Frequently Asked Questions about SLF4J

Generalities

About the SLF4J API

Implementing the SLF4J API

General questions about logging

Generalities

What is SLF4J?

SLF4J is a simple facade for logging systems allowing the end-user to plug-in the desired logging system at deployment time.

When should SLF4J be used?

In short, libraries and other embedded components should consider SLF4J for their logging needs because libraries cannot afford to impose their choice of logging framework on the end-user. On the other hand, it does not necessarily make sense for stand-alone applications to use SLF4J. Stand-alone applications can invoke the logging framework of their choice directly. In the case of logback, the question is moot because logback exposes its logger API via SLF4J.

SLF4J is only a facade, meaning that it does not provide a complete logging solution. Operations such

as configuring appenders or setting logging levels cannot be performed with SLF4J. Thus, at some point in time, any non-trivial application will need to directly invoke the underlying logging system. In other words, complete independence from the API underlying logging system is not possible for a stand-alone application. Nevertheless, SLF4J reduces the impact of this dependence to near-painless levels.

Suppose that your CRM application uses log4j for its logging. However, one of your important clients request that logging be performed through JDK 1.4 logging. If your application is riddled with thousands of direct log4j calls, migration to JDK 1.4 would be a relatively lengthy and error-prone process. Even worse, you would potentially need to maintain two versions of your CRM software. Had you been invoking SLF4J API instead of log4j, the migration could be completed in a matter of minutes by replacing one jar file with another.

SLF4J lets component developers to defer the choice of the logging system to the end-user but eventually a choice needs to be made.

Is SLF4J yet another logging facade?

SLF4J is conceptually very similar to JCL. As such, it can be thought of as yet another logging facade. However, SLF4J is much simpler in design and arguably more robust. In a nutshell, SLF4J avoid the class loader issues that plague JCL.

If SLF4J fixes JCL, then why wasn't the fix made in JCL instead of creating a new project?

This is a very good question. First, SLF4J static binding approach is very simple, perhaps even laughably so. It was not easy to convince developers of the validity of that approach. It is only after SLF4J was released and started to become accepted did it gain respectability in the relevant community.

Second, SLF4J offers two enhancements which tend to be underestimated. Parameterized log messages solve an important problem associated with logging performance, in a pragmatic way. Marker objects, which are supported by the `org.slf4j.Logger` interface, pave the way for adoption of advanced logging systems and still leave the door open to switching back to more traditional logging systems if need be.

When using SLF4J, do I have to recompile my application to switch to a different logging system?

No, you do not need to recompile your application. You can switch to a different logging system by removing the previous SLF4J binding and replacing it with the binding of your choice.

For example, if you were using the NOP implementation and would like to switch to log4j version 1.2, simply replace *slf4j-nop.jar* with *slf4j-log4j12.jar* on your class path but do not forget to add log4j-1.2.x.jar as well. Want to switch to JDK 1.4 logging? Just replace *slf4j-log4j12.jar* with *slf4j-jdk14.jar*.

What are SLF4J's requirements?

As of version 1.7.0, SLF4J requires JDK 1.5 or later. Earlier SLF4J versions, namely SLF4J 1.4, 1.5. and 1.6, required JDK 1.4.

Binding	Requirements
slf4j-nop	JDK 1.5
slf4j-simple	JDK 1.5
slf4j-log4j12	JDK 1.5, plus any other library dependencies required by the log4j appenders in use
slf4j-jdk14	JDK 1.5 or later
logback-classic	JDK 1.5 or later, plus any other library dependencies required by the logback appenders in use

Are SLF4J versions backward compatible?

From the clients perspective, the SLF4J API is backward compatible for all versions. This means than you can upgrade from SLF4J version 1.0 to any later version without problems. Code compiled with *slf4j-api-versionN.jar* will work with *slf4j-api-versionM.jar* for any versionN and any versionM. **To date, binary compatibility in slf4j-api has never been broken.**

However, while the SLF4J API is very stable from the client's perspective, SLF4J bindings, e.g. slf4j-simple.jar or slf4j-log4j12.jar, may require a specific version of slf4j-api. Mixing different versions of slf4j artifacts can be problematic and is strongly discouraged. For instance, if you are using slf4j-api-1.5.6.jar, then you should also use slf4j-simple-1.5.6.jar, using slf4j-simple-1.4.2.jar will not work.

