Java Logging APIs

Abstract

This is the Public Review draft of the Java Logging APIs proposed in JSR-047.

There are change bars relative to the Community Review draft (0.50).

There is a change history on page 79.

DRAFT 0.55

Editor: Graham Hamilton (kgh@sun.com)

October 18, 2000

1 Requirements

This section outlines the main requirements and goals for the Java Logging APIs.

1.1 Target Users

The central goal of the logging APIs is to support maintaining and servicing software at customer sites. There are four main target uses of the logs:

- Problem diagnosis by end users and system administrators.
- Problem diagnosis by field service engineers.
- Returning detailed information for diagnosis by the development team.
- Problem diagnosis by developers

These four uses have somewhat different requirements, but there is considerable overlap.

1.1.1 End Users and System Administrators

For end users and system administrators it is desirable to provide simple logging of common problems that can be fixed or tracked locally. This includes such things as running out of resources, security failures, and simple configuration errors. In general, this level of logging should be very easy to use, and the logs should be self explanatory.

Full internationalization of this logging information is required.

For system management purposes it is highly desirable that Java logging output can be (a) directed to existing logging services and (b) forwarded to network system management services.

1.1.2 Diagnosis by Field Service Engineers

Field service engineers will use the logs to help diagnose:

- Configuration errors
- Performance bottlenecks
- Bugs in the application or in the platform

The logging information used by service engineers may be considerably more complex and considerably more verbose than that used by system administrators. It must be possible to enable extra logging in particular subsystems and to enable subsystem specific logging options.

1.1.3 Diagnosis by the development organization

When a problem occurs in the field it may be necessary to return information to the original development organization for diagnosis.

This logging information may be extremely detailed and fairly inscrutable. It may include detailed tracing information on the internal execution of particular subsystems.

It must be easy for the system administrator or field service engineer to enable appropriate logging and to capture the log output. Typically it will be desirable to have fairly fine grain control over exactly what is logged and on the level of logging information that is produced.

Typically such logging information will not require internationalization.

1.1.4 Use by developers

Logging may be used to help debug an application under development. This may include logging information generated by the target application and also logging information generated by lower level libraries.

This usage is perfectly reasonable. However it it is definitely not a goal of the logging APIs to replace the use of normal debugging and profiling tools in the development environment.

1.2 Configuration requirements

There needs to be both static and dynamic control over logging.

1.2.1 Static Control

Static control should allow field service staff to setup a particular configuration and then relaunch the application with the new logging settings.

It should be easy for system administrators or end-users to configure the logging system. The default configuration should make only modest use of system resources.

Several programs may be sharing one copy of the JRE. It may be desirable to be able to have different logging configuration options for different programs, or to provide special options for a particular run of a given program.

1.2.2 Dynamic Control

It is necessary to allow dynamic updates to the logging configuration within a running program. For long lived services it may be desirable to make minor adjustments to logging without having to restart the service. Additionally, some programs way wish to programatically modify their logging configuration as they run.

1.3 Granularity of logging

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

1.3.1 Functional areas

A field service engineer might be interested in tracing all events in AWT, but have no interest in socket events or memory management.

It is therefore desirable to allow logging to be enabled or disabled for different functional areas of the system.

1.3.2 Logging Levels

Logging is relatively expensive. We want to allow fine grain logging, but in order to preserve system performance we will normally want to disable most logging.

It appears desirable to allow logging to be enabled based on a system of levels, so that a program can be configured to output logging for some levels while ignoring output at other levels.

1.3.3 Filters

In addition to having logging levels, it is also desirable to support a more general filter mechanism so that application code can attach arbitrary filters to control logging output.

1.4 Pluggable Handlers

It should be possible to direct logging information to a variety of back-end log Handlers.

Some back-end Handlers will be delivered as part of this JSR (see the javadoc below). Other Handlers may be developed by third parties and delivered on top of the core platform.

For example, an application server vendor may develop a Java logging service that forwards Java logging requests into a vendor specific logging mechanism.

1.5 Internationalization

The Logging Framework should support easy internationalization of log messages.

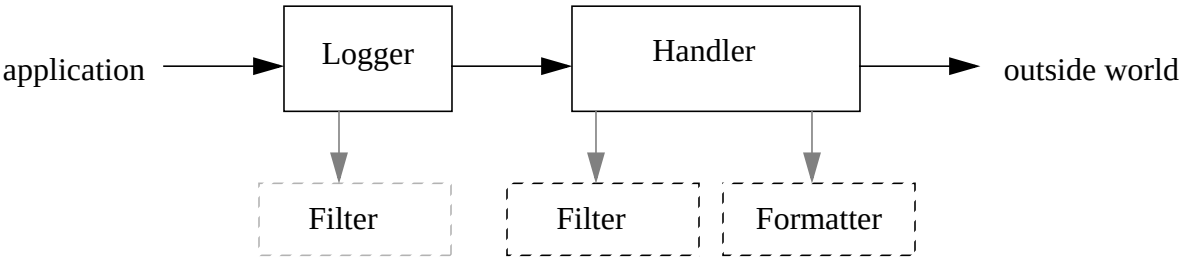
The support for internationalization should be based on the normal internationalization support in the Java platform.

2 Logging API Overview

The logging APIs are described in detail in javadoc later in this note. This section provides an overview of key elements.

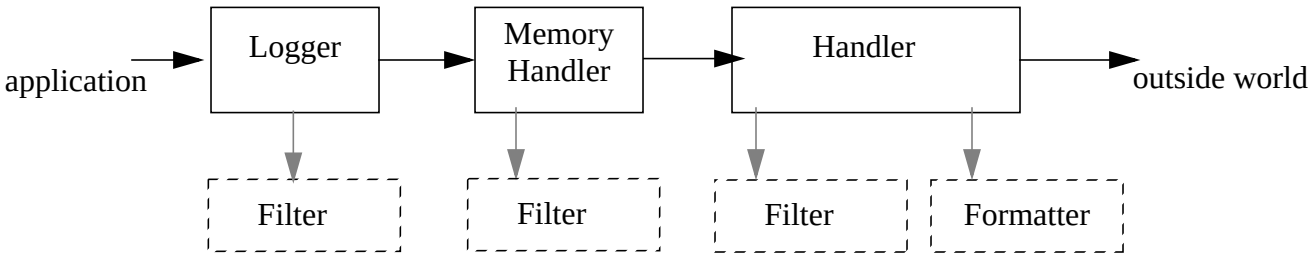
2.1 Overview of Control Flow

Applications make logging calls on *Logger* objects. These Logger objects allocate *LogRecord* objects which are passed to *Handler* objects for publication. Both Loggers and Handlers may use logging *Levels* and (optionally) *Filters* to decide if they are interested in a particular LogRecord. When it is necessary to publish a LogRecord externally, a Handler can (optionally) use a *Formatter* to localize and format the message, before publishing it to an I/O stream.



The LogManager class keeps track of a set of global Handlers. By default all Loggers send their output to this standard set of global Handlers. But Loggers may also be configured to ignore the global Handler list and/or to send output to specific target Handlers.

Some Handlers may direct output to other Handlers. For example, the *MemoryHandler* maintains an internal ring buffer of LogRecords and on trigger events it publishes its LogRecords through a target Handler. In such cases, any formatting is done by the last Handler in the chain.



The APIs are structured so that calls on the Logger APIs can be cheap when logging is disabled. If logging is disabled for a given log level, then the Logger can make a cheap comparison test and return. If logging is enabled for a given log level, the Logger is still careful to minimize costs before passing the LogRecord into the Handlers. In particular, localization and formatting (which are relatively expensive) are deferred until the Handler requests them. For example, a

MemoryHandler can maintain a circular buffer of LogRecords without having to pay formatting costs.

2.2 Log Levels

Each log message has an associated log *Level*. The Level gives a rough guide to the importance and urgency of a log message. Log level objects encapsulate an integer value, with higher values indicating higher priorities.

Seven standard log levels are defined in the Level class. They range from FINEST (the lowest priority, with the lowest value) to SEVERE (the highest priority, with the highest value).

2.3 Loggers

Client code sends log request to *Logger* objects. Each logger keeps track of a log level that it is interested in and discards log requests that are below this level.

Loggers are normally named entities, using dot separated names such as “java.awt”. The namespace is hierarchical and is managed by the LogManager. The namespace should typically be aligned with the Java packaging namespace, but is not required to follow it slavishly. For example, a Logger called “java.awt” might handle logging requests for classes in the java.awt package, but it might also handle logging for classes in sun.awt that support the client-visible abstractions defined in the java.awt package.

In addition to named Loggers, it is also possible to create anonymous Loggers that don’t appear in the shared namespace. See Section 2.14.

By default, loggers send their output to a set of global Handlers managed by the *LogManager* infrastructure class (see 2.7 below).

The LogManager class allows log levels to be updated for named loggers. Setting a level for a given named Logger will also affect all its children. So setting a log level for “javax.swing” will also affect javax.swing.text” and “javax.swing.table” and “javax.swing.table.header”. The LogManager keeps track of level settings and when a new named Logger is created it is assigned the appropriate Level for its point in the naming hierarchy.

2.4 Logging methods

Note to reviewers: I realize some people would prefer that all logging methods should require source class name and method name. I also realize other people would prefer that none of the methods take source class name or method name. We are trying to please a diverse community, and at the moment it looks like we need to support both variants.

The logger class provides a large set of convenience methods for generating log messages. For convenience, there are methods for each of logging levels, named after the logging level name. Thus rather than calling “logger.log(Constants.WARNING, ...” a developer can simply call the convenience method “logger.warning(...”

There are two different styles of logging methods, to meet the needs of different communities of users.

First, there are methods that take an explicit source class name and source method name. These methods are intended for developers who want to be able to quickly locate the source of any given logging message. An example of this style is:

```
void warning(String sourceClass, String sourceMethod, String msg);
```

Second, there are a set of methods that do not take explicit source class or source method names. These are intended for developers who want easy-to-use logging and do not require detailed source information.

```
void warning(String msg);
```

For this second set of methods, the Logging framework will make a "best effort" to determine which class and method called into the logging framework and will add this information into the LogRecord. However, it is important to realize that this automatically inferred information may only be approximate. The latest generation of virtual machines do extensive optimizations when JITing and may entirely remove stack frames, making it impossible to reliably locate the calling class and method.

Note to reviewers: This is exactly the same situation as with Throwable backtraces. The information in a Throwable backtrace is also only approximate. This issue has been discussed with the Java Virtual Machine spec lead, and it is clear that the current JVM spec does not require that accurate call stack information be provided and it is also very unlikely that such a requirement would be added, as it would prevent potential JIT optimizations.

2.5 Handlers

The following handlers will be provided as part of J2SE:

- *StreamHandler*. A simple handler for writing formatted records to an OutputStream.
- *ConsoleHandler*. A simple handler for writing formatted records to System.err.
- *FileHandler.* A handler to write formatted log records to either a single file, or a set of rotating log files.
- *SocketHandler*. A handler to write formatted log records to remote TCP ports.
- *MemoryHandler*. A handler to buffer log records in memory.

It is fairly straightforward to develop new Handlers. Developers requiring specific functionality can either develop a Handler from scratch, or subclass one of the provided Handlers.

2.6 Formatters

J2SE will include two standard Formatters:

- *XMLFormatter*. Writes detailed XML structured information.
- *SimpleFormatter*. Write brief "human readable" summaries of log records.

As with Handlers it is fairly straightforward to develop new Formatters.

2.7 The LogManager

There is a global LogManager object that keeps track of global logging information. This includes:

- A hierarchical namespace of named Loggers. This includes recording the logging level settings for various points in the namespace, which can be used to initialize the logging levels for new Loggers.
- A set of logging control properties read from the configuration file (see 2.8 below).
- A set of "global Handlers" that are the default handlers used by all Loggers (unless they are disabled for a given Logger)

There is a single LogManager object that can be retrieved using the static LogManager.getLogManager object. This is created during LogManager initialization, based on a system property. This property allows container applications (such as EJB containers) to substitute their own subclass of LogManager in place of the default class.

2.8 Configuration File

The logging configuration can be initialized using a logging configuration file that will be read at startup. This logging configuration file is in standard java.util.Properties format.

Alternatively, the logging configuration can be initialized by specifying a class that can be used for reading initialization properties. This mechanism allows configuration data to be read from arbitrary sources, such as LDAP, JDBC, etc. See the LogManager javadoc for more details.

There is a small set of global configuration information. This is specified in the description of the LogManager class and includes a list of global Handlers to install during startup.

The initial configuration may specify levels for particular loggers. These levels are applied to the named logger and any loggers below it in the naming hierarchy. The levels are applied in the order they are defined in the configuration file.

The initial configuration may contain arbitrary properties for use by Handlers or by subsystems doing logging. By convention these properties should use names starting with the name of the handler class or the name of the main Logger for the subsystem.

For example, the MemoryHandler uses a property "java.util.logging.MemoryHandler.size" to determine the default size for its ring buffer.

