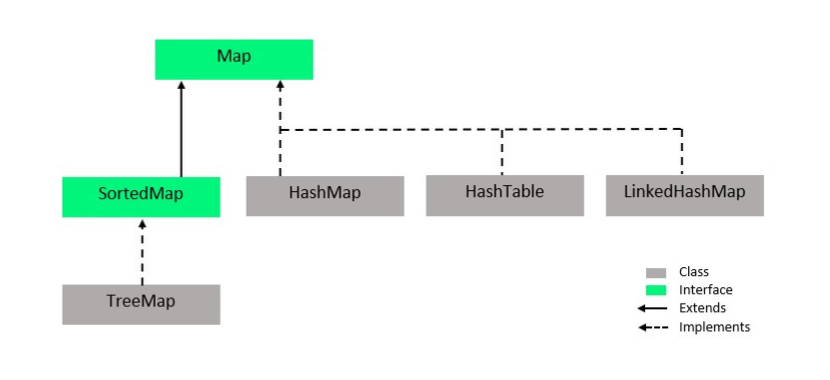
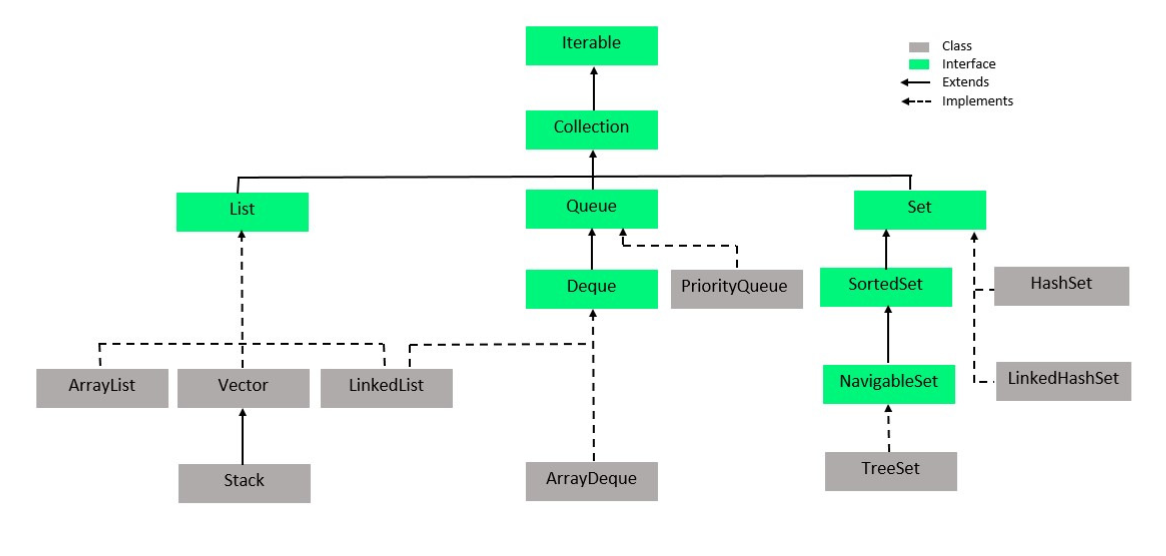
1. Дайте определение понятию “коллекция”.  
2. Назовите преимущества использования коллекций.  
3. Какие данные могут хранить коллекции?  
4. Какова иерархия коллекций?  
5. Что вы знаете о коллекциях типа List?  
6. Что вы знаете о коллекциях типа Set?  
7. Что вы знаете о коллекциях типа Queue?  
8. Что вы знаете о коллекциях типа Map, в чем их принципиальное отличие?  
9. Назовите основные реализации List, Set, Map.  
10. Какие реализации SortedSet вы знаете и в чем их особенность?  
11. В чем отличия/сходства List и Set?  
12. Что разного/общего у классов ArrayList и LinkedList, когда лучше использовать ArrayList, а когда LinkedList?  
13. В каких случаях разумно использовать массив, а не ArrayList?  
14. Чем отличается ArrayList от Vector?  
15. Что вы знаете о реализации классов HashSet и TreeSet?  
16. Чем отличаются HashMap и TreeMap? Как они устроены и работают? Что со временем доступа к объектам, какие зависимости?  
17. Что такое Hashtable, чем она отличается от HashMap? На сегодняшний день она deprecated, как все-таки использовать нужную функциональность?  
18. Что будет, если в Map положить два значения с одинаковым ключом?  
19. Как задается порядок следования объектов в коллекции, как отсортировать коллекцию?  
20. Дайте определение понятию “итератор”.  
21. Какую функциональность представляет класс Collections?  
22. Как получить не модифицируемую коллекцию?  
23. Какие коллекции синхронизированы?  
24. Как получить синхронизированную коллекцию из не синхронизированной?  
25. Как получить коллекцию только для чтения?  
26. Почему Map не наследуется от Collection?  
27. В чем разница между Iterator и Enumeration?  
28. Как реализован цикл foreach?  
29. Почему нет метода iterator.add() чтобы добавить элементы в коллекцию?  
30. Почему в классе iterator нет метода для получения следующего элемента без передвижения курсора?  
31. В чем разница между Iterator и ListIterator?  
32. Какие есть способы перебора всех элементов List?  
33. В чем разница между fail-safe и fail-fast свойствами?  
34. Что делать, чтобы не возникло исключение ConcurrentModificationException?  
35. Что такое стек и очередь, расскажите в чем их отличия?  
36. В чем разница между интерфейсами Comparable и Comparator?  
37. Почему коллекции не наследуют интерфейсы Cloneable и Serializable?