At initialization time, if SLF4J suspects that there may be a version mismatch problem, it emits a warning about the said mismatch.

I am getting `IllegalAccessError` exceptions when using SLF4J. Why is that?

Here are the exception details.

```
Exception in thread "main" java.lang.IllegalAccessError: tried to access field
org.slf4j.impl.StaticLoggerBinder.SINGLETON from class org.slf4j.LoggerFactory
   at org.slf4j.LoggerFactory.<clinit>(LoggerFactory.java:60)
```

This error is caused by the static initializer of the `LoggerFactory` class attempting to directly access the SINGLETON field of `org.slf4j.impl.StaticLoggerBinder`. While this was allowed in SLF4J 1.5.5 and earlier, in 1.5.6 and later the SINGLETON field has been marked as private access.

If you get the exception shown above, then you are using an older version of slf4j-api, e.g. 1.4.3, with a new version of a slf4j binding, e.g. 1.5.6. Typically, this occurs when your Maven *pom.ml* file incorporates hibernate 3.3.0 which declares a dependency on slf4j-api version 1.4.2. If your *pom.xml* declares a dependency on an slf4j binding, say slf4j-log4j12 version 1.5.6, then you will get illegal access errors.

To see which version of slf4j-api is pulled in by Maven, use the maven dependency plugin as follows.

```
mvn dependency:tree
```

If you are using Eclipse, please do not rely on the dependency tree shown by m2eclipse.

In your *pom.xml* file, explicitly declaring a dependency on slf4j-api matching the version of the declared binding will make the problem go away.

Please also read the FAQ entry on backward compatibility for a more general explanation.

Why is SLF4J licensed under X11 type license instead of the Apache Software License?

SLF4J is licensed under a permissive X11 type license instead of the ASL or the LGPL because the X11 license is deemed by both the Apache Software Foundation as well as the Free Software Foundation as compatible with their respective licenses.

Where can I get a particular SLF4J binding?

SLF4J bindings for SimpleLogger, NOPLogger, Log4jLoggerAdapter and JDK14LoggerAdapter are contained within the files *slf4j-nop.jar*, *slf4j-simple.jar*, *slf4j-log4j12.jar*, and *slf4j-jdk14.jar*. These files ship with the official SLF4J distribution. Please note that all bindings depend on *slf4j-api.jar*.

The binding for logback-classic ships with the logback distribution. However, as with all other bindings, the logback-classic binding requires *slf4j-api.jar*.

Should my library attempt to configure logging?

Embedded components such as libraries not only do not need to configure the underlying logging framework, they really should not do so. They should invoke SLF4J to log but should let

the end-user configure the logging environment. When embedded components try to configure logging on their own, they often override the end-user's wishes. At the end of the day, it is the end-user who has to read the logs and process them. She should be the person to decide how she wants her logging configured.

In order to reduce the number of dependencies of our software we would like to make SLF4J an optional dependency. Is that a good idea?

This question pops up whenever a software project reaches a point where it needs to devise a logging strategy.

Let Wombat be a software library with very few dependencies. If SLF4J is chosen as Wombat's logging API, then a new dependency on *slf4j-api.jar* will be added to Wombat's list of dependencies. Given that writing a logging wrapper does not seem that hard, some developers will be tempted to wrap SLF4J and link with it only if it is already present on the classpath, making SLF4J an optional dependency of Wombat. In addition to solving the dependency problem, the wrapper will isolate Wombat from SLF4J's API ensuring that logging in Wombat is future-proof.

On the other hand, any SLF4J-wrapper by definition depends on SLF4J. It is bound to have the same general API. If in the future a new and significantly different logging API comes along, code that uses the wrapper will be equally difficult to migrate to the new API as code that used SLF4J directly. Thus, the wrapper is not likely to future-proof your code, but to make it more complex by adding an additional indirection on top of SLF4J, which is an indirection in itself.

