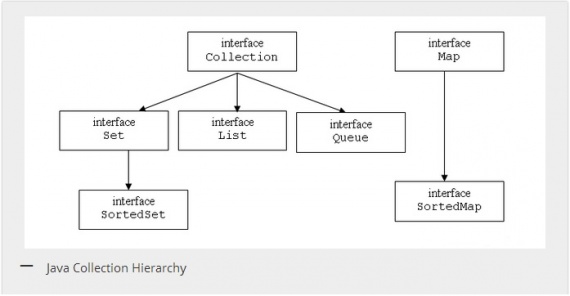
Общие вопросы

1. ***Что такое коллекции в Java? Перечислите их преимущества?***  
   По определению — коллекция это объект представляющий собой группу объектов. Как в теории множеств — множество это группа объектов. Просто, не так ли? До выхода *JDK 1.2*, существовали классы такие как *Vector и HashTable*, но не было фреймворка *Collection*. Потом было решено добавить поддержку многократно используемых структур данных. Данный фреймворк был разработан преимущественно Джошуа Блохом, и впервые появился в *JDK 1.2*.

В качестве главных преимуществ мы можем перечислить:

* + Уменьшаются затраты времени на написание кода
  + Улучшается производительность, благодаря использованию высокоэффективных алгоритмов и структур данных
  + Коллекции являются универсальным способом хранения и передачи данных, что упрощает взаимодействие разных частей кода
  + Простота в изучении, потому что необходимо выучить только самые верхние интерфейсы и поддерживаемые операции

1. ***Расскажите про иерархию коллекций?***  
     
   Как показано на картинке, фреймворк коллекций содержит один интерфейс верхнего уровня — *Collection*, от которого наследуются *Set, List и Queue*. Ниже мы рассмотрим еще множество классов, содержащихся в этих трех ветвях.  
   Запомните заголовок интерфейса *Collection*, это поможет вам с многими вопросами.

public interface Collection extends Iterable {  
//описание методов  
}

Также фреймворк содержит интерфейс *Map*, который не является наследником интерфейса *Collection*. Причину почему он не наследует *Collection*, мы разберем в четвертом вопросе.

1. ***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.  
     
   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.
2. ***Почему интерфейс Map не наследует интерфейс Collection?***  
   Хороший ответ на этот вопрос — «потому что они несовместимы». В интерфейсе *Collection* описан метод*add(Object o)*. Словари не могут содержать этот метод, потому что работают с парами ключ/значение. Также, словари имеют представления *keySet, valueSet,* которых нет в коллекциях.  
     
   В связи с этими различиями, интерфейс *Map* не может наследовать интерфейс *Collection*, и представляет собой отдельную ветвь иерархии.

Вопросы про списки

1. ***Почему мы используем списки? Какие основные классы реализуют интерфейс List?***  
   Списки в *Java* это упорядоченная коллекция элементов. Каждый элемент имеет индекс, начинающийся с нуля. Все индексы уникальны. Кроме методов описанных в интерфейсе *Collection*, списки имеют свои собственные методы, в основном для работы с элементами коллекциями по их индексу. Можно разделить эти методы на 3 группы — поиск элемента, получение конкретного элемента, перебор коллекции и выборка подгруппы. Все эти операции могут производиться по индексу элемента.  
     
   Основные классы, реализующие интерфейс *List* это *Stack, Vector, ArrayList и LinkedList*. За более подробной информацией по ним, обратитесь к документации.
2. ***Как преобразовать массив строк в ArrayList?***  
   Вопрос этот несколько глубже чем просто по программированию, как это видится новичкам. Цель его — проверить знание кандидатом служебных классов фреймворка *Collection*. Рассмотрим два таких класса, наиболее востребованных на собеседованиях — *Collections и Arrays*.  
     
   Класс *Collections* предоставляет статические методы для операций над коллекциями. Соответственно*Arrays* предоставляет статические методы для операций над массивами.

String[] words = {"аз", "буки", "веди", "глагол", "добро"};  
//Как вы можете обратить внимание, у нас есть массив строк String[] words.   
//В котором у нас лежат 5 строк.  
List wordList = Arrays.asList(words);  
//легким движением руки, а точнее вызовом Arrays.asList() мы превратили наш  
//массив строк в список List wordList.

Также хотелось бы отметить, что этот метод способен обрабатывать не только строки, он создаст список элементов любого типа, которого был массив.

