Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element	Thread Safety
ArrayList	Yes	Yes	No	Yes	Yes	No
LinkedList	Yes	No	No	Yes	Yes	No
HashSet	No	No	No	No	Yes	No
TreeSet	Yes	No	No	No	No	No
HashMap	No	Yes	Yes	No	Yes	No
TreeMap	Yes	Yes	Yes	No	No	No
Vector	Yes	Yes	No	Yes	Yes	Yes
Hashtable	No	Yes	Yes	No	No	Yes
Properties	No	Yes	Yes	No	No	Yes
Stack	Yes	No	No	Yes	Yes	Yes
CopyOnWriteArrayList	Yes	Yes	No	Yes	Yes	Yes
ConcurrentHashMap	No	Yes	Yes	No	No	Yes
CopyOnWriteArraySet	No	No	No	No	Yes	Yes

1. What is Java Collections Framework? List out some benefits of Collections framework?

Collections are used in every programming language and initial java release contained few classes for collections: **Vector**, **Stack**, **Hashtable**, **Array**. But looking at the larger scope and usage, Java 1.2 came up with Collections. Java Collections have come through a long way with usage of Generics and Concurrent Collection classes for thread-safe operations. It also includes blocking

interfaces and their implementations in java concurrent package.
Some of the benefits of collections framework are;

2. What is the benefit of Generics in Collections Framework?

Java 1.5 came with Generics and all collection interfaces and implementations use it heavily. Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error. This avoids ClassCastException at Runtime because you will get the error at compilation. Also Generics make code clean since we don't need to use casting and *instanceof* operator. I would highly recommend to go through **Java Generic Tutorial** to understand generics in a better way.

3. What are the basic interfaces of Java Collections Framework?

Collection is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface.

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 it's 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.

Some other interfaces
are `Queue`, `Dequeue`, `Iterator`, `SortedSet`, `SortedMap` and `ListIterator`.

4. What is an Iterator?

Iterator interface provides methods to iterate over any Collection. We can get iterator instance from a Collection using *iterator()* method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Java Collection iterator provides a generic way for traversal through the elements of a collection and implements **Iterator Design Pattern**.

5. What is difference between Enumeration and Iterator interface?

Enumeration is twice as fast as Iterator and uses very less memory. Enumeration is very basic and fits to basic needs. But Iterator is much safer as compared to Enumeration because it always denies other threads to modify the collection object which is being iterated by it.

Iterator takes the place of Enumeration in the Java Collections Framework. 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 it's functionality clear.

6. Why there is not method like Iterator.add() to add elements to the collection?

The semantics are unclear, given that the contract for Iterator makes no guarantees about the order of iteration. Note, however, that ListIterator does provide an add operation, as it does guarantee the order of the iteration.

7. What is different between Iterator and ListIterator?

- We can use Iterator to traverse Set and List collections whereas ListIterator can be used with Lists only.
- Iterator can traverse in forward direction only whereas ListIterator can be used to traverse in both the directions.

- ListIterator inherits from Iterator interface and comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.

8. What are different ways to iterate over a list?

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

```
List<String> strList = new ArrayList<>();

//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`.

9. What do you understand by iterator fail-fast property?

Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element. If there are any modifications found, it throws`ConcurrentModificationException`. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

10. What is difference between fail-fast and fail-safe?

Iterator fail-safe property work with the clone of underlying collection, hence it's not affected by any modification in the collection. By design, all the collection classes in `java.util` package are fail-fast whereas collection classes in `java.util.concurrent` are fail-safe.

Fail-fast iterators throw ConcurrentModificationException whereas fail-safe iterator never throws ConcurrentModificationException.

Check this post for CopyOnWriteArrayList Example.

11. Java ArrayList and ConcurrentModification Exception

ArrayList is one of the basic implementations of List interface and it's part of **Java Collections Framework**. We can use **iterator** to traverse through ArrayList elements.

