♦ Java Collection overview interview questions and answers

Posted on September 1, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1. What are the common data structures, and where would you use them?

A1. Arrays, Lists, Sets, Maps, Trees, and Graphs. Many leave out **Trees** and **Graphs** as Java does not provide classes out of the box.

Arrays are the most commonly used data structure. Arrays are of fixed size, indexed, and all containing elements are of the same type (i.e. a homogenous collection). For example, storing employee details just read from the database as *EmployeeDetail*[ ]. The *java.util.**Arrays*** class and the

*java.lang.**System*** class has a utility method to quickly copying a portion of an array.

Lists are known as arrays that can grow. These data structures are generally backed by a fixed sized array and they re-size themselves as necessary. A **list can have duplicate elements**.

Sets are like lists but they do not hold duplicate elements. Sets can be used when you have a requirement to store **unique elements**.

Stacks allow access to only one data item, which is the last item inserted (i.e. Last In First Out – **LIFO**). If you remove this item, you can access the next-to-last item inserted, and so on. The LIFO is achieved through restricting access only via methods like peek( ), push( ), and pop( ). This is useful in many programing situations like parsing mathematical expressions like (4+2) * 3, storing methods and exceptions in the order they occur, checking your source code to see if the brackets and braces are balanced properly, etc.

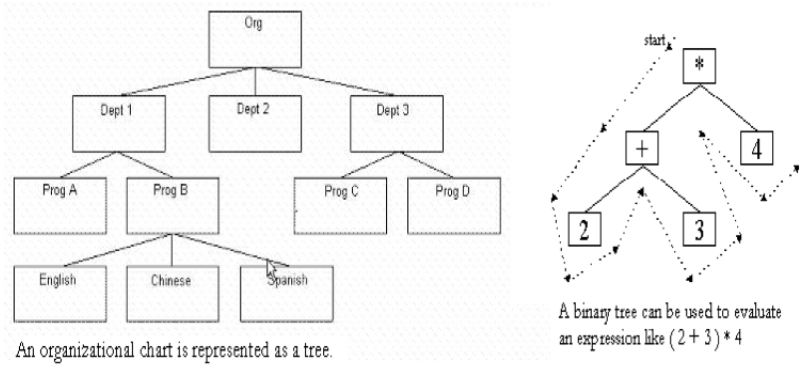
Queues are somewhat like a stack, except that in a queue the first item inserted is the first to be removed (i.e. First In First Out – FIFO). The **FIFO** is achieved through restricting access only via methods like peek( ), offer( ), and poll( ). For example, waiting in a line for a bus, a queue at the bank or super market teller, etc.

Maps are amortized constant-time access data structures that map keys to values. This data structure is backed by an array. It uses a hashing functionality to identify a location, and some type of collision detection algorithm is used to handle values that hash to the same location. For example, storing employee records with employee number as the key, storing name/value pairs read from a properties file, etc. Initialize them with an appropriate initial size to minimize the number of re-sizes.

Trees are the data structures that contain nodes with optional data elements and one or more child elements, and possibly

16 Technical Key Areas

each child element references the parent element to represent a hierarchical or ordered set of data elements. For example, a hierarchy of employees in an organization, storing the XML data as a hierarchy, etc. If every node in a tree can have utmost 2 children, the tree is called a binary tree. The binary trees are very common because the shape of a binary tree makes it easy to search and insert data. The edges in a tree represent quick ways to navigate from node to node. Java does not provide an implementation for this but it can be easily implemented as shown below. Just make a class Node with an ArrayList holding links to other Nodes that are child nodes.



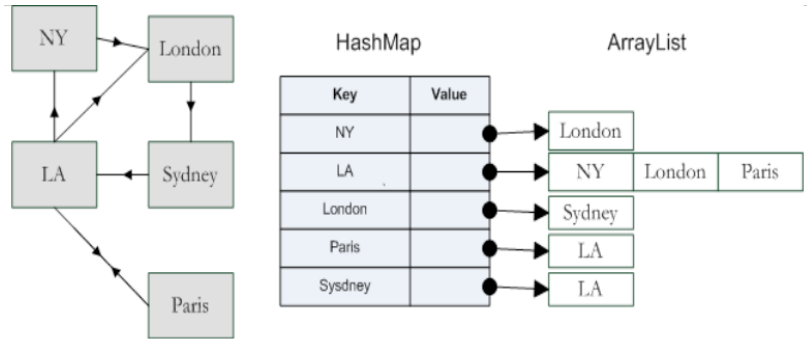
Tree Data Structure

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Node {
5      private String name;
6      private List<node> children = new ArrayList<
7      private Node parent;
8
9      public Node getParent( ) {
10         return parent;
11     }
12
13     public void setParent(Node parent) {
14         this.parent = parent;
15     }
16
17     public Node(String name) {
18         this.name = name;
19     }
20
21     public void addChild(Node child) {
22         children.add(child);
23     }
24
25     public void removeChild(Node child) {
```

⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managen

