

Datenstrukturen in Java

Datenstrukturen und Algorithmen im JDK

Andreas Klipp,
Stephan Prätsch

29. April 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

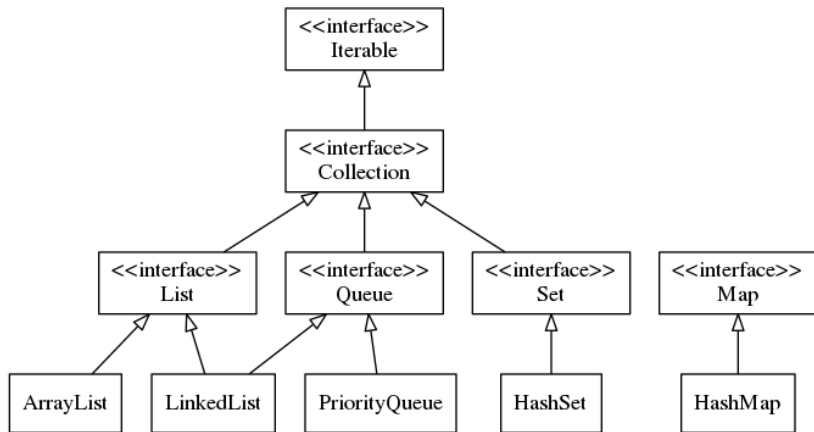
Warum das alles?

- 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

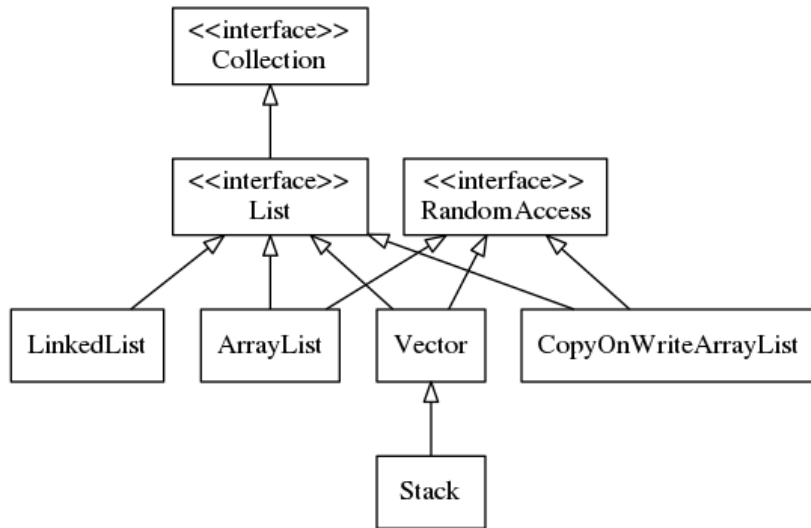
Übersicht der **vorhandenen** Datenstrukturen im JDK und deren **Besonderheiten**.

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
- "snapshot 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
- "snapshot 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

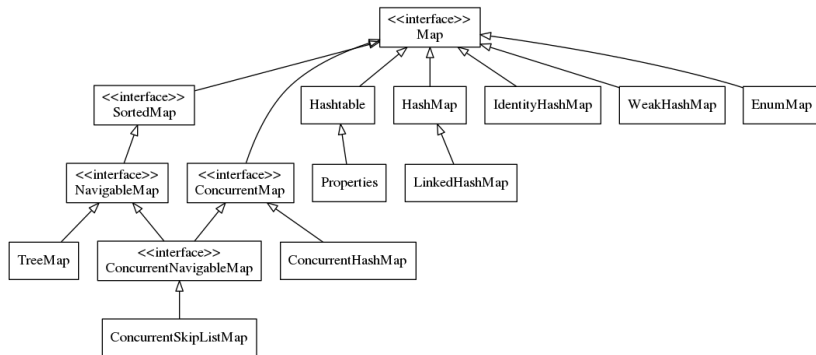
⇒ Tendenz ArrayList oder Collections.synchronizedList

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

java.util.Map



HashMap

- seit 1.2

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

HashMap

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

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

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

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

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

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`

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

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

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

HashMap - Zugriffszeiten

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

HashMap - Wann nehmen?

