Datenstrukturen in Java

Datenstrukturen und Algorithmen im JDK

Andreas Klipp,
Stephan Prätsch

29. April 2016

Inhaltsverzeichnis

mercateo

Einleitung

Einleitung

Warum das alles?

Einleitung

Warum das alles?

- Welche Liste soll ich nehmen?

mercateo

Einleitung

Warum das alles?

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

mercateo

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?

mercateo

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?

mercateo

Einleitung

Warum das alles?

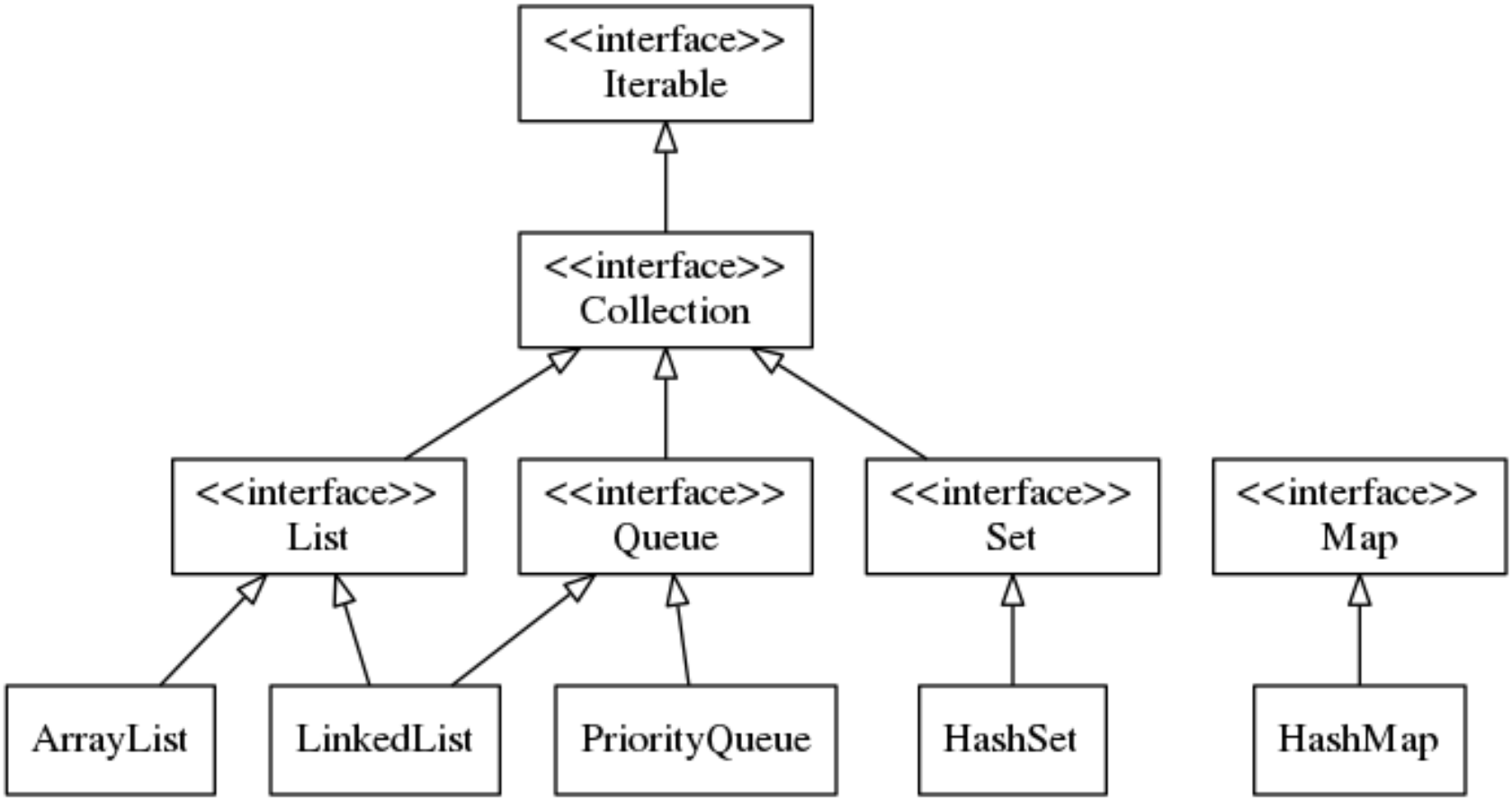
- Welche Liste soll ich nehmen?
- Ich brauche eine Map, die thread-safe ist. Welche nehme ich?
- Ich brauche eine Queue. Welche gibt's überhaupt?
- Mein Set soll sortiert sein. Gibt's ein SortedHashSet?