```
26          children.remove(child);
27      }
28
29      public String toString( ) {
30          return name;
31      }
32  }
33
```

Graphs are data structures that represent arbitrary relationships between members of any data sets that can be represented as networks of nodes and edges. A tree structure is essentially a more organized graph where each node can only have one parent node. Unlike a tree, a graph's shape is dictated by a physical or abstract problem. For example, nodes (or vertices) in a graph may represent cities, while edges may represent airline flight routes between the cities.



Graph Data Structure

Q2. What do you know about the big-O notation and can you give some examples with respect to different data structures?
A2. The **Big-O** notation simply describes how well an algorithm scales or performs in the worst case scenario as the number of elements in a data structure increases. The Big-O notation can also be used to describe other behavior such as memory consumption. At times you may need to choose a slower algorithm because it also consumes less memory. Big-o notation can give a good indication about performance for large amounts of data, but the only real way to know for sure is to have a performance benchmark with large data sets to take into account things that are not considered in Big-O notation like paging as virtual memory usage grows, etc. Although benchmarks are better, they

Written Test JEE (1)

How good are your .....?

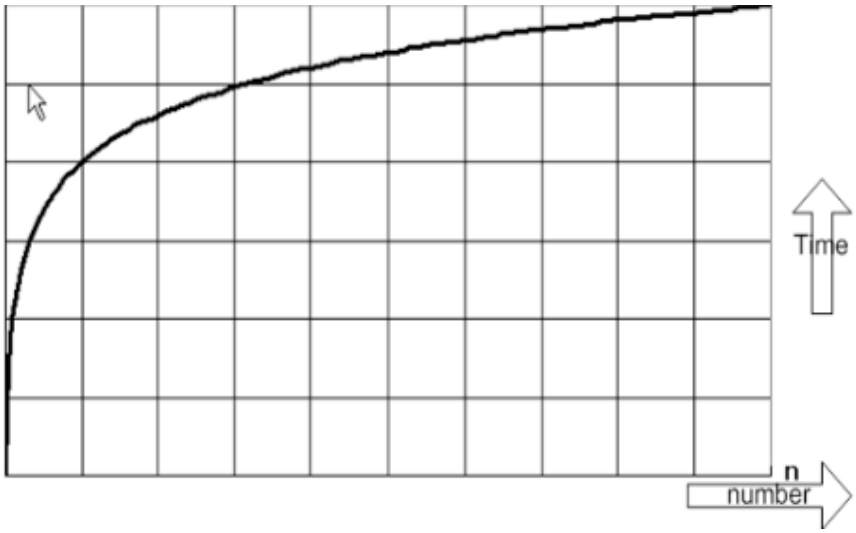
open all | close all
Career Making Know-
Job Hunting & Resum

aren't feasible during the design process, so Big-O complexity analysis is the choice.

O(1) Constant

E.g. Map look up by key: map.get(key);
E.g. Conditional logic: (a == 1) ? true : false;

O(log n) Logarithmic
Logarithmic means if 10 items takes at most x amount of time, then 100 items takes 2x amount of time, and 1,000 items takes at most 3x, and 10,000 items take 4x, and so on. In O(log n)running time increases logarithmically in proportion to the input size. Items are increasing 10 folds whilst the time is increasing only 2 fold. So, more efficient for larger number of items.



E.g. Binary search: search for 2 in a list of numbers 1,2,3,4,5,6,7 .

Step 1: Sort the data set in ascending order as binary search works on sorted data.
Step 2: Get the middle element (e.g. 4) of the data set and compare it against the search item (e.g. 2), and if it is equal return that element
Step 3: If search item value is lower, discard the second half of the data set and use the first half (e.g. 1,2,3). If the search item value is higher, then discard the first half and use the second half (e.g. 5,6,7)

Step 4: Repeat steps 2 and 3 until the search item is found or the last element is reached and search item is not found.

O(n) – Linear

Running time increases in direct proportion to the input size. if 1 item takes x amount of time, 10 items take 10x, and 1000 items takes 1000x.

E.g. Finding an item in an unsorted array or list — looping through n items in a for loop.

O(n log n) – Super Linear

Running time is midway between a linear algorithm and a polynomial algorithm.

E.g. Merge Sort: Collections.sort is an optimized merge sort which actually guarantees O(n log n). A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee n log(n) performance. For Merge sort worst case is O(n*log(n)), for Quick sort: O(n^2).

O(n^c) Polynomial

Running time grows quickly based on the size of the input.

E.g. For loop within a for loop – naive bubble sort.

O(c^n) Exponential

Running time grows even faster than a polynomial algorithm.

E.g. Recursive computation of Fibonacci numbers

