**11.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.”

**12.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://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java) 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 &amp;&amp; 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.

**13.Explain abstraction and encapsulation? How are they related?**

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

**14.Difference between interfaces and abstract classes?**

**15.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 String object, 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.

**16.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()).

**17.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.

Read More: [Writing a deadlock and resolving in java](https://howtodoinjava.com/java/multi-threading/writing-a-deadlock-and-resolving-in-java/)

**18.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.

Also, I will highly recommend to read [**full guide on serialization in java**](https://howtodoinjava.com/java/serialization/a-mini-guide-for-implementing-serializable-interface-in-java/).

**19.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.

**20.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.

Happy Learning !!