

Datenstrukturen in Java

Datenstrukturen und Algorithmen im JDK

Andreas Klipp,
Stephan Prätsch

11. März 2016



Inhaltsverzeichnis

- 1 Einleitung
- 2 `java.util.List`
- 3 `java.util.Map`
- 4 `java.util.Queue`
- 5 `java.util.Set`
- 6 Hilfsfunktionen
- 7 Abschluss

Einleitung

Warum das alles?

Warum das alles?

- Welche Liste soll ich nehmen?

Warum das alles?

- Welche Liste soll ich nehmen?
- Ich brauche eine Map, die thread-safe ist. Welche nehme ich?

Warum das alles?

- Welche Liste soll ich nehmen?
- Ich brauche eine Map, die thread-safe ist. Welche nehme ich?
- Ich brauche eine Queue. Welche gibt's überhaupt?

Warum das alles?

- Welche Liste soll ich nehmen?
- Ich brauche eine Map, die thread-safe ist. Welche nehme ich?
- Ich brauche eine Queue. Welche gibt's überhaupt?
- Mein Set soll sortiert sein. Gibt's ein SortedHashSet?

Einleitung

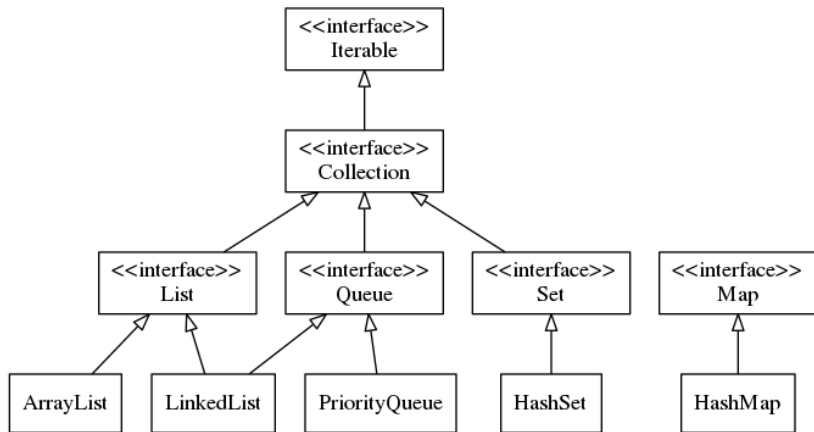
Warum das alles?

- Welche Liste soll ich nehmen?
- Ich brauche eine Map, die thread-safe ist. Welche nehme ich?
- Ich brauche eine Queue. Welche gibt's überhaupt?
- Mein Set soll sortiert sein. Gibt's ein SortedHashSet?

Ziel

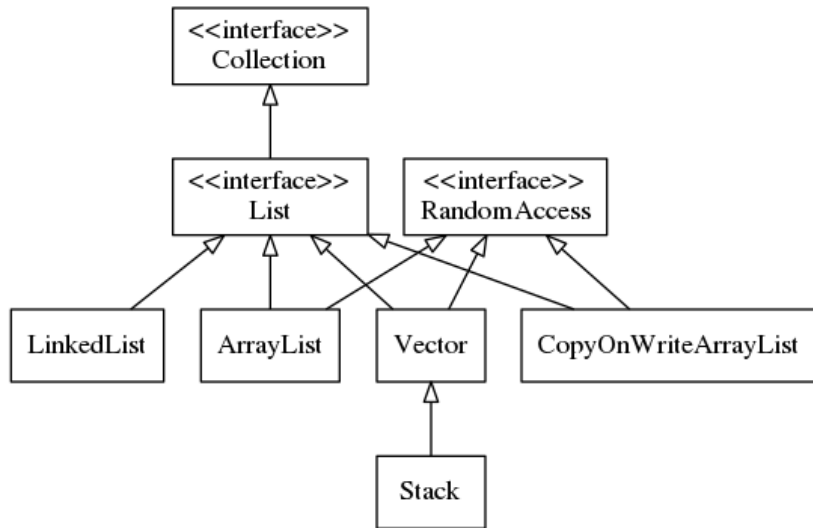
Übersicht der **vorhandenen** Datenstrukturen im JDK und deren **Besonderheiten**.

