Java-Success.com

Industrial strength Java/JEE Career Companion to open more doors

Register | Login | Logout | Contact Us

search here … Go

Home | Java FAQs | 600+ Java Q&As | Career | Tutorials | Member | Why?

Can u Debug? | Java 8 ready? | Top X | Productivity Tools | Judging Experience?

Home › Interview › Core Java Interview Q&A › Collection and Data structures › When to use which Java data structure? and why?

When to use which Java data structure? and why?

Posted on September 17, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

0 Like Share Tweet 0 G+1 Share

List, ***Set***, ***Map***, and ***Queue***(access the ends FIFO or LIFO) are the basic Java data structure interfaces for which there are different implementations to cater for different usage patterns. Java data structure interview questions are very popular, and pays to know the differences.

#1 **Array** (Employee[]) Vs **List** (List<Employee>):

- Array is a fixed length data structure whilst a List is a variable length Collection class.

- java.util.Arrays has the helper classes for the arrays and java.util.Collections has the helper classes for the Collection classes.
- An array can use primitive data types or object types, but the Collection classes can only use objects.

Q1. Which one to favor/use?
A1. Favor ***List***.

- Arrays are inflexible and do not have the expressive power of generic types.
- List gives you the data abstraction as you can swap ArrayList, LinkedList, CopyOnWriteArrayList, etc depending on the requirements.
- List allows you to add and subtract elements even it is an O(n) operation in worst case scenario.

#2: *ArrayList* (*ArrayList<employee>*)Vs ***LinkedList*** (Linked*List<employee>*)

- Insertions and deletions are faster in *LinkedList* compared to an *ArrayList* as *LinkedList* uses links (i.e. before and next reference) as opposed to an ArrayList, which uses an array under the covers, and may need to resize the array if the array gets full. Adding to an ArrayList has a worst case scenario of O(n) whilst *LinkedList* has O(1).
- *LinkedList* has more memory footprint than *ArrayList*. An *ArrayList* only holds actual object whereas *LinkedList* holds both data and reference of next and previous node.
- Random access has the worst case scenario of O(n) in *LinkedList* as to access 6th element in a *LinkedList* with 8 elements, you need to traverse through 1 to 5th element before you can get to the 6th element, whereas in an *ArrayList*, you can get the 6th element with O(1) with *list.get(5)*.

Q2. Which one to favor/use?
A2.. Almost always favor an ***ArrayList.*** *ArrayLists* are good for write once and randomly read many times, and for adding

16 Technical Key Areas

elements at the end, but bad at add/remove from the front or middle. *LinkedLists* are useful in rare usage patterns where insertion and deletion in the front and middle is significantly more than the retrievals, and the *LinkedList* can live with the **O(n)** cost of random access.

If you are after adding/removing from both ends, ***ArrayDeque*** is better than a *LinkedList*. The *Deque* interface is pronounced as "deck", and represents a double-ended queue. *ArrayDeque* is backed by an array. ArrayDeque can be used as both a Queue (i.e. FIFO) and a Stack (i.e. LIFO). ArrayDeque performs better than *LinkedList*, as *ArrayDeque* is backed by an array and Array is more cache friendly. Also, *ArrayDeque* is more memory efficient than *LinkedList* since it does not have to keep an additional reference to previous or next node. *ArrayDeque* does not take null elements, but a *LinkedList* does. The best operation in a *LinkedList* implementation is removing the current element during the iteration. *LinkedList* implementations are also not ideal to iterate. So, *LinkedList* is very rarely used.

#3: ***List*** Vs ***Set***

Set can't have duplicates whereas a List can have duplicates.

#4: ***HasMap*** Vs ***TreeMap***

TreeMap is an implementation of a *SortedMap*, where the order of the keys can be sorted, and when iterating over the keys, you can expect that keys will be in order. ***HashMap*** on the other hand, makes no such guarantee on the order.

Q3. Which one to favor/use?
A3. In general, favor ***HashMap*** as it is more efficient in general, and use a *Comparator* to sort when required. Use a ***TreeMap*** only when it is required to keep the keys in a sorted order to iterate over them.

#5: ***HashSet***, ***ArrayList*** Vs. ***CopyOnWriteSet***, ***CopyOnWriteArrayList***

80+ step by step Java Tutorials

100+ Java pre-interview coding tests

HashSet and *ArryList* are not thread-safe and you need to provide your own synchronization, whereas ***CopyOnWriteArraySet*** and ***CopyOnWriteArrayList*** are not only thread-safe, but more efficient as they allow concurrent multiple reads and single write. This concurrent read and write behavior is accomplished by making a brand new copy of the list every time it is altered.

Q4. Which one to favor/use?
A4.Favor ***CopyOnWriteArraySet*** or ***CopyOnWriteArrayList*** only when the number of reads is significantly more than the number of writes. It also gives you **fail safe** iteration when you want to add/remove elements during iteration.

