Short introduction to log4j

Ceki Gülcü March 2002

 The present short manual also borrows some text from *"The complete log4j manual"*, which contains more detailed and up to date information, by the same author (yours truly).

Abstract

This document describes the log4j API, its unique features and design rationale. Log4j is an open source project based on the work of many authors. It allows the developer to control which log statements are output with arbitrary granularity. It is fully configurable at runtime using external configuration files. Best of all, log4j has a gentle learning curve. Beware: judging from user feedback, it is also quite addictive.

Introduction

Almost every large application includes its own logging or tracing API. In conformance with this rule, the E.U. SEMPER project decided to write its own tracing API. This was in early 1996. After countless enhancements, several incarnations and much work that API has evolved to become log4j, a popular logging package for Java. The package is distributed under the Apache Software License, a fully-fledged open source license certified by the open source initiative. The latest log4j version, including full-source code, class files and documentation can be found at **http://logging.apache.org/log4j/**. By the way, log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages.

Inserting log statements into code is a low-tech method for debugging it. It may also be the only way because debuggers are not always available or applicable. This is usually the case for multithreaded applications and distributed applications at large.

Experience indicates that logging was an important component of the development cycle. It offers several advantages. It provides precise *context* about a run of the application. Once inserted into the code, the generation of logging output requires no human intervention. Moreover, log output can be saved in a persistent medium to be studied at a later time. In addition to its use in the development cycle, a sufficiently-rich logging package can also be viewed as an auditing tool.

As Brian W. Kernighan and Rob Pike put it in their truly excellent book *"The Practice of Programming"*

```
As personal choice, we tend not to use debuggers beyond getting a
stack trace or the value of a variable or two. One reason is that it
is easy to get lost in details of complicated data structures and
control flow; we find stepping through a program less productive
than thinking harder and adding output statements and self-checking
```

```
code at critical places. Clicking over statements takes longer than
scanning the output of judiciously-placed displays. It takes less
time to decide where to put print statements than to single-step to
the critical section of code, even assuming we know where that
is. More important, debugging statements stay with the program;
debugging sessions are transient.
```

Logging does have its drawbacks. It can slow down an application. If too verbose, it can cause scrolling blindness. To alleviate these concerns, log4j is designed to be reliable, fast and extensible. Since logging is rarely the main focus of an application, the log4j API strives to be simple to understand and to use.

Loggers, Appenders and Layouts

Log4j has three main components: *loggers*, *appenders* and *layouts*. These three types of components work together to enable developers to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported.

Logger hierarchy

The first and foremost advantage of any logging API over plain `System.out.println` resides in its ability to disable certain log statements while allowing others to print unhindered. This capability assumes that the logging space, that is, the space of all possible logging statements, is categorized according to some developer-chosen criteria. This observation had previously led us to choose *category* as the central concept of the package. However, since log4j version 1.2, `Logger` class has replaced the `Category` class. For those familiar with earlier versions of log4j, the `Logger` class can be considered as a mere alias to the `Category` class.

Loggers are named entities. Logger names are case-sensitive and they follow the hierarchical naming rule:

Named Hierarchy

> A logger is said to be an *ancestor* of another logger if its name followed by a dot is a prefix of the *descendant* logger name. A logger is said to be a *parent* of a *child* logger if there are no ancestors between itself and the descendant logger.

For example, the logger named `"com.foo"` is a parent of the logger named `"com.foo.Bar"`. Similarly, `"java"` is a parent of `"java.util"` and an ancestor of `"java.util.Vector"`. This naming scheme should be familiar to most developers.

The root logger resides at the top of the logger hierarchy. It is exceptional in two ways:

1. it always exists,
2. it cannot be retrieved by name.

Invoking the class static Logger.getRootLogger method retrieves it. All other loggers are instantiated and retrieved with the class static Logger.getLogger method. This method takes the name of the desired logger as a parameter. Some of the basic methods in the Logger class are listed below.

```
  package org.apache.log4j;

  public class Logger {

    // Creation & retrieval methods:
    public static Logger getRootLogger();
    public static Logger getLogger(String name);

    // printing methods:
    public void debug(Object message);
    public void info(Object message);
    public void warn(Object message);
    public void error(Object message);
    public void fatal(Object message);

    // generic printing method:
    public void log(Level l, Object message);
}
```

Loggers *may* be assigned levels. The set of possible levels, that is DEBUG, INFO, WARN, ERROR and FATAL are defined in the `org.apache.log4j.Level` class. Although we do not encourage you to do so, you may define your own levels by sub-classing the `Level` class. A perhaps better approach will be explained later on.

