# Core Java Interview Questions – Part 2

In [java interview questions series: part 1](https://howtodoinjava.com/java/interviews-questions/core-java-interview-questions-series-part-1/), we discussed some important questions which interviewer ask most of the time. Now is the time to take that discussion forward. In this post, I will talk about below given list of questions.

[Why finalize() method should be avoided?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#why-avoid-finalize)

[Why HashMap should not be used in multithreaded environment? Can it cause infinite loop as well?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#hashmap-in-multithreading)

[Explain abstraction and encapsulation? How are they related?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#abstraction-encalsulation)

[Difference between interfaces and abstract classes?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#interface-vs-abstractclass)

[How StringBuffer save the memory?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#stringbuffer-save-memory)

[Why wait and notify is declared in Object class instead of Thread ?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#wait-notify)

[Write Java program to create deadlock in Java and fix it ?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#deadlock-in-java)

[What happens if your Serializable class contains a member which is not serializable? How do you fix it?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#serializable)

[Explain transient and volatile keywords in java?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#transient-volatile)

[Difference between Iterator and ListIterator?](https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-2/#iterator-vs-listiterator)

# Why finalize() method should be avoided?

We all know the basic statement that [finalize()](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#finalize--) method is called by garbage collector thread before reclaiming the memory allocated to the object. See [this program](https://howtodoinjava.com/java/related-concepts/why-not-to-use-finalize-method-in-java/) which prove that finalize() invocation is not guaranteed at all. Other reasons can be:

1. finalize() methods do not work in chaining like constructors. It means like when you call a constructor then constructors of all super classes will be invokes implicitly. But, in case of finalize methods, this is not followed. Super class’s finalize() should be called explicitly.
2. Any Exception thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact it will not be logged in your log files. So bad, isn’t it?
3. Also, There is some performance penalty when finalize() in included in your class. In Effective java (2nd edition ) Joshua bloch says,

“Oh, and one more thing: there is a severe performance penalty for using finalizers. On my machine, the time to create and destroy a simple object is about 5.6 ns. Adding a finalizer increases the time to 2,400 ns. In other words, it is about 430 times slower to create and destroy objects with finalizers.”

# Why HashMap should not be used in multithreaded environment? Can it cause infinite loop as well?

We know that HashMap is non-synchronized collection where as its synchronized counter-part is HashTable. So, when you are accessing the collection in multithreaded environment and all threads are accessing a single instance of collection, then its safer to use HashTable for various obvious reasons e.g. to avoid dirty reads and to maintain data consistency. In worst case, this mutithreaded environment can result in infinite loop as well.

Yes, it is true. HashMap.get() can cause an infinite loop. Lets see how??

If you look at the source code [HashMap.get(Object key)](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap.get%28java.lang.Object%29) method, it looks like this:

*public Object get(Object key) {*

*Object k = maskNull(key);*

*int hash = hash(k);*

*int i = indexFor(hash, table.length);*

*Entry e = table[i];*

*while (true) {*

*if (e == null)*

*return e;*

*if (e.hash == hash && eq(k, e.key))*

*return e.value;*

*e = e.next;*

*}*

*}*

while(true){...} can always be a victim of infinite loop at runtime in multithreaded environment, IF, somehow e.next can point to itself. This will result in infinite loop. But, how e.next will point to itself (i.e.).

This can happen in void transfer(Entry[] newTable) method, which is invoked at time the HashMap resizing is done.

*do {*

*Entry next = e.next;*

*int i = indexFor(e.hash, newCapacity);*

*e.next = newTable[i];*

*newTable[i] = e;*

*e = next;*

*} while (e != null);*

This piece of code is prone to produce above condition, if resizing happen and at the same time other threads tried to modify the map instance.

Only way to avoid this scenario is to use synchronization in code, or better, use synchronized collection.

# Explain abstraction and encapsulation? How are they related?

#### Abstraction

[Abstraction](https://howtodoinjava.com/object-oriented/understanding-abstraction-in-java/) captures only those details about an object that are relevant to the current perspective.

In [object-oriented programming](https://howtodoinjava.com/object-oriented/object-oriented-principles/) theory, abstraction involves the facility to define objects that represent abstract “actors” that can perform work, report on and change their state, and “communicate” with other objects in the system.

Abstraction in any programming language works in many ways. It can be seen from creating subroutines to defining interfaces for making low level language calls. Some abstractions try to limit the breadth of concepts a programmer needs, by completely hiding the abstractions they in turn are built on, e.g. design patterns.

Typically abstraction can be seen in two ways:

**Data abstraction** is the way to create complex data types and exposing only meaningful operations to interact with data type, where as hiding all the implementation details from outside works.

**Control abstraction** is the process of identifying all such statements and expose them as a unit of work. We normally use this feature when we create a function to perform any work.

#### Encapsulation

Wrapping data and methods within classes in combination with implementation hiding (through access control) is often called encapsulation. The result is a data type with characteristics and behaviors. Encapsulation essentially has both i.e. information hiding and implementation hiding.

