Java Class Metadata: A User Guide

Andrew Dinn: Red Hat

Part 2: Per Class Metadata Memory Use

Introduction

This is the second in a series of articles discussing OpenJDK Class Metadata. The first article introduces Class Metadata and explains how to obtain summary NMT statistics detailing Class Metadata and Symbol storage costs. This one shows how to obtain a per-class breakdown of Class Metadata costs, using Wildfly as the example program. It explains how to separate fixed, per-class or per-method costs from variable costs that depend on details of the class or method definitions. Finally, it takes a specific Wildfly class as an example and relates some specific, non-fixed costs to design and implementation decisions taken when defining the class, offering alternatives that employ less metadata.

Two subsequent articles will provide: complete details of the JVM’s internal representation of metadata, explaining how the various component costs arise; a complete list of all available per-class stats with advice on how likely they are to contribute to the overall cost and, in the case of the more expensive ones, how to avoid design choices that may exacerbate cost.

The JVM's internal metadata model fragments and distributes the information contained in class bytecode into a network of linked structures. The JVM provides a diagnostic capability, again implemented as a jcmd option, which reports the amount of storage used by each of the various structures in this internal model, allowing both detailed and summary reports of per-class metadata memory use to be obtained at any time during program execution. The breakdown includes costs for methods and their component elements aggregated across all methods in the class.

Of course, as with any advertised feature, 'small print may apply'. To be more precise, stats are obtained at a JVM safepoint i.e. a point where the VM stops all threads so that nothing will change while it is summing up the memory statistics (this is not much of a limitation since safepoints can be triggered on demand with sub-second delays). Also, to use the jcmd class stats option you need to start the JVM with the flag -XX:+UnlockDiagnosticVMOptions.

The class stats diagnostic provides the raw data to identify what and how many metadata objects exist in metadata memory. However, that is not quite the full story. It can be tricky to account for why specific metadata is present on behalf of some specific loaded class and yet more tricky to correlate the amount of metadata with specific structural features of the original source code. Interpreting the diagnostic statistics and relating them back to the source code requires some understanding of how compiled source code (bytecode) is turned into metadata. Before addressing that issue it is worth learning how to obtain the stats and seeing, in brief, what they look like.

Note that this metadata diagnostic can be viewed as a complement to the heap dump diagnostic (also provided as a jcmd option) which allows you to identify storage used by Java instances in the Java heap(s).

How Do I Obtain Class Metadata Statistics?

Obtaining Metadata stats is actually very simple. Here is a basic 3-step how-to for gathering the stats:

1) Start your app with -XX:+UnlockDiagnosticVMOptions e.g.

```
$ java -XX:+UnlockDiagnosticVMOptions HelloWait
```

or

```
$ PREPEND_JAVA_OPTS=-XX:+UnlockDiagnosticVMOptions \
    <wildfly_dl_dir>/bin/standalone.sh
```

n.b. HelloWait is a variant of Hello which prints the usual message then waits for a line of input.

n.b.b. You may also want to enable native memory tracking by passing command line argument -XX:`NativeMemoryTracking=summary/detail`. However, this is not necessary if you only want to use the class stats diagnostic.

n.b.b.b. all figure quoted in this and the following articles are for OpenJDK jdk8u-b151.

2) Find the pid of the JVM of interest

```
$ jcmd -l
13506 sun.tools.jcmd.JCmd -l
13442 /home/adinn/jboss/klasscount/wildfly-10.1.0.Final ...
```

3) Request a metaspace stats dump (it is probably best to redirect the output to a file)

```
$ jcmd 13442 GC.class_stats > class_stats.txt
```

The result is a file that looks like this

```
  13442:
Index Super InstBytes KlassBytes annotations    CpAll
  MethodCount Bytecodes MethodAll    ROAll    RWAll    Total
    ClassName
    1    -1   5163352        480           0        0
           0         0         0       24      584      608
    [C
    2    -1   2517064        480           0        0
           0         0         0       24      584      608
    [Ljava.lang.Object;
    3    46   2306624        560           0     1384
           7       149      1824     1096     2984     4080
    java.util.HashMap$Node
      . . .
           22063568    6336992       55256 18873504
       93169   3664807   23219712 16172528 35387024 51559552
                         12.3%         0.1%    36.6%
           -      7.1%     45.0%    31.4%    68.6%   100.0%
```

n.b. each line of statistics includes too many entries to fit on a page so wrapped entries have been colour-coded to make it easier to correlate values with the titles in the first row.

