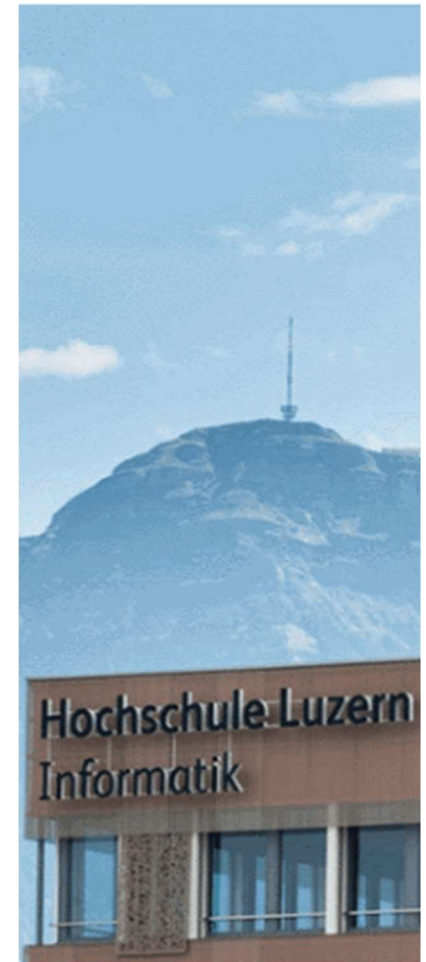


Algorithmen und Datenstrukturen

# **Datenstrukturen: Tipps für die Praxis (Java)**

Roland Gisler



# Inhalt

- Datenstrukturen und Nebenläufigkeit.
- Ergänzende Hinweise zu `equals()` und `hashCode()`.
- Minimieren Sie die Mutierbarkeit (Mutability).
- Nutzen Sie wenn immer möglich generische Collections.
- Collections den Arrays vorziehen.
- Datenstrukturen von Java präferieren.
- Alternative Datenstrukturen (Thirdparties).

# Lernziele

- Achtsame Auswahl der geeigneten Datenstrukturen.
- Verschiedene Tipps und Hinweise zum Umgang mit Datenstrukturen kennen.
- Vermeiden von typischen Programmierfehlern im Zusammenhang mit Datenstrukturen bei Java.
- Ausgewählte Hinweise aus Effective Java von Bloch kennen.
- Alternativen (Thirdparties) kennen und beurteilen können.

# **Datenstrukturen und Nebenläufigkeit**

## «Veraltete» Implementationen

- Es gibt immer wieder Hinweise, dass man z.B. folgende Klassen aus dem Java Collections Framework nicht mehr verwenden soll:
  - `java.util.Stack`
  - `java.util.Vector`
  - `java.util.HashTable`
- Tatsächlich existieren diese Klassen schon seit dem JDK 1.0 (also der ersten Version von Java) und sind darum tatsächlich alt.
- Aber das Alter ist **nicht** das Problem: All diesen Datenstrukturen ist gemeinsam, dass Sie deutlich **langsamer** sind, **weil** sie (im Gegensatz zu ihren «modernen» Pendanten) **synchronisiert** (thread safe, **synchronized**) sind!
  - Das ist somit nicht per se schlecht, sondern eine bewusste Entscheidung: Unnötige Synchronisation kostet (Lauf-)Zeit!

# Collections sind nicht thread safe implementiert

- Die aktuellen (jüngeren) Implementationen des Java Collection Frameworks sind mehrheitlich auf **maximale** Geschwindigkeit optimiert und somit in der Regel **nicht** synchronisiert.
- Wenn Sie nebenläufig programmieren und dafür eine **synchronisierte** Datenstruktur benötigen, können Sie über die Collections-Klasse für **jede** Implementation eine synchronisierte Variante erhalten!
  - Original-Implementation wird dazu «gewrappt».
  - Gelöst per Proxy-Pattern, mehr dazu im Modul VSK.
- Beispiel:

```
final List<Object> syncList =  
    Collections.synchronizedList(new ArrayList<>());  
syncList.add(new Object()); // Synchronisiert!
```

