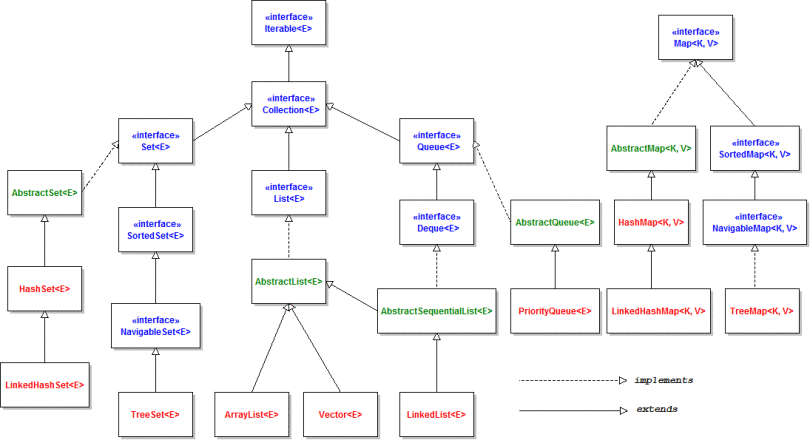
**1. Draw Collections Framework Class Diagram**

****

**2. What is HashMap and Map?**

**MAP:-**

- Map is the interface i.e., an abstract thing that defines how something can be used.

- Map's contents can be viewed as a set of keys, collection of values, or set of key-value mappings.

- Map is an object that maps keys to values.

- A map cannot contain duplicate keys

- Map accepts null values also both as key and value

**HashMap:-**

- HashMap is a class and is the implementation of Map interface. HashMap contains all the properties of a Map.

- HashMap doesnt guarentee any order when elements are retrieved.

**3. Difference between HashMap and HashTable? Can we make hashmap synchronized?**

**HashMap**

1) HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code.  
2) HashMap allows one null key and multiple null values.  
3) HashMap is fast.  
4) HashMap is traversed by Iterator.  
5) Iterator in HashMap is fail-fast.  
7) HashMap inherits AbstractMap class.

**Hashtable**

1) Hashtable is synchronized. It is thread-safe and can be shared with many threads.  
2) Hashtable doesn't allow any null key or value.  
3) Hashtable is slow.  
4) Hashtable is internally synchronized and can't be unsynchronized.  
5) Hashtable is traversed by Enumerator and Iterator.  
6) Enumerator in Hashtable is not fail-fast.  
7) Hashtable inherits Dictionary class.

**Explanation**

1. Hashtable is [synchronized](https://stackoverflow.com/questions/1085709/what-does-synchronized-mean), whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. Iterator in the HashMap is fail-safe while the enumerator for the Hashtable is not and throw ConcurrentModificationException if any other Thread modifies the map structurally by adding or removing any element except Iterator's own remove() method. But this is not a guaranteed behavior and will be done by JVM on best effort.

**Yes**, In order to synchronize it we should use [Collections.synchronizedMap(hashmap)](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedMap(java.util.Map)). It returns a thread-safe map backed up by the specified HashMap.

**Program for Synchronized HashMap:-**

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import java.util.Iterator;

public class HashMapSyncExample {

public static void main(String args[]) {

HashMap<Integer, String> hmap= new HashMap<Integer, String>();

hmap.put(2, "Anil");

hmap.put(44, "Ajit");

hmap.put(1, "Brad");

hmap.put(4, "Sachin");

hmap.put(88, "XYZ");

Map map= Collections.synchronizedMap(hmap);

Set set = map.entrySet();

synchronized(map){

Iterator i = set.iterator();

// Display elements

while(i.hasNext()) {

Map.Entry me = (Map.Entry)i.next();

System.out.print(me.getKey() + ": ");

System.out.println(me.getValue());

}

}

}

}

**4. Difference between Vector and ArrayList?**

- Vector is synchronized. ArrayList is not.

Collections.synchronizedList is normally used at the time of creation of the Arraylist to avoid any accidental unsynchronized access to the list.

- Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. *A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent***.**

- ArrrayList uses Iterator interface to traverse the elements.Vector uses eumeration interface to traverse the elements, But it can also use iterator.

**5. What is an Iterator?**

Iterators are used in [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) in Java to retrieve elements one by one. There are three iterators.

a) Enumeration

b) Iterator

c) ListIterator

**Enumeration:-** It is a interface used to get elements of legacy collections(Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality. Enumerations are also used to specify the input streams to a *SequenceInputStream*. We can create Enumeration object by calling *elements()* method of vector class on any vector object

There are two methods in Enumeration interface namely : hasMoreElements(),nextElement()

**Limitations of Enumeration :**

* Enumeration is for legacy classes(Vector, Hashtable) only. Hence it is not a universal iterator.
* Remove operations can’t be performed using Enumeration.
* Only forward direction iterating is possible.

**Iterator:-** It is a universal iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

Iterator object can be created by calling *iterator()* method present in Collection interface.

Iterator interface defines **three** methods: hasNext(),next(),remove()

Limitations of Iterator :

* Only forward direction iterating is possible.
* Replacement and addition of new element is not supported by Iterator.

**ListIterator:-** It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration.ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling listIterator() method present in List interface.

ListIterator interface extends Iterator interface. So all three methods of Iterator interface are available for ListIterator. In addition there are six more methods.

hasNext(),next(),nextIndex(),nextPrevious(),previous(),previousIndex(),remove(),set(),add()

**Limitations of ListIterator :** It is the most powerful iterator but it is only applicable for List implemented classes, so it is not a universal iterator.

**6.List vs Set vs Map. Purposes and definitions.**

**List:-** Java.util.List is a child interface of [Collection](https://www.geeksforgeeks.org/collections-in-java-2/). List is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

**Set:-** Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored. Basically, Set is implemented by HashSet, LinkedSet or TreeSet (sorted representation).

Set has various methods to add, remove clear, size, etc to enhance the usage of this interface.

**Map:-** The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the [Collection](https://www.geeksforgeeks.org/collections-in-java-2/)interface. Therefore it behaves a bit different from the rest of the collection types. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value ([HashMap](https://www.geeksforgeeks.org/hashmap-treemap-java)and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples)) but some do not ([TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java)). The order of a map depends on specific implementations,e.g [TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java)and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples)have predictable order, while [HashMap](https://www.geeksforgeeks.org/hashmap-treemap-java)does not.

**7.Pros and cons of ArrayList and LinkedList**

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

**pros of ArrayList:-**

ArrayList is better for storing and accessing data.

**cons of ArrayList:-**

- Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.

- ArrayList class can act as a list only because it implements List only.

**Pros of LinkedList:-**

- Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.

- LinkedList class can act as a list and queue both because it implements List and Deque interfaces.

- LinkedList is better for manipulating data.

- Flexibility - insert at (or delete from) any position in contant time

- No single allocation of memory needed - fragmented memory can be put to a better use

**cons of LinkedList:-**

- Complex to use and access - relatively complex as compared to arrays

- No constant time access to the elements - simply because it doesn't involve the simple arithmetic used by arrays to compute the memory address, so relatively inefficient as compared to arrays

**8.TreeSet vs LinkedHashSet**

**Performance and Speed :** First difference between them comes in terms of speed. HashSet is fastest, LinkedHashSet is bit slow on performance

**Ordering :** HashSet does not maintain any order while LinkedHashSet maintains insertion order of elements much like List interface

**Internal Implementation :** HashSet is backed by an HashMap instance, LinkedHashSet is implemented using HashSet and LinkedList

**9.What are relationships between equals and hash codes?**

The problem you will have is with collections where unicity of elements is calculated according to both .equals() and .hashCode(), for instance keys in a HashMap.

As its name implies, it relies on hash tables, and hash buckets are a function of the object's .hashCode().

If you have two objects which are .equals(), but have different hash codes, you lose!

The part of the contract here which is important is: objects which are .equals() MUST have the same .hashCode().

