COLLECTIONS IN JAVA

SEW S01

29. OKTOBER 2014

STEFAN ERCEG
4AHITT

Inhalt

1. Beschreibung von Collections

Collections, auch häufig Container genannt, kennzeichnen Datenstrukturen, die eine Gruppe von Daten bzw. Objekten zu einer Einheit zusammenfassen. Beispiele dafür wären eine Schulklasse (Gruppe von Schülern), ein Postfach (Gruppe von E-Mail-Adressen) oder ein Telefonverzeichnis (Gruppe von Name und Telefonnummer-Paaren). Java bietet dafür Container an, welche bestimmte Strukturen beherrschen, erproben und performant laufen lassen. Im Folgenden wird ein Vergleich der von Java zur Verfügung gestellten Collections aufgestellt. [1]

2. Übersicht der Interfaces

Man unterscheidet grob zwischen zwei Interfaces, nämlich den Collection und den Map Interfaces.

2.1 Collection Interfaces

Das Basisinterface „java.util.Collection“ bietet folgende Subinterfaces an (BlockingDeque, TransferQueue und BlockingDeque sind dabei Concurrent Collections):

- java.util.List
- java.util.Set
  - java.util.SortedSet
    - java.util.NavigableSet
- java.util.Queue
  - java.util.Deque
    - java.util.concurrent.BlockingDeque (Concurrent Collections)
  - java.util.concurrent.BlockingQueue (Concurrent Collections)
    - java.util.concurrent.TransferQueue (Concurrent Collections)
    - java.util.concurrent.BlockingDeque (Concurrent Collections)

[2]

2.1.1 Interface List

Das Interface List erbt die Methoden von dem Interface Collection und fügt neue hinzu. Bei einer Liste besitzen die Elemente eine festgelegte Reihenfolge, sie sind daher geordnet. In einer Liste werden doppelt vorkommende Elemente erlaubt. Die Schnittstelle schreibt Verhalten vor, welche von allen Implementierungen eingehalten werden. Jede Implementierung unterscheidet sich jedoch in ihren Eigenschaften wie z.B. der Performance, dem Speicherplatzbedarf oder der Möglichkeit der Nebenläufigkeit.
Folgende Implementierungen von List werden angeboten:

- AbstractList
- AbstractSequentialList
- ArrayList
- CopyOnWriteArrayList
- LinkedList
- Stack
- Vector

[3, 17, 18, 19, 20]

2.1.2 Interface Set

Das Interface Set ist ebenfalls ein Subinterface von dem Interface Collection und erbt somit dessen Methoden, es werden jedoch auch neue Methoden angeboten. Bei den Sets, auch Mengen genannt, handelt es sich um eine ungeordnete Sammlung von Elementen. Es werden keine doppelt vorkommenden Elemente erlaubt, d.h. jedes Element darf nur einmal vorkommen.
Folgende Implementierungs-Klassen von Set sind vorhanden:

- AbstractSet

- ConcurrentHashMap.KeySetView
- CopyOnWriteArraySet
- EnumSet
- HashSet
- LinkedHashSet

[4, 17, 19]

2.1.2.1 Interface SortedSet

Das Interface SortedSet erweitert das Interface Set und fügt einige neue Methoden hinzu. SortedSet besitzt ebenfalls ein Subinterface, welches NavigableSet heißt und der 2 Implementierungen besitzt. [5]

Wie der Name SortedSet schon verrät, bietet die Schnittstelle an, die Mengen in einer bestimmten Reihenfolge zu sortieren. [20]

Subinterface NavigableSet und dessen Implementierungen:

- java.util.NavigableSet
  - ConcurrentSkipListSet
  - TreeSet

[6]

Die Schnittstelle NavigableSet ermöglicht mit ihren Methoden zusätzlich zu einem bestimmten Element das nächstkleinere bzw. nächsthöhere auszugeben.

