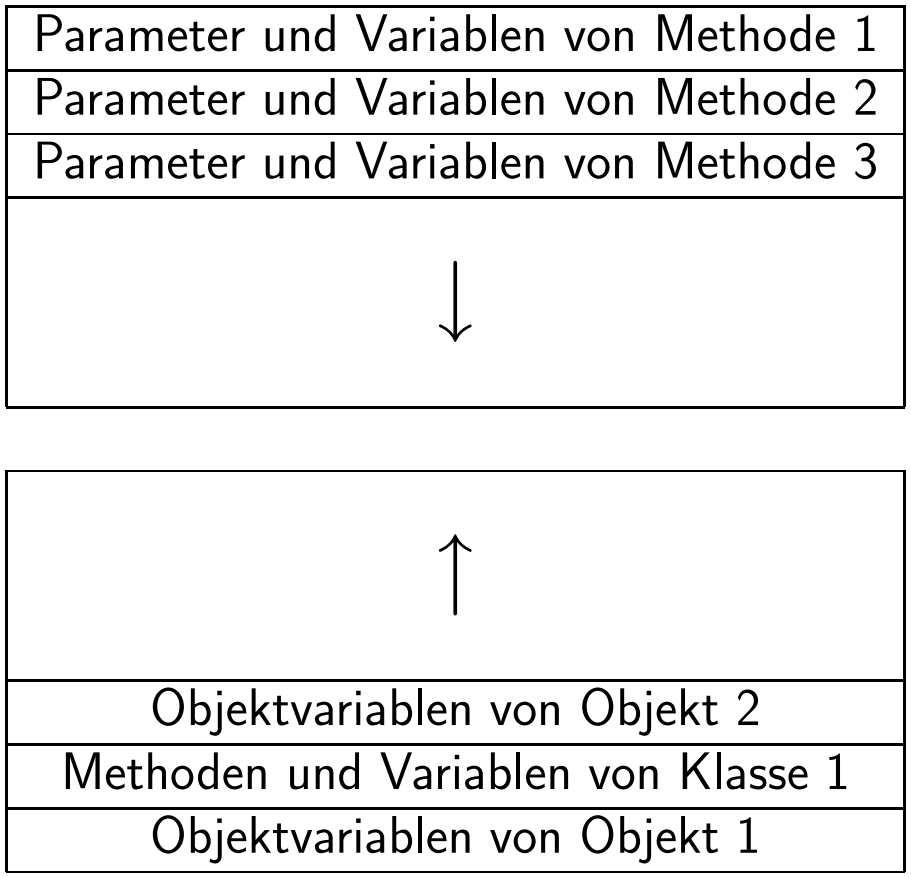
Vorsicht: Fallen

Speicherverwaltung

Speicherbereiche



Verwaltung der Speicherbereiche

Stack:

Speicherallokation bei Methodenaufruf

Speicherfreigabe bei Rückkehr aus Methode

Speicher bleibt durch Stack-Struktur stets kompakt

Heap:

Speicherallokation bei Objekterzeugung und beim Laden von Klassen

Speicherfreigabe durch **Garbage-Collection**

Schließen freier Löcher bei **Speicherkompaktierung**

Garbage-Collection

Garbage-Collector gibt Speicher für nicht mehr zugreifbare Objekte frei, sorgt oft auch für Speicherkompaktierung

Freigabe verzögert (je nach Speicherbedarf)

keine Freigabe wenn Objekt noch zugreifbar
(auch wenn kein Zugriff mehr erwartet wird)

nicht mehr benötigte Variablen auf `null` setzen: `x = null;`

Auf-`null`-setzen sinnvoll wenn

> Variable möglicherweise noch länger gültig ist
> und darauf sicher kein Zugriff mehr erfolgen wird

Fallen bei Garbage-Collection

gefühlt immer zum falschen Zeitpunkt
(effizient, aber manchmal merkbar längere Antwortzeiten)

Speicherverwaltung machtlos gegen schlechte Algorithmen

automatische Speicherverwaltung nicht überall sinnvoll
(z.B. sicherheitskritische Systeme)

Auf-`null`-setzen kann aufwendig sein

Eingriffe in Speicherverwaltung

`StackOverflowError` → `java -Xssn1m Prog` → meist erfolglos

Heapgröße ändern: `java -Xmsn6m -Xmxn66m Prog` (sinnvoll?)

Garbage-Collection explizit aufrufen:

```
Runtime r = Runtime.getRuntime();
r.gc();
```

Parameter der Gargabe-Collection einstellen (kompliziert)

vor Speicherfreigabe wird `finalize()` ausgeführt
(verzögert Speicherfreigabe, daher kaum verwendet)

Free-List umgeht Garbage-Collection

Aufgabe: Free-List versus Garbage-Collection

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Frage:

Wozu und wie kann man eine Free-List verwenden,
aber warum verzichtet man meist darauf?

Zeit: 2 Minuten

PK

Dateien und Co

Character-Streams auf Dateien

ungepuffert: `FileReader`, `FileWriter`
gepuffert: `BufferedReader`, `BufferedWriter`

Lesen mit `readLine` (ähnlich wie in Iterator)
Schreiben mit `write`

`String.format` mit variabler Argumentanzahl

Formatstring (z.B. `"%6d: %s\n"`) beschreibt Aussehen des Strings
je ein weiteres Argument für jedes % in Formatstring in String „eingefügt“

z.B.: `"%6d"` = 6-stellige ganze Zahl, die an dieser Stelle ausgegeben wird
z.B.: `"%s"` = beliebig langer String, der in den Formatstring eingefügt wird

Beispiele: gleiche Argumente

`new FileWriter(args[j], false)`: Überschreiben bestehender Datei

- `java Numbered2 a b c`: kopiert `a` numeriert nach `b` und `c`
- `java Numbered2 a b b`: kürzere Datei + Ende von längerer
- `java Numbered2 a a a`: leere Datei

`new FileWriter(args[j], true)`: Anhängen an bestehende Datei

- `java Numbered2 a b c`: kopiert `a` numeriert nach `b` und `c`
- `java Numbered2 a b b`: blockweise Überlappung
- `java Numbered2 a a a`: Endlosschleife

Fallen bei Dateien

Schließen vergessen wegen unüblicher Programmpfade

Zugriff auf Stream geht verloren

unterschiedliche Zeichen-Codierungen

selbe Datei unerwartet mehrfach geöffnet
(Lock-Dateien oder `FileLock` zur Vermeidung; `flush()`)

unterschiedliche Darstellung für Schreiben und Lesen