It is easier to analyse this data by trimming the process id output at the head of the listing and then importing it into a spreadsheet. In particular that allows the output to be sorted on different columns. This makes it easy to find outliers in each of the various statistics reported in a given column. Below is a table which shows the sort of view a spreadsheet provides listing just the first three columns. Note that each line of stats for a given class occupies two successive lines of the table.

Index	Super	**InstBytes**	KlassBytes	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
1	-1	5163352	480	0	0	0
	0	0	24	584	608	[C
2	-1	2517064	480	0	0	0
	0	0	24	584	608	[Ljava.lang.Object;
3	46	2306634	560	0	1384	7
	149	1824	1096	2984	4080	java.util.HashMap$Node

Each line details the storage used for the class named in the last column. The 3rd column is the only one which is not actually a metadata statistic. It identifies the amount of *heap bytes* used to store instances of the class and it is used to sort the listed classes. The remaining columns identify the number of off-heap bytes used for the JVM's internal model of the class, factored out into various different subsets which are totalled in the last but one column.

The full listing contains over 10,000 entries – one for each class loaded by EAP. The last two rows identify the total space used for a specific subset of the VM data summed across all classes, first as an absolute byte count then as a percentage of the total.

Index	Super	**InstBytes**	KlassBytes	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
10288	. . .					
	. . .					
		22063568	6336992	55256	18873504	93169
	3664807	23219712	16172528	35387025	51559552	
			12.3%	0.1%	36.6%	-
	7.1%	45.0%	31.4%	68.6%	100.0%	

Reading Class Metadata Statistics

Let's start with some basic orientation. The final entry in each line provides the name of the class for which metadata storage costs are being reported. The Index for each listed class provided in column 1 is reused in column 2 to identify the associated superclass (-1 indicates that the superclass is Object).

The remaining columns all indicate how much storage in *bytes* is used for some subset of a class's metadata class, slicing the cake in a variety of different ways. The bottom line numbers are in columns ROAll and RWAll (and Total). These columns include the total read-only storage and writeable storage (and their sum) needed to keep track of metadata for the class identified on that line. These summary figures are useful as a coarse measure, indicating which classes are 'fat' and which are 'lean'. However, that needs a great deal of qualifying before making a judgement that there is excess which might be trimmed.

InstBytes in column 3 is somewhat of an anomaly since it is not actually recording metadata use. It is the only significant interloper from Java heap stats, telling you how many bytes of heap are currently being used to store instances of the class. In all cases but one, this column is the product of two other stats (available, but not included in the default set), i.e. InstSize * InstCount. However, note the one special case of java.lang.Class (explained in the next article) where InstBytes gets inflated beyond the normal instance storage count.

The remaining columns record occurrence counts or sizings for different individual or aggregated JVM Metadata structures associated with a given class. There are essentially 4 groups of data corresponding to the 4 main groups of metadata structures created by the JVM. These 4 groups are summarized in the columns labelled KlassBytes, annotations(sic), CpAll and MethodAll. MethodCount identifies the number of methods in the class, equivalently the number of methods over which the MethodAll statistic has been aggregated. Dividing the latter by the former gives the average cost for each method in the class which is sometimes useful when trying to compare different classes. The final 3 stats, ROAll, RWAll and Total simply aggregate the 4 group stats. The other available stats, not shown in the default listing, pick out component stats from one of these four groups, and correspond to specific data structures used in the model.

It's important to understand what these fields mean before jumping to conclusions about how they relate to the way that the corresponding class has been coded. There are actually several other statistics which can be printed on request many of which break down some of the subsets reported above into even finer details -- we'll see some examples in the next article and a full list will be provided in a follow-up. Meanwhile, to get started, let’s investigate the significance of a few of the statistics in aggregate and then, picking an example class, relate the stats for that class back to the class definition.

Can I use Metadata Stats to improve my code?

One of the first things to look at in the stats is the total class count, with a view to questioning why all those classes are there. After Wildfly has booted and reached a steady state the JVM has loaded a large number of classes, roughly 10,200. That includes instantiable classes (also interfaces) defined by Wildfly and the JDK runtime (around 9,400), array classes derived from those instance classes (around 700) plus a small number of classes generated at runtime such as proxies, classes used to bind lambda forms, etc (around 100).