# Synchronized ist ungleich Concurrent

- Neben synchronisierten Datenstrukturen gibt es auch noch neuere Implementationen, welche einen **Concurrent**-Prefix im Namen tragen. Beispiele:
  - `java.util.concurrent.ConcurrentMap`
  - `java.util.concurrent.ConcurrentLinkedDeque`
  - `java.util.concurrent.ConcurrentSkipListSet`
- Diese Datenstrukturen sind so geschickt implementiert, dass sie mit ausgewählten (atomaren) Operationen parallele Operationen erlauben und trotzdem thread safe sind.
- Es gibt aber auch Thirdparties in diesem Bereich, als Beispiel: JCTools - <https://github.com/JCTools/JCTools>
- Mehr Input zum Thema erhalten Sie im Teil ➔ **Nebenläufigkeit!**

## **Ergänzende Hinweise zu equals()**



## Hinweise zur Implementation von equals()

- Reflexivität und Transitivität kann sehr leicht verletzt werden, wenn man `equals()` in einer Vererbungshierarchie überschreibt!
- Beispiel:
  - `Point(x,y)` spezialisiert zu `ColorPoint(x,y,color)`
  - `point.equals(colorPoint)`: Kennt keine Farbe!
  - `colorPoint.equals(point)`: `Point` ist **kein** `ColorPoint`!
- Wird bei einer Value-Klasse in einer Spezialisierung ein relevantes Attribut ergänzt, kann der equals-Kontrakt nicht eingehalten werden!
  - In diesem Fall nicht an `equals()` «basteln», sondern die Typhierarchie bzw. Vererbung hinterfragen.
- Die Methode `hashCode()` unbedingt **konsistent** überschreiben!

# Empfehlungen für gute equals()-Methoden

- Effective Java - Methods Common to all Objects:  
Item 8 - «Obey the general contract when overriding equals.»
- Fertig implementierte `equals()`-Methode kritisch prüfen, finalisieren und unbedingt mit Unit-Tests verifizieren.
  - **final**: Methode kann nicht mehr überschrieben werden.
  - somit pro Vererbungspfad nur maximal eine Implementation.
  - EqualsVerifier (<http://jqno.nl/equalsverifier/>) nutzen!
- Gleichheit möglichst auf Schlüssel-Eigenschaften beschränken
  - besser Performance (Vergleiche: «deep equals»)
- Die überschriebene `equals()`-Methode (bzw. generell jede überschriebene Methode) mit `@Override` annotieren!
- Bei Objekt-Referenzen `equals()` in der Regel weiter delegieren.

## **Ergänzende Hinweise zu hashCode()**

## Hinweise zur Implementation von hashCode()

- Wichtigster Grundsatz: Sind zwei Objekte gemäss `equals()`-Methode **gleich**, müssen Sie auch den **selben Hashwert** liefern.
  - Beide Methoden sollten somit auf den gleichen Attributen basieren.
- Sind Objekte **nicht gleich**, sollten sie **möglichst verschiedene** Hashwerte liefern, müssen aber nicht (keine perfekten Hashes).
- Performance von hash-basierenden Datenstrukturen ist direkt abhängig von der Qualität der `hashCode()`-Implementation!
- Bei wiederholten Aufrufen von `hashCode()` darf sich der hash-Wert **nicht** verändern!

# Hashwerte von elementaren Datentypen

- Grundsätzlich werden von allen Wrapperklassen der elementaren Typen statische Hilfsmethoden zur Berechnung der Hashes zur Verfügung gestellt, die man nutzen sollte. Beispiel:

```
int hash = Double.hashCode(doubleValue);
```

- Alternativ, bei elementaren Ganzzahltypen (byte, short,...):

```
int hash = (int) value;
```

- Alternativ, bei boolschen Werten:

```
int hash = (value ? 1 : 0);
```

- Die einzelnen Werte werden aufsummiert, die Zwischensummen jeweils mit einer Primzahl multipliziert.