If a given logger is not assigned a level, then it inherits one from its closest ancestor with an assigned level. More formally:

Level Inheritance

> The *inherited level* for a given logger *C*, is equal to the first non-null level in the logger hierarchy, starting at *C* and proceeding upwards in the hierarchy towards the `root` logger.

To ensure that all loggers can eventually inherit a level, the root logger always has an assigned level.

Below are four tables with various assigned level values and the resulting inherited levels according to the above rule.

Logger name	Assigned level	Inherited level
root	Proot	Proot
X	none	Proot
X.Y	none	Proot
X.Y.Z	none	Proot

Example 1

In example 1 above, only the root logger is assigned a level. This level value, `Proot`, is inherited by the other loggers `X`, `X.Y` and `X.Y.Z`.

Logger name	Assigned level	Inherited level
root	Proot	Proot
X	Px	Px
X.Y	Pxy	Pxy
X.Y.Z	Pxyz	Pxyz

Example 2

In example 2, all loggers have an assigned level value. There is no need for level inheritence.

Logger name	Assigned level	Inherited level
root	Proot	Proot
X	Px	Px
X.Y	none	Px
X.Y.Z	Pxyz	Pxyz

Example 3

In example 3, the loggers `root`, `X` and `X.Y.Z` are assigned the levels `Proot`, `Px` and `Pxyz` respectively. The logger `X.Y` inherits its level value from its parent `X`.

Logger name	Assigned level	Inherited level
root	Proot	Proot
X	Px	Px
X.Y	none	Px
X.Y.Z	none	Px

Example 4

In example 4, the loggers `root` and `X` and are assigned the levels `Proot` and `Px` respectively. The loggers `X.Y` and `X.Y.Z` inherits their level value from their nearest parent `X` having an assigned level..

Logging requests are made by invoking one of the printing methods of a logger instance. These printing methods are `debug`, `info`, `warn`, `error`, `fatal` and `log`.

By definition, the printing method determines the level of a logging request. For example, if `c` is a logger instance, then the statement `c.info("..")` is a logging request of level INFO.

A logging request is said to be *enabled* if its level is higher than or equal to the level of its logger. Otherwise, the request is said to be *disabled*. A logger without an assigned level will inherit one from the hierarchy. This rule is summarized below.

Basic Selection Rule

A log request of level *p* in a logger with (either assigned or inherited, whichever is appropriate) level *q*, is enabled if *p* >= *q*.

This rule is at the heart of log4j. It assumes that levels are ordered. For the standard levels, we have `DEBUG < INFO < WARN < ERROR < FATAL`.

Here is an example of this rule.

```
// get a logger instance named "com.foo"
Logger  logger = Logger.getLogger("com.foo");

// Now set its level. Normally you do not need to set the
// level of a logger programmatically. This is usually done
// in configuration files.
logger.setLevel(Level.INFO);

Logger barlogger = Logger.getLogger("com.foo.Bar");

// This request is enabled, because WARN >= INFO.
logger.warn("Low fuel level.");

// This request is disabled, because DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");

// The logger instance barlogger, named "com.foo.Bar",
// will inherit its level from the logger named
// "com.foo" Thus, the following request is enabled
```

```
   // because INFO >= INFO.
   barlogger.info("Located nearest gas station.");

   // This request is disabled, because DEBUG < INFO.
   barlogger.debug("Exiting gas station search");
```

Calling the `getLogger` method with the same name will always return a reference to the exact same logger object.

For example, in

```
   Logger x = Logger.getLogger("wombat");
   Logger y = Logger.getLogger("wombat");
```

`x` and `y` refer to *exactly* the same logger object.