The general contract of [hashCode()](http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#hashCode%28%29) is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

*- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.*

**10.What are the advantages of ArrayList over arrays ?**

- Arrays are of fixed length. You can not change the size of the arrays once they are created.

- You can not accommodate an extra element in an array after they are created.

- Memory is allocated to an array during it’s creation only, much before the actual elements are added to it.

Because of these drawbacks, use of arrays are less preferred. Instead of arrays, you can use ArrayList class which addresses all these drawbacks. Here are some advantages of using ArrayList over arrays.

1) You can define ArrayList as re-sizable array. Size of the ArrayList is not fixed. ArrayList can grow and shrink dynamically.

2) Elements can be inserted at or deleted from a particular position.

3) ArrayList class has many methods to manipulate the stored objects.

ArrayList class has methods to perform solo modifications ( add(), remove()… ), bulk modifications ( addAll(), removeAll(), retainAll()… ), searching( indexOf(), lasIndexOf() ) and iterations( iterator() ).

4) If generics are not used, ArrayList can hold any type of objects.

5) Many are of the assumption that multiple insertion and removal operations on ArrayList will decrease the performance of an application. But, there will be no significant change in the performance of an application if you use ArrayList instead of arrays. Below example shows time taken to add 1000 string elements to ArrayList and array.

6) You can traverse an ArrayList in both the directions – forward and backward using ListIterator.

7) ArrayList can hold multiple null elements.

8) ArrayList can hold duplicate elements.

**11.Principle of storing data in a hashtable**

Consider the real-life example of a telephone directory. Large directories (such as the White Pages) contain many hundreds of thousands of records; if the data were to be recorded without rhyme or reason, a search would take an incredibly long time.

Fortunately, telephone directories are sorted, but that isn't where most of the speed improvement comes from - it's the categorisation of each collection of numbers by the first letter of the name that allows rapid access to "relevant" data (i.e. the data collection that is closest to the target data, without having to look through irrelevant details.)

A *hash table* works on a similar principle. As with any other searching operation, there is a search key alongside each item of data (for instance, the surname of a telephone directory record.)

However, instead of searching through the data directly, trying to match the search key with any of the stored data, it uses what is known as a *hash function*. This takes the key and generates a *hash code*, which identifies which part of the structure the data resides. In the telephone directory example, the hash function is equivalent to the process of taking the first letter of a surname, and the hash code is that first letter.

A Java implementation of a hash table is created from similar items:

Each data item to store is encapsulated into an object that carries the search key along with the data (the key may even be part of the data)

The hash table is implemented as an array, inside a class that also encapsulates the functions of the table (add, retrieve, delete, etc.) The hash function is also stored here.

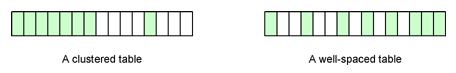
There are a number of different hash table variants:

**closed-addressing:** this is where data is stored inside the table

**open-addressing:** this is where there is a data structure at each element of the table, where all data with a matching hash code is stored

The Hash Function

The hash function must be well suited to the type of data that it is categorising. A function's quality can be assessed by how well it spreads data across all hash elements - having ideally an (approximately) equal amount of data in each.



A poor hash table will 'cluster' a majority of the data in only a few hash elements. This is undesirable, as it unnecessarily slows down the performance of the structure (the smaller the categories, the smaller the search time within them.)

In summary, a hash function's performance can be analysed by asking two questions:

does it use the entire range of the hash table?

how well does it spread the data across the table?

A good hash function must be capable of both in order to prevent collisions.

**Another Example:-**

Let's assume you want to fill up a library of books and not just stuff them in there, but you want to be able to easily find them again when you need them.

So, you decide that if the person that wants to read a book knows the title of the book and the exact title to boot, then that's all it should take. With the title, the person, with the aid of the librarian, should be able to find the book easily and quickly.

So, how can you do that? Well, obviously you can keep some kind of list of where you put each book, but then you have the same problem as searching the library, you need to search the list. Granted, the list would be smaller and easier to search, but still you don't want to search sequentially from one end of the library (or list) to the other.