# Empfehlungen für gute hashCode()-Methoden

- Effective Java - Methods Common to all Objects:  
Item 9 – «Always override hashCode when you override equals.»
- hashCode-Methode testen!
  - Implizit: mit EqualsVerifier-Tool
  - Explizit: Hashwerte von ausgewählten, typischen Objekten kritisch überprüfen!
- Implementation in der Art «**return 1;**» ist definitiv **keine** Option.
- Bei → **Immutable Objects** kann der Hashwert gecached werden, und muss somit nur bei der Erzeugung des Objektes berechnet werden.
- Nie versuchen **zu** genial zu sein!  
Donald E. Knuth: Optimieren Sie nicht!

# **Minimiere die Mutierbarkeit** (Immutable Objects)

# Minimiere die Mutierbarkeit von Objekten

- Effective Java - Classes and Interfaces:  
Item 15 – «Minimize mutability.»
- Unveränderbarkeit: Die Attribute (Zustand) eines Objektes lassen sich nicht mehr verändern, weil sie **finalisiert** sind.
  - Attribute verbleiben während der gesamten Lebensdauer im Zustand ihrer Erzeugung.
- Paradebeispiele aus den Java Library Klassen:  
**String, Integer, Long** etc.
  - Besonders gut eignen sich also eher kleine Value-Klassen.
- Daraus ergibt sich eine Optimierungsmöglichkeit: Hashwert von Immutable Objects muss nur einmal (bei Erzeugung) berechnet werden, und bleibt dann konstant → schnell!



# Eigenschaften einer unveränderbaren Klasse

- **Keine** Methoden welche den Zustand des Objektes verändern.
  - somit im Besonderen auch **keine** Setter-Methoden.
- Eine Spezialisierung der Klasse **muss** verhindert werden.
  - Darum werden immutable Klassen immer **final** deklarieren.
- Sämtliche Attribute sind **private** und ebenfalls **final** deklariert.
  - Attributwerte können nur einmalig bei der Erzeugung des Objektes gesetzt werden.
- Exklusiven Zugriff auf mutierbare Komponenten.
  - Enthält die Klasse (mutierbare) Objekte, darf der Zugriff darauf nur über explizite Getter-Methoden erfolgen
  - keinesfalls darf eine Referenz auf enthaltene, mutierbare Objekte nach aussen gelangen (➔ information hiding)

# Beispiel einer unveränderlichen Klasse - Point

```
public final class Point

    private final int x;
    private final int y;

    public Point(final int x, final int y) {
        this.x = x; this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }
}
```

<<immutable>> Point
- x : int - y : int
+ Point(x : int, y : int) + getX() : int + getY() : int

```
@Test
public void testImmutability() {
    assertImmutable(Point.class);
}
```

# Vorteile von unveränderbaren Objekten

- Immutable Object sind implizit thread safe
  - sie benötigen somit keine Synchronisation!
  - ➔ Ideal für nebenläufige Implementationen.
- Können beliebig frei ver- und geteilt werden.
  - Unproblematische Verwendung innerhalb von Collections.
  - Effizientere Nutzung des Speichers, schneller durch sharing.
- Gut als Attribut-Typen für andere (komplexere) Objekte geeignet
  - Komplexere Objekte werden durch Aggregation und Komposition zusammen gesetzt.
  - Weitergabe von Referenzen ist ungefährlich.
- Immutabilität lässt sich z.B. mit Mutability Detector prüfen
  - <https://github.com/MutabilityDetector/MutabilityDetector>

## Empfehlung – Immutable Klassen

- Betrachten Sie nicht jede Klasse / jedes Attribut das Sie entwerfen als zwingen mutierbar!
    - Finalisieren (**final**) Sie alles was sie können.
  - Versuchen Sie auch den folgenden Ansatz:  
Solange es keine guten Gründe für eine Veränderbarkeit gibt, implementieren Sie jede Klasse als Immutable!
  - Passen Sie auf, wenn sie über getter-Methoden Objekt-Referenzen zur Verfügung stellen: Diese Objekte dürfen die Immutable-Semantik nicht durchbrechen.
    - Im Notfall: Kopie von Objekten zurückliefern, so dass allfällig Änderungen darauf keinen Impact auf die Quelle haben.
- ➔ Das gilt auch für Collections!