Integer[] nums = {1, 2, 3, 4};  
List numList = Arrays.asList(nums);

1. ***Как отсортировать список в обратном порядке?***  
   Как и предыдущий, этот вопрос проверяет ваше знание служебных классов *Collection*

List reversedList = Collections.reverse(list);

Вопросы про множества

1. ***Почему мы используем множества? Какие основные классы реализуют интерфейс Set?***  
   Он моделирует математическое множество, из теории множеств. Интерфейс *Set* похож на *List*, но имеет некоторые отличия. Первое — это не упорядоченная коллекция. Следовательно, добавление/удаление элементов не требует их сортировки. Главная особенность множеств — уникальность элементов, то есть один и тот же элемент не может содержаться в множестве дважды.  
     
   Очень важными для функционирования множеств являются методы *equals()* и *hashCode()*, они позволяют сравнивать множества разных классов. Два множества являются идентичными только если они содержат одни и те же элементы.  
     
   Как следует из вышеизложенного, множества не поддерживают операций основанных на индексе элемента, как списки. Множества имеют только те методы которые описаны в интерфейсе *Collection.*  
     
   Основными классами, реализующими интерфейс *Set*, являются *EnumSet, HashSet, LinkedHashSet и TreeSet.* Если хотите узнать больше — почитайте соответствующие разделы документации *Java*.
2. ***Как хранятся элементы в HashSet?***  
   Как вы уже в курсе, *HashMap* хранит пары ключ/значение, и ключи должны быть уникальны. *HashSet*использует эту особенность *HashMap* для обеспечения уникальности своих элементов. В классе*HashSet*, словарь описан следующим образом:

private transient HashMap<E, Object> map;  
private static final Object PRESENT = new Object();

Итак, когда вы сохраняете элемент в множестве, оно кладет данный элемент в качестве ключа в словарь, а значением идет объект PRESENT, как это описано ниже:

public boolean add(E e) {  
        return map.put(e, PRESENT) == null;  
}

Я настоятельно рекомендую вам прочесть [эту статью](http://howtodoinjava.com/2012/10/09/how-hashmap-works-in-java/), это поможет вам с легкостью ответить на все связанные с *HashMap* вопросы.

1. ***Может ли элемент null быть добавлен в TreeSet или HashSet?***  
   Как видно из предыдущего ответа, в методе *add()* нет проверки на *null*. Также, *HashMap* позволяет один ключ *null*, следовательно, один элемент *null* может быть добавлен в *HashSet*.  
     
   *TreeSet* работает по тому же принципу что и *HashSet*, но использует *NavigableMap* для хранения элементов

private transient NavigableMap<E,Object> m;  
private static final Object PRESENT = new Object();

*NavigableMap* это класс-наследник *SortedMap*, а *SortedMap* не позволяет использование ключей *null*. Следовательно, и *TreeMap* не поддерживает хранение элементов типа *null*. Если вы попробуете добавить *null в TreeSet*, получите исключение *NullPointerException.*

Вопросы про словари

1. ***Почему мы используем словари (Map)? Какие основные классы реализуют интерфейс Map?***  
   Словари — специальный тип коллекции, которая используется для хранения пар ключ/значение. По этой причине он не является наследником интерфейса *Collection*. Словарь предоставляет методы для добавления пар ключ/значение, удаления, поиска и перебора по предоставляемым словарем представлениям данных.  
   Основные классы реализующие интерфейс *Map: HashMap, Hashtable, EnumMap, IdentityHashMap, LinkedHashMap и Properties.*
2. ***Что такое IdentityHashMap и WeakHashMap?***  
   *IdentityHashMap* похож на *HashMap* за одним исключением — для сравнения объектов используется сравнение указателей на объекты, если указатели не равны (указывают на объекты лежащие по разным адресам), значит объекты считаются различными. *IdentityHashMap* является довольно редко используемым. Хотя он реализует интерфейс *Map*, он нарушает один из основных принципов устройства*Map*, который требует использования метода *equals()* для сравнения объектов. *IdentityHashMap*используется только в тех случаях, когда требуется сравнение объектов по их адресам.  
     