“***Whatever changes, encapsulate it***“. It has been quoted as a famous design principle. For that matter in any class, changes can happen in data in runtime and changes in implementation can happen in future releases. So, encapsulation applies to both i.e. data as well as implementation.

SO, they can relate like following :

– Abstraction is more about ‘What‘ a class can do. [Idea]  
– Encapsulation is more about ‘How‘ to achieve that functionality. [Implementation]

# Difference between interfaces and abstract classes?

Basic differences between interfaces and abstract class can be counted as follows:

* ~~An interface cannot have any methods, whereas an abstract class can~~[Not true after [java 8 default methods](https://howtodoinjava.com/java8/default-methods-in-java-8/)]
* A class can implement many interfaces but can have only one superclass (abstract or not)
* An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

You should remember that : “When you can fully describe the concept in terms of “**what it does**” without needing to specify any of “**how it does**“, then you should use an interface. If you need to include some implementation details, then you will need to represent your concept in an abstract class.”

Also, if i talk differently : Are there many classes that can be “grouped together” and described by one noun? If so, have an abstract class by the name of this noun, and inherit the classes from it. For example Cat and Dog can both inherit from abstract class Animal, and this abstract base class will implement a method void breathe() which all animals will thus do in exactly the same fashion.

What kinds of verbs can be applied to my class, that might in general also be applied to others? Create an interface for each of these verbs. For example, All animals can be fed, so I will create an interface called IFeedable and have Animal implement that. Only Dog and Horse are nice enough though to implement ILikeable, but some are not.

As said by someone: the main difference is where you want your implementation. By creating an interface, you can move your implementation to any class that implements your interface. By creating an abstract class, you can share implementation for all derived classes in one central place, and avoid lots of bad things like code duplication.

# How StringBuffer save the memory?

A [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) is implemented as an [immutable object](https://howtodoinjava.com/java/related-concepts/how-to-make-a-java-class-immutable/); that is, when you initially decide to put something into a Stringobject, the JVM allocates a fixed-width array of exactly the size of your initial value. This is then treated as a constant inside the JVM, which allows for very significant performance savings in the case where the String’s value is not changed. However, if you decide to change the String’s contents in any way, what the JVM then essentially does is copy the contents of the original String into a temporary space, make your changes, then save those changes into a whole new memory array. Thus, making changes to a String’s value after initialization is a fairly expensive operation.

[StringBuffer](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html), on the other hand, is implemented as a dynamically – growable array inside the JVM, which means that any change operation can occur on the existing memory location, with new memory allocated only as-needed. However, there is no opportunity for the JVM to make optimizations around the StringBuffer, since its contents are assumed to be changeable at any instance.

# Why wait and notify is declared in Object class instead of Thread?

[Wait, notify, notifyAll methods](https://howtodoinjava.com/java/multi-threading/how-to-work-with-wait-notify-and-notifyall-in-java/) are only required when you want your threads to access a shared resource and a shared resource could be any java object which is on the heap. So, these methods are defined on the core Object class so that each object has control of allowing threads to wait on it’s monitor. Java doesn’t have any special object which is used for sharing a common resource. No such data structure is defined.So, onus is given on the Object class to be able to become shared resource providing it will helper methods like wait(),notify() and notifyAll().

Java is based on [Hoare](https://en.wikipedia.org/wiki/Tony_Hoare)‘s monitors idea. In Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:

*– a Thread  
– a monitor (any object)*

In the Java design, the thread can not be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on). This is a good design, because if we could make any other thread to wait on a desired monitor, this would lead to an “intrusion”, posing difficulties on designing /programming concurrent programs. Remember that in Java all operations that are intrusive in another thread’s execution are deprecated (e.g. stop()).

# Write Java program to create deadlock in Java and fix it ?

In java, a deadlock is a situation where minimum two threads are holding lock on some different resource, and both are waiting for other resource to complete its task. And, none is able to leave the lock on resource it is holding.

# What happens if your Serializable class contains a member which is not serializable? How do you fix it?

In this case, **NotSerializableException** will be thrown at runtime. To fix this issue, a very simple solution is to mark such fields transient. It means these fields will not be serialized. If you want to save the state of these fields as well then you should consider reference variables which already implements [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html) interface.

You also might need to use readResolve() and writeResolve() methods. Lets summarize this:

* First, make your non-serialisable field transient.
* In writeObject(), first call defaultWriteObject() on the stream to store all the non-transient fields, then call other methods to serialize the individual properties of your non-serializable object.
* In readObject(), first call defaultReadObject() on the stream to read back all the non-transient fields, then call other methods (corresponding to the ones you added to writeObject) to deserialise your non-serializable object.

# Explain transient and volatile keywords in java?

#### Transient

“The ***transient*** keyword in Java is used to indicate that a field should not be serialized.” According to language specification: Variables may be marked transient to indicate that they are not part of the persistent state of an object. For example, you may have fields that are derived from other fields, and should only be done so programmatically, rather than having the state be persisted via serialization.

For example, in class BankPayment.java fields like principal and rate can be serialized while interest can be calculated any time even after de-serialization.