You want something that, with the title of the book, can give you the right spot at once, so all you have to do is just stroll over to the right shelf, and pick up the book.

But how can that be done? Well, with a bit of forethought when you fill up the library and a lot of work when you fill up the library.

Instead of just starting to fill up the library from one end to the other, you devise a clever little method. You take the title of the book, run it through a small computer program, which spits out a shelf number and a slot number on that shelf. This is where you place the book.

The beauty of this program is that later on, when a person comes back in to read the book, you feed the title through the program once more, and get back the same shelf number and slot number that you were originally given, and this is where the book is located.

The program, as others have already mentioned, is called a hash algorithm or hash computation and usually works by taking the data fed into it (the title of the book in this case) and calculates a number from it.

For simplicity, let's say that it just converts each letter and symbol into a number and sums them all up. In reality, it's a lot more complicated than that, but let's leave it at that for now.

The beauty of such an algorithm is that if you feed the same input into it again and again, it will keep spitting out the same number each time.

Ok, so that's basically how a hash table works.

**12.Differences between Hashtable, ConcurrentHashMap and Collections.synchronizedMap()**

╔═══════════╦═══════════════╦════════════╦═════════════╗

║ Property ║ HashMap ║ Hashtable ║ Concurrent ║

║ ║ ║ ║ HashMap ║

╠═══════════╬═══════════════╬════════════╩═════════════╣

║ Null ║ allowed ║ not allowed ║

║values/keys║ ║ ║

╠═══════════╬═══════════════╬══════════════════════════╣

║thread-safe║ no ║ yes ║

╠═══════════╬═══════════════╬═══════════╦══════════════╣

║ Lock ║ not ║locks the ║ locks the ║

║ mechanism ║ applicable ║whole map ║ portion ║

╠═══════════╬═══════════════╩═══════════╬══════════════╣

║ Iterator ║ ║ weakly ║

║ ║ fail-fast ║ consistent ║

╚═══════════╩═══════════════════════════╩══════════════╝

**13.How are hash codes computed?**

The hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

Default implementation of hashCode() generally returns address value of object in memory whereas when hashCode() overridden for some classes then it totally depends on how it has been implemented in that class.

A hashcode is an integer value that represents the state of the object upon which it was called. That is why an Integer that is set to 1 will return a hashcode of "1" because an Integer's hashcode and its value are the same thing. A character's hashcode is equal to it's ASCII character code. If you write a custom type you are responsible for creating a good hashCode implementation that will best represent the state of the current instance.

**14.Is it possible that hash code is not unique?**

The term “unique hash code” is a red flag indicating that the programmer who uttered it does not understand hash codes and is about to do something incredibly stupid. Let me provide a simple example.

Every object has a HashCode method that computes a 32-bit number that can serve as a key in a dictionary or hash table. Used properly, the hash code lets you create very efficient data structures for looking things up by name. But there’s no magic involved, really.

Used as intended–as the keys in a dictionary or hash table–hash codes work very well. If you try to use them for something else, it’s not going to work.

Let’s say you foolishly decide to use a hash code for a “unique key” when indexing some strings.

The number of strings is essentially infinite. The number of unique values that a 32-bit hash code can represent is a little more than four billion. So it’s a certainty that two or more strings will produce the same hash code. You might think, if you’re only storing 100,000 values, that the chance of collision (two strings hashing to the same value) is so small as not to matter. You’d be wrong.

If you’re familiar with the Birthday problem you know that, in any group of 25 people selected at random, the chance of two having the same birthday (month and day) are better than 50%. In a group of 60 people, it’s almost a certainty. If you don’t believe the math, you can do some empirical research of your own.

Hash codes are a lot like birthdays in this regard. A good rule of thumb is that the chance of collision (two items hashing to the same value) is 50% when the number of items hashed is equal to the square root of the number of possible values. So with a 32-bit hash code, the chance of two items hashing to the same value will be 50% when you’ve hashed 2^16, or 65,536 items. Again, if you don’t believe me just write a program that generates random strings and try it.