The first thing that stands out is that roughly half (5,400) of the classes have zero in the InstanceCount column. You may wonder why a class would be loaded when no instances have been created. Well, firstly, some of those classes may be interfaces. Others may be abstract i.e. only their subclasses can be instantiated. Some of them may be static-only classes i.e. they serve simply to group related static methods and fields. However, in most cases a class with zero instance count will be capable of being instantiated at some point but will not have been because the relevant code paths have not been entered. In which case why has the class been loaded?

Normally the JVM avoids loading a class until it is actually needed and in many cases this is when the application is about to create an instance of the class. However, that is not the only reason for loading a class. When one class is loaded it may be necessary to load a related class because it is referenced by the first one. For example, it may be occur as the type of a field, method parameter, return value or local variable. In other cases the type may be referenced because it is mentioned in an inherited super or an implemented interface. Clearly, one dependent class reference can lead to another and the number of classes needed in order to instantiate a single class and execute its code can grow quite large. A high proportion of zero InstanceCount classes may indicate that the class base has not been cleanly factored into independent groups of classes which are loaded and used only as needed (ironically, interfaces are often be used to hide implementation details but a bad design can just as easily expose them).

In the case of Wildfly the high number of zero InstanceCount classes is to do with its chosen policy of initializing many services before they are needed by a specific user deployment. This doesn’t require creation of much *instance data* but it does require loading all the classes referenced during initialization, leading to a high *metadata* cost. The flip-side of this approach is that Wildfly is able to process and deploy user applications more quickly than it would if it had to initialize and load the necessary services at the point where an app was dropped into its deployment directory.

This point is underlined by the fact that the total instance byte count, roughly 22MB, is far outweighed by the total metadata byte count, roughly 51MB. This is an empty server running no deployments. Many of the small number of instances which do exist include are caretaker objects needed to manage the various subsystems in the app server which will not scale with application data. It is reasonable to expect that a production system employing all the features of the app server would create a much larger number of instances in its heap. However, this does indicate that a full app server like Wildfly is a large container which provides many services. If an app only needs a few such services or is intended to run at a very low hit rate then it might be better to depot it in a smaller container specific to the application’s needs.

The class stats show that the largest subgroup of the metadata storage records details of methods. The next largest subgroup is class constant pool data, mostly comprised of arrays storing numeric literals and references to UTF text or Strings in the Symbol and String dictionaries each with an associated byte tag array. Note that the constant pool overhead does not include the storage for the symbols/strings, just the cost of storing pointers to those symbols. The third largest subgroup is data that records details of Java classes. Annotation data occupies a negligible amount of space. Method bytecodes form a small but substantial part of the MethodAll total . Note that method bytecodes are the only element of the original class file that is retained essentially in its original format.

Methods 45%: comprising 93,000 methods/23,750,000 bytes i.e. roughly 9 methods per class and 250 bytes per method

CpAll 31%: comprising 10,200 constant pools/18,840,000 bytes i.e. roughly 1850 bytes per pool or 200 x 8-byte pool entries + 1-byte tags

KlassBytes 12%: comprising 10,200 classes/6,340,000 bytes i.e. roughly 620 bytes per class

ByteCodes 7%: 8,990,000 bytes giving an average of 40 bytes per method

These metadata statistics are not untypical of the average size and distribution of data amongst these categories of statistic for other code bases. Although methods dominate, the storage required to model an individual method is not actually as large as that required for a class. Most importantly, the space needed to model the class pool data found in a class file is often very high and much higher than that needed for the class per se. Even though the JVM manages to limit much of the overhead by, replacing repeated UTF text with pointers to shared Symbols or Strings, the number of entries found in the constant pool in a class's bytecode file is a significant driver of the overall runtime metadata cost for that class.

It is interesting to compare these average figures with the size of the core C++ objects which model classes and methods. The comparison cannot be precise for two reasons. Firstly, a class or a method will be modeled as a group of related C++ objects rather than a single instance. Secondly, different classes and methods will be modeled in different ways, depending upon their definition, making the total size dependent upon class structure. However, it is still possible to distinguish how much of the data cost is fixed overheads which apply for every class or method vs variable overheads which depend on class or method structure.

