1. **What is the importance of Collection API?**

Ans:- What if u want to have a collection of data items ?

e.g. Strings, numbers, characters ,objects etc.

one way is to create array. They are efficient.

But if we have following needs:

1. Dynamic array
2. Sorted data
3. uniqueness
4. key-value storage
5. effective way of persisting

Arrays cannot fulfill above needs. We have to use containers for that.

**Containers**

Give us advantages such as

* Dynamic
* Sorted Order
* Uniqueness
* Thread-safe
* Performance
* Key-value storage, convenient for retrieval by passing a key.
* Provide effective way of storing and maintaining data inside the file system.

**Iterators**

Allow us to traverse through the container.

**Algorithms**

Allow us to perform tasks such as min,max, sort etc. on the data structure.

1. **Explain in brief List, Set and Map.**

Ans:

**Set** is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

**List** is an ordered collection and can contain duplicate elements. You can access any element from its index. List is more like array with dynamic length.

A **Map** is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.

**3) What is difference between Enumeration and Iterator interface?**

ans:- both are used to traverse through the collection implementations. The difference is Iterators allow the caller to remove elements from the underlying collection that is not possible with Enumeration. Iterator method names have been improved to make its functionality clear.

1. **What is the difference between ArrayList and LinkedList?**

* ArrayList uses a **dynamic array**, making random access (get(index)) fast but insertions/removals slow.
* LinkedList uses a **doubly linked list**, making frequent insertions and deletions faster but access slower.

1. **What is the difference between HashSet and TreeSet?**

* HashSet does **not maintain any order** and provides O(1) time complexity for operations.
* TreeSet maintains **sorted order** and provides O(log n) time complexity for operations.
* Use HashSet when order is not required and performance is important, and TreeSet when sorting is needed.

1. How does put method work in HashMap?

Ans:

put operation

hashCode() is invoked on the key to determine the bucket.

hashcode - key of the first entry

subsequent entries

hashcode - different - different bucket

same [ hash collision case]

== - true - overwrite the value

false

equals - true - overwrite the value

false - linked list will be formed within a bucket i.e. same bucket having different Entries [Entry]

1. How does get method work in HashMap?

Ans:

get operation

hashcode - on the given key inside "get()" method

bucket is determined for search

== - true - get the value

false

equals - true - get the value

false - linked list will be traverse and subsequently == and equals are invoked.

1. **What do you mean by HashMap has by default capacity 16 and load factor 0.75?**

Ans:

**Default Capacity** is **16**, meaning there are 16 buckets initially.

**Load Factor** is **0.75**, meaning resizing occurs when the map is 75% full i.e. **capacity \* load factor** (e.g., 16 \* 0.75 = 12). When 12 buckets are full, while adding the **13th element**, the capacity **expands from 16 to 32.**

1. **What is the difference between HashMap, LinkedHashMap, and TreeMap?**

Ans:

HashMap does not maintain any order and provides **O(1) access time**.

LinkedHashMap maintains **insertion order**.

TreeMap stores keys in **sorted order** and has **O(log n) access time**.

1. **What is the difference between fail-fast and fail-safe iterators?**

Ans:

**Fail-fast iterators** (ArrayList, HashMap) throw ConcurrentModificationException if a collection is modified during iteration.

**Fail-safe iterators** (ConcurrentHashMap, CopyOnWriteArrayList) allow modification inside original collection by iterating over a copy of the collection, preventing exceptions.

1. **What is ConcurrentModificationException?**

Ans:

ConcurrentModificationException is a **runtime exception** that occurs when a collection (like ArrayList, HashMap, etc.) is **modified while being iterated** using iterators that do not support concurrent modifications.

1. **How does CopyOnArrayList work?**

Ans:

When we create iterator on CopyOnWriteArrayList, it creates a snapshot of original array on which iteration can be done. Now when we modify the arraylist ( add, remove etc) , modifications can be done on the original arraylist.

1. What is ConcurrentHashMap and how it is different from Hashtable and HashMap?

Ans:

**HashMap (Not Thread-Safe)**

* Allows **one null key** and multiple null values.
* **Not synchronized**, meaning multiple threads modifying it **may cause race conditions**.
* Requires **explicit synchronization** for thread safety (e.g., using Collections.synchronizedMap()).
* Faster in a **single-threaded** environment.

