**Log4j Tutorial**

log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License. log4j is a popular logging package written in Java. log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages.

log4j is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console, UNIX Syslog, etc.

log4j has three main components:

* **loggers**: Responsible for capturing logging information.
* **appenders**: Responsible for publishing logging information to various preferred destinations.
* **layouts**: Responsible for formatting logging information in different styles.

## log4j Features

* It is thread-safe.
* It is optimized for speed.
* It is based on a named logger hierarchy.
* It supports multiple output appenders per logger.
* It supports internationalization.
* It is not restricted to a predefined set of facilities.
* Logging behavior can be set at runtime using a configuration file.
* It is designed to handle Java Exceptions from the start.
* It uses multiple levels, namely ALL, TRACE, DEBUG, INFO, WARN, ERROR and FATAL.
* The format of the log output can be easily changed by extending the *Layout* class.
* The target of the log output as well as the writing strategy can be altered by implementations of the Appender interface.
* It is fail-stop. However, although it certainly strives to ensure delivery, log4j does not guarantee that each log statement will be delivered to its destination.

## Pros and Cons of Logging

Logging is an important component of the software development. A well-written logging code offers quick debugging, easy maintenance, and structured storage of an application's runtime information.

Logging does have its drawbacks also. 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.

# Architecture

log4j API follows a layered architecture where each layer provides different objects to perform different tasks. This layered architecture makes the design flexible and easy to extend in future.

There are two types of objects available with log4j framework.

* **Core Objects:** These are mandatory objects of the framework. They are required to use the framework.
* **Support Objects:** These are optional objects of the framework. They support core objects to perform additional but important tasks.

## Core Objects

### Logger Object

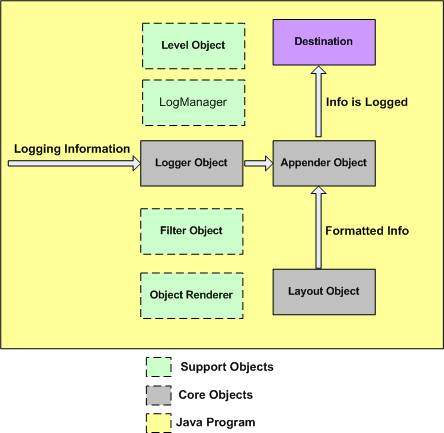
The top-level layer is the Logger which provides the Logger object. The Logger object is responsible for capturing logging information and they are stored in a namespace hierarchy.

### Layout Object

The layout layer provides objects which are used to format logging information in different styles. It provides support to appender objects before publishing logging information. Layout objects play an important role in publishing logging information in a way that is human-readable and reusable.

### Appender Object

This is a lower-level layer which provides Appender objects. The Appender object is responsible for publishing logging information to various preferred destinations such as a database, file, console, UNIX Syslog, etc. The following virtual diagram shows the components of a log4J framework:



**Support Objects**

There are other important objects in the log4j framework that play a vital role in the logging framework:

**Level Object**

The Level object defines the granularity and priority of any logging information. There are seven levels of logging defined within the API: OFF, DEBUG, INFO, ERROR, WARN, FATAL, and ALL.

**Filter Object**

The Filter object is used to analyze logging information and make further decisions on whether that information should be logged or not.

An Appender objects can have several Filter objects associated with them. If logging information is passed to a particular Appender object, all the Filter objects associated with that Appender need to approve the logging information before it can be published to the attached destination.

**ObjectRenderer**

The ObjectRenderer object is specialized in providing a String representation of different objects passed to the logging framework. This object is used by Layout objects to prepare the final logging information.

**LogManager**

The LogManager object manages the logging framework. It is responsible for reading the initial configuration parameters from a system-wide configuration file or a configuration class.

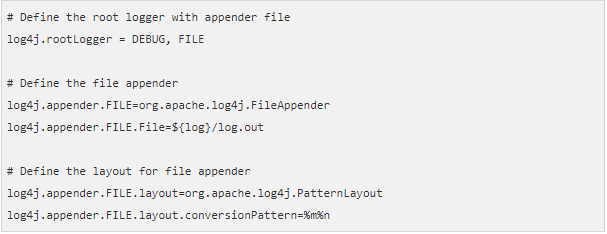
**Configuration**

The previous chapter explained the core components of log4j. This chapter explains how you can configure the core components using a configuration file. Configuring log4j involves assigning the Level, defining Appender, and specifying Layout objects in a configuration file.

The log4j.properties file is a log4j configuration file which keeps properties in key-value pairs. By default, the LogManager looks for a file named log4j.properties in the CLASSPATH.

Using the above syntax, we define the following in log4j.properties file:

* The level of the root logger is defined as DEBUG, The DEBUG appender named FILE to it.
* The appender FILE is defined as org.apache.log4j.FileAppender. It writes to a file named log.out located in the log directory.
* The layout pattern defined is %m%n, which means the printed logging message will be followed by a newline character.



It is important to note that log4j supports UNIX-style variable substitution such as ${variableName}.

**Debug Level**

We have used DEBUG with both the appenders. All the possible options are:

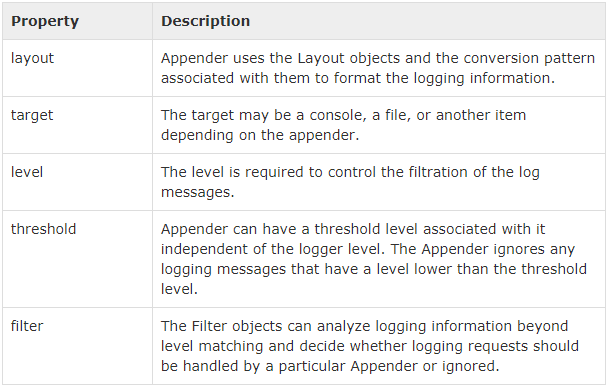
* TRACE
* DEBUG
* INFO
* WARN
* ERROR
* FATAL
* ALL

These levels are explained later in this tutorial.

**Appenders**

Apache log4j provides Appender objects which are primarily responsible for printing logging messages to different destinations such as consoles, files, sockets, NT event logs, etc.

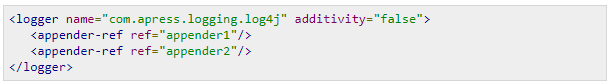
Each Appender object has different properties associated with it, and these properties indicate the behavior of that object.



We can add an Appender object to a Logger by including the following setting in the configuration file with the following method:



You can write same configuration in XML format as follows:



If you are willing to add Appender object inside your program then you can use following method:



The addAppender() method adds an Appender to the Logger object. As the example configuration demonstrates, it is possible to add many Appender objects to a logger in a comma-separated list, each printing logging information to separate destinations.

We have used only one appender FileAppender in our example above. All the possible appender options are:

* AppenderSkeleton
* AsyncAppender
* ConsoleAppender
* DailyRollingFileAppender
* ExternallyRolledFileAppender
* FileAppender
* JDBCAppender
* JMSAppender
* LF5Appender
* NTEventLogAppender
* NullAppender
* RollingFileAppender
* SMTPAppender
* SocketAppender
* SocketHubAppender
* SyslogAppender
* TelnetAppender
* WriterAppender

We would cover FileAppender in Logging in Files and JDBC Appender would be covered in Logging in Database.

**Layout**

We have used PatternLayout with our appender. All the possible options are:

* DateLayout
* HTMLLayout
* PatternLayout
* SimpleLayout
* XMLLayout

Using HTMLLayout and XMLLayout, you can generate log in HTML and in XML format as well.

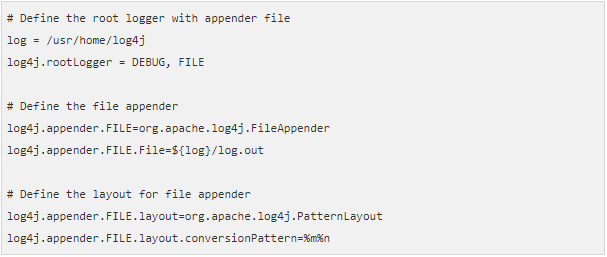
**Sample Program**

We have seen how to create a configuration file. This chapter describe how to generate debug messages and log them in a simple text file.