## Immutable bzw. Unmodifiable Collections

- Die Collections-Klasse bietet Methoden an, um von existierenden Collections unmodifizierbare Varianten zu erhalten!
- Analog zu den bereits erwähnten **synchronizedXxx**-Methoden, wird dabei für eine bestehende Liste ein Proxy erzeugt, das **keine** Modifikationen mehr an der Liste zulässt.
- Beispiel:

```
List<String> demo = new ArrayList<String>();  
List<String> unmodifiableList =  
    Collections.unmodifiableList(demo);  
unmodifiableList.add("Try");           // Laufzeitfehler*!
```

- Dadurch kann auch eine Liste von (immutable!) Objects als quasi immutable geschützt werden.

**Leere Collections, nicht null**

## Rückgabe von null-Objects

- Effective Java – Methods:  
Item 43 – «Return empty arrays or collections, not nulls.»
- Viel zu oft wird bei einer **leeren Menge** anstelle von leeren Collections (oder Arrays) eine **null**-Referenz zurückgegeben:

```
private final List<Wine> wineInStock = ...;  
  
public List<Wine> getWines() {  
    if (wineInStock.size() <= 0) {  
        return null;  
    }  
    ...  
}
```



- Das ist eine **sehr schlechte** Lösung!

## null-Werte müssen immer explizit behandelt werden

- Es gibt keinen Grund eine leere Collection (oder Array) speziell zu behandeln!
- Die explizite Rückgabe von `null`-Referenzen provoziert im Client nicht selten sogar noch mehr Aufwand:

```
final List<Wine> wines = shop.getWines();  
if (wines != null) {  
    for (final Wine wine : wines) {  
        System.out.println(wine.toString());  
    }  
}
```



- Hässlich und fehleranfällig:  
Die **null**-Referenz muss explizit abgefangen/geprüft werden!



## Rückgabe von null-Werten ist fehleranfälliger

- Die Rückgabe von `null`-Referenzen im (seltenen) Ausnahmefall einer leeren Menge ist zusätzlich fehleranfällig: Weil es selten ist, geht mitunter die notwendige `null`-Prüfung häufig vergessen!
- Als Begründung für `null`-Referenzen wird auch häufig erwähnt:  
«Leere Collection bzw. Array muss extra erzeugt werden!  
Ein `null`-Wert kostet hingegen nichts.»
- ➔ Das ist aber ein **sehr schlechtes** Argument
  - Performance-Optimierung an einer völlig unnötigen Stelle.
  - Es gibt Hilfsmethoden, um schnell leere Collections zu erhalten.

## Empfehlung – Empty Collections

- Geben Sie für leere Mengen **nie** `null` zurück, sondern immer eine **leere Collection**.
- In der Regel vereinfacht das auch den Code beim Aufrufer, weil es keinen Spezialfall mehr gibt: Eine Iteration über eine leere Collection ist **zulässig** und macht einfach «nichts».
- Über statische Hilfsmethoden der Klasse **Collections** können wir sogar direkt leere (sind gleichzeitig unmodifiable) Collections erhalten.
- Beispiele:

```
... = Collections.emptyList()  
... = Collections.emptySet()  
... = Collections.emptyMap()
```

# **Generische Datenstrukturen (ohne raw-Types) verwenden**

# Generische Klassen

- Mit Java 1.5 wurden Generics eingeführt. Damit haben viele Klassen einen (oder mehrere) Typ-Parameter erhalten.
- Aus der Kombination einer generisch implementierten Klasse und einem aktuellen Typs ergibt sich ein parametrisierter Typ.
  - **List<E>** – Generischer Typ mit Typparameter **E**.
  - **String** – Beispiel eines aktuellen Typ(-parameter).
  - **List<String>** - Parameterisierter Typ ‚List of String‘.
- Beispiel:

```
List<String> myList = new ArrayList<>();
```

- Hauptvorteil: Dadurch erhalten wir typsichere Klassen/Methoden und müssen deutlich seltener Typen prüfen und/oder casten.