Ziel

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

mercateo

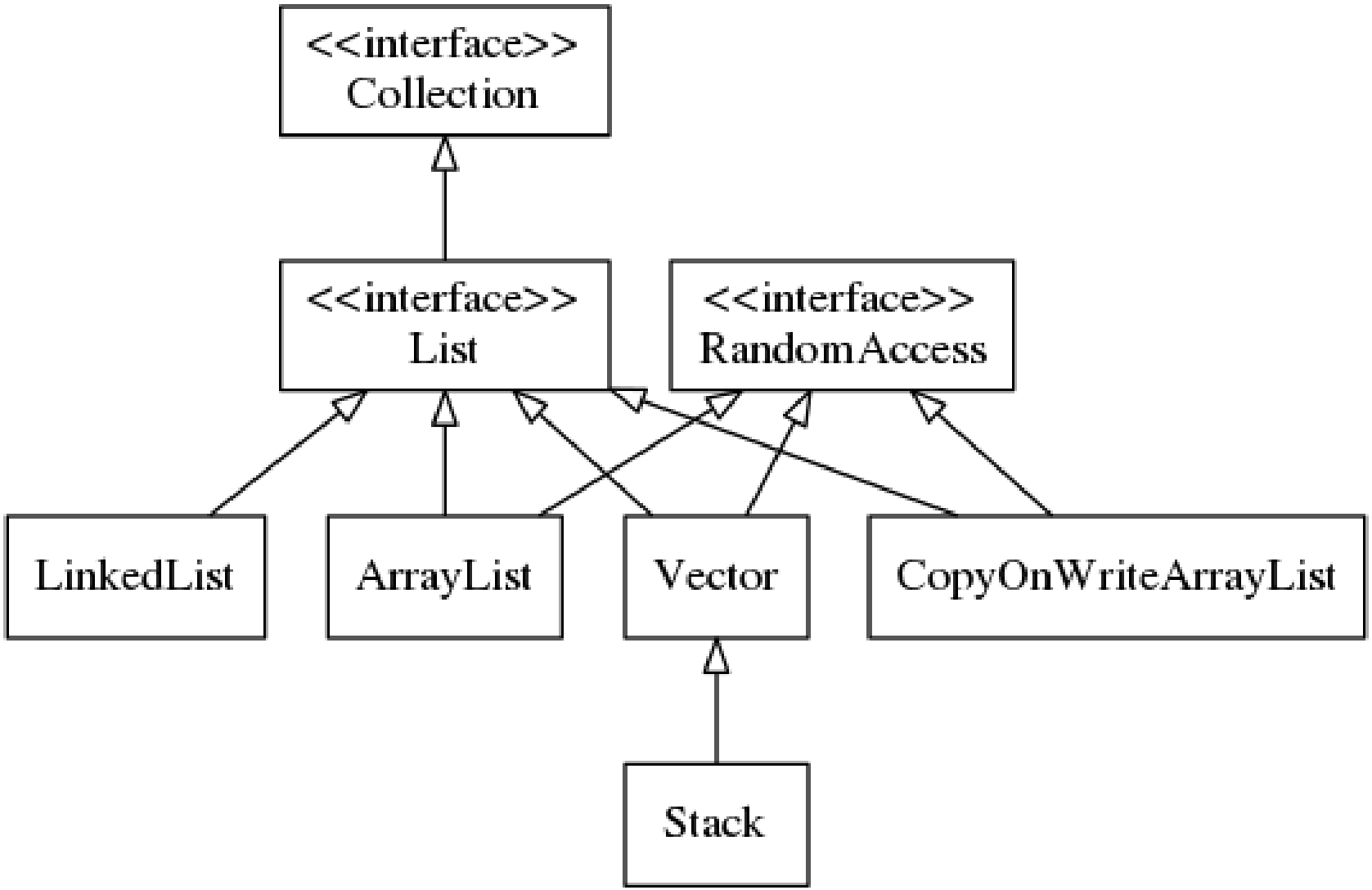
Stark gekürzte Übersicht



mercateo

java.util.List

java.util.List



mercateo

ArrayList

- Array beinhaltet Elemente

mercateo

ArrayList

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

mercateo

ArrayList

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

mercateo

ArrayList

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

mercateo

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)

mercateo

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.arrycopy (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.arrycopy (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.arrycopy (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.arrycopy (nativ, schnell)
- TODO 0 bzw. Startgröße, dann jeweils mal 2
- remove und add sollten nur am Ende der Liste geschehen
- nicht thread-safe

mercateo

ArrayList - Zugriffszeiten

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

mercateo

CopyOnWriteArrayList

- thread-safe Variante von ArrayList

mercateo

CopyOnWriteArrayList

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

mercateo

CopyOnWriteArrayList

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

mercateo

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)

mercateo

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

mercateo

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

mercateo

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

mercateo

CopyOnWriteArrayList vs Collections.synchronizedList(new ArrayList()

- synchronizedList synchronisiert immer, auch lesende Zugriffe

mercateo

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

mercateo

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

$\Rightarrow$ CopyOnWriteArrayList

mercateo

CopyOnWriteArrayList - Wann nehmen?

Wenn man eine ArrayList braucht,

mercateo

CopyOnWriteArrayList - Wann nehmen?

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

mercateo

CopyOnWriteArrayList - Wann nehmen?

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

mercateo

LinkedList

- double linked

LinkedList

- double linked
- null Elemente erlaubt

mercateo

LinkedList

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

mercateo

LinkedList

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

mercateo

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

mercateo

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

mercateo

LinkedList - Zugriffszeiten

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

mercateo

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

mercateo

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

mercateo

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

$\Rightarrow$ ArrayList

mercateo

LinkedList als Deque

Folie zuvor:

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

mercateo

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

mercateo

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

mercateo

LinkedList - Wann nehmen?

Wenn sehr oft remove(Object) genutzt wird oder

mercateo

LinkedList - Wann nehmen?

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

mercateo

LinkedList - Wann nehmen?

Wenn sehr oft remove(Object) genutzt wird oder
oft vorne Elemente eingefügt werden.
Kurz: **Eigentlich gar nicht. Besser ArrayList oder ArrayDeque**

mercateo

Stack

- LIFO: push, pop, peek, search

mercateo

Stack

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

mercateo

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

mercateo

Vector

- seit JDK 1.0

mercateo

Vector

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

mercateo

Vector

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

mercateo

Vector

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

mercateo

Vector

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

mercateo

Stack - Wann nehmen?