Seit Java 6 implementieren die Klassen ConcurrentSkipListSet und TreeSet direkt das Interface NavigableSet, welches die Schnittstelle SortedSet erweitert. [19]

2.1.3 Interface Queue

Das Interface Queue erbt vom Interface Collection. Wenn man Queue implementiert, kann man folgende Implementierungen verwenden:

- AbstractQueue
- ConcurrentLinkedQueue
- PriorityQueue

[7]

Diese Schnittstelle wird für Datenstrukturen angeboten, die nach dem FIFO (First in, First out)-Prinzip arbeiten. Beim Einfügen von Elementen wird das Ordnungskriterium der Queue entweder über die natürliche Ordnung der Elemente oder durch ein spezielles Vergleichsobjekt definiert werden. Wahlfreie Zugriffe werden nicht erlaubt. [19, 20]

2.1.3.1 Interface Deque

Bei einer Deque wird die FIFO-Verarbeitung durch die neuen hinzugefügten Methoden an beiden Enden (deque = Double Ended Queue) angeboten. [19] Das Interface Deque besitzt einige zusätzliche Implementierungen im Gegensatz zu dessen Superinterfaces Collection und Queue. Ein Subinterface

von Deque ist das Concurrent Collection Interface BlockingDeque, welches eine zusätzliche Implementierung besitzt.

- ArrayDeque
- ConcurrentLinkedDeque
- LinkedList

[8]

Subinterface BlockingDeque und dessen Implementierungen:

- java.util.concurrent.BlockingDeque
  - LinkedBlockingDeque

[10]

Die Blocking Deque ist eine Deque, welche Threads beim Einfügen oder Entfernen von Elementen der Deque blockiert. [21]

2.1.3.2 Interface BlockingQueue

Bei dem Interface BlockingQueue, dessen Subinterfaces und deren Implementierungen handelt es sich um Concurrent Collections.

Blockierende Queues bieten die Möglichkeit zu warten, sie dienen somit als Puffer. Verwendet man eine bestimmte blockierende Queue, kann man entscheiden, ob gar nicht gewartet, nur eine bestimmte Zeit gewartet oder auf eine unbestimmte Zeit gewartet werden soll. Möchte man blockierende Warteschlangen mit Prioritäten verwenden, bietet sich die Klasse „PriorityBlockingQueue“ an. [20]

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

[9]

Subinterfaces TransferQueue & BlockingDeque und deren Implementierungen:

- java.util.concurrent.TransferQueue
  - LinkedTransferQueue
- java.util.concurrent.BlockingDeque
  - LinkedBlockingDeque

[10, 11]

Transfer Queues sind im Prinzip blockierende Queues, bei welchem die Produzenten auf die Konsumenten warten müssen, um Elemente zu empfangen. Diese sind nützlich, wenn bei Message Passing Applikationen (beschreiben den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen) z.B. der Produzent auf den Empfang der Elemente vom Konsumenten wartet. [11]

Die Blocking Deque ist eine Deque, welche Threads beim Einfügen oder Entfernen von Elementen der Deque blockiert. [21] Das Interface BlockingDeque und dessen Implementierungs-Klasse LinkedBlockingDeque erben nicht nur vom Interface Deque, sondern auch von BlockingQueue. [10]

2.2 Map Interfaces

„Maps“ sind an sich keine Collections, jedoch enthalten diese Interfaces Collection-Views Operationen, daher werden sie ebenfalls als Collections angesehen. [2] Eine Map arbeitet mit einem assoziativen Speicher, d.h. ein Schlüssel wird mit einem Wert verbunden (key-value Paare). Bei den Schlüsseln dürfen dabei keine doppelten Elemente vorkommen, bei den Werten jedoch schon. [19]

Bei den Interfaces ConcurrentMap und ConcurrentNavigableMap handelt es sich um Concurrent Maps.
Folgende Subinterfaces werden von Java angeboten:

- java.util.SortedMap
  - java.util.NavigableMap
    - java.util.concurrent.ConcurrentNavigableMap (Concurrent Maps)
- java.util.concurrent.ConcurrentMap (Concurrent Maps)
  - java.util.concurrent.ConcurrentNavigableMap (Concurrent Maps)

[12]

2.2.1 Implementierungen vom Interface Map

Folgende Implementierungen vom Interface Map sind vorhanden:

- AbstractMap
- EnumMap
- HashMap
- Hashtable
- IdentityHashMap
- LinkedHashMap
- Properties
- WeakHashMap

[12]

2.2.2 Interface SortedMap

Wie der Name SortedMap schon verrät, bietet die Schnittstelle an, die Schlüssel in einer bestimmten Reihenfolge zu sortieren.

