The ObjectWeb Consortium

Interface Specification

MonoLog
-
Logging and Monitoring Specifications

AUTHORS:

S. Chassande-Barrioz (INRIA)

CONTRIBUTORS:

JB. Stefani (INRIA)

B. Dumant (Kelua)

Released: March 26, 2002

Status: Final Release

Version: 1.0

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

A product must contain a log system in order to provide debug or log information at runtime. The source code is instrumented by log actions. Java log (JSR 0047) and Log4j are two standard specifications of logging. Both products contain only class specifications without interfaces.

1.1 Goals

The MonoLog specification has been designed with the following goals:

- to standardize the instrumentation code,
- to support a component architecture,
- to allow an efficient implementation of logging,
- to abstract source code instrumentation from a specific implementation of logging,
- to support internationalisation,
- to support monitoring.

1.2 Target audience

This specification concerns all ObjectWeb developers, but also all developers who do not want to depend on a logging implementation. The use of this specification also permits to provide additional functionalities between the application and the log system.

1.3 Overview

The document describes a set of interfaces for the source code instrumentation and a convention to instrument the source code. The full javadoc for Monolog is available at the following URL: www.objectweb.org/archi/log/.

This specification does not contain a complete solution to the log system configuration problem. At this step, our main objective is not to abstract the configuration from an implementation, but only to abstract the source code instrumentation from an implementation. Nevertheless, as it is impossible to talk about instrumentation without considering the start of the log system, the configuration remains specific to an implementation or platform.

1.4 Document Convention

Description of MonoLog: Times New Roman:12

Example or source code: `Courier New:10`

2 LOGGING API OVERVIEW

2.1 Architecture

2.1.1 Overview of Control flow

Applications make logging calls on objects which implement the Logger interface. The Logger interface provides methods to publish events:

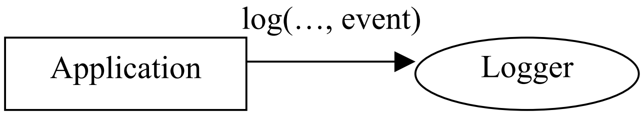

Figure 1 : Event logging on a Logger

As an anonymous Logger is not enough to have a complete log system, the TopicalLogger interface is defined as an extension of a Logger. A TopicalLogger is a named Logger.

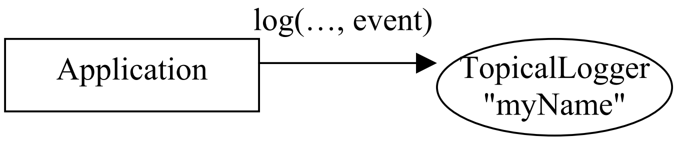

Figure 2 : Event logging on a TopicalLogger

A TopicalLogger can also route events to Handlers:

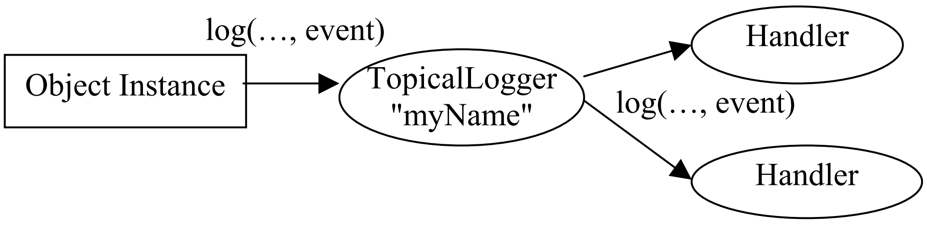

Figure 3 : Event Routing

The Handler interface is empty and is used as a tag to represent an output. As the Logger interface extends the Handler interface and the TopicalLogger interface extends the Logger interface, a TopicalLogger can also route events to other Loggers or other TopicalLoggers.

2.1.2 Granularity of logging

It appears desirable to support granularity of logging in two different ways, first by topic and second by priority or level.

To each logger is associated a topic or a set of topics. A topic is often the identifier of a sub-system. A topic could be a simple string or a complex entity.

Among log events of a sub-system, there are many sorts of events. Some events are very important and others are just used for debugging purposes. It appears desirable to allow logging to be enabled according to a limited number of predefined system levels, so that a program can be configured to output logging for some levels while ignoring output for other levels.

2.2 Level interface & BasicLevel class

2.2.1 Level

An event must be registered with a level which represents its importance. This specification provides a Level interface to represent this concept. Levels can be ordered, and enabling logging at a given level also enables logging at all higher levels.

