Java Logging

Sang Shin
Java Technology Architect
Sun Microsystems, Inc.
sang.shin@sun.com
www.javapassion.com

Topics

- What is and Why Java logging?
- Architecture of Java logging framework
- Logging example
- Logging levels
- Loggers
- Handlers
- Formatters
- Configuration
- Logging and performance

What is & Why Java Logging API?

What is a Java Logging API?

- Introduced in package *java.util.logging*
- The core package includes support for delivering plain text or XML-formatted log records to memory, output streams, consoles, files, and sockets. In addition, the logging APIs are capable of interacting with logging services that already exist on the host operating system.

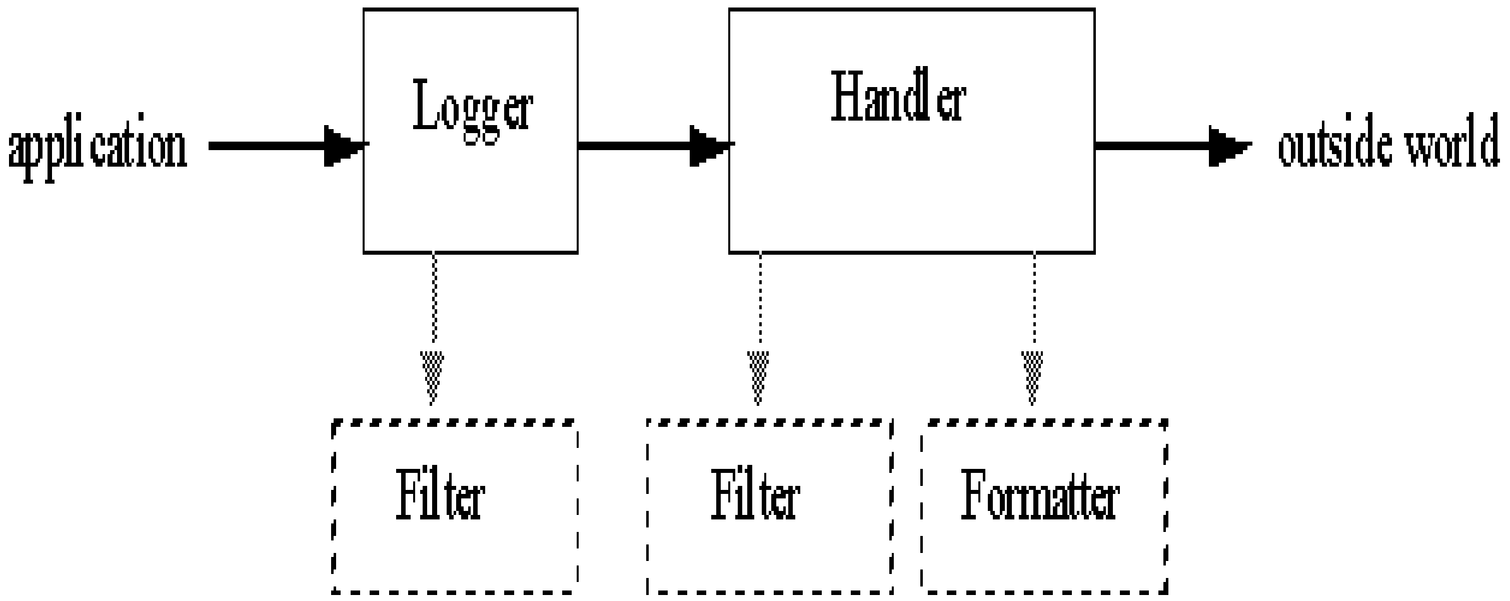
Why Use Java Logging API?

- Facilitate software servicing and maintenance at customer sites by producing log reports suitable for analysis by end users, system administrators, field service engineers, and software development teams
  - > Capture information such as security failures, configuration errors, performance bottlenecks, and/or bugs in the application or platform

Architecture of Java Logging Framework

Logger and Handler

- Applications make logging calls on *Logger* objects.
- The Logger objects allocate *LogRecord* objects which are passed to *Handler* objects for publication.



Filter and Formatter

- Both Loggers and Handlers may use (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

Logging Example First

Example

```
package com.wombat;

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");
    }
}
```

Logging Levels

Logging Levels

- The Logging 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
- The Level class defines seven standard log levels
  - FINEST (lowest priority)
  - SEVERE (highest priority)

Loggers

Logger

- When client code sends log requests 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.

Handlers

Handlers

- 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 that writes formatted log records either to a single file, or to a set of rotating log files.
- SocketHandler
  - > A handler that writes formatted log records to remote TCP ports.
- MemoryHandler
  - > A handler that buffers log records in memory.

Set up its own Logging Handler

```
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");
    }
}
```

Logging Methods

Logging Methods

- The Logger class provides a large set of convenience methods for generating log messages
- Two different styles of logging methods
  - > void warning(String sourceClass, String sourceMethod, String msg);
  - > void warning(String msg);

Formatters

Two Standard Formaters

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

Sample XML Output

```
<?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>
```

LogManager

LogManager

- There is a global *LogManager* object that keeps track of global logging information
  - A hierarchical namespace of named Loggers
  - A set of logging control properties read from the configuration file
- A LogManager object can be retrieved using the static *LogManager.getLogManager()* method
- LogManager object 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.

Configuration File

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
- 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.

Changing Configuration

```
// Dynamically adjust the logging configuration to send output
// to a specific file and to get lots of information on wombats

public static void main(String[] args){
    Handler fh = new FileHandler("%t/wombat.log");
    Logger.getLogger("").addHandler(fh);
    Logger.getLogger("com.wombat").setLevel("com.wombat",Level.FINEST);
    ...
}
```

Logging & Performance Implication

Logging can be disabled

- 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.

Java Logging

Sang Shin
Java Technology Architect
Sun Microsystems, Inc.
sang.shin@sun.com
www.javapassion.com