**Что такое Java Collections Framework?**

JavaCollectionFramework — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет разработчику пользоваться большим количесвом структур данных из «коробки».  
  
Базовые понятия  
На вершине иерархии в JavaCollectionFramework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие во фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» (словари).  
  
Collection — этот интерфейс находится в составе JDK c версии 1.2 и определяет основные методы работы с простыми наборами элементов, которые будут общими для всех его реализаций (например size(), isEmpty(), add(E e) и др.). Интерфейс был слегка доработан с приходом дженериков в Java 1.5. Так же в версии Java 8 было добавлено несколько новых метода для работы с лямбдами (такие как stream(), parallelStream(), removeIf(Predicate<? super E>filter) и др.).Интерфейс Collection расширяет инрерфейс Iterable.  
  
Важно также отметить, что эти медоды были реализованы непосредственно в интерфейсе как default-методы.  
  
Map. Данный интерфейс также находится в составе JDK c версии 1.2 и предоставляет разработчику базовые методы для работы с данными вида «ключ — значение».Также как и Collection, он был дополнен дженериками в версии Java 1.5 и в версии Java 8 появились дополнительные методы для работы с лямбдами, а также методы, которые зачастую реализовались в логике приложения (getOrDefault(Objectkey, V defaultValue), putIfAbsent(K key, V value)).  
- Hashtable — реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Эта коллекция была реализована раньше, чем JavaCollectionFramework, но в последствии была включена в его состав. Как и другие коллекции из Java 1.0, Hashtable является является синхронизированной (почти все методы помечены как synchronized). Из-за этой особенности у неё имеются существенные проблемы с производительностью и, начиная с Java 1.2, в большинстве случаев рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.  
-HashMap — коллекция является альтернативой Hashtable. Двумя основными отличиями от Hashtable являются то, что HashMap не синхронизирована и HashMap позволяет использовать null как в качестве ключа, так и значения. Так же как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n). Более подробную информацию о HashMap можно почитать здесь (актуально для Java< 8).  
- LinkedHashMap — это упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами (аналогично LinkedList). Но это преимущество имеет также и недостаток — увеличение памяти, которое занимет коллекция. Более подробная информация изложена в этой статье.  
-TreeMap — реализация Map основанная на красно-чёрных деревьях. Как и LinkedHashMap является упорядоченной. По-умолчанию, коллекция сортируется по ключам с использованием принципа "naturalordering", но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, которые указывается в качестве параметра при создании объекта TreeMap.  
-WeakHashMap — реализация хэш-таблицы, которая организована с использованием weakreferences. Другими словами, GarbageCollector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элеметна нет жёстких ссылок.  
  
Интерфейс List  
Реализации этого интерфейса представляют собой упорядоченные коллекции. Кроме того, разработчику предоставляется возможность доступа к элементам коллекции «по индексу» и «по значению»[«по значению» - всё равно запускается цикл по коллекции…] (так как реализации позволяют хранить дубликаты, результатом поиска по значению будет первое найденное вхождение).  
-Vector — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличии от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог — ArrayList.  
-Stack — данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например ArrayDeque.  
-ArrayList — как и Vector является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предплагается частое обращение к элементам по индексу. Из-за особенностей реализации поиндексное обращение к элементам выполняется за константное время O(1). Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции.  
-LinkedList — ещё одна реализация List. Позволяет хранить любые данные, включая null. Особенностью реализации данной коллекции является то, что в её основе лежит двунаправленный связный список (каждый элемент имеет ссылку на предыдущий и следующий). Благодаря этому, добавление и удаление из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца за константное O(1). Так же, ввиду реализации, данную коллекцию можно использовать как стек или очередь. Для этого в ней реализованы соответсвующие методы.  
  
Интерфейс Set  
Представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Является программной моделью математического понятия «множество».  
-HashSet — реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (newObject()). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.  
-LinkedHashSet — отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.  
-TreeSet — аналогично другим классам-реализациям интерфейса Set содержит в себе объект NavigableMap, что и обуславливает его поведение. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием "natural ordering".  
  
Интерфейс Queue  
Этот интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо метотов, определённых в интерфейсаCollection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Большинство реализаций данного интерфейса находится в пакете java.util.concurrent и подробно рассматриваются в данном обзоре.  
-PriorityQueue — является единственной прямой реализацией интерфейса Queue (была добавлена, как и интерфейс Queue, в Java 1.5), не считая класса LinkedList, который так же реализует этот интерфейс, но был реализован намного раньше. Особенностью данной очереди является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.  
-ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6. Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null. Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO.  
  