```
package com.journaldev.collections;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class ConcurrentListExample {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        // get the iterator
        Iterator<String> it = list.iterator();

        //manipulate list while iterating
        while(it.hasNext()){
            System.out.println("list is:"+list);
            String str = it.next();
            System.out.println(str);
            if(str.equals("2"))list.remove("5");
            if(str.equals("3"))list.add("3 found");
            //below code don't throw ConcurrentModificationException
            //because it doesn't change modCount variable of list
            if(str.equals("4")) list.set(1, "4");
        }
    }
```

```
}
```

When we run above program, we get `java.util.ConcurrentModificationException` as soon as the ArrayList is modified.

It happens because ArrayList iterator is **fail-fast** by design. What it means is that once the iterator is created, if the ArrayList is modified, it throws **ConcurrentModificationException**.

If you check the console log, you will notice that exception is thrown by Iterator `next()` method. If you will look into the ArrayList source code, following method is called everytime we invoke next() on iterator that throws exception.

```
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

Here `modCount` is the ArrayList variable that holds the modification count and every time we use `add`, `remove`or `trimToSize` method, it increments. `expectedModCount` is the iterator variable that is initialized when we create iterator with same value as `modCount`. This explains why we don't get exception if we use `set` method to replace any existing element.

So basically iterator throws `ConcurrentModificationException` if list size is changed.

12. CopyOnWriteArrayList in Java

CopyOnWriteArrayList in Java is a thread safe implementation of List interface. CopyOnWriteArrayList was added in Java 1.5 and part of Collections framework.

Sometimes we want to add or remove elements from the list if we find some specific element, in that case we should use concurrent collection class – `CopyOnWriteArrayList`. This is a thread-safe variant of java.util.ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

CopyOnWriteArrayList introduces extra overload to the processing but it's very effective when number of modifications are minimal compared to number of traversal operations.

If we change the implementation to `CopyOnWriteArrayList`, then we don't get any exception and below is the output produced.

Notice that it allows the modification of list, but it doesn't change the iterator and we get same elements as it was on original list.

13. How to avoid ConcurrentModification Exception while iterating a collection?

We can use concurrent collection classes to avoid `ConcurrentModificationException` while iterating over a collection, for example CopyOnWriteArrayList instead of ArrayList.
Check this post for ConcurrentHashMap Example.

14. ConcurrentHashMap Example

`ConcurrentHashMap` is the class that is similar to HashMap but works fine when you try to modify your map at runtime.

```
package com.journaldev.util;
```

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

	public static void main(String[] args) {

		//ConcurrentHashMap
		Map<String,String> myMap = new ConcurrentHashMap<String,String>();
		myMap.put("1", "1");
		myMap.put("2", "1");
		myMap.put("3", "1");
		myMap.put("4", "1");
		myMap.put("5", "1");
		myMap.put("6", "1");
		System.out.println("ConcurrentHashMap before iterator: "+myMap);
		Iterator<String> it = myMap.keySet().iterator();

		while(it.hasNext()){
			String key = it.next();
			if(key.equals("3")) myMap.put(key+"new", "new3");
		}
		System.out.println("ConcurrentHashMap after iterator: "+myMap);

		//HashMap
		myMap = new HashMap<String,String>();
		myMap.put("1", "1");
		myMap.put("2", "1");
		myMap.put("3", "1");
		myMap.put("4", "1");
		myMap.put("5", "1");
		myMap.put("6", "1");
		System.out.println("HashMap before iterator: "+myMap);
		Iterator<String> it1 = myMap.keySet().iterator();

		while(it1.hasNext()){
			String key = it1.next();
			if(key.equals("3")) myMap.put(key+"new", "new3");
		}
		System.out.println("HashMap after iterator: "+myMap);
	}

}
```

When we try to run the above class, output is

```
ConcurrentHashMap before iterator: {1=1, 5=1, 6=1, 3=1, 4=1, 2=1}
ConcurrentHashMap after iterator: {1=1, 3new=new3, 5=1, 6=1, 3=1, 4=1, 2=1}
HashMap before iterator: {3=1, 2=1, 1=1, 6=1, 5=1, 4=1}
Exception in thread "main" java.util.ConcurrentModificationException
	at java.util.HashMap$HashIterator.nextEntry(HashMap.java:793)
	at java.util.HashMap$KeyIterator.next(HashMap.java:828)
	at com.test.ConcurrentHashMapExample.main(ConcurrentHashMapExample.java:44)
```

Looking at the output, its clear that **ConcurrentHashMap** takes care of any new entry in the map whereas HashMap throws `ConcurrentModificationException`.

HashMap contains a variable to count the number of modifications and iterator use it when you call its next() function to get the next entry.

HashMap.java

```
/**
     * The number of times this HashMap has been structurally modified
     * Structural modifications are those that change the number of mappings in
     * the HashMap or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the HashMap fail-fast.  (See ConcurrentModificationException).
     */
```

```
    transient volatile int modCount;
```

Now to prove above point, lets change the code a little bit to come out of the iterator loop when we insert the new entry. All we need to do is add a break statement after the put call.