Stark gekürzte Übersicht



java.util.List

java.util.List



ArrayList

- Array beinhaltet Elemente

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector
- null Element erlaubt

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector
- null Element erlaubt
- remove Operationen verkleinern das Array nicht (`trimToSize()` verkleinert bis auf die aktuelle Größe)

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector
- null Element erlaubt
- remove Operationen verkleinern das Array nicht (`trimToSize()` verkleinert bis auf die aktuelle Größe)
- Re-size Operationen mittels `System.arraycopy` (nativ, schnell)

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector
- null Element erlaubt
- remove Operationen verkleinern das Array nicht (`trimToSize()` verkleinert bis auf die aktuelle Größe)
- Re-size Operationen mittels `System.arraycopy` (nativ, schnell)
- TODO 0 bzw. Startgröße, dann jeweils mal 2

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector
- null Element erlaubt
- remove Operationen verkleinern das Array nicht (`trimToSize()` verkleinert bis auf die aktuelle Größe)
- Re-size Operationen mittels `System.arraycopy` (nativ, schnell)
- TODO 0 bzw. Startgröße, dann jeweils mal 2
- remove und add sollten nur am Ende der Liste geschehen

ArrayList

- Array beinhaltet Elemente
- Kapazität kann explizit durch `ensureCapacity` erhöht werden
- fail fast Iterator
- quasi ein nicht synchronisierter Vector
- null Element erlaubt
- remove Operationen verkleinern das Array nicht (`trimToSize()` verkleinert bis auf die aktuelle Größe)
- Re-size Operationen mittels `System.arraycopy` (nativ, schnell)
- TODO 0 bzw. Startgröße, dann jeweils mal 2
- remove und add sollten nur am Ende der Liste geschehen
- nicht thread-safe

ArrayList - Zugriffszeiten

Operation	Laufzeit
add	$\mathcal{O}(1)$ / $\mathcal{O}(n)$ mit re-size
add(int, Object)	je kleiner die Position, desto länger
remove(Object)	$\mathcal{O}(n)$ TODO so viel wegen shifting? was ist shifting?
remove(int)	$\mathcal{O}(n)$ wegen shifting
get	$\mathcal{O}(1)$

CopyOnWriteArrayList

- thread-safe Variante von ArrayList

CopyOnWriteArrayList

- thread-safe Variante von ArrayList
- schreibende Operationen (add, set, ...) erstellen eine neue Kopie des Arrays

CopyOnWriteArrayList

- thread-safe Variante von ArrayList
- schreibende Operationen (add, set, ...) erstellen eine neue Kopie des Arrays
- "snaphot style iterator"

CopyOnWriteArrayList

- thread-safe Variante von ArrayList
- schreibende Operationen (add, set, ...) erstellen eine neue Kopie des Arrays
- "snaphot style iterator"
- diese Iteratoren unterstützen keine manipulativen Operationen (UnsupportedOperationException)

CopyOnWriteArrayList

- thread-safe Variante von ArrayList
- schreibende Operationen (add, set, ...) erstellen eine neue Kopie des Arrays
- "snaphot style iterator"
- diese Iteratoren unterstützen keine manipulativen Operationen (UnsupportedOperationException)
- null Elemente erlaubt

CopyOnWriteArrayList

- thread-safe Variante von ArrayList
- schreibende Operationen (add, set, ...) erstellen eine neue Kopie des Arrays
- "snaphot style iterator"
- diese Iteratoren unterstützen keine manipulativen Operationen (UnsupportedOperationException)
- null Elemente erlaubt
- seit Java 1.5

CopyOnWriteArrayList

- thread-safe Variante von ArrayList
- schreibende Operationen (add, set, ...) erstellen eine neue Kopie des Arrays
- "snaphot style iterator"
- diese Iteratoren unterstützen keine manipulativen Operationen (UnsupportedOperationException)
- null Elemente erlaubt
- seit Java 1.5
- lesen so teuer wie ArrayList, schreiben teurer wegen der Kopie