INCREASED VULNERABILITY It is actually worse than that. Wrappers will need to depend on certain internal SLF4J interfaces which change from time to time, contrary to the client-facing API which never changes. Thus, wrappers are usually dependent on the major version they were compiled with. A wrapper compiled against SLF4J version 1.5.x will not work with SLF4J 1.6 whereas client code using `org.slf4j.Logger`, `LoggerFactory`, `MarkerFactory`, `org.slf4j.Marker`, and `MDC` will work fine with any SLF4J version from version 1.0 and onwards.

It is reasonable to assume that in most projects Wombat will be one dependency among many. If each library had its own logging wrapper, then each wrapper would presumably need to be configured separately. Thus, instead of having to deal with one logging framework, namely SLF4J, the user of Wombat would have to detail with Wombat's logging wrapper as well. The problem will be compounded by each framework that comes up with its own wrapper in order to make SLF4J optional. (Configuring or dealing with the intricacies of five different logging wrappers is not exactly exciting nor endearing.)

The logging strategy adopted by the Velocity project is a good example of the "custom logging abstraction" anti-pattern. By adopting an independent logging abstraction strategy, Velocity developers have made life harder for themselves, but more importantly, they made life harder for their users.

Some projects try to detect the presence of SLF4J on the class path and switch to it if present. While this approach seems transparent enough, it will result in erroneous location information. Underlying logging frameworks will print the location (class name and line number) of the wrapper instead of the real caller. Then there is the question of API coverage as SLF4J support MDC and markers in addition to parameterized logging. While one can come up with a seemingly working SLF4J-wrapper within hours, many technical issues will emerge over time which Wombat developers will have to deal with. Note that SLF4J has evolved over several years and has 260 bug reports filed against it.

For the above reasons, developers of frameworks should resist the temptation to write their own logging wrapper. Not only is it a waste of time of the developer, it will actually make life more difficult for the users of said frameworks and make logging code paradoxically more vulnerable to change.

What about Maven transitive dependencies?

As an author of a library built with Maven, you might want to test your application using a binding, say slf4j-log4j12 or logback-classic, without forcing log4j or logback-classic as a dependency upon your users. This is rather easy to accomplish.

Given that your library's code depends on the SLF4J API, you will need to declare slf4j-api as a compile-time (default scope) dependency.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.12</version>
</dependency>
```

Limiting the transitivity of the SLF4J binding used in your tests can be accomplished by declaring the scope of the SLF4J-binding dependency as "test". Here is an example:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.12</version>
  <scope>test</scope>
</dependency>
```

Thus, as far as your users are concerned you are exporting slf4j-api as a transitive dependency of your library, but not any SLF4J-binding or any underlying logging system.

Note that as of SLF4J version 1.6, in the absence of an SLF4J binding, slf4j-api will default to a no-operation implementation.

How do I exclude commons-logging as a Maven dependency?

alternative 1) explicit exclusion

Many software projects using Maven declare commons-logging as a dependency. Therefore, if you wish to migrate to SLF4J or use jcl-over-slf4j, you would need to exclude commons-logging in all of your project's dependencies which transitively depend on commons-logging. Dependency exclusion is described in the Maven documentation. Excluding commons-logging explicitly for multiple dependencies distributed on several *pom.xml* files can be a cumbersome and a relatively error prone process.

alternative 2) provided scope

Commons-logging can be rather simply and conveniently excluded as a dependency by declaring it in the *provided* scope within the pom.xml file of your project. The actual commons-logging classes would be provided by jcl-over-slf4j. This translates into the following pom file snippet:

```
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>1.1.1</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>1.7.12</version>
</dependency>
```

The first dependency declaration essentially states that commons-logging will be "somehow" provided by your environment. The second declaration includes jcl-over-slf4j into your project. As jcl-over-slf4j is a perfect binary-compatible replacement for commons-logging, the first assertion becomes true.

