Коллекции в Java

Статья

категория	Java (/category/java)
дата	27.06.2013
автор	vovanok (/user/vovanok)
голосов	97

Алгоритмы + Структуры данных = Программы.
Никлаус Вирт.

Введение

При написании программы очень часто возникает потребность хранить набор каких-либо объектов. Это могут быть числа, строки, объекты пользовательских классов и т.п. В данной статье я постараюсь классифицировать и описать основные классы коллекций простым языком.

екоторых читателей может возникнуть вопрос: зачем нам коллекции, если у нас есть ссивы ? В самом деле, многие используют коллекции там где нужно и не нужно. Но вают ситуации, когда необходимо например динамическое изменение размера уктуры данных, или автоматическое упорядочение структуры данных по мере бавления элементов и т.п.

В данной статье речь пойдет именно о **Java Collections Framework**, так как существуют многочисленные альтернативы:
1. **Guava**(Google Collections Library) - Библиотека добавляет несколько полезных реализаций структур данных, таких как мультимножество, мультиотображение и двунаправленное отображение. Улучшена эффективность.
2. **Trove library** - Реализация коллекций, позволяющая хранить примитивы (в Java Collections Framework примитивы хранить нельзя, только оберточные типы), что позволяет повысить эффективность работы.
3. **PCJ**(Primitive Collections for Java) - так же как и Trove предназначены для примитивных типов, что позволит повысить эффективность.
4. Наконец Вы сами можете написать собственную коллекцию (тот же связной список). Но данный подход не рекомендуется :)

Как видим, выбрать есть из чего. Но для начала необходимо освоить базовые коллекции Java которыми пользуются чаще всего. А так же некоторые сторонние библиотеки реализуют интерфейсы **Java Collections Framework** (пример Guava http://guava-libraries.googlecode.com/svn/tags/release05/javadoc/overview-tree.html (http://guava-libraries.googlecode.com/svn/tags/release05/javadoc/overview-tree.html)). То есть знание иерархии классов базовых коллекций позволит более быстро освоить сторонние библиотеки.

Базовые интерфейсы

В библиотеке коллекций Java существует два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций:

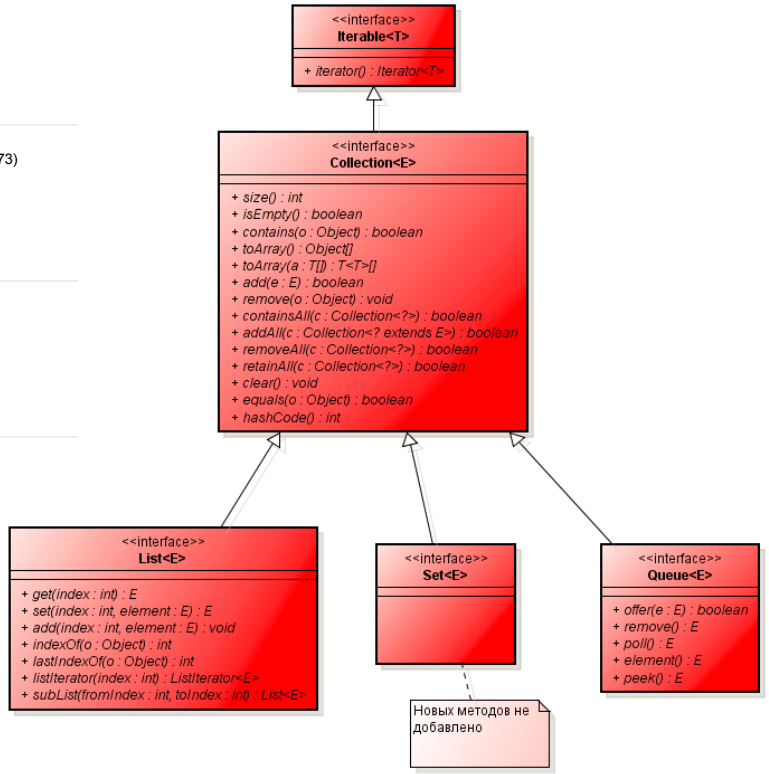
1. **Collection** - коллекция содержит набор объектов (элементов). Здесь определены основные методы для манипуляции с данными, такие как вставка (add, addAll), удаление (*remove, removeAll, clear*), поиск (*contains*)

2. **Map** - описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (тар). Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array). Никак НЕ относится к интерфейсу Collection и является самостоятельным.