**Заключение**  
JavaCollectionsFramework содержит большое количество различных структур данных, доступных в JDK «из коробки», которые в большинстве случаев покрывают все потребности при реализации логики приложения. При необходимости, разработчик может создать собственную реализацию, расширив или переопределив существующую логику, либо создав свою собственную реализацию подходящего интерфейса с нуля. Также существует некоторое количество готовых решений, которые являются альтернативой или дополнением к JavaCollectionsFramework. Наиболее популярными являются GoogleGuava и CommonsCollections.



**1. Дайте определение понятию “коллекция”.**

Коллекция – структура данных, содержащая элементы одного типа в пределах иерархии наследования этих элементов(например Number…он может содержать только наследники класса Number: [Integer,Double,Float]) с динамическим расширение размера контейнера и доступом к элементам структуры данных, зависящим от её реализации.

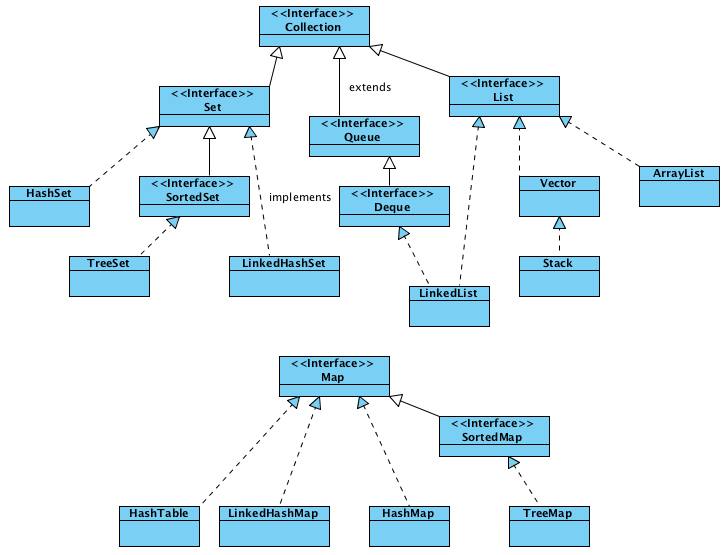
**2. Назовите преимущества использования коллекций.**

Массивы обладают значительными недостатками. Одним из них является конечный размер массива, как следствие, необходимость следить за размером массива. Другим — индексная адресация, что не всегда удобно, т.к. ограничивает возможности добавления и удаления объектов. Чтобы избавиться от этих недостатков уже несколько десятилетий программисты используют **рекурсивные типы данных**, такие как **списки** и **деревья**. Стандартный набор коллекций Java служит для избавления программиста от необходимости самостоятельно реализовывать эти типы данных и снабжает его дополнительными возможностями.

**3. Какие данные могут хранить коллекции?**

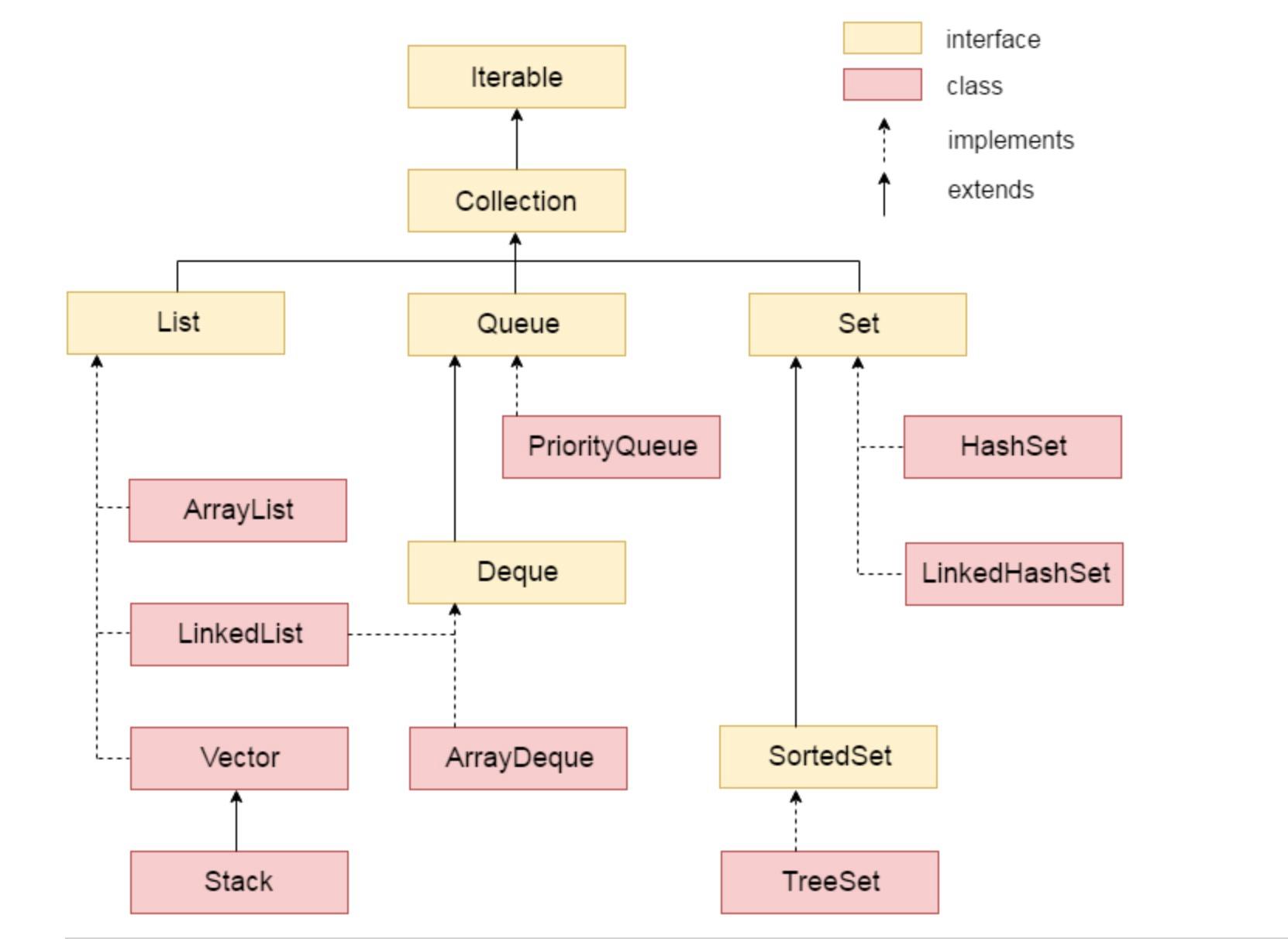
Коллекции могут хранить любые ссылочные типы данных.