Another rule of thumb is that you’re almost certain to get a collision when the number of items is four times the square root. So with a 32-bit hash code, the chance of getting a collision when adding 256,000 items is almost 100%.

This is second year Computer Science stuff. Maybe even first year? And yet I hear the term “unique hash code” thrown around distressingly often by experienced programmers who should know better. It’s frightening.

If you’re interested in a bit more discussion and some sample programs that illustrate this point better, see Birthdays, random Numbers, and hash keys.

http://blog.mischel.com/2017/11/02/birthdays-random-numbers-and-hash-keys/

There are 4,294,967,296 possible hashCode values. If we create 4,294,967,297 Objects, then, again, the pigeonhole principle tells us that no, it won't be unique.

**15.Can we put two elements with equal hash code to one hash map?**

A hashmap works like this (this is a little bit simplified, but it illustrates the basic mechanism):

It has a number of "buckets" which it uses to store key-value pairs in. Each bucket has a unique number - that's what identifies the bucket. When you put a key-value pair into the map, the hashmap will look at the hash code of the key, and store the pair in the bucket of which the identifier is the hash code of the key. For example: The hash code of the key is 235 -> the pair is stored in bucket number 235. (Note that one bucket can store more then one key-value pair).

When you lookup a value in the hashmap, by giving it a key, it will first look at the hash code of the key that you gave. The hashmap will then look into the corresponding bucket, and then it will compare the key that you gave with the keys of all pairs in the bucket, by comparing them with equals().

Now you can see how this is very efficient for looking up key-value pairs in a map: by the hash code of the key the hashmap immediately knows in which bucket to look, so that it only has to test against what's in that bucket.

Looking at the above mechanism, you can also see what requirements are necessary on the hashCode() and equals() methods of keys:

If two keys are the same (equals() returns true when you compare them), their hashCode() method must return the same number. If keys violate this, then keys that are equal might be stored in different buckets, and the hashmap would not be able to find key-value pairs (because it's going to look in the same bucket).

If two keys are different, then it doesn't matter if their hash codes are the same or not. They will be stored in the same bucket if their hash codes are the same, and in this case, the hashmap will use equals() to tell them apart.

**16.Iterator and modification of a List. ConcurentModificationException.**

java.util.ConcurrentModificationException is a very common exception when working with java collection classes. Java Collection classes are fail-fast, which means if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw ConcurrentModificationException. Concurrent modification exception can come in case of multithreaded as well as single threaded java programming environment.

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

public class ConcurrentModificationExceptionExample {

public static void main(String args[]) {

List<String> myList = new ArrayList<String>();

myList.add("1");

myList.add("2");

myList.add("3");

myList.add("4");

myList.add("5");

Iterator<String> it = myList.iterator();

while (it.hasNext()) {

String value = it.next();

System.out.println("List Value:" + value);

if (value.equals("3"))

myList.remove(value);

}

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

myMap.put("1", "1");

myMap.put("2", "2");

myMap.put("3", "3");

Iterator<String> it1 = myMap.keySet().iterator();

while (it1.hasNext()) {

String key = it1.next();

System.out.println("Map Value:" + myMap.get(key));

if (key.equals("2")) {

myMap.put("1", "4");

// myMap.put("4", "4");

}

}

}

}

**17.What is the significance of ListIterator? What is the difference b/w Iterator and ListIterator?**

Like Iterator, ListIterator is a Java Iterator, which is used to iterate elements one-by-one from a List implemented object.

It is available since Java 1.2.

It extends Iterator interface.

It is useful only for List implemented classes.

Unlike Iterator, It supports all four operations: CRUD (CREATE, READ, UPDATE and DELETE).

Unlike Iterator, It supports both Forward Direction and Backward Direction iterations.

It is a Bi-directional Iterator.

It 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().

NOTE:- What is CRUD operations in Collection API?

CREATE: Adding new elements to Collection object.

READ: Retrieving elements from Collection object.

UPDATE: Updating or setting existing elements in Collection object.

DELETE: Removing elements from Collection object.