2.9 Default Configuration

We need to decide on a default logging configuration to ship with the JRE. This is only a default, and can be overridden by ISVs, system admins, and end users. However defaults are important.

We need to make sure that the default configuration makes only limited use of disk space and also that it doesn't flood the user with information. At the same time we would like to make sure to always capture key failure information.

The proposed default configuration will establish two global Handlers, one for a file in the system temporary directory and one for the console. Only high priority messages will be recorded.

```
# Set the logging level for the root of the namespace.
# This becomes the default logging level for all Loggers.
.level= INFO
```

```
# List of global handlers
handlers = java.util.logging.FileHandler, \
                  java.util.logging.ConsoleHandler

# Properties for the FileHandler
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 3
java.util.logging.FileHandler.pattern = %t/java%u.%g.log

# Default level for ConsoleHandler. This can be used to
# limit the levels that are displayed on the console even
# when the global default has been set to a trace level.
java.util.logging.ConsoleHandler.level = INFO
```

2.10 Dynamic configuration updates

Programmers can update the logging configuration at run time in a variety of ways:

- FileHandlers, MemoryHandlers, and PrintHandlers can all be created with various attributes.
- The LogManager allows new global Handlers to be added and old ones removed.
- New Loggers can be created and can be supplied with specific Handlers.
- The LogManager.setLevel method allows new logging levels to be specified for trees within the Logger namespace.

2.11 Native Methods

There are no native APIs for logging.

Native code that wishes to use the Java Logging mechanisms should make normal JNI calls into the Java APIs for logging.

2.12 XML DTD

The XML DTD used by the XMLFormatter is specified in Appendix A.

The DTD is designed with a “<log>” element as the top-level document. Then individual log records are written as “<record>” elements.

Note to reviewers: Originally we tried to simply have a stream of <record> elements with no encompassing document. However standard XML parsers expect to deal with a single document element and cope poorly with a stream of independent elements. It is also necessary to have an XML header to specify the DTD and character-set encoding.

Note that in the event of JVM crashes it may not be possible to cleanly terminate an XMLFormatter stream with the appropriate closing </log>. Therefore tools that are analyzing log records should be prepared to cope with un-terminated streams.

2.13 Unique Message IDs

The Java Logging APIs do not provide any direct support for unique message IDs.

Those applications or subsystems requiring unique message IDs should define their own conventions and include the unique IDs in the message strings, where appropriate.

2.14 Security

The principal security requirement is that untrusted code should not be able to change the logging configuration. Specifically, if the logging configuration has been set up to log a particular category of information to a particular handler, then untrusted code should not be able to prevent or disrupt that logging.

A new security permission LoggingPermission is defined to control updates to the logging configuration.

Trusted applications are given the appropriate LoggingPermission so they can call any of the logging configuration APIs. Untrusted applets are a different story. Untrusted applets can create and use named Loggers in the normal way, but they are not allowed to change logging control settings, such as adding or removing handlers, or changing logging levels. However untrusted applets are able to create and use their own "anonymous" Loggers, using Logger.getAnonymousLogger. These anonymous Loggers are not registered in the global namespace and their methods are not access checked, allowing even untrusted code to change their logging control settings.

The logging framework does not attempt to prevent spoofing. The sources of logging calls cannot be determined reliably, so when a LogRecord is published that claims to be from a particular source class and source method, it may be a fabrication. Similarly, formatters such as the XMLFormatter do not attempt to protect themselves against nested log messages inside message strings. Thus, a spoof LogRecord might contain a spoof set of XML inside its message string to make it look as if there was an additional XML record in the output.

In addition, the logging framework does not attempt to protect itself against denial of service attacks. Any given logging client can flood the logging framework with meaningless messages in an attempt to conceal some important log message.

2.15 Configuration Management

Currently the APIs are structured so that an initial set of configuration information is read as properties from a configuration file. The configuration information may then be changed programatically by calls on the various logging classes and objects.

In addition, there are methods on LogManager that allow the configuration file to be reread. When this happens, the configuration file values will override any changes that have been made programatically.

2.16 Packaging

All the Logging classes are in the java.* part of the namespace, in the java.util.logging package. This helps to emphasize that they are part of the core platform and that people can rely on them being available.

2.17 Localization

Log messages may need to be localized.

Each Logger may have a Resource Bundle name associated with it. The corresponding Resource Bundle can be used to map between raw message strings and localized message strings.

Normally localization will be performed by Formatters. As a convenience, the Formatter class provides a formatMessage method that provides some basic localization and formatting support.

2.18 Remote access and serialization

As with most Java platform APIs, the logging APIs are designed for use inside a single address space. All calls are intended to be local. However, it is expected that some Handlers will want to forward their output to other systems. There are a variety of ways of doing this:

Some Handlers (such as the SocketHandler) may write data to other systems using the XMLFormatter. This provides a simple, standard, inter-change format which can be parsed and processed on a variety of systems.

Some Handlers may wish to pass LogRecord objects over RMI. The LogRecord class is therefore serializable. However there is a problem in how to deal with the LogRecord parameters. Some parameters may not be serializable and other parameters may have been designed to serialize much more state than is required for logging. To avoid these problems the LogRecord class has a custom writeObject method which converts the parameters to strings (using Object.toString()) before writing them out. See the javadoc for the LogRecord class for details.

Most of the logging classes are not intended to be serializable. Both Loggers and Handlers are very stateful classes and are tied into a specific virtual machine. In this respect they are analogous to the java.io classes which are also not serializable.

2.19 J2EE issues

For J2EE applications, the J2EE container typically controls the environment and provides the shared services that are used by all the contained components.

All the “global” state in the Logging APIs is maintained in the LogManager class. The LogManager APIs allow a program to substitute its own version of the LogManager class. Thus a J2EE container can replace the standard LogManager class with its own subclass that may implement different rules for managing “global” state. For example, a J2EE container might provide separate logging namespaces for logically distinct applications that are sharing the container.

Typically, J2EE containers will provide standard output Handlers for use by J2EE components. So typically J2EE components (such as EJBs) should not expect to create their own Handler classes, but should expect to use standard Handlers that have been configured as part of the container. However, J2EE components can freely create their own named Logger objects that use the standard Handlers.

3 Examples

3.1 Simple use

Here's a small program that does logging using the default configuration.

This program relies on the global handlers that were established by the LogManager based on the configuration file. It creates its own Logger object and then makes calls to that Logger object to report various events.

```
package com.wombat;
import java.util.logging.*;
public class Nose {
      // Obtain a suitable logger.
      private static Logger logger = Logger.getLogger("com.wombat.nose");
      public static void main(String argv[]) {
            // Log a FINE tracing message
            logger.fine("doing stuff");
            try {
                  Wombat.sneeze();
            } catch (Error ex) {
                  // Log the Error.
                  logger.log(Level.WARNING, "trouble sneezing", ex);
            }
            logger.fine("done");
      }
}
```

3.2 Changing the configuration

Here's a small program that dynamically adjusts the logging configuration to send output to a specific file and to get lots of information on wombats:

```
public static void main(String argv[]) {
      LogManager.removeAllGlobalHandlers();
      Handler fh = new FileHandler("%t/wombat.log");
      LogManager.addGlobalHandler(fh);
      LogManager.setLevel("com.wombat", Level.FINEST);
      ...
}
```

3.3 Simple use, ignoring global configuration

Here's a small program that sets up it own logging Handler and ignores the global configuration. Note that the logging output will still go to the global handlers.

```
package com.wombat;
import java.util.logging.*;
public class Nose {
      private static Logger logger = Logger.getLogger("com.wombat.nose");
      private static FileHandler fh = new FileHandler("mylog.txt");
      public static void main(String argv[]) {
            // Send logger output to our FileHandler.
            logger.addHandler(fh);
            // Request that every detail gets logged.
            logger.setLevel(Level.ALL);
            // Log a simple INFO message.
            logger.info("doing stuff");
            try {
                  Wombat.sneeze();
            } catch (Error ex) {
                  logger.log(Level.WARNING, "trouble sneezing", ex);
            }
            logger.fine("done");
      }
}
```

3.4 Sample XML output

Here's a small sample of what some XMLFormatter XML output looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2000-08-23 19:21:05</date>
  <millis>967083665789</millis>
  <sequence>1256</sequence>
  <logger>kgh.test.fred</logger>
  <level>INFO</level>
  <class>kgh.test.XMLTest</class>
  <method>writeLog</method>
  <thread>10</thread>
  <message>Hello world!</message>
</record>
</log>
```

package
java.util.logging

Class Summary

Interfaces

`Filter`	A Filter can be used to provide fine grain control over what is logged, beyond the control provided by log levels.

Classes

`ConsoleHandler`	This handler publishes log records to System.err.
`FileHandler`	Simple file logging Handler.
`Formatter`	A Formatter provides support for formatting LogRecords.
`Handler`	A Handler object takes log messages from a Logger and exports them.
`Level`	The Level class defines a set of standard logging levels that can be used to control logging output.
`Logger`	A Logger object is used to log messages for a specific system or application component.
`LoggingPermission`	The permission which the SecurityManager will check when code that is running with a SecurityManager calls one of the logging control methods (such as LogManager.setLevel).
`LogManager`	There is a single global LogManager object that is used to maintain a set of shared state about Loggers and log services.
`LogRecord`	LogRecord objects are used to pass logging requests between the logging framework and individual log Handlers.
`MemoryHandler`	Handler that buffers requests in a circular buffer in memory.
`SimpleFormatter`	Print a brief summary of the LogRecord in a human readable format.
`SocketHandler`	Simple network logging handler.
`StreamHandler`	Stream based logging Handler.
`XMLFormatter`	Format a LogRecord into a standard XML format.

java.util.logging
ConsoleHandler

Syntax

```
public class ConsoleHandler extends java.util.logging.StreamHandler
```

```
java.lang.Object
  |
  +--java.util.logging.Handler
        |
        +--java.util.logging.StreamHandler
              |
              +--java.util.logging.ConsoleHandler
```

Description

This handler publishes log records to System.err. By default the SimpleFormatter is used to generate brief summaries.

Configuration: By default each ConsoleHandler is initialized using the following LogManager configuration properties. If properties are not defined (or have invalid values) then the specified default values are used.

- java.util.logging.ConsoleHandler.level specifies the default level for the Handler (defaults to Level.INFO).
- java.util.logging.ConsoleHandler.filter specifies the name of a Filter class to use (defaults to no Filter).
- java.util.logging.ConsoleHandler.formatter specifies the name of a Formatter class to use (defaults to java.util.logging.SimpleFormatter).
- java.util.logging.ConsoleHandler.encoding the name of the character set encoding to use (defaults to the default platform encoding).

Since: 1.4

Member Summary	
Constructors	
ConsoleHandler()	Create a ConsoleHandler for System.err.
Methods	
close()	Override "StreamHandler.close" to do a flush but not to close the output stream.
publish(LogRecord)	Publish a LogRecord.

Constructors

ConsoleHandler()

```
public  ConsoleHandler()
```

Create a ConsoleHandler for System.err.

The ConsoleHandler is configured based on LogManager properties (or their default values).

Methods

close()

```
public void close()
```

Override "StreamHandler.close" to do a flush but not to close the output stream. That is, we do **not** close System.err.

Overrides: `StreamHandler.close()` in class `StreamHandler`

publish(LogRecord)

```
public void publish(LogRecord record)
```

Publish a LogRecord.

The logging request was made initially to a Logger object, which initialized the LogRecord and forwarded it here.

Overrides: `StreamHandler.publish(LogRecord)` in class `StreamHandler`

Parameters:

`record` - description of the log event

java.util.logging
FileHandler

Syntax

