traversiert werden: Weak-References.
  - Weak-Reference: Objekt wird gelöscht, wenn nur noch Weak-References auf dieses zeigen.
  - Soft-Reference: Objekt wird gelöscht, wenn nur noch Soft-References auf dieses zeigen und der Heap-Speicher knapp wird.

\+ Vorteil:
  + Objekte die nicht mehr benötigt werden, oder die bei Bedarf nachgeladen werden können, können freigegeben werden.

\- Nachteil
  – Problem mit Dangling-Pointer.
  – Falls das Objekt doch noch benötigt wird, muss dieses wieder erstellt werden (nachladen).

GC in Java: Weak/Soft-References

Klasse WeakReference in Package java.lang.ref

`WeakReference(T referent)`	Konstruktur, erzeugt Weak-Reference.
`void clear()`	Löscht die Weak-Referenz.
`T get()`	Gibt die Referenz des Objekts zurück.

```
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());
}
```

Output des Programms

```
Testumgebung@6acbcfc0
null

Process finished with exit code 0
```

GC in Java: Finalizer

ACHTUNG: Finalize ist seit Java Version 9 deprecated.

- Die Finalizer()-Methode (Teil der Object-Klasse) wird aufgerufen, bevor ein Objekt durch den Garbage-Collector gelöscht wird.
- Es gibt Ressourcen, die nicht automatisch verwaltet werden (Dateien, Fenster, usw. generell: Betriebssystem-Ressourcen). Diese müssen explizit wieder freigegeben werden, spätestens aber wenn das Objekt gelöscht wird:
  - Z.B. `BufferedReader br = new BufferedReader(new FileReader(path))` mittels `br.close()` wieder freigeben.
- Probleme bei der Anwendung von Finalizer:
  - Aufruf erst später (bei statischem 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).

GC in Java: Tuning

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Mit folgenden Einstellungen kann der Garbage-Collector beeinflusst werden. Die Punkte 4 bis 5 sollten nur im Ausnahmefall eingesetzt werden (Risiko von Artefakten).

Es gibt noch weitere Parameter.

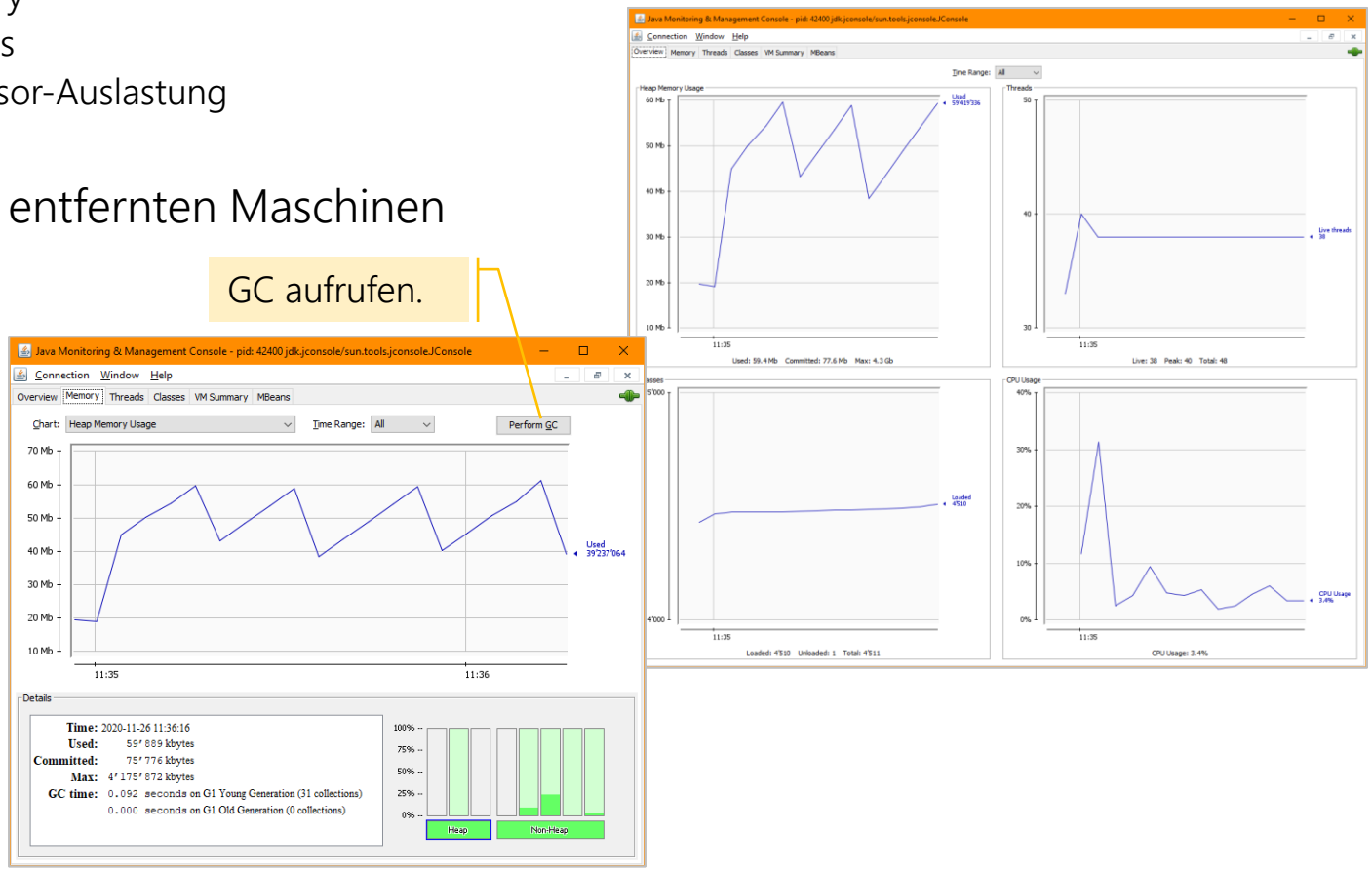
1. Festlegung der Grösse der vier Speicherbereiche im Heap.