1) Iterator is used for traversing List and Set both.

We can use ListIterator to traverse List only, we cannot traverse Set using ListIterator.

2) We can traverse in only forward direction using Iterator.

Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3) We cannot obtain indexes while using Iterator

We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

4) We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.

We can add element at any point of time while traversing a list using ListIterator.

5) We cannot replace the existing element value when using Iterator.

By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.

6) Methods of Iterator:

hasNext()

next()

remove()

Methods of ListIterator:

add(E e)

hasNext()

hasPrevious()

next()

nextIndex()

previous()

previousIndex()

remove()

set(E e)

**18.What is the Collections API?**

As the name indicates, collections is a group of objects known as its elements. Basically it is a package of data structures that includes ArrayLists, LinkedLists, HashSets, etc. A collection is simply an object that groups multiple elements into a single unit. It is also called as a container sometimes. It is used to store, retrieve, manipulate, and communicate aggregate data. Typically, it represents data items that form a natural group and allows duplicate elements while others do not. It consists of both ordered and unordered elements. There is no direct implementation of this interface however SDK provides implementations of more specific sub interfaces like Set and List. The manipulation and passing of collections is done by this interface.

The Two "standard" constructors should be provided by all the general-purpose Collection implementation classes. These classes typically implement Collection indirectly through one of its sub interfaces.

Void (no arguments) constructor which creates an empty collection.

Constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument.

The user can copy any collection using void constructor to produce an equivalent collection of the desired implementation type. As interfaces cannot contain constructors there is no way to enforce this convention. However all of the general-purpose Collection implementations comply this in the Java platform libraries.

The Java Collections API:

Java Collections of API (Application Programming Intreface) Consists of several interfaces, and classes that implement those interfaces, within the java.util package. It provides tools for maintaining a data container of objects. Each primitive value must be wrapped in an object of its appropriate wrapper class (Boolean, Character, Integer, Double, etc.) to maintain a collection of primitive data. It is an alternative tool to the creation of custom data structures.

You must be familiar with collections if you have worked with Java programming language . Collection implementations included Vector, Hashtable, and array are available in earlier (pre-1.2) versions of the Java platform, but those versions do not include the collections framework. Hence the new version of the Java platform contains the collections framework.

**19.How can we access elements of a collection?**

To access, modify or remove any element from any collection we need to first find the element, for which we have to cycle through the elements of the collection. There are three possible ways to cycle through the elements of any collection.

Using Iterator interface

Using ListIterator interface

Using for-each loop

**20.What is the difference between a queue and tack?**

Stack is a collection of objects that works in LIFO (Last in First out) mechanism while Queue is FIFO (First in First out). This means that the object that is inserted first is removed last in a stack while an object that is inserted first is removed first in a queue.

**21.What is the Properties class?**

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

Properties define the following instance variable. This variable holds a default property list associated with a Properties object.

Following is the list of constructors provided by the properties class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **Properties( )**  This constructor creates a Properties object that has no default values. |
| 2 | **Properties(Properties propDefault)**  Creates an object that uses propDefault for its default values. In both cases, the property list is empty. |

Apart from the methods defined by Hashtable, Properties define the following methods −

**String getProperty(String key)**

Returns the value associated with the key. A null object is returned if the key is neither in the list nor in the default property list.

**String getProperty(String key, String defaultProperty)**

Returns the value associated with the key; defaultProperty is returned if the key is neither in the list nor in the default property list.

**void list(PrintStream streamOut)**

Sends the property list to the output stream linked to streamOut.

**void list(PrintWriter streamOut)**

Sends the property list to the output stream linked to streamOut.  
**void load(InputStream streamIn)** throws IOException

Inputs a property list from the input stream linked to streamIn.

**Enumeration propertyNames( )**

Returns an enumeration of the keys. This includes those keys found in the default property list, too.

**Object setProperty(String key, String value)**

Associates value with the key. Returns the previous value associated with the key, or returns null if no such association exists.

**void store(OutputStream streamOut, String description)**

After writing the string specified by description, the property list is written to the output stream linked to streamOut.

