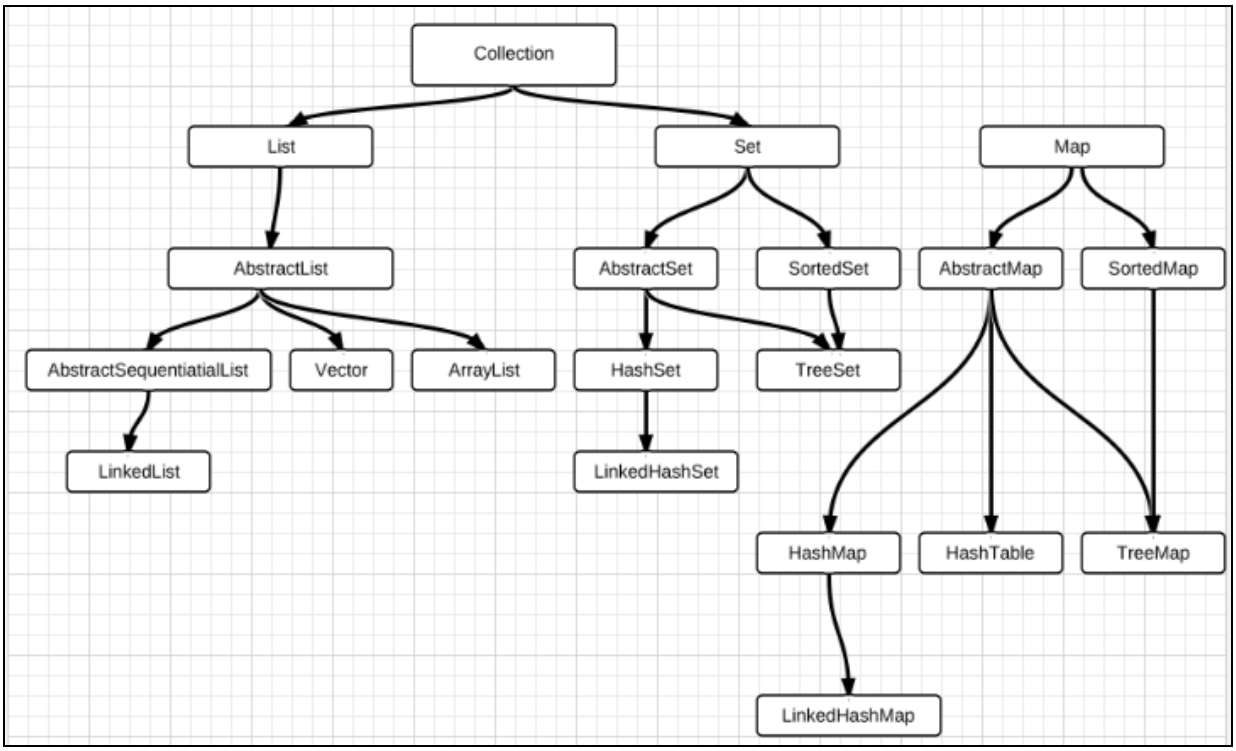
2011

Java Collection - Interview Questions

Version 1.0



1. What is List

This is an ordered collection (also known as a sequence). The List interface extends the Collection interface to define an ordered collection, permitting duplicates. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

2. What is ArrayList

It is resizable-array implementation of the List interface. If we need to support random access, without inserting or removing elements from any place to other than the end, then ArrayList offers you the optimal collection.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added an ArrayList, its capacity grows automatically.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

This implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.

List list = Collections.synchronizedList(new ArrayList(...));

The iterators returned by this class's iterator and listIterator methods are ***fail-fast***: if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException.

3. What is LinkedList

It is Doubly Linked list implementation of the List interface.

It may provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list.

This implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.

The iterators returned by this class's iterator and listIterator methods are ***fail-fast***: if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException.

4. What is difference between ArrayList & LinkedList

1. Adding and deleting at the start and middle of the ArrayList is slow, because all the later elements have to be copied forward or backward. (Using System.arrayCopy()) Whereas Linked lists are faster for inserts and deletes anywhere in the list, since all we do is update a few next and previous pointers of a node.

2. Each element of a linked list (especially a doubly linked list) uses a bit more memory than its equivalent in array list, due to the need for next and previous pointers.