**4. Какова иерархия коллекций?**

[](https://javastudy.ru/wp-content/uploads/2016/01/CollectionsHierarchy.png)

Здесь следует обратить внимание, что interfaceMap не входит в иерархию interfaceCollection.

С Java 1.6 классы TreeSet и TreeMap имплементируют интерфейсы NavigableSet и NavigableMap, которые расширяют интерфейсы SortedSet и SortedMap соответственно (SortedSet и SortedMap расширяют Set и Map).



Подробная статья про коллекции с описанием основных методов: http://www.quizful.net/post/Java-Collections

**5. Что вы знаете о коллекциях типа List?**

List — это упорядоченный список. Объекты хранятся в порядке их добавления в список. Доступ к элементам списка осуществляется по индексу.

**6. Что вы знаете о коллекциях типа Set?**

Set — множество **неповторяющихся** объектов. В коллекции этого типа разрешено наличие только одной ссылки типа null.

**7. Что вы знаете о коллекциях типа Queue?**

Queue — коллекция, предназначенная для хранения элементов в порядке добавления.

Доступ к элементам осуществляется механизмом FIFO (first-in-first-out, «первым вошел — первым вышел»).

В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Метод offer() вставляет элемент в очередь, если это не удалось — возвращает false. Этот метод отличается от метода add() интерфейса Collection тем, что метод add() может не выполнить добавление элемента только с использованием unchecked исключения.

Методы remove() и poll() удаляют верхушку очереди и возвращают ее. Какой элемент будет удален (первый или последний) зависит от реализации очереди. Методы remove() и poll() отличаются лишь поведением, когда очередь пустая: метод remove() генерирует исключение, а метод poll() возвращает null.

Методы element() и peek() возвращают (но не удаляют) верхушку очереди.

**java.util.Queue<E>** реализует FIFO–буфер. Позволяет добавлять и получать объекты. При этом объекты могут быть получены в том порядке, в котором они были добавлены.

**Реализации**: java.util.ArrayDeque<E>, java.util.LinkedList<E>.

**java.util.Deque<E>** наследует java.util.Queue<E>. Двунаправленная очередь. Позволяет добавлять и удалять объекты с двух концов. Так же может быть использован в качестве стека.

**Реализации**: java.util.ArrayDeque<E>, java.util.LinkedList<E>.

Подробнее http://www.seostella.com/ru/article/2012/08/09/kollekcii-collections-v-java-queue.html

**8. Что вы знаете о коллекциях типа Map, в чем их принципиальное отличие?**

Интерфейс java.util.Map<K,V> используется для отображения каждого элемента из множества объектов (ключей) на список объектов(значений). При этом, каждый конкретный ключ из множества связан с конкретным значением из списка.

Интерфейс java.util.Map<K,V> описывает функциональность ассоциативных массивов.