Thus, it is possible to configure a logger and then to retrieve the same instance somewhere else in the code without passing around references. In fundamental contradiction to biological parenthood, where parents always preceed their children, log4j loggers can be created and configured in any order. In particular, a "parent" logger will find and link to its descendants even if it is instantiated after them.

Configuration of the log4j environment is typically done at application initialization. The preferred way is by reading a configuration file. This approach will be discussed shortly.

Log4j makes it easy to name loggers by *software component*. This can be accomplished by statically instantiating a logger in each class, with the logger name equal to the fully qualified name of the class. This is a useful and straightforward method of defining loggers. As the log output bears the name of the generating logger, this naming strategy makes it easy to identify the origin of a log message. However, this is only one possible, albeit common, strategy for naming loggers. Log4j does not restrict the possible set of loggers. The developer is free to name the loggers as desired.

Nevertheless, naming loggers after the class where they are located seems to be the best strategy known so far.

Appenders and Layouts

The ability to selectively enable or disable logging requests based on their logger is only part of the picture. Log4j allows logging requests to print to multiple destinations. In log4j speak, an output destination is called an *appender*. Currently, appenders exist for the console, files, GUI components, remote socket servers, JMS, NT Event Loggers, and remote UNIX Syslog daemons. It is also possible to log asynchronously.

More than one appender can be attached to a logger.

The addAppender method adds an appender to a given logger. **Each enabled logging request for a given logger will be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy.** In other words, appenders are inherited additively from the logger hierarchy. For example, if a console appender is added to the root logger, then all enabled logging requests will at least print on the console. If in addition a file appender is added to a logger, say *C*, then enabled logging requests for *C* and *C*'s children will print on a file *and* on the console. It is possible to override this default behavior so that appender accumulation is no longer additive by setting the additivity flag to `false`.

The rules governing appender additivity are summarized below.

Appender Additivity

The output of a log statement of logger *C* will go to all the appenders in *C* and its ancestors. This is the meaning of the term "appender additivity".

However, if an ancestor of logger *C*, say *P*, has the additivity flag set to `false`, then *C*'s output will be directed to all the appenders in *C* and it's ancestors upto and including *P* but not the appenders in any of the ancestors of *P*.

Loggers have their additivity flag set to `true` by default.

The table below shows an example:

Logger Name	Added Appenders	Additivity Flag	Output Targets	Comment
root	A1	not applicable	A1	The root logger is anonymous but can be accessed with the Logger.getRootLogger() method. There is no default appender attached to root.
x	A-x1, A-x2	true	A1, A-x1, A-x2	Appenders of "x" and root.
x.y	none	true	A1, A-x1, A-x2	Appenders of "x" and root.

x.y.z	A-xyz1	true	A1, A-x1, A-x2, A-xyz1	Appenders in "x.y.z", "x" and root.
security	A-sec	false	A-sec	No appender accumulation since the additivity flag is set to `false`.
security.access	none	true	A-sec	Only appenders of "security" because the additivity flag in "security" is set to `false`.

More often than not, users wish to customize not only the output destination but also the output format. This is accomplished by associating a *layout* with an appender. The layout is responsible for formatting the logging request according to the user's wishes, whereas an appender takes care of sending the formatted output to its destination. The PatternLayout, part of the standard log4j distribution, lets the user specify the output format according to conversion patterns similar to the C language `printf` function.

For example, the PatternLayout with the conversion pattern "%r [%t] %-5p %c - %m%n" will output something akin to:

```
176 [main] INFO  org.foo.Bar - Located nearest gas station.
```

The first field is the number of milliseconds elapsed since the start of the program. The second field is the thread making the log request. The third field is the level of the log statement. The fourth field is the name of the logger associated with the log request. The text after the '-' is the message of the statement.

Just as importantly, log4j will render the content of the log message according to user specified criteria. For example, if you frequently need to log `Oranges`, an object type used in your current project, then you can register an `OrangeRenderer` that will be invoked whenever an orange needs to be logged.

Object rendering follows the class hierarchy. For example, assuming oranges are fruits, if you register an `FruitRenderer`, all fruits including oranges will be rendered by the `FruitRenderer`, unless of course you registered an orange specific `OrangeRenderer`.

Object renderers have to implement the ObjectRenderer interface.

