Validierung

Validierung von Daten

- alle von außen kommenden Daten auf *Plausibilität* prüfen (erscheinen zuverlässig genug um damit weiterzurechnen)
- klare Regeln nötig, was als plausibel gilt
- zu lockere und zu restriktive Kriterien vermeiden
- falsche Daten möglichst früh aussieben (beim Einlesen)
- zentrales Modul für Prüfungen sinnvoll
- „Never trust the user!“ (User $\neq$ Entwickler)
- Plausibilitätsprüfung (User) $\neq$ Zusicherung (Entwickler)

Validierung von Programmen

- Verifikation: Programm *richtig entwickelt*?
- Validierung: *richtiges Programm* entwickelt?
- Maßnahmen:
  - Gespräche mit künftigen Anwendern
  - Benutzeroberflächen-Prototypen
  - Mitarbeit künftiger Anwender
  - inkrementelle Entwicklungsmethoden
  - kurze Releasezyklen

Kräfte hinter Programm-Validierung

- Auftraggeber
- Anwender
- Softwareentwickler
- Marktwert und Kostenfaktoren

Validierung als wirtschaftlicher Begriff

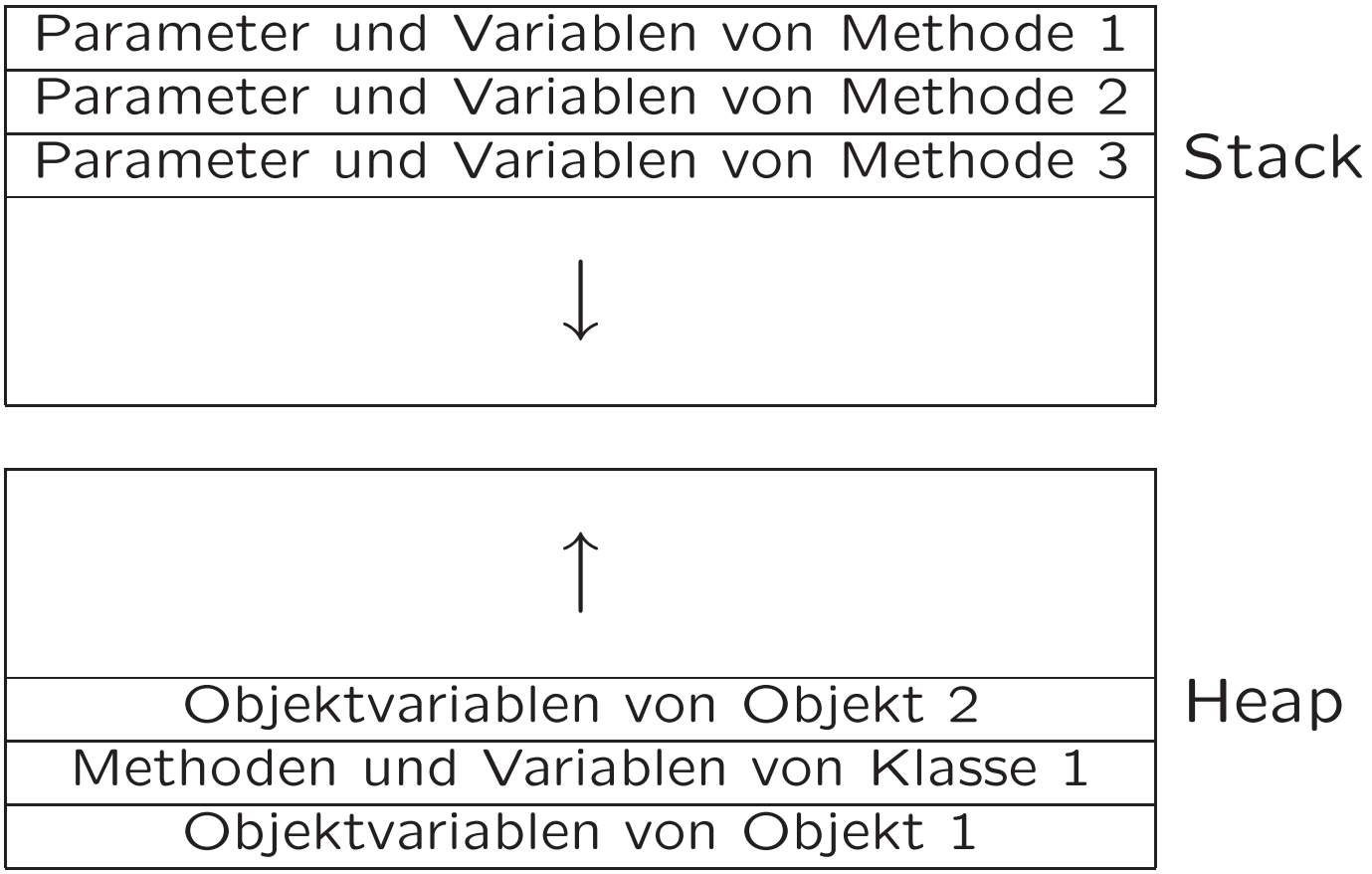
- Zahlt sich Weiterentwicklung aus?
- Wohin soll die Weiterentwicklung gehen?
- Welche unterstützenden Maßnahmen sind sinnvoll?

Manchmal: Validierung = Abnahmetest

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 frei Löcher bei *Speicherkompaktierung*

Garbage-Collection

- Garbage-Collector gibt Speicher für nicht mehr zugreifbare Objekte frei, sorgt oft auch für Speicherkompaktierung
- Freigabe verzögert (ja 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` $\Rightarrow$ `java -Xssn1m Prog` $\Rightarrow$ meist erfolglos
- Heapgröße ändern: `java -Xmsn6m -Xmxn66m Prog` (für Test)
- Garbage-Collection explizit aufrufen:

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

- Parameter der Gargabe-Collection einstellen (kompliziert)
- vor Speicherfreigabe wird `finalize()` ausgeführt
  (verzögert Speicherfreigabe, kaum verwendet)
- *Free-List* umgeht Garbage-Collection

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 `"%6d: %s\n"` beschreibt Ergebnis
  - je ein weiteres Argument für jedes '%' in Formatstring
  - z.B.: `"%6d"` = 6-stellige ganze Zahl
  - z.B.: `"%s"` = beliebig langer String

Beispiele: gleiche Argumente

`new FileWriter(args[j], false)`: überschreiben

- `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

- `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