# Keine raw-Typen mehr verwenden

- Effective Java – Generics:  
Item 23 – «Don't use raw types in new code!»
- Aus technischen Gründen (Rückwärtskompatibilität) sind die so genannten raw-Implementationen noch immer vorhanden.  
Beispiel:

```
List myRawList = new ArrayList();
```



➔ Diese raw-Typen sollte man **nicht mehr verwenden!**

- Selbst wenn man wirklich nur Objekte vom Typ `Object` verwenden will, sollte man diese dennoch typisieren:

```
List<Object> myObjects = new ArrayList<>();
```




## Warum raw-Types schlecht sind

- Was ist der Unterschied zwischen einer raw-List und `List<Object>`?
- Es gibt in Java explizite Regeln was bei Generics ein Subtype (Spezialisierung) ist und was nicht:  
Eine `List<String>` ist zwar ein Subtype von `List`, aber  
Eine `List<String>` ist **kein** Subtype von `List<Object>`.
- Man verliert durch die Verwendung von raw-Types also deutlich mehr Typsicherheit als man denkt.
- Besonders wichtig beim Design von Schnittstellen!

## Schlechtes Beispiel: Verwendung des raw-Types

```
public static void main(final String[] args) {  
    final List<String> strings = new ArrayList<>();  
    unsafeAdd(strings, new Integer(42));  
    final String string = strings.get(0);  
}  
  
static void unsafeAdd(List list, Object object) {  
    list.add(o);  
}
```



- Was passiert hier?

Das Programm **kompiliert**, aber es gibt einen **Laufzeitfehler**:

Exception in thread "main" java.lang.ClassCastException:  
java.lang.Integer cannot be cast to java.lang.String

## Gutes Beispiel: Parameterisierter Typ (Object)

```
public static void main(final String[] args) {  
    final List<String> strings = new ArrayList<>();  
    unsafeAdd(strings, new Integer(42));  
    final String s = strings.get(0);  
}  
  
static void unsafeAdd(List<Object> list, Object o) {  
    list.add(o);  
}
```

- Nun haben wir einen **Kompilerfehler** (besser):

unsafeAdd(List<Object>, Object) is not applicable for the arguments (List<String>, Integer)

- Konsequenter Einsatz von Generics macht unseren Code also viel besser und sicherer!





## Empfehlungen zu Generics

- Nutzen Sie so oft wie möglich generische Typen und profitieren Sie von der damit gegebenen Typsicherheit!
- Achten Sie speziell bei Schnittstellen (die ggf. Teil einer API sind) auf gut gewählte (generische) Typen.
- Vermeiden Sie wo immer möglich die Verwendung von raw-Types.
- Versuchen Sie den bei Generics ab und zu auftretenden Warnings (Compiler oder IDE) hartnäckig auf den Grund zu gehen und das Problem wirklich zu lösen.
  - Unterdrücken von Warnings (**@SupressWarnings**) nur in Ausnahmefällen, bewusst, dokumentiert und auf dem kleinsten möglichen Scope!

# **Präferiere Collections vor Arrays**

# Generische Listen sind besser als Arrays

- Effective Java – Generics:  
Item 25 – «Prefer lists to arrays.»
- Arrays und (generischen) Listen unterscheiden sich in **zwei** wichtigen Punkten:
  - Arrays sind **kovariant**, Listen sind **invariant**.
  - Arrays werden **reified**, Listen nutzen **erasure**.
- Aufgrund dieser zwei wesentlichen Unterschiede interagieren Arrays und Listen **nicht besonders gut** miteinander!
- Aber was heisst das nun genau?