Configuration

Inserting log requests into the application code requires a fair amount of planning and effort. Observation shows that approximately 4 percent of code is dedicated to logging. Consequently, even moderately sized

applications will have thousands of logging statements embedded within their code. Given their number, it becomes imperative to manage these log statements without the need to modify them manually.

The log4j environment is fully configurable programmatically. However, it is far more flexible to configure log4j using configuration files. Currently, configuration files can be written in XML or in Java properties (key=value) format.

Let us give a taste of how this is done with the help of an imaginary application `MyApp` that uses log4j.

```
import com.foo.Bar;

// Import log4j classes.
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class MyApp {

  // Define a static logger variable so that it references the
  // Logger instance named "MyApp".
  static Logger logger = Logger.getLogger(MyApp.class);

  public static void main(String[] args) {

    // Set up a simple configuration that logs on the console.
    BasicConfigurator.configure();

    logger.info("Entering application.");
    Bar bar = new Bar();
    bar.doIt();
    logger.info("Exiting application.");
  }
}
```

`MyApp` begins by importing log4j related classes. It then defines a static logger variable with the name `MyApp` which happens to be the fully qualified name of the class.

`MyApp` uses the `Bar` class defined in the package `com.foo`.

```
package com.foo;
import org.apache.log4j.Logger;

public class Bar {
  static Logger logger = Logger.getLogger(Bar.class);

  public void doIt() {
    logger.debug("Did it again!");
  }
}
```

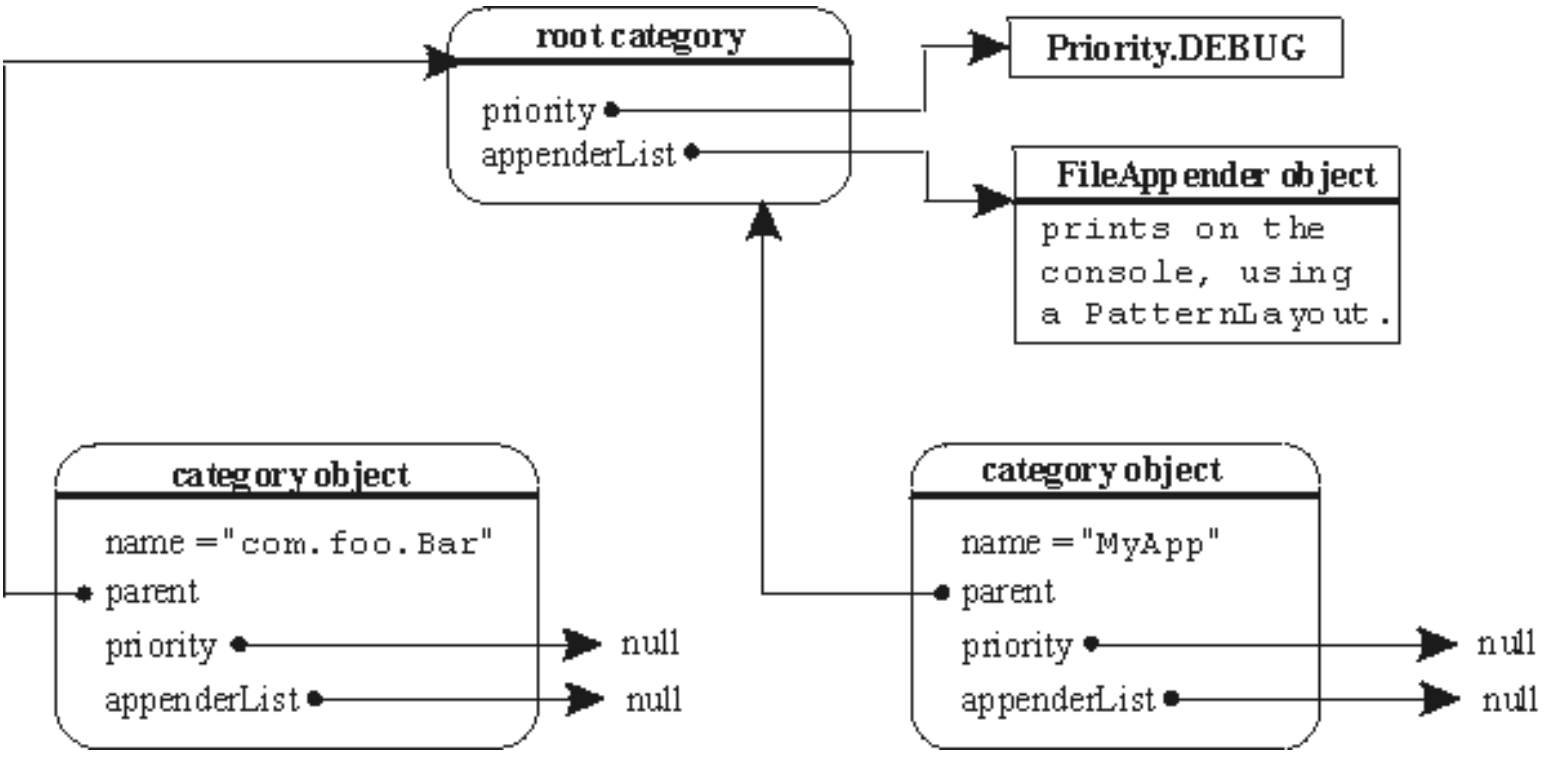
The invocation of the BasicConfigurator.configure method creates a rather simple log4j setup. This method is hardwired to add to the root logger a ConsoleAppender. The output will be formatted using a PatternLayout set to the pattern "%-4r [%t] %-5p %c %x - %m%n".