Хотя фреймворк называется **Java Collections Framework**, интерфейс **map** и его реализации **входят** в фреймворк тоже !
Интерфейсы Collection и Map являются базовыми, но они не есть единственными. Их расширяют другие интерфейсы, добавляющие дополнительный функционал. О них мы ещё поговорим.

Интерфейс Collection

Давайте рассмотрим основные интерфейсы, относящиеся к **Collection**:



Как видно с диаграммы, интерфейс **Collection** не является базовым (какая интрига :D). Интерфейс **Collection** расширяет интерфейс **Iterable**, у которого есть только один метод iterator(). Это значит что любая коллекция, которая есть наследником **Iterable** должна возвращать итератор.

Итератор(http://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80 (http://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80)) - объект, абстрагирующийся за единым интерфейсом доступ к элементам коллекции. Итератор это паттерн позволяющий получить доступ к элементам любой коллекции без вникания в суть ее реализации.

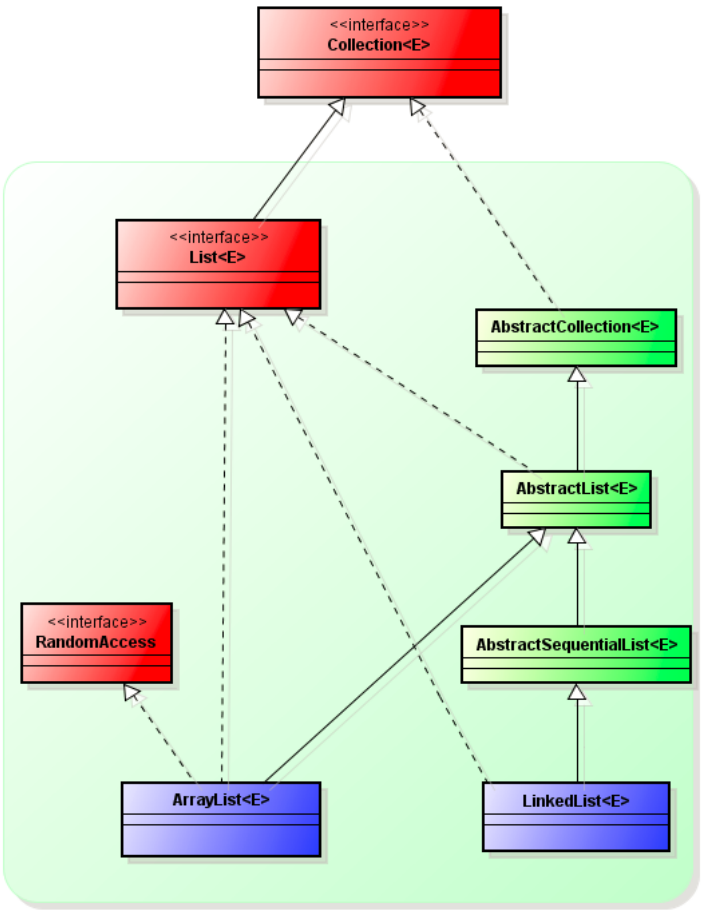
Идем дальше. Как видим с рисунка, интерфейс **Collection** расширяют интерфейсы **List, Set** и **Queue**. Давайте рассмотрим, зачем нужен каждый.

1. **List** - Представляет собой неупорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.
2. **Set** - описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).
3. **Queue** - очередь. Сразу запоминаем как правильно произносится: Queue - КЬЮ (http://www.youtube.com/watch?feature=player_embedded&v=ugauQ769kVc#at=22 (http://www.youtube.com/watch?feature=player_embedded&v=ugauQ769kVc#at=22)). Это

коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Реализации интерфейса List

Сразу смотрим на иерархию классов.



Красным здесь выделены интерфейсы, зеленым - абстрактные классы, а синим готовые реализации. Сразу хочу заметить что здесь не вся иерархия, а только основная её часть.

Как видим на рисунке, между интерфейсом и конкретной реализацией коллекции существует несколько абстрактных классов. Это сделано для того, что бы вынести общий функционал в абстрактный класс, таким образом реализовать повторное использование кода.

ArrayList - пожалуй самая часто используемая коллекция. ArrayList инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.
Так как ArrayList использует массив, то время доступа к элементу по индексу минимально (В отличии от LinkedList). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n * 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.