```
public class FileHandler extends java.util.logging.StreamHandler
```

```
java.lang.Object
  |
  +--java.util.logging.Handler
        |
        +--java.util.logging.StreamHandler
              |
              +--java.util.logging.FileHandler
```

Description

Simple file logging Handler.

The Handler can either write to a specified file, or it can write to a rotating set of files.

For a rotating set of files, as each file reaches a given size limit, it is closed, rotated out, and a new file opened. Successively older files are named by adding "0", "1", "2", etc into the base filename.

By default buffering is enabled in the IO libraries but each log record is flushed out when it is complete.

By default the XMLFormatter class is used for formatting.

Configuration: By default each FileHandler is initialized using the following LogManager configuration properties. If properties are not defined (or have invalid values) then the specified default values are used.

- java.util.logging.FileHandler.level specifies the default level for the Handler (defaults to Level.ALL).
- java.util.logging.FileHandler.filter specifies the name of a Filter class to use (defaults to no Filter).
- java.util.logging.FileHandler.formatter specifies the name of a Formatter class to use (defaults to java.util.logging.XMLFormatter)
- java.util.logging.FileHandler.encoding the name of the character set encoding to use (defaults to the default platform encoding).
- java.util.logging.FileHandler.limit specifies an approximate maximum amount to write (in bytes) to any one file. If this is zero, then there is no limit. (Defaults to no limit).
- java.util.logging.FileHandler.count specifies how many output files to cycle through (defaults to 1).
- java.util.logging.FileHandler.pattern specifies a pattern for generating the output file name. See below for details. (Defaults to "%t/java.log").

A pattern consists of a string that includes the following special components that will be replaced at runtime:

- "/" the local pathname separator
- "%t" the system temporary directory
- "%h" the value of the "user.home" system property
- "%g" the generation number to distinguish rotated logs
- "%u" a unique number to resolve conflicts
- "%%" translates to a single percent sign "%"

If no "%g" field has been specified and the file count is greater than one, then the generation number will be added to the end of the generated filename, after a dot.

Thus for example a pattern of "%t/java%g.log" with a count of 2 would typically cause log files to be written on Solaris to /var/tmp/java0.log and /var/tmp/java1.log whereas on Windows 95 they would be typically written to to C:\TEMP\java0.log and C:\TEMP\java1.log

Generation numbers follow the sequence 0, 1, 2, etc.

Normally the "%u" unique field is set to 0. However, if the FileHandler tries to open the filename and finds the file is currently in use by another process it will increment the unique number field and try again. This will be repeated until FileHandler finds a file name that is not currently in use. If there is a conflict and no "%u" field has been specified, it will be added at the end of the filename after a dot. (This will be after any automatically added generation number.)

Thus if three processes were all trying to log to fred%u.%g.txt then they might end up using fred0.0.txt, fred1.0.txt, fred2.0.txt as the first file in their rotating sequences.

Note that the use of unique ids to avoid conflicts is only guaranteed to work reliably when using a local disk file system.

Since: 1.4

Member Summary	
Constructors	
`FileHandler()`	Construct a default FileHandler.
`FileHandler(String)`	Initialize a FileHandler to write to the given filename.
`FileHandler(String, int, int)`	Initialize a FileHandler to write to a set of files.
Methods	
`close()`	Close all the files.
`publish(LogRecord)`	Format and publish a LogRecord.

Constructors

FileHandler()

```
public  FileHandler()
```

Construct a default FileHandler. This will be configured entirely from LogManager properties (or their default values).

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

FileHandler(String)

```
public  FileHandler(java.lang.String pattern)
```

Initialize a FileHandler to write to the given filename.

The FileHandler is configured based on LogManager properties (or their default values) except that the given pattern argument is used as the filename pattern, the file limit is set to no limit, and the file count is set to one.

There is no limit on the amount of data that may be written, so use this with care.

Parameters:
`pattern` - the name of the output file

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

FileHandler(String, int, int)

```
public  FileHandler(java.lang.String pattern, int limit, int count)
```

Initialize a FileHandler to write to a set of files. When (approximately) the given limit has been written to one file, another file will be opened. The output will cycle through a set of count files.

The FileHandler is configured based on LogManager properties (or their default values) except that the given pattern argument is used as the filename pattern, the file limit is set to the limit argument, and the file count is set to the given count argument.

The count must be at least 1.

Parameters:
`pattern` - the pattern for naming the output file

`limit` - the maximum number of bytes to write to any one file

`count` - the number of files to use

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

Methods

close()

```
public void close()
```

Close all the files.

Overrides: `StreamHandler.close()` in class `StreamHandler`

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

publish(LogRecord)

```
public void publish(LogRecord record)
```

Format and publish a LogRecord.

Overrides: `StreamHandler.publish(LogRecord)` in class `StreamHandler`

Parameters:
`record` - description of the log event

java.util.logging
Filter

Syntax
```
public interface Filter
```

Description
A Filter can be used to provide fine grain control over what is logged, beyond the control provided by log levels.

Each Logger and each Handler can have a filter associated with it. The Logger or Handler will call the isLoggable method to check if a given LogRecord should be published. If isLoggable returns false, the LogRecord will be discarded.

Since: 1.4

Member Summary	
Methods	
`isLoggable(LogRecord)`	Check if a given log record should be published.

Methods

isLoggable(LogRecord)

```
public boolean isLoggable(LogRecord record)
```

Check if a given log record should be published.

Parameters:
`record` - a LogRecord

Returns: true if the log record should be published.

java.util.logging
Formatter

Syntax

```
public abstract class Formatter
```

```
java.lang.Object
  |
  +--java.util.logging.Formatter
```

Direct Known Subclasses: `SimpleFormatter`, `XMLFormatter`

Description

A Formatter provides support for formatting LogRecords.

Typically each logging Handler will have a Formatter associated with it. The Formatter takes a LogRecord and converts it to a string.

Some formatters (such as the XMLFormatter) need to wrap head and tail strings around a set of formatted records. The getHeader and getTail methods can be used to obtain these strings.

Since: 1.4

Member Summary	
Constructors	
`Formatter()`	Construct a new formatter.
Methods	
`format(LogRecord)`	Format the given log record and return the formatted string.
`formatMes-sage(LogRecord)`	Localize and format the message string from a log record.
`getHead(Handler)`	Return the header string for a set of formatted records.
`getTail(Handler)`	Return the tail string for a set of formatted records.

Constructors

Formatter()

```
protected  Formatter()
```

Construct a new formatter.

Methods

format(LogRecord)

`public abstract java.lang.String format(LogRecord record)`

Format the given log record and return the formatted string.

The resulting formatted String will normally include a localized and formated version of the LogRecord's message field. The LogRecord.formatMessage convenience method can (optionally) be used to localize and format the message field.

Parameters:
`record` - the log record to be formatted.

Returns: the formatted log record

formatMessage(LogRecord)

`public java.lang.String formatMessage(LogRecord record)`

Localize and format the message string from a log record. This method is provided as a convenience for Formatter subclasses to use when they are performing formatting.

The message string is first localized to a format string using the record's ResourceBundle. (If there is no ResourceBundle, or if the message key is not found, then the key is used as the format string.) The format String may use either java.text style formatting or (starting in JDK 1.4) printf style formatting:

- If there are no parameters, no formatter is used.
- Otherwise, if the string contains "{0" then java.text.MessageFormat is used to format the string.
- Otherwise printf style formatting is used.

NOTE TO REVIEWERS: I am making a forward reference here to the printf style formatting that is planned for Merlin. This formatting will be fully internationalizable and is likely to be significantly easier to use than the current java.text formatting, so it seems useful to support it.

Parameters:
`record` - the log record containing the raw message

Returns: a localized and formatted message

getHead(Handler)

`public java.lang.String getHead(Handler h)`

Return the header string for a set of formatted records.

This base class returns an empty string, but this may be overriden by subclasses.

Parameters:
`h` - The target handler.

Returns: header string

getTail(Handler)

`public java.lang.String getTail(Handler h)`

Return the tail string for a set of formatted records.

This base class returns an empty string, but this may be overriden by subclasses.

Parameters:
h - The target handler.

Returns: tail string

java.util.logging
Handler

Syntax

```
public abstract class Handler
```

```
java.lang.Object
  |
  +--java.util.logging.Handler
```

Direct Known Subclasses: `MemoryHandler`, `StreamHandler`

Description

A Handler object takes log messages from a Logger and exports them. It might for example, write them to a console or write them to a file, or send them to a network logging service, or forward them to an OS log, or whatever.

A Handler can be disabled by doing a setLevel(Level.OFF) and can be re-enabled by doing a setLevel with an appropriate level.

Handler classes typically use LogManager properties to set default values for Filter, Formatter, and Level. See the specific documentation for each concrete Handler class.

Since: 1.4

Member Summary	
Constructors	
`Handler()`	Default constructor.
`Handler(Formatter)`	Construct a Handler with the given formatter.
Methods	
`close()`	Close the Handler and free all associated resources.
`flush()`	Flush any buffered output.
`getEncoding()`	Return the character encoding for this Handler.
`getException()`	Report on the most recent exception (if any) to have been detected while writing to this Handler.
`getFilter()`	Get the current filter for this Handler.
`getFormatter()`	Return the Formatter for this Handler.
`getLevel()`	Get the log level specifying which messages will be logged by this Handler.
`isLoggable(LogRecord)`	Check if this Handler would actually log a given LogRecord.
`publish(LogRecord)`	Publish a LogRecord.
`setEncoding(String)`	Set the character encoding used by this Handler.
`setException(Exception)`	Set the most recent IO exception.
`setFilter(Filter)`	Set a filter to control output on this Handler.
`setFormatter(Formatter)`	Set a Formatter.
`setLevel(Level)`	Set the log level specifying which message levels will be logged by this handler.

Constructors

Handler()

```
protected  Handler()
```

Default constructor. The resulting Handler has a log level of ALL, no formatter, and no filter.

Handler(Formatter)

```
protected  Handler(Formatter formatter)
```

Construct a Handler with the given formatter.

Parameters:
`formatter` - Formatter to be used to format output.

Methods

close()

```
public abstract void close()
```

Close the Handler and free all associated resources.

The close method will perform a "flush" and then close the Handler. After close has been called this Handler should no longer be used. Method calls may either be silently ignored or may throw runtime exceptions.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

flush()

```
public abstract void flush()
```

Flush any buffered output.

getEncoding()

```
public java.lang.String getEncoding()
```

Return the character encoding for this Handler.

Returns: The encoding name. May be null, which indicates the default encoding should be used.

getException()

```
public java.lang.Exception getException()
```

Report on the most recent exception (if any) to have been detected while writing to this Handler.

Does a "flush" and then reports the underlying IO status. If the most recent write has been successful then the result will be null. Otherwise it will be an exception value that a subclass has set using setException.

Returns: null if recent IO operations have succeeded, otherwise the most recently detected exception.

getFilter()

`public Filter getFilter()`

Get the current filter for this Handler.

Returns: a filter object (may be null)

getFormatter()

`public Formatter getFormatter()`

Return the Formatter for this Handler.

Returns: the formatter (may be null).

getLevel()

`public Level getLevel()`

Get the log level specifying which messages will be logged by this Handler. Message levels lower than this level will be discarded.

Returns: the level of messages being logged.

isLoggable(LogRecord)

`public boolean isLoggable(LogRecord record)`

Check if this Handler would actually log a given LogRecord.

This method checks if the LogRecord has an appropriate level and whether it satisfies any Filter. It also may make other Handler specific checks that might prevent a handler from logging the LogRecord.

Parameters:
`record` - a LogRecord

Returns: true if the log record would be logged.

publish(LogRecord)

`public abstract void publish(LogRecord record)`

Publish a LogRecord.

The logging request was made initially to a Logger object, which initialized the LogRecord and forwarded it here.

The Handler is responsible for formatting the message, when and if necessary. The formatting should include localization.

Parameters:
`record` - description of the log event

setEncoding(String)

`public void setEncoding(java.lang.String encoding)`

Set the character encoding used by this Handler.

The encoding should be set before any LogRecords are written to the Handler.

Parameters:

`encoding` - The name of a supported character encoding. May be null, to indicate the default platform encoding.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

`UnsupportedEncodingException` - if the named encoding is not supported.

setException(Exception)

```
protected void setException(java.lang.Exception exception)
```

Set the most recent IO exception.

This protected method allows subclasses to record any exception that happens while doing IO on this handler, so it can later be retrieved by a call on getException.

Parameters:

`exception` - An exception that occurred during a write. May be null, to clear exception state.

setFilter(Filter)

```
public void setFilter(Filter newFilter)
```

Set a filter to control output on this Handler.

For each call of "publish" the Handler will call this Filter (if it is non-null) to check if the log record should be published or discarded.

Parameters:

`newFilter` - a filter object (may be null)

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

setFormatter(Formatter)

```
public void setFormatter(Formatter newFormatter)
```

Set a Formatter. This formatter will be used to format LogRecords for this Handler.

Some Handlers may not use Formatters, in which case the formatter will be remembered, but not used.

Parameters:

`newFormatter` - the formatter to use (may not be null)

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

setLevel(Level)

```
public void setLevel(Level newLevel)
```

Set the log level specifying which message levels will be logged by this handler. Message levels lower than this value will be discarded.

The intention is to allow developers to turn on voluminous logging, but to limit the messages that are sent to certain Handlers.

Parameters:

`newLevel` - the new value for the log level

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

java.util.logging
Level

Syntax