**Реализации**: java.util.HashMap<K,V>, java.util.LinkedHashMap<K,V>, java.util.TreeMap<K,V>, java.util.WeakHashMap<K,V>.

java.util.SortedMap<K,V> наследует java.util.Map<K,V>. Реализации этого интерфейса обеспечивают хранение элементов множества ключей в порядке возрастания (см. java.util.SortedSet). **Реализации**: java.util.TreeMap<K,V>.

http://developer.alexanderklimov.ru/android/java/map.php

**9. Назовите основные реализации List, Set, Map.**

|  |  |  |
| --- | --- | --- |
| Интерфейс | Класс/Реализация | Описание |
| List | ArrayList | Список |
| LinkedList | Список |
| Vector | Вектор |
| Stack | Стек |
| Set | HashSet | Множество |
| TreeSet | Множество |
| SortedSet (расширяющий интерфейс) | Отсортированное множество |
| Map | HashMap | Карта/Словарь |
| TreeMap | Карта/Словарь |
| SortedMap (расширяющий интерфейс) | Отсортированный словарь |
| Hashtable | Хеш-таблица |

**10. Какие реализации SortedSet вы знаете и в чем их особенность?**

java.util.SortedSet<E> наследует java.util.Set<E>. Реализации этого интерфейса, помимо того что следят за уникальностью хранимых объектов, поддерживают их в естественном порядке сортировки(элементы в SortedSet ДОЛЖНЫ БЫТЬ Comparable!!!!!!!). Отношение порядка между объектами может быть определено, как с помощью метода compareTo интерфейса java.lang.Comparable<T>, так и при помощи специального класса-компаратора, наследующего интерфейс java.util.Comparator<T>.

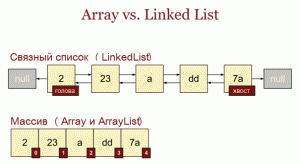
Реализации: java.util.TreeSet<E> — коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add,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

**11. В чем отличия/сходства List и Set?**

Оба унаследованы от Collection, а значит имеют одинаковый набор и сигнатуры методов. List хранит объекты в порядке вставки, элемент можно получить по индексу. Set не может хранить одинаковых элементов, не гарантирует порядок их хранения, а доступ к элементам только посредством итерации.

**12. Что разного/общего у классов ArrayList и LinkedList, когда лучше использовать ArrayList, а когда LinkedList?**

**ArrayList** реализован внутри в виде обычного массива. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Зато в нем быстро реализованы взятие и изменение элемента – операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

[](https://javastudy.ru/wp-content/uploads/2016/01/ArrayListAndLinkedList.gif)

**LinkedList** реализован внутри по-другому. Он реализован в виде связного списка: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей. А вот чтобы получить элемент с номером 130, нужно пройтись последовательно по всем объектам от 0 до 130. Другими словами операции set и get тут реализованы очень медленно.

Пример:

|  |
| --- |
|  |
|  | **Список связей. (LinkedList)** |
|  | Константно храниться только ссылка на ПЕРВЫЙ и ПОСЛЕДНИЙ элементы. |
|  | Ссылки на другие элементы получаются итеративно. |
|  |  |
|  | Добавление элемента в середину списка по индексу: |
|  | - Находится текущий элемент по индексу, при этом учитывается его отдалённость от ПЕРВОГО |
|  | или ПОСЛЕДНЕГО элемента (ближе к первому -инкрементируестя, ближе к последнему - декрементруестя). |
|  | - Берётся ссылка на предыдущий элемент"Cc" относительно текущего"Dd". |
|  | - Созаётся новый элемент(Node...тут-"S"). При этом, как предыдущий в него задаётся "Cc", а последующий - "Dd". |
|  | - a b i - не трогаются вообще. |
|  | - Тперь у элемента "Cc" ссылка на следующий объект, а у эл-та "Dd" ссылка на предыдущий ЗАМЕНЯЮТСЯ НА "S" !! |
|  |  |
|  | - При вставке НЕ ПЕРВЫМ и НЕ ПОСЛЕДНИМ ЭЛЕМЕНТОМ, изменяются лишь ссылки у 2х объектов и создаётся один новый. |
|  | Но если вставка вместо первого или последнего, то ссылка изменяется у одного объекта и создаётся один новый. |
|  | \*/ |
|  | List<String> linkedStringLists = new LinkedList<>(List.of("Aa", "Bb", "Cc", "Dd", "Ii")); |
|  | linkedStringLists.add(3, "S"); |

Посмотри на таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
| Описание | Операция | ArrayList | LinkedList |
| Взятие элемента | get | Быстро | Медленно |
| Присваивание элемента | set | Быстро | Медленно |
| Добавление элемента | add | Быстро | Быстро |
| Вставка элемента | add(i, value) | Медленно | Быстро |
| Удаление элемента | remove | Медленно | Быстро |

Если необходимо вставлять (или удалять) в середину коллекции много элементов, то лучше использовать LinkedList. Во всех остальных случаях – ArrayList.

LinkedList требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в ArrayList элементы просто идут по порядку

Из лекции javarush.ru

Структуры данных в картинках. LinkedList: http://habrahabr.ru/post/127864/

**13. В каких случаях разумно использовать массив, а не ArrayList?**

Если коротко, то Oracle пишет — используйте ArrayList вместо массивов. Если ответить на этот вопрос нужно по-другому, то можно сказать следующее: массивы могут быть быстрее и кушать меньше памяти. Списки теряют в производительности из-за возможности автоматического увеличения размера и сопутствующих проверок. Плюс к этому, что размер списка увеличивается не на 1, а на большее кол-во элементов (+15)\*. Так же доступ к [10] в массиве может быть быстрее, чем вызов get(10) у списка.