Following is a simple configuration file created for our example. Let us revise it once again:

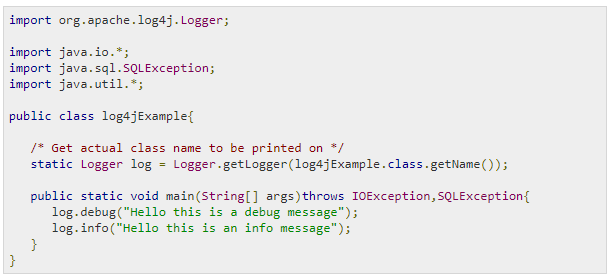
* The level of the root logger is defined as DEBUG and attaches appender named FILE to it.
* The appender FILE is defined as org.apache.log4j.FileAppender and writes to a file named log.out located in the log directory.
* The layout pattern defined is %m%n, which means the printed logging message will be followed by a newline character.

The contents of log4j.properties file are as follows:



**Using log4j in Java Program**

The following Java class is a very simple example that initializes, and then uses, the log4j logging library for Java applications.



**Compile and Execute**

Here are the steps to compile and run the above-mentioned program. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

All the libraries should be available in CLASSPATH and your log4j.properties file should be available in PATH. Follow the steps give below:

* Create log4j.properties as shown above.
* Create log4jExample.java as shown above and compile it.
* Execute log4jExample binary to run the program.

You would get the following result inside /usr/home/log4j/log.out file:



**Logging Methods**

Logger class provides a variety of methods to handle logging activities. The Logger class does not allow us to instantiate a new Logger instance but it provides two static methods for obtaining a Logger object:

* public static Logger getRootLogger();
* public static Logger getLogger(String name);

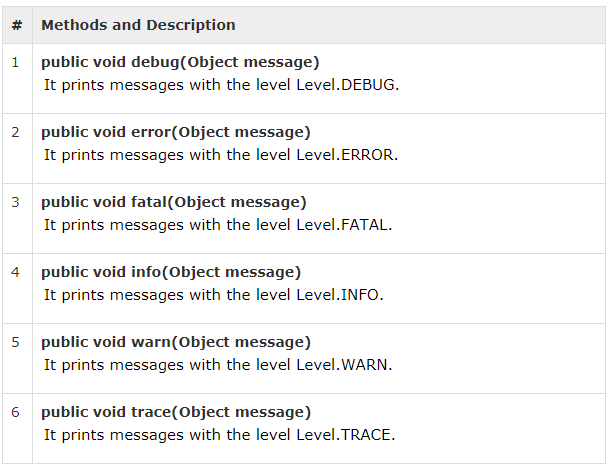
The first of the two methods returns the application instance's root logger and it does not have a name.

Any other named Logger object instance is obtained through the second method by passing the name of the logger. The name of the logger can be any string you can pass, usually a class or a package name as we have used in the last chapter and it is mentioned below:



**Logging Methods**

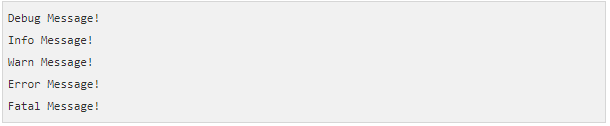
Once we obtain an instance of a named logger, we can use several methods of the logger to log messages. The Logger class has the following methods for printing the logging information.



All the levels are defined in the org.apache.log4j.Level class and any of the above mentioned methods can be called as follows:



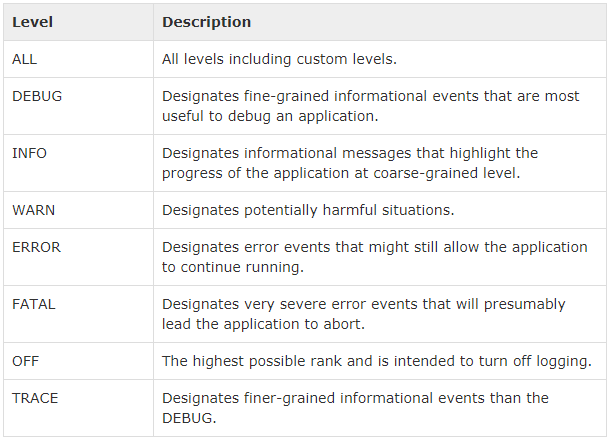
When you compile and run LogClass program, it would generate the following result:



All the debug messages make more sense when they are used in combination with levels. We will cover levels in the next chapter and then, you would have a good understanding of how to use these methods in combination with different levels of debugging.

**Logging Levels**

The org.apache.log4j.Level levels. You can also define your custom levels by sub-classing the Level class.



**How do Levels Works?**

A log request of level **p** in a logger with 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 ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.

The Following example shows how we can filter all our DEBUG and INFO messages. This program uses of logger method setLevel(Level.X) to set a desired logging level. This example would print all the messages except Debug and Info:



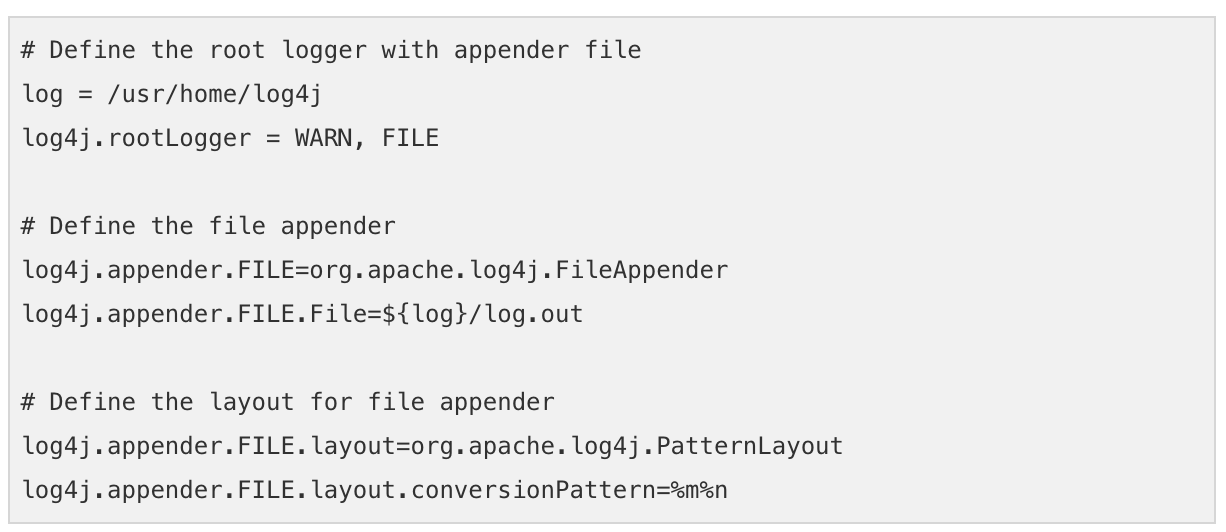
When you compile and run the LogClass program, it would generate the following result:



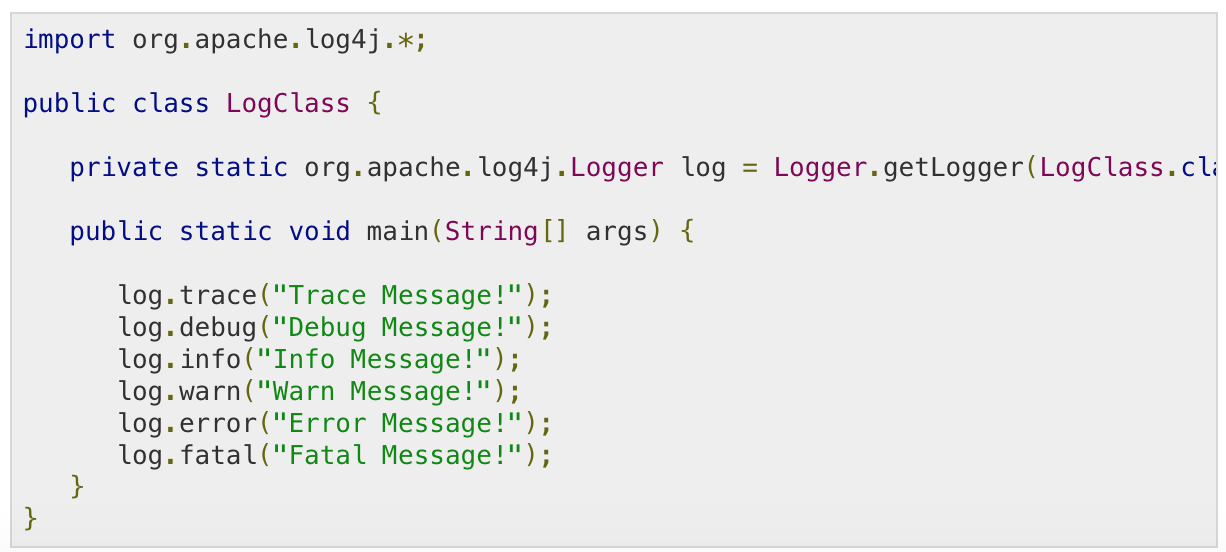
**Setting Levels using Configuration File**

log4j provides you configuration file based level setting which sets you free from changing the source code when you want to change the debugging level.

