Java Memory and Performance Interview Questions

How would you improve performance of a Java application?

- Pool valuable system resources like threads, database connections, socket connections etc. Emphasize on reuse of threads from a pool of threads. Creating new threads and discarding them after use can adversely affect performance. Also consider using multi-threading in your single-threaded applications where possible to enhance performance. Optimize the pool sizes based on system and application specifications and requirements. Having too many threads in a pool also can result in performance and scalability problems due to consumption of memory stacks and CPU context switching (i.e. switching between threads as opposed to doing real computation.).
- Minimize network overheads by retrieving several related items simultaneously in one remote invocation if possible. Remote method invocations involve a network round-trip, marshaling and unmarshaling of parameters, which can cause huge performance problems if the remote interface is poorly designed.

Most applications need to retrieve data from and save/update data into one or more databases. Database calls are remote calls over the network. In general data should be lazily loaded (i.e. load only when required as opposed to pre-loading from the database with a view that it can be used later) from a database to conserve memory but there are use cases (i.e. need to make several database calls) where eagerly loading data and caching can improve performance by minimizing network trips to the database. Data can be eagerly loaded with a help of SQL scripts with complex joins or stored procedures and cached using third party frameworks or building your own framework.

How would you refresh your cache?

You could say that one of the two following strategies can be used:

- Timed cache strategy where the cache can be replenished periodically (i.e. every 30 minutes, every hour etc). This is a simple strategy applicable when it is acceptable to show dirty data at times and also the data in the database does not change very frequently.
- Dirty check strategy where your application is the only one which can mutate (i.e. modify) the data in the database. You can set a "isDirty" flag to true when the data is modified in the database through your application and consequently your cache can be refreshed based on the "isDirty" flag.

How would you refresh your cache if your database is shared by more than one application?

You could use one of the following strategies:

- Database triggers: You could use database triggers to communicate between applications sharing the same database and write pollers which polls the database periodically to determine when the cache should be refreshed.
- XML messaging to communicate between other applications sharing the same database or separate databases to determine when the cache should be refreshed.
- Optimize your I/O operations: use buffering when writing to and reading from files and/or streams. Avoid writers/readers if you are dealing with only ASCII characters. You can use streams instead, which are faster. Avoid premature flushing of buffers. Also make use of the performance and scalability enhancing features such as non-blocking and asynchronous I/O, mapping of file to memory etc offered by the NIO (New I/O).
- Establish whether you have a potential memory problem and manage your objects efficiently: remove references to the short-lived objects from long-lived objects like Java collections etc to minimize any potential memory leaks. Also reuse objects where possible. It is cheaper to recycle objects than creating new objects each time. Avoid creating extra objects unnecessarily. For example use mutable StringBuffer/StringBuilder classes instead of immutable String objects in computation expensive loops and use static factory methods instead of constructors to recycle immutable objects. Automatic garbage collection is one of the most highly touted conveniences of Java. However, it comes at a price. Creating and destroying objects occupies a significant chunk of the JVM's time. Wherever possible, you should look for ways to minimize the number of objects created in your code:
    - For complex objects that are used frequently, consider creating a pool of recyclable objects rather than always instantiating new objects. This adds additional burden on the programmer to manage the pool, but in selected cases it can represent a significant performance gain. Use flyweight design pattern to create a pool of shared objects. Flyweights are typically instantiated by a flyweight factory that creates a limited number of flyweights based on some criteria. Invoking object does not directly instantiate flyweights. It gets it from the flyweight factory, which checks to see if it has a flyweight that fits a specific criteria (e.g. with or without GST etc) in the pool (e.g. HashMap). If the flyweight exists then return the reference to the flyweight. If it does not exist, then instantiate one for the specific criteria and add it to the pool (e.g. HashMap) and then return it to the invoking object.
    - If repeating code within a loop, avoid creating new objects for each iteration. Create objects before entering the loop (i.e. outside the loop) and reuse them if possible.
    - Use lazy initialization when you want to distribute the load of creating large amounts of objects. Use lazy initialization only when there is merit in the design.
- Where applicable apply the following performance tips in your code:
    - Use ArrayLists, HashMap etc as opposed to Vector, Hashtable etc where possible. This is because the methods in ArrayList, HashMap etc are not synchronized (Refer Q15 in Java Section). Even better is to use just arrays where possible.
    - Set the initial capacity of a collection (e.g. ArrayList, HashMap etc) and StringBuffer/StringBuilder appropriately. This is because these classes must grow periodically to accommodate new elements. So, if you have a very large ArrayList or a StringBuffer, and you know the size in advance then you can speed things up by setting the initial size appropriately.
    - Minimize the use of casting or runtime type checking like instanceof in frequently executed methods or in loops. The "casting" and "instanceof" checks for a class marked as final will be faster. Using "instanceof" construct is not only ugly but also