von Oracle

2. Auswahl eines bestimmten Garbage-Collectors (Java HotSpot bietet 4 Varianten, es gibt noch weitere). Die VM wählt diesen normalerweise selbst aus:
   1. Serial GC: Single Thread, Generationen, Mark-Compact GC (Flag –XX:+UseSerialGC). Eignet sich z.B. für Microservices und Container mit wenig Speicher (100 MB).
   2. Parallel GC: Analog Serial GC aber mit multiple Threads (Flag -XX:+UseParallelGC). Falls Antwortzeit und Latenzzeit keine Rolle spielt, z.B. für Batchprozesse, oder falls Pausen von z.B. einer Sekunde akzeptabel sind.
   3. Garbage First (G1) GC [1N] (Empfohlen und in der Regel Default): Generationen, inkrementell, parallel, hauptsächlich concurrent (Flag -XX:+UseG1GC). Kompromiss zwischen Performance und Latenzzeiten.
   4. Z Garbage Collector (ZGC) [2N]: Ähnlich G1, keine Generationen, unterbricht die laufenden Threads höchstens wenige ms, allerdings auf Kosten eines gewissen Durchsatzes (Flag -XX:+UseZGC).
3. Expliziter Aufruf des Garbage-Collectors.
4. Verwendung von Weak/Soft-References.
5. Einsatz von Finalizer (deprecated).