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

mercateo

Stack - Wann nehmen?

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

mercateo

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

mercateo

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

mercateo

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

$\Rightarrow$ Tendenz ArrayList oder Collections.synchronizedList

mercateo

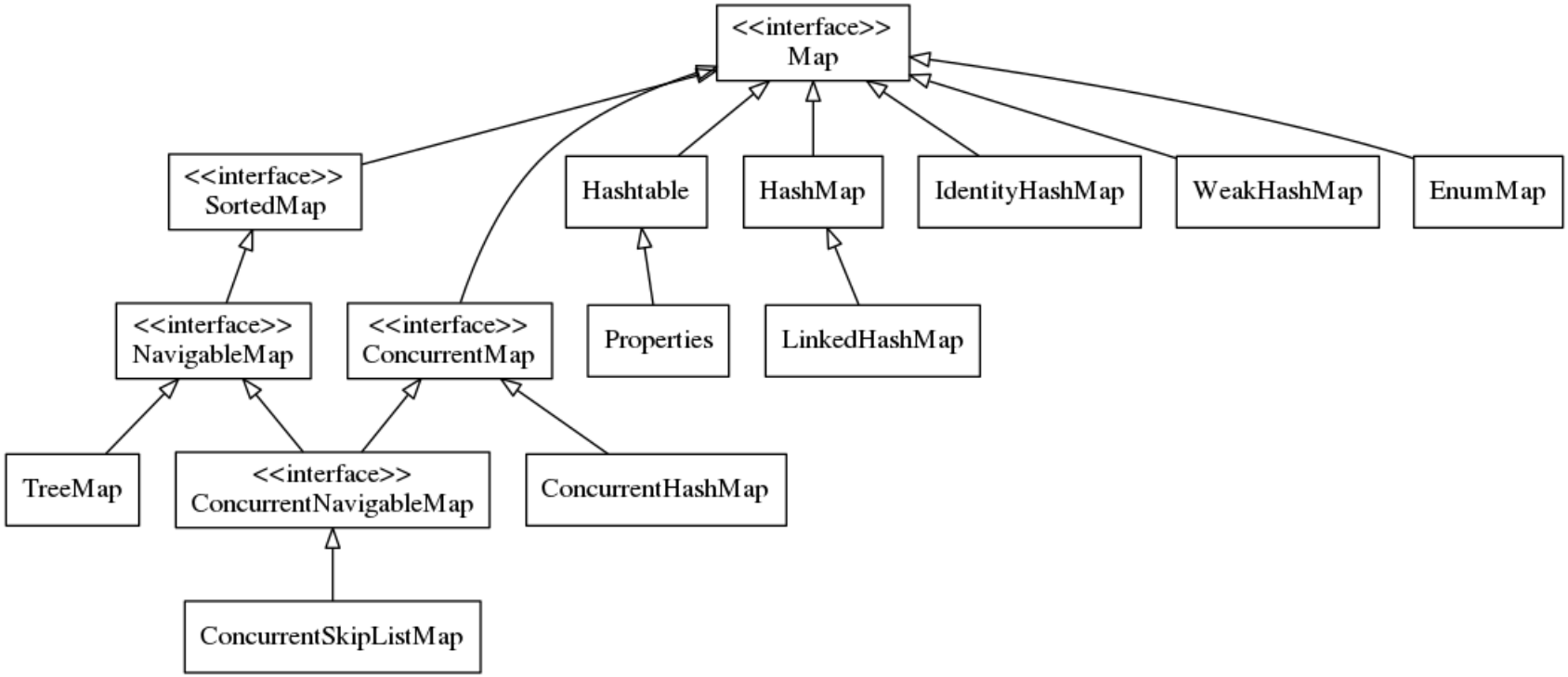
List - Übersicht

	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

mercateo

java.util.Map

java.util.Map



mercateo

HashMap

- seit 1.2

mercateo

HashMap

- seit 1.2
- null keys und values erlaubt

HashMap

- seit 1.2
- null keys und values erlaubt
- wie Hashtable, aber mit nulls und nicht thread-safe

mercateo

HashMap

- seit 1.2
- null keys und values erlaubt
- wie Hashtable, aber mit nulls und nicht thread-safe
- get/put in $\mathcal{O}(1)$

mercateo

HashMap

- seit 1.2
- null keys und values erlaubt
- wie Hashtable, aber mit nulls und nicht thread-safe
- get/put in $\mathcal{O}(1)$
- nicht thread-safe

mercateo

HashMap

- seit 1.2
- null keys und values erlaubt
- wie Hashtable, aber mit nulls und nicht thread-safe
- get/put in $\mathcal{O}(1)$
- nicht thread-safe
- fail-fast iterator

mercateo

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity
- loadFactor = Ab wann automatisch rehash

mercateo

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity
- loadFactor = Ab wann automatisch rehash
- rehash: interne Datenstruktur wird neu gebaut $\Rightarrow$ Kapazität steigt um Faktor 2

mercateo

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity
- loadFactor = Ab wann automatisch rehash
- rehash: interne Datenstruktur wird neu gebaut ⇒ Kapazität steigt um Faktor 2
- Konstruktor: initialCapacity und loadFactor

```
DEFAULT_LOAD_FACTOR = 0.75f
DEFAULT_INITIAL_CAPACITY = 1 << 4; // = 16
```