Note that by default, the root logger is assigned to `Level.DEBUG`.

The output of MyApp is:

```
0    [main] INFO  MyApp  - Entering application.
36   [main] DEBUG com.foo.Bar  - Did it again!
51   [main] INFO  MyApp  - Exiting application.
```

The figure below depicts the object diagram of `MyApp` after just having called the `BasicConfigurator.configure` method.



As a side note, let me mention that in log4j child loggers link only to their existing ancestors. In

particular, the logger named `com.foo.Bar` is linked directly to the `root` logger, thereby circumventing the unused `com` or `com.foo` loggers. This significantly increases performance and reduces log4j's memory footprint.

The `MyApp` class configures log4j by invoking `BasicConfigurator.configure` method. Other classes only need to import the `org.apache.log4j.Logger` class, retrieve the loggers they wish to use, and log away.

The previous example always outputs the same log information. Fortunately, it is easy to modify `MyApp` so that the log output can be controlled at run-time. Here is a slightly modified version.

```
import com.foo.Bar;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class MyApp {

  static Logger logger = Logger.getLogger(MyApp.class.getName());

  public static void main(String[] args) {


    // BasicConfigurator replaced with PropertyConfigurator.
    PropertyConfigurator.configure(args[0]);

    logger.info("Entering application.");
    Bar bar = new Bar();
    bar.doIt();
    logger.info("Exiting application.");
  }
}
```

This version of `MyApp` instructs `PropertyConfigurator` to parse a configuration file and set up logging accordingly.

Here is a sample configuration file that results in exactly same output as the previous `BasicConfigurator` based example.

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

Suppose we are no longer interested in seeing the output of any component belonging to the `com.foo` package. The following configuration file shows one possible way of achieving this.

```
log4j.rootLogger=DEBUG, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout

# Print the date in ISO 8601 format
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

# Print only messages of level WARN or above in the package com.foo.
log4j.logger.com.foo=WARN
```

The output of `MyApp` configured with this file is shown below.

```
2000-09-07 14:07:41,508 [main] INFO  MyApp - Entering application.
2000-09-07 14:07:41,529 [main] INFO  MyApp - Exiting application.
```

As the logger `com.foo.Bar` does not have an assigned level, it inherits its level from `com.foo`, which was set to WARN in the configuration file. The log statement from the `Bar.doIt` method has the level DEBUG, lower than the logger level WARN. Consequently, `doIt()` method's log request is suppressed.

Here is another configuration file that uses multiple appenders.

```
log4j.rootLogger=debug, stdout, R

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=example.log

log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

Calling the enhanced MyApp with the this configuration file will output the following on the console.