Unfortunately, while declaring commons-logging in the provided scope gets the job done, your IDE, e.g. Eclipse, will still place *commons-logging.jar* on your project's class path as seen by your IDE. You would need to make sure that *jcl-over-slf4j.jar* is visible before *commons-logging.jar* by your IDE.

alternative 3) empty artifacts

An alternative approach is to depend on an **empty** *commons-logging.jar* artifact. This clever approach first was imagined and initially supported by Erik van Oosten.

The empty artifact is available from a http://version99.qos.ch a high-availability Maven repository, replicated on several hosts located in different geographical regions.

The following declaration adds the version99 repository to the set of remote repositories searched by

Maven. This repository contains empty artifacts for commons-logging and log4j. By the way, if you use the version99 repository, please drop us a line at <version99 AT qos.ch>.

```
<repositories>
  <repository>
    <id>version99</id>
    <!-- highly available repository serving empty artifacts -->
    <url>http://version99.qos.ch/</url>
  </repository>
</repositories>
```

Declaring version 99-empty of commons-logging in the <dependencyManagement> section of your project will direct all transitive dependencies for commons-logging to import version 99-empty, thus nicely addressing the commons-logging exclusion problem. The classes for commons-logging will be provided by jcl-over-slf4j. The following lines declare commons-logging version 99-empty (in the dependency management section) and declare jcl-over-slf4j as a dependency.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>99-empty</version>
    </dependency>
    ... other declarations...
  </dependencies>
</dependencyManagement>

<!-- Do not forget to declare a dependency on jcl-over-slf4j in the        -->
<!-- dependencies section. Note that the dependency on commons-logging     -->
<!-- will be imported transitively. You don't have to declare it yourself. -->
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.12</version>
  </dependency>
  ... other dependency declarations
</dependencies>
```

About the SLF4J API

Why don't the printing methods in the Logger interface accept message of type Object, but only messages of type String?

In SLF4J 1.0beta4, the printing methods such as debug(), info(), warn(), error() in the Logger interface were modified so as to accept only messages of type String instead of Object.

Thus, the set of printing methods for the DEBUG level became:

```
debug(String msg);
debug(String format, Object arg);
debug(String format, Object arg1, Object arg2);
debug(String msg, Throwable t);
```

Previously, the first argument in the above methods was of type Object.

This change enforces the notion that logging systems are about decorating and handling messages of type String, and not any arbitrary type (Object).

Just as importantly, the new set of method signatures offer a clearer differentiation between the overloaded methods whereas previously the choice of the invoked method due to Java overloading rules were not always easy to follow.

It was also easy to make mistakes. For example, previously it was legal to write:

```
logger.debug(new Exception("some error"));
```

Unfortunately, the above call did not print the stack trace of the exception. Thus, a potentially crucial piece of information could be lost. When the first parameter is restricted to be of type String, then only the method

```
debug(String msg, Throwable t);
```

can be used to log exceptions. Note that this method ensures that every logged exception is accompanied with a descriptive message.

Can I log an exception without an accompanying message?

In short, no.

If e is an Exception, and you would like to log an exception at the ERROR level, you must add an accompanying message. For example,

```
logger.error("some accompanying message", e);
```

You might legitimately argue that not all exceptions have a meaningful message to accompany them. Moreover, a good exception should already contain a self explanatory description. The accompanying message may therefore be considered redundant.

While these are valid arguments, there are three opposing arguments also worth considering. First, on many, albeit not all occasions, the accompanying message can convey useful information nicely complementing the description contained in the exception. Frequently, at the point where the exception is logged, the developer has access to more contextual information than at the point where the exception is thrown. Second, it is not difficult to imagine more or less generic messages, e.g. "Exception caught", "Exception follows", that can be used as the first argument for `error(String msg, Throwable t)` invocations. Third, most log output formats display the message on a line, followed by the exception on a separate line. Thus, the message line would look inconsistent without a message.

In short, if the user were allowed to log an exception without an accompanying message, it would be the job of the logging system to invent a message. This is actually what the throwing(String sourceClass, String sourceMethod, Throwable thrown) method in java.util.logging package does. (It decides on its own that accompanying message is the string "THROW".)

It may initially appear strange to require an accompanying message to log an exception. Nevertheless, this is common practice in *all* log4j derived systems such as java.util.logging, logkit, etc. and of course log4j itself. It seems that the current consensus considers requiring an accompanying message as a good a thing (TM).

What is the fastest way of (not) logging?

SLF4J supports an advanced feature called parameterized logging which can significantly boost logging performance for *disabled* logging statement.

For some Logger logger, writing,