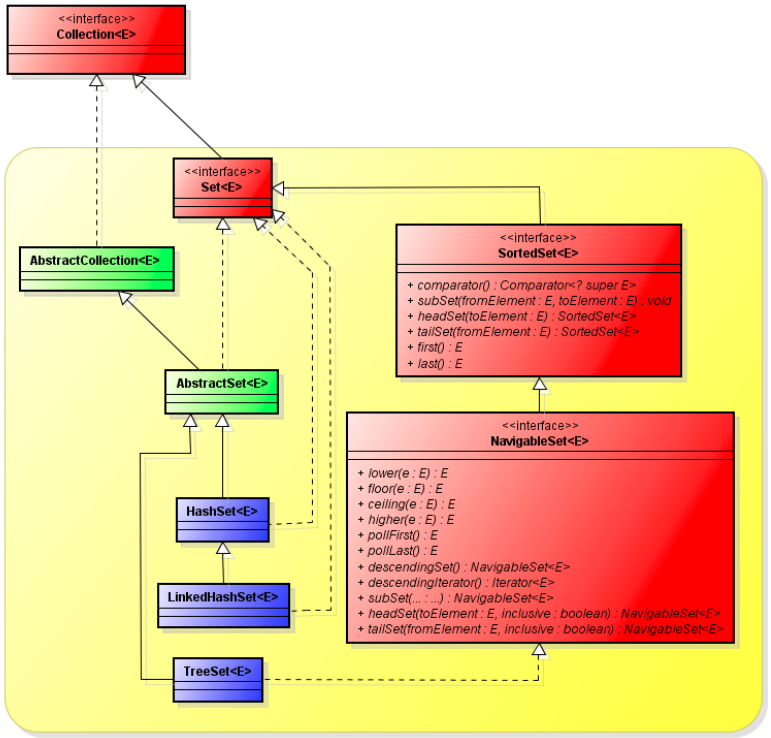
LinkedList - Двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и

последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти и по скорости выполнения операций.

Часто на собеседованиях спрашивают про отличия ArrayList и LinkedList. И какой когда нужно использовать. См. вопрос собеседования:
http://www.quizful.net/interview/java/54AubfnDy6Ti
(http://www.quizful.net/interview/java/54AubfnDy6Ti)

Реализации интерфейса Set

Смотрим следующую диаграмму. Пытаемся вникнуть :)



HashSet - коллекция, не позволяющая хранить одинаковые объекты(как и любой Set). **HashSet** инкапсулирует в себе объект **HashMap** (то-есть использует для хранения хэш-таблицу).
Как большинство читателей, вероятно, знают, хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемого хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически — вы никогда не увидите самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), **remove()** и **size()** , даже для больших наборов.

Если Вы хотите использовать **HashSet** *для хранения объектов СВОИХ классов, то вы ДОЛЖНЫ переопределить методы hashCode() и equals()*, иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object (который скорее-всего вернет разный хэш-код для ваших объектов).
Важно отметить, что класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть другой тип коллекций, такой как класс TreeSet.

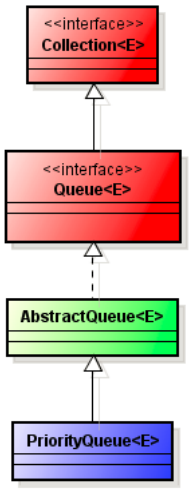
LinkedHashSet - поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с

применением итератора, элементы извлекаются в том порядке, в каком они были добавлены.

TreeSet - коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add, remove и contains потребуется гарантированное время log(n).

Реализации интерфейса Queue

Здесь я привел очень упрощенную иерархию.



PriorityQueue - единственная прямая реализация интерфейса **Queue** (не считая **LinkedList**, который больше является списком, чем очередью).
Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс **Comparable**), либо с помощью интерфейса **Comparator**, полученному в конструкторе.

