Java Logging

Sang Shin
Founder & Chief Instructor
www.JPassion.com
"Learn with JPassion!"

Topics

- What is and Why Java logging?
- Architecture of Java logging framework
- Logging example
- Logging levels
- Handlers
- Formatters
- Filters
- Logger hierarchy
- LogManager
- 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 a variety of output streams: memory, output streams, consoles, files, and sockets.

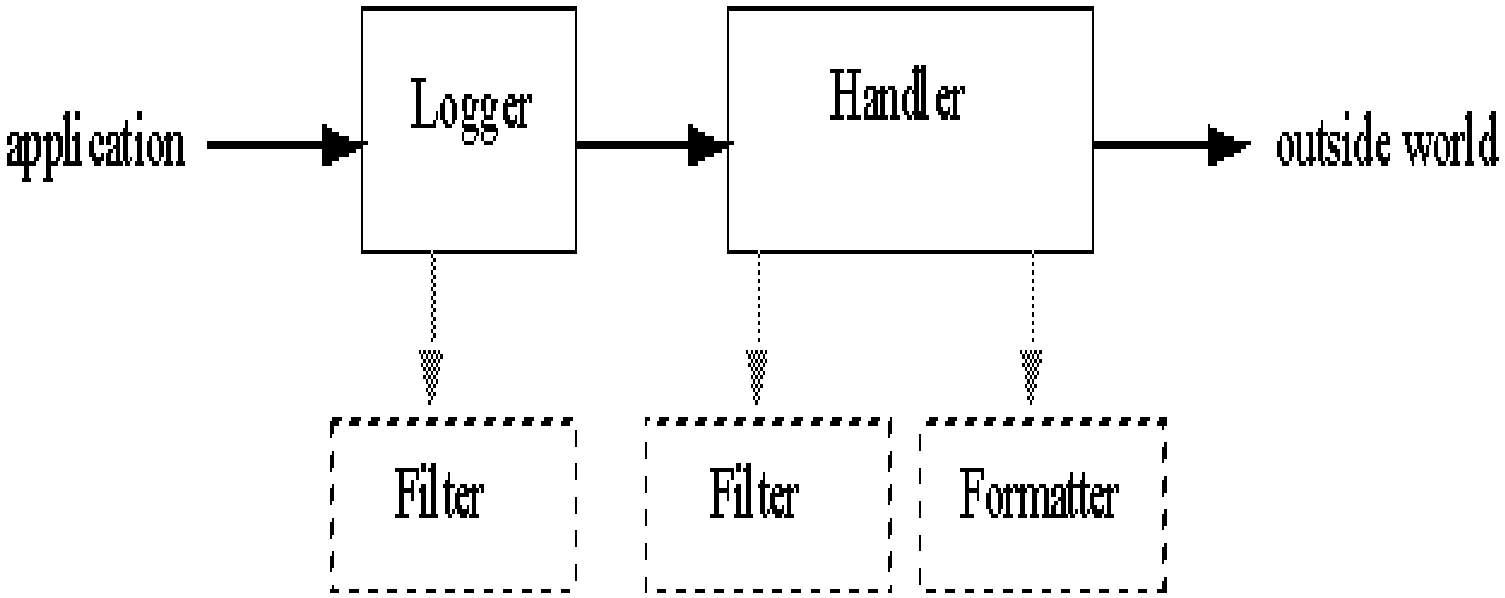
Why Use Java Logging API?

- During development
  - Could be used as debugging tool
- After deployment
  - 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 particular set of LogRecord's
- 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

Logging Example Code

```
package com.wombat;

public class Nose{
    // Obtain a suitable logger for the named subsystem.
    // If a logger has already been created with the given name it is returned.
    // Otherwise a new logger is created.
    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 warning message
            logger.log(Level.WARNING,"trouble sneezing",ex);
        }
        logger.fine("done");
    }
}
```

Lab:

Exercise 1.1: Simple Logging
1019_javase_logging.zip

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)
- Setting Log level
  - Default logging level could be configured based on the *LogManager* configuration
  - *Logger.setLevel(Level.SEVERE)*

How Does Logger Handle Logging Level?

- 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.

Lab:

Exercise 1.2: Logging Level
1019_javase_logging.zip

Handlers

Handler and Formatter

- Handler receives log message from the logger and is responsible for publication of it
- Each handlers output can be configured with a formatter

Types of 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.

Setting up a Handler for a Logger

```
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.xml");
    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");
    }
}
```

Lab:

Exercise 2: Log to File & Multiple Handlers
1019_javase_logging.zip

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

Formatters

- 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.
- Ready-to-use formatter
  - SimpleFormatter
    - Writes brief "human-readable" summaries of log records.
  - XMLFormatter
    - Writes detailed XML-structured information
- Custom formatter
  - Implement *String format(LogRecord record)* method

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>
```

Lab:

Exercise 3: Fomatter
1019_javase_logging.zip

Filters

Filter

- 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.

Lab:

Exercise 4: Filter
1019_javase_logging.zip

Logger Hierarchy

Logger Hierarchy

- Loggers can have parent-child (hierarchical) relationship
  - The parent's logging level is inherited by the child
- The relationship is through the logger name
  - Parent logger
    - private Logger logger = Logger.getLogger("sam.logging");
  - Child logger
    - private Logger logger = Logger.getLogger("sam.logging.child");

Lab:

Exercise 5: Hierarchical Logging
1019_javase_logging.zip

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.

Demo:

LogManagerExample
1019_javase_logging.zip

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 During Runtime

```
// 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.

Learn with Passion!
JPassion.com