CopyOnWriteArrayList vs Collections.synchronizedList(new ArrayList())

- synchronizedList synchronisiert immer, auch lesende Zugriffe

CopyOnWriteArrayList vs Collections.synchronizedList(new ArrayList())

- synchronizedList synchronisiert immer, auch lesende Zugriffe
- Iterator der synchronizedList muss eigenständig synchronisiert werden (fail fast), CopyOneWrite hat fail save

CopyOnWriteArrayList vs Collections.synchronizedList(new ArrayList())

- synchronizedList synchronisiert immer, auch lesende Zugriffe
- Iterator der synchronizedList muss eigenständig synchronisiert werden (fail fast), CopyOnWrite hat fail save

⇒ CopyOnWriteArrayList

CopyOnWriteArrayList - Wann nehmen?

Wenn man eine ArrayList braucht,

CopyOnWriteArrayList - Wann nehmen?

Wenn man eine ArrayList braucht,
die thread-safe sein soll,

CopyOnWriteArrayList - Wann nehmen?

Wenn man eine ArrayList braucht,
die thread-safe sein soll,
mit wenig schreibenden, aber vielen lesenden Zugriffen.

LinkedList

- double linked

LinkedList

- double linked
- null Elemente erlaubt

LinkedList

- double linked
- null Elemente erlaubt
- Operationen mit Index $\mathcal{O}(n)$: Traversierung durch gesamte Liste, vorn oder hinten beginnend

LinkedList

- double linked
- null Elemente erlaubt
- Operationen mit Index $\mathcal{O}(n)$: Traversierung durch gesamte Liste, vorn oder hinten beginnend
- nicht thread-safe

LinkedList

- double linked
- null Elemente erlaubt
- Operationen mit Index $\mathcal{O}(n)$: Traversierung durch gesamte Liste, vorn oder hinten beginnend
- nicht thread-safe
- fail fast Iterator

LinkedList

- double linked
- null Elemente erlaubt
- Operationen mit Index $\mathcal{O}(n)$: Traversierung durch gesamte Liste, vorn oder hinten beginnend
- nicht thread-safe
- fail fast Iterator
- Deque Eigenschaften sind herausstechend: addFirst, getFirst, removeFirst, addLast, getLast und removeLast

LinkedList - Zugriffszeiten

Operation	Laufzeit
add	$\mathcal{O}(1)$
remove(Object)	$\mathcal{O}(1)$
remove(int)	$\mathcal{O}(n)$
get	$\mathcal{O}(n)$

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet
- LinkedList erzeugt pro Eintrag ein Node-Object (Overhead)

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet
- LinkedList erzeugt pro Eintrag ein Node-Object (Overhead)
- System.arraycopy muss sehr effizient sein, so dass es ein Vorteil von ArrayList ist

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet
- LinkedList erzeugt pro Eintrag ein Node-Object (Overhead)
- System.arraycopy muss sehr effizient sein, so dass es ein Vorteil von ArrayList ist
- ArrayList hat den Performance-Parameter *initial capacity*

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet
- LinkedList erzeugt pro Eintrag ein Node-Object (Overhead)
- System.arraycopy muss sehr effizient sein, so dass es ein Vorteil von ArrayList ist
- ArrayList hat den Performance-Parameter *initial capacity*
- LinkedList ist schneller beim Einfügen vorne und Löschen in der Mitte

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet
- LinkedList erzeugt pro Eintrag ein Node-Object (Overhead)
- System.arraycopy muss sehr effizient sein, so dass es ein Vorteil von ArrayList ist
- ArrayList hat den Performance-Parameter *initial capacity*
- LinkedList ist schneller beim Einfügen vorne und Löschen in der Mitte
- Collectors.toList() erstellt eine neue ArrayList