```
logger.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
```

incurs the cost of constructing the message parameter, that is converting both integer `i` and `entry[i]` to a String, and concatenating intermediate strings. This, regardless of whether the message will be logged or not.

One possible way to avoid the cost of parameter construction is by surrounding the log statement with a test. Here is an example.

```
if(logger.isDebugEnabled()) {
  logger.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
}
```

This way you will not incur the cost of parameter construction if debugging is disabled for logger. On the other hand, if the logger is enabled for the DEBUG level, you will incur the cost of evaluating whether the logger is enabled or not, twice: once in debugEnabled and once in debug. This is an

insignificant overhead because evaluating a logger takes less than 1% of the time it takes to actually log a statement.

Better yet, use parameterized messages

There exists a very convenient alternative based on message formats. Assuming `entry` is an object, you can write:

```
Object entry = new SomeObject();
logger.debug("The entry is {}.", entry);
```

After evaluating whether to log or not, and only if the decision is affirmative, will the logger implementation format the message and replace the '{}' pair with the string value of `entry`. In other words, this form does not incur the cost of parameter construction in case the log statement is disabled.

The following two lines will yield the exact same output. However, the second form will outperform the first form by a factor of at least 30, in case of a *disabled* logging statement.

```
logger.debug("The new entry is "+entry+".");
logger.debug("The new entry is {}.", entry);
```

A two argument variant is also available. For example, you can write:

```
logger.debug("The new entry is {}. It replaces {}.", entry, oldEntry);
```

If three or more arguments need to be passed, you can make use of the `Object...` variant of the printing methods. For example, you can write:

```
logger.debug("Value {} was inserted between {} and {}.", newVal, below, above);
```

This form incurs the hidden cost of construction of an Object[] (object array) which is usually very small. The one and two argument variants do not incur this hidden cost and exist solely for this reason (efficiency). The slf4j-api would be smaller/cleaner with only the Object... variant.

Array type arguments, including multi-dimensional arrays, are also supported.

SLF4J uses its own message formatting implementation which differs from that of the Java platform. This is justified by the fact that SLF4J's implementation performs about 10 times faster but at the cost of being non-standard and less flexible.

Escaping the "{}" pair

The "{}" pair is called the *formatting anchor*. It serves to designate the location where arguments need to be substituted within the message pattern.

SLF4J only cares about the *formatting anchor*, that is the '{' character immediately followed by '}'. Thus, in case your message contains the '{' or the '}' character, you do not have to do anything special unless the '}' character immediately follows '}'. For example,

```
logger.debug("Set {1,2} differs from {}", "3");
```

which will print as "Set {1,2} differs from 3".

You could have even written,

```
logger.debug("Set {1,2} differs from {{}}", "3");
```

which would have printed as "Set {1,2} differs from {3}".

In the extremely rare case where the the "{}" pair occurs naturally within your text and you wish to disable the special meaning of the formatting anchor, then you need to escape the '{' character with '\', that is the backslash character. Only the '{' character should be escaped. There is no need to escape the '}' character. For example,

```
logger.debug("Set \\{} differs from {}", "3");
```

will print as "Set {} differs from 3". Note that within Java code, the backslash character needs to be written as '\\'.

In the rare case where the "\{}" occurs naturally in the message, you can double escape the formatting anchor so that it retains its original meaning. For example,

```
logger.debug("File name is C:\\\\{}.", "file.zip");
```

will print as "File name is C:\file.zip".

How can I log the string contents of a single (possibly complex) object?

In relatively rare cases where the message to be logged is the string form of an object, then the parameterized printing method of the appropriate level can be used. Assuming `complexObject` is an object of certain complexity, for a log statement of level DEBUG, you can write:

