
Log4J

Ashley J.S Mills

<ashley@ashleymills.com>
Copyright © 2005 The University Of Birmingham

Table of Contents

1. Introduction	1
2. Installation	1
3. log4j Basic Concepts	1
3.1. Logger	2
3.2. Appender	3
3.2.1. Using A ConsoleAppender	3
3.2.2. Using A FileAppender	3
3.2.3. Using A WriterAppender	3
3.3. Layout	4
3.4. Basic Examples Illustrating this	4
3.4.1. SimpleLayout and FileAppender	4
3.4.2. HTMLLayout and WriterAppender	4
3.4.3. PatternLayout and ConsoleAppender	5
4. Using External Configuration Files	5
5. References (And links you may find useful)	7

1. Introduction

Logging within the context of program development constitutes inserting statements into the program that provide some kind of output information that is useful to the developer. Examples of logging are trace statements, dumping of structures and the familiar `System.out.println` or `printf` debug statements. `log4j` offers a hierarchical way to insert logging statements within a Java program. Multiple output formats and multiple levels of logging information are available.

By using a dedicated logging package, the overhead of maintaining thousands of `System.out.println` statements is alleviated as the logging may be controlled at runtime from configuration scripts. `log4j` maintains the log statements in the shipped code. By formalising the process of logging, some feel that one is encouraged to use logging more and with higher degree of usefulness.

2. Installation

In order to use the tools we are about to install it is necessary to setup the operating environment so that the tools know where to find stuff they need and the operating system knows where to find the tools. A understanding of how to do this is essential as you will be asked to change the operating environment. I have comprehensively covered this in documents entitled *Configuring A Windows Working Environment* [[../winenvars/winenvarshome.html](#)] and *Configuring A Unix Working Environment* [[../unixenvars/unixenvarshome.html](#)].

1. Download the `log4j` distribution from <http://jakarta.apache.org/log4j/docs/download.html>.
2. Extract the archived files to some suitable directory.
3. Add the file `dist/lib/log4j-1.2.6.jar` to your `CLASSPATH` environment variable.
4. Download <http://apache.rmpc.co.uk/dist/xml/xerces-j/Xerces-J-bin.2.6.0.zip> and unzip it to a temporary directory. Copy the files `xercesImpl.jar` and `xmlParserAPIs.jar` to some permanent location and append their paths to the `CLASSPATH` environment variable.

3. log4j Basic Concepts

The use of `log4j` revolves around 3 main things:

1. *public class* `Logger`
`Logger` is responsible for handling the majority of log operations.
2. *public interface* `Appender`

Appender is responsible for controlling the output of log operations.

3. *public abstract class Layout*

Layout is responsible for formatting the output for *Appender*.

3.1. Logger

The logger is the core component of the logging process. In log4j, there are 5 normal levels *Levels* of logger available (not including custom *Levels*), the following is borrowed from the log4j API (<http://jakarta.apache.org/log4j/docs/api/index.html>):

- *static Level DEBUG*
The DEBUG Level designates fine-grained informational events that are most useful to debug an application.
- *static Level INFO*
The INFO level designates informational messages that highlight the progress of the application at coarse-grained level.
- *static Level WARN*
The WARN level designates potentially harmful situations.
- *static Level ERROR*
The ERROR level designates error events that might still allow the application to continue running.
- *static Level FATAL*
The FATAL level designates very severe error events that will presumably lead the application to abort.

In addition, there are two special levels of logging available: (descriptions borrowed from the log4j API <http://jakarta.apache.org/log4j/docs/api/index.html>):

- *static Level ALL*
The ALL Level has the lowest possible rank and is intended to turn on all logging.
- *static Level OFF*
The OFF Level has the highest possible rank and is intended to turn off logging.

The behaviour of loggers is hierarchical. The following table illustrates this:

Figure 1. Logger Output Hierarchy

		Will Output Messages Of Level				
		DEBUG	INFO	WARN	ERROR	FATAL
Logger Level	DEBUG					
	INFO					
	WARN					
	ERROR					
	FATAL					
ALL						
OFF						

A logger will only output messages that are of a level greater than or equal to it. If the level of a logger is not set it will inherit the level of the closest ancestor. So if a logger is created in the package *com.foo.bar* and no level is set for it, it will inherit the level of the logger created in *com.foo*. If no logger was created in *com.foo*, the logger created in *com.foo.bar* will inherit the level of the *root* logger, the *root* logger is always instantiated and available, the *root* logger is assigned the level *DEBUG*.