mercateo

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity
- loadFactor = Ab wann automatisch rehash
- rehash: interne Datenstruktur wird neu gebaut $\Rightarrow$ Kapazität steigt um Faktor 2
- Konstruktor: initialCapacity und loadFactor
  ```
  DEFAULT_LOAD_FACTOR = 0.75f
  DEFAULT_INITIAL_CAPACITY = 1 << 4; // = 16
  ```
- wenn $number\ entries \geq loadFactor * capacity$ dann rehash

mercateo

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity
- loadFactor = Ab wann automatisch rehash
- rehash: interne Datenstruktur wird neu gebaut $\Rightarrow$ Kapazität steigt um Faktor 2
- Konstruktor: initialCapacity und loadFactor
  `DEFAULT_LOAD_FACTOR = 0.75f`
  `DEFAULT_INITIAL_CAPACITY = 1 << 4; // = 16`
- wenn $number\ entries \geq loadFactor * capacity$ dann rehash
- wenn viele puts, dann sollte initialCapacity groß genug sein, um Anzahl an rehashes klein zu halten

mercateo

HashMap - interne Struktur

- capacity = Anzahl an buckets, initialCapacity = Start capacity
- loadFactor = Ab wann automatisch rehash
- rehash: interne Datenstruktur wird neu gebaut $\Rightarrow$ Kapazität steigt um Faktor 2
- Konstruktor: initialCapacity und loadFactor
  `DEFAULT_LOAD_FACTOR = 0.75f`
  `DEFAULT_INITIAL_CAPACITY = 1 << 4; // = 16`
- wenn $number\ entries \geq loadFactor * capacity$ dann rehash
- wenn viele puts, dann sollte initialCapacity groß genug sein, um Anzahl an rehashes klein zu halten
- aber initialCapacity nicht zu hoch und loadFactor nicht zu niedrig setzen, sonst zu viele rehashes

mercateo

HashMap - Zugriffszeiten

Operation	Laufzeit
get	$\mathcal{O}(1)$
put	$\mathcal{O}(1)$

mercateo

HashMap - Wann nehmen?

Synchronisation egal,

HashMap - Wann nehmen?

Synchronisation egal,
Ordnung egal,

mercateo

HashMap - Wann nehmen?

Synchronisation egal,
Ordnung egal,
oft get/put

mercateo

LinkedHashMap

- null erlaubt

LinkedHashMap

- null erlaubt
- add, contains, remove $\mathcal{O}(1)$

mercateo

LinkedHashMap

- null erlaubt
- add, contains, remove $\mathcal{O}(1)$
- iteration $= \mathcal{O}(\mathit{size})$, HashMap $= \mathcal{O}(\mathit{capacity})$, schneller falls $\mathit{size} < \mathit{capacity}$

mercateo

LinkedHashMap

- null erlaubt
- add, contains, remove $\mathcal{O}(1)$
- iteration $= \mathcal{O}(size)$, HashMap $= \mathcal{O}(capacity)$, schneller falls $size < capacity$
- initial capacity and load factor wie HashMap

mercateo

LinkedHashMap

- null erlaubt
- add, contains, remove $\mathcal{O}(1)$
- iteration $= \mathcal{O}(\mathit{size})$, HashMap $= \mathcal{O}(\mathit{capacity})$, schneller falls $\mathit{size} < \mathit{capacity}$
- initial capacity and load factor wie HashMap
- not synchronized, nicht thread-safe

mercateo

LinkedHashMap

- null erlaubt
- add, contains, remove $\mathcal{O}(1)$
- iteration $= \mathcal{O}(\mathit{size})$, HashMap $= \mathcal{O}(\mathit{capacity})$, schneller falls $\mathit{size} < \mathit{capacity}$
- initial capacity and load factor wie HashMap
- not synchronized, nicht thread-safe
- fail-fast iterator

mercateo

LinkedHashMap

- null erlaubt
- add, contains, remove $\mathcal{O}(1)$
- iteration $= \mathcal{O}(\mathit{size})$, HashMap $= \mathcal{O}(\mathit{capacity})$, schneller falls $\mathit{size} < \mathit{capacity}$
- initial capacity and load factor wie HashMap
- not synchronized, nicht thread-safe
- fail-fast iterator
- Reihenfolge nach Einfügen (insertion-order)

mercateo

LinkedHashMap - LRU-Cache

Least Recently Used

1. *new LinkedHashMap(initialCapacity, loadFactor, accessOrder)*
   accessOrder = true für access-order, von least nach most-recently

mercateo

LinkedHashMap - LRU-Cache

Least Recently Used

1. *new LinkedHashMap(initialCapacity, loadFactor, accessOrder)*
   accessOrder = true für access-order, von least nach most-recently
2. um nach put / putAll zu aktualisieren removeEldestEntry überschreiben

```
@Override
protected boolean removeEldestEntry(Map.Entry eldest) {
   return size() > 100;
}
```

mercateo

LinkedHashMap - Zugriffszeiten

Operation	Laufzeit
add	$\mathcal{O}(1)$
contains	$\mathcal{O}(1)$
remove	$\mathcal{O}(1)$
get	$\mathcal{O}(TODO)$
put	$\mathcal{O}(TODO)$

mercateo

LinkedHashMap - Wann nehmen?

Reihenfolge wichtig oder

mercateo

LinkedHashMap - Wann nehmen?