An object which implements the Level interface can be used in log methods to specify the event importance. This interface provides a method to determine if two Level instances are comparable (ie ordered), a method to compare Level instances, and a method to get an integer value representing the level.

```
… // Level interface definition

      boolean isComparableWith(Level level);

      int compareTo(Level level);

      int getIntValue();
…
```

This specification also suggests applying the flyweight pattern to this Level interface. A Level can be represented by a simple integer value. To apply the pattern, all log methods are duplicated to take into consideration an integer parameter instead of a Level parameter. The Level interface is defined in the org.objectweb.util.monolog.api package.

2.2.2 BasicLevel

To be complete, this specification must predefine a set of levels. Indeed during the instrumentation, developers must write calls to log an event with a specific level. One of the goals of this specification is to be independent of the logging implementation. To respect this goal, predefined variables or constants are needed. This specification provides the org.objectweb.util.monolog.api.BasicLevel class which contains only static but not final variables to represent these predefined levels.

To respect the flyweight pattern chosen before, for each predefined level, a Level variable and an integer variable are declared.

Their values are not defined, in order to leave it to the implementation to set them. The following table describes the meaning of the five predefined levels. The declaration of the BasicLevel class is just after the array.

Level name	Details
FATAL	In general, FATAL messages should describe events that are of considerable importance and which will prevent continuation of the program execution. They should be intelligible to end users and to system administrators.
ERROR	The ERROR level designates error events that might still allow the application to continue running.
WARN	In general, WARN messages should describe events that will be of interest to end users or system managers, or which indicate potential problems.
INFO	The INFO level designates informational messages that highlight the progress of the application at a coarse-grained level.
DEBUG	DEBUG messages might include things like minor (recoverable) failures. Logging calls for entering, returning, or throwing an exception can be traced at this level.

```
… // BasicLevel interface definition
      public static Level LEVEL_FATAL ;
      public static int FATAL ;

      public static Level LEVEL_ERROR ;
      public static int ERROR ;

      public static Level LEVEL_WARN ;
      public static int WARN ;

      public static Level LEVEL_INFO ;
      public static int INFO ;

      public static Level LEVEL_DEBUG ;
      public static int DEBUG ;
```

Figure 4 : Predefined levels

2.2.3 Level extension

This specification defines five basic levels. But it is possible to a MonoLog user to define additional levels. This specification allows this type of extension with some constraints or advices:

- If levels are not ordered, the additional levels must be defined by an implementation of the Level interface.
- If levels are ordered, all additional level must have an integer value between the FATAL level and the DEBUG level.
- If levels are ordered, it is possible to define a level with relative integer value with an existent level (MyLevel.FINE = BasicLevel.DEBUG + 2). This possibility imposes that all MonoLog implementations define a set of sparse integer values for the levels.

2.3 Handler

Handler is an empty interface that represents an output. For example a handler might be a console, a file, a socket, or a Logger.

It is not necessary to specify methods in the Handler interface. Indeed methods that could be defined in this interface will concern only the configuration aspect. This version of the specification only describes the instrumentation aspect. At the moment these methods are specific to an implementation, and only used by the implementation of the log system. The application does not need to know the Handler interface during the instrumentation step.

2.4 Logger

A Logger implementation receives event messages from an object and exports them. Each Logger is associated with a log level and discards log requests that are below this level. Furthermore the Logger interface extends the Handler interface and represents therefore a type of output.

log Methods

These methods are used to log an event at a specific level. The type of the event parameter is java.lang.Object, in order not to impose a specific type. When the log method is called, the Logger checks if the event parameter is loggable at the specified level. This interface provides log methods with many parameters. The list of log methods is the following:

```
… // Logger interface definition
void log(Level l, Object evt);
void log(int   l, Object evt);
void log(Level l, Object evt, Throwable t);
void log(int   l, Object evt, Throwable t);
void log(Level l, Object evt, Object location, Object method);
void log(int   l, Object evt, Object location, Object method);
void log(Level l, Object evt, Throwable t, Object location, Object method);
void log(int   l, Object evt, Throwable t, Object location, Object method);
…
```

All these methods are duplicated with an int parameter instead of Level. The "evt" parameter is usually a textual representation of the event. Three methods have been added to the basic methods in order to simplify the instrumentation.

In some cases, it appears to be desirable to log an exception. Giving the exception instance as parameter can be interesting to unify the format of output.

Furthermore, in order to localize better an event, it is interesting to give as parameters two objects which represent the object and the method of the object where the event has occurred. A reference to the current object could be used as location parameter.