```
if(key.equals("3")){
                                        myMap.put(key+"new", "new3");
                                        break;
                                }
```

Now execute the modified code and the output will be:

```
ConcurrentHashMap before iterator: {1=1, 5=1, 6=1, 3=1, 4=1, 2=1}
ConcurrentHashMap after iterator: {1=1, 3new=new3, 5=1, 6=1, 3=1, 4=1, 2=1}
HashMap before iterator: {3=1, 2=1, 1=1, 6=1, 5=1, 4=1}
HashMap after iterator: {3=1, 2=1, 1=1, 3new=new3, 6=1, 5=1, 4=1}
```

Finally, what if we won't add a new entry but update the existing key-value pair. Will it throw exception?

Change the code in the original program and check yourself.

```
//myMap.put(key+"new", "new3");
myMap.put(key, "new3");
```

15. What is UnsupportedOperationException?

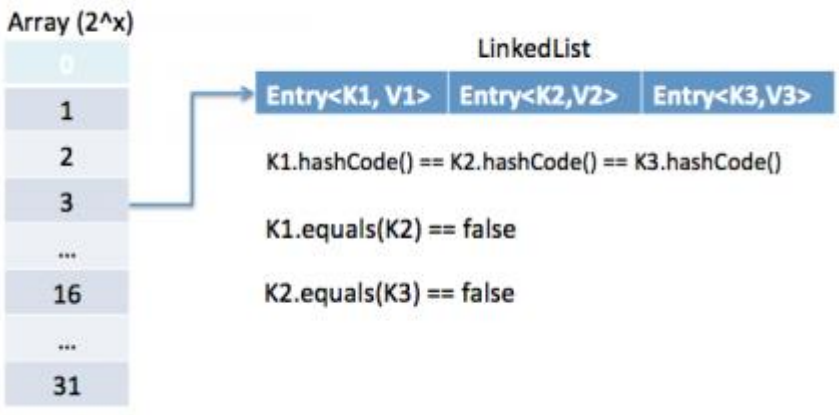
`UnsupportedOperationException` is the exception used to indicate that the operation is not supported. It's used extensively in JDK classes, in collections framework`java.util.Collections.UnmodifiableCollection` throws this exception for all `add` and `remove`operations.

16. How HashMap works in Java?

HashMap stores key-value pair in `Map.Entry` static nested class implementation. HashMap works on hashing algorithm and uses hashCode() and equals() method in `put` and `get` methods.

When we call `put` method by passing key-value pair, HashMap uses Key hashCode() with hashing to find out the index to store the key-value pair. The Entry is stored in the LinkedList, so if there are already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and store this key-value Entry.

When we call `get` method by passing Key, again it uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return it's value. Below image will explain these detail clearly.



The other important things to know about HashMap are capacity, load factor, threshold resizing. HashMap initial default capacity is **16** and load factor is 0.75. Threshold is capacity multiplied by load factor and whenever we try to add an entry, if map size is greater than threshold, HashMap rehashes the contents of map into a new array with a larger capacity. The capacity is always power of 2, so if you know that you need to store a large number of key-value pairs, for example in caching data from database, it's good idea to initialize the HashMap with correct capacity and load factor.

17. What is the importance of hashCode() and equals() methods?

HashMap uses Key object hashCode() and equals() method to determine the index to put the key-value pair. These methods are also used when we try to get value from HashMap. If these methods are not implemented correctly, two different Key's might produce same hashCode() and equals() output and in that case rather than storing it at different location, HashMap will consider them same and overwrite them.

Similarly all the collection classes that doesn't store duplicate data use hashCode() and equals() to find duplicates, so it's very important to implement them correctly. The implementation of equals() and hashCode() should follow these rules.

- If `o1.equals(o2)`, then `o1.hashCode() == o2.hashCode()` should always be `true`.
- If `o1.hashCode() == o2.hashCode` is true, it doesn't mean that `o1.equals(o2)` will be `true`.

18. Can we use any class as Map key?

We can use any class as Map Key, however following points should be considered before using them.

- If the class overrides equals() method, it should also override hashCode() method.
- The class should follow the rules associated with equals() and hashCode() for all instances. Please refer earlier question for these rules.
- If a class field is not used in equals(), you should not use it in hashCode() method.
- Best practice for user defined key class is to make it immutable, so that hashCode() value can be cached for fast performance. Also immutable classes make sure that hashCode() and equals() will not change in future that will solve any issue with mutability.
  For example, let's say I have a class `MyKey` that I am using for HashMap key.