If we recall, each thread in java has its own local memory space as well and it does all read/write operations in its local memory. Once all operations are done, it write back the modified state of variable in main memory from where all threads access this variable. Normally, this is the default flow inside JVM. But, the volatile modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. It means every time thread want to read the state of variable, it must flush its local memory state and update the variable from main memory.

#### Volatile

volatile is most useful in lock-free algorithms. You mark the variable holding shared data as volatile when you are not using locking to access that variable and you want changes made by one thread to be visible in another, or you want to create a “happens-after” relation to ensure that computation is not re-ordered, again, to ensure changes become visible at the appropriate time.

The volatile should be used to safely publish immutable objects in a multi-threaded Environment. Declaring a field like public volatile ImmutableObject foo secures that all threads always see the currently available instance reference.

# Difference between Iterator and ListIterator?

We can use Iterator to traverse a Set or a List or a Map. But ListIterator can only be used to traverse a List only. Other differences can be listed as below.

You can –

1. iterate backwards.
2. obtain the index at any point.
3. add a new value at any point.
4. set a new value at that point.

e.g.

*List<String> names = new ArrayList<String>();*

*names.add("Alex");*

*names.add("Bob");*

*names.add("Charles");*

*System.out.println(names);*

*ListIterator<String> listIterator = names.listIterator();*

*//Add a value at any place using ListIterator*

*while(listIterator.hasNext()){*

*listIterator.next();*

*listIterator.add("Lokesh");*

*}*

*System.out.println(names);*

*listIterator = names.listIterator();*

*//Set a value at any place using ListIterator*

*while(listIterator.hasNext()){*

*listIterator.next();*

*listIterator.set("John");*

*}*

*System.out.println(names);*

*Output:*

*[Alex, Bob, Charles]*

*[Alex, Lokesh, Bob, Lokesh, Charles, Lokesh]*

*[John, John, John, John, John, John]*

Clearly, I am able to add elements at random places in list while iterating over it – and similarily I can change any element as well. Using Iterator, it is not possible.

# Core Java Interview Questions – Part 3

In [**interview questions series: part 1**](https://howtodoinjava.com/java/interviews-questions/core-java-interview-questions-series-part-1/) and [**part 2**](https://howtodoinjava.com/java/interviews-questions/core-java-interview-questions-series-part-2/) we discussed some important questions which interviewer ask most of the time. Now is the time to take that discussion forward. In this post, I will talk about below given list of questions.

Deep copy and shallow copy?

What is synchronization? Class level locking and object level locking?

Difference between sleep() and wait()?

Can you assign null to this reference variable?

What if the difference between && and &??

How to override equals and hashCode() methods?

Explain all access modifiers?

What is garbage collection? Can we enforce it?

What is native keyword?

What is serialization? Explain the catches?

# Deep copy and shallow copy?

A clone is an exact copy of the original. In java, it essentially means the ability to create an object with similar state as the original object. The clone() method provides this functionality.

Shallow copies duplicate as little as possible.  By default, java cloning is shallow copy or ‘field by field copy’ i.e. as the Object class does not have idea about the structure of class on which clone() method will be invoked. So, JVM when called for cloning, do following things:

1) If the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned.

2) If the class contains members of any class type then only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated. Here, we want a clone which is independent of original and making changes in clone should not affect original.

Deep cloning requires satisfaction of following rules.

1. No need to separately copy primitives.
2. All the member classes in original class should support cloning and in clone method of original class in context should call super.clone() on all member classes.
3. If any member class does not support cloning then in clone method, one must create a new instance of that member class and copy all its attributes one by one to new member class object. This new member class object will be set in cloned object.

[Read more about cloning here](https://howtodoinjava.com/java/cloning/a-guide-to-object-cloning-in-java/).

# What is synchronization? Object level locking and class level locking?

***Synchronization*** refers to multi-threading. A synchronized block of code can only be executed by one thread at a time. Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method’s object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Synchronization in java is achieved using synchronized keyword. You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.

***Object level locking*** is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.

***Class level locking*** prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of  DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

[Read more about synchronization here.](https://howtodoinjava.com/java/multi-threading/thread-synchronization-object-level-locking-and-class-level-locking/)

# Difference between sleep() and wait()?

sleep() is a method which is used to hold the process for few seconds or the time you wanted but in case of wait() method thread goes in waiting state and it won’t come back automatically until we call the notify() or notifyAll().

The major difference is that wait() releases the lock or monitor while sleep() doesn’t releases any lock or monitor while waiting. Wait is used for inter-thread communication while sleep is used to introduce pause on execution, generally.

Thread.sleep() sends the current thread into the “Not Runnable” state for some amount of time. The thread keeps the monitors it has aquired — i.e. if the thread is currently in a synchronized block or method no other thread can enter this block or method. If another thread calls t.interrupt() it will wake up the sleeping thread. Note that sleep is a static method, which means that it always affects the current thread (the one that is executing the sleep method). A common mistake is to call t.sleep() where t is a different thread; even then, it is the current thread that will sleep, not the t thread.

object.wait() sends the current thread into the “Not Runnable” state, like sleep(), but with a twist. Wait is called on a object, not a thread; we call this object the “lock object.” Before lock.wait() is called, the current thread must synchronize on the lock object; wait() then releases this lock, and adds the thread to the “wait list” associated with the lock. Later, another thread can synchronize on the same lock object and call lock.notify(). This wakes up the original, waiting thread. Basically, wait()/notify() is like sleep()/interrupt(), only the active thread does not need a direct pointer to the sleeping thread, but only to the shared lock object.