Beim Interface SortedMap ist wie bei SortedSet ebenfalls ein Subinterface vorhanden. Dieses wird NavigableMap genannt und besitzt eine Implementierung und wiederum ein Subinterface, welches eine Concurrent Map ist und eine Implementierungs-Klasse anbietet. [13]

Subinterfaces NavigableMap & ConcurrentNavigableMap und deren Implementierungen:

- java.util.NavigableMap
  - TreeMap
  - java.util.concurrent.ConcurrentNavigableMap
    - ConcurrentSkipListMap

[14, 15]

Die Schnittstelle NavigableMap ermöglicht mit ihren zusätzlichen Methoden den Zugriff auf das kleinste (Methode „firstKey()“) oder größte Element (Methode „lastKey()“). Es können ebenfalls mit Methoden wie „subMap()“ und „tailMap()“ Teilsichten des Assoziativspeichers gebildet werden.

Seit Java 6 implementieren die Klassen ConcurrentSkipListMap und TreeMap direkt das Interface NavigableMap, welches die Schnittstelle SortedMap erweitert. [19]

Das Interface ConcurrentNavigableMap bietet dasselbe an wie dessen Superinterface NavigableMap. Der einzige Unterschied ist der, dass es threadsicher ist. [20]

2.2.3 Interface ConcurrentMap

Das Interface ConcurrentMap repräsentiert, wie der Name schon verrät, eine Map, welche fähig ist, eine Nebenläufigkeit bei den Zugriffen (puts und gets) zu verwalten. Bei den Lesezugriffen (gets) muss dabei sichergestellt werden, dass vorher nebenläufige Schreibzugriffe (puts) abgeschlossen wurden. Dieses besitzt eine Implementierungs-Klasse und das Subinterface ConcurrentNavigableMap, welches ebenfalls von NavigableMap erbt.

- ConcurrentHashMap

[16]

Subinterface ConcurrentNavigableMap und dessen Implementierungen:

- java.util.concurrent.ConcurrentNavigableMap
  - ConcurrentSkipListMap

Das Interface ConcurrentNavigableMap und dessen Implementierungs-Klasse ConcurrentSkipListMap erben nicht nur vom Interface NavigableMap, sondern auch von ConcurrentMap. [15]

3. Beschreibungen der Implementierungs-Klassen von Collection

3.1 Implementierungs-Klassen von List

3.1.1 AbstractList

Wie der Name schon verrät, handelt es sich bei der Klasse AbstractList um eine abstrakte Klasse, dies bedeutet, dass von dieser Klasse kein Objekt erzeugt werden kann. Mit dieser Klasse ist es einfach, eigene List-Implementierungen auf der Basis von List-Elementen mit wahlfreiem Zugriff zu implementieren. [20, 22]

3.1.2 AbstractSequentialList

Die abstrakte Klasse AbstractSequentialList erbt von AbstractList. Diese verwendet man als Basisklasse, wenn die eigene List-Implementierung auf sequentiellem Datenzugriff, wie z.B. in einer verketteten Liste, basieren soll. Die beiden Klassen AbstractList und AbstractSequentialList dienen somit als Basis der konkreten Listen. [20, 23]

3.1.3 LinkedList

Diese Klasse erweitert die abstrakte Klasse AbstractSequentialList. Bei der LinkedList handelt es sich um eine doppelt verkettete Liste. Auf die Elemente kann man nicht über einen bestimmten Index zugreifen, da sie nacheinander abgearbeitet werden. Jedes Element, bis auf das erste und letzte Element, besitzt einen Vorgänger und einen Nachfolger. Man verwendet zum Bearbeiten einer LinkedList typischerweise einen Iterator. Damit man sich in der LinkedList in beide Richtungen bewegen kann, wird ein ListIterator verwendet, welcher die gleichen Methoden wie ein normaler Iterator besitzt.

Da LinkedList ebenfalls die Interfaces Queue und Deque implementiert, kann es dessen Methoden ebenfalls verwenden. [17, 19, 25]

3.1.4 ArrayList

Die ArrayList ähnelt von allen Collections am ehesten einem Array. Die Elemente liegen direkt nebeneinander, sind durchnummeriert und man kann auf ein Element über seinen Index zugreifen.

Ein Nachteil beim Array ist, dass man vor seiner ersten Benutzung festlegen muss, wie groß es ist. Dies kann nachträglich nicht mehr geändert werden. Die ArrayList kann man jedoch jederzeit um ein paar Elemente erweitern.