Synchronisation egal,

HashMap - Wann nehmen?

Synchronisation egal,
Ordnung egal,

HashMap - Wann nehmen?

Synchronisation egal,
Ordnung egal,
oft get/put

LinkedHashMap

- null erlaubt

LinkedHashMap

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

LinkedHashMap

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

LinkedHashMap

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

LinkedHashMap

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

LinkedHashMap

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

LinkedHashMap

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

LinkedHashMap - LRU-Cache

Least Recently Used

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

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;
}
```

LinkedHashMap - Zugriffszeiten

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

LinkedHashMap - Wann nehmen?

Reihenfolge wichtig oder

LinkedHashMap - Wann nehmen?

Reihenfolge wichtig oder
schnelle Iteration (schneller als HashMap)

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`)

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

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"

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

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 $O(1)$

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

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

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

IdentityHashMap - Wann nehmen?

nur wenn Referenz-Gleichheit gebraucht wird (sehr selten)

WeakHashMap

- weak keys (als weak reference)

WeakHashMap

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

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

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
- ⇒ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't

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
- ⇒ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't
- ⇒ workaround: values in WeakReference einpacken

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
- ⇒ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't
- ⇒ workaround: values in WeakReference einpacken
- null key, null value supported

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
- ⇒ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't
- ⇒ workaround: values in WeakReference einpacken
- null key, null value supported
 - Effizienz wie HashMap

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
- ⇒ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't
- ⇒ workaround: values in WeakReference einpacken
- null key, null value supported
 - Effizienz wie HashMap
 - nicht thread-safe

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
- ⇒ values sollten nicht (in)direkt auf ihre keys verweisen, sonst werden sie nicht GC't
- ⇒ workaround: values in WeakReference einpacken
- null key, null value supported
 - Effizienz wie HashMap
 - nicht thread-safe
 - fail-fast iterator

WeakHashMap - Wann nehmen?

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

WeakHashMap - Wann nehmen?

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

ConcurrentHashMap

- ist keine spezielle HashMap, wie LinkedHashMap

ConcurrentHashMap

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

ConcurrentHashMap

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

ConcurrentHashMap

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

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

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

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)

ConcurrentHashMap - Wann nehmen?

wenn eine thread-safe Map benötigt wird oder

ConcurrentHashMap - Wann nehmen?

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

ConcurrentHashMap - Wann nehmen?

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

Hashtable

- fail-fast iterator

Hashtable

- fail-fast iterator
- thread-safe

Hashtable

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

Hashtable

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

Hashtable

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

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)

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

Hashtable - Wann nehmen?

Gar nicht mehr

Hashtable - Wann nehmen?

Gar nicht mehr

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

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

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)

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

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

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

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

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)

Properties - Wann nehmen?

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

Alternative: `java.util.prefs.Preferences`

EnumMap

- keys = enum values

EnumMap

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

EnumMap

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

EnumMap

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

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

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

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

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 $\mathcal{O}(1)$

EnumMap - Wann nehmen?

wenn keys enums sind

TreeMap

- fail-fast iterator

TreeMap

- fail-fast iterator
- nicht thread-safe

TreeMap

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

TreeMap

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

TreeMap

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

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 $\mathcal{O}(\log(n))$

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

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

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	$\mathcal{O}(\log(n))$

TreeMap - Wann nehmen?

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

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator

ConcurrentSkipListMap

- Reihenfolge: Natürliche Ordnung oder Comparator
- concurrent variant von SkipLists [Wikipedia SkipList](#)

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)

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

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

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

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)

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

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

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

ConcurrentSkipListMap - Wann nehmen?

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

Map - Übersicht

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

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>)

erzeugt eine synchronisierte List

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

Hilfsfunktionen für Maps

`java.util.Collections.synchronizedMap(Map<K, V>)`

erzeugt eine synchronisierte Map

```
Map<String, String> m = ...  
Map<String, String> s = Collections.synchronizedMap(m);
```

Hilfsfunktionen für Maps

`java.util.Collections.synchronizedMap(Map<K, V>)`

erzeugt eine synchronisierte Map

```
Map<String, String> m = ...  
Map<String, String> s = Collections.synchronizedMap(m);
```

`java.util.Collections.unmodifiableMap(Map<K, V>)`