For classes *per se* (i.e. ignoring how methods are modeled), the C++ type which models a Java class is the C++ class called Klass (I hope that sentence makes it clear why it is spelled with a 'K'). However, Klass is never actually instantiated directly. An array class with an object base type will be modeled as an instance of a subtype, ObjArrayKlass, of size 240 bytes. Array classes with primitive base, such as type, byte[], short[], etc, are modeled as instances of a different subtype, TypeArrayKlass, of size 224 bytes. A user defined class or interface will be modeled as an instance of a 3rd subtype, InstanceKlass, of size 424 bytes.

Fields declared in Klass model structure common to all Java classes. Fields in ObjArrayKlass and TypeArrayKlass model extra structure common to Java array classes (they actually differ very little, inheriting most of their fields from a common C++ parent class ArrayKlass). Fields of InstanceKlass model extra structure common to all instantiable classes and interfaces.

Some of the variable data cost is auxiliary C++ objects referenced from a Klass. A small contribution is made by 3 pointer fields, C++ arrays of Klass* pointers, defined in Klass itself. These list: 1) locally implemented interfaces, 2) indirectly implemented interfaces and/or 3) an ordered list of all secondary super types not on the direct super line (n.b. the array references may be null, shared across the fields or even be common, default values, depending on the inheritance pattern). Additionally, an InstanceKlass references an array of Method* pointers one for each implemented method and an array of short (16-bit) ints describing each field using 6 shorts, mostly CP indexes for field name, type etc. Obviously, the extra storage cost associated with these auxiliary arrays varies according to where the class sits in the object/array type hierarchy and how many methods and fields the InstanceKlass has. In most cases, these field values only make a minor contribution to the variable costs.

A much larger contribution to variable costs is made by a variety of data appended to the Klass instance at allocation time in a variable-length tail. The vast majority of this is the VTable and ITables appended to an InstanceKlass. Essentially, they are a lookup table used to resolve calls to virtual methods or interface methods. So, a VTable is an array of code pointers, one for each method which can be called via a virtual method invocations. Some entries point to code for an inherited method, others to code for a locally defined method. In rare cases a subclass may not define any new virtual methods so it could actually share its super’s VTable. However, this happens rarely enough that OpenJDK simply embeds a VTable in each InstanceKlass. So, the storage cost is determined by how many no-local, non-final methods the class declares.

An ITable performs a similar job, to a VTable, providing a lookup mechanism for interface method invocations. The tail area includes an ITable for each interface implemented by a method, linked in a chain. Each ITable is preceded by a Klass pointer identifying the interface to which the table belongs. Once again, the storage costs involved depend upon the number of methods in each interface and the number of implemented interfaces. Note that for both VTables and ITables the number of entries in a subclass or subinterface table is always the same as or greater than those in the class or interface being extended.

The fixed MethodBytes costs for modelling a method are split between two main types, Method and ConstMethod, which, respectively, store runtime derived state (like entry point addresses, etc) and runtime invariant state (like return type, etc). Neither is subtyped and their sizes are 88 bytes and 48 bytes, respectively. Variable costs include a method profiling object referenced from a field of class Method, which only gets generated as a side effect of JIT compilation, and a variety of auxiliary data appended to the ConstMethod object’s tail, including stack maps, exception tables, compressed line number tables, etc, whose presence and size depends on how the method is defined.

Looking at the figures quoted above for Wildfly, the constant Klass overhead accounts for about 4,200,000 bytes of the total 6,340,000 bytes (i.e. 9,500 * 240 + 700 * 188 bytes). So, the variable auxiliary data forms roughly 1/3rd of the total storage. A similar calculation for method data shows that the fixed overhead accounts for around 12,800,000 (93,000 * 136 bytes) of the total 23,700,000 MethodAll bytes i.e. just over half of the total.

Although these aggregate figures are illuminating they serve mostly to set expectations when attempting to locate specific opportunities for optimizing code. The averages elide an enormous amount of variation in the per-class and per-method statistics which give much more fruitful clues as to how to lower metadata costs. The best way to identify such opportunities is to sort the output on one of the columns.

For example, sorting on KlassBytes results in the following entries to the top of the table

Index	Super	InstBytes	**KlassBytes**	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
1422	8851	80	11320	0	52376	950
	20406	194104	120952	152320	273272	EjbLogger_$logger
2883	5251	24	10576	0	32960	1260
	26712	272440	168344	162904	331248	ORBUtilSystemException
1708	8851	64	10576	0	46752	857
	16746	185832	107120	150024	257144	ControllerLogger_$logger
492	8851	64	9336	0	38512	702
	11650	135792	81776	113368	195144	InfinispanLog_$logger

It turns out that when the table is sorted by KlassBytes almost all of the top entries are logger classes generated by the JBoss Logging package (if `ORBUtilSystemException` is excluded then the remaining top 35 entries are Jboss Logging related classes and 32 are the generated classes with names ending in _$logger).