```
public class Level implements java.io.Serializable
```

```
java.lang.Object
  |
  +--java.util.logging.Level
```

All Implemented Interfaces: java.io.Serializable

Description

The Level class defines a set of standard logging levels that can be used to control logging output. The logging Level objects are ordered and are specified by ordered integers. Enabling logging at a given level also enables logging at all higher levels.

Clients should normally use the predefined Level constants such as Level.SEVERE.

The levels in descending order are:

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)

In addition there is a level OFF that can be used to turn off logging, and a level ALL that can be used to enable logging of all messages.

It is possible for third parties to define additional logging levels by subclassing Level. In such cases subclasses should take care to chose unique integer level values and to ensure that they maintain the Object uniqueness property across serialization by defining a suitable readResolve method.

Since: 1.4

Member Summary	
Fields	
ALL	ALL indicates that all messages should be logged.
CONFIG	CONFIG is a message level for static configuration messages.
FINE	FINE is a message level providing tracing information.
FINER	FINER indicates a fairly detailed tracing message.
FINEST	FINEST indicates a highly detailed tracing message
INFO	INFO is a message level for informational messages.
OFF	OFF is a special level that can be used to turn off logging.
SEVERE	SEVERE is a message level indicating a serious failure.

Member Summary	
WARNING	WARNING is a message level indicating a potential problem.
Constructors	
Level(String, int)	Create a named Level with a given integer value.
Methods	
equals(Object)	Compare two objects for value equality.
hashCode()	Generate a hashcode.
intValue()	Get the integer value for this level.
parse(String)	Parse a level name string into a Level.
toString()	

Fields

ALL

```
public static final Level ALL
```

ALL indicates that all messages should be logged.

CONFIG

```
public static final Level CONFIG
```

CONFIG is a message level for static configuration messages.

CONFIG messages are intended to provide a variety of static configuration information, to assist in debugging problems that may be associated with particular configurations. For example, CONFIG message might include the CPU type, the graphics depth, the GUI look-and-feel, etc.

FINE

```
public static final Level FINE
```

FINE is a message level providing tracing information.

All of FINE, FINER, and FINEST are intended for relatively detailed tracing. The exact meaning of the three levels will vary between subsystems, but in general, FINEST should be used for the most voluminous detailed output, FINER for somewhat less detailed output, and FINE for the lowest volume (and most important) messages.

In general the FINE level should be used for information that will be broadly interesting to developers who do not have a specialized interest in the specific subsystem.

FINE messages might include things like minor (recoverable) failures. Issues indicating potential performance problems are also worth logging as FINE.

FINER

```
public static final Level FINER
```

FINER indicates a fairly detailed tracing message. By default logging calls for entering, returning, or throwing an exception are traced at this level.

FINEST

```
public static final Level FINEST
```

FINEST indicates a highly detailed tracing message

INFO

```
public static final Level INFO
```

INFO is a message level for informational messages.

Typically INFO messages will be written to the console or its equivalent. So the INFO level should only be used for reasonably significant messages that will make sense to end users and system admins.

OFF

```
public static final Level OFF
```

OFF is a special level that can be used to turn off logging.

SEVERE

```
public static final Level SEVERE
```

SEVERE is a message level indicating a serious failure.

In general SEVERE messages should describe events that are of considerable importance and which will prevent normal program execution. They should be reasonably intelligible to end users and to system administrators.

WARNING

```
public static final Level WARNING
```

WARNING is a message level indicating a potential problem.

In general WARNING messages should describe events that will be of interest to end users or system managers, or which indicate potential problems.

Constructors

Level(String, int)

```
protected  Level(java.lang.String name, int value)
```

Create a named Level with a given integer value.

Note that this constructor is "protected" to allow subclassing. In general clients of logging should use one of the constant Level objects such as SEVERE or FINEST. However, if clients need to add new logging levels, they may subclass Level and define new constants.

Parameters:

`name` - the name of the Level, for example "SEVERE".

`value` - an integer value for the level.

Methods

equals(Object)

```
public boolean equals(java.lang.Object ox)
```

Compare two objects for value equality.

Overrides: java.lang.Object.equals(java.lang.Object) in class java.lang.Object

Returns: true if and only if the two objects have the same level value.

hashCode()

```
public int hashCode()
```

Generate a hashcode.

Overrides: java.lang.Object.hashCode() in class java.lang.Object

Returns: a hashcode based on the level value

intValue()

```
public final int intValue()
```

Get the integer value for this level. This integer value can be used for efficient ordering comparisons between Level objects.

Returns: the integer value for this level.

parse(String)

```
public static Level parse(java.lang.String name)
```

Parse a level name string into a Level.

The argument string may consist of either a level name or an integer value.

For example:

- "SEVERE"
- "1000"

Parameters:
`level` - string to be parsed

Returns: parsed value

Throws: `NullPointerException` - if the name is null

`IllegalArgumentException` - if the value is neither one of the known names nor an integer.

toString()

```
public final java.lang.String toString()
```

Overrides: java.lang.Object.toString() in class java.lang.Object

Returns: the string name of the Level, for example "INFO".

java.util.logging
Logger

Syntax

```
public class Logger
```

```
java.lang.Object
  |
  +--java.util.logging.Logger
```

Description

A Logger object is used to log messages for a specific system or application component. Loggers are normally named, using a hierarchical dot-separated namespace. Logger names can be arbitrary strings, but they should normally be based on the package name or class name of the logged component, such as java.net or javax.swing. In additon it is possible to create "anonymous" Loggers that are not stored in the Logger namespace.

Logger objects may be obtained by calls on one of the getLogger factory methods. These will either create a new Logger or return a suitable existing Logger.

Logging messages will be forwarded to registered Handler objects, which can forward the messages to a variety of destinations, including consoles, files, OS logs, etc.

On each logging call the Logger initially performs a cheap check of the request level (e.g. SEVERE or FINE) against a log level maintained by the logger. If the request level is lower than the log level, the logging call returns immediately.

The log level is originally initialized based on the properties from the logging configuration file, as described in the description of the LogManager class. However it may also be dynamically changed by calls on the Logger.setLevel or LogManager.setLevel methods.

After passing this initial (cheap) test, the Logger will allocate a LogRecord to describe the logging message. It will then call a Filter (if present) to do a more detailed check on whether the record should be published. If that passes it will then publish the LogRecord to the target Handlers.

Most of the logger output methods take a "msg" argument. This msg argument may be either a raw value or a localization key. During formatting, if the logger has a localization ResourceBundle and if the ResourceBundle has a mapping for the msg string, then the msg string is replaced by the localized value. Otherwise the original msg string is used.

Formatting (including localization) is the responsibility of the output Handler, which will typically call a Formatter.

Note that formatting need not occur synchronously. It may be delayed until a LogRecord is actually written to an external sink.

There are various logging convenience methods:

First, to make it easier to identify the source of a logging request there are a set of logging methods that take the source class name and the source method name. For example:

- void warning(String sourceClass, string sourceMethod, String msg);

Second, for developers who do not wish to provide this source information there are also methods that do not require these arguments. For example:

- void warning(String msg);

For this second set of methods, the Logging framework will make a "best effort" to determine which class and method called into the logging method. However, it is important to realize that this automatically inferred information may only

be approximate (or may even be quite wrong!). Virtual machines are allowed to do extensive optimizations when JITing and may entirely remove stack frames, making it impossible to reliably locate the calling class and method.

All methods on Logger are multi-thread safe.

Subclassing Information: Subclasses of Logger are required to provide their own getLogger factory methods. In order to intercept all logging output, subclasses need only override the log(LogRecord) method. All the other log() methods are implemented as calls on this log(LogRecord) method.

Since: 1.4

Member Summary

Fields

`global`	The "global" Logger object is provided as a convenience to developers who are making casual use of the Logging package.

Constructors

`Logger(String, String)`	Protected method to construct a logger for a named subsystem.

Methods

`addHandler(Handler)`	Add a log Handler to receive logging messages.
`config(String)`	Log a CONFIG message.
`config(String, Object[])`	Log a CONFIG message, with an array of object arguments.
`config(String, String, String)`	Log a CONFIG message, specifying source class and method.
`config(String, String, String, Object[])`	Log a CONFIG message, specifying source class and method, with an array of object arguments.
`entering(String, String)`	Log a procedure entry.
`entering(String, String, Object[])`	Log a procedure entry, with parameters.
`exiting(String, String)`	Log a procedure return.
`exiting(String, String, Object)`	Log a procedure return, with result object.
`fine(String)`	Log a FINE message.
`fine(String, Object[])`	Log a FINE message, with an array of object arguments.
`fine(String, String, String)`	Log a FINE message, specifying source class and method.
`fine(String, String, String, Object[])`	Log a FINE message, specifying source class and method, with an array of object arguments.
`finer(String)`	Log a FINER message.
`finer(String, Object[])`	Log a FINER message, with an array of object arguments.
`finer(String, String, String)`	Log a FINER message, specifying source class and method.
`finer(String, String, String, Object[])`	Log a FINER message, specifying source class and method, with an array of object arguments.
`finest(String)`	Log a FINEST message.
`finest(String, Object[])`	Log a FINEST message, with an array of object arguments.

Member Summary	
finest(String, String, String)	Log a FINEST message, specifying source class and method.
finest(String, String, String, Object[])	Log a FINEST message, specifying source class and method, with an array of object arguments.
getAnonymousLogger()	Create an anonymous Logger.
getAnonymousLogger(String)	Create an anonymous Logger.
getFilter()	Get the current filter for this Logger.
getHandlers()	Get the Handlers associated with this logger.
getLevel()	Get the log level specifying which messages will be logged by this logger.
getLogger(String)	Find or create a logger for a named subsystem.
getLogger(String, String)	Find or create a logger for a named subsystem.
getName()	Get the name for this logger.
getResourceBundle()	Retrieve the localization resource bundle for this logger.
getResourceBundleName()	Retrieve the localization resource bundle name for this logger.
getUseGlobalHandlers()	Discover whether or not this logger is sending its output to the set of global handlers managed by the LogManager.
info(String)	Log an INFO message.
info(String, Object[])	Log an INFO message, with an array of object arguments.
info(String, String, String)	Log an INFO message, specifying source class and method.
info(String, String, String, Object[])	Log an INFO message, specifying source class and method, with an array of object arguments.
isLoggable(Level)	Check if a message of the given level would actually be logged by this logger.
log(Level, String)	Log a message, with no arguments.
log(Level, String, Object[])	Log a message, with an array of object arguments.
log(Level, String, String, String)	Log a message, specifying source class and method, with no arguments.
log(Level, String, String, String, Object[])	Log a message, specifying source class and method, with an array of object arguments.
log(Level, String, String, String, Throwable)	Log a message, specifying source class and method, with associated Throwable information.
log(Level, String, Throwable)	Log a message, with associated Throwable information.
log(LogRecord)	Log a LogRecord.
removeHandler(Handler)	Remove a log Handler.
setFilter(Filter)	Set a filter to control output on this Logger.
setLevel(Level)	Set the log level specifying which message levels will be logged by this logger.
setUseGlobalHandlers(boolean)	Specify whether or not this logger should send its output to the set of global handlers managed by the LogManager.
severe(String)	Log a SEVERE message.
severe(String, Object[])	Log a SEVERE message, with an array of object arguments.
severe(String, String, String)	Log a SEVERE message, specifying source class and method.
severe(String, String, String, Object[])	Log a SEVERE message, specifying source class and method, with an array of object arguments.

Member Summary	
throwing(String, String, Throwable)	Log throwing an exception.
warning(String)	Log a WARNING message.
warning(String, Object[])	Log a WARNING message, with an array of object arguments.
warning(String, String, String)	Log a WARNING message, specifying source class and method.
warning(String, String, String, Object[])	Log a WARNING message, specifying source class and method, with an array of object arguments.

Fields

global

```
public static final Logger global
```

The "global" Logger object is provided as a convenience to developers who are making casual use of the Logging package. Developers who are making serious use of the logging package (for example in products) should create and use their own Logger objects, with appropriate names, so that logging can be controlled on a suitable per-Logger granularity.

The global logger is initialized by calling Logger.getLogger("global").

Constructors

Logger(String, String)

```
protected  Logger(java.lang.String name, java.lang.String resourceBundleName)
```

Protected method to construct a logger for a named subsystem.

The logger will be initially configured with a level of Level.ALL and with useGlobalHandlers true.

Parameters:

`name` - A name for the logger. This should be a dot-separated name and should normally be based on the package name or class name of the subsystem, such as java.net or javax.swing. It may be null for anonymous Loggers.

`resourceBundleName` - name of ResourceBundle to be used for localizing messages for this logger. May be null if none of the messages require localization.

Throws: `MissingResourceException` - if the ResourceBundleName is non-null and no corresponding resource can be found.

Methods

addHandler(Handler)

```
public void addHandler(Handler handler)
```

Add a log Handler to receive logging messages.

By default, Loggers send their output to all the global handlers. Application programmers only need to use the addHandler method if they are configuring a Logger with a non-global output handler. Adding a non-global handler does not affect the use of global handlers.

Parameters:
`handler` - a logging Handler

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

config(String)

```
public void config(java.lang.String msg)
```

Log a CONFIG message.

If the logger is currently enabled for the CONFIG message level then the given message is forwarded to all the registered output Handler objects.

Parameters:
`msg` - The string message (or a key in the message catalog)

config(String, Object[])

```
public void config(java.lang.String msg, java.lang.Object[] params)
```

Log a CONFIG message, with an array of object arguments.

If the logger is currently enabled for the CONFIG message level then the given message is forwarded to all the registered output Handler objects.

Parameters:
`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

config(String, String, String)