Following is an example configuration file which would perform the same task as we did using the log.setLevel(Level.WARN) method in the above example.



Let us now use our following program:



Now compile and run the above program and you would get following result in /usr/home/log4j/log.out file:



**Log Formatting**

Apache log4j provides various Layout objects, each of which can format logging data according to various layouts. It is also possible to create a Layout object that formats logging data in an application-specific way.

All Layout objects receive a LoggingEvent object from the Appender objects. The Layout objects then retrieve the message argument from the LoggingEvent and apply the appropriate ObjectRenderer to obtain the String representation of the message.

**The Layout Types**

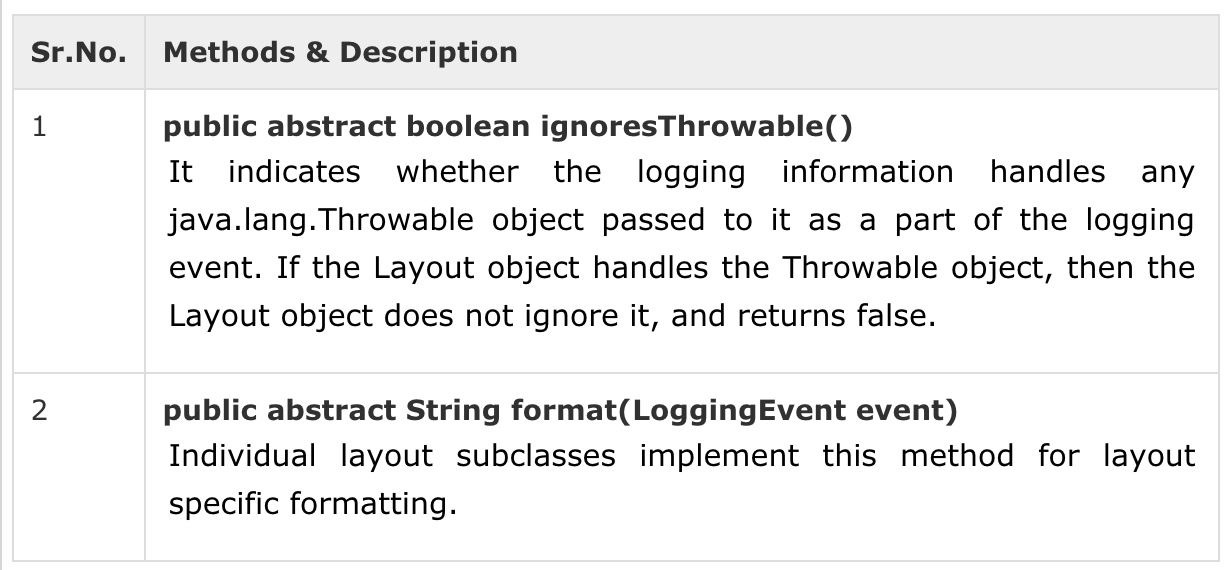
The top-level class in the hierarchy is the abstract class org.apache.log4j.Layout. This is the base class for all other Layout classes in the log4j API.

The Layout class is defined as abstract within an application, we never use this class directly; instead, we work with its subclasses which are as follows:

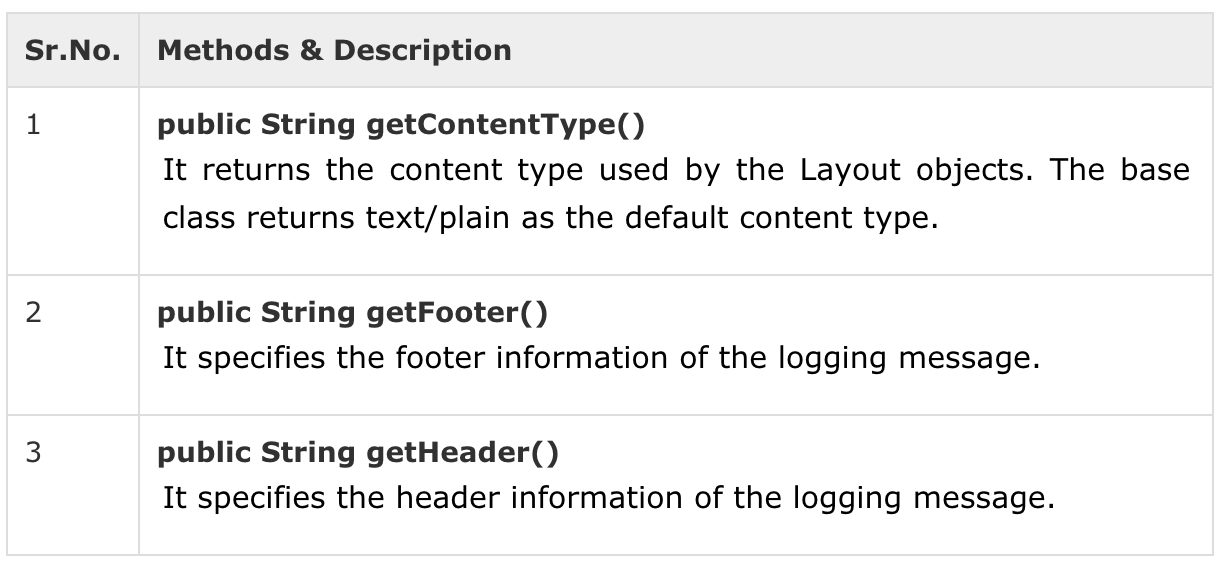
* DateLayout
* HTMLLayout
* PatternLayout.
* SimpleLayout
* XMLLayout

**The Layout Methods**

This class provides a skeleton implementation of all the common operations across all other Layout objects and declares two abstract methods.



Apart from these abstract methods, the Layout class provides concrete implementation for the methods listed below:



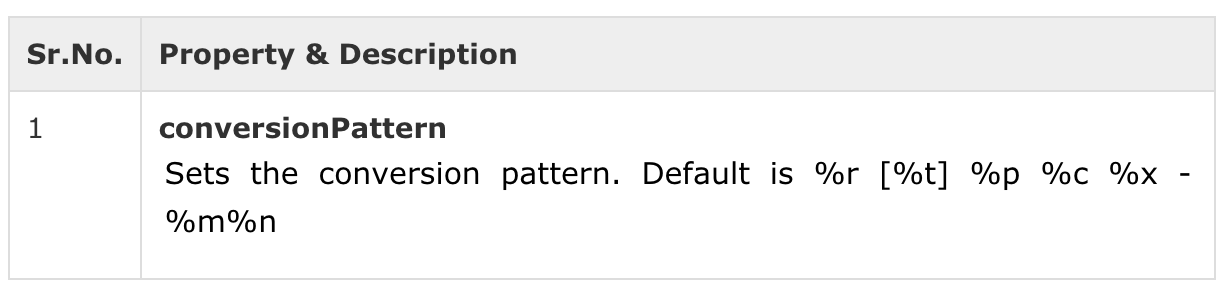
Each subclass can return class-specific information by overriding the concrete implementation of these methods.

**PatternLayout**

If you want to generate your logging information in a particular format based on a pattern, then you can use org.apache.log4j.PatternLayout to format your logging information.