GC in Java: Tools

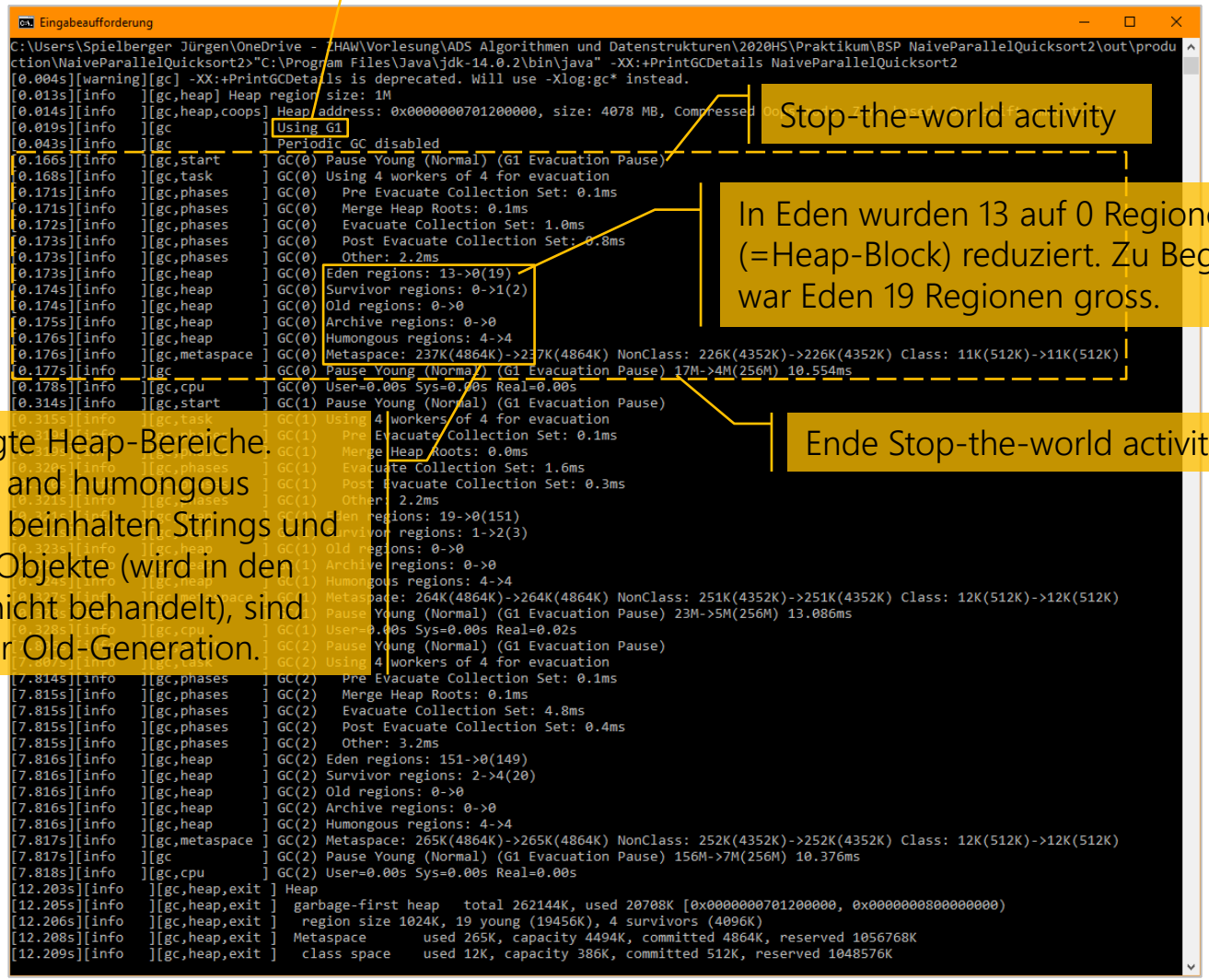
- JConsole - das Werkzeug zur Java Systemüberwachung:
  - jconsole.exe im <jdk>\bin Verzeichnis.
  - Zur Anzeige von:
    1. Memory
    2. Threads
    3. Prozessor-Auslastung
    4. ...
- Auch auf entfernten Maschinen möglich.



GC in Java: Tools

Verwendeter GC

Anzeigen von GC-Informationen: -XX:+PrintGC oder -XX:+PrintGCDetails



Zusammenfassung

- Optimierung durch Parallelisierung
  - Threadpools
  - Fork/Join
- Speicherverwaltung
  - Stack, Heap, Meta-Space
- Einfache Speicherverwaltung
- Automatische Speicherverwaltung
  - Einfache Verfahren
  - Automatische Verfahren
- Weak/Soft-References
- Finalizer
- Tuning des Garbage-Collector in Java
- Garbage-Collector-Tools