Logging StaticLog

When Static References to Log objects can be used

There are two very common patterns used with logging:

```java
public class Foo {
    private Log log = LogFactory.getLog(Foo.class);
    ....
}
```

and

```java
public class Foo {
    private static Log log = LogFactory.getLog(Foo.class);
    ....
}
```

The use of the static qualifier can be beneficial in some circumstances. However in others it is a very bad idea indeed, and can have unexpected consequences. This page describes when each solution is appropriate.

The Static Problem

The technical result of using static is obvious: there is only one Log reference shared across all instances of the class. This is clearly memory efficient; only one reference (4 or 8 bytes) is needed no matter how many instances are created. It is also CPU-efficient; the lookup required to find the Log instance is only done once, when the class is first referenced.

When writing stand-alone application code the use of static is a good idea.

However when the java code concerned is a library that may be deployed within a container of some sort (such as a J2EE server) then problems can occur. It is common for containers to create a hierarchy of java ClassLoader objects such that each "application" deployed within the container has its own ClassLoader but where there is some shared classloader that is in the ancestry of all the deployed "applications". In this case, when a class holding a static reference to a Log instance is deployed at the "application" level (eg at the "webapp" level in a servlet or j2ee container) there is also no problem; if multiple applications deploy the class in question then they each have their own copy of the class and there are no interactions with other applications.

However consider the case when a class using "private static Log log = ..." is deployed via a ClassLoader that is in the ancestry of multiple supposedly independent "applications". In this case, the log member is initialised only once, because there is only one copy of the class. That initialisation (typically) occurs the first time any code tries to instantiate that class or call a static method on it. When initialisation of the class occurs, what should the log member be set to? The options are:

- To reference an underlying Log object which is part of a hierarchy configured from information at the "container" level, ie not associated with any particular "application"
- To reference an underlying Log object which is part of a hierarchy configured from information somehow related to the current application
- To reference a "proxy" object which will determine what the "current application" is each time a log method is invoked, and delegate to the appropriate underlying Log object at that time.

The first option means that logging cannot be configured at a per-application level. Whatever logging configuration is set up will apply equally to every application within the container, and all the output from those applications will be mixed together. This is a major issue.

The second option means that the Log object will be configured according to the first application that called it. Other applications within the container will have their output sent to the destination configured for that first application. Obviously this is a major issue.

The third option allows per-application logging configuration, and correctly filters and outputs the required log messages. However the performance penalty paid is very large; this isn't an acceptable solution either.

Note that this discussion hasn't talked about "Thread Context ClassLoaders" or other technical points. The problem is not with the details but with the general concept: when the Log object is shared among multiple applications it isn't possible to have per-application configuration and reasonable performance.

The real root cause of this problem is that SHARED data (static members on a class) is being shared across supposedly independent "applications". If no classes are "shared" between applications then the problem does not exist. However it appears that (to my personal frustration) container vendors continue to encourage the use of shared classes, and application developers continue to use it.

The alternative solution is: avoid the use of "static" references to Log objects in any code that might be deployed into a shared classpath.

Does SLF4J have this problem?

Yes. SLF4J does suffer from exactly the issues listed above, and the recommendation is the same: avoid static references to log objects from code that might be deployed in a shared classpath.
There are a number of possible scenarios here. First however we need to define a term “TCCL-aware”. A logging library is “TCCL-aware” when its initialisation code uses the Thread Context ClassLoader to attempt to locate its configuration files, rather than the ClassLoader of the classes in the logging library itself. As an example, in its standard form, log4j is not “TCCL-aware”. However it can be made “TCCL-aware” via a “Contextual RepositorySelector” as described here:

http://www.qos.ch/logging/sc.jsp

Now onto some possible scenarios:

Suppose SLF4J is deployed at a shared level, that there is a class in the shared classpath with a static Logger reference, and that there is a class in the application classpath with a static logger. When the underlying logging library is not TCCL-aware then logging configuration is only read from the “shared” classpath, ie applies globally to all output from all applications deployed within the container. Output is correct, but there is no ability to enable debug logging for just one application, and all output from all applications is mixed together. When the underlying logging library is TCCL-aware, then there is per-application logging configuration. In addition, output from those classes deployed at the application level is correct. However the class deployed at the shared level will have its Log object initialised using the configuration for the very first application that calls that class; when other applications call that same class the logging output will go to the destination of the first application. Not good.

If SLF4J is deployed at both the shared and the application level, and parent-first classloading is selected for an application then behaviour is identical to the above. Even when child-first classloading is selected, if the underlying logging library is TCCL-aware then things also go poorly. However if child-first classloading is selected AND the underlying logging library is not TCCL-aware then the results are almost bearable: output from the shared class can only be configured globally, ie all output goes to the same destination regardless of which application made the call.

Unfortunately in most cases the authors of library code are not in the position of being able to specify to the library users where the code will be deployed, what classloading order will be selected, or what the behaviour of the underlying logging library should be. Therefore it is necessary to avoid the use of static references to SLF4J loggers for exactly the same reasons as commons-logging.