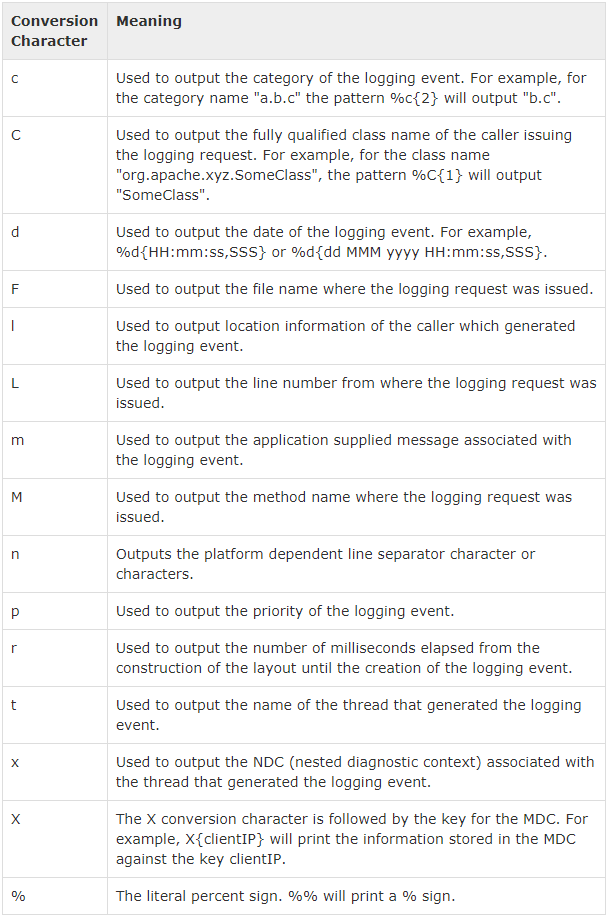
The PatternLayout class extends the abstract org.apache.log4j.Layout class and overrides the format() method to structure the logging information according to a supplied pattern.

PatternLayout is also a simple Layout object that provides the following-Bean Property which can be set using the configuration file:



**Pattern Conversion Characters**

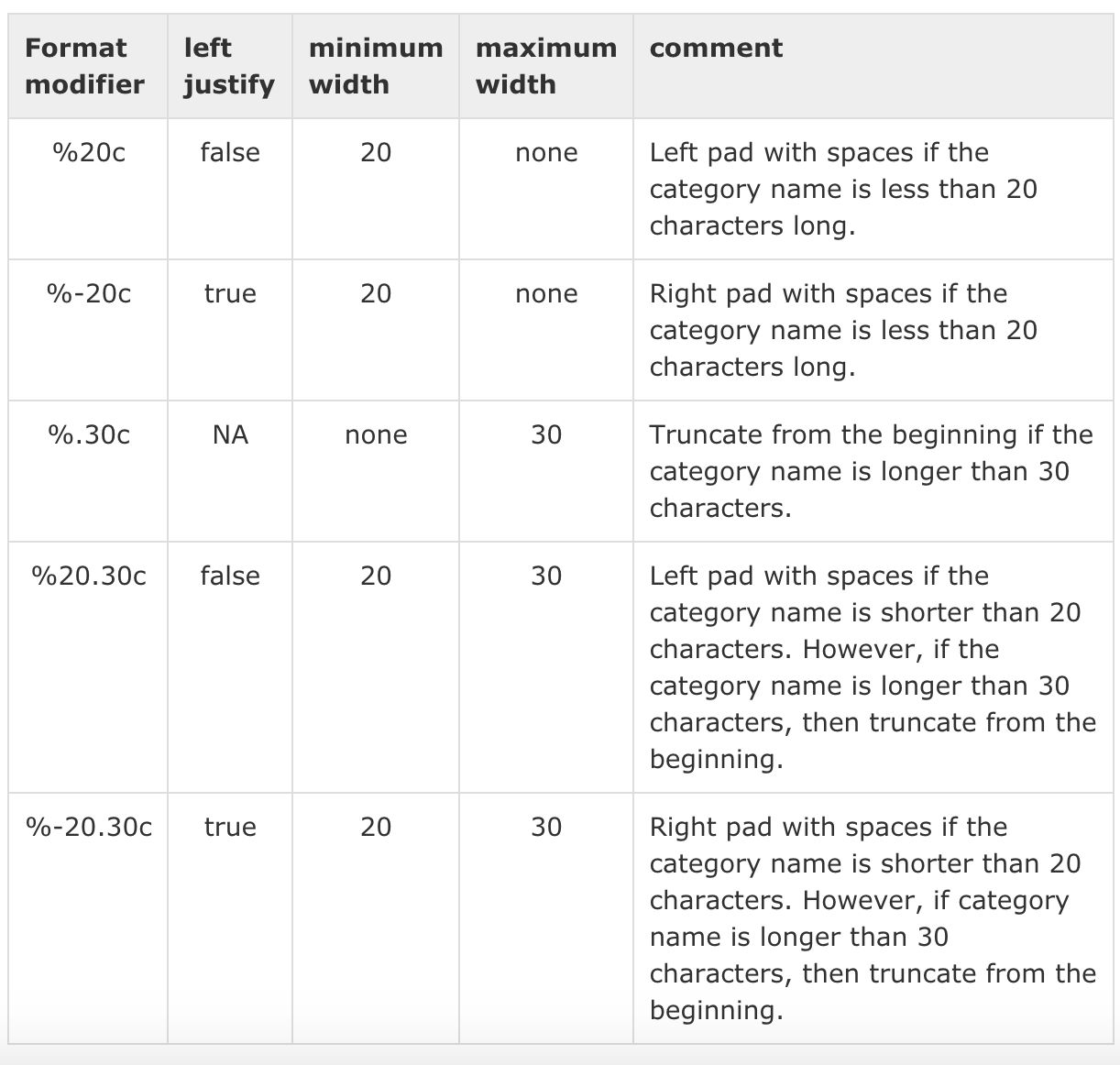
The following table explains the characters used in the above pattern and all other characters that you can use in your custom pattern:



**Format Modifiers**

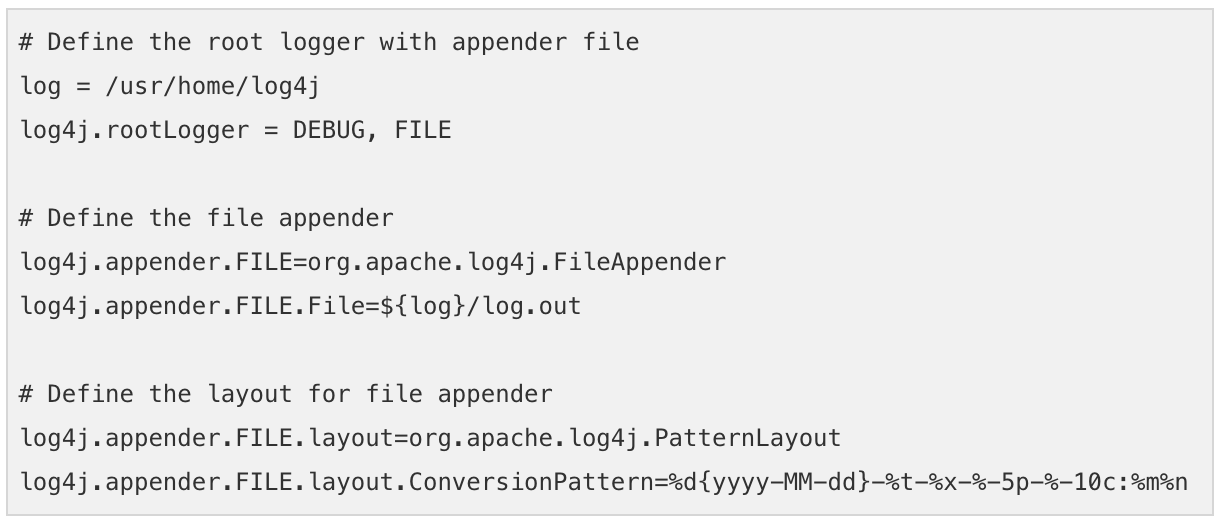
By default, the relevant information is displayed as output as is. However, with the aid of format modifiers, it is possible to change the minimum field width, the maximum field width, and justification.