```
Map<String, String> unmodifiableMap = Collections.unmodifiableMap(map);  
unmodifiableMap.put("a", "b"); // UnsupportedOperationException
```

Hilfsfunktionen für Maps

`java.util.Collections.synchronizedMap(Map<K, V>)`

erzeugt eine synchronisierte Map

```
Map<String, String> m = ...  
Map<String, String> s = Collections.synchronizedMap(m);
```

`java.util.Collections.unmodifiableMap(Map<K, V>)`

```
Map<String, String> unmodifiableMap = Collections.unmodifiableMap(map);  
unmodifiableMap.put("a", "b"); // UnsupportedOperationException
```

`java.util.Collections.emptyMap()`

unmodifizierbare, leere Map

```
Map<String, String> empty = Collections.emptyMap();  
empty.put("a", "b"); // UnsupportedOperationException
```

Hilfsfunktionen für Maps

`java.util.Collections.singletonMap(K, V)`

unmodifizierbare, ein-elementige Map

```
Map<String, String> single = Collections.singletonMap("a", "b");  
single.put("a", "b");// UnsupportedOperationException
```


Hilfsfunktionen für Maps

`java.util.Map.computeIfAbsent(K, Function<? super K, ? extends V>)`

füge in eine existierende Liste ein, ggf. erzeuge diese Liste

```
void addOld(Map<String, List<String>> map, String key, String value) {  
    List<String> list = map.get(key);  
    if (list == null) {  
        list = new ArrayList<>();  
        map.put(key, list);  
    }  
    list.add(value);  
}
```

Hilfsfunktionen für Maps

`java.util.Map.computeIfAbsent(K, Function<? super K, ? extends V>)`

füge in eine existierende Liste ein, ggf. erzeuge diese Liste

```
void addOld(Map<String, List<String>> map, String key, String value) {  
    List<String> list = map.get(key);  
    if (list == null) {  
        list = new ArrayList<>();  
        map.put(key, list);  
    }  
    list.add(value);  
}
```

```
void addNew(Map<String, List<String>> map, String key, String value) {  
    map.computeIfAbsent(key, x -> new ArrayList<>()).add(value);  
}
```

Hilfsfunktionen für Maps

java.util.Map.getDefault(Object, V)

default value in Map

```
// alt
Map<String, String> map = new HashMap<>();
String s = map.get("a");
if (s == null) {
    s = "default";
}
System.out.println(s);
```

Hilfsfunktionen für Maps

java.util.Map.getDefault(Object, V)

default value in Map

```
// alt
Map<String, String> map = new HashMap<>();
String s = map.get("a");
if (s == null) {
    s = "default";
}
System.out.println(s);

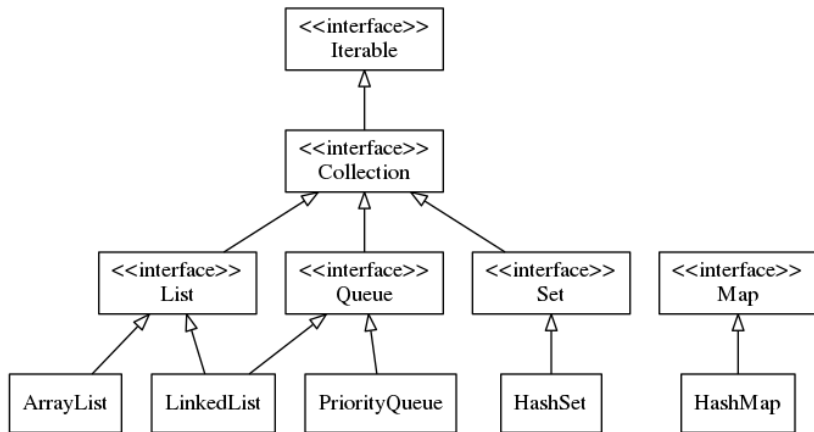
Map<String, String> map = new HashMap<>();
map.put("a", "a");
System.out.println(map.getDefault("a", "default"));
System.out.println(map.getDefault("b", "default"));
```

Abschluss

Abschluss

Warum das alles

Stark gekürzte Übersicht



Leicht gekürzte Übersicht