There are a number of ways to create a logger, one can retrieve the *root* logger:

```
Logger logger = Logger.getRootLogger();
```

One can create a new logger:

```
Logger logger = Logger.getLogger("MyLogger");
```

More usually, one instantiates a static logger globally, based on the name of the class:

```
static Logger logger = Logger.getLogger(test.class);
```

All these create a logger called "logger", one can set the level with:

```
logger.setLevel((Level)Level.WARN);
```

You can use any of 7 levels; Level.DEBUG, Level.INFO, Level.WARN, Level.ERROR, Level.FATAL, Level.ALL and Level.OFF.

3.2. Appender

The *Appender* controls how the logging is output. The Appenders available are (descriptions borrowed from the log4j API <http://jakarta.apache.org/log4j/docs/api/index.html>):

1. *ConsoleAppender*: appends log events to System.out or System.err using a layout specified by the user. The default target is System.out.
2. *DailyRollingFileAppender* extends FileAppender so that the underlying file is rolled over at a user chosen frequency.
3. *FileAppender* appends log events to a file.
4. *RollingFileAppender* extends FileAppender to backup the log files when they reach a certain size.
5. *WriterAppender* appends log events to a Writer or an OutputStream depending on the user's choice.
6. *SMTPAppender* sends an e-mail when a specific logging event occurs, typically on errors or fatal errors.
7. *SocketAppender* sends LoggingEvent objects to a remote a log server, usually a SocketNode.
8. *SocketHubAppender* sends LoggingEvent objects to a set of remote log servers, usually a SocketNodes
9. *SyslogAppenders* sends messages to a remote syslog daemon.
10. *TelnetAppender* is a log4j appender that specializes in writing to a read-only socket.

One may also implement the *Appender* interface to create ones own ways of outputting log statements.

3.2.1. Using A ConsoleAppender

A ConsoleAppender can be created like this:

```
ConsoleAppender appender = new ConsoleAppender(new PatternLayout());
```

Which creates a console appender, with a default PatternLayout. The default output of *System.out* is used.

3.2.2. Using A FileAppender

A FileAppender can be created like this:

```
FileAppender appender = null;
try {
    appender = new FileAppender(new PatternLayout(), "filename");
} catch (Exception e) {}
```

The constructor in use above is:

```
FileAppender(Layout layout, String filename)
    Instantiate a FileAppender and open the file designated by filename.
```

Another useful constructor is:

```
FileAppender(Layout layout, String filename, boolean append)
    Instantiate a FileAppender and open the file designated by filename.
```

So that one may choose whether or not to append the file specified or not. If this is not specified, the default is to append.

3.2.3. Using A WriterAppender

A WriterAppender can be created like this:

```
WriterAppender appender = null;
try {
    appender = new WriterAppender(new PatternLayout(), new FileOutputStream("filename"));
} catch (Exception e) {}
```

This `WriterAppender` uses the constructor that takes a `PatternLayout` and an `OutputStream` as arguments, in this case a `FileOutputStream` is used to output to a file, there are other constructors available.

3.3. Layout

The Appender must have an associated *Layout* so it knows how to format the output. There are three types of *Layout* available:

1. *HTMLayout* formats the output as a HTML table.
2. *PatternLayout* formats the output based on a *conversion pattern* specified, or if none is specified, the default conversion pattern.
3. *SimpleLayout* formats the output in a very simple manner, it prints the *Level*, then a dash '-' and then the log message.

