**String**

1. string对象是immutable的，创建之后不可更改(原因是它的char[]是final的，而StringBuilder和StringBuffer的char[]都不是final的)可以共享，维护于string字符串池(heap中的PenGen)中。注意：

xiaoxiang：string是immutable的。另外，还有基本类型的包装类，BigInteger和BigDecimal也是immutable。

StringBuilder不是线程安全的。StringBuffer是线程安全的

a. 不论任何包任何类只要值相同就引用的是字符串池的同一对象，通过**常量表达式**(包括被final修饰的常量变量参与的)计算出的字符串（原则：编译期确定）也在字符串常量池中;

xiaoxiang：final在字符串常量池中

b.在程序**运行过程**中通过+连接起来的字符串对象是新建的，不存在于字符串池而是在堆中；

c.通过intern()返回的值相同的string对象也在字符串中

d.字符串对象实例是在load class时被放到字符串池中的

2.substring()

int size = original.count;

char[] originalValue = original.value;

char[] v;

if (originalValue.length > size) {

// The array representing the String is bigger than the new

// String itself. Perhaps this constructor is being called

// in order to trim the baggage, so make a copy of the array.

int off = original.offset;

v = Arrays.copyOfRange(originalValue, off, off+size);

} else {

// The array representing the String is the same

// size as the String, so no point in making a copy.

v = originalValue;

}

http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html

3.StringBuffer



因为a += b is the equivalent of

a = new StringBuilder()

.append(a)

.append(b)

.toString();

执行一遍toString就会做一遍字符串拷贝从而影响性能

public synchronized String toString() {

return new String(value, 0, count);

}

所以上面的Joinwords所用时间会以字符串长度的长度线性增长，X+2X+3X+….NX=O(XN2 )

而如果使用StringBuilder或者StringBuffer,则只在最后一次做toString,拷贝一次，时间复杂度为O(XN)





链接：

<http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html>

<http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html>

引申一下，String c = “a”+”b”创建了几个对象

由于”a”+”b”在编译期就被优化，完全没有执行加号操作，将“ab”已放入字符串池，所以只有一个对象，但如果是字符串变量做加法，如String c = a + b (a 和b都是字符串)实际上是new StringBuild().append(a).append(b).toString()。

String s="abc"; 一个对象“abc”在常量池中  
  
String s1=new String("abc");本来是两个对象，常量池中的“abc“，堆中的new String("abc")，但是常量池中“abc”已经创建了，不会在创建，所以这一次只有一个  
  
String s2=new String("a"+"b"); 两个对象，常量池中的“ab“，堆中的new String("ab")，不会有“a”，“b"，"a"+"b"常量池中自动优化为“ab”

**HashMap:**

Weakhashmap

1. Weakhashmap对key使用了weakreference,在Key不被强引用后会自动清除entry,但注意这里的自动也不是全自动，是在下次访问map的时候做explunge
2. Weakhashmap适合做cache的原因是，当通过方法向Hashmap里添加缓存数据时，当退出方法后，Key只存在weak refernce，所以整个entry可以被remove, 最好是用softreference, 因为它只会在内存空间不足的时候才会explunge entry，正好起到了cache的作用
3. Weakhashmap的key的equals方法最好只通过==判断，而且最好不用string, 否则字符串池中将一直存在强引用，没办法remove

**TreeSet, LinkedHashSet and HashSet**   
Read more: <http://javarevisited.blogspot.com/2012/11/difference-between-treeset-hashset-vs-linkedhashset-java.html#ixzz3L0KunKSx>

Treemap默认是对key进行排序，传入comparator也是对key进行排序，如果相对treemap的value进行排序，需要先将map中的entryset都取出，进行按值排序，接着把entryset赋值回Map。

**private** **static** Map<String, Integer> sortByComparator(Map<String, Integer> unsortMap) {

*// Convert Map to List*

List<Map.Entry<String, Integer>> list =

**new** LinkedList<Map.Entry<String, Integer>>(unsortMap.entrySet());

*// Sort list with comparator, to compare the Map values*

Collections.sort(list, **new** Comparator<Map.Entry<String, Integer>>() {

**public** **int** compare(Map.Entry<String, Integer> o1,

Map.Entry<String, Integer> o2) {

**return** (o1.getValue()).compareTo(o2.getValue());

}

});

*// Convert sorted map back to a Map*

Map<String, Integer> sortedMap = **new** LinkedHashMap<String, Integer>();

x **for** (Iterator<Map.Entry<String, Integer>> it = list.iterator(); it.hasNext();) {

Map.Entry<String, Integer> entry = it.next();

sortedMap.put(entry.getKey(), entry.getValue());