```
//MyKey name argument passed is used for equals() and hashCode()
MyKey key = new MyKey("Pankaj"); //assume hashCode=1234
myHashMap.put(key, "Value");

// Below code will change the key hashCode() and equals()
// but it's location is not changed.
key.setName("Amit"); //assume new hashCode=7890

//below will return null, because HashMap will try to look for key
//in the same index as it was stored but since key is mutated,
```

```
• //there will be no match and it will return null.
```

```
myHashMap.get(new MyKey("Pankaj"));
```

This is the reason why String and Integer are mostly used as HashMap keys.

19. What are different Collection views provided by Map interface?

Map interface provides three collection views:

0. **Set keySet()**: Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.
1. **Collection values()**: Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.
2. **Set<Map.Entry<K, V>> entrySet()**: Returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own

remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

20. What is difference between HashMap and Hashtable?

HashMap and Hashtable both implements Map interface and looks similar, however there are following difference between HashMap and Hashtable.

0. HashMap allows null key and values whereas Hashtable doesn't allow null key and values.
1. Hashtable is synchronized but HashMap is not synchronized. So HashMap is better for single threaded environment, Hashtable is suitable for multi-threaded environment.
2. `LinkedHashMap` was introduced in Java 1.4 as a subclass of HashMap, so incase you want iteration order, you can easily switch from HashMap to LinkedHashMap but that is not the case with Hashtable whose iteration order is unpredictable.
3. HashMap provides Set of keys to iterate and hence it's fail-fast but Hashtable provides Enumeration of keys that doesn't support this feature.
4. Hashtable is considered to be legacy class and if you are looking for modifications of Map while iterating, you should use ConcurrentHashMap.

21. How to decide between HashMap and TreeMap?

For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative. If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a HashMap, then convert the map to a TreeMap for sorted key traversal.

22. What are similarities and difference between ArrayList and Vector?

ArrayList and Vector are similar classes in many ways.

0. Both are index based and backed up by an array internally.
1. Both maintains the order of insertion and we can get the elements in the order of insertion.
2. The iterator implementations of ArrayList and Vector both are fail-fast by design.
3. ArrayList and Vector both allows null values and random access to element using index number.

These are the differences between ArrayList and Vector.

4. Vector is synchronized whereas ArrayList is not synchronized. However if you are looking for modification of list while iterating, you should use CopyOnWriteArrayList.
5. ArrayList is faster than Vector because it doesn't have any overhead because of synchronization.
6. ArrayList is more versatile because we can get synchronized list or read-only list from it easily using Collections utility class.

23. What is difference between Array and ArrayList? When will you use Array over ArrayList?

Arrays can contain primitive or Objects whereas ArrayList can contain only Objects.
Arrays are fixed size whereas ArrayList size is dynamic.
Arrays doesn't provide a lot of features like ArrayList, such as addAll, removeAll, iterator etc.

Although ArrayList is the obvious choice when we work on list, there are few times when array are good to use.

- If the size of list is fixed and mostly used to store and traverse them.
- For list of primitive data types, although Collections use autoboxing to reduce the coding effort but still it makes them slow when working on fixed size primitive data types.
- If you are working on fixed multi-dimensional situation, using [][] is far more easier than List<List<>>

24. What is difference between ArrayList and LinkedList?

ArrayList and LinkedList both implement List interface but there are some differences between them.

0. ArrayList is an index based data structure backed by Array, so it provides random access to it's elements with performance as O(1) but LinkedList stores data as list of nodes where every node is linked to it's previous and next node. So even though there is a method to get the element using index, internally it traverse from start to reach at the index node and then return the element, so performance is O(n) that is slower than ArrayList.
1. Insertion, addition or removal of an element is faster in LinkedList compared to ArrayList because there is no concept of resizing array or updating index when element is added in middle.
2. LinkedList consumes more memory than ArrayList because every node in LinkedList stores reference of previous and next elements.

25. Which collection classes provide random access of it's elements?

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it's elements. Downloadjava collections pdf for more information.

26. What is EnumSet?

`java.util.EnumSet` is Set implementation to use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. EnumSet is not synchronized and null elements are not allowed. It also provides some useful methods like copyOf(Collection c), of(E first, E… rest) and complementOf(EnumSet s).

Check this post for java enum tutorial.

27. Which collection classes are thread-safe?

Vector, Hashtable, Properties and Stack are synchronized classes, so they are thread-safe and can be used in multi-threaded environment. Java 1.5 Concurrent API included some collection classes that allows modification of collection while iteration because they work on the clone of the collection, so they are safe to use in multi-threaded environment.

28. What are concurrent Collection Classes?

Java 1.5 Concurrent package (`java.util.concurrent`) contains thread-safe collection classes that allow collections to be modified while iterating. By design Iterator implementation in `java.util`packages are fail-fast and throws ConcurrentModificationException. But Iterator implementation in`java.util.concurrent` packages are fail-safe and we can modify the collection while iterating. Some of these classes are `CopyOnWriteArrayList`, `ConcurrentHashMap`, `CopyOnWriteArraySet` .

Read these posts to learn about them in more detail.

- Avoid ConcurrentModificationException
- CopyOnWriteArrayList Example
- HashMap vs ConcurrentHashMap

29. What is BlockingQueue?

`java.util.concurrent.BlockingQueue` is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