[Read the difference in detail here.](https://howtodoinjava.com/java/multi-threading/difference-between-sleep-and-wait/)

# Can you assign null to this reference variable?

NO. You can’t. In java, left hand side of an assignment statement must be a variable. ‘this’ is a special keyword which represent the current instance always. This is not any variable.

Similarly, null can not be assigned to ‘super’ or any such keyword for that matter.

# What if the difference between && and &??

& is bitwise and && is logical.

* & evaluates both sides of the operation.
* && evaluates the left side of the operation, if it’s true, it continues and evaluates the right side.

[Read here for deep understanding.](https://en.wikipedia.org/wiki/Bitwise_operation)

# How to override equals and hashCode() methods?

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

hashCode() method is used to get a unique integer for given object. This integer is used for determining the bucket location, when this object needs to be stored in some HashTable like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.

equals() method, as name suggest, is used to simply verify the equality of two objects.  Default implementation simply check the object references of two objects to verify their equality.

Below are the important points to keep remember while overriding these functions.

1. Always use same attributes of an object to generate hashCode() and equals() both. As in our case, we have used employee id.
2. equals() must be consistent (if the objects are not modified, then it must keep returning the same value).
3. Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().
4. If you override one, then you should override the other.

[Read more interesting facts and how to guide here.](https://howtodoinjava.com/java/related-concepts/working-with-hashcode-and-equals-methods-in-java/)

# Explain all access modifiers?

Java classes, fields, constructors and methods can have one of four different access modifiers:

***Private*** If a method or variable is marked as private, then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.  
If a class is marked as private then no external class an access the class. This doesn’t really make so much sense for classes though. Therefore, the access modifier private is mostly used for fields, constructors and methods.

***Default*** The default access level is declared by not writing any access modifier at all. Default access levels means that code inside the class itself + code inside classes in the same package as this class, can access the class, field, constructor or method. Therefore, the default access modifier is also sometimes called a package access modifier.

Subclasses cannot access methods and member variables in the superclass, if they have default accessibility declared, unless the subclass is located in the same package as the superclass.

***Protected*** The protected access modifier does the same as the default access, except subclasses can also access protected methods and member variables of the superclass. This is true even if the subclass is not located in the same package as the superclass.

***public***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifiers** | Same Class | Same Package | Subclass | Other ackages |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| default | Y | Y | N | N |
| private | Y | N | N | N |

The public access modifier means that all code can access the class, field, constructor or method, regardless of where the accessing code is located.

# What is garbage collection? Can we enforce it?

Garbage collection is an automatic memory management feature in many modern programming languages, such as Java and languages in the .NET framework. Languages that use garbage collection are often interpreted or run within a virtual machine like the JVM. In each case, the environment that runs the code is also responsible for garbage collection. A GC has two goals: any unused memory should be freed, and no memory should be freed unless the program will not use it anymore.

Can you force garbage collection?? Nope, System.gc() is as close as you can get. Your best option is to call System.gc() which simply is a hint to the garbage collector that you want it to do a collection. There is no way to force and immediate collection though as the garbage collector is non-deterministic. Also, under the documentation for OutOfMemoryError it declares that it will not be thrown unless the VM has failed to reclaim memory following a full garbage collection. So if you keep allocating memory until you get the error, you will have already forced a full garbage collection.

[Read more about garbage collection here.](https://howtodoinjava.com/java/garbage-collection/revisiting-memory-management-and-garbage-collection-mechanisms-in-java/)

# What is native keyword? Explain in detail?

The native keyword is applied to a method to indicate that the method is implemented in native code using JNI. It marks a method, that it will be implemented in other languages, not in Java.

Native methods were used in the past to write performance critical sections but with Java getting faster this is now less common. Native methods are currently needed when

* You need to call a library from Java that is written in other language.
* You need to access system or hardware resources that are only reachable from the other language (typically C). Actually, many system functions that interact with real computer (disk and network IO, for instance) can only do this because they call native code.

The downsides of using native code libraries are also significant:

1. JNI / JNA have a tendency to destabilize the JVM, especially if you try to do something complicated. If your native code gets native code memory management wrong, there’s a chance that you will crash the JVM. If your native code is non-reentrant and gets called from more than one Java thread, bad things will happen … sporadically. And so on.
2. Java with native code is harder to debug than pure Java or pure C/C++.
3. Native code can introduce significant platform dependencies / issues for an otherwise platform independent Java app.
4. Native code requires a separate build framework, and that may have platform / portability issues as well.

# What is serialization? Explain the catches?

In computer science, in the context of data storage and transmission, serialization is the process of translating data structures or object state into a format that can be stored  and “resurrected” later in the same or another computer environment.  When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.