Standardmäßig besitzt eine ArrayList eine Kapazität von zehn Elementen. Benötigt man mehr Elemente, wird eine größere ArrayList angelegt und umkopiert. Dies tut die ArrayList jedoch von sich aus. [17, 26]

3.1.5 Vector

Wie beim Array, kann man beim Vector auf ein bestimmtes Element über dessen numerischen Index zugreifen.

Vector arbeitet vom Prinzip her genauso wie die ArrayList (siehe 3.1.4), der einzige Unterschied ist der, dass die Methoden des Vectors synchronisiert sind. Sie laufen daher ununterbrechbar in Umgebungen mit mehreren Threads und dies kann zu Konsequenzen in der Laufzeit führen. [17, 24]

3.1.6 Stack

Die Implementierungsklasse Stack erbt von der Klasse Vector und repräsentiert ein LIFO (Last in, First out) Stack von Objekten. Es besitzt im Gegensatz zum Vector fünf zusätzliche Methoden, welche es erlauben, einen Vektor wie einen Stack zu behandeln. [28]

3.1.7 CopyOnWriteArrayList

CopyOnWriteArrayList implementiert das Interface List wie ArrayList, Vector und Linked List, jedoch ist es eine threadsichere Collection. Diese Implementierung fertigt, wie der Name schon sagt, eine Kopie von der ArrayList an. Es wird ebenfalls beim Iterator von CopyOnWriteArrayList im Gegensatz zum Iterator von ArrayList keine Exception geworfen, wenn dieser eine bestimmte Änderung während dem Iterieren erkannt hat. [27]

3.2 Implementierungs-Klassen von Set

3.2.1 AbstractSet

AbstractSet ist eine abstrakte Implementierung von Set, welche es einfacher macht seine eigenen Implementierungen zu erstellen. AbstractSet implementiert dabei lediglich dieselben Methoden, die auch für die AbstractCollection benötigt werden. [29]

3.2.2 ConcurrentSkipListSet

Diese Implementierungsklasse erweitert die Klasse AbstractSet und implementiert die Schnittstelle NavigableSet. Es stellt eine nebenläufige Implementierung dar und die Elemente werden nach der natürlichen Ordnung sortiert. [30]

3.2.3 CopyOnWriteArraySet

Ein CopyOnWriteArraySet verwendet intern eine CopyOnWriteArrayList für alle Operationen. Somit besitzen dieselben Grundeigenschaften, wie z.B., dass CopyOnWriteArraySet ebenfalls threadsicher ist. [31]

3.2.4 EnumSet

EnumSet repräsentiert eine spezielle Implementierung vom Interface Set. Hierbei können nur Enum-Konstanten, welche vor dem Erstellen des Sets definiert wurden, verwendet werden, es dürfen daher keine anderen Objekte in einem EnumSet aufbewahrt werden. Intern arbeiten Enum Sets wie Bit Vektoren, was eine hohe Kompaktheit und Effizienz erlangt. [32]

3.2.5 HashSet

HashSet liefert von allen Implementierungsklassen von Set die beste Performance, liefert aber keine Gewähr bezüglich der Reihenfolge. Die Elemente werden in einer hash-basierten Datenstruktur verwaltet, dadurch kann man ein bestimmtes schnell finden. Möchte man die HashSet sortieren, müssen die Elemente nachträglich umkopiert und dann sortiert werden. [17, 19, 33]

3.2.6 LinkedHashSet

LinkedHashSet erweitert die Klasse HashSet, indem es die Reihenfolge der Einfügung beibehält und nach dieser sortiert. Jeder Eintrag im Set enthält eine doppelt verknüpfte Liste. [17, 34]

3.2.7 TreeSet

Bei dieser Implementierungsklasse werden die Elemente immer sortiert verwaltet. Falls ein neues Element zum TreeSet hinzugefügt wird, fügt es das Element automatisch sortiert in die Datenstruktur ein. Aus diesem Grund ist ein TreeSet etwas langsamer als ein HashSet, vor allem bei der Suche nach einem einzigen Element. [17, 35]

3.2.8 ConcurrentHashMap.KeySetView

Die ConcurrentHashMap (siehe 4.11) wird hier als Set von Keys angesehen. Diese Klasse kann nicht direkt instanziiert werden. [36]