```
public void config(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg)
```

Log a CONFIG message, specifying source class and method.

If the logger is currently enabled for the CONFIG message level then the given message is forwarded to all the registered output Handler objects.

Parameters:
`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

config(String, String, String, Object[])

```
public void config(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg, java.lang.Object[] params)
```

Log a CONFIG message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the CONFIG message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

entering(String, String)

```
public void entering(java.lang.String sourceClass, java.lang.String sourceMethod)
```

Log a procedure entry.

This is a convenience method that can be used to log entry to a method. A LogRecord with message "ENTRY", log level FINER, and the given sourceMethod and sourceClass is logged.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that is being entered

entering(String, String, Object[])

```
public void entering(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.Object[] params)
```

Log a procedure entry, with parameters.

This is a convenience method that can be used to log entry to a method. A LogRecord with message "ENTRY", log level FINER, and the given sourceMethod, sourceClass, and params is logged.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that is being entered

`params` - array of parameters to the method being entered

exiting(String, String)

```
public void exiting(java.lang.String sourceClass, java.lang.String sourceMethod)
```

Log a procedure return.

This is a convenience method that can be used to log returning from a method. A LogRecord with message "RETURN", log level FINER, and the given sourceMethod and sourceClass is logged.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of the method

exiting(String, String, Object)

```
public void exiting(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.Object result)
```

Log a procedure return, with result object.

This is a convenience method that can be used to log returning from a method. A LogRecord with message "RETURN", log level FINER, and the gives sourceMethod, sourceClass, and result object is logged.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of the method

`result` - Object that is being returned

fine(String)

```
public void fine(java.lang.String msg)
```

Log a FINE message.

If the logger is currently enabled for the FINE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

fine(String, Object[])

```
public void fine(java.lang.String msg, java.lang.Object[] params)
```

Log a FINE message, with an array of object arguments.

If the logger is currently enabled for the FINE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

fine(String, String, String)

```
public void fine(java.lang.String sourceClass, java.lang.String sourceMethod, java.lang.String
            msg)
```

Log a FINE message, specifying source class and method.

If the logger is currently enabled for the FINE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

fine(String, String, String, Object[])

```
public void fine(java.lang.String sourceClass, java.lang.String sourceMethod, java.lang.String
            msg, java.lang.Object[] params)
```

Log a FINE message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the FINE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

finer(String)

```
public void finer(java.lang.String msg)
```

Log a FINER message.

If the logger is currently enabled for the FINER message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

finer(String, Object[])

```
public void finer(java.lang.String msg, java.lang.Object[] params)
```

Log a FINER message, with an array of object arguments.

If the logger is currently enabled for the FINER message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

finer(String, String, String)

```
public void finer(java.lang.String sourceClass, java.lang.String sourceMethod, java.lang.String
            msg)
```

Log a FINER message, specifying source class and method.

If the logger is currently enabled for the FINER message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

finer(String, String, String, Object[])

```
public void finer(java.lang.String sourceClass, java.lang.String sourceMethod, java.lang.String
            msg, java.lang.Object[] params)
```

Log a FINER message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the FINER message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

finest(String)

```
public void finest(java.lang.String msg)
```

Log a FINEST message.

If the logger is currently enabled for the FINEST message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

finest(String, Object[])

```
public void finest(java.lang.String msg, java.lang.Object[] params)
```

Log a FINEST message, with an array of object arguments.

If the logger is currently enabled for the FINEST message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

finest(String, String, String)

```
public void finest(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg)
```

Log a FINEST message, specifying source class and method.

If the logger is currently enabled for the FINEST message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

finest(String, String, String, Object[])

```
public void finest(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg, java.lang.Object[] params)
```

Log a FINEST message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the FINEST message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

getAnonymousLogger()

```
public static Logger getAnonymousLogger()
```

Create an anonymous Logger. The newly created Logger is not registered in the LogManager namespace and will not be affected by updates to the logging configuration or by calls on LogManager.setLevel. There will also be no access checks on updates to the logger.

This factory method is primarily intended for use from applets. Because the resulting Logger is anonymous it can be kept private by the creating class. This removes the need for normal security checks, which in turn allows untrusted applet code to update the control state of the Logger. For example an applet can do a setLevel or an addHandler on an anonymous Logger.

The new logger is initially configured with the logging level for the root name ("") from the LogManager class. It is initially configured to use the LogManager's global handlers.

Returns: a newly created private Logger

getAnonymousLogger(String)

```
public static Logger getAnonymousLogger(java.lang.String resourceBundleName)
```

Create an anonymous Logger. The newly created Logger is not registered in the LogManager namespace and will not be affected by updates to the logging configuration or by calls on LogManager.setLevel. There will also be no access checks on updates to the Logger.

This factory method is primarily intended for use from applets. Because the resulting Logger is anonymous it can be kept private by the creating class. This removes the need for normal security checks, which in turn allows untrusted applet code to update the control state of the Logger. For example an applet can do a setLevel or an addHandler on an anonymous Logger.

The new logger is initially configured with the logging level for the root name ("") from the LogManager class. It is initially configured to use the LogManager's global handlers.

Parameters:

`resourceBundleName` - name of ResourceBundle to be used for localizing messages for this logger.

Returns: a newly created private Logger

Throws: `MissingResourceException` - if the named ResourceBundle cannot be found.

getFilter()

`public Filter getFilter()`

Get the current filter for this Logger.

Returns: a filter object (may be null)

getHandlers()

`public Handler[] getHandlers()`

Get the Handlers associated with this logger.

The resulting array includes all Handlers set with addHandler, but excludes global Handlers.

Returns: an array of all registered Handlers

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

getLevel()

`public Level getLevel()`

Get the log level specifying which messages will be logged by this logger. Message levels lower than this level will be discarded.

Returns: the level of messages being logged.

getLogger(String)

`public static Logger getLogger(java.lang.String name)`

Find or create a logger for a named subsystem. If a logger has already been created with the given name it is returned. Otherwise a new logger is created.

If a new logger is created its log level will be configured based on the LogManager configuration and it will configured to send logging output to all global output Handlers. It will be registered in the LogManager global namespace. This will mean it will be affected by subsequent LogManager.setLevel calls.

Parameters:

`name` - A name for the logger. This should be a dot-separated name and should normally be based on the package name or class name of the subsystem, such as java.net or javax.swing

Returns: a suitable Logger

getLogger(String, String)

`public static Logger getLogger(java.lang.String name, java.lang.String resourceBundleName)`

Find or create a logger for a named subsystem. If a logger has already been created with the given name it is returned. Otherwise a new logger is created.

If a new logger is created its log level will be configured based on the LogManager and it will configured to send logging output to all global output Handlers. It will be registered in the LogManager global namespace. This will mean it will be affected by subsequent LogManager.setLevel calls.

If the named Logger already exists and does not yet have a localization resource bundle then the given resource bundle name is used. If the named Logger already exists and has a different resource bundle name then an IllegalArgumentException is thrown.

Parameters:

`name` - A name for the logger. This should be a dot-separated name and should normally be based on the package name or class name of the subsystem, such as java.net or javax.swing

`resourceBundleName` - name of ResourceBundle to be used for localizing messages for this logger.

Returns: a suitable Logger

Throws: `MissingResourceException` - if the named ResourceBundle cannot be found.

`IllegalArgumentName` - if the Logger already exists and uses a different resource bundle name.

getName()

```
public java.lang.String getName()
```

Get the name for this logger.

Returns: logger name. Will be null for anonymous Loggers.

getResourceBundle()

```
public java.util.ResourceBundle getResourceBundle()
```

Retrieve the localization resource bundle for this logger.

Returns: localization bundle (may be null)

getResourceBundleName()

```
public java.lang.String getResourceBundleName()
```

Retrieve the localization resource bundle name for this logger.

Returns: localization bundle name (may be null)

getUseGlobalHandlers()

```
public boolean getUseGlobalHandlers()
```

Discover whether or not this logger is sending its output to the set of global handlers managed by the LogManager.

Returns: true if output is to be sent to the LogManager's global handlers.

info(String)

```
public void info(java.lang.String msg)
```

Log an INFO message.

If the logger is currently enabled for the INFO message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

info(String, Object[])

```
public void info(java.lang.String msg, java.lang.Object[] params)
```

Log an INFO message, with an array of object arguments.

If the logger is currently enabled for the INFO message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

info(String, String, String)

```
public void info(java.lang.String sourceClass, java.lang.String sourceMethod, java.lang.String
            msg)
```

Log an INFO message, specifying source class and method.

If the logger is currently enabled for the INFO message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

info(String, String, String, Object[])

```
public void info(java.lang.String sourceClass, java.lang.String sourceMethod, java.lang.String
            msg, java.lang.Object[] params)
```

Log an INFO message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the INFO message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

isLoggable(Level)

```
public boolean isLoggable(Level level)
```

Check if a message of the given level would actually be logged by this logger.

Parameters:

`level` - a message logging level

Returns: true if the given message level is currently being logged.

log(Level, String)

```
public void log(Level level, java.lang.String msg)
```

Log a message, with no arguments.

If the logger is currently enabled for the given message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`level` - One of the message level identifiers, e.g. SEVERE

`msg` - The string message (or a key in the message catalog)

log(Level, String, Object[])

```
public void log(Level level, java.lang.String msg, java.lang.Object[] params)
```

Log a message, with an array of object arguments.

If the logger is currently enabled for the given message level then a corresponding LogRecord is created and forwarded to all the registered output Handler objects.

Parameters:

`level` - One of the message level identifiers, e.g. SEVERE

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

log(Level, String, String, String)

```
public void log(Level level, java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg)
```

Log a message, specifying source class and method, with no arguments.

If the logger is currently enabled for the given message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`level` - One of the message level identifiers, e.g. SEVERE

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

log(Level, String, String, String, Object[])

```
public void log(Level level, java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg, java.lang.Object[] params)
```

Log a message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the given message level then a corresponding LogRecord is created and forwarded to all the registered output Handler objects.

Parameters:

`level` - One of the message level identifiers, e.g. SEVERE

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

log(Level, String, String, String, Throwable)

```
public void log(Level level, java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg, java.lang.Throwable thrown)
```

Log a message, specifying source class and method, with associated Throwable information.

If the logger is currently enabled for the given message level then the given arguments are stored in a LogRecord which is forwarded to all registered output handlers.

Note that the thrown argument is stored in the LogRecord thrown property, rather than the LogRecord parameters property. Thus is it processed specially by output Formatters and is not treated as a formatting parameter to the LogRecord message property.

Parameters:

`level` - One of the message level identifiers, e.g. SEVERE

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`thrown` - Throwable associated with log message.

log(Level, String, Throwable)

```
public void log(Level level, java.lang.String msg, java.lang.Throwable thrown)
```

Log a message, with associated Throwable information.

If the logger is currently enabled for the given message level then the given arguments are stored in a LogRecord which is forwarded to all registered output handlers.

Note that the thrown argument is stored in the LogRecord thrown property, rather than the LogRecord parameters property. Thus is it processed specially by output Formatters and is not treated as a formatting parameter to the LogRecord message property.

Parameters:

`level` - One of the message level identifiers, e.g. SEVERE

`msg` - The string message (or a key in the message catalog)

`thrown` - Throwable associated with log message.

log(LogRecord)

```
public void log(LogRecord record)
```

Log a LogRecord.

All the other logging methods in this class call through this method to actually perform any logging. Subclasses can override this single method to capture all log activity.

The current logger's name, resource bundle, and resource bundle name will be set into the LogRecord.

Parameters:

`record` - the LogRecord to be published

removeHandler(Handler)

```
public void removeHandler(Handler handler)
```

Remove a log Handler.

Returns silently if the given Handler is not found.

Parameters:
`handler` - a logging Handler

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

setFilter(Filter)

```
public void setFilter(Filter newFilter)
```

Set a filter to control output on this Logger.

After passing the initial "level" check, the Logger will call this Filter to check if a log record should really be published.

Parameters:
`newFilter` - a filter object (may be null)

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

setLevel(Level)

```
public void setLevel(Level newLevel)
```

Set the log level specifying which message levels will be logged by this logger. Message levels lower than this value will be discarded. The level value Level.OFF can be used to turn off logging.

Parameters:
`newLevel` - the new value for the log level

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

setUseGlobalHandlers(boolean)

```
public void setUseGlobalHandlers(boolean useGlobalHandlers)
```

Specify whether or not this logger should send its output to the set of global handlers managed by the LogManager.

Parameters:
`useGlobalhandlers` - true if output is to be sent to the LogManager's global handlers.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

severe(String)

```
public void severe(java.lang.String msg)
```

Log a SEVERE message.

If the logger is currently enabled for the SEVERE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:
`msg` - The string message (or a key in the message catalog)

severe(String, Object[])

```
public void severe(java.lang.String msg, java.lang.Object[] params)
```

Log a SEVERE message, with an array of object arguments.

If the logger is currently enabled for the SEVERE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

severe(String, String, String)