unmaintainable. Look at using visitor pattern

- Do not compute constants inside a large loop. Compute them outside the loop. For applets compute it in the init() method. Avoid nested loops (i.e. a "for" loop within another "for" loop etc) where applicable and make use of a Collection class
- Exception creation can be expensive because it has to create the full stack trace. The stack trace is obviously useful if you are planning to log or display the exception to the user. But if you are using your exception to just control the flow, which is not recommended, then throw an exception, which is precreated. An efficient way to do this is to declare a public static final Exception in your exception class itself.
- Avoid using System.out.println and use logging frameworks like Log4J etc, which uses I/O buffers
- Minimize calls to Date, Calendar, etc related classes.
  For example:

```
//Inefficient code
public boolean isInYearCompanyWasEstablished(Date
dateSupplied) {
Calendar cal = Calendar.getInstance();
cal.set(1998, Calendar.JAN, 01,0,0,0); //Should be read
from a .proprerties file
Date yearStart = cal.getTime();
cal.setTime(1998,Calendar.DECEMBER, 31,0,0,0);//Should be
read from .properties.
Date yearEnd = cal.getTime();
return dateSupplied.compareTo(yearStart) >=0 &&
dateSupplied.compareTo(yearEnd) <= 0;
}
```

The above code is inefficient because every time this method is invoked 1 "Calendar" object and two "Date" objects are unnecessarily created. If this method is invoked 50 times in your application then 50 "Calendar" objects and 100 "Date" objects are created. A more efficient code can be written as shown below using a static initializer block:

```
//efficient code
private static final YEAR_START;
private static final YEAR_END;
static{
Calendar cal = Calendar.getInstance();
cal.set(1998, Calendar.JAN, 01,0,0,0); //Should be read
from a .proprerties file
Date YEAR_START = cal.getTime();
cal.setTime(1998,Calendar.DECEMBER, 31,0,0,0);//Should be
read from .properties.
Date YEAR_END = cal.getTime();
}
public boolean isInYearCompanyWasEstablished(Date
dateSupplied) {
return dateSupplied.compareTo(YEAR_START) >=0 &&
dateSupplied.compareTo(YEAR_END) <= 0;
}
```

No matter, how many times you invoke the method

isInYearCompanyWasEstablished(""), only 1 "Calendar" object 2 "Date" objects are created, since the static initializer block is executed only once when the class is loaded into the JVM.
- Minimize JNI calls in your code.

How would you detect and minimize memory leaks in Java?

In Java, memory leaks are caused by poor program design where object references are long lived and the garbage collector is unable to reclaim those objects.

Detecting memory leaks:

- Use tools like JProbe, OptimizeIt etc to detect memory leaks.
- Use operating system process monitors like task manager on NT systems, ps, vmstat, iostat, netstat etc on UNIX systems.
- Write your own utility class with the help of totalMemory() and freeMemory() methods in the Java Runtime class. Place these calls in your code strategically for pre and post memory recording where you suspect to be causing memory leaks. An even better approach than a utility class is using dynamic proxies or Aspect Oriented Programming (AOP) for pre and post memory recording where you have the control of activating memory measurement only when needed.