3.3 Implementierungs-Klassen von Queue

3.3.1 AbstractQueue

Die abstrakte Klasse AbstractQueue erweitert die Klasse AbstractCollection und implementiert das Interface Queue. AbstractQueue bietet wenige Queue Operationen an, welche geeignet sind, wenn keine Null-Elemente erlaubt sind. [37]

3.3.2 ArrayBlockingQueue

Die ArrayBlockingQueue erbt von der Klasse Abstract Queue und implementiert das Interface BlockingQueue. Die Daten werden hierbei intern in einem Array abgespeichert und das vorderste Element der Queue ist jenes, welches sich schon am längsten in der Queue befindet. Neue Elemente werden am Ende der Queue abgelegt. Es kann eine festgelegte Größe für die zu speichernden Elemente festgelegt werden. [38]

3.3.3 ConcurrentLinkedQueue

Die ConcurrentLinkedQueue stellt eine ungebundene threadsichere Queue dar, welche auf verlinkte Knoten basiert. Eine ConcurrentLinkedQueue ist dann angebracht, wenn viele Threads sich den Zugriff zu einer gemeinsamen Collection teilen. Wie viele andere nebenläufige Collection-Implementierungen werden auch bei dieser Klasse keine null-Elemente erlaubt. [39]

3.3.4 DelayQueue

Unter der DelayQueue versteht man eine ungebundene BlockingQueue, welche aus verzögerten Elementen besteht. Ein Element kann erst entfernt werden, wenn die Verzögerungszeit abgelaufen ist. Auch bei dieser Queue werden keine null-Elemente zugelassen. [40]

3.3.5 LinkedBlockingQueue

Die LinkedBlockingQueue, die eine optional gebundene BlockingQueue darstellt, behält die Elemente intern in einer verknüpften Struktur (verknüpfte Knoten). Diese Struktur kann, falls gewünscht, auch eine Obergrenze besitzen. Falls keine Obergrenze definiert wurde, wird der Maximalwert von Integer als Obergrenze verwendet. LinkedQueues besitzen typischerweise eine höhere Verarbeitungsmenge als Array-basierte Queues, jedoch auch eine schlechtere Performance bei den meisten nebenläufigen Applikationen. [41]

3.3.6 LinkedBlockingDeque

Die LinkedBlockingQueue präsentiert eine optional gebundene BlockingDeque, welche ebenfalls auf verknüpften Knoten basiert. Die verknüpften Knoten werden dynamisch bis zum maximalen Kapazitätsverbrauch der Deque erstellt. [42]

3.3.7 LinkedTransferQueue

Die Implementierungsklasse LinkedTransferQueue stellt eine ungebunde TransferQueue, basiert auf verknüpften Knoten, dar. [43]

3.3.8 PriorityQueue

Bei der PriorityQueue werden die Elemente, welche nicht null sein dürfen, nach ihrer natürlichen Ordnung sortiert in einem Heap-Speicher gehalten. Bei Anfragen wird das jeweils kleinste Element geliefert. Die Klasse implementiert im Gegensatz zu den übrigen kein FIFO (First in, First out)-Verhalten. [19, 45]

3.3.9 PriorityBlockingQueue

Eine PriorityBlockingQueue arbeitet vom Prinzip her genauso wie die PriorityQueue, nur dass bei dieser Klasse ankommende Operationen blockiert werden. Diese Klasse implementiert ebenfalls wie die Klasse PriorityQueue kein FIFO (First in, First out)-Verhalten. [19, 44]

3.3.10 SynchronousQueue

SynchronousQueue repräsentiert eine blockierende Queue zum Austausch von genau einem Element. Fügt ein Thread ein Element in die Queue hinzu, muss ein anderer Thread auf dieses Element warten. Es wird daher keine Kapazität benötigt, da die Elemente direkt konsumiert und nicht zwischengelagert werden müssen. [19, 46]

3.3.11 ArrayDeque

Bei der ArrayDeque, welche das Interface Deque implementiert, werden die Daten intern in einem Array abgespeichert. ArrayDeques können beliebig groß sein, da keine Beschränkung bezüglich der Kapazität gegeben worden ist. Sie sind nicht threadsicher und null-Elemente sind auch hier verboten. Diese Klasse ist im Gegensatz zu Stack, wenn sie als Stack verwendet wird, und Linked List, wenn sie als Queue verwendet wird, schneller. [47]