3.4. Basic Examples Illustrating this

3.4.1. SimpleLayout and FileAppender

Here is a very simplistic example of a program implementing a `SimpleLayout` and `FileAppender`:

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.FileAppender;
public class simpandfile {
    static Logger logger = Logger.getLogger(simpandfile.class);
    public static void main(String args[]) {
        SimpleLayout layout = new SimpleLayout();

        FileAppender appender = null;
        try {
            appender = new FileAppender(layout, "output1.txt", false);
        } catch (Exception e) {}

        logger.addAppender(appender);
        logger.setLevel((Level) Level.DEBUG);

        logger.debug("Here is some DEBUG");
        logger.info("Here is some INFO");
        logger.warn("Here is some WARN");
        logger.error("Here is some ERROR");
        logger.fatal("Here is some FATAL");
    }
}
```

You can download it: [simpandfile.java](#) [files/simpandfile.java]. And checkout the output produced: [output1.txt](#) [files/output1.txt].

3.4.2. HTMLayout and WriterAppender

Here is a very simplistic example of a program implementing a `HTMLayout` and `WriterAppender`:

```
import java.io.*;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.HTMLLayout;
import org.apache.log4j.WriterAppender;
public class htmlandwrite {
    static Logger logger = Logger.getLogger(htmlandwrite.class);
    public static void main(String args[]) {
        HTMLLayout layout = new HTMLLayout();

        WriterAppender appender = null;
        try {
            FileOutputStream output = new FileOutputStream("output2.html");
            appender = new WriterAppender(layout, output);
        } catch (Exception e) {}

        logger.addAppender(appender);
    }
}
```

```

logger.setLevel((Level) Level.DEBUG);

logger.debug("Here is some DEBUG");
logger.info("Here is some INFO");
logger.warn("Here is some WARN");
logger.error("Here is some ERROR");
logger.fatal("Here is some FATAL");
}
}

```

You can download it: [htmlandwrite.java](#) [files/htmlandwrite.java]. And checkout the output produced: [output2.html](#) [files/output2.html].

3.4.3. PatternLayout and ConsoleAppender

Here is a very simplistic example of a program implementing a PatternLayout and ConsoleAppender:

```

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.ConsoleAppender;
public class consandpatt {
    static Logger logger = Logger.getLogger(consandpatt.class);
    public static void main(String args[]) {

        // Note, %n is newline
        String pattern = "Milliseconds since program start: %r %n";
        pattern += "Classname of caller: %C %n";
        pattern += "Date in ISO8601 format: %d{ISO8601} %n";
        pattern += "Location of log event: %l %n";
        pattern += "Message: %m %n %n";

        PatternLayout layout = new PatternLayout(pattern);
        ConsoleAppender appender = new ConsoleAppender(layout);

        logger.addAppender(appender);
        logger.setLevel((Level) Level.DEBUG);

        logger.debug("Here is some DEBUG");
        logger.info("Here is some INFO");
        logger.warn("Here is some WARN");
        logger.error("Here is some ERROR");
        logger.fatal("Here is some FATAL");
    }
}

```

You can download it: [consandpatt.java](#) [files/consandpatt.java]. And checkout the output produced: [output2.txt](#) [files/output2.txt].

4. Using External Configuration Files

Log4j is usually used in conjunction with external configuration files so that options do not have to be hard-coded within the software. The advantage of using an external configuration file is that changes can be made to the options without having to recompile the software. A disadvantage could be, that due to the *io* instructions used, it is slightly slower.

There are two ways in which one can specify the external configuration file: a plain text file or an XML file. Since everything is written in XML these days, this tutorial will focus on the XML approach but will also include relevant plain text examples. To begin with, examine the sample XML config file shown below:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <appender name="ConsoleAppender" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.SimpleLayout" />
    </appender>

    <root>
        <priority value ="debug" />
        <appender-ref ref="ConsoleAppender" />
    </root>

</log4j:configuration>

```

The file starts with a standard XML declaration followed by a DOCTYPE declaration which indicates the DTD(Document Type Definition), this defines the structure of the XML file, what elements may be nested within other elements etc. This file is provided in the log4j distribution under `src/java/org/apache/log4j/xml`. Next comes the all-encapsulating *log4j:configuration* element, which was specified as the root element in the DOCTYPE declaration. Nested within the root element are two structures:

```
<appender name="ConsoleAppender" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.SimpleLayout"/>
</appender>
```