Java provides automatic serialization which requires that the object be marked by implementing the java.io.Serializable interface. Implementing the interface marks the class as “okay to serialize,” and Java then handles serialization internally. There are no serialization methods defined on the Serializable interface, but a serializable class can optionally define methods with certain special names and signatures that if defined, will be called as part of the serialization/deserialization process.

Once an object is serialized, changes in its class break the de-serialization process. To identify the future changes in your class which will be compatible and others which will prove incompatible, please read the[**full guide here**](https://howtodoinjava.com/java/serialization/a-mini-guide-for-implementing-serializable-interface-in-java/). In short, I am listing down here:

**Incompatible changes**

* Deleting fields
* Moving classes up or down the hierarchy
* Changing a non-static field to static or a non-transient field to transient
* Changing the declared type of a primitive field
* Changing the writeObject or readObject method so that it no longer writes or reads the default field data
* Changing a class from Serializable to Externalizable or vice-versa
* Changing a class from a non-enum type to an enum type or vice versa
* Removing either Serializable or Externalizable
* Adding the writeReplace or readResolve method to a class

**Compatible changes**

* Adding fields
* Adding/ Removing classes
* Adding writeObject/readObject methods [defaultReadObject or defaultWriteObject should be called first]
* Removing writeObject/readObject methods
* Adding java.io.Serializable
* Changing the access to a field
* Changing a field from static to non-static or transient to non transient

# Top 40 Java collection interview questions and answers

Without argument, java collections is one of the most important area where you will be tested in any position whether junior or senior. The scope is so much wide, that its almost impossible to cover all the questions. Yet based on my previous interviews, I am attempting to put as many as possible GOOD **java collection interview questions** you must know.

I am aiming both beginners and senior level questions, so bear with me if you found some questions too basic because they might be useful for some junior developers.

**Java collection interview questions**

General questions

[1) What is the Java Collections API? List down its advantages?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#what_is_collection_in_java)

[2) Explain Collections hierarchy?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#collections_hierarchy)

[3) Why Collection interface does not extend Cloneable and Serializable interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#why_collection_not_extend_cloneable_serializable)

[4) Why Map interface does not extend Collection interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#why_map_not_extend_collection)

List interface related

[5) Why we use List interface? What are main classes implementing List interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#what_is_list_in_java)

[6) How to convert an array of String to ArrayList?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#convert_array_of_String_to_ArrayList)

[7) How to reverse the list?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#reverse_list)

Set interface related

[8) Why we use Set interface? What are main classes implementing Set interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#what_is_set_in_java)

[9) How HashSet store elements?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#how_set_store_elements)

[10) Can a null element added to a TreeSet or HashSet?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#null_in_set)

Map interface related

[11) Why we use Map interface? What are main classes implementing Map interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#what_is_map_in_java)

[12) What are IdentityHashMap and WeakHashMap?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#identityHashMap_weakHashMap_differences)

[13) Explain ConcurrentHashMap? How it works?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#how_concurrentHashMap_works)

[14) How hashmap works?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#how_hashmap_works)

[15) How to design a good key for hashmap?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#good_key_for_hashmap)

[16) What are different Collection views provided by Map interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#different_collection_views)

[17) When to use HashMap or TreeMap?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#hashmap_or_treemap)

Tell the difference questions

[18) Difference between Set and List?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_set_and_list)

[19) Difference between List and Map?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_map_and_list)

[20) Difference between HashMap and HashTable?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_hashmap_and_hashtable)

[21) Difference between Vector and ArrayList?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_vector_and_arraylist)

[22) Difference between Iterator and Enumeration?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_iterator_and_enumerator)

[23) Difference between HashMap and HashSet?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_hashmap_and_hashset)

[24) Difference between Iterator and ListIterator?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_iterator_and_listiterator)

[25) Difference between TreeSet and SortedSet?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_treeset_and_sortedset)

[26) Difference between ArrayList and LinkedList?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#difference_arraylist_and_linkedlist)

More questions

[27) How to make a collection read only?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#read_only_collection)

[28) How to make a collection thread safe?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#thread_safe_collection)

[29) Why there is not method like Iterator.add() to add elements to the collection?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#why_no_iterator_add)

[30) What are different ways to iterate over a list?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#different_ways_to_iterate_list)

[31) What do you understand by iterator fail-fast property?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#fail_fast_iterator)

[32) What is difference between fail-fast and fail-safe?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#fail_safe_vs_fail_fast)

[33) How to avoid ConcurrentModificationException while iterating a collection?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#avoid_ConcurrentModificationException)

[34) What is UnsupportedOperationException?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#UnsupportedOperationException)

[35) Which collection classes provide random access of it’s elements?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#random_access_collections)

[36) What is BlockingQueue?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#BlockingQueue)

[37) What is Queue and Stack, list their differences?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#queue_and_stack_difference)

[38) What is Comparable and Comparator interface?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#Comparable_and_Comparator)

[39) What are Collections and Arrays class?](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#Collections_and_arrays)

[40) Recommended resources](https://howtodoinjava.com/interview-questions/useful-java-collection-interview-questions/#resources)

Without wasting time, let dig into java collections concepts.