```
1 public int fib(int n) {
2     if (n <= 1) return n;
3     else return fib(n - 2) + fib(n - 1);
4 }
5
```

O(n!) Factorial

Running time grows the fastest and becomes quickly unusable for even small values of n.

E.g. Recursive computation of factorial

```
1 public void nFactorial(int n) {
2   for(int i=0; i<n; i=n-1) {
3     nfactorial(i);
4   }
5 }
6
```

Q3. What are the core interfaces of the Java collection framework?

A3. *Collection*, *List*, *Set*, *SortedSet*, *Map*, *SortedMap*, *ConcurrentMap*, *Queue*, *Dequeue*, *BlockingQueue*, *Iterator* (for uni-directional traversal), and **ListIterator** (for bi-directional traversal).

Abstract Data Structure	Concrete Implementation
List	*ArrayList*, *CopyOnWriteArrayList*, *LinkedList*, *Stack*, etc.
Set	*HashSet*, *EnumSet*, *CopyOnWriteArraySet*, *LinkedHashSet*, *TreeSet*, etc
Map	*HashMap*, *ConcurrentHashMap*, *EnumMap*, *WeakHashMap*, *TreeMap*, etc
SortedSet	*TreeSet*
SortedMap	*TreeMap*
ConcurrentMap	*ConcurrentHashMap*
Queue	*ArrayBlockingQueue*, *PriorityQueue*, *PriorityBlockingQueue*, *SynchronousQueue*, etc
BlockingQueue	*ArrayBlockingQueue*, *DelayQueue*, *PriorityBlockingQueue*, *SynchronousQueue*, etc.
Iterator	*Scanner, etc.* The implementing class provides the implementation using the iterator design pattern. E.g. myList.*iterator()*
ListIterator	The implementing class provides the implementation using the iterator design pattern. E.g. myList.*listIterator()*

Java Collection framework

Q4. What is the purpose of adding the default method stream() to the Collection interface in Java 8?

A4. A **stream** is an infinite sequence of consumable

elements (i.e a data structure) for the consumption of an operation or iteration. Any Collection can be exposed as a stream. The operations you perform on a stream can either be

— **intermediate** (map, filter, sorted, limit, skip,concat, substream, distinct, etc) producing another stream or
— **terminal** (forEach, reduce, collect, sum, max, count, matchAny, findFirst, findAny, etc) producing an object that is not a stream.

Basically, you are building a pipeline as in Unix

```
1 ls -l | grep "Dec" | Sort +4n | more
```

stream( ) is a default method added to the Collection interface in Java 8. The stream( ) returns a *java.util.Stream* interface with multiple abstract methods like filter, map, sorted, collect, etc. *DelegatingStream* implements these abstract methods.