   *WeakHashMap* это реализация интерфейса *Map*, которая содержит слабые ссылки на элементы. То есть, если за пределами *WeakHashMap* не осталось ни одной ссылки на его элемент, этот элемент удаляется сборщиком мусора. Класс предназначен для использования с объектами, у которых метод *equals()*проверяет идентичность объектов с помощью оператора *==*. После того как элемент будет удален сборщиком мусора, он уже не может быть восстановлен, и к большому удивлению программиста найти его в словаре больше не получится.
3. ***Объясните что такое ConcurrentHashMap? Как оно работает?***  
   *Взято с официальной документации:*  
   Реализация словаря полностью поддерживающая многопоточное добавление/удаление/поиск элементов. Данный класс следует тем же спецификациям что и Hashtable, и содержит методы соответствующие методам Hashtable. Однако, хотя все операции являются потокобезопасными, операция по выборке элементов не блокирует таблицу, и вообще нет возможности запретить весь доступ к таблице. Этот класс совместим с Hashtable во всем кроме вопросов многопоточной синхронизации.
4. ***Как работает hashmap?***  
   Самый важный вопрос, который скорее всего будет задан на собеседовании программисту любого уровня. Вы должны хорошо разбираться в этой теме, и не только потому что это самый задаваемый вопрос, но и потому что понимание устройства hashmap позволяет вам легче разобраться в других особенностях работы коллекций.  
     
   Ответ на этот вопрос очень обширный, и полностью его можно прочесть в этой статье — [как работает*hashmap*](http://howtodoinjava.com/2012/10/09/how-hashmap-works-in-java/). А на данный момент просто запомните что *HashMap* работает на основе хэширования. Словарь, по определению, это объект который связывает ключи и значения. Для хранения таких структур, он использует внутренний класс *Entry*.

static class Entry implements Map.Entry  
{  
final K key;  
V value;  
Entry next;  
final int hash;  
...//Еще много кода тут  
}

Переменные *key* и *value* служат для хранения ключа и значения. А сами объекты *Entry* лежат в массиве

/\*\*  
\* Размер таблицы меняется по необходимости,   
\* и обязательно должен быть равен степени двойки  
\*/  
transient Entry[] table;

Индекс нужного элемента в массиве вычисляется по хэш-коду ключа. Больше информации можете получить по ссылке в начале ответа.

1. ***Как создать хороший ключ для словаря?***  
   Следующий хороший вопрос, который обычно задают следом за вопросом о функционировании*HashMap*. Итак, главное ограничение — ключ должен быть таким, чтобы потом по нему можно было получить из словаря значение. Иначе в его использовании просто нет смысла. Если вы понимаете как функционирует hashmap, вы знаете что его работа сильно зависит от методов *hashCode()* и *equals()*объектов-ключей.  
     
   Как следует из вышеизложенного, хороший ключ должен давать один и тот же hashCode снова и снова, независимо от того сколько раз он запрашивается. А также, одинаковые ключи, при вызове метода*equals()* должны возвращать *true*, а разные — *false*.  
     
   Из чего следует, что лучшими кандидатами на роль ключа являются неизменяемые классы.  
     
   Можете почитать еще по [адресу](http://howtodoinjava.com/2013/05/02/how-to-design-a-good-key-for-hashmap/).
2. ***Какие представления содержимого предоставляет интерфейс Map?***  
   Интерфейс *Map* предоставляет три представления хранящихся данных:
   * множество всех ключей
   * множество всех значений
   * множество объектов *Entry*, содержащих в себе и ключ и значение

Перемещаться по ним можно с помощью итераторов.

1. ***Когда нужно использовать HashMap, а когда TreeMap?***  
   *HashMap* это очень широко используемый класс, и вы это знаете. Так что, я ограничусь тем, что скажу что в нем хранятся пары ключ/значение и он позволяет проводить над ними многие операции.  
     
   *TreeMap* это особая разновидность *HashMap*. Разница в том, что ключи в *TreeMap* хранятся упорядоченно. По умолчанию применяется «естественная сортировка». Переопределить сортировку можно предоставив экземпляр класса *Comparator*, метод compare которого и будет использован для сортировки ключей.  
     
   Обратите внимание, что все ключи добавленные в словарь должны реализовывать интерфейс*Comparable* (это необходимо для сортировки). Более того, все ключи должны быть взаимно совместимыми:*k1.compareTo(k2)* не должен вызывать *ClassCastException* для любых *k1* и *k2* хранящихся в словаре. Если пользователь попытается положить в словарь ключ который нарушает это условие (к примеру, строковый ключ в словарь где все ключи типа *Integer*), метод *put(Object key, Object value)*должен вызвать *ClassCastException*.