# Java collection interview general questions

#### 1) What is the Java Collection framework? List down its advantages?

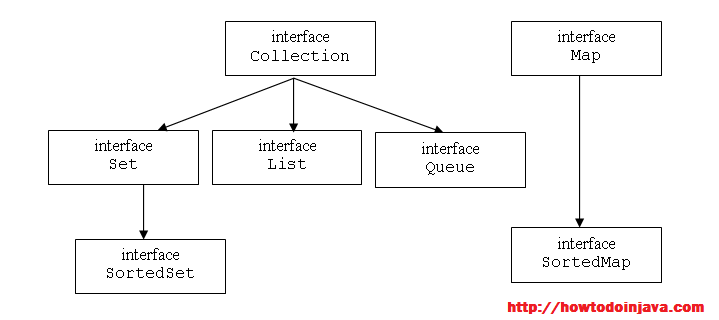
By definition, a collection is **an object that represents a group of objects**. Like in set theory, a set is group of elements. Easy enough !!

Prior to JDK 1.2, JDK has some utility classes such as Vector and HashTable, but there was no concept of Collection framework. Later from JDK 1.2 onwards, JDK felt the need of having a consistent support for reusable data structures. Finally, the collections framework was designed and developed primarily by Joshua Bloch, and was **introduced in JDK 1.2**.

Its most noticeable **benefits of java collections** can be listed as:

* Reduced programming effort due to ready to use code
* Increased performance because of high-performance implementations of data structures and algorithms
* Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth
* Easy to learn APIs by learning only some top level interfaces and supported operations

#### 2) Explain Collection’s hierarchy?

Java Collection Hierarchy

As shown in above image, collection framework has one interface at top i.e. **Collection**. It is **extended by Set, List and Queue interfaces**. Then there are loads of other classes in these 3 branches which we will learn in following questions.

Remember the signature of Collection interface. It will help you in many question.

|  |
| --- |
| public interface Collection extends Iterable {  //method definitions  } |

Framework also consist of Map interface, which is part of collection framework. but it does not extend Collection interface. We will see the reason in 4th question in this question bank.

#### 3) Why Collection interface does not extend Cloneable and Serializable interface?

Well, simplest answer is “**there is no need to do it**“. Extending an interface simply means that you are creating a subtype of interface, in other words a more specialized behavior and Collection interface is not expected to do what Cloneable and Serializable interfaces do.

Another reason is that not everybody will have a reason to have Cloneable collection because if it has very large data, then every **unnecessary clone operation will consume a big memory**. Beginners might use it without knowing the consequences.

Another reason is that **Cloneable and Serializable are very specialized behavior** and so should be implemented only when required. For example, many concrete classes in collection implement these interfaces. So if you want this feature. use these collection classes otherwise use their alternative classes.

#### 4) Why Map interface does not extend Collection interface?

A good answer to this interview question is “**because they are incompatible**“. Collection has a method add(Object o). Map can not have such method because it need key-value pair. There are other reasons also such as Map supports keySet, valueSet etc. Collection classes does not have such views.

Due to such big differences, Collection interface was not used in Map interface, and it was build in separate hierarchy.

# Java collection interview – List interface questions

#### 5) Why we use List interface? What are main classes implementing List interface?

A java list is a **“ordered” collection of elements**. This ordering is a **zero based index**. It does not care about duplicates. Apart from methods defined in Collection interface, it does **have its own methods** also which are largely to manipulate the collection **based on index location of element**. These methods can be grouped as search, get, iteration and range view. All above operations support index locations.

The main classes implementing List interface are: **Stack, Vector, ArrayList and LinkedList**. Read more about them in java documentation.

#### 6) How to convert an array of String to arraylist?

This is more of a programmatic question which is seen at beginner level. The intent is to check the knowledge of applicant in Collection utility classes. For now, lets learn that there are two utility classes in Collection framework which are mostly seen in interviews i.e. **Collections and Arrays**.

Collections class provides some static functions to perform specific operations on collection types. And Arrays provide utility functions to be performed on array types.

|  |
| --- |
| //String array  String[] words = {"ace", "boom", "crew", "dog", "eon"};  //Use Arrays utility class  List wordList = Arrays.asList(words);  //Now you can iterate over the list |

Please not that this function is not specific to String class, it will return List of element of any type, of which the array is. e.g.

|  |
| --- |
| //String array  Integer[] nums = {1,2,3,4};  //Use Arrays utility class  List numsList = Arrays.asList(nums); |

#### 7) How to reverse the list?

This question is just like above to test your knowledge of **Collections** utility class. Use it **reverse()** method to reverse the list.

|  |
| --- |
| Collections.reverse(list); |

# Java collection interview – Set interface questions

#### 8) Why we use Set interface? What are main classes implementing Set interface?

It **models the mathematical set in set theory**. Set interface is like List interface but with some differences. First, it is **not ordered collection**. So no ordering is preserved while adding or removing elements. The main feature it does provide is “**uniqueness of elements**“. It does not support duplicate elements.

Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

Based on above reasons, it **does not have operations based on indexes of elements like List**. It only has methods which are inherited by Collection interface.

Main classes implementing Set interface are :**EnumSet, HashSet, LinkedHashSet, TreeSet**. Read more on related java documentation.

#### 9) How HashSet store elements?

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;  } |

I will highly suggest you to read this post: [**How HashMap works in java?**](https://howtodoinjava.com/java/collections/how-hashmap-works-in-java/) This post will help you in answering all the HashMap related questions very easily.

#### 10) Can a null element added to a TreeSet or HashSet?

As you see, There is no null check in add() method in previous question. And HashMap also allows one null key, so **one “null” is allowed in HashSet**.

TreeSet uses the same concept as HashSet for internal logic, but uses NavigableMap for storing the elements.

|  |
| --- |
| private transient NavigableMap<E,Object> m;  // Dummy value to associate with an Object in the backing Map  private static final Object PRESENT = new Object(); |

NavigableMap is subtype of SortedMap which does not allow null keys. So essentially,**TreeSet also does not support null keys**. It will throw NullPointerException if you try to add null element in TreeSet.

# Java collection interview – Map interface questions

#### 11) Why we use Map interface? What are main classes implementing Map interface?

Map interface is a special type of collection which is **used to store key-value pairs**. It does not extend Collection interface for this reason. This interface provides methods to add, remove, search or iterate over various views of Map.

Main classes implementing Map interface are:**HashMap, Hashtable, EnumMap, IdentityHashMap, LinkedHashMap and Properties.**

#### 12) What are IdentityHashMap and WeakHashMap?

**IdentityHashMap** is similar to HashMap except that**it uses reference equality when comparing elements**. IdentityHashMap class is not a widely used Map implementation. While this class implements the Map interface, it intentionally violates Map’s general contract, which mandates the use of the equals() method when comparing objects. IdentityHashMap is designed for use only in the rare cases wherein reference-equality semantics are required.

**WeakHashMap** is an implementation of the Map interface **that stores only weak references to its keys**. Storing only weak references allows a key-value pair to be garbage collected when its key is no longer referenced outside of the WeakHashMap. This class is intended primarily for use with key objects whose equals methods test for object identity using the == operator. Once such a key is discarded it can never be recreated, so it is impossible to do a look-up of that key in a WeakHashMap at some later time and be surprised that its entry has been removed.

#### 13) Explain ConcurrentHashMap? How it works?

Taking from java docs:

**A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates**. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

Read more about how [**ConcurrentHashMap interview questions**](https://howtodoinjava.com/java/collections/popular-hashmap-and-concurrenthashmap-interview-questions/).

#### 14) How hashmap works?

The **most important question** which is most likely to be seen in every level of job interviews. You must be very clear on this topic., not only because it is most asked question but also it will open up your mind in further questions related to collection APIs.

Answer to this question is very large and you should read it my post: [**How HashMap works?**](https://howtodoinjava.com/java/collections/how-hashmap-works-in-java/) For now, lets remember that HashMap works **on principle of Hashing**. A map by definition is : “An object that maps keys to values”. To store such structure, **it uses an inner class Entry**:

|  |
| --- |
| static class Entry implements Map.Entry  {  final K key;  V value;  Entry next;  final int hash;  ...//More code goes here  } |

Here key and value variables are used to store key-value pairs. Whole entry object is stored in an array.

|  |
| --- |
| /\*\*  \* The table, re-sized as necessary. Length MUST Always be a power of two.  \*/  transient Entry[] table; |

The index of array is calculated on basis on hashcode of Key object. Read more of linked topic.

#### 15) How to design a good key for hashmap?

Another good question usually followed up after answering how hashmap works. Well, the most important constraint is **you must be able to fetch the value object back in future**. Otherwise, there is no use of having such a data structure. If you understand the working of hashmap, you will find it largely depends on hashCode() and equals() method of Key objects.

So a good key object**must provide same hashCode() again and again**, no matter how many times it is fetched. Similarly, same keys**must return true when compare with equals() method and different keys must return false**.

For this reason,**immutable classes are considered best candidate for HashMap keys**.

Read more : [**How to design a good key for HashMap?**](https://howtodoinjava.com/java/collections/how-to-design-a-good-key-for-hashmap/)

#### 16) What are different Collection views provided by Map interface?

Map interface provides 3 views of key-values pairs stored in it:

* *key set view*
* *value set view*
* *entry set view*

All the views can be navigated using iterators.

#### 17) When to use HashMap or TreeMap?

HashMap is well known class and all of us know that. So, I will leave this part by saying that it 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.

Please note that **all keys inserted into the map must implement the Comparable interface** (this is necessary to decide the ordering). Furthermore, all such keys must be mutually comparable: k1.compareTo(k2) must not throw a ClassCastException for any keys k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint (for example, the user attempts to put a string key into a map whose keys are integers), the put(Object key, Object value) call will throw a ClassCastException.