Структуры данных в картинках. ArrayList: http://habrahabr.ru/post/128269/

Еще о ArrayList на сайте http://developer.alexanderklimov.ru/android/java/arraylist.php

**14. Чем отличается ArrayList от Vector?**

Vector deprecated. У Vector некоторые методы синхронизированы и поэтому они медленные. В любом случае Vector не рекомендуется использовать вообще.

**15. Что вы знаете о реализации классов HashSet и TreeSet?**

Название Hash… происходит от понятия хэш-функция. Хэш-функция — это функция, сужающая множество значений объекта до некоторого подмножества целых чисел. Класс Object имеет метод hashCode(), который используется классом HashSet для эффективного размещения объектов, заносимых в коллекцию. В классах объектов, заносимых в HashSet, этот метод должен быть переопределен (override).

HashSet реализован на основе хеш-таблицы, а TreeSet — на основе бинарного дерева.

Подробнеео Set, HashSet, LinkedHashSet, TreeSet: http://developer.alexanderklimov.ru/android/java/set.php

HashSet гораздо быстрее чем TreeSet (константное время против логарифмического для большинства операций, таких как add, remove, contains), но TreeSet гарантирует упорядоченность объектов. Оба не синхронизированы.

HashSet

* предоставляет константное время для add(), remove(), contains() и size()
* порядок элементов в контейнере может меняться
* производительность итерации по контейнеру зависит от емкости и «коэффициента загрузки» (рекомендуется оставлять loadfactor значением по умолчанию равным 0.75, что является хорошим компромиссом между временем доступа и объемом хранимых данных)

TreeSet

* время для базовых операций add(), remove(), contains() — log(n)
* гарантирует порядок элементов
* не предоставляет каких-либо параметров для настройки производительности
* предоставляет дополнительные методы для упорядоченного списка: first(), last(), headSet(), tailSet() и т.д.

Годный ответ на StackOverflow http://stackoverflow.com/questions/1463284/hashset-vs-treeset

**16. Чем отличаются HashMap и TreeMap? Как они устроены и работают? Что со временем доступа к объектам, какие зависимости?**

В целом ответ про HashSet и TreeSet подходит и к этому вопросу.

HashMap работает строго быстрее TreeMap.

TreeMap реализован на красно-черном дереве, время добавления/поиска/удаления элемента — O(log N), где N — число элементов в TreeMap на данный момент.

У HashMap время доступа к отдельному элементу — O(1) при условии, что хэш-функция (Object.hashCode()) определена нормально (что является правдой в случае Integer).

Общая рекомендация — если не нужна упорядоченность, использовать HashMap. Исключение — ситуация с вещественными числами, которые в качестве ключей почти всегда очень плохи. Для них нужно использовать TreeMap, предварительно поставив ему компаратор, который сравнивает вещественные числа так, как это нужно в данной задаче. Например, для обычных геометрических задач два вещественных числа могут считаться равными, если отличаются не более, чем на 1e-9.

Структуры данных в картинках. HashMap: http://habrahabr.ru/post/128017/

**17. Что такое Hashtable, чем она отличается от HashMap? На сегодняшний день она deprecated, как все-таки использовать нужную функциональность?**

Некоторые методы HashTable синхронизированы, поэтому она медленнее HashMap.

* HashTable синхронизирована, а HashMap нет.
* HashTable не позволяет иметь null ключи или значения. HashMap позволяет иметь один null ключ и сколько угодно null значений.
* У HashMap есть подкласс LinkedHashMap, который добавляет возможности по итерации. Если вам нужна эта функциональность, то можно легко переключаться между классами.

Общее замечание — не рекомендуется использовать HashTable даже в многопоточных приложениях. Для этого есть ConcurrentHashMap.

http://stackoverflow.com/questions/40471/differences-between-hashmap-and-hashtable

**18. Что будет, если в Map положить два значения с одинаковым ключом?**

Последнее значение перезапишет предыдущее.

**19. Как задается порядок следования объектов в коллекции, как отсортировать коллекцию?**

Класс ТгееМар полностью реализует интерфейс SortedMap. Он реализован как бинарное дерево поиска, значит его элементы хранятся в упорядоченном виде. Это  значительно ускоряет поиск нужного элемента. Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

В этом классе четыре конструктора:

ТгееМар() — создает пустой объект с естественным  порядком элементов;  
TreeМар(Comparator с) — создает пустой объект, в котором порядок задается объектом сравнения с;  
ТгееМар(Map f) — создает объект, содержащий все элементы отображения f, с естественным порядком его элементов;  
ТгееМар(SortedMapsf) — создает объект, содержащий все элементы отображения sf, в том же порядке.

Интерфейс Comparator описывает два метода сравнения:

intcompare(Object obj1, object obj2) — возвращает отрицательное число, если obj1 в каком-то смысле меньше obj2; нуль, если они считаются равными; положительное число, если obj1 больше obj2. Для читателей, знакомых с теорией множеств, скажем, что этот метод сравнения обладает свойствами тождества, антисимметричности и транзитивности;

booleanequals(Objectobj) — сравнивает данный объект с объектом obj, возвращая true, если объекты совпадают в каком-либо смысле, заданном этим методом.