**2 Hashtable (Thread-Safe but Slow)**

* **Synchronized on every method**, meaning only **one thread can access it at a time**.
* **No null keys or null values** allowed.
* Slower because of **full map-level locking**, even for reads.

**3 ConcurrentHashMap (Thread-Safe and Fast)**

* **Uses bucket-level locks (Java 8+), improving concurrency.**

Only 1 bucket locked per write operation, so e.g. **if you insert 4 unique elements**, and they are mapped to **4 different buckets**, then **4 separate buckets will be locked individually during each write operation**.

* **Multiple threads can read and write simultaneously** (unlike Hashtable).
* **Does not allow null keys or null values.**
* **Faster than Hashtable in multi-threaded environments.**

1. What is the difference between Comparable and Comparator interfaces?

Ans:

Both Comparable and Comparator are used to compare objects in Java, but they have different use cases and implementations.

Comparable (Natural Ordering)

Defines default sorting logic within the class itself.

Uses compareTo(T o) method.

Modifies the class itself to implement sorting.

Used when only one sorting logic is needed.

Comparator (Custom Sorting)

Used for custom sorting outside the class.

Uses compare(T o1, T o2) method.

Does not modify the original class (keeps it clean).

Useful when we need multiple sorting criteria.

Imp:

Use Comparable when default sorting is needed inside the class.

Use Comparator when custom sorting or multiple sorting criteria are required.

1. **What is WeakHashMap and how is it different from HashMap?**

Ans:

WeakHashMap is a special implementation of Map in Java where keys are weak references. This means that if a key is no longer strongly referenced elsewhere, the Garbage Collector (GC) will automatically remove it from the map.

It is part of java.util and is useful in caching scenarios where you don’t want to prevent objects from being garbage collected.HashMap keeps **strong references**, preventing garbage collection.

HashMap keeps **strong references**, preventing garbage collection.

Comparative study:

Map<String, String> myMap = **new** HashMap<>();

String key = **new** String("temp");

myMap.put(key, "value");

System.***out***.println("myMap before making key null\t"+myMap);

key = **null**;

System.*gc*(); // Key may be garbage collected

System.***out***.println("myMap after making key null\t"+myMap);

System.***out***.println("Let's see WeakHashMap now");

Map<String, String> weakMap = **new** WeakHashMap<>();

String key1 = **new** String("hello");

weakMap.put(key1, "value1");

System.***out***.println("weakMap before making key null\t"+weakMap);

key1 = **null**;

System.*gc*(); // Key may be garbage collected

System.***out***.println("weakMap after making key null\t"+weakMap);

1. **Can you store null keys and values in a HashMap?**

Ans:

Yes, HashMap allows **one null key** and **multiple null values**.  
However, Hashtable **does not allow null keys or values**.

1. Why is TreeSet not synchronized?

ans:

TreeSet is not synchronized to avoid performance overhead.

To make it thread-safe, use:

SortedSet<Integer> syncSet = Collections.synchronizedSortedSet(new TreeSet<>());

1. **What are different ways to iterate over a list?**

Ans:- We can iterate over a list in two different ways – using iterator and using for-each loop.

|  |  |
| --- | --- |
|  | List<String> strList = new ArrayList<String>();  //using for-each loop  for(String obj : strList)  {      System.out.println(obj);  }  //using iterator  Iterator<String> it = strList.iterator();  while(it.hasNext())  {      String obj = it.next();      System.out.println(obj);  } |

Using iterator is more thread-safe because it makes sure that if underlying list elements are modified, it will throw **ConcurrentModificationException**.

1. **Why there are no concrete implementations of Iterator interface?**

Ans: Iterator interface declare methods for iterating a collection but its implementation is responsibility of the Collection implementation classes. Every collection class that returns an iterator for traversing has its own Iterator implementation nested class.  
This allows collection classes to choose whether iterator is fail-fast or fail-safe. For example ArrayList iterator is fail-fast whereas CopyOnWriteArrayList iterator is fail-safe.

Each collection type (e.g., ArrayList, LinkedList, HashSet) needs a different iteration logic based on its internal structure.

The responsibility of implementing Iterator is left to each collection’s specific inner class.