}

**return** sortedMap;

}

**ArrayList**

For [LinkedList<E>](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html)

* get(int index) is O(n)
* add(E element) is O(1)
* add(int index, E element) is O(n)
* remove(int index) is O(n)
* Iterator.remove() is O(1) <--- main benefit of LinkedList<E>
* ListIterator.add(E element) is O(1) <--- main benefit of LinkedList<E>

For [ArrayList<E>](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

* get(int index) is O(1) <--- main benefit of ArrayList<E>
* add(E element) is O(1) amortized, but O(n) worst-case since the array must be resized and copied, for a new header
* add(int index, E element) is O(n - index) amortized, but O(n) worst-case (as above, for a new header)
* remove(int index) is O(n - index) (i.e. removing last is O(1))
* Iterator.remove() is O(n - index)
* ListIterator.add(E element) is O(n - index)

前面用linkedList，后面用ArrayList

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用ArrayList会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用LinkedList了。

**ListIterator**

ListIterator has add() method because of its ability to traverse or iterate in both direction of collection. it maintains two pointers in terms of previous and next call and in position to add new element without affecting current iteration.  
  
Read more: <http://javarevisited.blogspot.com/2011/11/collection-interview-questions-answers.html#ixzz3LCA69VTa>

1. ListIterator有add()方法，可以向List中添加对象，而Iterator不能   
  
2. ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。   
  
3. ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。   
  
4. 都可实现删除对象，但是ListIterator可以实现对象的修改，set()方法可以实现。Iterator仅能遍历，不能修改。

**多线程：**

1. CopyOnWriteList

在CopyOnWriteArrayList里处理写操作（包括add、remove、set等）是先将原始的数据通过Arrays.copyof()来生成一份新的数组，然后在新的数据对象上进行写，写完后再将原来的引用指向到当前这个数据对象，并且加锁。

读操作是在引用的当前对象上进行读（包括get，iterator等），不存在加锁和阻塞。

因为每次使用CopyOnWriteArrayList.add都要引起数组拷贝， 所以应该避免在循环中使用CopyOnWriteArrayList.add。可以在初始化完成后设置到CopyOnWriteArrayList中，或者使用CopyOnWriteArrayList.addAll方法

copyonwritearraylist在遍历过程中另一个线程修改不会报concurrentmodification exception的原因是它的遍历器特殊(实现listiterator)，它会赋值一个final 的 snapshot，另一个线程怎么改，遍历的这个也不会受到影响，那个改完了赋值回去，也必须在下次遍历的时候重新拿itrator才看出变化, 这种copyonwritearraylist主要用到遍历多改动少的场合，线程安全，改动多得场合性能开销就大了，不适合.注意COWIterator的remove set add都会直接抛UnsupportedOperationException.

* Reads do not block, and effectively pay only the cost of a volatile read;
* Writes do not block reads (or vice versa), *but* **only one write can occur at once**;
* Unlike ConcurrentHashMap, write operations that write or access multiple elements in the list (such as addAll(), retainAll()) will be atomic.

1. ConcurrentHashmap

concurrenthashmap是对Map的一部分加锁，更具体说是对bucket级别加锁，为的是高并发性，可以同时多个线程读，甚至可以同时读写（应用在不同的bucket），同时写不允许，一个时间只有一个线程可以写操作。 对比synchronizedhashmap, 通过collections.synchronizedHashmap(map)得到, 它用的是对象级的锁，不是bucket级别，任何put get 方法都是synchronized的，一个时刻只有一个线程可以进行操作，低并发性

*The iterator for a synchronized collection is****not****synchronized itself*. Access through the iterator is equivalent to access through the underlying unsynchronized collection.  
Iterators don't like it anyway if the collection is modified in any way except through the iterator itself.

ConcurrentHashmap不允许有Null的key 和 value

The main reason that nulls aren't allowed in ConcurrentMaps (ConcurrentHashMaps, ConcurrentSkipListMaps) is that ambiguities that may be just barely tolerable in non-concurrent maps can't be accommodated. The main one is that if map.get(key) returns null, you can't detect whether the key explicitly maps to null vs the key isn't mapped. In a non-concurrent map, you can check this via map.contains(key), but in a concurrent one, the map might have changed between calls.