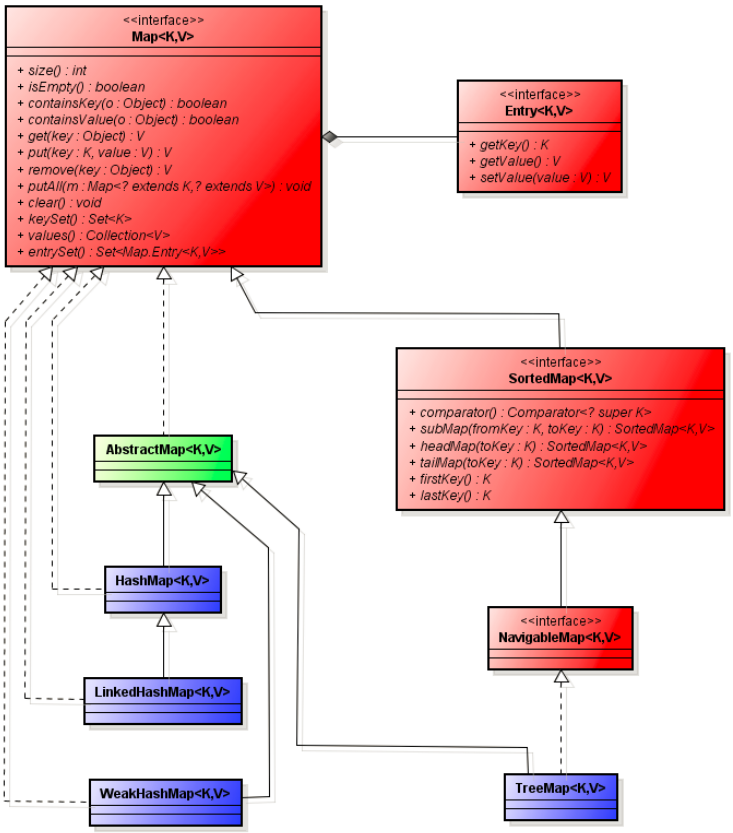
Реализации интерфейса Map

Интерфейс **Map** соотносит уникальные ключи со значениями. Ключ — это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект карты. После того как это значение сохранено, вы можете получить его по ключу. Интерфейс **Map** — это обобщенный интерфейс, объявленный так, как показано ниже.

interface Map<K, V>

Здесь K указывает тип ключей, а V — тип хранимых значений.

Иерархия классов очень похожа на иерархию Set'а:



HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени. Хорошая статья http://habrahabr.ru/post/128017/ (http://habrahabr.ru/post/128017/)

LinkedHashMap - расширяет класс **HashMap**. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса LinkedHashMap, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ.
Рекомендую так же прочитать http://habrahabr.ru/post/129037/ (http://habrahabr.ru/post/129037/)

TreeMap - расширяет класс **AbstractMap** и реализует интерфейс **NavigatebleMap**. Он создает коллекцию, которая для хранения элементов применяет *дерево*. *Объекты сохраняются в отсортированном порядке по возрастанию*. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.
Моя статья про TreeMap http://www.quizful.net/post/Java-TreeMap (http://www.quizful.net/post/Java-TreeMap)

WeakHashMap - коллекция, использующая слабые ссылки для ключей (а не значений). Слабая ссылка *(англ. weak reference)* — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению. Ссылки, не являющиеся слабыми, также иногда именуют «сильными». http://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%B0%D0%B1%D0%B0%D1%8F_%D1%81%D1%81%D1%8B%D0%BB%D0%BA%D0%E (http://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%B0%D0%B1%D0%B0%D1%8F_%D1%81%D1%81%D1%8B%D0%BB%D0%BA%D0%

Устаревшие коллекции

Следующие коллекции являются устаревшими, и их использование не рекомендуется, но не запрещается.

1. **Enumeration** — аналог интерфейса Iterator.

2. **Vector** — аналог класса ArrayList; поддерживает упорядоченный список элементов, хранимых во "внутреннем" массиве.

3. **Stack** — класс, производный от Vector, в который добавлены методы вталкивания (push) и выталкивания (pop) элементов, так что список может трактоваться в терминах, принятых для описания структуры данных стека (stack).

4. **Dictionary** — аналог интерфейса Map, хотя представляет собой абстрактный класс, а не интерфейс.

5. **Hashtable** — аналог HashMap.

Все методы Hashtable, Stack, Vector являются синхронизированными, что делает их менее эффективными в одно поточных приложениях.

Синхронизированные коллекции

Получить синхронизированные объекты коллекций можно с помощью статических методов **synchronizedMap** и **synchronizedList** класса **Collections**.