3.3.12 ConcurrentLinkedDeque

Die ConcurrentLinkedDeque stellt eine ungebundene nebenläufige Queue dar, welche auf verlinkte Knoten basiert. Eine ConcurrentLinkedDeque ist dann angebracht, wenn viele Threads sich den Zugriff zu einer gemeinsamen Collection teilen. Wie viele andere nebenläufige Collection-Implementierungen werden auch bei dieser Klasse keine null-Elemente erlaubt. [48]

3.3.13 LinkedList

siehe 3.1.3

4. Beschreibungen der Implementierungs-Klassen von Map

4.1 AbstractMap

AbstractMap ist eine abstrakte Implementierung von Map, welche es einfacher macht seine eigenen Implementierungen zu erstellen. Wenn man von der Klasse AbstractMap erbt, können die Basismethoden von der Map, wie z.B. „put()“, „entrySet()“, „remove()“, „get()“ und „keySet()“, verwendet werden. [49]

4.2 EnumMap

Eine EnumMap verwendet als Schlüssel einen Enum-Typ, welcher vor dem Erstellen der Map definiert werden muss. Die Daten in einer EnumMap werden intern wie in einem Array abgespeichert, was zu einer hohen Kompaktheit und Effizienz führt. Die key-value Paare werden in der natürlichen Ordnung der Schlüssel geordnet. Es werden keine null-Schlüssel erlaubt. [50]

4.3 HashMap

Die Klasse HashMap ist dafür geeignet, viele Elemente unsortiert zu speichern und sie über die Schlüssel schnell wieder verfügbar zu machen. Das interne Hashing-Verfahren erfolgt schnell, die Sortierung der Schlüssel ändert sich jedoch über die Zeit konstant. Null-Schlüssel und null-Werte werden bei der HashMap genehmigt. [51]

4.4 Hashtable

Hashtable ist älter als die HashMap und nicht Bestandteil des Java Collection Frameworks, wurde aber hier dazugezählt, weil sie ähnliche Aufgaben erfüllt. Im Gegensatz zur HashMap ist die Hashtable synchronisiert und erlaubt keine null-Elemente für die Schlüssel und die Werte. Das Auslesen erfolgt über eine Enumeration und nicht über einen Iterator. Die Enumeration gehört ebenfalls nicht zum Java Collection Framework. [17, 52]

4.5 IdentityHashMap

Möchte man die Schlüssel nicht immer über „equals“, sondern über den Vergleichsoperator „==“ vergleichen, ist eine IdentityHashMap dafür geeignet. Dies kann sehr nützlich sein, wenn man viel mit Referenzen umgehen muss. [53]

4.6 LinkedHashMap

Da die HashMap zwar schnell ist, jedoch keine Garantie über die Sortierung liefern kann, kann eine LinkedHashMap verwendet werden. Diese garantiert die Reihenfolge des Einfügens bei einer höheren Geschwindigkeit als bei der TreeMap (siehe 4.9). [17, 54]

4.7 Properties

Die Klasse Properties stellt eine Sonderform des Assoziativspeichers dar, bei der die key-value-Paare immer den Datentyp String besitzen. Da man diese Einträge in einer Datei speichern und sie wieder auslesen kann, kann man auf diese Weise fest zusammengebundene Zeichenketten aus dem Programmtext externalisieren und die Werte auch ohne Neuübersetzung verändern. [19, 55]

4.8 WeakHashMap

Bei einer WeakHashMap können die Elemente vom Garbage Collector zurückgefordert werden, was sie sehr nützlich für die Cache- bzw. Lookupspeicherung macht. Man kann ein Objekt als Schlüssel verwenden, ohne eine feste Referenz zu dem Objekt zu erstellen. [56]

4.9 TreeMap

Die TreeMap implementiert das Interface NavigableMap, welches wiederum von der Schnittstelle SortedMap erbt. Elemente, die in eine TreeMap eingefügt werden, werden gleich sortiert eingefügt. Möchte man die Schlüssel einer TreeMap sortieren, müssen die Elemente eine natürliche Ordnung besitzen oder ein externer Comparator muss die Ordnung festlegen. [19, 20, 57]

4.10 ConcurrentSkipListMap

Die ConcurrentSkipListMap implementiert das Interface ConcurrentNavigableMap. Diese Map kann geklont werden und ist serialisiert. Die Elemente werden in der natürlichen Ordnung sortiert und es werden keine null-Schlüssel und null-Werte erlaubt. [58]