If it is not necessary to specify the instance, but only the class name, two solutions are possible:

- either the class name could be used as location parameter,
- or the implementation knows how to find it in the execution context. Indeed it is possible with the java.lang.Throwable class to obtain a string which represents the calls stack. It is then easy to extract the object class name and the method name during the log action.

Level management

```
… // Logger interface definition
      void setIntLevel(int level);
      void setLevel(Level l);

      int getCurrentIntLevel();
      Level getCurrentLevel();
…
```

These methods are accessors to the current logger level. These methods are duplicated to apply the flyweight pattern and to allow using either an integer value or a Level implementation.

Activation of Logger

```
… // Logger interface definition
      void turnOn();
      void turnoff();
      boolean isOn();
…
```

A Logger can be enabled or disabled by these methods. Even if the logger is disabled, the accessors to the level information return the right value.

Level check

```
… // Logger interface definition
      boolean isLoggable(int level);
      boolean isLoggable(Level l);
…
```

These methods allow checking if a level parameter is loggable by the current Logger. The level parameter is compared to the current level. These methods return always false if the logger is disabled.

2.4.1 How to find a Logger ?

Two possibilities to obtain a Logger are considered:

- either find a LoggerFactory, and call a "getLogger" method,
- or initialise a component with a Logger. This allows specifying the Logger that a given component must use.

2.5 TopicalLogger

2.5.1 Definition & properties

A TopicalLogger is a Logger extension with the following properties:

- A TopicalLogger dispatches events to a set of Handlers. A TopicalLogger is a sort of message router.

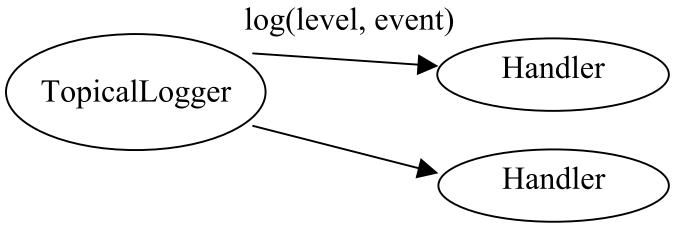

Figure 5 : TopicalLogger and its Handlers

- A topic is associated with each TopicalLogger. A topic is represented by a dotted string, which is used to build a hierarchical namespace. The latter should typically be aligned with the Java packaging namespace.

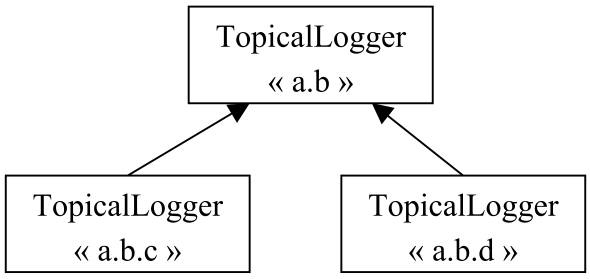

Figure 6 : Hierarchical namespace

- The name hierarchy of TopicalLogger allows adding properties inheritance. For example, a TopicalLogger with the “a.b.c” name can inherit of the Handlers list and the level from the “a.b” parent (see an example in 2.5.3 section).
- Another property for a TopicalLogger is the capacity to have several topics. This is important when a component is used by several other components. This will allow events logged by the shared component to appear for each component using this shared component. A consequence of this property is that a Logger may have several parents.

2.5.2 The use

The use of the TopicalLogger interface is not the same as that of the Logger interface. The provided methods are used during the configuration of the log system. This configuration is done during the start of the log system, but can also be done dynamically. The configuration actions on a TopicalLogger are the change of the hierarchical namespace or the Handler list. The TopicalLogger interface also provides the inherited methods of the Logger interface.

2.5.3 TopicalLogger API

Handlers list management

```
… // TopicalLogger interface definition
      boolean addHandler(Handler h);
      Handler removeHandler(Handler h);
...
```

A TopicalLogger manages a list of Handler instances. These methods allow adding or removing a handler from this list. The addHandler method returns true only if the Handler did not exist. The removeHandler method returns the removed handler or null if did not exist.

Topics management

```
… // TopicalLogger interface definition
      boolean addTopic(String topic);
      Enumeration getTopics();
      String removeTopic(String topic);
...
```

These methods allow adding or removing a topic to/from a TopicalLogger. These actions change the hierarchical structure, but also the list of handlers. The list of handlers of a TopicalLogger is composed of its handlers and all handlers inherited from its parents. Adding or removing a topic changes the inherited handlers list.