Following table covers various modifiers scenarios:



**PatternLayout Example**

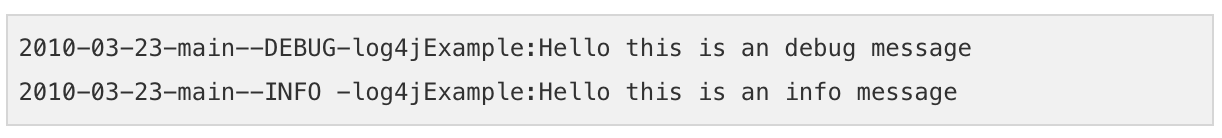
Following is a simple configuration file for PatternLayout:



Now consider the following Java Example which would generate logging information:



Compile and run the above program. It would create a log.out file in /usr/home/log4j directory which would have the following log information:



**HTMLLayout**

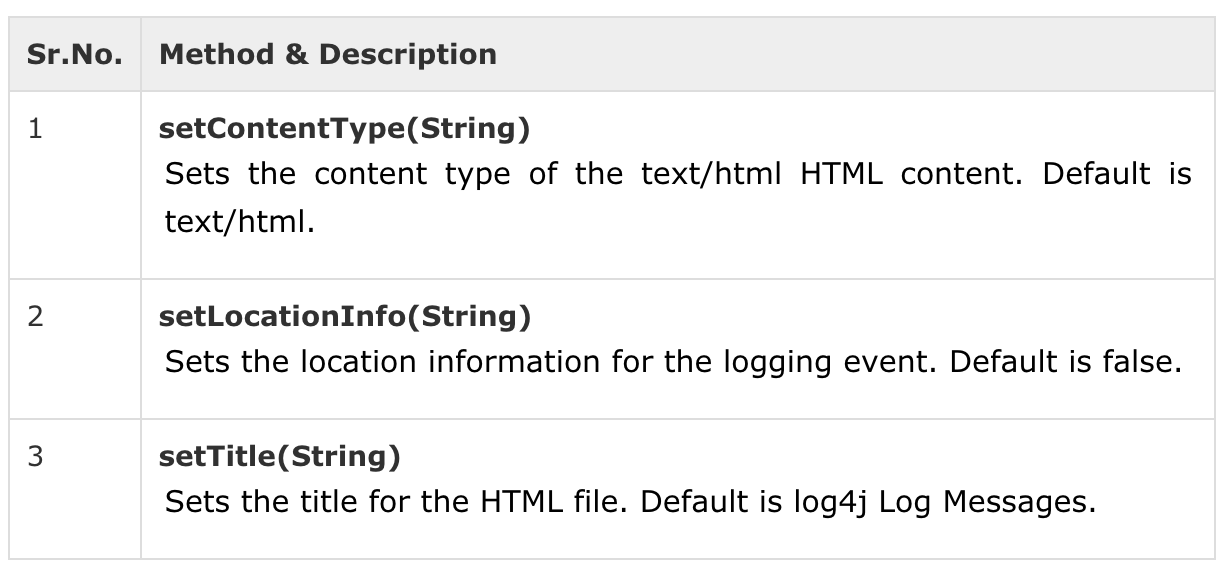
If you want to generate your logging information in an HTML-formatted file, then you can use org.apache.log4j.HTMLLayout to format your logging information.

The HTMLLayout class extends the abstract org.apache.log4j.Layout class and overrides the format() method from its base class to provide HTML-style formatting.

It provides the following information to be displayed:

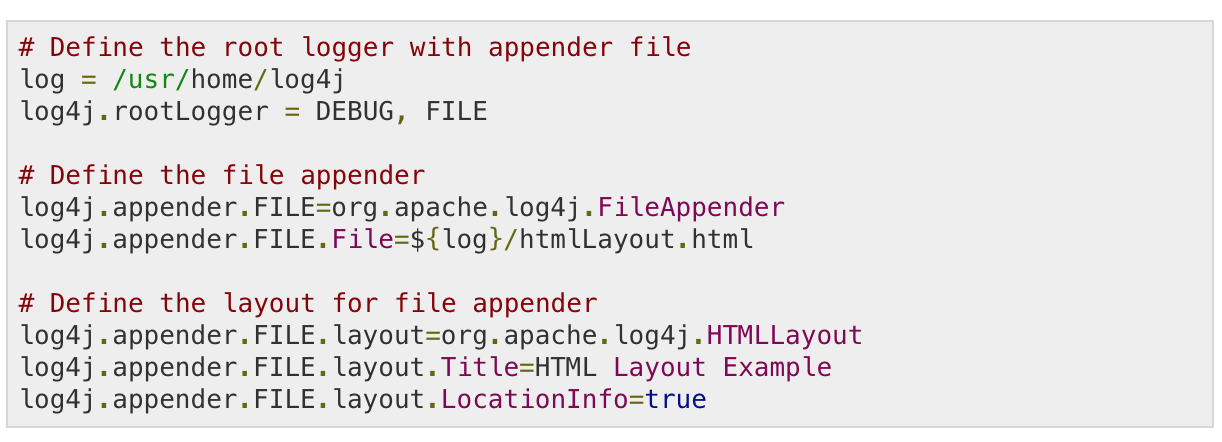
* The time elapsed from the start of the application before a particular logging event was generated.
* The name of the thread that invoked the logging request.
* The level associated with this logging request.
* The name of the logger and logging message.
* The optional location information for the program file and the line number from which this logging was invoked.

HTMLLayout is a very simple Layout object that provides the following methods:



**HTMLLayout Example**

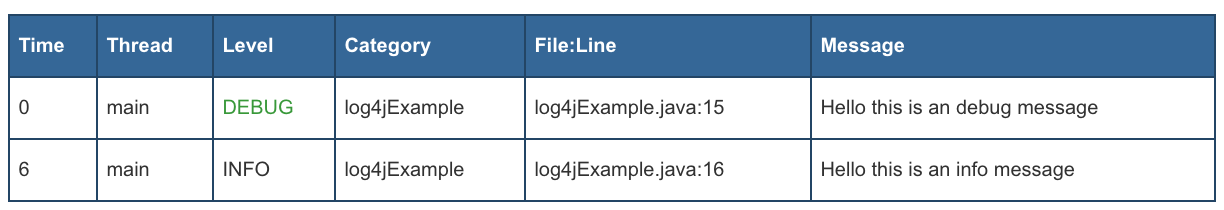
Following is a simple configuration file for HTMLLayout:



Now consider the following Java Example which would generate logging information:



Compile and run the above program. It would create an htmlLayout.html file in /usr/home/log4j directory which would have the following log information:



You would use a web browser to open htmlLayout.html file. It is also important to note that the footer for the </html> and </body> tags is completely missing.

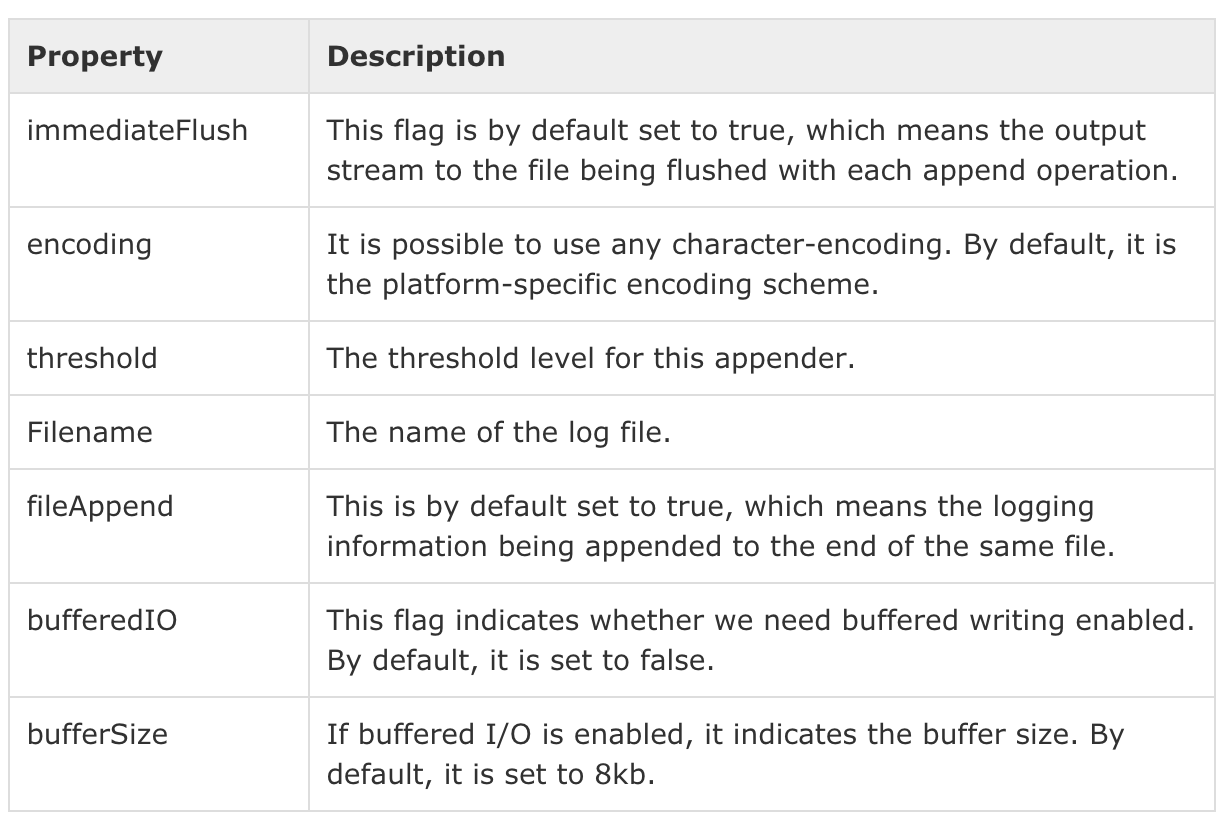
One of the big advantages of having the log file in HTML format is that it can be published as a web page for remote viewing.