LinkedList vs ArrayList

- Laut [Oracle Doku](#): ArrayList ist besser
- ArrayList mit konstantem Zugriff auf Positionen / Index
- ArrayList ist meist schneller: Performance testen, bevor man sich für LinkedList entscheidet
- LinkedList erzeugt pro Eintrag ein Node-Object (Overhead)
- System.arraycopy muss sehr effizient sein, so dass es ein Vorteil von ArrayList ist
- ArrayList hat den Performance-Parameter *initial capacity*
- LinkedList ist schneller beim Einfügen vorne und Löschen in der Mitte
- `Collectors.toList()` erstellt eine neue ArrayList

⇒ ArrayList

LinkedList als Deque

Folie zuvor:

Deque Eigenschaften sind herausstechend: addFirst, getFirst, removeFirst, addLast, getLast und removeLast

LinkedList als Deque

Folie zuvor:

Deque Eigenschaften sind herausstechend: addFirst, getFirst, removeFirst, addLast, getLast und removeLast

Aber [Java Performance Tuning Guide](#) sagt:

- Wenn man schnellen LinkedList Code schreiben möchte, muss man ListIterators verwenden

LinkedList als Deque

Folie zuvor:

Deque Eigenschaften sind herausstechend: addFirst, getFirst, removeFirst, addLast, getLast und removeLast

Aber [Java Performance Tuning Guide](#) sagt:

- Wenn man schnellen LinkedList Code schreiben möchte, muss man ListIterators verwenden
- Wenn Queue / Deque benötigt wird, lieber ArrayDeque als LinkedList

LinkedList - Wann nehmen?

Wenn sehr oft `remove(Object)` genutzt wird oder

LinkedList - Wann nehmen?

Wenn sehr oft `remove(Object)` genutzt wird oder
oft vorne Elemente eingefügt werden.

LinkedList - Wann nehmen?

Wenn sehr oft `remove(Object)` genutzt wird oder
oft vorne Elemente eingefügt werden.

Kurz: **Eigentlich gar nicht. Besser ArrayList oder ArrayDeque**

Stack

- LIFO: push, pop, peek, search

Stack

- LIFO: push, pop, peek, search
- seit JDK 1.0

Stack - Wann nehmen?

Ein besseres Interface für LIFO Operationen stellt Deque zur Verfügung:

Stack - Wann nehmen?

Ein besseres Interface für LIFO Operationen stellt Deque zur Verfügung:
Gar nicht mehr

Vector

- seit JDK 1.0

Vector

- seit JDK 1.0
- wie ein Array (bzw. ArrayList)

Vector

- seit JDK 1.0
- wie ein Array (bzw. ArrayList)
- thread-safe

Vector

- seit JDK 1.0
- wie ein Array (bzw. ArrayList)
- thread-safe
- alle public Methoden synchronized: einfach, deshalb langsam

Vector

- seit JDK 1.0
- wie ein Array (bzw. ArrayList)
- thread-safe
- alle public Methoden synchronized: einfach, deshalb langsam
- fail-fast Iterator

Stack - Wann nehmen?

- existiert nur noch wegen Abwärtskompatibilität (wurde in Collections "reingepresst")

Stack - Wann nehmen?

- existiert nur noch wegen Abwärtskompatibilität (wurde in Collections "reingepresst")
- wenn **kein thread-safe** benötigt wird, dann **ArrayList**, weil schneller

Stack - Wann nehmen?

- existiert nur noch wegen Abwärtskompatibilität (wurde in Collections "reingepresst")
- wenn **kein thread-safe** benötigt wird, dann **ArrayList**, weil schneller
- wenn **thread-safe** benötigt wird, dann unklar: Vector, Collections.synchronizedList oder CopyOnWriteArrayList

Stack - Wann nehmen?

- existiert nur noch wegen Abwärtskompatibilität (wurde in Collections "reingepresst")
- wenn **kein thread-safe** benötigt wird, dann **ArrayList**, weil schneller
- wenn **thread-safe** benötigt wird, dann unklar: Vector, Collections.synchronizedList oder CopyOnWriteArrayList
- bei Collections.synchronizedList wird zwischen Datenstruktur und Synchronisation getrennt

Stack - Wann nehmen?

