Wie gut kennst Du das Collections Framework?


@thfro

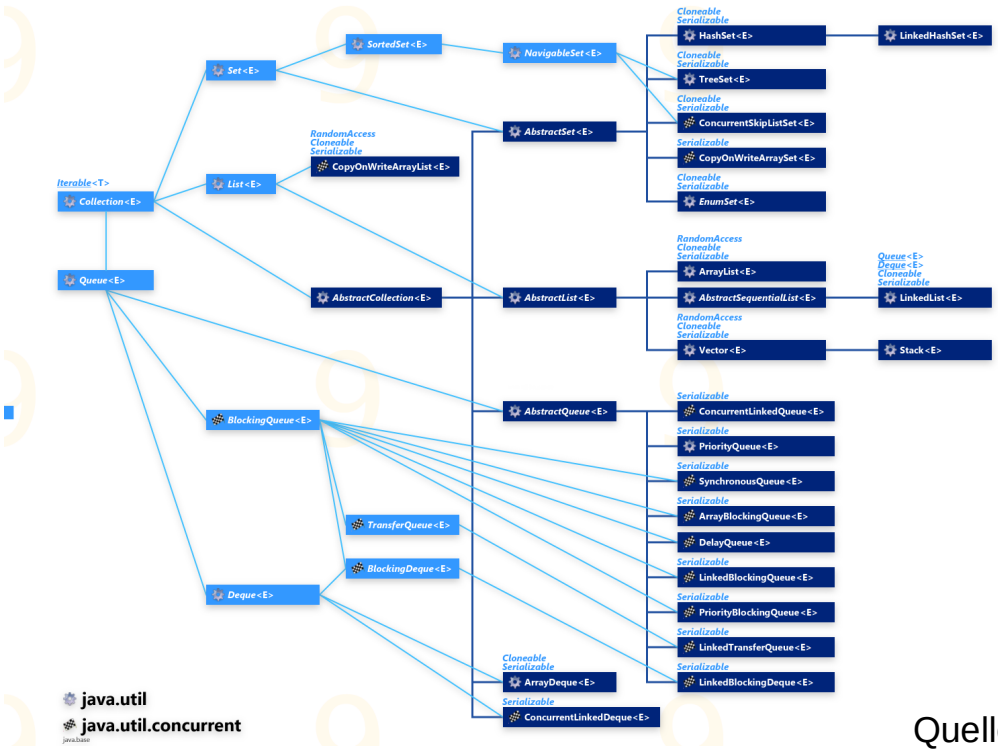
Historie

JDK 1.0	1996	`Vector`, `Stack`, `Hashtable`, `Dictionary`
J2SE 1.2	1998	Collections Framework
J2SE 5.0	2004	Generics, `java.util.concurrent`
Java SE 6	2006	`NavigableMap`, `NavigableSet`
Java SE 8	2014	Support für Streams, Predicates, etc.
Java SE 9	2017	Statische Factory-Methoden: `List.of(1, 2, 3);`

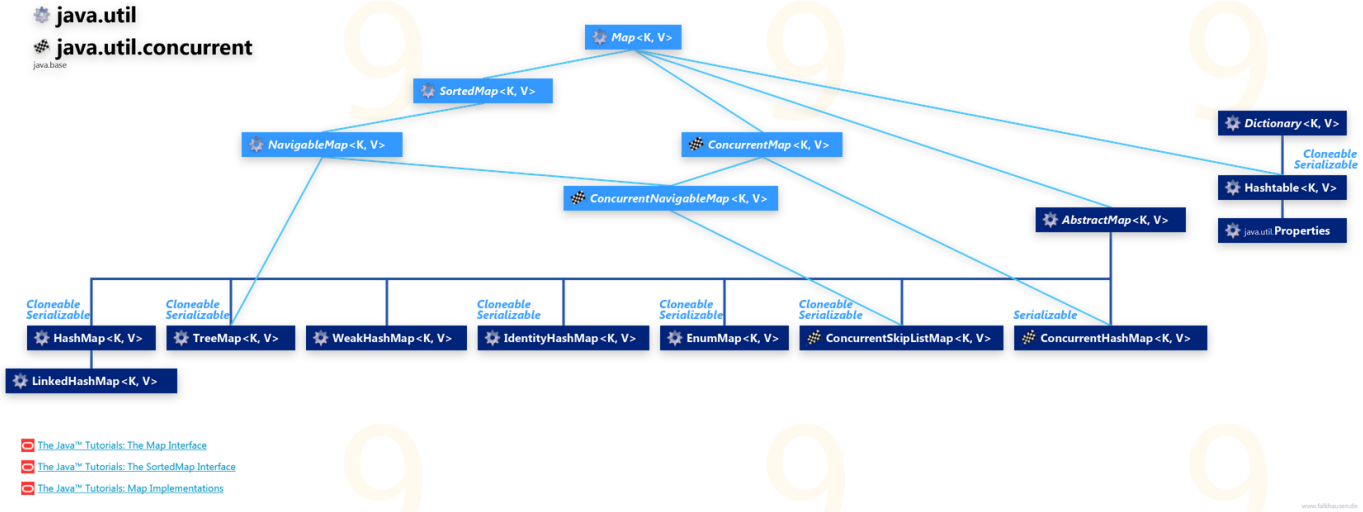
Wer die Wahl hat…

Implementierungen im Collections Framework (Java SE 9)