Вопросы о различиях разных коллекций

18. ***Назовите различия между Set и List?***  
*Список ключевых различий:*  
Множества — неупорядоченные коллекции, тогда как списки — упорядоченные, где каждый элемент имеет индекс начинающийся с нуля.  
Списки могут содержать два и более одинаковых элемента, а множества не могут.  
В списке может содержаться сколько угодно элементов *null*, в множестве — только один.  
  
19. ***Назовите различия между List и Map?***  
Самый легкий вопрос. Список это коллекция элементов, а словарь это коллекция пар ключ/значение. Есть еще множество изменений, но все они являются следствием данного. У них разный интерфейс верхнего уровня, разный набор методов и разные представления данных.  
В большинстве случаев достаточно только первого ответа.  
  
20. ***Назовите различия между HashMap и HashTable?***  
*Есть несколько различий между HashMap и HashTable в Java:*

* *HashTable* потокобезопасна, а *HashMap* нет
* *HashTable* не может содержать элементы *null*, тогда как *HashMap* может содержать один ключ *null* и любое количество значений null
* Третье ключевое отличие между ними — итератор у HashMap, в отличие от перечислителя HashTable, работает по принципу fail-fast (выдает исключение при любой несогласованности данных)

21. ***Назовите различия между Vector и ArrayList?***  
*Отметим некоторые различия:*

* Все методы *Vector* потокобезопасны, а у *ArrayList* — нет
* *Vector* это устаревший класс добавленный в первом релизе *JDK*. *ArrayList* появился в *JDK 1.2*, вместе с остальными классами фреймворка *Collection*
* По умолчанию, *Vector* удваивает свой размер когда заканчивается выделенная под элементы память.*ArrayList* же увеличивает свой размер только на половину

22. ***Назовите различия между Iterator и Enumeration?***  
*Итераторы отличаются от перечислителей по трем признакам:*

* Существуют итераторы, которые позволяют удалять элементы из своей коллекции во время перебора, с использованием метода *remove()*. Класс Iterator данного функционала не поддерживает. С помощью перечислителя нельзя добавлять/удалять элементы
* Перечислители присутствуют в устаревших классах, таких как *Vector/Stack*, тогда как итераторы есть во всех современных классах-коллекциях
* Еще небольшое отличие в том, что у итераторов и перечислителей различается наименование методов, то есть *Enumeration.hasMoreElements()* соответствует *Iterator.hasNext()* и*Enumeration.nextElement()*соответствует *Iterator.next()* и т.д.

23. ***Назовите различия между HashMap и HashSet?***  
*HashMap* — коллекция пар ключ/значение, тогда как HashSet это упорядоченная коллекция уникальных элементов. И это все, не требуется больше никаких объяснений.  
  
24. ***Назовите различия между Iterator и ListIterator?***  
*Есть три различия:*

* *Iterator* может использоваться для перебора элементов *Set*, *List* и *Map*. В отличие от него, *ListIterator*может быть использован только для перебора элементов коллекции *List*
* *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()*
* При помощи *ListIterator* вы можете модифицировать список, добавляя/удаляя элементы с помощью методов *add() и remove()*. *Iterator* не поддерживает данного функционала

25. ***Назовите различия между TreeSet и SortedSet?***  
*SortedSet* это интерфейс, реализуемый классом *TreeSet*. Вот и все!  
  
26.***Назовите различия между ArrayList и LinkedList?***

* *LinkedList* хранит элементы в двусвязном списке, тогда как *ArrayList* хранит их в массиве, размер которого может динамически меняться
* *LinkedList* поддерживает добавление/удаление элементов за фиксированное время, но только последовательный доступ к элементам. То есть, вы можете перебрать список с начала в конец и с конца в начало, но получение элемента в середине списка займет время пропорциональное размеру списка. *ArrayList* же позволяет получить любой элемент по его индексу за фиксированное время. Но добавление/удаление элементов туда требует затрат времени пропорциональных размеру, потому что нужно подвинуть все элементы с места вставки/удаления и до конца списка, либо чтобы освободить место для вставляемого либо чтобы убрать пропуск на месте удаленного
* *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

И еще вопросы

27. ***Как сделать 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.  
  
28.***Как сделать потокобезопасную коллекцию?***  
*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.  
  
29.***Почему не существует метода Iterator.add() для добавления элементов 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.***Какие существуют способы перебирать элементы списка?***  
*Есть 4 способа:*

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

Читайте [статью](http://www.mkyong.com/java/how-do-loop-iterate-a-list-in-java/).  
  