Minimizing memory leaks:

In Java, typically memory leak occurs when an object of a longer lifecycle has a reference to objects of a short life cycle. This prevents the objects with short life cycle being garbage collected. The developer must remember to remove the references to the short-lived objects from the long-lived objects. Objects with the same life cycle do not cause any issues because the garbage collector is smart enough to deal with the circular references

- Design applications with an object's life cycle in mind, instead of relying on the clever features of the JVM. Letting go of the object's reference in one's own class as soon as possible can mitigate memory problems. Example: myRef = null;
- Unreachable collection objects can magnify a memory leak problem. In Java it is easy to let go of an entire collection by setting the root of the collection to null. The garbage collector will reclaim all the objects (unless some objects are needed elsewhere).
- Use weak references if you are the only one using it. The WeakHashMap is a combination of HashMap and WeakReference. This class can be used for programming problems where you need to have a HashMap of information, but you would like that information to be garbage collected if you are the only one referencing it.
- Free native system resources like AWT frame, files, JNI etc when finished with them. Example: Frame, Dialog, and Graphics classes require that the method dispose() be called on them when they are no longer used, to free up the system resources they reserve.

Why does the JVM crash with a core dump or a Dr.Watson error?

Any problem in pure Java code throws a Java exception or error. Java exceptions or errors will not cause a core dump (on UNIX systems) or a Dr.Watson error (on WIN32systems). Any serious Java problem will result in an OutOfMemoryError thrown by the JVM with the stack trace and consequently JVM will exit. These Java stack traces are very useful for identifying the cause for an abnormal exit of the JVM. So is there a way to know that OutOfMemoryError is about to occur? The Java J2SE 5.0 has a package called java.lang.management which has useful JMX beans that we can use to manage the JVM. One of these beans is the MemoryMXBean. An OutOfMemoryError can be thrown due to one of the following 4 reasons:

- JVM may have a memory leak due to a bug in its internal heap management implementation. But this is highly unlikely because JVMs are well tested for this.
- The application may not have enough heap memory allocated for its running. You can allocate more JVM heap size (with Xmx parameter to the JVM) or decrease the amount of memory your application takes to overcome this. To increase the heap space:

  ```
  java -Xms1024M -Xmx1024M
  ```

  Care should be taken not to make the -Xmx value too large because it can slow down your application. The secret is to make the maximum heap size value the right size.
- Another not so prevalent cause is the running out of a memory area called the "perm" which sits next to the heap. All the binary code of currently running classes is archived in the "perm" area. The "perm" area is important if your application or any of the third party jar files you use dynamically generate classes. For example: "perm" space is consumed when XSLT templates are dynamically compiled into classes, J2EE application servers, JasperReports, JAXB etc use Java reflection to dynamically generate classes and/or large amount of classes in your application. To increase perm space:

  ```
  java -XX:PermSize=256M -XX:MaxPermSize=256M
  ```
- The fourth and the most common reason is that you may have a memory leak in your application Know your worst friend, the Garbage Collector

So why does the JVM crash with a core dump or Dr.Watson error?

Both the core dump on UNIX operating system and Dr.Watson error on WIN32 systems mean the same thing. The JVM is a process like any other and when a process crashes a core dump is created. A core dump is a memory map of a running process. This can happen due to one of the following reasons:

- Using JNI (Java Native Interface) code, which has a fatal bug in its native code. Example: using Oracle OCI drivers, which are written partially in native code or JDBC-ODBC bridge drivers, which are written in non Java code. Using 100% pure Java drivers (communicates directly with the database instead of through client software utilizing the JNI) instead of native drivers can solve this problem. We can use Oracle thin driver, which is a 100% pure Java driver.
- The operating system on which your JVM is running might require a patch or a service pack