3. ArrayList may also have a performance issue when the internal array fills up. The ArrayList has to create a new array and copy all the elements there. The ArrayList has a growth algorithm of (n*3)/2+1, meaning that each time the buffer is too small it will create a new one of size (n*3)/2+1 where n is the number of elements of the current buffer. Hence if we can guess the number of elements that we are going to have, then it makes sense to create a ArrayList with that capacity during object creation

(using constructor new ArrayList(capacity)). Whereas LinkedList should not have such capacity issues.

5. What is Vector

The Vector class implements a grow able array of objects. Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

This implementation is synchronized.

6. What is difference between ArrayList & Vector

1. Vector & ArrayList both classes are implemented using dynamically resizable arrays, providing fast random access and fast traversal. ArrayList and Vector class both implement the List interface.
2. Vector is synchronized whereas ArrayList is not. Even though Vector class is synchronized, still when you want programs to run in multithreading environment using ArrayList with Collections.synchronizedList () is recommended over Vector.
3. ArrayList has no default size while vector has a default size of 10.
4. The Enumerations returned by Vector's elements method are not fail-fast. Whereas ArrayList does not have any method returning Enumerations.

7. What is Set

It is a collection that contains no duplicate elements. This also means that a set can contain at most one null value.

Set has 3 general purpose implementations.

- HashSet
- TreeSet
- LinkedHashSet

8. What is HashSet

The HashSet class implements the Set interface and it uses a hash table data structure for its implementation. This class permits the null element.

A HashSet does not guarantee any ordering of the elements. It does not guarantee that the order will remain constant over time.

This implementation is not synchronized. If multiple threads access an HashSet instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.

Set s = Collections.synchronizedSet(new HashSet (...));

The HashSet uses a hash code to place items in their location. When an object is inserted, it finds the appropriate bucket corresponding to the objects by using getHashCode () method on the key to determine where it should be put in its internal buckets. After that it calls the equals method to compare the equality of objects in the bucket. If an equals returns true with any object of the bucket then the new object is not inserted.

9. What is LinkedHashSet

A LinkedHashSet is an **ordered version** of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

This implementation is not synchronized. If multiple threads access a LinkedHashSet instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.

Set s = Collections.synchronizedSet(new LinkedHashSet (...));

10. What is SortedSet

A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements.

All elements inserted into an sorted set must implement the Comparable interface (or be accepted by the specified Comparator). Furthermore, all such elements must be mutually comparable: e1.compareTo(e2) (or comparator.compare(e1, e2)) must not throw a ClassCastException for any elements e1 and e2 in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

11. What is TreeSet

TreeSet provides an implementation of the **Set** interface that uses a tree for storage. Objects are stored in sorted, ascending order, sorted according to the *natural order* of the elements. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

This implementation is not synchronized. If multiple threads access a LinkedHashSet instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.

SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

12. What is Map?

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements.

13. What is HashMap?

It is hash table based implementation of Map interface. *It allows null key and null values*. The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls. This class makes *no guarantees as to the order of the map*; in particular, it does not guarantee that the order will remain constant over time.

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.

Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally.

Map m = Collections.synchronizedMap(new HashMap(...));

14. What is HashTable?

This class used a hash table data structure to store the data and maps keys to values. Any non-null object can be used as a key or as a value. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hash table, the objects used as keys must implement the hashCode method and the equals method.

It is synchronized.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the rehash method.

The Iterators returned by the iterator and listIterator methods of the Collections returned by all of Hashtable's "collection view methods" are *fail-fast*. The Enumerations returned by Hashtable's keys and values methods are *not* fail-fast.

15. What is TreeMap?

Red-Black tree based implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the *natural order* for the key's class.

Note that this implementation is not synchronized**.** If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally.

Map m = Collections.synchronizedMap(new TreeMap(...));

16. What is SortedMap?

A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time.

All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) (or comparator.compare(k1, k2)) must not throw a ClassCastException for any elements k1 and k2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

17. What is LinkedHashMap?

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (*insertion-order*). Note that insertion order is not affected if a key is *re-inserted* into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)

This class provides all of the optional Map operations, and permits null elements. Note that this implementation is not synchronized**.**

18. Why to override equals() and hashCode()? and How i can implement both equals() and hashCode() for Set ?

If we are implementing HashSet to store unique object then we need to implement equals () and hashCode method. If two objects are equal according to the equals () method, they must have the same hashCode() value (although the reverse is not generally true).