SLF4J does have less probability of encountering problems than commons-logging when used incorrectly, however. The reasons are:

- Currently not a lot of logging libraries are “TCCL-aware”, and
- Very little library code of the type deployed into shared classpath locations currently uses SLF4J.

Commons-logging is itself TCCL-aware specifically to enable application-level configuration of logging libraries that are not TCCL-aware; effectively therefore every logging library qualifies as “TCCL-aware” when used with commons logging. Commons-logging is also very widely used by many libraries that are frequently deployed via shared classloaders.

Note that this is section is NOT intended as criticism of SLF4J. As described earlier, the problem is due to the fundamental conflict between isolating logging between applications while sharing a static Log reference, and therefore SLF4J faces the same constraints as commons-logging in these scenarios.

Does java.util.logging have this problem?

Yes. Code using the java.util.logging api can suffer from exactly the issues listed above, and the recommendation is the same: avoid static references to log objects from code that might be deployed in a shared classpath.

The java1.4 java.util.logging package is an API, allowing multiple implementations of that API. Java runtimes provide one very simple implementation of the API, but containers such as J2EE servers typically plug in an alternate more sophisticated implementation.

When using the default implementation, the “static” problem doesn’t exist because the standard implementation is not container-aware. Of course this has the side-effect of making per-application configuration impossible for the reasons described earlier.

When a container provides an implementation that is container-aware (so that per-application log configuration is possible), then exactly the same situation occurs: when code in the shared classpath creates a Log object, it must be:

- initialised with config NOT from any application, or
- initialised with config from one of the available applications, or
- a proxy that determines the appropriate config each time a method is called

As described above, the first gives no per-app config, the second misdirects logging to the wrong app when called from a different application, and the third is extremely inefficient.

So just as for commons-logging and SLF4J, it is necessary to avoid static Log objects when using java.util.logging unless you know that the underlying implementation that will be available at runtime is not container-aware.

Just to be completely clear: this only applies to code deployed via a classloader that is shared across multiple “independent” applications. None of this discussion is relevant to normal application code, or to library code that is never deployed into a shared classpath. These categories of code can safely use static references.

Alternatives to static loggers

In most cases, simply leaving out the "static" qualifier from the Log reference is the correct solution. Even when a Class is shared between supposedly-independent applications (because the .class file is deployed via a shared classloader), instances of that class are not shared. The TCCL active when the instance is created and its Log member initialised will therefore be the same TCCL active when any of its methods which generate log output are called.

Note that in the case of commons-logging this does NOT increase the number of Log objects created; each instance will have a reference to the same Log object.

However the "lookup" of that Log object does need to be performed each time an instance is created; for objects which have short lifetimes this may be undesirable. In this situation, if the short-lived object has a reference to some longer-lived one then the longer-lived one can host the Log object, eg
public ShortLiver {
    private Owner owner;
    ShortLiver(Owner owner) {
        this.owner = owner;
    }
    public void doSomething() {
        owner.getShortLiverLog().debug("doSomething called");
    }
}

Alternatively, the Log object can be retrieved when necessary:

public ShortLiver {
    public void doSomething() {
        Log log = LogFactory.getLog(ShortLiver.class);
        log.debug("doSomething called");
    }
}

Note that this applies only to code that may be deployed in a shared ClassLoader. Normal application code need not be concerned with this issue at all.

What about static methods?

When Log objects are not static then logging from static methods presents a particular problem. In general, calling LogFactory.getLog within the static method is the appropriate solution:

public static void doStuff(...) {
    Log log = LogFactory.getLog("stuff");
    log.warn(...);
}

If the static method has a reference to some object that it could retrieve a logger from it, eg

public static void doStuff(AppContext context, ...) {
    context.getStuffLog().warn("oops");
}

This doesn't seem widely applicable though. Fancier solutions like retrieving log objects from a collection in a thread-local variable are probably not worth trying; the LogFactory.getLog method isn't that slow.

Container-assisted logging

A few containers have tried to work around the problems listed above. In particular, JBoss provides a custom log4j "filter" that uses internal knowledge about the container to make configuration at the container level work in a somewhat acceptable manner. As described above, deploying commons-logging (or SLF4J) at the shared level, and deploying a non-TCCL-aware underlying library has the undesirable effect of imposing a single logging configuration on all contained applications. However the jboss custom filter then allows log messages flowing through that single logging instance to be separated out again into different destinations according to the originating application. This has the desired effect in the end, although it is somewhat inefficient; turning up logging to debug on one application actually turns up logging to debug on all applications, though output from applications other than the one of interest is then suppressed again before being output.

Fixing Existing Library Code

If you already have some library-type code that uses static log objects, and want to fix it after reading this page, be careful of one issue: changing a static member to a non-static one changes the serialization format of the class. The serialVersionUID of the class must change, and as a result serialized versions of the old class will not load.

A possible (untested) alternative is to add a method like this:
Like static members, transient ones are not part of the serialized data so this should be a serialization-compatible change.

Another approach is to implement the readResolve method that is automatically called at deserialization time if present:

```java
private transient Log log = LogFactory.getLog(Some.class);
private Object readResolve() {
    log = LogFactory.getLog(Some.class);
    return this;
}
```

See javadoc of Serializable interface for further information.