4.11 ConcurrentHashMap

Eine ConcurrentHashMap ähnelt einer HashMap sehr, nur dass hier eine interne Nebenläufigkeit gewährleistet wird. Dies bedeutet, dass Blöcke bei der Verwendung von ConcurrentHashMaps in Multithread-Applikationen nicht synchronisiert werden müssen, [59]

Referenzen

[1] K. Lipinski (2003, 2014). Collections-Framework [Online]. Available at: http://www.itwissen.info/definition/lexikon/Collections-Framework-collections framework.html [abgerufen am 08.10.2014]

[2] Oracle (1993, 2014): Java Platform SE 8 – Collections Framework Overview [Online]. Available at: http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html [abgerufen am 15.10.2014]

[3] Oracle (1993, 2014): Java Platform SE 8 API – List [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/List.html [abgerufen am 18.10.2014]

[4] Oracle (1993, 2014): Java Platform SE 8 API – Set [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Set.html [abgerufen am 18.10.2014]

[5] Oracle (1993, 2014): Java Platform SE 8 API – SortedSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html [abgerufen am 18.10.2014]

[6] Oracle (1993, 2014): Java Platform SE 8 API – NavigableSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/NavigableSet.html [abgerufen am 18.10.2014]

[7] Oracle (1993, 2014): Java Platform SE 8 API – Queue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html [abgerufen am 18.10.2014]

[8] Oracle (1993, 2014): Java Platform SE 8 API – Deque [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Deque.html [abgerufen am 18.10.2014]

[9] Oracle (1993, 2014): Java Platform SE 8 API – BlockingQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html [abgerufen am 18.10.2014]

[10] Oracle (1993, 2014): Java Platform SE 8 API – BlockingDeque [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html [abgerufen am 18.10.2014]

[11] Oracle (1993, 2014): Java Platform SE 8 API – TransferQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html [abgerufen am 18.10.2014]

[12] Oracle (1993, 2014): Java Platform SE 8 API – Map [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Map.html [abgerufen am 20.10.2014]

[13] Oracle (1993, 2014): Java Platform SE 8 API – SortedMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html [abgerufen am 20.10.2014]

[14] Oracle (1993, 2014): Java Platform SE 8 API – NavigableMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/NavigableMap.html [abgerufen am 20.10.2014]

[15] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentNavigableMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html [abgerufen am 20.10.2014]

[16] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html [abgerufen am 20.10.2014]

[17] A. Willemer (2011). Java Collection Framework [Online]. Available at: http://www.willemer.de/informatik/java/collection.htm [abgerufen am 23.10.2014]

[18] T. Horn (1998, 2007). Java Collections [Online]. Available at: http://www.torsten-horn.de/techdocs/java-collections.htm [abgerufen am 23.10.2014]

[19] C. Ullenboom (2011). Openbook „Java ist auch eine Insel“ – Kap. 13 Datenstrukturen und Algorithmen [Online]. Available at: http://openbook.galileocomputing.de/javainsel9/javainsel_13_005.htm [abgerufen am 28.10.2014]

[20] C. Heinisch, F. Müller-Hofmann, J.Goll (überarbeitete Auflage 2011). Openbook „Java als erste Programmiersprache – Vom Einsteiger zum Profi“ – Kap. 18 Collections (Seite 682 bis 740) [Online]. Available at: http://books.google.at/books?id=Lw3YfrDifosC [abgerufen am 28.10.2014]

[21] J. Jenkov (2012). Java Concurrency Util – BlockingDeque [Online]. Available at: http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html [abgerufen am 29.10.2014]

[22] Oracle (1993, 2014): Java Platform SE 8 API – AbstractList [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html [abgerufen am 29.10.2014]

[23] Oracle (1993, 2014): Java Platform SE 8 API – AbstractSequentialList [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/AbstractSequentialList.html [abgerufen am 29.10.2014]

[24] Oracle (1993, 2014): Java Platform SE 8 API – Vector [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html [abgerufen am 29.10.2014]

[25] Oracle (1993, 2014): Java Platform SE 8 API – LinkedList [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html [abgerufen am 29.10.2014]

[26] Oracle (1993, 2014): Java Platform SE 8 API – ArrayList [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html [abgerufen am 29.10.2014]