## Kovarianz und Invarianz

- **Generische Klassen** sind **invariant**, weil:

Eine `List<String>` ist **kein** Subtype von `List<Object>`!

```
// Kompilerfehler:  
List<Object> objects = new ArrayList<Long>();
```

- **Arrays** hingegen sind **kovariant**:

Ein `String[]`-Array **ist** ein Subtype von `Object[]`!

```
Object[] array = new Long[1];    // Kompiliert!  
array[0] = "Das passt nicht!«;  // Laufzeitfehler!
```

## Reify und erasure

- Arrays werden reified: Arrays kennen zur Laufzeit ihren Typ und können diesen somit auch zur Laufzeit prüfen.
  - Es resultieren somit ggf. Laufzeitfehler.
- Generics hingegen sind mit type erasure implementiert: Generics kennen zwar ihren Typ zur **Kompilierzeit**, aber für die Laufzeit wird die Information gelöscht.
  - Wegen Rückwärtskompatibilität mit raw-Types.
  - Dadurch aber auch schneller.
- Diese (vermeintliche) Unsicherheit bei den Generics wurde behoben und mehr als egalisiert, in dem der Compiler bei der Typprüfung von Generics **viel strenger** und aufmerksamer ist!
  - Fehlerfreier, typsicherer Code **ohne** Laufzeitprüfung.

# Es gibt keine generischen Arrays in Java!

- Weil **Arrays** und **Generics** so **unterschiedlich** implementiert sind, gibt es in Java auch **keine** generischen Arrays!
- Tatsächlich ist **keiner** der folgenden Ausdrücke erlaubt:

```
... = new List<E>[];  
... = new List<String>[];  
... = new E[];
```



- Arrays sind zur Kompilierzeit **nicht** typsicher, was von den generischen Typen aber verlangt (und auch **garantiert**) wird.
  - Somit könnte bei generischen Arraytypen zur Laufzeit eine **ClassCastException** auftreten → das ist aber nicht erlaubt.
  - Konsequenz: Arrays gibt es nicht generisch!

## Empfehlung – Collections den Arrays vorziehen

- Arrays sind (wie schon im Modul OOP empfohlen) für eher kleine Datenmengen von elementaren Datentypen gut und effizient.
  - Auch als «interne» Datenstrukturen (➔ information hiding) kann man sie (wie bei unseren exemplarischen Implementationen von Datenstrukturen) gut verwenden.
- Aber: Auf **Interfaces** oder gar auf **API**-Ebene sollte man möglichst auf Arrays verzichten! Sie sind mit den generischen Typen nicht wirklich gut verträglich!
- Und: Für dynamische, typisierte Objektmengen verwenden wird ohnehin mit Vorteil die Collections aus dem Java Collection Framework!
  - Zum Beispiel weil sie mehrheitlich dynamisch sind.

# **Thirdparty - Datenstrukturen**



## Thirdparty - Datenstrukturen

- In der aktuellen Version ist Java mit sehr leistungsfähigen Datenstrukturen für vielfältige Zwecke bestückt.
- Darum gilt: Verwenden Sie wenn immer möglich bestehende Datenstrukturen aus dem Java Collection Framework.
  - Intensiv getestet, erprobt und performant.
- Trotzdem kann es manchmal Spezialfälle geben, wenn man sehr spezielle Anforderungen hat, zum Beispiel:
  - extrem grosse Datenmengen.
  - stark dynamische Datenmengen mit elementaren Datentypen.
  - verteilte Daten.
- In diesen Fällen lohnt sich auch mal ein Blick über den Tellerrand des Java Collections Frameworks hinaus!

# Beispiele von Thirdparty-Datenstrukturen Libraries

Hier eine kleine Auswahl an Projekten:

- Eclipse Collections (ehemals: Goldman-Sachs (GS-) Collections)  
<https://www.eclipse.org/collections/>
  - Kompakte, aussagestarke API mit vereinfachenden Funktionen.
  - Verspricht weniger Speicher zu benötigen.
- HPPC – High Performance Primitive Collections  
<http://labs.carrotsearch.com/hppc.html>
  - Grosses Set an Collections für elementare Datentypen.
  - Direkte Konkurrenz zu Arrays, aber dynamisch und grösser.
- Google Guava – Google Core Libraries for Java  
<https://github.com/google/guava>
  - Umfangreiche Library, Collections sind nur ein Teil davon.