## Java collection interview – Tell the difference questions

#### 18) Difference between Set and List?

The most noticeable differences are :

* *Set is unordered collection where List is ordered collection based on zero based index.*
* *List allow duplicate elements but Set does not allow duplicates.*
* *List does not prevent inserting null elements (as many you like), but Set will allow only one null element.*

#### 19) Difference between List and Map?

Perhaps most easy question. **List is collection of elements where as map is collection of key-value pairs**. There is actually lots of differences which originate from first statement. They have**separate top level interface, separate set of generic methods, different supported methods and different views of collection**.

I will take much time hear as answer to this question is enough as first difference only.

#### 20) Difference between HashMap and HashTable?

There are several differences between HashMap and Hashtable in Java:

* Hashtable is synchronized, whereas HashMap is not.
* Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
* The third significant difference between HashMap vs Hashtable is that Iterator in the HashMap is a fail-fast iterator while the enumerator for the Hashtable is not.

#### 21) Difference between Vector and ArrayList?

Lets note down the differences:

* All the methods of Vector is synchronized. But, the methods of ArrayList is not synchronized.
* Vector is a Legacy class added in first release of JDK. ArrayList was part of JDK 1.2, when collection framework was introduced in java.
* By default, Vector doubles the size of its array when it is re-sized internally. But, ArrayList increases by half of its size when it is re-sized.

#### 22) Difference between Iterator and Enumeration?

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.

#### 23) Difference between HashMap and HashSet?

HashMap is collection of key-value pairs whereas HashSet is un-ordered collection of unique elements. That’s it. No need to describe further.

#### 24) Difference between Iterator and ListIterator?

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.

#### 25) Difference between TreeSet and SortedSet?

SortedSet is an interface which TreeSet implements. That’ it !!

#### 26) Difference between ArrayList and LinkedList?

* 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.

## More collection interview questions

#### 27) How to make a collection read only?

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.

#### 28) How to make a collection thread safe?

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.

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

The sole purpose of an Iterator is to enumerate through a collection. All collections contain the add() method to serve your purpose. There would be no point in adding to an Iterator because the **collection may or may not be ordered**. And **add() method can not have same implementation for ordered and unordered collections**.

#### 30) What are different ways to iterate over a list?

You can iterate over a list using following ways:

* Iterator loop
* For loop
* For loop (Advance)
* While loop

Read more : <http://www.mkyong.com/java/how-do-loop-iterate-a-list-in-java/>

#### 31) What do you understand by iterator fail-fast property?

**Fail-fast Iterators fail as soon as they realized that structure of Collection has been changed since iteration has begun**. Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection.

Fail-fast behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throws ConcurrentModificationException.

#### 32) What is difference between fail-fast and fail-safe?

You have understood fail-fast in previous question. **Fail-safe iterators** are just opposite to fail-fast. **They never fail if you modify the underlying collection on which they are iterating**, because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator.

Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException.

#### 33) How to avoid ConcurrentModificationException while iterating a collection?

You should first try to **find another alternative iterator which are fail-safe**. For example if you are using List and you can use ListIterator. If it is legacy collection, you can use enumeration.

If above options are not possible then you can use one of three changes:

* If you are using JDK1.5 or higher then you can use ConcurrentHashMap and CopyOnWriteArrayList classes. It is the recommended approach.
* You can convert the list to an array and then iterate on the array.
* You can lock the list while iterating by putting it in a synchronized block.

Please note that last two approaches will cause a performance hit.

#### 34) What is UnsupportedOperationException?

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.

#### 35) Which collection classes provide random access of it’s elements?

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it’s elements.

#### 36) What is BlockingQueue?

**A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.**

BlockingQueue methods come in four forms: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up.

Read the example usage of blocking queue in post :

[**How to use blocking queue?**](https://howtodoinjava.com/java-5/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/)

#### 37) What is Queue and Stack, list down their differences?

**A collection designed for holding elements prior to processing.** Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.  
**Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.**

**Stack is also a form of Queue but one difference, it is LIFO (last-in-first-out).**

Whatever the ordering used, the head of the queue is that element which would be removed by a call to remove() or poll(). Also note that Stack and Vector are both synchronized.

**Usage:** Use a queue if you want to process a stream of incoming items in the order that they are received.Good for work lists and handling requests.  
Use a stack if you want to push and pop from the top of the stack only. Good for recursive algorithms.

#### 38) What is Comparable and Comparator interface?

In java. all collection which have feature of automatic sorting, uses compare methods to ensure the correct sorting of elements. For example classes which use sorting are TreeSet, TreeMap etc.

**To sort the data elements a class needs to implement Comparator or Comparable interface**. That’s why all Wrapper classes like Integer,Double and String class implements Comparable interface.

**Comparable helps in preserving default natural sorting, whereas Comparator helps in sorting the elements in some special required sorting pattern.** The instance of comparator if passed usually as collection’s constructor argument in supporting collections.

#### 39) What are Collections and Arrays classes?

**Collections and Arrays classes are special utility classes to support collection framework core classes.** They provide utility functions to get read-only/ synchronized collections, sort the collection on various ways etc.

Arrays also helps array of objects to convert in collection objects. Arrays also have some functions which helps in copying or working in part of array objects.