Для каждой коллекции можно реализовать эти два метода, задав конкретный способ сравнения элементов, и определить объект класса SortedMap вторым конструктором. Элементы коллекции будут автоматически отсортированы в заданном порядке.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public static void main(String[] args) {          Comparator comparator = new Comparator<String>() {  *@Override*              public int compare(String obj1, String obj2) {                  if (obj1 == null) {                      return -1;                  }                  if (obj2 == null) {                      return 1;                  }                  if (obj1.equals(obj2)) {                      return 0;                  }                  return obj1.compareTo(obj2);              }          };          TreeMap<String, String> treeMap1 = new TreeMap<>(comparator);            //or          TreeMap<Integer, String>treeMap = new TreeMap<>(new Comparator<Integer>() {  *@Override*              public int compare(Integer o1, Integer o2) {                  return Integer.compare(o1,o2);              }          });      } |

**20. Дайте определение понятию “итератор”.**

Итератор — объект, позволяющий перебирать элементы коллекции. Напримерforeach реализован с использованием итератора. Одним из ключевых методов интерфейса Collection является метод Iterator<E>iterator(). Он возвращает итератор — то есть объект, реализующий интерфейс Iterator.

Интерфейс Iterator имеет следующее определение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface Iterator <E>{      E next();      booleanhasNext();      voidremove();  } |

**21. Какую функциональность представляет класс Collections**

Некоторые из методов

|  |  |
| --- | --- |
| Сигнатура метода | Описание |
| Collections.sort(ListmyList) | Сортирует список в естественном порядке. |
| Collections.sort(List, Comparator c) | Сортировка с использованием компаратора. |
| Collections.shuffle(ListmyList) | Перемешивает коллекцию в случайном порядке. |
| Collections.reverse(ListmyList) | Переворачивает коллекцию в обратном порядке. |
| Collections.binarySearch(List mlist, T key) | поиск в коллекции по ключу с использованием бинарного поиска. |
| Collections.copy(List dest, List src) | Копирует коллекцию источник src в dest. |
| Collections.frequency(Collection c, Object o) | Возвращает число вхождений объекта в коллекции. |
| Collections.synchronizedCollection(Collection c) | Возвращает синхронизированную (потокобезопасную) коллекцию. |

**22. Как получить не модифицируемую коллекцию?**

Коллекцию, доступную только для чтения можно получить с помощью методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static <T>SortedSet<T>unmodifiableSortedSet(SortedSet<T> s) {          return new UnmodifiableSortedSet<>(s);      }    public static <T> List<T>unmodifiableList(List<? extends T> list) {      return (list instanceofRandomAccess ?              new UnmodifiableRandomAccessList<>(list) :              new UnmodifiableList<>(list));  }  ит.д. длякаждоготипа (Map, SortedMapит.п.) |

**23. Какие коллекции синхронизированы?**

Для этого используется пакет Concurrent. Атак @Deprecated HashTable, Vector.

**24. Как получить синхронизированную коллекцию из не синхронизированной?**

Используйте следующие методы:

* Collections.synchronizedList(list);
* Collections.synchronizedSet(set);
* Collections.synchronizedMap(map);

Все они принимают коллекцию в качестве параметра, и возвращают потокобезопасную коллекцию с теми же элементами внутри.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static <T> Set<T>synchronizedSet(Set<T> s) {          returnnewSynchronizedSet<>(s);      }    и т.д. для каждого типа коллекции |

**25. Как получить коллекцию только для чтения?**

Используйте следующие методы:

* Collections.unmodifiableList(list);
* Collections.unmodifiableSet(set);
* Collections.unmodifiableMap(map);

Все они принимают коллекцию в качестве параметра, и возвращают коллекцию только для чтения с теми же элементами внутри.

**26. Почему Map не наследуется от Collection?**

Они не совместимы, т.к. созданы для различных структур данных. Map использует пару ключ-значение.

**27. В чем разница между Iterator и Enumeration?**

Enumeration в два раза быстрее Iterator и использует меньше памяти. Iterator потокобезопасен, т.к. не позволяет другим потокам модифицировать коллекцию при переборе. Enumeration можно использовать только для read-only коллекций. Так же у него отсутствует метод remove();

Enumeration: hasMoreElement(), nextElement()

Iterator: hasNext(), next(), remove()

**28. Как реализован цикл foreach?**

Реализован на основе Iterator.

|  |  |
| --- | --- |
| 1 | for(тип итер\_пер : коллекция) блок\_операторов |

**29. Почему нет метода iterator.add() чтобы добавить элементы в коллекцию?**

Единственная задача итератора это перебор коллекции. Каждая коллекция имеет метод add() которым вы можете воспользоваться. Нет смысла добавлять этот метод в итератор, потому что коллекции могут быть упорядоченными и неупорядоченными, и метод add() при этом должен быть устроен по разному.