```
logger.debug("{}", complexObject);
```

The logging system will invoke `complexObject.toString()` method only after it has ascertained that the log statement was enabled. Otherwise, the cost of `complexObject.toString()` conversion will be advantageously avoided.

Why doesn't the `org.slf4j.Logger` interface have methods for the FATAL level?

The Marker interface, part of the `org.slf4j` package, renders the FATAL level largely redundant. If a given error requires attention beyond that allocated for ordinary errors, simply mark the logging statement with a specially designated marker which can be named "FATAL" or any other name to your liking.

Here is an example,

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.Marker;
import org.slf4j.MarkerFactory;

class Bar {
  void foo() {
    Marker fatal = MarkerFactory.getMarker("FATAL");
    Logger logger = LoggerFactory.getLogger("aLogger");

    try {
      ... obtain a JDBC connection
    } catch (JDBException e) {
      logger.error(fatal, "Failed to obtain JDBC connection", e);
    }
  }
}
```

While markers are part of the SLF4J API, only logback supports markers off the shelf. For example, if you add the `%marker` conversion word to its pattern, logback's `PatternLayout` will add marker data to its output. Marker data can be used to filter messages or even trigger an outgoing email at the end of an individual transaction.

In combination with logging frameworks such as log4j and java.util.logging which do not support markers, marker data will be silently ignored.

Markers add a new dimension with infinite possible values for processing log statements compared to five values, namely ERROR, WARN, INFO, DEBUG and TRACE, allowed by levels. At present time, only logback supports marker data. However, nothing prevents other logging frameworks from making use of marker data.

Why was the TRACE level introduced only in SLF4J version 1.4.0?

The addition of the TRACE level has been frequently and hotly debated request. By studying various projects, we observed that the TRACE level was used to disable logging output from certain classes *without* needing to configure logging for those classes. Indeed, the TRACE level is by default disabled in log4j and logback as well most other logging systems. The same result can be achieved by adding the appropriate directives in configuration files.

Thus, in many of cases the TRACE level carried the same semantic meaning as DEBUG. In such

cases, the TRACE level merely saves a few configuration directives. In other, more interesting occasions, where TRACE carries a different meaning than DEBUG, Marker objects can be put to use to convey the desired meaning. However, if you can't be bothered with markers and wish to use a logging level lower than DEBUG, the TRACE level can get the job done.

Note that while the cost of evaluating a disabled log request is in the order of a few nanoseconds, the use of the TRACE level (or any other level for that matter) is discouraged in tight loops where the log request might be evaluated millions of times. If the log request is enabled, then it will overwhelm the target destination with massive output. If the request is disabled, it will waste resources.

In short, although we still discourage the use of the TRACE level because alternatives exist or because in many cases log requests of level TRACE are wasteful, given that people kept asking for it, we decided to bow to popular demand.

Does the SLF4J logging API support I18N (internationalization)?

Yes, as of version 1.5.9, SLF4J ships with a package called `org.slf4j.cal10n` which adds localized/internationalized logging support as a thin layer built upon the CAL10N API.

Is it possible to retrieve loggers without going through the static methods in LoggerFactory?

Yes. `LoggerFactory` is essentially a wrapper around an `ILoggerFactory` instance. The `ILoggerFactory` instance in use is determined according to the static binding conventions of the SLF4J framework. See the `getSingleton()` method in `LoggerFactory` for details.

However, nothing prevents you from using your own `ILoggerFactory` instance. Note that you can also obtain a reference to the `ILoggerFactory` that the `LoggerFactory` class is using by invoking the `LoggerFactory.getILoggerFactory()` method.

Thus, if SLF4J binding conventions do not fit your needs, or if you need additional flexibility, then do consider using the `ILoggerFactory` interface as an alternative to inventing your own logging API.

In the presence of an exception/throwable, is it possible to parameterize a logging statement?