Reihenfolge wichtig oder
schnelle Iteration (schneller als HashMap)

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"
- null keys/values erlaubt

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"
- null keys/values erlaubt
- keine Reihenfolge

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"
- null keys/values erlaubt
- keine Reihenfolge
- get/put in O(1)

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"
- null keys/values erlaubt
- keine Reihenfolge
- get/put in O(1)
- expected maximum size sollte genutzt werden, erweitern ist teuer

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"
- null keys/values erlaubt
- keine Reihenfolge
- get/put in O(1)
- expected maximum size sollte genutzt werden, erweitern ist teuer
- nicht thread-safe

mercateo

IdentityHashMap

- hashed mit System.identityHashCode(Object) anstatt der hashCode Implementierung
- Referenz-Gleichheit anstatt equals
  if (k1==k2) anstatt
  if (k1==null ? k2==null : k1.equals(k2)) (HashMap)
- betrifft nur keys
- verletzt bewusst den Map-Vertrag "equals() zum Vergleichen"
- null keys/values erlaubt
- keine Reihenfolge
- get/put in O(1)
- expected maximum size sollte genutzt werden, erweitern ist teuer
- nicht thread-safe
- fail-fast iterator

mercateo

IdentityHashMap - Wann nehmen?

nur wenn Referenz-Gleichheit gebraucht wird (sehr selten)

mercateo

WeakHashMap

- weak keys (als weak reference)

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

$\Rightarrow$ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

$\Rightarrow$ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

$\Rightarrow$ workaround: values in WeakReference einpacken

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

$\Rightarrow$ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

$\Rightarrow$ workaround: values in WeakReference einpacken

- null key, null value supported

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

$\Rightarrow$ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

$\Rightarrow$ workaround: values in WeakReference einpacken

- null key, null value supported
- Effizienz wie HashMap

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

$\Rightarrow$ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

$\Rightarrow$ workaround: values in WeakReference einpacken

- null key, null value supported
- Effizienz wie HashMap
- nicht thread-safe

mercateo

WeakHashMap

- weak keys (als weak reference)
- GC räumt Key weg auch wenn es einen Value gibt
- Wenn ein Key weggeräumt wurde, wird der Value (hard reference) aus der Map entfernt

$\Rightarrow$ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

$\Rightarrow$ workaround: values in WeakReference einpacken

- null key, null value supported
- Effizienz wie HashMap
- nicht thread-safe
- fail-fast iterator

mercateo

WeakHashMap - Wann nehmen?

object reference als key und GC soll die Map beaufsichtigen (also sehr selten)

mercateo

WeakHashMap - Wann nehmen?

object reference als key und GC soll die Map beaufsichtigen (also sehr selten)
data cache implementations

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich
- kann Hashtable komplett ersetzen, falls man nur auf thread-safety angewiesen ist, nicht auf die Synchronisations-Details von Hashtable

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich
- kann Hashtable komplett ersetzen, falls man nur auf thread-safety angewiesen ist, nicht auf die Synchronisations-Details von Hashtable
- read-Methoden stellen (irgend)einen Zustand dar, vielleicht auch nur die Hälfte von einem putAll

mercateo

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich
- kann Hashtable komplett ersetzen, falls man nur auf thread-safety angewiesen ist, nicht auf die Synchronisations-Details von Hashtable
- read-Methoden stellen (irgend)einen Zustand dar, vielleicht auch nur die Hälfte von einem putAll
- fail-safe iterator

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich
- kann Hashtable komplett ersetzen, falls man nur auf thread-safety angewiesen ist, nicht auf die Synchronisations-Details von Hashtable
- read-Methoden stellen (irgend)einen Zustand dar, vielleicht auch nur die Hälfte von einem putAll
- fail-safe iterator
- Änderung der Größe ist teuer

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich
- kann Hashtable komplett ersetzen, falls man nur auf thread-safety angewiesen ist, nicht auf die Synchronisations-Details von Hashtable
- read-Methoden stellen (irgend)einen Zustand dar, vielleicht auch nur die Hälfte von einem putAll
- fail-safe iterator
- Änderung der Größe ist teuer
- Größe optimieren: Konstruktor-Parameter concurrencyLevel gibt Anzahl parallel schreibender Threads an

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap
- ist wie Hashtable (bspw. thread-safe)
- lesende Operationen blockieren nicht
- jeglichen Zugriff blockieren ist nicht möglich
- kann Hashtable komplett ersetzen, falls man nur auf thread-safety angewiesen ist, nicht auf die Synchronisations-Details von Hashtable
- read-Methoden stellen (irgend)einen Zustand dar, vielleicht auch nur die Hälfte von einem putAll
- fail-safe iterator
- Änderung der Größe ist teuer
- Größe optimieren: Konstruktor-Parameter concurrencyLevel gibt Anzahl parallel schreibender Threads an
- kein null erlaubt (weder key noch value)

mercateo

ConcurrentHashMap - Wann nehmen?

wenn eine thread-safe Map benötigt wird oder

mercateo

ConcurrentHashMap - Wann nehmen?

wenn eine thread-safe Map benötigt wird oder
Hashtable ersetzt werden soll oder

mercateo

ConcurrentHashMap - Wann nehmen?

wenn eine thread-safe Map benötigt wird oder
Hashtable ersetzt werden soll oder
fail-safe Iterator

mercateo

Hashtable

- fail-fast iterator

mercateo

Hashtable

- fail-fast iterator
- thread-safe

Hashtable

- fail-fast iterator
- thread-safe
- seit 1.0

mercateo

Hashtable

- fail-fast iterator
- thread-safe
- seit 1.0
- keine null-keys / values

mercateo

Hashtable

- fail-fast iterator
- thread-safe
- seit 1.0
- keine null-keys / values
- keys müssen hashCode/equals implementieren

mercateo

Hashtable

- fail-fast iterator
- thread-safe
- seit 1.0
- keine null-keys / values
- keys müssen hashCode/equals implementieren
- (initial) capacity and load factor: für Speicher- und Laufzeit-Optimierung (bspw. rehash)

mercateo

Hashtable

- fail-fast iterator
- thread-safe
- seit 1.0
- keine null-keys / values
- keys müssen hashCode/equals implementieren
- (initial) capacity and load factor: für Speicher- und Laufzeit-Optimierung (bspw. rehash)
- alles mit public ist synchronisiert

mercateo

Hashtable - Wann nehmen?