**Logging in Files**

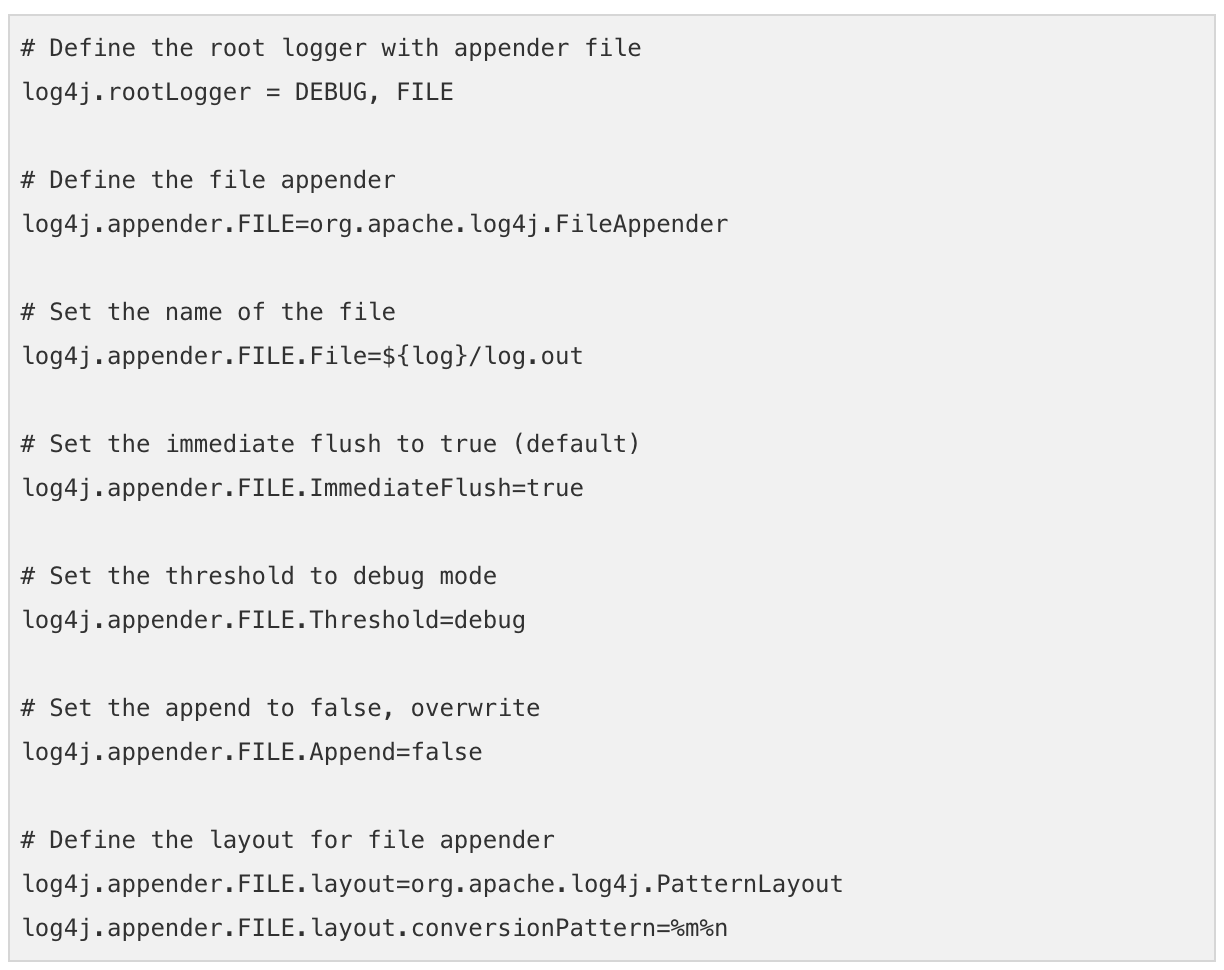
To write your logging information into a file, you would have to use org.apache.log4j.FileAppender.

**FileAppender Configuration**

FileAppender has the following configurable parameters:



Following is a sample configuration file log4j.properties for FileAppender:



If you wish to have an XML configuration file equivalent to the above log4j.properties file, then here is the content:

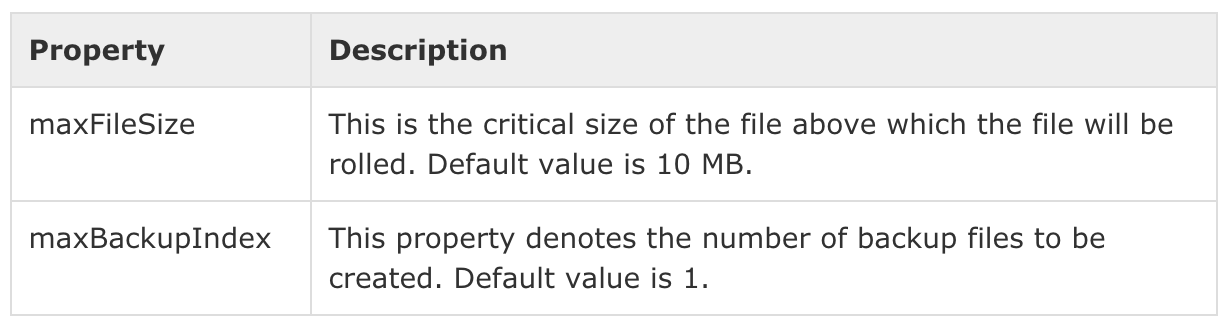


**Logging in Multiple Files**

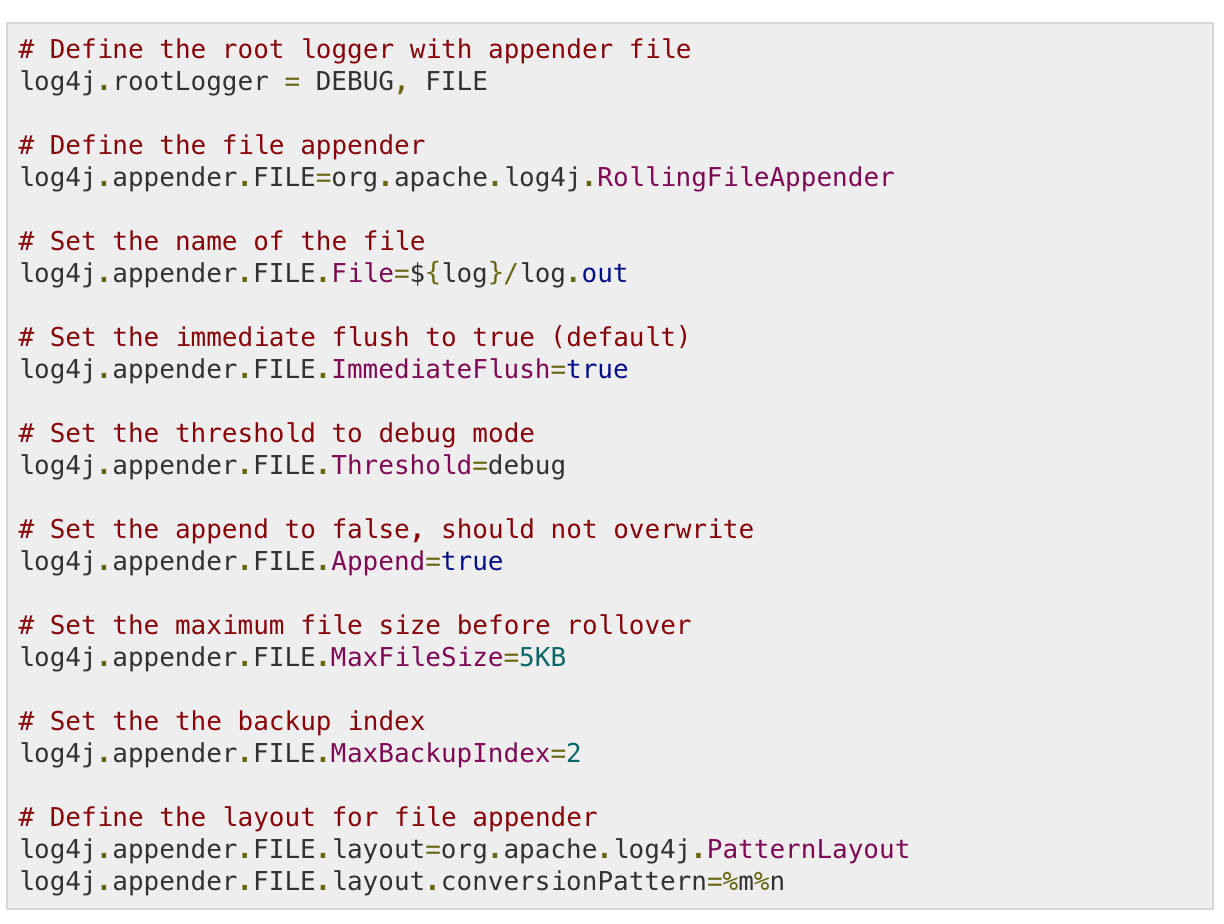
You may want to write your log messages into multiple files for certain reasons, for example, if the file size reached to a certain threshold.