Yes, as of SLF4J 1.6.0, but not in previous versions. The SLF4J API supports parametrization in the presence of an exception, assuming the exception is the last parameter. Thus,

```
String s = "Hello world";
try {
  Integer i = Integer.valueOf(s);
} catch (NumberFormatException e) {
  logger.error("Failed to format {}", s, e);
}
```

will print the `NumberFormatException` with its stack trace as expected. The java compiler will invoke the error method taking a String and two Object arguments. SLF4J, in accordance with the programmer's most probable intention, will interpret `NumberFormatException` instance as a throwable instead of an unused `Object` parameter. In SLF4J versions prior to 1.6.0, the `NumberFormatException` instance was simply ignored.

If the exception is not the last argument, it will be treated as a plain object and its stack trace will NOT be printed. However, such situations should not occur in practice.

Implementing the SLF4J API

How do I make my logging framework SLF4J compatible?

Adding supporting for the SLF4J is surprisingly easy. Essentially, you coping an existing binding and tailoring it a little (as explained below) does the trick.

Assuming your logging system has notion of a logger, called say `MyLogger`, you need to provide an adapter for `MyLogger` to `org.slf4j.Logger` interface. Refer to slf4j-jcl, slf4j-jdk14, and slf4j-log4j12

modules for examples of adapters.

Once you have written an appropriate adapter, say `MyLoggerAdapter`, you need to provide a factory class implementing the `org.slf4j.ILoggerFactory` interface. This factory should return instances `MyLoggerAdapter`. Let `MyLoggerFactory` be the name of your factory class.

Once you have the adapter, namely `MyLoggerAdapter`, and a factory, namely `MyLoggerFactory`, the last remaining step is to modify the `StaticLoggerBinder` class so that it returns an new instance of `MyLoggerFactory`. You will also need to modify the `loggerFactoryClassStr` variable.

For Marker or MDC support, you could use the one of the existing NOP implementations.

In summary, to create an SLF4J binding for your logging system, follow these steps:

1. start with a copy of an existing module,
2. create an adapter between your logging system and `org.slf4j.Logger` interface
3. create a factory for the adapter created in the previous step,
4. modify `StaticLoggerBinder` class to use the factory you created in the previous step

---

How can my logging system add support for the `Marker` interface?

Markers constitute a revolutionary concept which is supported by logback but not other existing logging systems. Consequently, SLF4J conforming logging systems are allowed to ignore marker data passed by the user.

However, even though marker data may be ignored, the user must still be allowed to specify marker data. Otherwise, users would not be able to switch between logging systems that support markers and those that do not.

The `MarkerIgnoringBase` class can serve as a base for adapters or native implementations of logging systems lacking marker support. In `MarkerIgnoringBase`, methods taking marker data simply invoke the corresponding method without the Marker argument, discarding any Marker data passed as argument. Your SLF4J adapters can extend `MarkerIgnoringBase` to quickly implement the methods in `org.slf4j.Logger` which take a `Marker` as the first argument.

How does SLF4J's version check mechanism work?

The version check performed by SLF4J API during its initialization is an *elective* process. Conforming SLF4J implementations may choose *not* to participate, in which case, no version check will be performed.

However, if an SLF4J implementation decides to participate, than it needs to declare a variable called REQUESTED_API_VERSION within its copy of the `StaticLoggerBinder` class. The value of this variable should be equal to the version of the slf4j-api.jar it is compiled with. If the implementation is upgraded to a newer version of slf4j-api, than you also need to update the value of REQUESTED_API_VERSION.

For each version, SLF4J API maintains a list of compatible versions. SLF4J will emit a version mismatch warning only if the requested version is not found in the compatibility list. So even if your SLF4J binding has a different release schedule than SLF4J, assuming you update the SLF4J version you use every 6 to 12 months, you can still participate in the version check without incurring a mismatch warning. For example, logback has a different release schedule but still participates in version checks.

As of SLF4J 1.5.5, all bindings shipped within the SLF4J distribution, e.g. slf4j-log4j12, slf4j-simple and slf4j-jdk14, declare the REQUESTED_API_VERSION field with a value equal to their SLF4J version. It follows that, for example if slf4j-simple-1.5.8.jar is mixed with slf4j-api-1.6.0.jar, given that 1.5.8 is not on the compatibility list of SLF4J version 1.6.x, a version mismatch warning will be issued.