Gar nicht mehr

mercateo

Hashtable - Wann nehmen?

Gar nicht mehr

Wenn thread-safe nicht nötig, dann **HashMap**

Wenn thread-safe nötig, dann **ConcurrentHashMap**

mercateo

Properties

- seit 1.0

Properties

- seit 1.0
- extends Hashtable< Object,Object>

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)

mercateo

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte

mercateo

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte
- load(Reader) / store(Writer, String)

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte
- load(Reader) / store(Writer, String)
- store schreibt keine defaults

mercateo

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte
- load(Reader) / store(Writer, String)
- store schreibt keine defaults
- Kommentare mit # sind konfigurierbar unterstützt

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte
- load(Reader) / store(Writer, String)
- store schreibt keine defaults
- Kommentare mit # sind konfigurierbar unterstützt
- Leerzeichen in keys und führende in values werden mit "\ " geschrieben

mercateo

Properties

- seit 1.0
- extends Hashtable< Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte
- load(Reader) / store(Writer, String)
- store schreibt keine defaults
- Kommentare mit # sind konfigurierbar unterstützt
- Leerzeichen in keys und führende in values werden mit "\ " geschrieben
- thread-safe, weil Hashtable

mercateo

Properties

- seit 1.0
- extends Hashtable<Object,Object>
- setProperty(String key, String value) nutzen anstatt put/putAll (lassen Object zu)
- Konstruktor nimmt eine Properties Instanz als default-Werte
- load(Reader) / store(Writer, String)
- store schreibt keine defaults
- Kommentare mit # sind konfigurierbar unterstützt
- Leerzeichen in keys und führende in values werden mit "\ " geschrieben
- thread-safe, weil Hashtable
- obwohl von Hashtable abgeraten wird, wird von Properties (noch) nicht abgeraten

Properties - Meine Meinung

falsch implementiert

extends Hashtable< Object,Object>, man soll aber nur mit Strings arbeiten $\Rightarrow$ wieso dann nicht extends Hashtable< String,String>?

Properties - Wann nehmen?

key-value-Paare als Konfiguration, die man persistieren möchte (siehe load/store)

mercateo

Properties - Wann nehmen?

key-value-Paare als Konfiguration, die man persistieren möchte (siehe load/store)
Alternative: java.util.prefs.Preferences

mercateo

EnumMap

- keys = enum values

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value
- sehr kompakt und effizient

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value
- sehr kompakt und effizient
- Reihenfolge ist die, wie die Enums deklariert sind

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value
- sehr kompakt und effizient
- Reihenfolge ist die, wie die Enums deklariert sind
- weakly consistent iterators

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value
- sehr kompakt und effizient
- Reihenfolge ist die, wie die Enums deklariert sind
- weakly consistent iterators
- keine null keys, aber null values erlaubt

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value
- sehr kompakt und effizient
- Reihenfolge ist die, wie die Enums deklariert sind
- weakly consistent iterators
- keine null keys, aber null values erlaubt
- nicht thread-safe

mercateo

EnumMap

- keys = enum values
- intern ein Array pro enum-value
- sehr kompakt und effizient
- Reihenfolge ist die, wie die Enums deklariert sind
- weakly consistent iterators
- keine null keys, aber null values erlaubt
- nicht thread-safe
- basic operations in $\mathcal{O}(1)$

mercateo

EnumMap - Wann nehmen?

wenn keys enums sind

mercateo

TreeMap

- fail-fast iterator

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe
- nutzt compareTo anstatt equals/hashcode

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe
- nutzt compareTo anstatt equals/hashcode
- intern Red-Black-Tree

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe
- nutzt compareTo anstatt equals/hashcode
- intern Red-Black-Tree
- Reihenfolge: Natürliche Ordnung oder Comparator

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe
- nutzt compareTo anstatt equals/hashcode
- intern Red-Black-Tree
- Reihenfolge: Natürliche Ordnung oder Comparator
- containsKey, get, put and remove in $\mathcal{O}(log(n))$

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe
- nutzt compareTo anstatt equals/hashcode
- intern Red-Black-Tree
- Reihenfolge: Natürliche Ordnung oder Comparator
- containsKey, get, put and remove in $\mathcal{O}(log(n))$
- equals und compareTo müssen konsistent sein
  $equals \Leftrightarrow compareTo == 0$

mercateo

TreeMap

- fail-fast iterator
- nicht thread-safe
- nutzt compareTo anstatt equals/hashcode
- intern Red-Black-Tree
- Reihenfolge: Natürliche Ordnung oder Comparator
- containsKey, get, put and remove in $\mathcal{O}(log(n))$
- equals und compareTo müssen konsistent sein
  $equals \Leftrightarrow compareTo == 0$
- Map.Entry Paar (von firstEntry, lastEntry, ...) ist Snapshot, Entry.setValue ist nicht möglich

mercateo

TreeMap - Zugriffszeiten

Operation	Laufzeit
add	$\mathcal{O}(TODO)$
contains	$\mathcal{O}(TODO)$
containsKey	$\mathcal{O}(log(n))$
remove	$\mathcal{O}(log(n))$
get	$\mathcal{O}(log(n))$
put	$\mathcal{O}(log(n))$

mercateo

TreeMap - Wann nehmen?