BlockingQueue interface is part of java collections framework and it's primarily used for implementing producer consumer problem. We don't need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue as it's handled by implementation classes of BlockingQueue.

Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.
Check this post for use of BlockingQueue for producer-consumer problem.

30. What is Queue and Stack, list their differences?

Both Queue and Stack are used to store data before processing them. `java.util.Queue` is an interface whose implementation classes are present in java concurrent package. Queue allows retrieval of element in First-In-First-Out (FIFO) order but it's not always the case. There is also Deque interface that allows elements to be retrieved from both end of the queue.
Stack is similar to queue except that it allows elements to be retrieved in Last-In-First-Out (LIFO) order.
Stack is a class that extends Vector whereas Queue is an interface.

31. What is Collections Class?

`java.util.Collections` is a utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that

operate on collections, “wrappers”, which return a new collection backed by a specified collection, and a few other odds and ends.

This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse etc.

32. What is Comparable and Comparator interface?

Java provides Comparable interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. Comparable interface has compareTo(T obj) method which is used by sorting methods. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as argument.

But, in most real life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. This is the situation where we need to use `Comparator` interface because`Comparable.compareTo(Object o)` method implementation can sort based on one field only and we can’t chose the field on which we want to sort the Object.

Comparator interface `compare(Object o1, Object o2)` method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if first argument is less than the second one and returns zero if they are equal and positive int if first argument is greater than second one.

Check this post for use of Comparable and Comparator interface to sort objects.

33. What is difference between Comparable and Comparator interface?

Comparable and Comparator interfaces are used to sort collection or array of objects.

Comparable interface is used to provide the natural sorting of objects and we can use it to provide sorting based on single logic.
Comparator interface is used to provide different algorithms for sorting and we can chose the comparator we want to use to sort the given collection of objects.

34. How can we sort a list of Objects?

If we need to sort an array of Objects, we can use `Arrays.sort()`. If we need to sort a list of objects, we can use `Collections.sort()`. Both these classes have overloaded sort() methods for natural sorting (using Comparable) or sorting based on criteria (using Comparator).
Collections internally uses Arrays sorting method, so both of them have same performance except that Collections take sometime to convert list to array.

35. While passing a Collection as argument to a function, how can we make sure the function will not be able to modify it?

We can create a read-only collection using `Collections.unmodifiableCollection(Collection c)` method before passing it as argument, this will make sure that any operation to change the collection will throw `UnsupportedOperationException`.

36. How can we create a synchronized collection from given collection?

We can use `Collections.synchronizedCollection(Collection c)` to get a synchronized (thread-safe) collection backed by the specified collection.

37. What are common algorithms implemented in Collections Framework?

Java Collections Framework provides algorithm implementations that are commonly used such as sorting and searching. Collections class contain these method implementations. Most of these algorithms work on List but some of them are applicable for all kinds of collections.
Some of them are sorting, searching, shuffling, min-max values.

38. What is Big-O notation? Give some examples?

The Big-O notation describes the performance of an algorithm in terms of number of elements in a data structure. Since Collection classes are actually data structures, we usually tend to use Big-O notation to chose the collection implementation to use based on time, memory and performance.

Example 1: ArrayList `get(index i)` is a constant-time operation and doesn't depend on the number of elements in the list. So it's performance in Big-O notation is O(1).
Example 2: A linear search on array or list performance is O(n) because we need to search through entire list of elements to find the element.

39. What are best practices related to Java Collections Framework?

- Chosing the right type of collection based on the need, for example if size is fixed, we might want to use Array over ArrayList. If we have to iterate over the Map in order of insertion, we need to use TreeMap. If we don't want duplicates, we should use Set.
- Some collection classes allows to specify the initial capacity, so if we have an estimate of number of elements we will store, we can use it to avoid rehashing or resizing.

- Write program in terms of interfaces not implementations, it allows us to change the implementation easily at later point of time.
- Always use Generics for type-safety and avoid ClassCastException at runtime.
- Use immutable classes provided by JDK as key in Map to avoid implementation of hashCode() and equals() for our custom class.
- Use Collections utility class as much as possible for algorithms or to get read-only, synchronized or empty collections rather than writing own implementation. It will enhance code-reuse with greater stability and low maintainability.

40. What is Java Priority Queue?

PriorityQueue is an unbounded queue based on a priority heap and the elements are ordered in their natural order or we can provide Comparator for ordering at the time of creation. PriorityQueue doesn't allow null values and we can't add any object that doesn't provide natural ordering or we don't have any comparator for them for ordering. Java PriorityQueue is not thread-safe and provided O(log(n)) time for enqueing and dequeing operations. Check this post for java priority queue example.

41. Why can't we write code as `List<Number> numbers = new ArrayList<Integer>();`?

Generics doesn't support sub-typing because it will cause issues in achieving type safety. That's why List<T> is not considered as a subtype of List<S> where S is the super-type of T. To understanding why it's not allowed, let's see what could have happened if it has been supported.