Note that SLF4J versions prior to 1.5.5 did not have a version check mechanism. Only slf4j-api-1.5.5.jar and later can emit version mismatch warnings.

General questions about logging

Should Logger members of a class be declared as static?

We used to recommend that loggers members be declared as instance variables instead of static. After further analysis, **we no longer recommend one approach over the other.**

Here is a summary of the pros and cons of each approach.

Advantages for declaring loggers as static	Disadvantages for declaring loggers as static
1. common and well-established idiom 2. less CPU overhead: loggers are retrieved and assigned only once, at hosting class initialization 3. less memory overhead: logger declaration will consume one reference per class	1. For libraries shared between applications, not possible to take advantage of repository selectors. It should be noted that if the SLF4J binding and the underlying API ships with each application (not shared between applications), then each application will still have its own logging environment. 2. not IOC-friendly
Advantages for declaring loggers as instance variables	**Disadvantages for declaring loggers as instance variables**
1. Possible to take advantage of repository selectors even for libraries shared between applications. However, repository selectors only work if the underlying logging system is logback-classic. Repository selectors do not work for the SLF4J+log4j combination. 2. IOC-friendly	1. Less common idiom than declaring loggers as static variables 2. higher CPU overhead: loggers are retrieved and assigned for each instance of the hosting class 3. higher memory overhead: logger declaration will consume one reference per instance of the hosting class

Explanation

Static logger members cost a single variable reference for all instances of the class whereas an instance logger member will cost a variable reference for every instance of the class. For simple classes instantiated thousands of times there might be a noticeable difference.

However, more recent logging systems, e.g log4j or logback, support a distinct logger context for each application running in the application server. Thus, even if a single copy of *log4j.jar* or *logback-classic.jar* is deployed in the server, the logging system will be able to differentiate between applications and offer a distinct logging environment for each application.

More specifically, each time a logger is retrieved by invoking `LoggerFactory.getLogger()` method, the underlying logging system will return an instance appropriate for the current application. Please note that within the *same* application retrieving a logger by a given name will always return the same logger. For a given name, a different logger will be returned only for different applications.

If the logger is static, then it will only be retrieved once when the hosting class is loaded into memory. If the hosting class is used in only in one application, there is not much to be concerned about. However, if the hosting class is shared between several applications, then all instances of the shared class will log into the context of the application which happened to first load the shared class into memory - hardly the behavior expected by the user.

Unfortunately, for non-native implementations of the SLF4J API, namely with slf4j-log4j12, log4j's repository selector will not be able to do its job properly because slf4j-log4j12, a non-native SLF4J binding, will store logger instances in a map, short-circuiting context-dependent logger retrieval. For native SLF4J implementations, such as logback-classic, repository selectors will work as expected.

The Apache Commons wiki contains an informative article covering the same question.

Logger serialization

Contrary to static variables, instance variables are serialized by default. As of SLF4J version 1.5.3, logger instances survive serialization. Thus, serialization of the host class no longer requires any special action, even when loggers are declared as instance variables. In previous versions, logger instances needed to be declared as `transient` in the host class.

Summary

In summary, declaring logger members as static variables requires less CPU time and have a slightly smaller memory footprint. On the other hand, declaring logger members as instance variables requires more CPU time and have a slightly higher memory overhead. However, instance variables make it possible to create a distinct logger environment for each application, even for loggers declared in shared libraries. Perhaps more important than previously mentioned considerations, instance variables are IOC-friendly whereas static variables are not.

See also related discussion in the commons-logging wiki.

Is there a recommended idiom for declaring a logger in a class?

The following is the recommended logger declaration idiom. For reasons explained above, it is left to the user to determine whether loggers are declared as static variables or not.

```
package some.package;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyClass {
  final (static) Logger logger = LoggerFactory.getLogger(MyClass.class);
  ... etc
}
```

Unfortunately, give that the name of the hosting class is part of the logger declaration, the above logger declaration idiom is not is *not* resistant to cut-and-pasting between classes.