**30. Почему в классе iterator нет метода для получения следующего элемента без передвижения курсора?**

Итератор похож на указатель своими основными операциями: он указывает на отдельный элемент коллекции объектов (предоставляет *доступ к элементу*) и содержит функции для перехода к другому элементу списка (следующему или предыдущему). Контейнер, который реализует поддержку итераторов, должен предоставлять первый элемент списка, а также возможность проверить, перебраны ли все элементы контейнера (является ли итератор конечным). Таким образом без курсора просто нельзя будет реализовать безошибочное передвижение по коллекции.

**31. В чем разница между Iterator и ListIterator?**

Есть три различия:

1. Iterator может использоваться для перебора элементов Set, List и Map. В отличие от него, ListIterator может быть использован только для перебора элементов коллекции List
2. Iterator позволяет перебирать элементы только в одном направлении, при помощи метода next(). Тогда как ListIterator позволяет перебирать список в обоих направлениях, при помощи методов next() и previous()
3. При помощи ListIterator вы можете модифицировать список, добавляя/удаляя элементы с помощью методов add() и remove(). Iterator не поддерживает данную функциональность.

**32. Какие есть способы перебора всех элементов List?**

Есть 4 способа:

* Цикл с итератором
* Цикл for
* Расширенный цикл for
* Цикл while

**33. В чем разница между fail-safe и fail-fast свойствами?**

В противоположность fail-fast, итераторы fail-safe не вызывают никаких исключений при изменении структуры, потому что они работают с клоном коллекции вместо оригинала.  
Итератор коллекции CopyOnWriteArrayList и итератор представления keySet коллекции ConcurrentHashMap являются примерами итераторов fail-safe.

**34. Что делать, чтобы не возникло исключение ConcurrentModificationException?**

Первым делом, можно подобрать другой итератор, работающий по принципу fail-safe. К примеру, если вы используете List, то можете взять ListIterator. Если же вам нужна устаревшая коллекция — то используйте перечислители.  
В том случае, когда вышеизложенное вам не подходит, у вас есть три варианта:  
При использовании JDK 1.5 или выше, вам подойдут классы ConcurrentHashMap и CopyOnWriteArrayList. Это самый лучший вариант  
Вы можете преобразовать список в массив и перебирать массив  
Вы можете блокировать изменения списка на время перебора с помощью блока synchronized  
Обратите внимание, что последние два варианта негативно скажутся на производительности.

**35. Что такое стек и очередь, расскажите в чем их отличия?**

Коллекции созданы для того чтобы хранить элементы для дальнейшей обработки. Кроме базовых операций интерфейса Collection, очереди поддерживают дополнительные операции добавления, удаления и проверки состояния элемента.  
Обычно, но не обязательно очереди работают по принципу FIFO — первым пришел, первым ушел.  
Стэк — почти как очередь, но работает по принципу LIFO — последним пришел, первым ушел.  
Независимо от порядка добавления/удаления, голова очереди это элемент, который будет удален при вызове методов remove() или poll(). Также обратите внимание на то, что Stack и Vector оба потокобезопасны.

Использование: используйте очередь если вы хотите обрабатывать поток элементов в том же порядке в котором они поступают. Хорошо для списка заданий и обработки запросов.  
Используйте стэк если вы хотите класть и удалять элементы только с вершины стэка, что полезно в рекурсивных алгоритмах.

**36. В чем разница между интерфейсами Comparable и Comparator?**

В Java все коллекции, поддерживающие автоматическую сортировку, используют методы сравнения для того чтобы правильно рассортировать элементы. В качестве примера таких классов мы можем указать TreeSet, TreeMap и т.д.  
Для того чтобы рассортировать элементы, класс должен реализовать интерфейсы Comparator или Comparable. Именно поэтому классы-обертки как Integer, Double и String реализуют интерфейс Comparable.  
Интерфейс Comparable помогает сохранять естественную сортировку, тогда как Comparator позволяет сортировать элементы по разным особым шаблонам. Экземпляр компаратора обычно передается конструктору коллекции, если коллекция это поддерживает. Следует отметить, что интерфейс Comparable может быть реализован именно элементами коллекции или ключами Map, а Comparator реализуется отдельным объектом (это удобно, так как можно заготовить несколько реализаций для разных правил сортировок, не меняя при этом код элементов коллекции/ключей Map).

**37. Почему коллекции не наследуют интерфейсы Cloneable и Serializable?**

Ну, простейший ответ — «потому что не надо». Функционал предоставляемый интерфейсами Cloneable и Serializable просто не нужен для коллекций. (Тут стоит сделать исключение для ArrayList и LinkedList, которые их реализуют).

Еще одна причина — далеко не всегда нужен подкласс Cloneable потому что каждая операция клонирования потребляет очень много памяти, и неопытные программисты могут расходовать ее сами не понимая последствий.

И последняя причина — клонирование и сериализация являются очень узкоспецифичными операциями, и реализовывать их нужно только когда это необходимо. Многие классы коллекции реализуют данные интерфейсы, но совершенно незачем закладывать их для всех коллекций вообще. Если вам нужно клонирование и сериализация — просто воспользуйтесь теми классами где она есть, если нет — остальными классами.