Not only do these classes have a very large KlassBytes count, they also have a very large Method count and MethodAll count. The large Method count is, perhaps, not surprising since logger classes provide separate methods to log each possible error situation. However, by the same token this also means that they are very good candidates for optimization. Any unnecessary metadata cost in the implementation may potentially be multiplied by this high method count.

Looking at the largest generated class org.jboss.as.ejb3.logging.EjbLogger_$logger it has 950 methods which occupy 190,000 MethodAll bytes, on the face of it quite economical as it represents only 200 bytes [per method.

Index	Super	InstBytes	**KlassBytes**	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
1422	8851	80	11320	0	52376	950
	20406	194104	120952	152320	273272	EjbLogger_$logger

The classes involved here form a diamond inheritance lattice comprising two interfaces and two classes. JBoss Logging provides an interface called BasicLogger which declares about a dozen trace and format methods which know how to format data to a log file. The implementation of this class is provided by class DelegatingBasicLogger. For every message the EJB code wants to log it declares a corresponding method in interface EjbLogger anc lcients ar eexpected to include this class in order to be able to log messages. The abstract methods declared in this latter class include annotations which JBoss Logging uses to generate the underlying implementation class EjbLogger_$logger. The generated class inherits from DelegatingBasicLogger so that generated methods are able to use the common trace and format methods.

```
           EjbLogger   ---   extends    --->     BasicLogger
             /                                      /
        implements                              implements
           /                                      /
EjbLogger_$logger   ---   extends   --->   DelegatingBasicLogger
```

As an example of how this works, here is an original interface method along with decompiled code showing the way the underlying implementation is generated

```
  public interface EjbLogger extends BasicLogger {
    . . .
    @LogMessage(level = ERROR)
    @Message(id = 4, value = "failed to get tx manager status; 
ignoring")
    void getTxManagerStatusFailed(@Cause Throwable cause);
    . . .

public class EjbLogger_$logger extends DelagatingBasicLogger {
    . . .
    void getTxManagerStatusFailed(@Cause Throwable cause) {
      error(getTxManagerStatusFailed$str(), cause)
    }
    . . .
```

The generated code calls one of the log functions declared by BasicLogger (and defined in DelegatingBasicLogger) detremined by the error level attribute of the @LogMessage annotation. It passes in whatever arguments are declared in the interface method declaration and these are formatted into the log using a format String defined in the @Message annotation. An auxiliary Throwable argument is formatted separately by calling the Throwable's printStackTrace method.

Looking at the class stats listing for the interface EjbLogger it can be seen that it defines only 475 methods even though the generated class defines 950 methods:

Index	Super	InstBytes	**KlassBytes**	annotations	CpAll	MethodCount	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
8205	46	0	488	0	15536	475	66	69184	29264	60032	89296	EjbLogger

That is because the generated code actually includes two methods for every method in the parent interface. This can be shown by running javap on the relevant classes

```
$ javap -classpath .../org/jboss/as/ejb3/main/wildfly-ejb3-
10.1.0.Final.jar org.jboss.as.ejb3.logging.EjbLogger
public interface org.jboss.as.ejb3.logging.EjbLogger extends
org.jboss.logging.BasicLogger {
  . . .
  public abstract void cacheRemoveFailed(java.lang.Object);
  public abstract void cacheEntryNotFound(java.lang.Object);
  public abstract void asyncInvocationFailed(java.lang.Throwable);
  . . .
$ javap -classpath ../org/jboss/as/ejb3/main/wildfly-ejb3-
10.1.0.Final.jar org.jboss.as.ejb3.logging.EjbLogger_\$logger
public class org.jboss.as.ejb3.logging.EjbLogger_$logger extends
org.jboss.logging.DelegatingBasicLogger implements
org.jboss.as.ejb3.logging.EjbLogger,org.jboss.logging.BasicLogger,ja
va.io.Serializable {
  public org...EjbLogger_$logger(org.jboss.logging.Logger);
  public final void cacheRemoveFailed(java.lang.Object);
  protected java.lang.String cacheRemoveFailed$str();
  public final void cacheEntryNotFound(java.lang.Object);
  protected java.lang.String cacheEntryNotFound$str();
  public final void asyncInvocationFailed(java.lang.Throwable);
  protected java.lang.String asyncInvocationFailed$str();
  . . .
```

The generated class not only includes an implementation of each logging method declared in the interface but also a separate, auxiliary method which returns the format string declared in the @Message annotation. The auxiliary is named by adding the suffix $str to the original method name.