Here an *Appender* is created and called "ConsoleAppender", note that any name could have been chosen, it is because of the con-trivinity of examples that this name was chosen. The class for the appender is then specified in full, when referring to classes, one al-ways uses the fully qualified class name. An *Appender* must always have a *name* and a *class* specified. Nested within *Appender* is the *layout* element which defines the layout to be a *SimpleLayout*. *Layout* must always have the *class* attribute.

```
<root>
  <priority value ="debug" />
  <appender-ref ref="ConsoleAppender"/>
</root>
```

The root element always exists and cannot be sub-classed. The example shows the priority being set to "debug" and the appender setup by including an *appender-ref* element, of which, more than one may be specified. See the file `src/java/org/apache/log4j/xml/log4j.dtd` in your log4j distribution for more information about the structure of an XML configuration file. The configuration file is pulled into the Java program like this:

```
DOMConfigurator.configure("configurationfile.xml");
```

The *DOMConfigurator* is used to initialise the log4j environment using a DOM tree. Here is the example xml configuration file: `plainlog4jconfig.xml` [files/plainlog4jconfig.xml]. Here is a program which implements this configuration file: `files/externalxmltest.java` [externalxmltest.java]:

```
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
public class externalxmltest {
  static Logger logger = Logger.getLogger(externalxmltest.class);
  public static void main(String args[]) {
    DOMConfigurator.configure("xmllog4jconfig.xml");
    logger.debug("Here is some DEBUG");
    logger.info("Here is some INFO");
    logger.warn("Here is some WARN");
    logger.error("Here is some ERROR");
    logger.fatal("Here is some FATAL");
  }
}
```

Here is an XML configuration file for a *Logger* implementing a *FileAppender* using a *PatternLayout*:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="appender" class="org.apache.log4j.FileAppender">
    <param name="File" value="Identify-Log.txt"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] %p - %m%n"/>
    </layout>
  </appender>

  <root>
    <priority value ="debug"/>
    <appender-ref ref="appender"/>
  </root>

</log4j:configuration>
```

You can download this example from here: `xmllog4jconfig2.xml` [files/xmllog4jconfig2.xml]. For more examples of using xml files to configure a log4j environment, see the `src/java/org/apache/log4j/xml/examples/` directory in the log4j distribu-tion.

Here is the configuration file discussed above, expressed in the form of a plain text file:

```
# initialise root logger with level DEBUG and call it BLAH
log4j.rootLogger=DEBUG, BLAH
# add a ConsoleAppender to the logger BLAH
log4j.appender.BLAH=org.apache.log4j.ConsoleAppender
# set set that layout to be SimpleLayout
log4j.appender.BLAH.layout=org.apache.log4j.SimpleLayout
```

You can download it here: [plainlog4jconfig.txt](#) [files/plainlog4jconfig.txt]. Here is a program implementing this:

```
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
public class externalplaintest {
    static Logger logger = Logger.getLogger(externalplaintest.class);
    public static void main(String args[]) {
        PropertyConfigurator.configure("plainlog4jconfig.xml");
        logger.debug("Here is some DEBUG");
        logger.info("Here is some INFO");
        logger.warn("Here is some WARN");
        logger.error("Here is some ERROR");
        logger.fatal("Here is some FATAL");
    }
}
```

You can download an example program that uses this configuration file here: [files/externalplaintest.java](#). For more examples of using plain text files to configure a log4j environment, see the `examples` directory in the log4j distribution.

The use of external example files has only been briefly discussed here, it is assumed that you have the capacity to learn more by yourself by studying the examples provided with the log4j distribution and experimenting.

5. References (And links you may find useful)

- <http://jakarta.apache.org/log4j/docs/manual.html>
Short introduction to log4j - Ceki Gülcü - March 2002
- <http://www.vipan.com/htdocs/log4jhelp.html>
Don't Use System.out.println! Use Log4j - Vipin Singla
- <http://www.opensymphony.com/guidelines/logging.jsp>
LOG4J / OpenSymphony Logging Primer
- <http://builder.com.com/article.jhtml?id=u00820020124kev01.htm>
Add logging to your Java Applications - Kevin Brown