```
List<Long> listLong = new ArrayList<Long>();

listLong.add(Long.valueOf(10));

List<Number> listNumbers = listLong; // compiler error
```

```
listNumbers.add(Double.valueOf(1.23));
```

As you can see from above code that IF generics would have been supporting sub-typing, we could have easily add a Double to the list of Long that would have

42. What is identityHashMap?

Ans) The IdentityHashMap uses == for equality checking instead of equals(). This can be used for both performance reasons, if you know that two different elements will never be equals and for preventing spoofing, where an object tries to imitate another.

43. What is WeakHashMap?

Ans) A hashtable-based Map implementation with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently than other Map implementations.

Collection Tutorial

Set Interface

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.

The Java platform contains three general-purpose Set implementations: `HashSet`, `TreeSet`, and`LinkedHashSet`. Set interface doesn't allow random-access to an element in the Collection. You can use iterator or foreach loop to traverse the elements of a Set.

List Interface

List is an ordered collection and can contain duplicate elements. You can access any element from it's index. List is more like array with dynamic length. List is one of the most used Collection type. `ArrayList`and `LinkedList` are implementation classes of List interface.

List interface provides useful methods to add an element at specific index, remove/replace element based on index and to get a sub-list using index.

```
List strList = new ArrayList<>();

//add at last
strList.add(0, "0");

//add at specified index
strList.add(1, "1");

//replace
strList.set(1, "2");