```
 INFO [main] (MyApp2.java:12) - Entering application.
DEBUG [main] (Bar.java:8) - Doing it again!
 INFO [main] (MyApp2.java:15) - Exiting application.
```

In addition, as the root logger has been allocated a second appender, output will also be directed to the `example.log` file. This file will be rolled over when it reaches 100KB. When roll-over occurs, the old version of `example.log` is automatically moved to `example.log.1`.

Note that to obtain these different logging behaviors we did not need to recompile code. We could just as easily have logged to a UNIX Syslog daemon, redirected all `com.foo` output to an NT Event logger, or forwarded logging events to a remote log4j server, which would log according to local server policy, for example by forwarding the log event to a second log4j server.

Default Initialization Procedure

The log4j library does not make any assumptions about its environment. In particular, there are no default log4j appenders. Under certain well-defined circumstances however, the static inializer of the `Logger` class will attempt to automatically configure log4j. The Java language guarantees that the static initializer of a class is called once and only once during the loading of a class into memory. It is important to remember that different classloaders may load distinct copies of the same class. These copies of the same class are considered as totally unrelated by the JVM.

The default initialization is very useful in environments where the exact entry point to the application depends on the runtime environment. For example, the same application can be used as a stand-alone application, as an applet, or as a servlet under the control of a web-server.

The exact default initialization algorithm is defined as follows:

1. Setting the **log4j.defaultInitOverride** system property to any other value then "false" will cause log4j to skip the default initialization procedure (this procedure).

2. Set the `resource` string variable to the value of the **log4j.configuration** system property. *The preferred way to specify the default initialization file is through the **log4j.configuration** system property.* In case the system property **log4j.configuration** is not defined, then set the string variable `resource` to its default value "log4j.properties".

3. Attempt to convert the `resource` variable to a URL.

4. If the resource variable cannot be converted to a URL, for example due to a `MalformedURLException`, then search for the `resource` from the classpath by calling `org.apache.log4j.helpers.Loader.getResource(resource, Logger.class)` which returns a URL. Note that the string "log4j.properties" constitutes a malformed URL.

   See Loader.getResource(java.lang.String) for the list of searched locations.

5. If no URL could not be found, abort default initialization. Otherwise, configure log4j from the URL.

   The PropertyConfigurator will be used to parse the URL to configure log4j unless the URL ends with the ".xml" extension, in which case the DOMConfigurator will be used. You can optionaly specify a custom configurator. The value of the **log4j.configuratorClass** system property is taken as the fully qualified class name of your custom configurator. The custom configurator you specify *must* implement the Configurator interface.

Example Configurations

Default Initialization under Tomcat

The default log4j initialization is particularly useful in web-server environments. Under Tomcat 3.x and 4.x, you should place the `log4j.properties` under the `WEB-INF/classes` directory of your web-applications. Log4j will find the properties file and initialize itself. This is easy to do and it works.

You can also choose to set the system property **log4j.configuration** before starting Tomcat. For Tomcat 3.x The `TOMCAT_OPTS` environment variable is used to set command line options. For Tomcat 4.0, set the `CATALINA_OPTS` environment variable instead of `TOMCAT_OPTS`.

Example 1

The Unix shell command

```
export TOMCAT_OPTS="-Dlog4j.configuration=foobar.txt"
```

tells log4j to use the file `foobar.txt` as the default configuration file. This file should be place under the `WEB-INF/classes` directory of your web-application. The file will be read using the PropertyConfigurator. Each web-application will use a different default configuration file because each file is relative to a web-application.

Example 2

The Unix shell command

```
export TOMCAT_OPTS="-Dlog4j.debug -Dlog4j.configuration=foobar.xml"
```

tells log4j to output log4j-internal debugging information and to use the file `foobar.xml` as the default configuration file. This file should be place under the `WEB-INF/classes` directory of your web-application. Since the file ends with a `.xml` extension, it will read using the DOMConfigurator. Each web-application will use a different default configuration file because each file is relative to a web-application.

Example 3

The Windows shell command