Example:

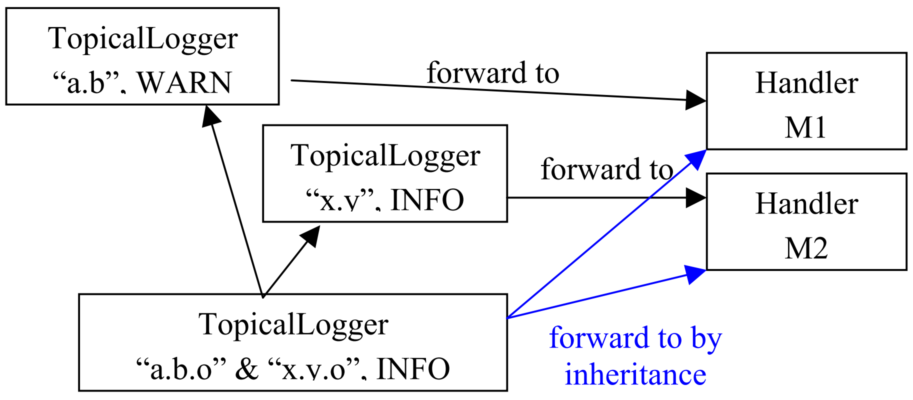

Figure 7 : Multiple names

This example presents a TopicalLogger "o" with two topics: 'a.b.o' and 'x.y.o'. Each TopicalLogger parent has a handler. The children inherit handlers from their parents. The children level is the lowest of their parents.

If the 'a.b.o' topic is removed, then the TopicalLogger children will not forward events to the M1 Handler.

2.6 LoggerFactory

2.6.1 Goals

- To provide Logger instances
- To allow instrumentation to be independent of the logging implementation.
- To allow the use of the same logging implementation for all the components of a given application.

2.6.2 Api

The org.objectweb.util.monolog.api.LoggerFactory interface provides two methods to fetch Logger. If the Logger described by the parameters does not exist, then the LoggerFactory must return a new instance of Logger. The list of methods is the following:

```
… // LoggerFactory interface definition
      Logger getLogger(String key) ;
      Logger getLogger(String key, String resourceBundleName) ;
      void setResourceBundleName(String resourceBundleName) ;
      String getResourceBundleName() ;
```

The key parameter is a description of the expected Logger. In simple cases, the key is the initial topic of the Logger.

The second log method allows specifying the name of a resource bundle in order to internationalise the logging. This option is useful when a resource bundle needs to be specified by component.

The LoggerFactory interface also provides accessors to a resource bundle name associated to a LoggerFactory. This interface allows defining a resource bundle name used by all Logger.

2.6.3 How to find a LoggerFactory ?

In a component-based approach, there are two ways to fetch a LoggerFactory:

- Either among initialisation parameters there is a LoggerFactory reference registered with a well-known name
- Or the component lookup up the naming service the well-known name.

In simpler architecture types, this specification suggests to keep a static reference within an intermediate class. This is a sort of LoggerFactory manager. This specification presents an example of LoggerFactory Manager which provides only two static methods to set or get the unique LoggerFactory instance:

```
package org.Objectweb.util.monolog.lib;
public class DefaultLoggerFactoryManager {
      private static LoggerFactory factory = null;
      public static LoggerFactory getLoggerFactory() {
            return factory;
      }
      public static void setLoggerFactory(LoggerFactory lf) {
            factory= lf;
      }
}
```