//remove
strList.remove("1");
```

Collections class provide some useful algorithm for List – `sort`, `shuffle`, `reverse`, `binarySearch` etc.

Queue Interface

Queue is a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue.

Dequeue Interface

A linear collection that supports element insertion and removal at both ends. The name deque is short for “double ended queue” and is usually pronounced “deck”. Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element.

Map Interface

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.

The Java platform contains three general-purpose Map implementations: `HashMap`, `TreeMap`, and`LinkedHashMap`.

The basic operations of Map are `put`, `get`, `containsKey`, `containsValue`, `size`, and `isEmpty`.

ListIterator Interface

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator’s current position in the list.

A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

SortedSet Interface

SortedSet is a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

SortedMap Interface

Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

Java Collections Classes

Java Collections framework comes with many implementation classes for the interfaces. Most common implementations are ArrayList, HashMap and HashSet. Java 1.5 included Concurrent implementations; for example ConcurrentHashMap and CopyOnWriteArrayList. Usually Collection classes are not thread-safe and their iterator is fail-fast. In this section, we will learn about commonly used collection classes.

HashSet Class

This is the basic implementation the Set interface that is backed by a HashMap. It makes no guarantees for iteration order of the set and permits the **null** element.

This class offers constant time performance for basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. We can set the initial capacity and load factor for this collection. The load factor is a measure of how full the hash map is allowed to get before its capacity is automatically increased.

TreeSet Class

A `NavigableSet` implementation based on a `TreeMap`. The elements are ordered using their natural ordering, or by a `Comparator` provided at set creation time, depending on which constructor is used.

Refer: Java Comparable Comparator

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal.

ArrayList Class

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Further reading: Java ArrayList and Iterator

LinkedList Class

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

HashMap Class

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits null. This class makes no guarantees for the order of the map.

This implementation provides constant-time performance for the basic operations (`get` and `put`). It provides constructors to set initial capacity and load factor for the collection.

Further Read: HashMap vs ConcurrentHashMap

TreeMap Class

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a

precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

PriorityQueue Class

Queue processes it's elements in FIFO order but sometimes we want elements to be processed based on their priority. We can use PriorityQueue in this case and we need to provide a Comparator implementation while instantiation the PriorityQueue. PriorityQueue doesn't allow null values and it's unbounded. For more details about this, please head over to **Java Priority Queue** where you can check it's usage with a sample program.

Collections class

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse etc.

Synchronized Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — Collection, Set, List, Map, SortedSet, and SortedMap — has one static factory method.

```
public static  Collection synchronizedCollection(Collection c);

public static  Set synchronizedSet(Set s);
```

```
public static  List synchronizedList(List list);

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

public static  SortedSet synchronizedSortedSet(SortedSet s);

public static <K,V> SortedMap<K,V>
synchronizedSortedMap(SortedMap<K,V> m);
```

Each of these methods returns a synchronized (thread-safe) Collection backed up by the specified collection.

Unmodifiable wrappers

Unmodifiable wrappers take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an `UnsupportedOperationException`. It's main usage are;

- To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.
- To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, clients can look but not modify, while you maintain full access.

These methods are;

```
public static  Collection unmodifiableCollection(Collection<?
extends T> c);

public static  Set unmodifiableSet(Set<? extends T> s);

public static  List unmodifiableList(List<? extends T> list);
```

```
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K,
? extends V> m);

public static  SortedSet unmodifiableSortedSet(SortedSet<?
extends T> s);

public static <K,V> SortedMap<K, V>
unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

Thread Safe Collections

Java 1.5 Concurrent package (`java.util.concurrent`) contains thread-safe collection classes that allow collections to be modified while iterating. By design iterator is fail-fast and throws ConcurrentModificationException. Some of these classes are `CopyOnWriteArrayList`, `ConcurrentHashMap`,`CopyOnWriteArraySet`.

Read these posts to learn about them in more detail.

- Avoid ConcurrentModificationException
- CopyOnWriteArrayList Example
- HashMap vs ConcurrentHashMap

Collections API Algorithms

Java Collections Framework provides algorithm implementations that are commonly used such as sorting and searching. Collections class contain these method implementations. Most of these algorithms work on List but some of them are applicable for all kinds of collections.

Sorting

The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship. Two forms of the operation are provided. The simple form takes a List and sorts it according to its elements' natural ordering. The second form of

sort takes a Comparator in addition to a List and sorts the elements with the Comparator.

Shuffling

The shuffle algorithm destroys any trace of order that may have been present in a List. That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. This algorithm is useful in implementing games of chance.

Searching

The binarySearch algorithm searches for a specified element in a sorted List. This algorithm has two forms. The first takes a List and an element to search for (the “search key”). This form assumes that the List is sorted in ascending order according to the natural ordering of its elements. The second form takes a Comparator in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator. The sort algorithm can be used to sort the List prior to calling binarySearch.

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more Collections.

- **frequency**: counts the number of times the specified element occurs in the specified collection
- **disjoint**: determines whether two Collections are disjoint; that is, whether they contain no elements in common

Min and Max values

The min and the max algorithms return, respectively, the minimum and maximum element contained in a specified Collection. Both of these operations come in two

forms. The simple form takes only a Collection and returns the minimum (or maximum) element according to the elements' natural ordering.
The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.