```
set TOMCAT_OPTS=-Dlog4j.configuration=foobar.lcf -Dlog4j.
configuratorClass=com.foo.BarConfigurator
```

tells log4j to use the file `foobar.lcf` as the default configuration file. This file should be place under the `WEB-INF/classes` directory of your web-application. Due to the definition of the **log4j.configuratorClass** system property, the file will be read using the `com.foo.BarConfigurator` custom configurator. Each web-application will use a different default configuration file because each file is relative to a web-application.

Example 4

The Windows shell command

```
set TOMCAT_OPTS=-Dlog4j.configuration=file:/c:/foobar.lcf
```

tells log4j to use the file `c:\foobar.lcf` as the default configuration file. The configuration file is fully specified by the URL `file:/c:/foobar.lcf`. Thus, the same configuration file will be used for all web-applications.

Different web-applications will load the log4j classes through their respective classloaderss. Thus, each image of the log4j environment will act independetly and without any mutual synchronization. For example, `FileAppenders` defined exactly the same way in multiple web-application configurations

will all attempt to write the same file. The results are likely to be less than satisfactory. You must make sure that log4j configurations of different web-applications do not use the same underlying system resource.

Initialization servlet

It is also possible to use a special servlet for log4j initialization. Here is an example,

```
package com.foo;

import org.apache.log4j.PropertyConfigurator;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.IOException;

public class Log4jInit extends HttpServlet {

  public
  void init() {
    String prefix =  getServletContext().getRealPath("/");
    String file = getInitParameter("log4j-init-file");
    // if the log4j-init-file is not set, then no point in trying
    if(file != null) {
      PropertyConfigurator.configure(prefix+file);
    }
  }

  public
  void doGet(HttpServletRequest req, HttpServletResponse res) {
  }
}
```

Define the following servlet in the web.xml file for your web-application.

```
<servlet>
  <servlet-name>log4j-init</servlet-name>
  <servlet-class>com.foo.Log4jInit</servlet-class>

  <init-param>
    <param-name>log4j-init-file</param-name>
    <param-value>WEB-INF/classes/log4j.lcf</param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>
```

Writing an initialization servlet is the most flexible way for initializing log4j. There are no constraints on the code you can place in the `init()` method of the servlet.

Nested Diagnostic Contexts

Most real-world systems have to deal with multiple clients simultaneously. In a typical multithreaded implementation of such a system, different threads will handle different clients. Logging is especially well suited to trace and debug complex distributed applications. A common approach to differentiate the logging output of one client from another is to instantiate a new separate logger for each client. This promotes the proliferation of loggers and increases the management overhead of logging.

A lighter technique is to uniquely stamp each log request initiated from the same client interaction. Neil Harrison described this method in the book "Patterns for Logging Diagnostic Messages," in *Pattern Languages of Program Design 3*, edited by R. Martin, D. Riehle, and F. Buschmann (Addison-Wesley, 1997).

To uniquely stamp each request, the user pushes contextual information into the NDC, the abbreviation of *Nested Diagnostic Context*. The NDC class is shown below.

```
public class NDC {
  // Used when printing the diagnostic
  public static String get();

  // Remove the top of the context from the NDC.
  public static String pop();

  // Add diagnostic context for the current thread.
  public static void push(String message);