## Empfehlungen – Thirdparty Collection Libraries

- Jede Thirdparty-Library ist auch ein Risiko: Man wird/ist davon abhängig. Da Collections in einer Software eine ziemlich zentrale Rolle einnehmen, ist hier besondere Vorsicht angebracht.
- Erste Wahl sollten immer die in Java bereits enthaltenen Collections sein.
- Nur bei echten Problemen bzw. ganz spezifischen Anforderungen sollte man Thirdparty-Collections in Auge fassen.
- Unabdingbar ist dann aber ein Profiling (Zeit- und Speicher-messungen) mit realen Datenmengen und Typen, um einen Nutzen wirklich erkennen zu können.

## **Ein (kurzer) Blick zurück**

## **Datenstrukturen – Ein Blick zurück**

- Arrays, Listen (einfach oder doppelt Verknüpft), Bäume in diversen Spezialisierungen (z.B. binärer Suchbaum), Hashtabellen, ...
- Ein bescheidener Einblick in mögliche Implementierungen all dieser Datenstrukturen, verbunden mit den jeweiligen Vor- und Nachteilen.
- Denken Sie daran: Die Implementationen von Listen, Stacks und Queues in den letzten Wochen waren exemplarisch zu Lernzwecken gedacht.
- In der Praxis verwenden Sie so oft wie möglich bereits vorhandene und bewährte Datenstrukturen.

# Zusammenfassung

- Datenstrukturen sollen stets achtsam und unter Berücksichtigung aller Anforderungen ausgewählt werden.
- Die **Collections**-Klasse stellt viele hilfreiche Methoden zur Verfügung um vorhandene Implementationen zu modifizieren.
  - Wrapper für synchronisierte und unmodifizierbare Collections.
  - Helper-Methoden für leere Collections.
- Tipps und Hinweise zum Umgang mit Datenstrukturen: Immutable Objects, `equals()` und `hashCode()`, etc.
- Gegenüberstellung: Generics versus Arrays
  - Covarianz vs. Invarianz, Laufzeit- vs. Kompiler-Fehler
- Ausgewählte Hinweise aus Effective Java von Bloch.
- Alternativen (Thirdparties) kennen und beurteilen können.

# Literaturtipp

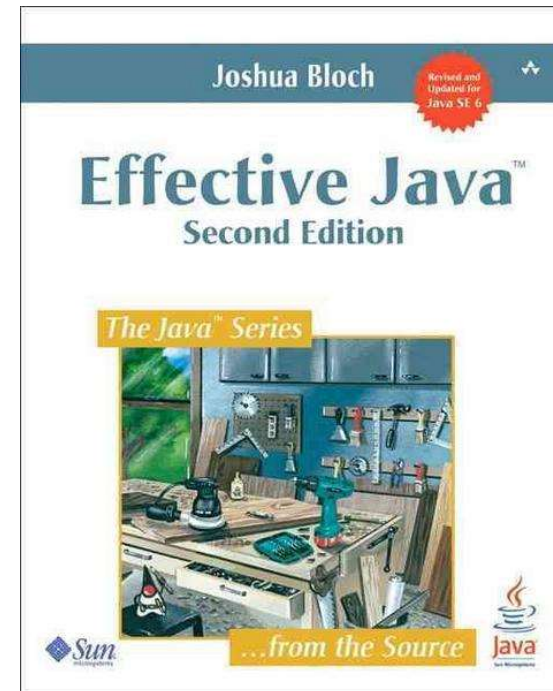
*Joshua Bloch:*

## **Effective Java, Second Edition**

Addison-Wesley

Mai 2008

ISBN: 978-0-321-35668-0



**Fragen?**