2.7 Architecture: Summary

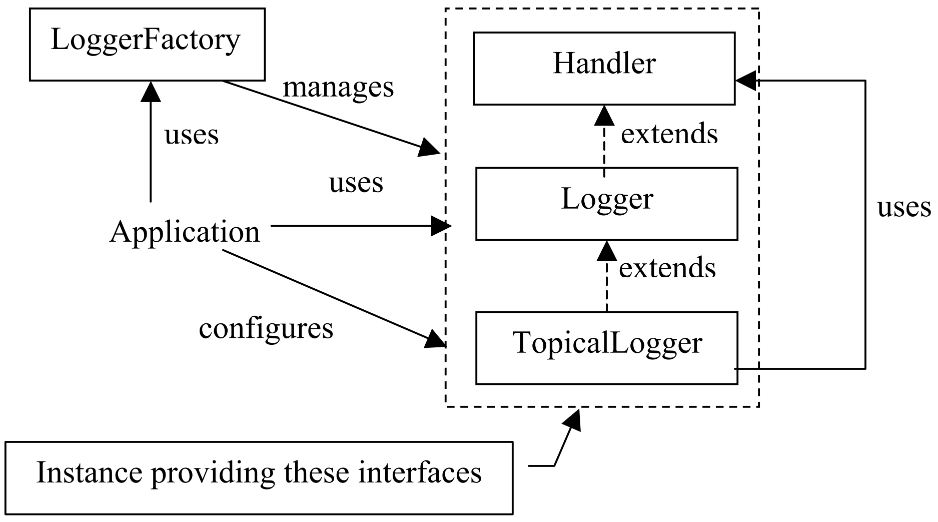

Figure 8: Interactions

This schema shows all interactions between an application and the MonoLog interfaces. The application interacts with the LoggerFactory to obtain a Logger reference. With this reference the application can log events. This scenario concerns the instrumentation aspect.

In most cases, the Logger instance implements the TopicalLogger interface too and can be cast to the TopicalLogger type. The TopicalLogger interface allows configuring the instance. The application or an administrator can configure the instance.

2.7.1 Relation between interfaces and aspects:

This following array shows the relations between interfaces and aspects. Each interface is only linked to one aspect.

	Logger	TopicalLogger	Handler	LoggerFactory	Level
Instrumentation	X			X	[X]
Configuration		X	X		[X]

3 INSTRUMENTATION CONVENTION

This part of the specification gives an example of source code instrumentation.

3.1 Header & declaration

To log event is necessary to fetch a Logger implementation from a LoggerFactory. As explained in section 2.5, a component must first fetch a LoggerFactory. The example below shows the use of DefaultLoggerFactoryManager, and the needed imports.

```
import org.objectweb.util.monolog.api.Logger;
import org.objectweb.util.monolog.api.LoggerFactory;
import org.objectweb.util.monolog.lib.DefaultLoggerFactoryManager;

static Logger logger = DefaultLoggerFactoryManager.
            getLoggerFactory().getLogger("org.ow.toto");
```

A static final variable 'trace' can be also defined. This variable is not necessary but allows withdrawing the logging source code at the compilation time.

```
static final boolean trace = true; //or false
```

3.2 Logging an event

3.2.1 Pre checking

To log an event, there are several alternatives:

- The simple way is to call the log method with the level and the message:

  ```
  logger.log(BasicLevel.DEBUG, ...);
  ```

- It is possible to prefix the call by a test of the Logger level. This allows avoiding to build the message if it is not necessary (This is useful as a message is often a complex argument built with many objects).

  ```
  if (logger.isLoggable(BasicLevel.DEBUG) )
        logger.log(BasicLevel.DEBUG, ...);
  ```

- Another possibility allows withdrawing logging code at compile time, by adding the test of the static constant 'trace' to the previous example:

  ```
  if ( trace && logger.isLoggable(BasicLevel.DEBUG) )
        logger.log(BasicLevel.DEBUG, ...);
  ```

3.2.2 Parameters of log methods

The Logger provides several methods to log an event, each one corresponding to a use case as described here:

- log(level, message): This is the basic log method. The log implementation can find the context via methods of the java.lang.Throwable class.
- log(level, message, throwable): This methods permits to give a message and a Throwable which represents the context. The throwable parameter can be used to give an Exception.
- log(level, message, location, method): This method permits to specify an instance and a method. The location parameter is the instance, and the method parameter can be the name of the method or the object java.lang.reflect.Method.
- log(level, message, throwable, location, method): this method permits to log a error and to specify the instance where the error occurred.

4 IMPLEMENTATIONS & WRAPPERS

4.1 Log4j Wrapper

The proposed levels are the same as the log4j Priorities. The log4j wrapper has to manage the multiple topic property, and provide a LoggerFactory implementation.

Monolog types	Java Log types
Handler	Appender
Logger	Category
TopicalLogger	
LoggerFactory	CategoryFactory
Level	Priority

Figure 9: Type mapping between Monolog and log4j

4.2 JavaLog Wrapper

The mapping of MonoLog level to Java Log Level is the following:

MonoLog levels	Java Log Priorities
FATAL	SEVERE
ERROR	SEVERE
WARN	WARNING
INFO	INFO
DEBUG	FINEST

Figure 10: Level mapping between Monolog and Java log

For example, when logging an event with the ERROR Level, Java Log will receive a log call with SEVERE Level.

In addition of the level conversion, the Java Log wrapper has to implement the multiple topic property, and provide a LoggerFactory implementation.

Monolog types	Java Log types
Handler	Handler
Logger	Logger
TopicalLogger	
LoggerFactory	LogManager
Level	Level

Figure 11: Type mapping between Monolog and Java Log