**22.Which implementation of the List interface provides for the fastest insertion of a new element into the middle of the List?**

ArrayList and Vector both use an array to store the elements of the list. When an element is inserted into the middle of the list the elements that follow the insertion point must be shifted to make room for the new element. The LinkedList is implemented using a doubly linked list; an insertion requires only the updating of the links at the point of insertion. Therefore, the LinkedList allows for fast insertions and deletions.

**23.How can we use hashset in collection interface?**

Set interface is collection which does not allow duplicate values. A Set implementation like HashSet, checks hashCode() and equals(Object) methods overridden in an object before adding it. Order of it’s containing values is not guaranteed.

import java.util.HashSet;

import java.util.Set;

public class User {

private String userName;

public User(final String name) {

this.userName = name;

}

public String getUserName() {

return this.userName;

}

@Override

public boolean equals(final Object obj) {

boolean isEqual = false;

if(obj == null) {

isEqual = false;

} else if(!(this.getClass().equals(obj.getClass()))) {

isEqual = false;

} else if(this.userName.equalsIgnoreCase(((User)obj).getUserName())) {

isEqual = true;

}

return isEqual;

}

@Override

public int hashCode() {

return this.userName.toUpperCase().hashCode();

}

@Override

public String toString() {

return this.userName;

}

// for test

public static void main(String[] args) {

Set<User> chatRoomUsers = new HashSet<User>();

chatRoomUsers.add(new User("user1"));

chatRoomUsers.add(new User("user1"));

chatRoomUsers.add(new User("user5"));

chatRoomUsers.add(new User("uSEr1"));

chatRoomUsers.add(new User("User1"));

chatRoomUsers.add(new User("user2"));

chatRoomUsers.add(new User("user4"));

chatRoomUsers.add(new User("user3"));

// print chat room users

for (User usr : chatRoomUsers) {

System.out.println(usr);

}

}

}

**25. Can you limit the initial capacity of vector in java?**

we can construct an empty vector constructor with specified initial capacity  
  
public vector(int initialcapacity)

**26. What method should the key class of Hashmap override?**

You need to implement hashCode() and equals().

compareTo() is additionally required for sorted map/set.

**27. What is the difference between Enumeration and Iterator?**

*- Enumeration* only traverses the *Collection*object. You can’t do any modifications to *Collection* while traversing the *Collection* using *Enumeration*. Where as *Iterator* interface allows us to remove an element while traversing the *Collection*object.  *Iterator* has *remove()* method which is not there in the *Enumeration* interface. Below is the list of *Enumeration* and *Iterator*methods.

- *Iterator* is a fail-fast in nature. i.e it throws *ConcurrentModificationException* if a collection is modified while iterating other than it’s own *remove()* method. Where as *Enumeration* is fail-safe in nature. It doesn’t throw any exceptions if a collection is modified while iterating

- As *Iterator* is fail-fast in nature and doesn’t allow modification of a collection by other threads while iterating, it is considered as safe and secure than *Enumeration*.

**28. Collections class and Arrays class**

**Arrays class in Java:-**

There are often times when we need to do following tasks on an array in Java.

Fill an array with a particular value. We usually do it with the help of a for loop.

- Sort an array.

- Binary search in sorted array.

And many more..

The Arrays class of the java.util package contains several static methods that we can use to fill, sort, search, etc in arrays. This class is a member of the Java Collections Framework and is present in java.util.arrays.

**public static String toString(int[] a)** The string representation consists of a list of the array’s elements, enclosed in square brackets (“[]”). Adjacent elements are separated by the characters a comma followed by a space. Elements are converted to strings as by String.valueOf(int). Returns “null” if a is null.

**public static void sort(int[] a)** – Sorts the specified array into ascending numerical order.

**public static void sort(int[] a, int fromIndex, int toIndex)** If we wish to sort a specified range of the array into ascending order. we can use this. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.

**public static int binarySearch(int[] a, int key)** Returns an int value for the index of the specified key in the specified array. Returns a negative number if the specified key is not found in the array. For this method to work properly, the array must first be sorted by the sort method.