To write your logging information into multiple files, you would have to use org.apache.log4j.RollingFileAppender class which extends the FileAppender class and inherits all its properties.

We have the following configurable parameters in addition to the ones mentioned above for FileAppender:



Following is a sample configuration file log4j.properties for RollingFileAppender.



If you wish to have an XML configuration file, you can generate the same as mentioned in the initial section and add only additional parameters related to RollingFileAppender.

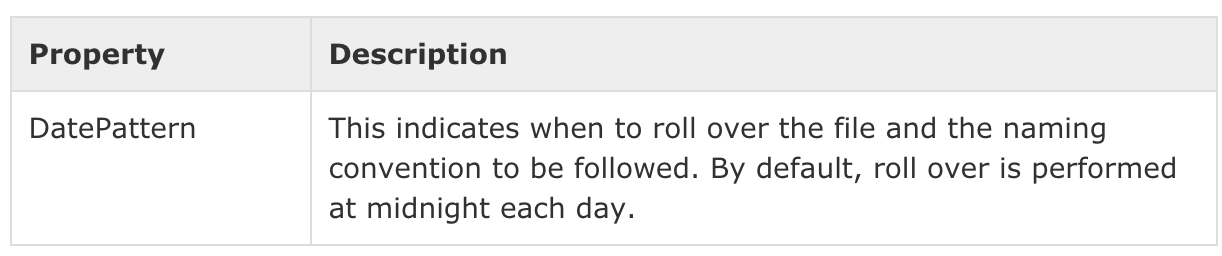
This example configuration demonstrates that the maximum permissible size of each log file is 5 KB. Upon exceeding the maximum size, a new log file will be created. Since maxBackupIndex is defined as 2, once the second log file reaches the maximum size, the first log file will be erased and thereafter, all the logging information will be rolled back to the first log file.

**Daily Log File Generation**

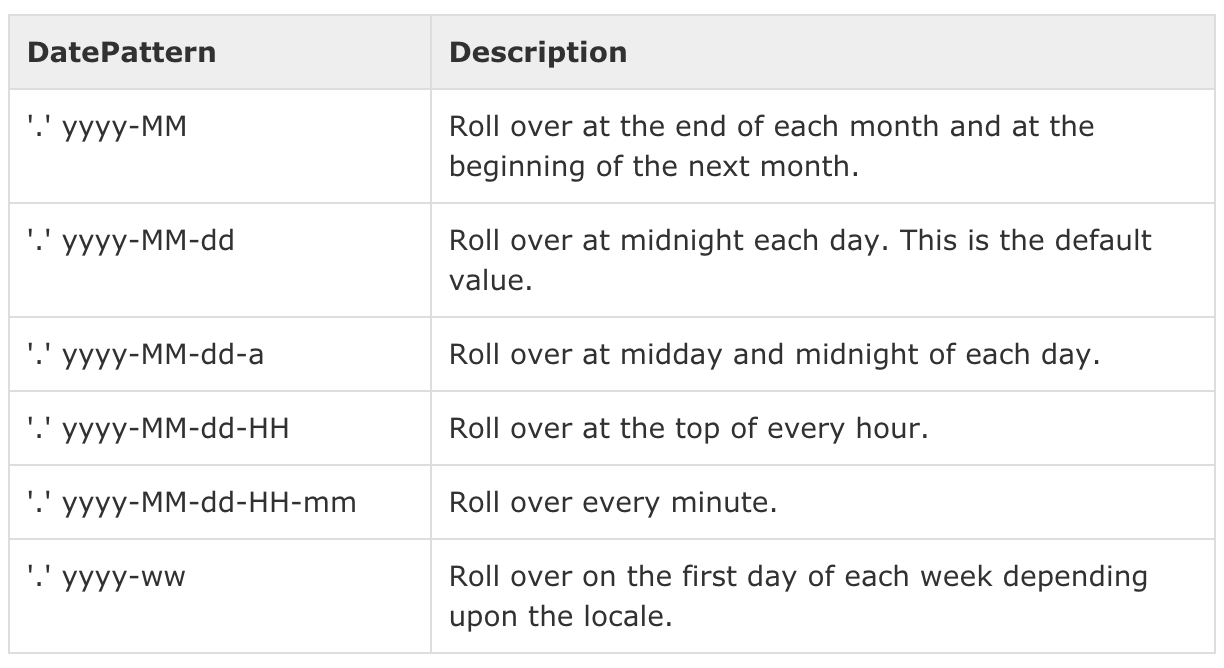
There may be a requirement to generate your log files on a daily basis to keep a clean record of your logging information.

To write your logging information into files on a daily basis, you would have to use org.apache.log4j.DailyRollingFileAppender class which extends the FileAppender class and inherits all its properties.

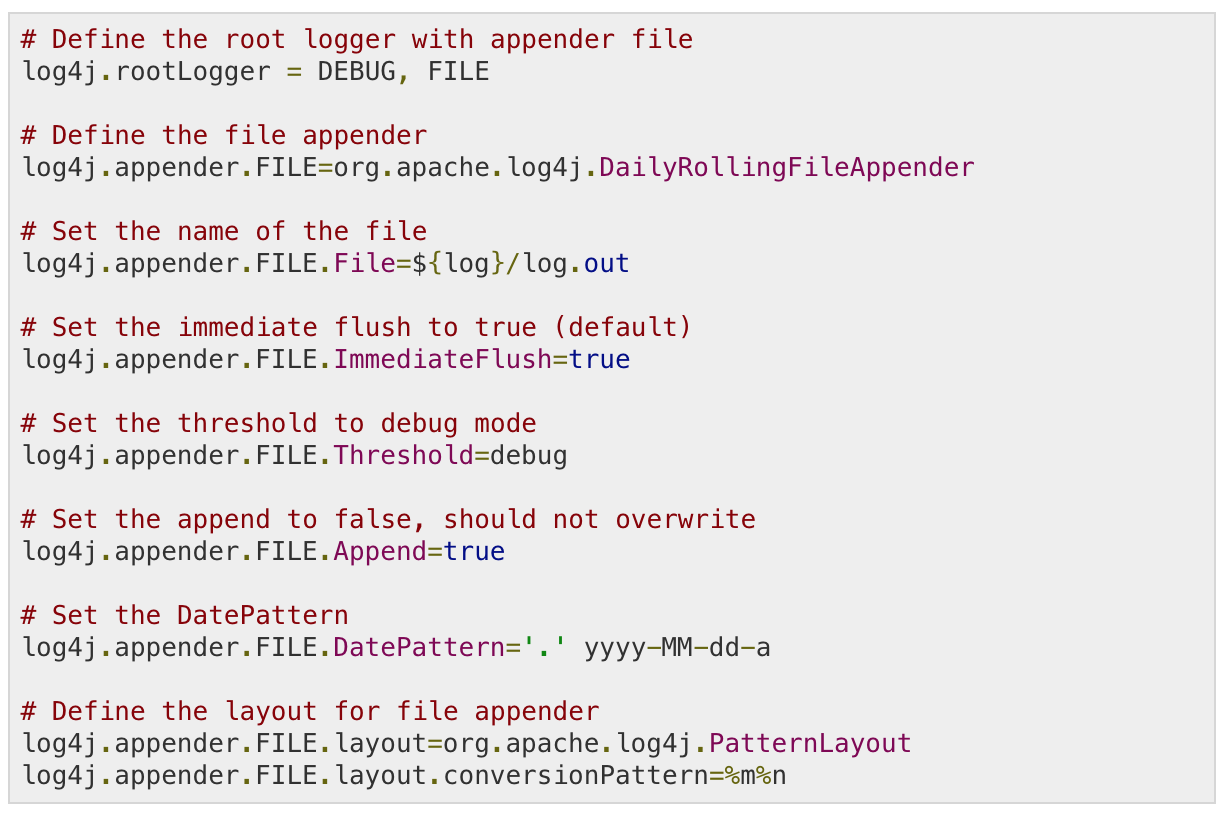
There is only one important configurable parameter in addition to the ones mentioned above for FileAppender:



DatePattern controls the rollover schedule using one of the following patterns:



Following is a sample configuration file log4j.properties to generate log files rolling over at midday and midnight of each day.