1. What are the changes Java8 brought to HashMap as far as linked list is created inside the bucket in case of Hash collision?

Ans:

**Changes in HashMap Implementation in Java 8 (Handling Hash Collisions Efficiently)**

In **Java 7 and earlier**, when multiple keys map to the same bucket (due to hash collisions), **entries were stored as a linked list inside the bucket**.

However, **Java 8 introduced a major optimization**:  
If a bucket contains **many entries (i.e., high collision rate)**, the linked list is **converted into a balanced binary search tree (BST)** for better performance.

**Why Was This Change Introduced?**

* In Java 7, when multiple keys collide and fall into the same bucket, they form a **linked list**.
* Searching for a key in a **linked list takes O(n) time** in the worst case.
* Java 8 **replaces the linked list with a balanced tree** (TreeNode) **if the number of elements in a bucket exceeds a threshold (8)**.
* **Tree-based lookups take O(log n) time**, which improves performance for large HashMaps.

1. **How HashSet store elements?**

Ans:

You must know that HashMap store key-value pairs, with one condition i.e. keys will be unique. HashSet uses Map’s this feature to ensure uniqueness of elements. In HashSet class, a map declaration is as below:

|  |  |
| --- | --- |
|  | private transient HashMap<E,Object> map;    //This is added as value for each key  private static final Object PRESENT = new Object(); |

So **when you store a element in HashSet, it stores the element as key in map and “PRESENT” object as value**. (See declaration above).

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

1. What is the difference between HashMap and TreeMap?

Ans:

HashMap is used to store key-value pairs and allows to perform many operations on such collection of pairs.

TreeMap is special form of HashMap. **It maintains the ordering of keys** which is missing in HashMap class. This ordering is **by default “natural ordering”**. The default ordering can be override by providing an instance of Comparator class, whose compare method will be used to maintain ordering of keys.

**All keys inserted into the map must implement the Comparable interface** (this is necessary to decide the ordering).

1. **if you want to store object of any user defined class as a key inside TreeMap or as an element inside TreeSet, what care u will take?**

Ans:- The class has to either implement **Comparable** or **Comparator** interface.

1. **Difference between Iterator and Enumeration.**

Ans:- Iterators differ from enumerations in three ways:

* Iterators allow the caller to remove elements from the underlying collection during the iteration with its remove() method. You can not add/remove elements from a collection when using enumerator.
* Enumeration is available in legacy classes i.e Vector/Stack etc. whereas Iterator is available in all modern collection classes.
* Another minor difference is that Iterator has improved method names e.g. Enumeration.hasMoreElement() has become Iterator.hasNext(), Enumeration.nextElement() has become Iterator.next() etc.

1. **Difference between Iterator and ListIterator.**

Ans:- There are three Differences are there:

* We can use Iterator to traverse Set and List and also Map type of Objects. But List Iterator can be used to traverse for List type Objects, but not for Set type of Objects.
* By using Iterator we can retrieve the elements from Collection Object in forward direction only whereas List Iterator, which allows you to traverse in either directions using hasPrevious() and previous() methods.
* ListIterator allows you modify the list using add() remove() methods. Using Iterator you can not add, only remove the elements.

1. **Difference between ArrayList and LinkedList.**

* Ans:- LinkedList store elements within a doubly-linked list data structure. ArrayList store elements within a dynamically resizing array.
* LinkedList allows for constant-time insertions or removals, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but grabbing an element in the middle takes time proportional to the size of the list. ArrayLists, on the other hand, allow random access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap.
* LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

1. **How to make a collection read only?**

Ans:- Use following methods:

* Collections.unmodifiableList(list);
* Collections.unmodifiableSet(set);
* Collections.unmodifiableMap(map);

These methods takes collection parameter and return a new read-only collection with same elements as in original collection.

1. **How to make a collection thread safe?**

Ans:- Use below methods:

* Collections.synchronizedList(list);
* Collections.synchronizedSet(set);
* Collections.synchronizedMap(map);

Above methods take collection as parameter and return same type of collection which are synchronized and thread safe.

1. **What is UnsupportedOperationException?**

Ans:- This exception is thrown **on invoked methods which are not supported by actual collection type**.