```
public void severe(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg)
```

Log a SEVERE message, specifying source class and method.

If the logger is currently enabled for the SEVERE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

severe(String, String, String, Object[])

```
public void severe(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg, java.lang.Object[] params)
```

Log a SEVERE message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the SEVERE message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

throwing(String, String, Throwable)

```
public void throwing(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.Throwable thrown)
```

Log throwing an exception.

This is a convenience method to log that a method is terminating by throwing an exception. The logging is done using the FINER level.

If the logger is currently enabled for the given message level then the given arguments are stored in a LogRecord which is forwarded to all registered output handlers. The LogRecord's message is set to "THROW".

Note that the thrown argument is stored in the LogRecord thrown property, rather than the LogRecord parameters property. Thus is it processed specially by output Formatters and is not treated as a formatting parameter to the LogRecord message property.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of the method.

`thrown` - The Throwable that is being thrown.

warning(String)

```
public void warning(java.lang.String msg)
```

Log a WARNING message.

If the logger is currently enabled for the WARNING message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

warning(String, Object[])

```
public void warning(java.lang.String msg, java.lang.Object[] params)
```

Log a WARNING message, with an array of object arguments.

If the logger is currently enabled for the WARNING message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

warning(String, String, String)

```
public void warning(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg)
```

Log a WARNING message, specifying source class and method.

If the logger is currently enabled for the WARNING message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

warning(String, String, String, Object[])

```
public void warning(java.lang.String sourceClass, java.lang.String sourceMethod,
            java.lang.String msg, java.lang.Object[] params)
```

Log a WARNING message, specifying source class and method, with an array of object arguments.

If the logger is currently enabled for the WARNING message level then the given message is forwarded to all the registered output Handler objects.

Parameters:

`sourceClass` - name of class that issued the logging request

`sourceMethod` - name of method that issued the logging request

`msg` - The string message (or a key in the message catalog)

`params` - array of parameters to the message

java.util.logging
LoggingPermission

Syntax

```
public final class LoggingPermission extends java.security.BasicPermission
```

```
java.lang.Object
  |
  +--java.security.Permission
        |
        +--java.security.BasicPermission
              |
              +--java.util.logging.LoggingPermission
```

All Implemented Interfaces: java.security.Guard, java.io.Serializable

Description

The permission which the SecurityManager will check when code that is running with a SecurityManager calls one of the logging control methods (such as LogManager.setLevel).

Currently there is only one named LoggingPermission. This is "control" and it grants the ability to control the logging configuration, for example by adding or removing Handlers, by adding or removing Filters, or by changing logging levels.

Programmers do not normally create LoggingPermission objects directly. Instead they are created by the security policy code based on reading the security policy file.

Since: 1.4

See Also: `java.security.BasicPermission`, `java.security.Permission`, `java.security.Permissions`, `java.security.PermissionCollection`, `java.lang.SecurityManager`

Member Summary	
Constructors	
`LoggingPermission(String, String)`	Creates a new LoggingPermission object.

Constructors

LoggingPermission(String, String)

```
public  LoggingPermission(java.lang.String name, java.lang.String actions)
```

Creates a new LoggingPermission object. This constructor exists for use by the Policy object to instantiate new Permission objects.

Parameters:

`name` - Permission name. Must be "control".

`actions` - Must be either null or the empty string.

java.util.logging
LogManager

Syntax

```
public class LogManager

java.lang.Object
  |
  +--java.util.logging.LogManager
```

Description

There is a single global LogManager object that is used to maintain a set of shared state about Loggers and log services.

This LogManager object:

- Manages a hierarchical namespace of Logger objects. All named Loggers are stored in this namespace.
- Manages a set of logging control properties. These are simple key-value pairs that can be used by Handlers and other logging objects to configure themselves.
- Manages a list of "global" logging handlers. By default Loggers will publish all (relevant) LogRecords to all the global handlers.

The global LogManager object can be retrieved using LogManager.getLogManager(). The LogManager object is created during class initialization and cannot subsequently be changed.

Note to reviewers: The main reason for having a LogManager object rather than simply a class with static methods is to allow container applications (especially J2EE containers) to replace the standard LogManager class with their own subclasses. We do not support on-the-fly replacement of the LogManager object as that would introduce tricky sychnronzation problems in propagating state between the old and the new objects. Making the LogManager object non-replaceable also allows client code to cache the object.

At startup the LogManager class is located using the java.util.logging.manager system property.

By default, the LogManager reads its initial configuration from a properties file "lib/logging.properties" in the JRE directory. If you edit that property file you can change the default logging configuration for all uses of that JRE.

In addition, the LogManager uses two optional system properties that allow more control over reading the initial configuration:

- "java.util.logging.config.class"
- "java.util.logging.config.file"

These two properties may be set via the Preferences API, or as command line property definitions to the "java" command, or as system property definitions passed to JNI_CreateJavaVM.

If the "java.util.logging.config.class" property is set, then the property value is treated as a class name. The given class will be loaded, an object will be instantiated, and that object's constructor is responsible for reading in the initial configuration. (That object may use other system properties to control its configuration.)

If "java.util.logging.config.class" property is **not** set, then the "java.util.logging.config.file" system property can be used to specify a properties file (in java.util.Properties format). The initial logging configuration will be read from this file.

If neither of these properties is defined then, as described above, the LogManager will read its initial configuration from a properties file "lib/logging.properties" in the JRE directory.

The properties for loggers and Handlers will have names starting with the dot-separated name for the handler or logger.

The global logging properties may include:

- A property "handlers". This defines a whitespace separated list of class names for handler classes to load and register as global handlers. Each class name must be for a Handler class which has a default constructor.
- A property "config". This property is intended to allow arbitrary configuration code to be run. The property defines a whitespace separated list of class names. A new instance will be created for each named class. The default constructor of each class may execute arbitrary code to update the logging configuration, such as setting logger levels, adding handlers, adding filters, etc.

Note that all classes loaded during LogManager configuration must be on the system class path. That includes the LogManager class, any config classes, and any handler classes.

Loggers are organized into a naming hierarchy based on their dot separated names. Thus "a.b.c" is a child of "a.b", but "a.b1" and a.b2" are peers.

All properties whose names end with ".level" are assumed to define log levels for Loggers. Thus "foo.level" defines a log level for the logger called "foo" and (recursively) for any of its children in the naming hierarchy. Log Levels are applied in the order they are defined in the properties file. Thus level settings for child nodes in the tree should come after settings for their parents. The property name ".level" can be used to set the level for the root of the tree.

All methods on the LogManager object are multi-thread safe.

Since: 1.4

Member Summary

Constructors

`LogManager()`	Protected constructor.

Methods

`addGlobalHandler(Handler)`	Add a global log Handler to receive output from all loggers.
`addLogger(Logger)`	Add a named logger.
`addPropertyChangeListener(PropertyChangeListener)`	Add an event listener to be invoked when the logging properties are re-read.
`checkAccess()`	Check that the current context is trusted to modify the logging configuration.
`flush()`	Flush all the global handlers.
`getGlobalHandlers()`	Get an array of all the global Handlers.
`getLevel(String)`	Get the log level for a given logger name.
`getLogger(String)`	Method to find a named logger.
`getLoggerNames()`	Get an enumeration of known logger names.
`getLogManager()`	Return the global LogManager object.
`getProperty(String)`	Get the value of a logging property.
`publish(LogRecord)`	Method to post a log record to all global handlers.
`readConfiguration()`	Reinitialize the logging properties and reread the logging configuration.
`readConfiguration(InputStream)`	Reinitialize the logging properties and reread the logging configuration from the given stream, which should be in java.util.Properties format.
`removeAllGlobalHandlers()`	Remove all global log handlers.
`removeGlobalHandler(Handler)`	Remove a global log handler.
`removePropertyChangeListener(PropertyChangeListener)`	Remove an event listener for property change events.

Member Summary	
setLevel(String, Level)	Set a log level for a given set of loggers.

Constructors

LogManager()

```
protected  LogManager()
```

Protected constructor. This is protected so that conatiner applications (such as J2EE containers) can subclass the object. It is non-public as it is intended that there only be one LogManager object, whose value is retrieved by calling Logmanager.getLogManager.

Methods

addGlobalHandler(Handler)

```
public void addGlobalHandler(Handler handler)
```

Add a global log Handler to receive output from all loggers.

Parameters:
`handler` - An output Handler

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

addLogger(Logger)

```
public boolean addLogger(Logger l)
```

Add a named logger. This does nothing and returns false if a logger with the same name is already registered.

The Logger factory methods call this method to register each newly created Logger.

The application must retain its own reference to the Logger object to avoid it being garbage collected. The LogManager will only retain a weak reference.

Parameters:
`l` - the new logger.

Returns: true if the argument logger was registered successfully, false if a logger of that name already exists.

Throws: `NullPointerException` - if the logger name is null.

addPropertyChangeListener(PropertyChangeListener)

```
public void addPropertyChangeListener(java.beans.PropertyChangeListener l)
```

Add an event listener to be invoked when the logging properties are re-read.

Parameters:

`l` - event listener

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

checkAccess()

```
public void checkAccess()
```

Check that the current context is trusted to modify the logging configuration. This requires LoggingPermission("control").

If the check fails we throw a SecurityException, otherwise we return normally.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

flush()

```
public void flush()
```

Flush all the global handlers.

getGlobalHandlers()

```
public Handler[] getGlobalHandlers()
```

Get an array of all the global Handlers.

Returns: array of global Handlers

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

getLevel(String)

```
public Level getLevel(java.lang.String name)
```

Get the log level for a given logger name. See the setLevel method for more details.

Parameters:
`name` - The logger name

Returns: the log level

getLogger(String)

```
public Logger getLogger(java.lang.String name)
```

Method to find a named logger.

Note that since untrusted code may create loggers with arbitrary names this method should not be relied on to find Loggers for security sensitive logging.

Parameters:
`name` - name of the logger

Returns: matching logger or null if none is found

getLoggerNames()

```
public java.util.Enumeration getLoggerNames()
```

Get an enumeration of known logger names.

Note: Loggers may be added dynamically as new classes are loaded. This method only reports on the loggers that are currently registered.

Returns: enumeration of logger name strings

getLogManager()

```
public static LogManager getLogManager()
```

Return the global LogManager object.

getProperty(String)

```
public java.lang.String getProperty(java.lang.String name)
```

Get the value of a logging property.

Parameters:
`name` - property name

Returns: property value

publish(LogRecord)

```
public void publish(LogRecord record)
```

Method to post a log record to all global handlers.

This is used by Loggers when they wish to publish a LogRecord to all the global handlers.

readConfiguration()

```
public void readConfiguration()
```

Reinitialize the logging properties and reread the logging configuration.

The same rules are used for locating the configuration properties as are used at startup. So normally the logging properties will be re-read from the same file that was used at startup.

All existing global Handlers will be closed and a new set of global Handlers will be created based on the new properties.

Any log level definitions in the new configuration file will be applied using LogManager.setLevel() in the order they are read.

A PropertyChangeEvent will be fired after the properties are read.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

readConfiguration(InputStream)

```
public void readConfiguration(java.io.InputStream ins)
```

Reinitialize the logging properties and reread the logging configuration from the given stream, which should be in java.util.Properties format. A PropertyChangeEvent will be fired after the properties are read.

All existing global Handlers will be closed and a new set of global Handlers will be created based on the new properties.

Any log level definitions in the new configuration file will be applied using LogManager.setLevel() in the order they are read.

Parameters:
`ins` - stream to read properties from

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

removeAllGlobalHandlers()

```
public void removeAllGlobalHandlers()
```

Remove all global log handlers.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

removeGlobalHandler(Handler)

```
public void removeGlobalHandler(Handler handler)
```

Remove a global log handler.

Returns silently if the given Handler is not found.

Parameters:
`handler` - An output Handler

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

removePropertyChangeListener(PropertyChangeListener)

```
public void removePropertyChangeListener(java.beans.PropertyChangeListener l)
```

Remove an event listener for property change events.

Returns silently if the given listener is not found.

Parameters:
`l` - event listener

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

setLevel(String, Level)

```
public void setLevel(java.lang.String name, Level level)
```

Set a log level for a given set of loggers. Subsequently the target loggers will only log messages whose types are greater than or equal to the given level. The level value Level.OFF can be used to turn off logging.

The given log level applies to the named Logger (if it exists), and on any other named Loggers below that name in the naming hierarchy. The name and level are recorded, and will be applied to any new Loggers that are later created matching the given name.

Thus, setting a level on "x.y" would affect loggers called "x.y", "x.y.z", "x.y.foo.bah" but would not affect loggers called "x.yy" or "x.z".

The empty string "" can be used to identify the root.

Parameters:

`name` - The logger name to update.

`level` - the new log level, e.g. Level.SEVERE or Level.FINER

Returns: the previous value of the level

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

java.util.logging
LogRecord

Syntax