**public static int[] copyOf(int[] original, int newLength)** Copies the specified array and length. It truncates the array if provided length is smaller and pads if provided .

**public static int[] copyOfRange(int[] original, int from, int to)** Copies the specified range of the specified array into a new array. The initial index of the range (from) must lie between zero and original.length, inclusive.

**public static void fill(int[] a, int val)** Fills all elements of the specified array with the specified value.

**public static void fill(int[] a, int fromIndex, int toIndex, int val)** – Fills elements of the specified array with the specified value from the fromIndex element, but not including the toIndex element.

**public static List asList(T… a)** It takes an array and creates a wrapper that implements List, which makes the original array available as a list. Nothing is copied and all, only a single wrapper object is created. Operations on the list wrapper are propagated to the original array. This means that if you shuffle the list wrapper, the original array is shuffled as well, if you overwrite an element, it gets overwritten in the original array, etc. Of course, some List operations aren’t allowed on the wrapper, like adding or removing elements from the list, you can only read or overwrite the elements.

**Collection Class in Java**

Collections class in java is a useful utility class to work with collections in java. The java.util.Collections class directly extends the Object class and exclusively consists of the static methods that operate on Collections or return them.

Collections class contains 3 fields: EMPTY\_LIST, EMPTY\_SET, EMPTY\_MAP, which can be used to get immutable empty List, Map and Set respectively.

**boolean addAll(Collection c, T... elements):** This method adds all of the provided elements to the specified collection at once. The elements can be provided as a comma-separated list.

**void sort(List list, Comparator c):** This method sorts the provided list according to the natural ordering. We can also pass in s Comparator, if we want some custom ordering.

Queue asLifoQueue(Deque deque): This method returns a view of Deque as a Last-in-first-out (Lifo) Queue.

The methods add and remove are mapped to push, pop respectively and so on. This can be useful when we would like to use a method requiring a Queue but we need Lifo ordering.

**int binarySearch(List<? extends Comparable> list, T key):** This method searches the key using binary search in the specified list. The list should be sorted by natural ordering, before calling this method, otherwise, the result will be undefined.

It returns the index of the element in the sorted list if the element is found, in other cases, it returns (-(insertion point)-1). Where, the insertion point is defined as the point at which the key would be inserted into the list, i.e. the index of the first element greater than the key, or list.size() if all elements in the list are less than the specified key.

Note that this guarantees that the return value will be >= 0 if the key is found.

We can also pass in a Comparator, which indicates that the list is sorted in the order induced by the specified comparator

**Collection checkedCollection(Collection c, Class type):** This method provides a dynamically typesafe view of the provided collection. It is useful to keep an eye on the collection, that any wrongly typed element is not inserted in it.

Similarly, we have check methods for specific collections such as List, Map, Set, etc.

**void copy(List dest, List src):** This method copies all of the elements from source list to destination list. After this operation is performed, the index of each copied element in the destination list will be identical to its index in the source list.

**boolean disjoint(Collection c1, Collection c2):** This method returns true if the two specified collections have no elements in common.

**void fill(List list, T obj):** This method replaces all of the elements of the specified list with the specified element.

**int frequency(Collection c, Object o**): This method returns the number of elements in the specified collection which are equal to the specified object.

From our previous example, if we say:

**int indexOfSubList(List source, List target):** This method returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.

static ArrayList list(Enumeration e) and Enumeration **enumeration(Collection c):** These methods returns an array list from enumeration and an enumeration from a collection, respectively, so as to provide interoperability between legacy APIs that return enumerations and new APIs that require collections.

**T max(Collection coll, Comparator comp):** This method returns the maximum element in collection according to the natural ordering of elements.

**Collection<T> synchronizedCollection(Collection<T> c):** This method returns a synchronized (thread-safe) collection backed by the provided collection. It’s convinient to get synchronized collections from any collection object as and when required. API also provides us with synchronizedList method which returns thread-safe list backed by provided list in the argument.