[27] J.Paul (September 2012). What is CopyOnWriteArrayList in Java [Online]. Available at: http://java67.blogspot.co.at/2012/09/what-is-copyonwritearraylist-in-java-example-vs-arraylist.html [abgerufen am 29.10.2014]

[28] Oracle (1993, 2014): Java Platform SE 8 API – Stack [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html [abgerufen am 29.10.2014]

[29] Oracle (1993, 2014): Java Platform SE 8 API – AbstractSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html [abgerufen am 29.10.2014]

[30] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentSkipListSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html [abgerufen am 29.10.2014]

[31] Oracle (1993, 2014): Java Platform SE 8 API – CopyOnWriteArraySet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html [abgerufen am 29.10.2014]

[32] Oracle (1993, 2014): Java Platform SE 8 API – EnumSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html [abgerufen am 29.10.2014]

[33] Oracle (1993, 2014): Java Platform SE 8 API – HashSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html [abgerufen am 29.10.2014]

[34] Oracle (1993, 2014): Java Platform SE 8 API – LinkedHashSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html [abgerufen am 29.10.2014]

[35] Oracle (1993, 2014): Java Platform SE 8 API – TreeSet [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html [abgerufen am 29.10.2014]

[36] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentHashMap.KeySetView [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html [abgerufen am 29.10.2014]

[37] Dev Manuals (May 2011): Java Collection Framework - AbstractQueue [Online]. Available at: http://www.devmanuals.com/tutorials/java/collections/AbstractQueue/ [abgerufen am 29.10.2014]

[38] Oracle (1993, 2014): Java Platform SE 8 API – ArrayBlockingQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html [abgerufen am 29.10.2014]

[39] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentLinkedQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [abgerufen am 29.10.2014]

[40] Oracle (1993, 2014): Java Platform SE 8 API – DelayQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html [abgerufen am 29.10.2014]

[41] Oracle (1993, 2014): Java Platform SE 8 API – LinkedBlockingQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html [abgerufen am 29.10.2014]

[42] Oracle (1993, 2014): Java Platform SE 8 API – LinkedBlockingDeque [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html [abgerufen am 29.10.2014]

[43] Oracle (1993, 2014): Java Platform SE 8 API – LinkedTransferQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html [abgerufen am 29.10.2014]

[44] Oracle (1993, 2014): Java Platform SE 8 API – PriorityBlockingQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html [abgerufen am 29.10.2014]

[45] Oracle (1993, 2014): Java Platform SE 8 API – PriorityQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html [abgerufen am 29.10.2014]

[46] Oracle (1993, 2014): Java Platform SE 8 API – SynchronousQueue [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html [abgerufen am 29.10.2014]

[47] Oracle (1993, 2014): Java Platform SE 8 API – ArrayDeque [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html [abgerufen am 29.10.2014]

[48] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentLinkedDeque [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html [abgerufen am 29.10.2014]

[49] Oracle (1993, 2014): Java Platform SE 8 API – AbstractMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html [abgerufen am 29.10.2014]

[50] Oracle (1993, 2014): Java Platform SE 8 API – EnumMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html [abgerufen am 29.10.2014]

[51] Oracle (1993, 2014): Java Platform SE 8 API – HashMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html [abgerufen am 29.10.2014]

[52] Oracle (1993, 2014): Java Platform SE 8 API – Hashtable [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html [abgerufen am 29.10.2014]

[53] Oracle (1993, 2014): Java Platform SE 8 API – IdentityHashMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/IdentityHashMap.html [abgerufen am 29.10.2014]

[54] Oracle (1993, 2014): Java Platform SE 8 API – LinkedHashMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html [abgerufen am 29.10.2014]

[55] Oracle (1993, 2014): Java Platform SE 8 API – Properties [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/Properties.html [abgerufen am 29.10.2014]

[56] Oracle (1993, 2014): Java Platform SE 8 API – WeakHashMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html [abgerufen am 29.10.2014]

[57] Oracle (1993, 2014): Java Platform SE 8 API – TreeMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html [abgerufen am 29.10.2014]

[58] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentSkipListMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html [abgerufen am 29.10.2014]

[59] Oracle (1993, 2014): Java Platform SE 8 API – ConcurrentHashMap [Online]. Available at: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html [abgerufen am 29.10.2014]