For example, if you make a read-only list list using “Collections.unmodifiableList(list)” and then call add() or remove() method, what should happen. It should clearly throw UnsupportedOperationException. This exception is also thrown when we try to invoke “add()” or “remove()” methods on fail-safe iterator of CopyOnWriteArrayList or ConcurrentHashMap.

1. **How do you remove an entry from a Collection? and subsequently what is difference between remove() method of Collection and remove() method of Iterator, which one you will use, while removing elements during iteration?**

ans:- Collection interface defines remove(Object obj) method to remove objects from Collection. List interface adds another method remove(int index), which is used to remove object at specific index. You can use any of these method to remove an entry from Collection, while not iterating. Things change, when you iterate. Suppose you are traversing a List and removing only certain elements based on logic, then you need to use Iterator's remove() method. This method removes current element from Iterator's perspective. If you use Collection's or List's remove() method during iteration then your code will throw ConcurrentModificationException. That's why it's advised to use Iterator remove() method to remove objects from Collection.

30) **What is hash collision? How does map implementations deal with it?**

Ans:- When two or more different keys produce the same hash value, it’s called a Hash C**ollision**. A map implementation deals with collisions by storing all the key/object pairs that have the same hash value in a same bucket ( i.e. in the form of linked list). Retrieving an object that resulted in a collision when it was stored is a two-step process. The key will be hashed to find the location where the key/object pair should be. The linked list then has to be searched to sort out the particular key you are searching on from all others that have the same hash value.

1. 31) **What is the difference between “put()” method of Map vs “add()” method of Set?**

Ans:- **“put” method of Map**

\* If the map previously contained a mapping for the key, the old

\* value is replaced.

public V put(K key, V value) { …….. }

**“add” method of Set**

\* Adds the specified element to this set if it is not already present.

\* If this set already contains the element, the call leaves the set

\* unchanged and returns false.

public boolean add(E e) {}

32) what do you mean by collections store references and not the copies of the java classes?

Ans:

It means that when you store any object as an element in the List/Set or key/value in the Map, their references are stored and not the copies.

e.g.

Sample s=new Sample();

If I add “s” inside the list, that would mean Sample object on the heap is referred by two references :

One on the stack i.e. s

And other inside the list. Both references refer to the same object.

However it is important to understand that the moment we serialize this list inside the filesystem , copy of Sample class object would be stored inside the list.

32) **What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?**  
ans:- If the size of the Map exceeds a given threshold defined by load-factor e.g. if load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like ArrayList,  Java HashMap re-size itself by creating a new bucket array of size twice of previous size of HashMap , and then start putting every old element into that new bucket array. This process is called rehashing because it also applies hash function to find new bucket location. 

33) **Why String, Integer and other wrapper classes are considered good keys?**  
ans:- String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because String is immutable and final,and overrides equals and hashcode() method. Other wrapper class also shares similar property. Immutabiility is required, in order to prevent changes on fields used to calculate hashCode() because if key object return different hashCode during insertion and retrieval than it won't be possible to get object from HashMap. Immutability is best as it offers other advantages as well like thread-safety, If you can  keep your hashCode same by only making certain fields final, then you go for that as well. Since equals() and hashCode() method is used during reterival of value object from HashMap, it’s important that key object correctly override these methods and follow contact. If unequal object return different hashcode than chances of collision will be less which subsequently improve performance of HashMap.

34) **How do you traverse through a collection using its Iterator?**

Ans:- To use an iterator to traverse through the contents of a collection, follow these steps:

* Obtain an iterator to the start of the collection by calling the collection implementation’s ***iterator()*** method.
* Set up a loop that makes a call to ***hasNext()***. Have the loop iterate as long as ***hasNext()*** returns **true**.
* Within the loop, obtain each element by calling **next()**.

35) **How do you decide when to use ArrayList and When to use LinkedList?**

Ans:- If you need to support random access, without inserting or removing elements from any place other than the end, then ArrayList offers the optimal collection. If, however, you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially, then LinkedList offers the better implementation.

36) **discuss the problem of thread-safety with synchronized collection classes as well as synchronized wrapper classes.**

Ans:- both are thread safe. As their individual methods are synchronized.

I.e.