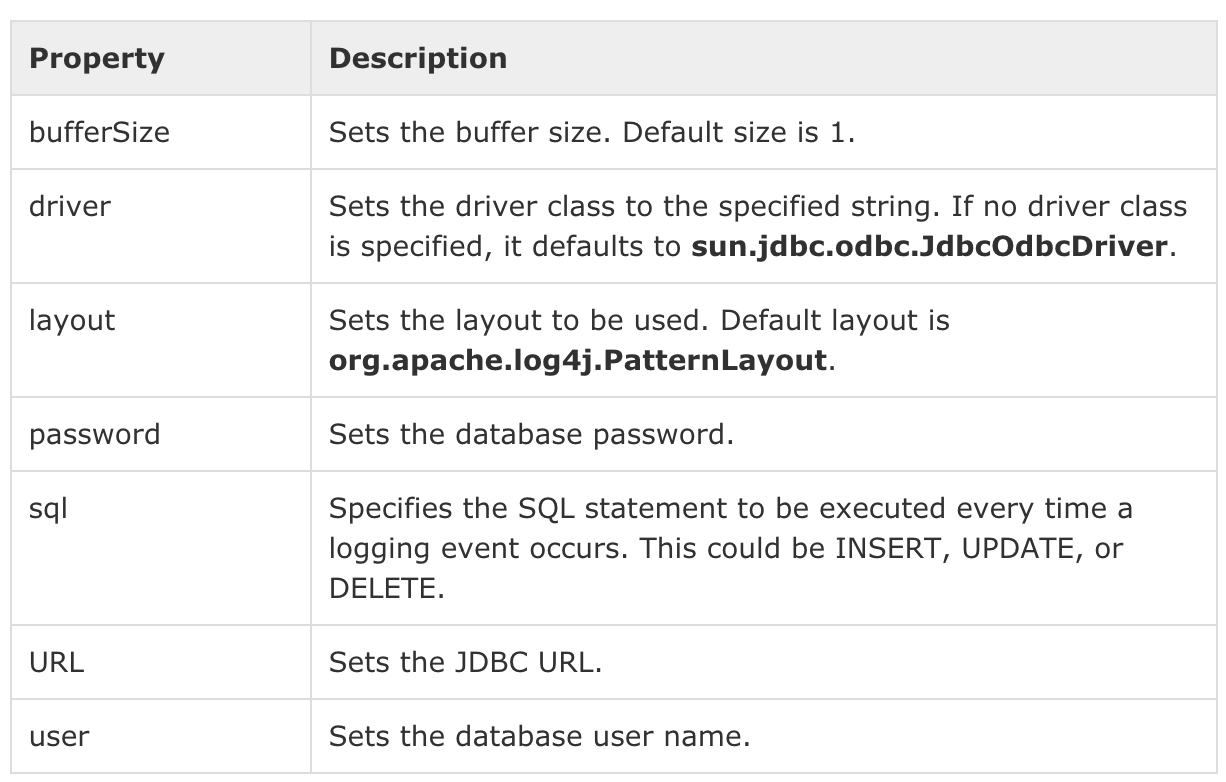


If you wish to have an XML configuration file, you can generate the same as mentioned in the initial section and add only additional parameters related to DailyRollingFileAppender.

**Logging in Database**

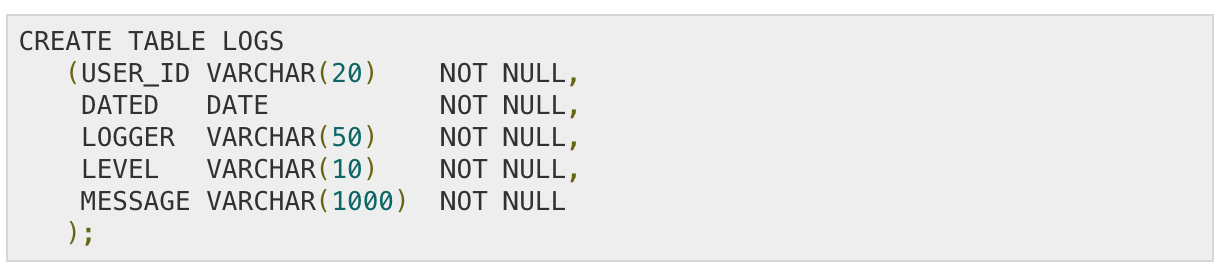
The log4j API provides the org.apache.log4j.jdbc.JDBCAppender object, which can put logging information in a specified database.

**JDBCAppender Configuration**



**Log Table Configuration**

Before you start using JDBC based logging, you should create a table to maintain all the log information. Following is the SQL Statement for creating the LOGS table:



**Sample Configuration File**

Following is a sample configuration file log4j.properties for JDBCAppender which will is be used to log messages to a LOGS table.



For MySQL database, you would have to use the actual DBNAME, user ID and password, where you have created LOGS table. The SQL statement is to execute an INSERT statement with the table name LOGS and the values to be entered into the table.

JDBCAppender does not need a layout to be defined explicitly. Instead, the SQL statement passed to it uses a PatternLayout.

If you wish to have an XML configuration file equivalent to the above log4j.properties file, then here is the content:



**Sample Program**

The following Java class is a very simple example that initializes and then uses the Log4J logging library for Java applications.



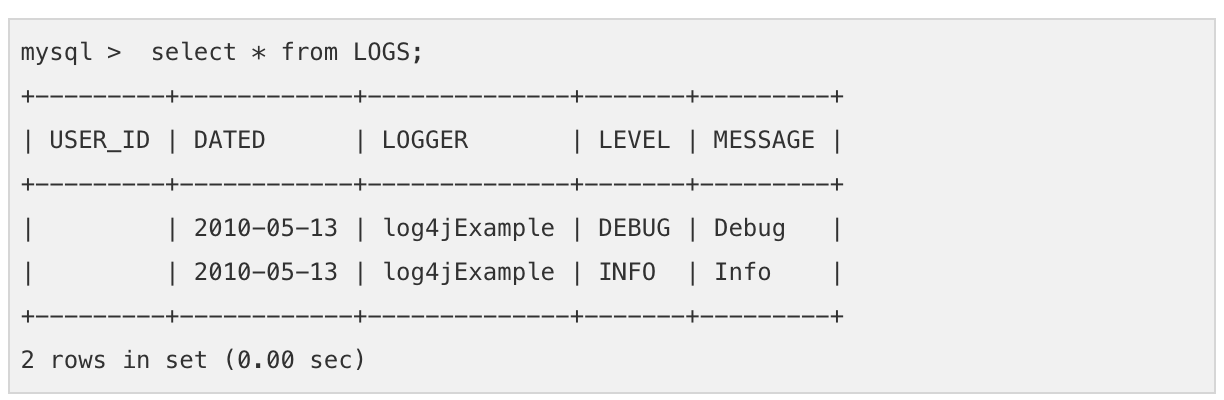
**Compile and Execute**

Here are the steps to compile and run the above-mentioned program. Make sure you have set PATH and CLASSPATH appropriately before proceeding for compilation and execution.

All the libraries should be available in CLASSPATH and your log4j.properties file should be available in PATH. Follow the given steps:

* Create log4j.properties as shown above.
* Create log4jExample.java as shown above and compile it.
* Execute log4jExample binary to run the program.

Now check your LOGS table inside DBNAME database and you would find the following entries:



Note − Here **x** is used to output the Nested diagnostic Context (NDC) associated with the thread that generated the logging event. We use NDC to distinguish clients in server-side components handling multiple clients. Check Log4J Manual for more information on this.

Log4j <https://www.tutorialspoint.com/log4j/log4j_quick_guide.htm>

Junit <https://www.tutorialspoint.com/junit/junit_quick_guide.htm>