- existiert nur noch wegen Abwärtskompatibilität (wurde in Collections "reingepresst")
- wenn **kein thread-safe** benötigt wird, dann **ArrayList**, weil schneller
- wenn **thread-safe** benötigt wird, dann unklar: Vector, Collections.synchronizedList oder CopyOnWriteArrayList
- bei Collections.synchronizedList wird zwischen Datenstruktur und Synchronisation getrennt

⇒ Tendenz ArrayList oder Collections.synchronizedList

List - Übersicht

	thread-safe	Iterator	Reihenfolge	null value
LinkedList		fail-fast	insertion	erlaubt
ArrayList		fail-fast	insertion	erlaubt
Vector	ja	fail-fast	insertion	erlaubt
Stack		fail-fast	insertion	erlaubt
CopyOnWriteArrayList	ja	fail-safe	insertion	erlaubt

java.util.Map

Maps

java.util.Queue

Queues

java.util.Set

Sets

Hilfsfunktionen

Hilfsfunktionen für Listen

java.util.Collections.emptyList()

```
List<String> l = Collections.emptyList();
```

Hilfsfunktionen für Listen

java.util.Collections.emptyList()

```
List<String> l = Collections.emptyList();
```

com.google.common.collect.Lists.newArrayList(T...)

```
List<String> l = Lists.newArrayList("a", "b", "c", "d");
```

Hilfsfunktionen für Listen

java.util.Collections.emptyList()

```
List<String> l = Collections.emptyList();
```

com.google.common.collect.Lists.newArrayList(T...)

```
List<String> l = Lists.newArrayList("a", "b", "c", "d");
```

java.util.AbstractList.subList(int, int)

from-index inklusiv, to-index exklusiv

```
List<String> l = ...  
List<String> sub = l.subList(2,5);
```

Hilfsfunktionen für Listen

java.util.Collections.emptyList()

```
List<String> l = Collections.emptyList();
```

com.google.common.collect.Lists.newArrayList(T...)

```
List<String> l = Lists.newArrayList("a", "b", "c", "d");
```

java.util.AbstractList.subList(int, int)

from-index inklusiv, to-index exklusiv

```
List<String> l = ...  
List<String> sub = l.subList(2,5);
```

java.util.Collections.unmodifiableList(List<? extends T>)

```
Collections.unmodifiableList(list).add("a"); // wirft UnsupportedOperationException
```

Hilfsfunktionen für Listen

java.util.Collections.singletonList(T)

```
List<String> list = Collections.singletonList("a");
```

Hilfsfunktionen für Listen

java.util.Collections.singletonList(T)

```
List<String> list = Collections.singletonList("a");
```

java.util.Collections.synchronizedList(List<T>)

wrappt eine nicht synchronisierte List

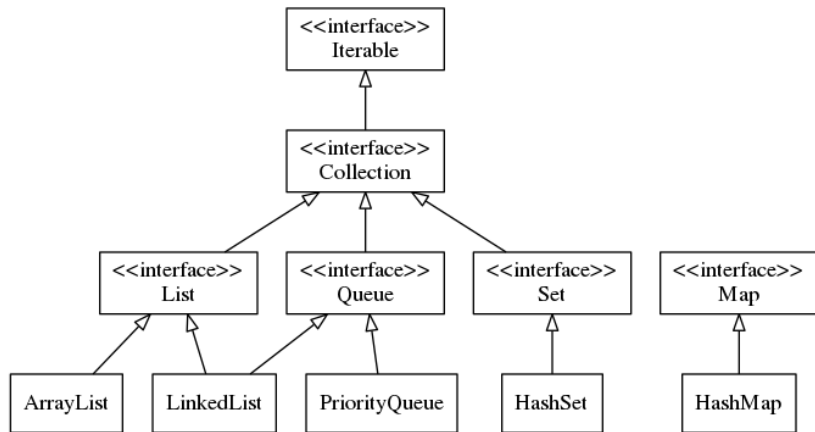
```
List<String> l = ...  
List<String> s = Collections.synchronizedList(l);
```

Abschluss

Abschluss

Warum das alles

Stark gekürzte Übersicht



Leicht gekürzte Übersicht