wenn man eine sortierte / navigierbare Map braucht, die nicht thread-safe ist

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)

  einfügen, suchen, löschen in $\mathcal{O}(log(n))$

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList
  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)
  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList
  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)
  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe
- weakly consistent iterators

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList
  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)
  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe
- weakly consistent iterators
- aufsteigende Iteration ist schneller als absteigende

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)

  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe
- weakly consistent iterators
- aufsteigende Iteration ist schneller als absteigende
- Map.Entry Methoden unterstützen kein Entry.setValue (auch entrySet nicht)

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)

  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe
- weakly consistent iterators
- aufsteigende Iteration ist schneller als absteigende
- Map.Entry Methoden unterstützen kein Entry.setValue (auch entrySet nicht)
- size-Methode ist nicht in konstanter Zeit: Iteration über alle Element und ein valid-Check

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)

  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe
- weakly consistent iterators
- aufsteigende Iteration ist schneller als absteigende
- Map.Entry Methoden unterstützen kein Entry.setValue (auch entrySet nicht)
- size-Methode ist nicht in konstanter Zeit: Iteration über alle Element und ein valid-Check
- putAll, equals, toArray, containsValue, und clear sind nicht zwingend atomar

mercateo

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists Wikipedia SkipList

  entspricht in etwa einer binären Suche bei der Suche können Elemente aufgrund Verlinkung übersprungen werden (skip)

  einfügen, suchen, löschen in $\mathcal{O}(log(n))$
- thread-safe
- weakly consistent iterators
- aufsteigende Iteration ist schneller als absteigende
- Map.Entry Methoden unterstützen kein Entry.setValue (auch entrySet nicht)
- size-Methode ist nicht in konstanter Zeit: Iteration über alle Element und ein valid-Check
- putAll, equals, toArray, containsValue, und clear sind nicht zwingend atomar
- weder null key noch null value

mercateo

ConcurrentSkipListMap - Wann nehmen?

wenn man eine sortierte / navigierbare Map braucht, die thread-safe ist und man keine null-Keys/Values hat

mercateo

Map - Übersicht

	thread-safe	Iterator	Reihenfolge	nulls
HashMap	nein	fail-fast		erlaubt
LinkedHashMap	nein	fail-fast	insertion / access	erlaubt
IdentityHashMap	nein	fail-fast		erlaubt
WeakHashMap	nein	fail-fast		erlaubt
ConcurrentHashMap	ja	fail-safe		weder key noch value
Hashtable	ja	fail-fast		weder key noch value
Properties	ja	fail-fast		weder key noch value
EnumMap	nein	weakly	wie das Enum	values
TreeMap	nein	fail-fast	natural / Comparator	values oder Comparator
ConcurrentSkipListMap	ja	weakly	natural / Comparator	weder key noch value

mercateo

java.util.Queue

mercateo

java.util.Queue

Queues

mercateo

java.util.Set

mercateo

java.util.Set

Sets

Hilfsfunktionen

Hilfsfunktionen für Listen

java.util.Collections.emptyList()

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

mercateo

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");
```

mercateo

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);
```

mercateo

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");
```

mercateo

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);
```

