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 Validierung

- Auftraggeber
- Anwender
- 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!

Skriptum: Teil 2

Beschränkte Ressourcen

Speicherverwaltung

Garbage Collection

- Garbage Collector gibt Speicher für nicht mehr zugreifbare Objekte automatisch wieder frei
- 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

Beschränkte Ressourcen

Dateien und Co

Beispiel: Datei mit Zeilennummern

```
import java.io.*;
public class Numbered {
    private static final String errmsg =
                          "Usage: java Numbered <in> <out>";
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println(errmsg);
            return;
        }
        ...  // eigentlicher Programmcode
    }
}
```

```
try {  // in main in Numbered (continued) ...
    BufferedReader in = null; BufferedWriter out = null;
    try {
        String line;
        in = new BufferedReader(new FileReader(args[0]));
        out = new BufferedWriter(new FileWriter(args[1]));
        for (int i=1; (line=in.readLine()) != null; i++)
            out.write(String.format("%6d: %s\n", i, line));
    }
    finally {
        if (in != null) in.close();
        if (out != null) out.close();
    }
}
catch (IOException ex) {
    System.err.println("I/O Error: " + ex.getMessage());
}
```

Character Streams auf Dateien

- gepuffert: `FileReader`, `FileWriter`
- ungepuffert: `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

Beispiel: 2 Dateien mit Zeilennummern

```
import java.io.*;
public class Numbered2 {
    private static final String errmsg =
                "Usage: java Numbered <in> <out1> <out2>";
    public static void main(String[] args) {
        if (args.length != 3) {
            System.err.println(errmsg);
            return;
        }
        ...  // eigentlicher Programmcode
    }
}
```

```
try { BufferedReader in=null;
      BufferedWriter out1=null, out2=null;
    try { String line;
        in = new BufferedReader(new FileReader(args[0]));
        out1 = new BufferedWriter(new FileWriter(args[1]));
        out2 = new BufferedWriter(new FileWriter(args[2]));
        for (int i=1; (line=in.readLine()) != null; i++) {
           out1.write(String.format("%6d: %s\n", i, line));
           out2.write(String.format("%4d: %s\n", i, line));
        }
    }
    finally { if (in != null) in.close();
              if (out1 != null) out1.close();
              if (out2 != null) out2.close();
    }
}
catch (IOException ex) {
    System.err.println("I/O Error: " + ex.getMessage());
}
```

Beispiele: gleiche Argumente

- `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 statt überschreiben

- `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 Ressourcen geht verloren
- unterschiedliche Zeichen-Codierungen
- selbe Datei unerwartet mehrfach geöffnet
  (Lock-Dateien, `FileLock`, `flush()`)
- unterschiedliche Darstellung für Schreiben und Lesen