#6: ***HashMap*** Vs ***ConcurrentHashMap***

HashMap is not thread-safe and you need to provide your own synchronization with *Collections.synchornizedMap(hashMap)*, which will return a collection which is almost equivalent to the legacy ***Hashtable***, where every modification operation on Map is locked. As the name implies, ***ConcurrentHashMap*** provides thread-safety by dividing the whole map into different partitions based upon concurrency level and locking only particular portion instead of locking the whole map. *ConcurrentHashMap* does not allow NULL key values, whereas *HashMap* there can only be one null key.

Q5. Which one to favor/use?
A5.Favor *ConcurrentHashMap* for scalability.

#7: new ***SynchronousQueue( )*** Vs. new ***ArrayBlockingQueue(1)*** or ***LinkedBlockingQueue(1)***

SynchronousQueue is a queue that can only contain a single element internally. A thread inserting an element into a *SynchronousQueue* blocked until another thread takes that

How good are your .....?

element from the queue. Similarly, if a thread tries to take an element, and no element is currently present, that thread will be blocked until a thread inserts an element into the *SynchronousQueue*.

SynchronousQueue provides extremely good throughput when compared to a unit sized *ArrayBlockingQueue(1)* or *LinkedBlockingQueue(1)*. *SynchronousQueue* is the default *BlockingQueue* used in the *Executor* framework for the *Executors.newCachedThreadPool( )*.

Q6. Which one to favor/use?
A6.

- Use *SynchronousQueue* in scenarios where a task to be executed asynchronously while discarding any further requests until the task is finished.
-

```
executor = new ThreadPoolExecutor(
     1, 1,
     2000, TimeUnit.SECONDS,
     new SynchronousQueue&lt;runnable&gt;()
     new ThreadPoolExecutor.DiscardPolicy()
```

- Use *SynchronousQueue* to improve application performance.

#8: ***HashMap*** Vs ***LinkedHashMap***

LinkedHashMap will iterate in the order in which the entries were put into the map. *HashMap* does not provide any grantees about the iteration order.

Q7. What is a ***BlockingQueue***?
A7. ***BlockingQueue*** is a ***Queue*** that supports additional operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. The main advantage is that a *BlockingQueue* is that it provides a correct, thread-safe implementation with throttling.

- The producers are throttled to add elements if the consumers get too far behind.
- If the Queue capacity is limited, the memory consumption will be limited as well.

Q8. Why ***Vector***, ***Stack***, and ***Hashtable*** are legacy data structures and not to be used?
A8. All methods in these classes are synchronized (i.e. coarse grained lock), hence not efficient. Favor the concurrent data structures for concurrent read and single write.

#9: *WeakHashMap*

Q9. When will you use a WeakHashMap?
A9. A WeakHashMap is good to implement **canonical maps**. If you want to associate some extra information to a particular object that you have a strong reference to, you put an entry in a WeakHashMap with that object as the key, and the extra information as the map value. Then, as long as you keep a strong reference to the object, you will be able to check the map to retrieve the extra information. Once you release the strong reference to the key object, the map entry will be cleared and the memory used by the extra information will be released.

Q10. Is a WeakHashMap good for caching?
A10. **No**. A cache is a memory location where you can store data that is otherwise expensive to obtain frequently from a database, ldap, flat file, or other external systems. A WeakHashMap is not good for caching because a WeakHashMap stores the keys using WeakReference objects that means as soon as the key is not strongly referenced from somewhere else in your program, the entry may be removed and be available for garbage collection. This is not good, and what you really want to have is your cached objects removed from your map only when the JVM is running low on memory. This is where a **SoftReference** comes in handy. A SoftReference will only be garbage-collected when the JVM is running low on memory and the object that the key is pointing to is not being accessed from any other strong

reference. The standard Java library does not provide a Map implementation using a SoftReference, but you can implement your own by extending the AbstractMap class.

Q. Is it a good idea to implement your own caching mechanism?
A. Implementing your own cache mechanism is often not a trivial task. Cache needs to be regularly updated, and possibly distributed. A better option would be to use one of the third-party frameworks like OSCache, Ehcache, JCS and Cache4J.

Popular Posts

♦ 11 Spring boot interview questions & answers
823 views
♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers
765 views
18 Java scenarios based interview Questions and Answers
399 views
001A: ♦ 7+ Java integration styles & patterns interview questions & answers
388 views
01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers
295 views
♦ 7 Java debugging interview questions & answers
293 views
01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints
285 views
♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers
279 views
♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers
239 views
001B: ♦ Java architecture & design concepts interview questions & answers

201 views

Bio | **Latest Posts**

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ 4 Java Collection Quizzes

04: Can you think of a time where you …? open-ended Java interview Q&A ›

Posted in Collection and Data structures**,** member-paid**,** Which Data Structure & WHy

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from "Blah blah" to "Wow"? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼

© Disclaimer

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.

© 2016 Java-Success.com ↑ Responsive Theme powered by WordPress