Map	8
Set	6
List	3
Queue	11



Quelle: falkhausen.de



Quelle: falkhausen.de

Auswahlkriterien

Laufzeitverhalten
(Performanz, Speicherverbrauch)

Thread-Sicherheit

…

O-Notation

Klassifizierung des Laufzeitverhaltens von Algorithmen

$O(1)$	konstant
$O(\log_2 n)$	logarithmisch
$O(n)$	linear
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(2^n)$	exponentiell

List

Geordnete Sammlung (Sequenz)

Präzise Kontrolle darüber wo in der Liste
ein Element eingefügt wird

Erlaubt (typischerweise) Duplikate und NULL-Elemente

List

4 Methoden für index-basierten Zugriff:
`add`, `get`, `remove`, `set`

“…may execute in time proportional to index”

”Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.”

```
public List<Company> findAllCompanies() {

  TypedQuery<Company> query = em.createNamedQuery(FIND_ALL, Company.class);
  return query.getResultList();

}
```


List

2 Methoden für die Suche nach Objekten:
`contains`, `lastIndexOf`

*“From a performance standpoint,
these methods should be used with caution.
In many implementations they will perform costly linear searches.”*

`equals()` sollte sinnvoll implementiert sein!

Wie viele Elemente kann eine `List` maximal aufnehmen?

(quasi) unbegrenzte Kapazität.

Wie viele Elemente kann ein `Array` maximal aufnehmen?

Fixe Kapazität.

Wie viele Elemente kann eine `ArrayList` maximal aufnehmen?

Default: 10

ArrayList

Implementierung von `List` auf Basis eines Arrays

Default-Kapazität: 10 => sinnvoll initialisieren!

Vergrößerung = Kopieren der Daten in neues Array

`grow()` verwendet `Arrays.copyOf(...)`

Vor dem Hinzufügen einer großen Anzahl von Elementen sollte Kapazität entsprechend erhöht werden:
=> `ensureCapacity()`

ArrayList

Vergleichbar mit `Vector`, jedoch nicht `synchronized`

Bei nebenläufigem Zugriff ist Synchronisation erforderlich - Monitor oder Wrapper verwenden:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

Iterator-Verhalten: fail-fast

```
Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
    Integer number = iterator.next();
    numbers.add(50);
}
```


LinkedList

Doppelt verkettete Liste

Beliebige Kapazität – Hinzufügen von Elementen sehr effizient

Index-basierter Zugriff ineffizient: lineare Suche entlang der Kette (je nach Index beginnt Suche am ersten oder letzten Element)

Nebenläufige Eigenschaften analog zu `ArrayList`

Was tun bei nebenläufigen Zugriffen?

`ArrayList` und `LinkedList` sind nicht thread-sicher

Synchronisierung kompletter Liste ist nicht effizient
(Monitor oder Wrapper)

=> Wir benötigen eine spezialisierte Implementierung von `List`

CopyOnWriteArrayList

Thread-sichere Variante von `ArrayList`

Alle verändernden Operationen (`add`, `remove`, etc.)
werden auf einer neuen Kopie der Liste ausgeführt

Iterator arbeitet auf einem “Snapshot” des Arrays,
daher keine Seiteneffekte durch nebenläufige Änderungen

Einsatz sinnvoll, wenn lesende / traversierende Zugriffe
stark überwiegen

Lesson Learned

Read the f*** JavaDoc!

Index-basierte Schleife über Elemente kann teuer sein!

Suche (`contains`, `lastIndexOf`) kann teuer sein!

`ArrayList` mit sinnvoller Kapazität initialisieren

`LinkedList` einsetzen, falls Anzahl der Elemente unbekannt

Nebenläufiger Einsatz: synchronisieren oder `CopyOnWriteArrayList`

Map

Abbildung von Schlüsseln auf Werte

3 Sichten (Collection Views): Keys, Values, Entries

Jeder Schlüssel zeigt auf genau einen Wert,
doppelte Schlüssel nicht erlaubt

=> `equals()` muss sinnvoll implementiert sein
=> Vorsicht bei veränderlichen Objekten als Schlüssel!