```
1  package com.java8.examples;
2  import java.math.BigDecimal;
3  import java.util.Arrays;
4  import java.util.List;
5  import java.util.stream.Collectors;
6
7  public class EmployeeTest {
8
9   private static List<Employee> employees = Array
10         new Employee("Steve", BigDecimal.valueOf
11         new Employee("Peter", BigDecimal.valueOf
12         new Employee("Sam", BigDecimal.valueOf(7
13         new Employee("John", BigDecimal.valueOf(
14
15
16    public static void main(String[] args) {
17        //e is the parameter for Employee
18        List<Employee> fullTimeEmployees = employ
19            .filter(e -> e.getWorkType() == Emplo
20            .collect(Collectors.toList()); // ret
21        fullTimeEmployees.forEach(e -> System.out
22    }
23 }
24
```

The above example creates a new list of full time employees. The operations **.stream( )**, **.filter( )** create the **intermediate**

streams, hence they are chained, and the **.collect** is the terminal operation that returns the final list of full-time employees.

Q5. Describe typical use cases or examples of common uses for various data structures provided by the Collection framework in Java?
A5. Use **arrays** when the amount of data is reasonably small and the amount of data is predictable in advance.

If the initial collection size cannot be determined upfront, the primary implementations like **ArrayList**, **HashMap**, or **HashSet** should do the job unless you require a special usage pattern. Those special usage patterns are like **access sequences** like FIFO, LIFO, etc, **duplicates** are allowed or not, **ordering** needs to be maintained or not, **concurrent access** is required or not, etc.

Use a List if duplicates are allowed, and a Set if **duplicates** are not allowed.

Use a stack for Last In First Out (**LIFO**) access. For example, you may want to track online forms as a user opens them and then be able to back out of the open forms in the reverse order. LIFO stack operations is provided by the **Dequeue** interface, which stands for double-ended queue and its implementations.

If you conceptually have a producer-consumer pattern, for example a producer thread produces a list of jobs for a number of consumer threads to pick up, then a **Queue** (i.e. FIFO) implementation is more appropriate. This of course could be done with a synchronized *LinkedList*, but a queue will provide a better concurrency optimization by eliminating random access. The *BlockingQueue* is an efficient implementation of a typical Queue interface. There are other specific implementations like *DelayQueue*, *PriorityQueue*, *SynchronousQueue*, etc. to cater for the variations in the usage.

Use a **TreeSet** or **TreeMap** if you like your objects to be in sorted order by using a balanced red-black tree. A red-black tree is a balanced binary tree, meaning a parent node has maximum 2 children and as an entry is inserted, the tree is monitored as to keep it well-balanced. Balanced binary tree ensures fast lookup time of O(log n). A HashSet or a HashMap has a much faster access time of O(1), but won't maintain the entries in a sorted order.

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.

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.

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**.

Use an immutable collection (aka **unmodifiable collection**) if you don't want to allow accidental addition or removal of elements once created. The objects stored in a collection needs to implement its own immutability if required to be prevented from any accidental modifications. The **java.util.concurrent** package has a number of classes that allow thread-safe concurrent access.

Q6. What are some of the multi-threading considerations in using the different data structures?
A6. **If accessed by a single thread**, synchronization is not required, and Arrays, ArrayLists, HashSets, ArrayDeque, etc can be used as a local variable. If your collections are used as local variables, the synchronization is a complete overkill, and degrades performance considerably. On the contrary, if declaring it as an instance or static variable it is a bad practice to assume that the application is always going to be single threaded. What if the application needs to scale to handle concurrent access from multiple threads in the future? so, code in a thread-safe manner.

If accessed by multiple threads, prefer a concurrent collection like a **copy-on-write lists and sets**, concurrent maps, etc over a synchronized collection for a more optimized concurrent access. Stay away from the legacy classes like Vectors and HashTables. In a multi-threaded environment, some operations may need to be atomic to produce correct results. This may require appropriate synchronizations (i.e. locks). Improper implementation in a multi-threaded environment can cause unexpected behaviors and results.

Q7. What are some of the performance and memory considerations in regards to data structures?
A7. The choices you make for a program's data structures and algorithms affect that program's memory usage (for data structures) and CPU time (for algorithms that interact with those data structures). Sometimes you discover an inverse

relationship between memory usage and CPU time. For example, a one-dimensional array occupies less memory than a doubly linked list that needs to associate links with data items to find each data item's predecessor and successor. This requires extra memory. In contrast, a one-dimensional array's insertion/deletion algorithms are slower than a doubly linked list's equivalent algorithms because inserting a data item into or deleting a data item from a one-dimensional array requires data item movement to expose an empty element for insertion or close an element made empty by deletion. Here are some points to keep in mind.

— The most important thing to keep in mind is **scalability**. Assuming that a collection will always be small is a dangerous thing to do, and it is better to assume that it will be big. Don't just rely on the general theories (e.g. Big-O theory) and rules. Profile your application to identify any potential memory or performance issues for a given platform and configuration in a production or production-like (aka QA) environment.

— Initialize your collection with an **appropriate initial capacity** to minimize the number of times it has to grow for lists and sets, and number of times it has to grow and rehash for maps.

```
1 List list = new ArrayList(40);
2 Map map = new HashMap(40);
3
```

Q8. What are the different options to sort a collection of objects?

A8.

Option 1: To sort a given list **naturally**. For example, the String and wrapper classes like Integer, Double, etc implements the *Comparable* interface and provide the **compareTo(..)** *method for sorting. If you define your own objects like Product, then you can implement the Comparable interface and provide the implementation of compareTo(T t) method for natural ordering of that object as shown below.*

*The natural ordering for a class Product needs to be **consistent with the equals method**, and it is said to be consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class Product.*

```
1 Collections.sort(shoppingBasket);
```

***Option 2**: Writing your own **Comparator** implementation to custom sort. Multiple comparators can be written to sort the same collection in different ways for different requirements. An anonymous Comparator implementation is shown below to sort by size of the string.*

```
1 Collections.sort(shoppingBasket, new Comparator<L
2       public int compare(List<String> o1, List<St
3           return o2.size( ) - o1.size( );
4       }
5 });
6
```

Option 3: If you are using **Java 8**, using the **functional programming approach**.