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
```

Синхронизированные обрамления коллекций synchronizedMap и synchronizedList иногда называют условно потоко безопасными - все операции в отдельности потоко безопасны, но последовательности операций, где управляющий поток зависит от результатов предыдущих операций, могут быть причиной конкуренции за данные. (источник http://www.ibm.com/developerworks/ru/library/j-jtp07233/ (http://www.ibm.com/developerworks/ru/library/j-jtp07233/))
Условная безопасность потоков, обеспечиваемая synchronizedList и synchronizedMap представляет скрытую угрозу - разработчики полагают, что, раз эти коллекции синхронизированы, значит, они полностью потоко безопасны, и пренебрегают должной синхронизацией составных операций. В результате, хотя эти программы и работают при лёгкой нагрузке, но при серьёзной нагрузке они могут начать выкидывать NullPointerException или ConcurrentModificationException.

Кроме того всегда существует возможность "классической" синхронизации с помощью блока synchronized.

Собираем все воедино

Итак, смотрим на получившейся диаграмму классов:

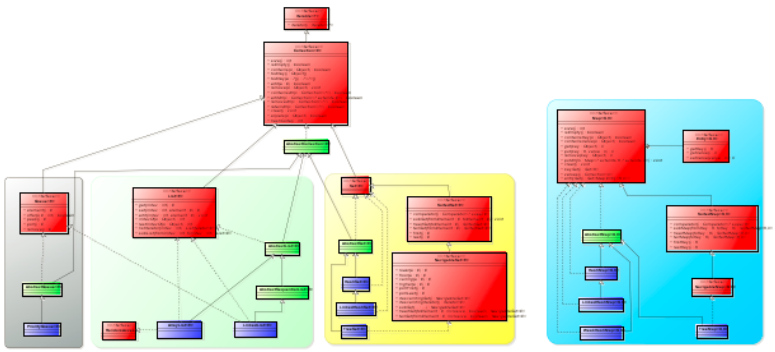


Рис 7
Большая картинка: http://piccy.info/view3/4760074/fd5ec046ce4336b8003475b57e56e02b/ (http://piccy.info/view3/4760074/fd5ec046ce4336b8003475b57e56e02b/)
Как видим диаграмма достаточно массивная. Но такая архитектура считается эталонной в ООП.

Заключение

Надеюсь эта статья была вам полезной. Если в комментариях наберётся достаточно пожеланий, я напишу вторую часть статьи, где приведу примеры использования всех

этих коллекций. (Представьте только: на собеседовании вас спрашивают про иерархию коллекций в java, а Вы им рисуете предыдущий рисунок. Как они будут удивлены :D)
Спасибо за внимание !!!

Также для закрепления знаний рекомендую пройти тесты:
Тест знаний Java - Основы (http://www.quizful.net/test/java_se_basic)
Тест знаний Java - Средний уровень (http://www.quizful.net/test/java_se_midlevel)

Если Вам понравилась статья, проголосуйте за нее

Голосов: 97 Голосовать

Комментариев: 25 ↑ () обновить ()

barban 18.03.2015 | 20:21:53 #

а почему нигде нет упоминания о Vector-е?

ответить

T1bald 18.01.2015 | 18:52:06 #

Очень не хватает примеров применения коллекций!

ответить

ShulgaYuriy 05.01.2015 | 16:51:36 #

Благодарю за статью! И Шилдт так доходчиво не написал бы)

ответить

ProgerGeek 14.11.2014 | 14:09:21 #

"HashSet инкапсулирует в себе объект HashMap" - удивился, когда прочел, полез в официальную доку, ничего подобного не нашел, как то странно,как по мне, HashSet через HashMap реализовывать, откуда информация?

ответить

() **vovanok (/user/vovanok)** 14.11.2014 | 15:15:25 #

Информация взята из исходного кода :) Смотрите конструктор.

ответить

() **demigod** 10.09.2014 | 15:16:36 #

"1. List - Представляет собой неупорядоченную коллекцию..." немного не верно. List представляет собой упорядоченную, но не отсортированную коллекцию. Согласитесь - упорядоченность и отсортированность разные вещи. List сохраняет порядок добавления элементов.

ответить

mzilenas 24.03.2014 | 13:02:53 #

"TreeMap - расширяет класс AbstractMap и реализует интерфейс NavigatebleMap." а должно буть NavigableMap .

ответить

paulart 11.01.2014 | 19:37:16 #

Спасибо, очень полезные схемы!
Попробовал сделать вариант для части схемы, чтобы меньше было пересечений и наглядней структура:
http://javeine.blogspot.com/2014/01/public-interface-collection.html
(http://javeine.blogspot.com/2014/01/public-interface-collection.html)

ответить

panser (/user/panser) 05.01.2014 | 18:32:24 #

отличная. за вторую!

ответить

miroshnik1993 11.11.2013 | 04:39:42 #

Спасибо за статью. Очень хотелось бы вторую часть)

ответить

Iggor 14.10.2013 | 12:13:28 #

я тоже за 2 часть

ответить