```
public class LogRecord implements java.io.Serializable
```

```
java.lang.Object
  |
  +--java.util.logging.LogRecord
```

All Implemented Interfaces: java.io.Serializable

Description

LogRecord objects are used to pass logging requests between the logging framework and individual log Handlers.

When a LogRecord is passed into the logging framework it logically belongs to the framework and should no longer be used or updated by the client application.

Serialization notes:

- The LogRecord class is serializable.
- Because objects in the parameters array may not be serialiable, during serialization all objects in the parameters array are written as the corresponding Strings (using Object.toString).
- The ResourceBundle is not transmitted as part of the serialized form, but the resource bundle name is, and the recipient object's readObject method will atempt to locate a suitable resource bundle.
- The "thrown" property is transmitted as part of the serialized form, but because the Throwable class does not preserve stack frame information across serialization, the stack frame info will be lost from the "thrown" object. However, during serialization the writeObject method will ensure that a String version of the stack trace is serialized, and that String will be available through the getThrownStackTrace method.

Since: 1.4

Member Summary	
Constructors	
`LogRecord(Level, String)`	Construct a LogRecord with the given level and message values.
Methods	
`getLevel()`	Get the logging message level, for example Level.SEVERE.
`getLoggerName()`	Get the source Logger name's
`getMessage()`	Get the "raw" log message, before localization or formatting.
`getMillis()`	Get event time in milliseconds since 1970.
`getParameters()`	Get the parameters to the log message.
`getResourceBundle()`	Get the localization resource bundle
`getResourceBundleName()`	Get the localization resource bundle name
`getSequenceNumber()`	Get the sequence number.
`getSourceClassName()`	Get the name of the class that (allegedly) issued the logging request.

Member Summary	
getSourceMethodName()	Get the name of the method that (allegedly) issued the logging request.
getThreadID()	Get an identifier for the thread where the message originated.
getThrown()	Get any throwable associated with the log record.
getThrownBackTrace()	Get a backtrace for any thrown associated with the log record.
setLevel(Level)	Set the logging message level, for example Level.SEVERE.
setLoggerName(String)	Set the source Logger name.
setMessage(String)	Set the "raw" log message, before localization or formatting.
setMillis(long)	Set event time.
setParameters(Object[])	Set the parameters to the log message.
setResourceBundle(ResourceBundle)	Set the localization resource bundle.
setResourceBundleName(String)	Set the localization resource bundle name.
setSequenceNumber(long)	Set the sequence number.
setSourceClassName(String)	Set the name of the class that (allegedly) issued the logging request.
setSourceMethodName(String)	Set the name of the method that (allegedly) issued the logging request.
setThreadID(int)	Set an identifier for the thread where the message originated.
setThrown(Throwable)	Set a throwable associateded with the log event.

Constructors

LogRecord(Level, String)

```
public  LogRecord(Level level, java.lang.String msg)
```

Construct a LogRecord with the given level and message values.

The sequence property will be initialized with a new unique value. These sequence values are allocated in increasing order within a VM.

The millis property will be initialized to the current time.

The thread ID property wwill be initialized with a unique ID for the current thread.

All other properties will be initialized to "null".

Parameters:

`level` - a logging level value

`msg` - the raw non-localized logging message

Methods

getLevel()

```
public Level getLevel()
```

Get the logging message level, for example Level.SEVERE.

Returns: the logging message level

getLoggerName()

```
public java.lang.String getLoggerName()
```

Get the source Logger name's

Returns: source logger name (may be null)

getMessage()

```
public java.lang.String getMessage()
```

Get the "raw" log message, before localization or formatting.

May be null, which is equivalent to the empty string "".

This message may be either the final text or a localization key.

During formatting, if the source logger has a localization ResourceBundle and if that ResourceBundle has an entry for this message string, then the message string is replaced with the localized value.

Returns: the raw message string

getMillis()

```
public long getMillis()
```

Get event time in milliseconds since 1970.

Returns: event time in millis since 1970

getParameters()

```
public java.lang.Object[] getParameters()
```

Get the parameters to the log message.

Returns: the log message parameters. May be null if there are no parameters.

getResourceBundle()

```
public java.util.ResourceBundle getResourceBundle()
```

Get the localization resource bundle

This is the ResourceBundle that should be used to localize the message string before formatting it. The result may be null if the message is not localizable, or if no suitable ResourceBundle is available.

Parameters:

`bundle` - localization bundle (may be null)

getResourceBundleName()

```
public java.lang.String getResourceBundleName()
```

Get the localization resource bundle name

This is the name for the ResourceBundle that should be used to localize the message string before formatting it. The result may be null if the message is not localizable.

Parameters:
`bundle` - localization bundle name (may be null)

getSequenceNumber()

```
public long getSequenceNumber()
```

Get the sequence number.

Sequence numbers are normally assigned in the LogRecord constructor, which assignes unique sequence numbers to each new LogRecord in increasing order.

Returns: the sequence number

getSourceClassName()

```
public java.lang.String getSourceClassName()
```

Get the name of the class that (allegedly) issued the logging request.

Note that this sourceClassName is not verified and may be spoofed. This information may either have been provided as part of the logging call, or it may have been inferred automatically by the logging framework. In the latter case, the information may only be approximate and may in fact describe an earlier call on the stack frame.

May be null if no information could be obtained.

Returns: the source class name

getSourceMethodName()

```
public java.lang.String getSourceMethodName()
```

Get the name of the method that (allegedly) issued the logging request.

Note that this sourceMethodName is not verified and may be spoofed. This information may either have been provided as part of the logging call, or it may have been inferred automatically by the logging framework. In the latter case, the information may only be approximate and may in fact describe an earlier call on the stack frame.

May be null if no information could be obtained.

Returns: the source method name

getThreadID()

```
public int getThreadID()
```

Get an identifier for the thread where the message originated.

This is a thread identifier within the Java VM and may or may not map to any operating system ID.

Returns: thread ID

getThrown()

```
public java.lang.Throwable getThrown()
```

Get any throwable associated with the log record.

If the event involved an exception, this will be the exception object. Otherwise null.

Returns: a throwable

getThrownBackTrace()

```
public java.lang.String getThrownBackTrace()
```

Get a backtrace for any thrown associated with the log record.

If the event involved an exception, this will be the the backtrace from the exception object. Otherwise it will return null. The exact backtrace format is implementation specific and will vary beween VMs.

Note that this method is preferrable to obtaining a backtrace from the "getThrown()" Throwable, as the backtrace may have been lost from that object if it was serialized and then deserialized.

setLevel(Level)

```
public void setLevel(Level level)
```

Set the logging message level, for example Level.SEVERE.

Parameters:
`level` - the logging message level

setLoggerName(String)

```
public void setLoggerName(java.lang.String name)
```

Set the source Logger name.

Parameters:
`name` - the source logger name (may be null)

setMessage(String)

```
public void setMessage(java.lang.String message)
```

Set the "raw" log message, before localization or formatting.

Parameters:
`message` - the raw message string

setMillis(long)

```
public void setMillis(long millis)
```

Set event time.

Parameters:
`millis` - event time in millis since 1970

setParameters(Object[])

```
public void setParameters(java.lang.Object[] parameters)
```

Set the parameters to the log message.

Parameters:
`parameters` - the log message parameters.

setResourceBundle(ResourceBundle)

```
public void setResourceBundle(java.util.ResourceBundle bundle)
```

Set the localization resource bundle.

Parameters:

`bundle` - localization bundle (may be null)

setResourceBundleName(String)

```
public void setResourceBundleName(java.lang.String name)
```

Set the localization resource bundle name.

Parameters:

`name` - localization bundle name (may be null)

setSequenceNumber(long)

```
public void setSequenceNumber(long seq)
```

Set the sequence number.

Sequence numbers are normally assigned in the LogRecord constructor, so it should not normally be necessary to use this method.

setSourceClassName(String)

```
public void setSourceClassName(java.lang.String sourceClassName)
```

Set the name of the class that (allegedly) issued the logging request.

Parameters:

`sourceClassName` - the source class name

setSourceMethodName(String)

```
public void setSourceMethodName(java.lang.String sourceMethodName)
```

Set the name of the method that (allegedly) issued the logging request.

Parameters:

`sourceMethodName` - the source method name

setThreadID(int)

```
public void setThreadID(int threadID)
```

Set an identifier for the thread where the message originated.

Parameters:

`threadID` - the thread ID

setThrown(Throwable)

```
public void setThrown(java.lang.Throwable thrown)
```

Set a throwable associateded with the log event.

Parameters:

`throwable` - a throwable

java.util.logging
MemoryHandler

Syntax

```
public class MemoryHandler extends java.util.logging.Handler
```

```
java.lang.Object
  |
  +--java.util.logging.Handler
        |
        +--java.util.logging.MemoryHandler
```

Description

Handler that buffers requests in a circular buffer in memory.

Normally this Handler simply stores incoming LogRecords into its memory buffer and discards earlier records. This buffering is very cheap and avoids formatting costs. On certain trigger conditions, the MemoryHandler will push out its current buffer contents to a target Handler, which will typically publish them to the outside world.

There are three main models for triggering a push of the buffer:

- An incoming LogRecord has a type that is greater than a pre-defined level, the "pushLevel".
- An external class calls the "push" method explicitly.
- A subclass overrides the "log" method and scans each incoming LogRecord and calls "push" if a record matches some desired criteria.

Configuration: By default each MemoryHandler is initialized using the following LogManager configuration properties. If properties are not defined (or have invalid values) then the specified default values are used.

- java.util.logging.MemoryHandler.level specifies the level for the Handler (defaults to Level.ALL).
- java.util.logging.MemoryHandler.filter specifies the name of a Filter class to use (defaults to no Filter).
- java.util.logging.MemoryHandler.size defines the buffer size (defaults to 1000).
- java.util.logging.MemoryHandler.push defines the pushLevel (defaults to level.SEVERE).

Since: 1.4

Member Summary	
Constructors	
MemoryHandler()	Create a MemoryHandler and configure it based on LogManager configuration properties.
MemoryHandler(Handler, int, Level)	Create a MemoryHandler.
Methods	
close()	Close the Handler and free all associated resources.
flush()	Causes a flush on the target handler.
getPushLevel()	Get the pushLevel.
isLoggable(LogRecord)	Check if this Handler would actually log a given LogRecord into its internal buffer.
publish(LogRecord)	Store a LogRecord in an internal buffer.
push()	Push any buffered output to the target Handler.

Member Summary	
`setPushLevel(Level)`	Set the pushLevel.

Constructors

MemoryHandler()

```
public  MemoryHandler()
```

Create a MemoryHandler and configure it based on LogManager configuration properties.

MemoryHandler(Handler, int, Level)

```
public  MemoryHandler(Handler target, int size, Level pushLevel)
```

Create a MemoryHandler.

The MemoryHandler is configured based on LogManager properties (or their default values) except that the given pushLevel argument and buffer size argument are used.

Parameters:

`target` - the Handler to which to publish output.

`size` - the number of log records to buffer

`pushLevel` - message level to push on

Methods

close()

```
public void close()
```

Close the Handler and free all associated resources. This will also close the target Handler.

Overrides: `Handler.close()` in class `Handler`

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

flush()

```
public void flush()
```

Causes a flush on the target handler.

Note that the current contents of the MemoryHandler buffer are **not** written out. That requires a "push".

Overrides: `Handler.flush()` in class `Handler`

getPushLevel()

```
public Level getPushLevel()
```

Get the pushLevel.

Parameters:
`the` - value of the pushLevel

isLoggable(LogRecord)

```
public boolean isLoggable(LogRecord record)
```

Check if this Handler would actually log a given LogRecord into its internal buffer.

This method checks if the LogRecord has an appropriate level and whether it satisfies any Filter. However it does **not** check whether the LogRecord would result in a "push" of the buffer contents.

Overrides: `Handler.isLoggable(LogRecord)` in class `Handler`

Parameters:
`record` - a LogRecord

Returns: true if the log record would be logged.

publish(LogRecord)

```
public void publish(LogRecord record)
```

Store a LogRecord in an internal buffer.

If there is a filter, its isLoggable method is called to check if the given log record isLoggable. If not we return. Otherwise the given record is copied into an internal circular buffer. Then the record's type field is compared with the pushLevel. If the given level is greater than or equal to the pushLevel then "push" is called to write all buffered records to the target output Handler.

Overrides: `Handler.publish(LogRecord)` in class `Handler`

Parameters:
`record` - description of the log event

push()

```
public void push()
```

Push any buffered output to the target Handler.

The buffer is then cleared.

setPushLevel(Level)

```
public void setPushLevel(Level newLevel)
```

Set the pushLevel. After a log record is copied into our internal buffer, if its level is greater than or equal to the pushLevel, then "push" will be called.

Parameters:
`newLevel` - the new value of the pushLevel

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

java.util.logging
SimpleFormatter

Syntax
```
public class SimpleFormatter extends java.util.logging.Formatter
```