31.***Как вы понимаете работу свойства итератора fail-fast?***  
Итератор *fail-fast* вызывает исключение сразу как обнаружит что структура коллекции была изменена после начала перебора. Под изменением структуры мы понимаем добавление/удаление/изменение элемента коллекции, в то время как другой поток перебирает эту коллекцию.  
  
32.***Какая разница между fail-fast и fail-safe?***  
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*.  
  
33.***Как избежать ConcurrentModificationException во время перебора коллекции?***  
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.  
  
34.***Что такое UnsupportedOperationException?***  
Это исключение возникает при вызове метода, который не поддерживается данной коллекцией.  
К примеру, если вы создадите список только для чтения с помощью *Collections.unmodifiableList(list)*, а потом у него вызовете метод *add()* или *remove()*, вы точно получите это исключение.  
  
35.***Какие классы коллекций позволяют доступ к любому элементу?***  
Классы *ArrayList, HashMap, TreeMap, Hashtable* позволяют доступ к любому элементу.  
  
36.***Что такое BlockingQueue?***  
Это очередь, которая позволяет обрабатывать ситуации, когда из пустой очереди пытаются получить элемент, или когда пытаются добавить элемент в очередь, а место в ней закончилось.  
Методы блокирующей очереди могут быть *четырех* типов: Первые вызывают исключение, вторые возвращают специальные значения (*null или false*, в зависимости от операции), третьи блокируют текущий поток до тех пор пока операция не может быть выполнена, и четвертые блокируют поток только на определенный промежуток времени.  
Примеры можно посмотреть в этой [статье](http://howtodoinjava.com/2012/10/20/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/).  
  
37.***Что такое очередь и стэк, перечислите разницу между ними?***  
Коллекции, созданные 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.  
  
(А теперь отвлечемся и объясним все чуть иначе.  
Итак, представьте обойму автомата. Вы вставляете туда патроны по одному, только с одного конца. И с этого же конца, по одному их будет забирать затворный механизм при стрельбе. Это — стэк, положить элемент можно только на его вершину, снять оттуда же и больше ниоткуда. Последний положенный элемент будет снят первым.  
Очередь — как трубка, в которую вы можете закатывать шарики, и с другого конца их забирать. Очередь позволяет класть элементы только в свой хвост, а забирать только с головы. Тот элемент который был первым положен, первым же будет и забран.   
Общее у очереди и стэка только то, что у них обоих есть фиксированное место куда можно положить элемент, и фиксированное место откуда его можно забрать — все остальные элементы недоступны.(Прим. пер.))  
  
38. ***Что такое интерфейсы 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.  
  
(Снова объясняю по другому.  
Интерфейс *Comparator* описывает метод *int compare(T o1, T o2)*, то есть объект реализующий этот интерфейс может быть использован для сравнения двух других объектов одного класса. Если данный метод вернет 0 — объекты равны, +1 и -1 говорят о том что первый объект «больше» второго или наоборот. Какой смысл вы вложите в понятие «больше» и «меньше» применительно к вашим объектам — исключительно ваше дело. То есть — вы описываете класс, реализующий интерфейс *Сomparator*, и в данном методе определяете как вы хотите сравнивать ваши объекты. Потом передаете объект этого класса конструктору коллекции, в которой хранятся объекты сравниваемые данным методом, и коллекция сможет их сортировать по возрастанию/убыванию. Повторяю — у вас есть коллекция в которой хранятся объекты класса T, и есть еще один объект, другого класса, реализующего интерфейс *Comparator*, вся суть которого в том что он умеет сравнивать эти объекты класса T, больше он ни для чего не нужен.  
Интерфейс *Comparable* описывает метод *int compareTo(T o)*, то есть вызывается он не для сравнения двух других объектов, а для сравнения себя самого с кем-то еще. То есть, вам не нужен отдельный объект-сравниватель, вы закладываете механизм сравнения в сам элемент хранящийся в коллекции. Что и в какой ситуации удобнее — решать вам.(Прим. пер.))  
  
39.***Что такое классы Collections и Arrays?***  
Классы *Collections и Arrays* это специальные служебные классы, для работы с классами-коллекциями. Они позволяют преобразовывать коллекции, делать их защищенными от записи или потокобезопасными, сортировать их разными способами и т.д.