Two scenarios

Case 1) : If you don't implement equals() and hashCode() method :

When you are adding objects to HashSet , HashSet checks for uniqueness using equals() and hashCode() method the class (ex. Emp class). If there is no equals () and hashCode () method the Emp class then HashSet checks default Object classes equals () and hashCode () method.

In the Object class , equals method is –

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

Under this default implementation, two references are equal only if they refer to the exact same object. Similarly, the default implementation of hashCode () provided by Object is derived by mapping the memory address of the object to an integer value.

This will fail to check if two Emp object with same employee name .
For Example :

```
Emp emp1 = new Emp();
emp1.setName("sat");
```

```
Emp emp2 = new Emp();
emp2.setName("sat");
```

Both the objects are same but based on the default above method both objects are different because references and hashCode are different. So in the HashSet you can find the duplicate emp objects.

To overcome the issue equals() and hashcode() method need to override.

Case 2) : If you override equals() and hashcode() method

Example : implement equals and hashcode

```
public class Emp {
   private long empId;
   String name = "";
public boolean equals(Object o) {
      if (o == this)
      return true;
      if (!(o instanceof Emp))
      return false;
      Emp emp = (Emp)o;
      return emp. empId == empId &&
      emp. name == name;

      }

   public int hashCode(){
      int result = 10;
      result = result * new Integer(String.valueOf(empId)).intValue();
      return result;
   }
}
```

In the equals() , it check for name is same or not. This way you can find out the objects are equals or not.

In the hashCode () also it return some unique value for each object. In this way if two Emp object has same empId then it will say both are same object.

Now HashSet store only unique objects.
If you do

```
Emp emp1 = new Emp();
emp1.setName("sat");
Emp emp2 = new Emp();
emp2.setName("sat");

if(emp1.equals(emp2)){
        System.out.println("equal");
    }
```

This will print : equal

19. What is the difference between java.util.Iterator and java.util.ListIterator?

Iterator: Enables you to traverse through a collection in the forward direction only, for obtaining or removing elements

ListIterator: extends Iterator, and allows bidirectional traversal of list and also allows the modification of elements.

20. Difference between HashMap and HashTable?

1. HashMap allows null keys and null values while HashTable doesn't allow null keys and values.
2. HashMap is not synchronized while HashTable is synchronized.
3. Iterator in the HashMap is fail-fast while the enumerator for the Hashtable isn't.

21. Where will you use Hashtable and where will you use HashMap?

There are multiple aspects to this decision:

1. The basic difference between a Hashtable and HashMap is that, Hashtable is synchronized while HashMap is not. Thus whenever there is a possibility of multiple threads accessing the same instance, one should use Hashtable. While if not multiple threads are going to access the same instance then use HashMap. Non synchronized data structure will give better performance than the synchronized one.

2. If there is a possibility in future that - there can be a scenario when you may require retaining the order of objects in the Collection with key-value pair then HashMap can be a good choice. As one of HashMap's subclasses is LinkedHashMap, so in the event that

you'd want predictable iteration order (which is insertion order by default), you can easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable. Also if you have multiple thread accessing you HashMap then Collections.synchronizedMap () method can be leveraged. Overall HashMap gives you more flexibility in terms of possible future changes.

22. What is the Difference between Enumeration and Iterator interface?

Iterators differ from enumerations in following ways:

1. Enumeration contains 2 methods namely hasMoreElements() & nextElement() whereas Iterator contains three methods namely hasNext(), next(),remove().
2. Iterator adds an optional remove operation, and has shorter method names. Using remove () we can delete the objects but Enumeration interface does not support this feature.

23. What is the difference between Sorting performance of Arrays.sort() vs Collections.sort() ? Which one is faster? Which one to use and when?

Both methods have same algorithm the only difference is type of input to them. Collections.sort() has a input as List so it does a translation of List to array and vice versa which is an additional step while sorting. So this should be used when you are trying to sort a list. Arrays.sort is for arrays so the sorting is done directly on the array. So clearly it should be used when you have a array available with you and you want to sort it.