On the face of it there is no need to use an auxiliary method here. The string literal could be used directly in the generated code. Clearly ,the declaration of the auxiliary as `protected` implies that the generated class is expecting to be subclassed. This is indeed the case, the reason being to allow a subclass to re-implement this behaviour. Locale-specific subclasses of the _$logger class are expected to re-implement the $str methods as appropriate to the locale, while still inheriting the necessary log methods.

So, with this design the metadata cost includes 475 method objects in class EjbLogger and 950 methods in class EjbLogger_$logger. Also, since all the log methods are either public or protected the vtable for EjbLogger_$logger contains 950 8 byte words for entries above and beyond those inherited from DelegatingBasicLogger rand the itable for EjbLogger embedded in class EjbLogger_$logger contains 475 8 byte words. That explains why its KlassBytes count is so high.

A further cost using this design is that it also requires the logger to load (by name) and instantiate the relevant subclass (by reflection) if it uses anything other than the default locale setting. So, that requires one more Klass, up to 475 more methods and another VTable and ITable for the subclass with 475 slots (plus 200 slots for methods inheroted from . DelegatingBasicLogger.

There is actually one further hidden cost to generating the auxiliary methods which is not shown in the class metadata stats. Each generated logger method has the same name as the method in its parent interface. So, the class pool entries for the two classes contain, amongst other entries, 475 unique method name UTF Strings for the public API methods. Most of the names are quite long names, at least 20 characters and often more. So, these give rise to 475 Symbols in the symbol table each of which references a byte array containing 20 or more characters. The auxiliary class methods to retrieve the format strings are formed by appending “$str” to the original method name. So, use of this design gives rise to another 475 Symbols and associated, slightly longer byte arrays.

It would use less metadata simply to bypass any reliance on subclassing & overriding and generate each alternative, locale-specific implementation under the same name, ensuring the correct implementation is loaded by, for example, placing the desired implementation in a jar located earlier in the class path. Another way to achieve the same result would be to generate a generic class and locale-specific variants under different names, all of them implementing the generic EjbLogger interface but have the logger load (by name) the desired locale-specific class and create an instance (by reflection). Either way, the application would only ever need to load 1 interface Klass with 475 associated Methods and 1 subclass Klass with 475 associated Methods and a 475 entry ITable but no local contribution to the VTable (all local methods of EjbLogger and the generated variants could be declared final).

An even better solution would be to avoid the use of an interface. EjbLogger could be converted to a dummy class with empty methods, which would still allow other code to be compiled and linked against it. The generation stage could implement an alternative implementation with methods derived from the annotations, possibly supplemented by a translation step if a locale-specific variant is required. That would reduce the overhead from 2 Klass instances to one, 950 Methods to 475 and avoid the need for any ITables (or local VTable entries).

The above point about final methods hints at another small opportunity to reduce metadata here that is worth mentioning. All the generated loggers inherit implementations of the 200 or so methods declared in interface BasicLogger from their parent class DelegatingBasicLogger i.e. methods like:

```
  public boolean isTraceEnabled();
  public void trace(java.lang.Object);
  public void trace(java.lang.Object, java.lang.Throwable);
  public void trace(java.lang.String, java.lang.Object,
java.lang.Throwable);
  public void trace(java.lang.String, java.lang.Object,
java.lang.Object[], java.lang.Throwable)
  . . .
```

The implementations in DelegatingBasicLogger are not declared as final which says that they may, potentially, be overridden in generated implementations like EjbLogger_$logger. That means they must be included as entries in the VTable for both parent and generated child. It is arguably quite inappropriate for specific logger implementations to provide their own overriding implementations. By changing the declaration of these methods to be final they can be omitted from the VTables of DelegatingBasicLogger and all the generated XXX_$logger classes which extend it, saving 200 words in the parent class and in each generated subclass.