```
1 //java 8 approach fro multi-fields
2   Comparator<Person> multiFieldComparator =
3             Comparator.comparing(Person::getGende
4                       .thenComparing(Person::getN
5                       .thenComparing(Person::getA
6
```

Q9. What are some of the best practices relating to the Java Collection framework?

A9.

#1. **Choose the right type of data structure based on usage patterns** like fixed size or required to grow, duplicates allowed or not, ordering is required to be maintained or not, traversal is forward only or bi-directional, inserts at the end only or any arbitrary position, more inserts or more reads, concurrently accessed or not, modification is allowed or not, homogeneous or heterogeneous collection, etc. Also, keep

multi-threading, atomicity, memory usage and performance considerations discussed earlier in mind.

#2. Don't assume that your collection is always going to be small as it can potentially grow bigger with time. So your collection should scale well.

#3. Program in terms of **interface not implementation**: For example, you might decide a LinkedList is the best choice for some application, but then later decide ArrayList might be a better choice for performance reason.

Bad:

```
1 ArrayList<String> list = new ArrayList<String>(10
```

Good:

```
1 // program to interface so that the implementatio
2 List<String> list = new ArrayList<String>(100);
3 List<String> list2 = new LinkedList<String>(100);
4
```

#4. Return zero length collections or arrays as opposed to returning a null in the context of the fetched list is actually empty. Returning a null instead of a zero length collection is more error prone, since the programmer writing the calling method might forget to handle a return value of null.

```
1 List<String> emptyList = Collections.emptyList( )
2 Set<Integer> emptySet = Collections.emptySet( );
3
```

#5. Use generics for type safety, readability, and robustness.

#6. **Immutable objects should be used as keys for the HashMaps**: Generally you use java.lang.Integer or java.lang.String class as the key, which are immutable Java objects. If you define your own key class, then it is a best practice to make the key class an immutable object. If you want to insert a new key, then you will always have to

instantiate a new object as you cannot modify an immutable object. If the keys were made mutable, you could accidentally modify the key after adding to a HashMap, which can result in you not being able to access the object later on. The object will still be in the HashMap, but you will not be able to retrieve it as you have the wrong key (i.e. a mutated key).

#7. Use copy-on-write classes and concurrent maps for better scalability. It also prevents **ConcurrentModificationException** being thrown while preserving thread safety. These classes provide **fail-safe** iteration as opposed to non-concurrent classes like ArrayList, HashSet, etc use **fail-fast** iteration leading to ConcurrentModificationException if you try to remove an element while iterating over a collection.

#8. Memory usage and performance can be improved by setting the appropriate initial capacity when creating a collection. For example,

If you are likely to have an ArrayList with say 11 elements, but if you initialize the ArrayList as follows,

```
1 List<String> myList = New ArrayList<String>( );
```

By default, the capacity is 10. When you add the 11th element to the array list, it will have to resize or grow using the following formula (oldCapacity * 3)/2 + 1. This will be equal to 10*3/2+1=16. So it creates a new array with size of 16 and and copies all the old 10 elements to the new array and adds the 11th element to the new array. The old array with 10 elements become eligible for garbage collection. So resizing too many times can adversely impact performance and memory. So it is a best practice to set the initial capacity to an appropriate value so that the lists don't have to resize too often. The above declaration for 11 elements can be improved by setting the initial capacity to either 11 or greater.

```
1 List<String> myList =  new ArrayList<String>(11);
```

Same is true for HashMaps as well. As a general rule, the default load factor of 0.75 offers a good tradeoff between time and space costs. Higher load factor values decrease the space overhead, but increases the lookup cost through methods like get( ), put( ), etc. The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. This means the load factor should not be changed from 0.75, unless you have some specific optimization you are going to do. Initial capacity is the only thing you want to change, and set it according to number of items you want to store. You should set the initial capacity to (no. of likely items / 0.75) + 1 to ensure that the table will always be large enough, and no rehashing will occur. For 11 items it will be (11/0.75) + 1 = 16.

```
1 Map<String> myMap = new  HashMap<String>(16);
```

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**

‹ 18 Java scenarios based interview Questions and Answers

Mocks, stubs, domain, and anemic objects interview Q&A ›

Posted in Collection and Data structures, member-paid

Tags: Core Java FAQs, Java/JEE FAQs

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