Seit 1.8: `getOrDefault`, `putIfAbsent`, `computeIfAbsent`
Seit 10: `copyOf`

HashMap

Map auf Basis einer Hash-Tabelle

Keine Zusagen über Sortierung

`hashCode()` und `equals()` der Schlüssel von zentraler Bedeutung

Für `containsValue()` ist `equals()` der Elemente wichtig

Nicht synchronisiert, Iterator mit “fail-fast” Verhalten

HashMap

Performanz abhängig von Initial Capacity und Load Factor

Zu klein gewählt => Häufiges Re-Hashing

Zu groß gewählt => Schlechte Performanz bei Iteration
(Kosten für Iteration proportional zur Kapazität: capacity + size)

*“Note that using many keys with the same hashCode()
is a sure way to slow down performance of any hash table”*

SortedMap & NavigableMap

Zugriff auf ersten / letzten Schlüssel (`firstKey`, `lastKey`)

Zugriff auf Teilmengen der `Map` (`headMap`, `tailMap`, `subMap`)

Zugriff auf Schlüssel (oder deren Einträge),
die < <= => > als ein gegebener Schlüssel sind
(`lowerKey`, `floorKey`, `higherKey`, `ceilingKey`)

Entfernen des ersten/letzten Eintrags (`pollFirstEntry`, `pollLastEntry`)

Map in umgekehrter Sortierreihenfolge (`descendingMap`)

TreeMap

Map auf Basis eines Red-Black Tree (Balanced Binary Tree)

Natural Ordering (`Comparable<T>`) oder `Comparator<T>`
=> Konsistenz mit `equals()` unbedingt erforderlich

Kosten sehr unterschiedlich zu `HashMap`:
O(log n) für `get`, `put`, `remove`, `containsKey`

Nicht synchronisiert, Iterator mit “fail-fast” Verhalten

LinkedHashMap

`HashMap` mit doppelt verketteter Liste der Einträge

Liste definiert die Iterations-Reihenfolge
(Default: Einfügereihenfolge, optional: LRU)

Abhilfe für Problem der normalerweise
chaotischen Iterationsreihenfolge einer `HashMap`
ohne die Kosten einer `TreeMap`

Performanzeigenschaften quasi identisch zu `HashMap`,
Iteration nur noch abhängig von Menge der Elemente

Thread-sichere Maps

`ConcurrentHashMap`

Thread-sichere `Map` auf Basis einer Hashtabelle
Lesende Zugriffe ohne Sperren
`concurrencyLevel` => Anzahl der Segmente

`ConcurrentSkipListMap`

Thread-sichere `NavigableMap` auf Basis einer Skip List
Natural Ordering (`Comparable<T>`) oder `Comparator<T>`
O(log n) für `get`, `put`, `remove`, `containsKey`

Spezialisierte Maps

`WeakHashMap`

Speichert Schlüssel als `WeakReference`

`EnumMap`

Alle Schlüssel müssen aus gleicher Enumeration stammen
Extrem kompakte und effiziente Implementierung (als Array!)

`IdentityHashMap`

Testet Gleichheit auf Basis von Referenzen (anstelle von `equals`)
=> Verletzung des Vertrags des `Map` Interfaces
=> nur für sehr seltene Einsatzgebiete zu verwenden

Lesson Learned

8 verschiedene Map-Implementierungen:
Use the right tool for the job!

Auswahlkriterien:

- Wird Sortierung benötigt?
- Wird nebenläufiger Zugriff benötigt?
- Bietet eine der spezialisierten Maps einen Mehrwert?

Keys + Elemente benötigen sinnvolle `equals()` / `hashCode()` Methoden

HashMaps sinnvoll initialisieren!

Set

Collection, die keine Duplikate enthält

=> Elemente benötigen sinnvolle Implementierung von `equals()`
=> Veränderliche Objekte führen zu undefiniertem Verhalten des Sets

Manche Sets haben Restriktionen bezüglich des Typs ihrer Elemente

Set

Viele Set-Implementierungen basieren auf anderen Collections:

Set-Implementierung	**Interne Datenstruktur**
`HashSet`	`HashMap`
`LinkedHashSet`	`HashMap`
`TreeSet*`	`TreeMap`
`ConcurrentSkipListSet*`	`ConcurrentSkipListMap`
`CopyOnWriteArraySet`	`CopyOnWriteArrayList`

hashCode() ebenfalls wichtig!

`*SortedSet`

EnumSet

Alle Elemente müssen aus der gleichen Enumeration stammen

Interne Implementierung als Bit-Vektor
=> extrem kompakt und effizient
=> alle Operationen in O(1)

Use Case: Typsichere Alternative zu int-basierten „Bit Flags“

NULL-Elemente nicht erlaubt

Nicht thread-sicher

Zusammenfassung

Collections Framework bietet weit mehr als nur `ArrayList` & `HashMap`

Viele spezialisierte Implementierungen für bestimmte Einsatzzwecke

Einsatz der falschen Implementierung kann diverse unerwünschte Folgen haben!

`equals` und `hashCode` haben zentrale Bedeutung!

Grundlegende Kenntnisse über die Eigenschaften der Collection-Implementierungen sollten selbstverständlich sein

Wofür die Zeit zu kurz war…

Queues (11 Implementierungen!)

Hilfreiche Methoden der Klasse `java.util.Collections`

Apache Commons Collections

Guava: Google Core Libraries for Java

Thilo Frotscher

Entwicklung, Beratung und kundenspezifisches Training

Enterprise Java, APIs, Services & Systemintegration

thilo@frotscher.com

@thfro