Vector myvect=new Vector();

or

List list = Collections.synchronizedList(new ArrayList(...));  
  
In java.util.Collections' JavaDoc you can read that "It is imperative  
that the user manually synchronize on the returned list or vector when iterating  
over it:"  
  
synchronized(list) {  
Iterator i = list.iterator(); // Must be in synchronized block  
while (i.hasNext())  
foo(i.next());  
}

Now if synchronizedList gives me a thread-safe List implementation , why do I need to use explicit synchronization?

|  |
| --- |
| Vector and synchronizedList() both end up synchronizing each individual method call - but they do *not* provide any protection against other threads changing the thread *in between* individual method calls. So any time you rely on some sort of consistent state in between two method calls, there is a possibility of failure.   For example:   1. **for** (Foo f : fooVector) { 2. doSomethingWith(f); 3. }      1. Iterator<Foo> it = fooVector.iterator(); 2. **while** (it.hasNext()) { 3. Foo f = (Foo) it.next(); 4. doSomethingWith(f); 5. }   OK, that's normal enough. But if fooVector is also accessible to other threads which can modify it, problems can occur. What happens if something changes in between the calls to it.hasNext() and it.next()?   1. Iterator<Foo> it = fooVector.iterator(); 2. **while** (it.hasNext()) { 3. // other thread calls fooVector.remove(0) 4. Foo f = (Foo) it.next(); // now this line may throw NoSuchElementException 5. doSomethingWith(f); 6. } |
|  |

To avoid this problem, it's necessary to synchronize as described in the List.iterator() API:

1. **synchronized** (fooVector) {
2. Iterator<Foo> it = fooVector.iterator();
3. **while** (it.hasNext()) {
4. Foo f = (Foo) it.next();
5. doSomethingWith(f);
6. }
7. }

But this problem is not limited to use of Iterators. Here's another example:

1. **if** (!fooVector.isEmpty()) {
2. Foo f = fooVector.remove(0);
3. doSomethingWith(f);
4. }

Here, again - what if, in between the isEmpty() and the remove(0), another thread removes the only element in fooVector? You get a NoSuchElementException.   
  
Or a more common example, getting the last element of a List:

1. Foo lastFoo = fooList.get(fooList.size() - 1);

The call to fooList.size() must execute before fooList.get(). What happens if a single element is removed from the list, in between these two operations? NoSuchElementException. Or if a single element is added, anywhere in the list (except the very end)- oops, you just got the next-to-last element, rather than the last one.

To summarize, Both synchronized collection classes and synchronized wrapper classes seem to give people a false sense of confidence that their code is thread-safe, when it really isn't. Additional synchronization is almost always needed

37) **what is the requirement of putting any key inside TreeMap or element inside TreeSet?**

Ans:- Whether u want to put any key inside TreeMap or element inside TreeSet u need to fulfil one of the following requirements:

* 1. Key must implement “Comparable” interface so that u will define “compareTo(Object)” method and provide the strategy to compare a particular field.
  2. If key is not implementing “Comparable”, then u need to create an implementation of “Comparator” by defining “compare(Object,Object)” to provide the strategy to compare a particular field. Now pass this Comparator implementation to TreeMap or TreeSet constructor. When u put a key inside TreeMap or add an element inside TreeSet, outcome of compare() method will be considered.

**Suppose there is an Employee class. We add Employee class objects to the ArrayList. Mention the steps need to be taken , if I want to sort the objects in ArrayList using the employeeId attribute present  in Employee class.**  
a. Implement the Comparable interface for the Employee class and now to compare the objects by employeeId we will override the emp1.compareTo(emp2)  
b. We will now call Collections class sort method and pass the list as argument , that is ,  
     Collections.sort(empList)

38) How can you make non-thread safe classes like ArrayList or HashMap into thread-safe classes?

Ans:

in collection api there are some classes which are thread safe (synchronized non-static methods) and other classes are non-thread safe.

we can make non-thread safe classes as thread-safe.

e.g. we can make ArrayList which is non-thread safe as thread safe.

List<Integer>mylist=new ArrayList<Integer>();

List<Integer>mylist1=Collections.synchronizedList(mylist);

how to convert HashMap which is non-thread safe to thread safe

Map<String,Integer>mymap=new HashMap<>();

Map<String,Integer>mymap1=Collections.synchronizedMap(mymap);