```
java.lang.Object
  |
  +--java.util.logging.Formatter
        |
        +--java.util.logging.SimpleFormatter
```

Description
Print a brief summary of the LogRecord in a human readable format. The summary will typically be 1 or 2 lines.

Since: 1.4

Member Summary	
Constructors	
`SimpleFormatter()`	
Methods	
`format(LogRecord)`	Format the given LogRecord.

Constructors

SimpleFormatter()
```
public  SimpleFormatter()
```

Methods

format(LogRecord)
```
public java.lang.String format(LogRecord record)
```

Format the given LogRecord.

Overrides: `Formatter.format(LogRecord)` in class `Formatter`

Parameters:
`record` - the log record to be formatted.

Returns: a formatted log record

java.util.logging
SocketHandler

Syntax

```
public class SocketHandler extends java.util.logging.StreamHandler
```

```
java.lang.Object
  |
  +--java.util.logging.Handler
        |
        +--java.util.logging.StreamHandler
              |
              +--java.util.logging.SocketHandler
```

Description

Simple network logging handler.

LogRecords are published to a network stream connection. By default the XMLFormatter class is used for formatting.

Configuration: By default each SocketHandler is initialized using the following LogManager configuration properties. If properties are not defined (or have invalid values) then the specified default values are used.

- java.util.logging.SocketHandler.level specifies the default level for the Handler (defaults to Level.ALL).
- java.util.logging.SocketHandler.filter specifies the name of a Filter class to use (defaults to no Filter).
- java.util.logging.SocketHandler.formatter specifies the name of a Formatter class to use (defaults to java.util.logging.XMLFormatter).
- java.util.logging.SocketHandler.encoding the name of the character set encoding to use (defaults to the default platform encoding).
- java.util.logging.SocketHandler.host specifies the target host name to connect to (no default).
- java.util.logging.SocketHandler.port specifies the target TCP port to use (no default).

The output IO stream is buffered, but is flushed after each log record.

Since: 1.4

Member Summary	
Constructors	
SocketHandler()	Create a SocketHandler, using only LogManager properties (or their defaults).
SocketHandler(String, int)	Construct a SocketHandler using a specified host and port.
Methods	
close()	Close this output stream.
publish(LogRecord)	Format and publish a LogRecord.

Constructors

SocketHandler()

```
public  SocketHandler()
```

Create a SocketHandler, using only LogManager properties (or their defaults).

SocketHandler(String, int)

```
public  SocketHandler(java.lang.String host, int port)
```

Construct a SocketHandler using a specified host and port. The SocketHandler is configured based on LogManager properties (or their default values) except that the given target host and port arguments are used.

Parameters:

hostName - target host.

`port` - target port.

Methods

close()

```
public void close()
```

Close this output stream.

Overrides: `StreamHandler.close()` in class `StreamHandler`

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

publish(LogRecord)

```
public void publish(LogRecord record)
```

Format and publish a LogRecord.

Overrides: `StreamHandler.publish(LogRecord)` in class `StreamHandler`

Parameters:

`record` - description of the log event

java.util.logging
StreamHandler

Syntax

```
public class StreamHandler extends java.util.logging.Handler
```

```
java.lang.Object
  |
  +--java.util.logging.Handler
        |
        +--java.util.logging.StreamHandler
```

Direct Known Subclasses: `ConsoleHandler`, `FileHandler`, `SocketHandler`

Description

Stream based logging Handler.

This is primarily intended as a base class or support class to be used in implementing other logging Handlers.

LogRecords are published to a given java.io.OutputStream.

Configuration: By default each StreamHandler is initialized using the following LogManager configuration properties. If properties are not defined (or have invalid values) then the specified default values are used.

- java.util.logging.StreamHandler.level specifies the default level for the Handler (defaults to Level.ALL).
- java.util.logging.StreamHandler.filter specifies the name of a Filter class to use (defaults to no Filter).
- java.util.logging.StreamHandler.formatter specifies the name of a Formatter class to use (defaults to java.util.logging.SimpleFormatter).
- java.util.logging.StreamHandler.encoding the name of the character set encoding to use (defaults to the default platform encoding).

Since: 1.4

Member Summary	
Constructors	
`StreamHandler()`	Create a StreamHandler, with no current output stream.
`StreamHandler(OutputStream, Formatter)`	Create a StreamHandler with a given formatter and output stream.
Methods	
`close()`	Close the current output stream.
`flush()`	Flush any buffered messages.
`isLoggable(LogRecord)`	Check if this Handler would actually log a given LogRecord.
`publish(LogRecord)`	Format and publish a log record.
`setEncoding(String)`	Set (or change) the character encoding used by this Handler.
`setOutputStream(OutputStream)`	Change the output stream.

Constructors

StreamHandler()

```
public  StreamHandler()
```

Create a StreamHandler, with no current output stream.

StreamHandler(OutputStream, Formatter)

```
public  StreamHandler(java.io.OutputStream out, Formatter formatter)
```

Create a StreamHandler with a given formatter and output stream.

Parameters:

`output` - the target output stream

`formatter` - Formatter to be used to format output

Methods

close()

```
public void close()
```

Close the current output stream.

The Formatter's "tail" string is written to the stream before it is closed. In addition, if the formatter's "head" string has not yet been written to the stream, it will be written before the "tail" string.

Overrides: `Handler.close()` in class `Handler`

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

`SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

flush()

```
public void flush()
```

Flush any buffered messages.

Overrides: `Handler.flush()` in class `Handler`

isLoggable(LogRecord)

```
public boolean isLoggable(LogRecord record)
```

Check if this Handler would actually log a given LogRecord.

This method checks if the LogRecord has an appropriate level and whether it satisfies any Filter. It will also return false if no output stream has been assigned yet.

Overrides: `Handler.isLoggable(LogRecord)` in class `Handler`

Parameters:
`record` - a LogRecord

Returns: true if the log record would be logged.

publish(LogRecord)

```
public void publish(LogRecord record)
```

Format and publish a log record.

The StreamHandler first checks if there is an OutputStream and if the given LogRecord has at least the required log level. If not it silently returns. If so, it calls any associated Filter to check if the record should be published. If so, it calls its Formatter to format the record and then writes the result to the current output stream.

If this is the first LogRecord to be written to a given OutputStream, the Formatter's "head" string is written to the stream before the LogRecord is written.

Overrides: `Handler.publish(LogRecord)` in class `Handler`

Parameters:
`record` - description of the log event

setEncoding(String)

```
public void setEncoding(java.lang.String encoding)
```

Set (or change) the character encoding used by this Handler.

The encoding should be set before any LogRecords are written to the Handler.

Overrides: `Handler.setEncoding(String)` in class `Handler`

Parameters:
`encoding` - The name of a supported character encoding. May be null, to indicate the default platform encoding.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

`UnsupportedEncodingException` - if the named encoding is not supported.

setOutputStream(OutputStream)

```
protected void setOutputStream(java.io.OutputStream out)
```

Change the output stream.

If there is a current output stream then the formatter's tail string is written and the stream is flushed and closed. Then the output stream is replaced with the new output stream.

Parameters:
`out` - New output stream. May not be null.

Throws: `SecurityException` - if a security manager exists and if the caller does not have LoggingPermission("control").

java.util.logging
XMLFormatter

Syntax

```
public class XMLFormatter extends java.util.logging.Formatter
```

```
java.lang.Object
  |
  +--java.util.logging.Formatter
        |
        +--java.util.logging.XMLFormatter
```

Description

Format a LogRecord into a standard XML format.

The DTD specification is provided as Appendix A to the Java Logging APIs specification.

The XMLFormatter can be used with arbitrary character encodings, but it is recommended that it normally be used with UTF-8. The character encoding can be set on the output Handler.

Since: 1.4

Member Summary	
Constructors	
XMLFormatter()	
Methods	
format(LogRecord)	Format the given message to XML.
getHead(Handler)	Return the header string for a set of XML formatted records.
getTail(Handler)	Return the tail string for a set of XML formatted records.

Constructors

XMLFormatter()

```
public  XMLFormatter()
```

Methods

format(LogRecord)

```
public java.lang.String format(LogRecord record)
```

Format the given message to XML.

Overrides: `Formatter.format(LogRecord)` in class `Formatter`

Parameters:
`record` - the log record to be formatted.

Returns: a formatted log record

getHead(Handler)

```
public java.lang.String getHead(Handler h)
```

Return the header string for a set of XML formatted records.

Overrides: `Formatter.getHead(Handler)` in class `Formatter`

Parameters:
`h` - The target handler.

Returns: header string

getTail(Handler)

```
public java.lang.String getTail(Handler h)
```

Return the tail string for a set of XML formatted records.

Overrides: `Formatter.getTail(Handler)` in class `Formatter`

Parameters:
`h` - The target handler.

Returns: tail string

4 Change History

4.1 Changes between 0.50 and 0.55:

- The ConsoleHandler, FileHandler, MemoryHandler, SocketHandler, and StreamHandler classes now use a LogManager “filter” property to locate a default Filter class to act as a Filter on the Handler.
- The ConsoleHandler, FileHandler, SocketHandler, and StreamHandler classes now use a LogManager “formatter” property to locate a default Formatter class to act as the Formatter on the Handler.
- The ConsoleHandler, FileHandler, SocketHandler, and StreamHandler classes now use a LogManager “encoding” property to determine a default character set encoding for the Handler.
- The javadoc in the class header for each of the Handler classes has been revised to be more consistent in describing LogManager properties and to be clearer on what default values are used if a property is not defined (or is not valid).
- The javadoc in each of the concrete Handler classes (in both the class header and in the constructors) has been revised to specify that the LogManager properties (or their defaults) are used as the default configuration for each Handler, unless explicitly specified otherwise in a constructor.
- Removed the (false) assumption that XMLFormatter should always use UTF-8. I have been reassured that it is OK to use any encoding (including the platform default encoding) provided the encoding is specified in the XML header, which we do.
- The following constructors were removed. Most of them had originally been added in response to a request to provide more consistent constructors. However, now that a richer set of properties can be defined in the LogManager configuration, this particular set of constructors no longer makes particular sense. For example it is unlikely that someone will create a SocketHandler and want to use the default properties except for Formatter. People are likely to either want a default handler configuration, or to override wider sets of the configuration properties (especially the properties that control where output is sent).

  ConsoleHandler(Formatter)
  FileHandler(Formatter)
  SocketHandler(Formatter)
  StreamHandler(Formatter)
- Added Section 2.19 on J2EE issues. I reviewed this with the spec lead for J2EE 1.3.
- Various minor clarifications (generally in response to specification bugs filed by the test team).

Appendix A: DTD for XMLFormatter output

```
<!-- DTD used by the java.util.logging.XMLFormatter -->
<!-- This provides an XML formatted log message.   -->

<!-- The document type is "log" which consists of a sequence
    of record elements -->
<!ELEMENT log (record*)>

<!-- Each logging call is described by a record element. -->
<!ELEMENT record (date, millis, sequence, logger?, level,
class?, method?, thread?, message, key?, catalog?, param*, exception?)>

<!-- Date and time when LogRecord was created in ISO 8601 format -->
<!ELEMENT date (#PCDATA)>

<!-- Time when LogRecord was created in milliseconds since
    midnight January 1st, 1970, UTC.  -->
<!ELEMENT millis (#PCDATA)>

<!-- Unique sequence number within source VM.  -->
<!ELEMENT sequence (#PCDATA)>

<!-- Name of source Logger object.  -->
<!ELEMENT logger (#PCDATA)>

<!-- Logging level, may be either one of the constant
    names from java.util.logging.Constants (such as "SEVERE"
    or "WARNING") or an integer value such as "20".  -->
<!ELEMENT level (#PCDATA)>


<!-- Fully qualified name of class that issued
    logging call, e.g. "javax.marsupial.Wombat". -->
<!ELEMENT class (#PCDATA)>

<!-- Name of method that issued logging call.
    It may be either an unqualified method name such as
    "fred" or it may include argument type information
    in parenthesis, for example "fred(int,String)".  -->
<!ELEMENT method (#PCDATA)>

<!-- Integer thread ID. -->
<!ELEMENT thread (#PCDATA)>
```

```
<!-- The message element contains the text string of a log message. -->
<!ELEMENT message (#PCDATA)>

<!-- If the message string was localized, the key element provides
     the original localization message key. -->
<!ELEMENT key (#PCDATA)>

<!-- If the message string was localized, the catalog element provides
     the logger's localization resource bundle name. -->
<!ELEMENT catalog (#PCDATA)>

<!-- If the message string was localized, each of the param elements
     provides the String value (obtained using Object.toString())
     of the corresponding LogRecord parameter. -->
<!ELEMENT param (#PCDATA)>

<!-- An exception consists of an optional message string followed
     by a series of StackFrames.  Exception elements are used
     for Java exceptions and other java Throwables.  -->
<!ELEMENT exception (message?, frame+)>

<!-- A frame describes one line in a Throwable backtrace. -->
<!ELEMENT frame (class, method, line?)>

<!-- an integer line number within a class's source file. -->
<!ELEMENT line (#PCDATA)>
```