mercateo

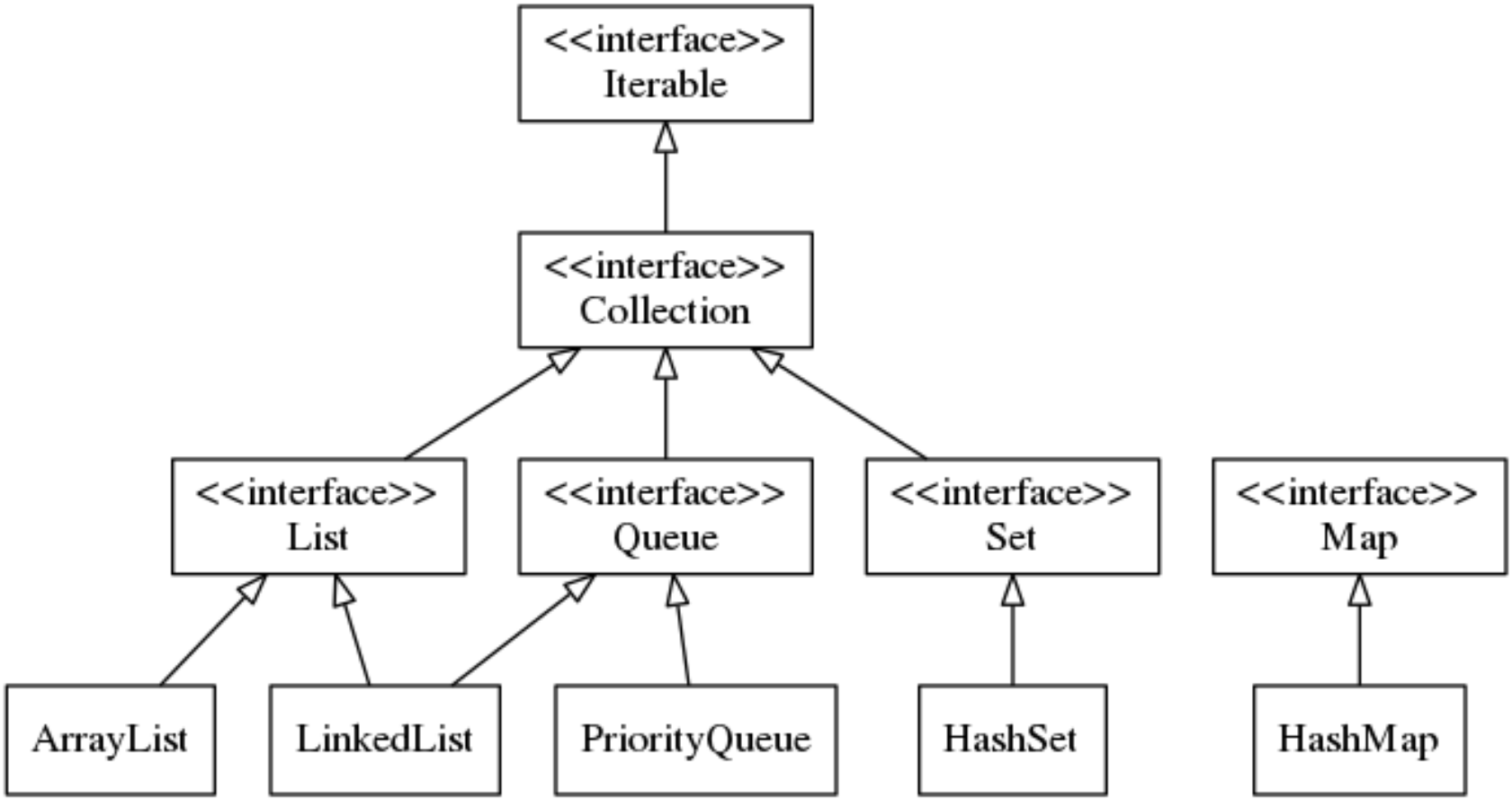
Abschluss

Abschluss

Warum das alles

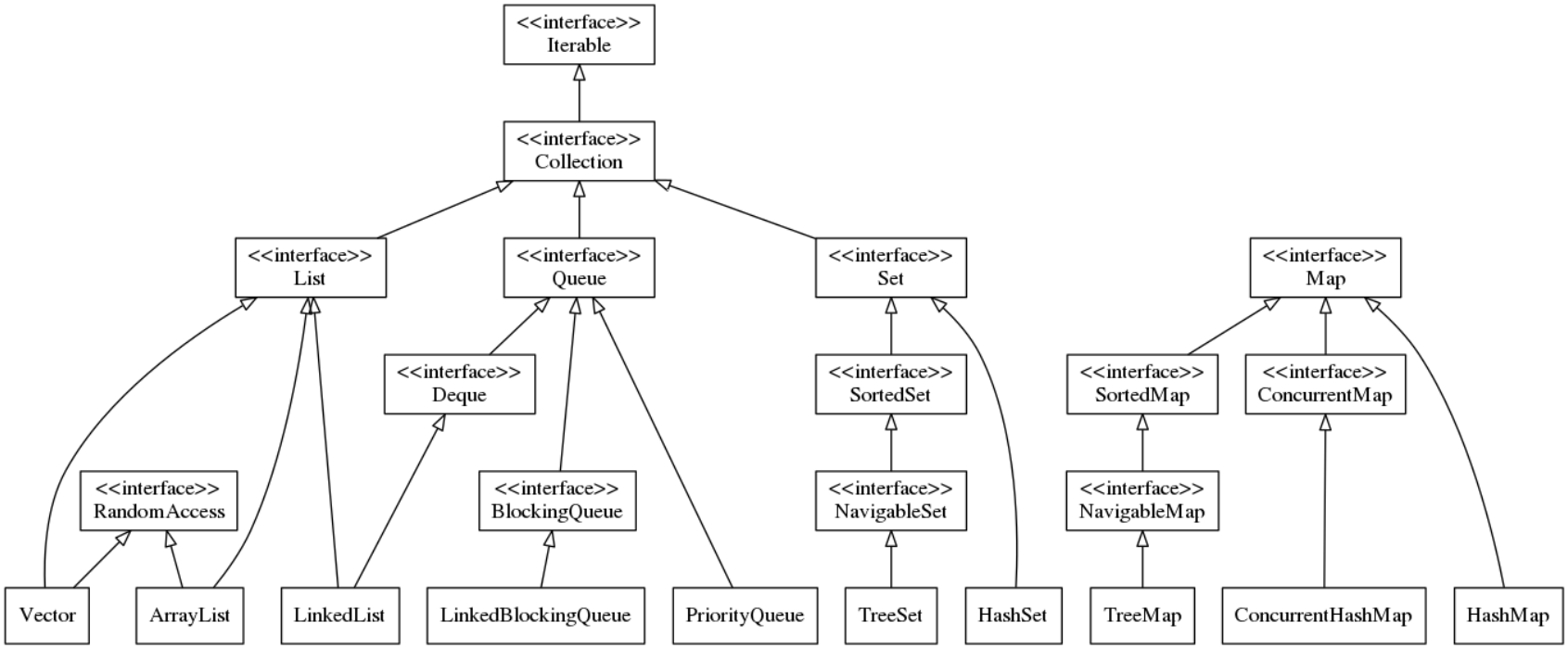
mercateo

Stark gekürzte Übersicht



mercateo

Leicht gekürzte Übersicht



mercateo