注意，concurrenthashmap实现的是弱一致性，而hashtable是强一致性，例如concurrenthashmap keyset的Iterator就不是fail fast，所以当一个线程对map进行删除操作，而另一个线程同步遍历，未必可以看到一致性数据，而且clear方法是逐一对segment执行清空操作，并且没有加锁，这样在交互操作时也有可能出现不一致。

1. Threadlocal
2. Threadlocal的内部结构？

答：threadlocal类里有一个静态内部类是threadlocalhashmap, 但这个hashmap的对象却是作为thread类的一个属性。Hashmap的key是threadlocal对象，value是threadlocal的值。

1. 为什么threadlocalhashmap是thread的属性而非Threadlocal？

答：因为这样做的好处就是如果thread销毁，threadlocalhashmap也跟着一起销毁，没有外部引用可以transverse到，而如果放到threadlocal里，thread销毁还需要额外销毁map所占用的内存，其次如果在threadlocal这个类里控制，其实就相当于这个map的结构变为Key为threadId, value为threadlocal变量的值，对多个线程对这个Map的访问就需要有额外的同步，但实际上threadlocal从设计出发点并不需要考虑并发控制，因为这个local变量应该对外界是不可见的。

1. 为什么threadlocalhashmap的key是weakreference?

答：threadlocalhashmap的entry继承了weakreference, 但是只有key是通过weakreference引用的，value并不是，这样当线程结束（eg. run方法执行结束，变量out of scope），对key之前的强引用消失，只剩下weakreference, 可以回收，但是value object并不能回收（stale entry）, 默认为强引用。

1. Threadlocal不会产生内存泄漏问题了么？

答：正如上面的解释，Object强引用不会被回收，所以会产生内存泄露的问题。有两种方法回收value object.

1. 结束线程，map被回收，自然包括value对象
2. - GC threadlocal对象，当只有weak reference引用这些threadlocal对象的时候

* 通过set()往这个tread的hashmap里添加若干threadlocal,这些threadlocal 就会遍历hashmap， remove 一些stale entry，和weakhashmap不一样的是threadlocalhashmap只有在set的时候才会expunge stale entry, 而weakhashmap是在get的时候expunge stale entry

Iii 也可以通过调用Threadlocal.remove() 去主动remove 对应entry, 这里有一个问题如果不主动remove entry在线程池的环境中，如果不及时clean掉这些threadlocal, 如果这个线程对象可以被重用的话，那么threadlocal里的值有可能是上一次操作的最后结果，而且如果线程一直运行，有可能造成内存泄露的问题。

1. ReentrantLock
2. lockInterruptibly()

i. 线程在sleep或wait,join， 此时如果别的进程调用此进程的 interrupt（）方法，此线程会被唤醒并被要求处理InterruptedException；(thread在做IO操作时也可能有类似行为，见java thread api)  
ii. 此线程在运行中， 则不会收到提醒。但是 此线程的 “打扰标志”会被设置， 可以通过isInterrupted()查看并 作出处理。

iii. lockInterruptibly()和上面的第一种情况是一样的， 线程在请求lock并被阻塞时，如果被interrupt，则“此线程会被唤醒并被要求处理InterruptedException”。

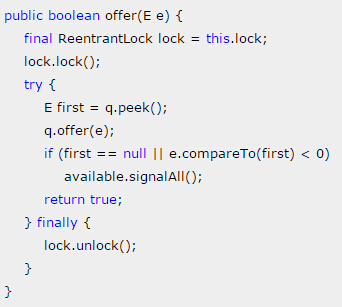
## Synchronized is not possible to interrupt a thread that is waiting to acquire a lock, nor is it possible to poll for a lock or attempt to acquire a lock without being willing to wait forever for it. Synchronization also requires that locks be released in the same stack frame in which they were acquired, which most of the time is the right thing (and interacts nicely with exception handling), but a small number of cases exist where non-block-structured locking can be a big win.

## 对于synchronized的缺点之前也简单的说了一些，实际使用中比较烦扰的几点是：a.只有一个"条件"与锁相关联，这对于大量并发线程的情况是很难管理(等待和唤醒)；b.多线程竞争一个锁时，其余未得到锁的线程只能不停的尝试获得锁，而不能中断。这种情况对于大量的竞争线程会造成性能的下降等后果。JDK5以后提供了ReentrantLock的同步机制对于前面提的两种情况有相对的改善。

## DelayQueue

## -Priority queue内部存储是通过堆的数据结构，排序是堆排序

## - DelayQueue内部的实现使用了一个优先队列。当调用DelayQueue的offer方法时，把Delayed对象加入到优先队列q中。如下：



## -通过PriorityQueue，使得超时的Item最先被处理，利用BlockingQueue，将Item处理的操作阻塞住。避免了遍历方式的轮训，提高了性能。

**Singleton**

Singleton with exception

private static class SingletonObjectFactoryHolder{

private static final ObjectFactory INSTANCE;

static {

try {

INSTANCE = new ObjectFactory();

} catch (Exception e) {

throw new ExceptionInInitializerError(e);

}

}

}