How would you improve performance of a Java application?

Best practices for designing Java applications

* Pool valuable system resources
  + Threads – Reuse threads
  + Database connections
  + Socket connection
* Optimize I/O operations
* Minimize network overheads
* Caching

Best practice while coding the application

ArrayLists,

HashMap etc as opposed to Vector, Hashtableetc where possible. This is because the

methods in ArrayList, HashMapetc are not synchronized 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. (Refer **Q15, Q17** in Java Section).

Minimise 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** (Refer

**Q11** in How would you go about…? section) to avoid “instanceof” construct.

􀂃Do not compute constants inside a large loop. Compute them outside the loop. For applets compute it

in the init() method.

􀂃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

􀂃Minimise calls to Date, Calendar, etc related classes.

􀂃Minimise JNI calls in your code.

Set performance requirements in the specifications, include a performance focus in the analysis and design

and also create a performance test environment.

**Pool valuable system resources** like threads, database connections, socket connections etc. Emphasise 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. Optimze the pool sizes based on system and application specifications and

requirements.

􀂃**Optimize your I/O operations:** use buffering (Refer **Q21** in Java section) 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).

􀂃**Minimize network overheads** by retrieving several related items simultaneously in one remote invocation if

possible. Remote method invocations involve a network round-trip, marshalling and unmarshalling of

parameters, which can cause huge performance problems if the remote interface is poorly designed. (Refer

**Q125** in Enterprise section).

􀂃**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 (Refer **Q64** in Java

section) to minimise 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 as discussed in **Q17** in Java section. 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 minimise the number of objects

created in your code:

􀂃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.

􀂃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 select cases can represent an order of magnitude performance gain.

􀂃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**:

How would you detect and minimise 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, OptimizeItetc to detect memory leaks.
* Use operating system process monitors like task manager on NT systems, ps, vmstat, iostat, netstatetc 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.

**Minimising 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 (Refer **Q33** in Java section).

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 (Refer **Q32** in Java section) 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 JDK 1.5 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 as

discussed in **Q64** in Java section.

[Good read/reference: “**Know Your Worst Friend, the Garbage Collector**” http://java.syscon.

com/read/84695.htm by Romain Guy]

**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.

􀂃The JVM implementation you are using may have a bug in translating system resources like threads, file

handles, sockets etc from the platform neutral Java byte code into platform specific operations. If this JVM’s

translated native code performs an illegal operation then the **operating system will instantly kill the**

**process and mostly will generate a core dump file,** which is a hexadecimal file indicating program’s state

in memory at the time of error. The core dump files are generated by the operating system in response to

certain signals. Operating system signals are responsible for notifying certain events to its threads and

processes. The JVM can also intercept certain signals like **SIGQUIT** which is kill -3 < process id > from the

operating system and it responds to this signal by printing out a Java stack trace and then continue to run.

The JVM continues to run because the JVM has a special built-in debug routine, which will trap **the signal -3**.

On the other hand signals like **SIGSTOP** (kill -23 <process id>) and **SIGKILL** (kill -9 <process id>) will cause

the JVM process to stop or die. The following JVM argument will indicate JVM not to pause on **SIGQUIT**

signal from the operating system.

**Java –Xsqnopause**