  // Remove the diagnostic context for this thread.
  public static void remove();
}
```

The NDC is managed per thread as a *stack* of contextual information. Note that all methods of the `org.apache.log4j.NDC` class are static. Assuming that NDC printing is turned on, every time a log request is made, the appropriate log4j component will include the *entire* NDC stack for the current thread in the log output. This is done without the intervention of the user, who is responsible only for placing the correct information in the NDC by using the `push` and `pop` methods at a few well-defined points in the code. In contrast, the per-client logger approach commands extensive changes in the code.

To illustrate this point, let us take the example of a servlet delivering content to numerous clients. The servlet can build the NDC at the very beginning of the request before executing other code. The contextual information can be the client's host name and other information inherent to the request, typically information contained in cookies. Hence, even if the servlet is serving multiple clients simultaneously, the logs initiated by the same code, i.e. belonging to the same logger, can still be distinguished because each client request will have a different NDC stack. Contrast this with the complexity of passing a freshly instantiated logger to all code exercised during the client's request.

Nevertheless, some sophisticated applications, such as virtual hosting web servers, must log differently depending on the virtual host context and also depending on the software component issuing the request. Recent log4j releases support multiple hierarchy trees. This enhancement allows each virtual host to possess its own copy of the logger hierarchy.

Performance

One of the often-cited arguments against logging is its computational cost. This is a legitimate concern as even moderately sized applications can generate thousands of log requests. Much effort was spent measuring and tweaking logging performance. Log4j claims to be fast and flexible: speed first, flexibility second.

The user should be aware of the following performance issues.

1. **Logging performance when logging is turned off.**

   When logging is turned off entirely or just for a set of levels, the cost of a log request consists of a method invocation plus an integer comparison. On a 233 MHz Pentium II machine this cost is typically in the 5 to 50 nanosecond range.

   However, The method invocation involves the "hidden" cost of parameter construction.

   For example, for some logger `cat`, writing,

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

   incurs the cost of constructing the message parameter, i.e. converting both integer `i` and `entry[i]` to a String, and concatenating intermediate strings, regardless of whether the message will be logged or not. This cost of parameter construction can be quite high and it depends on the size of

the parameters involved.

To avoid the parameter construction cost write:

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

This will not incur the cost of parameter construction if debugging is disabled. On the other hand, if the logger is debug-enabled, it will incur twice the cost of evaluating whether the logger is enabled or not: once in `debugEnabled` and once in `debug`. This is an insignificant overhead because evaluating a logger takes about 1% of the time it takes to actually log.

In log4j, logging requests are made to instances of the Logger class. Logger is a class and not an interface. This measurably reduces the cost of method invocation at the cost of some flexibility.

Certain users resort to preprocessing or compile-time techniques to compile out all log statements. This leads to perfect performance efficiency with respect to logging. However, since the resulting application binary does not contain any log statements, logging cannot be turned on for that binary. In my opinion this is a disproportionate price to pay in exchange for a small performance gain.

2. **The performance of deciding whether to log or not to log when logging is turned on.**

   This is essentially the performance of walking the logger hierarchy. When logging is turned on, log4j still needs to compare the level of the log request with the level of the request logger. However, loggers may not have an assigned level; they can inherit them from the logger hierarchy. Thus, before inheriting a level, the logger may need to search its ancestors.

   There has been a serious effort to make this hierarchy walk to be as fast as possible. For example, child loggers link only to their existing ancestors. In the `BasicConfigurator` example shown earlier, the logger named `com.foo.Bar` is linked directly to the root logger, thereby circumventing the nonexistent `com` or `com.foo` loggers. This significantly improves the speed of the walk, especially in "sparse" hierarchies.

   The typical cost of walking the hierarchy is typically 3 times slower than when logging is turned off entirely.

3. **Actually outputting log messages**

   This is the cost of formatting the log output and sending it to its target destination. Here again, a serious effort was made to make layouts (formatters) perform as quickly as possible. The same is true for appenders. The typical cost of actually logging is about 100 to 300 microseconds. See org.apache.log4.performance.Logging for actual figures.

Although log4j has many features, its first design goal was speed. Some log4j components have been rewritten many times to improve performance. Nevertheless, contributors frequently come up with new optimizations. You should be pleased to know that when configured with the SimpleLayout performance tests have shown log4j to log as quickly as `System.out.println`.

Conclusions

Log4j is a popular logging package written in Java. One of its distinctive features is the notion of inheritance in loggers. Using a logger hierarchy it is possible to control which log statements are output at arbitrary granularity. This helps reduce the volume of logged output and minimize the cost of logging.

One of the advantages of the log4j API is its manageability. Once the log statements have been inserted into the code, they can be controlled with configuration files. They can be selectively enabled or disabled, and sent to different and multiple output targets in user-chosen formats. The log4j package is designed so that log statements can remain in shipped code without incurring a heavy performance cost.

Acknowledgments

Many thanks to N. Asokan for reviewing the article. He is also one of the originators of the logger concept. I am indebted to Nelson Minar for encouraging me to write this article. He has also made many useful suggestions and corrections to this article. Log4j is the result of a collective effort. My special thanks go to all the authors who have contributed to the project. Without exception, the best features in the package have all originated in the user community.