

Tutoriales de Programacion Java

Blog dedicado a temas de programación actuales usando el lenguaje de programación Java y las últimas versiones de sus APIs y Herramientas.

[Página principal](#) [Página de Concursos](#) [Tutoriales UPAO 2010](#) [Presentaciones Capacitación](#)

SÁBADO, 23 DE ABRIL DE 2011

Log4j para Creación de Eventos de Log

El uso de logging o de bitácoras dentro del contexto de desarrollo de aplicaciones constituye insertar sentencias dentro de la aplicación, que proporcionan un tipo de información de salida que es útil para el desarrollador. Un ejemplo de logging son las sentencias de rastreo o de seguimiento que colocamos en la aplicación para asegurarnos de que esta haya pasado por los flujos adecuados. Esto normalmente es realizado usando "`System.out.println`".

El problema con este tipo de sentencias, es que algunas veces olvidamos quitarlas después de que han servido a su propósito específico, y terminamos con una aplicación que genera cientos o miles de salidas innecesarias a la consola. Otro problema es que todas las llamadas tienen el mismo significado, es decir, no podemos saber exactamente cuál sentencia representa una salida de debug (que seguramente olvidamos quitar), cuál representa una salida de información útil, y cuál representa un error que se ha generado; las tres se ven exactamente igual para nosotros.

En este tutorial aprenderemos cómo usar `log4j`, un framework especializado para el logging o creación de bitácoras en las aplicaciones Java. También veremos dos formas distintas de configurarlo y las diferencias entre ambas formas.

`log4j` es un framework que ofrece una forma jerárquica de insertar sentencias de log dentro de una aplicación Java. Con él, se tienen disponibles múltiples formatos de salida, y múltiples niveles de información de log.

Usando un paquete dedicado para realizar el log, se elimina la carga de mantener cientos de sentencias "`System.out.println`", al mismo tiempo que el logging puede ser controlado en tiempo de ejecución por scripts de configuración.

`log4j` tiene tres componentes principales:

- **Loggers**
- **Appenders**
- **Layouts**

Estos tres tipos de componentes trabajan juntos para permitir a los desarrolladores hacer el log de mensajes de acuerdo al tipo y nivel de mensaje, controlar en tiempo de ejecución como es que estos mensajes son formateados y dónde son reportados.

Loggers

Los **loggers** son los componentes más esenciales del proceso de logging. Son los responsables de capturar la información de logging.

Los **loggers** son entidades con nombre. Los nombres de los **loggers** son case-sensitive y siguen una regla jerárquica de nombres: Se dice que un logger es un ancestro de otro **logger** si su nombre, seguido de un punto, es un prefijo del nombre del **logger** descendiente. Un **logger** se dice que es padre de un **logger** hijo si no hay ancestros entre él mismo y el **logger** descendiente.

Por ejemplo, el **logger** "`com.foo`" es padre del **logger** "`com.foo.Bar`". De forma similar, "`java`" es pariente de "`java.util`" y ancestro de "`java.util.Vector`".

Esto es importante porque, como veremos más adelante, cuando realizamos una configuración para un **logger**, digamos para "`java.util`", esta configuración es heredada por todos sus descendientes. Normalmente tenemos un **logger** por cada una de las clases de las cuales nos interesa obtener información, y **logger** tiene **el mismo nombre que la clase**.

Existe un **logger** especial llamado "`rootLogger`", el cual reside en la cima de la jerarquía de **loggers**. Este **logger** tiene dos características:

- Siempre existe
- No puede ser recuperado por nombre

Los **loggers** pueden tener niveles asignados. Los niveles normales que puede tener un **logger** son, de menor a mayor prioridad:

- **TRACE**: Se usa para información más detallada que el nivel debug.
- **DEBUG**: Se utiliza para mensajes de información detallada que son útiles para debugear una aplicación.
- **INFO**: Se utiliza para mensajes de información que resaltan el progreso de la aplicación de una forma general.
- **WARN**: Se utiliza para situaciones que podrían ser potencialmente dañinas.
- **ERROR**: Se usa para eventos de error que podrían permitir que la aplicación continúe ejecutándose.
- **FATAL**: Se usa para errores muy graves, que podrían hacer que la aplicación dejara de funcionar.

Además hay otros dos niveles especiales que son:

- **ALL**: Tiene el nivel más bajo posible y se usa para activar todo el logging.
- **OFF**: Tiene el nivel más alto posible y se usa para evitar cualquier mensaje de log

Si a un **logger** no se le asigna un nivel de forma explícita, este hereda el de su ancestro más cercano. Si no tiene ancestros, hereda el del **rootLogger**, que por default es **DEBUG**.

Las peticiones de logging son hechas invocando uno de los métodos de impresión de la instancia de logger. Estos métodos son: "`trace`", "`debug`", "`info`", "`warn`", "`error`", y "`fatal`".

Por definición, cada método de impresión determina el nivel de una petición de logging. Por ejemplo "`info`" es una petición de nivel **INFO**.

Una petición de logging está habilitada si su nivel es mayor que o igual que el nivel de su **logger**. De otra forma se dice que la petición está deshabilitada. El comportamiento de los **loggers** es jerárquico. La siguiente tabla ilustra esto:

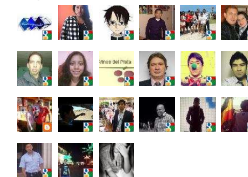
DONACIONES



JAVA TUTORIALES EN FACEBOOK

SEGUIDORES

Seguidores (196) [Siguiente](#)



[Seauir](#)

ARCHIVO DEL BLOG

- 2016 (1)
- 2015 (2)
- 2013 (1)
- 2012 (2)
- ▼ 2011 (11)
 - diciembre (1)
 - noviembre (1)
 - octubre (1)
 - junio (2)
 - ▼ abril (1)
 - [Log4j para Creación de Eventos de Log](#)
 - febrero (4)
 - enero (1)
- 2010 (10)
- 2009 (22)

DATOS PERSONALES

Alex

Programador Java con algunos años de experiencia en múltiples poyectos y con múltiples APIs y herramientas deseoso de compartir experiencias con el resto de programadores.

[Ver todo mi perfil](#)

Nivel de los mensajes que se mostrarán					
	DEBUG	INFO	WARN	ERROR	FATAL
DEBUG					
INFO					
WARN					
ERROR					
FATAL					
Nivel del Logger					
ALL					
OFF					

Esto quiere decir que si establecemos el nivel de log en "DEBUG" se mostrarán los mensajes de nivel "DEBUG", "INFO", "WARN", "ERROR", y "FATAL". Si, por el contrario, establecemos el nivel en "ERROR", solo se mostrarán los mensajes de nivel "ERROR", y "FATAL".

Appenders

Log4j permite a las peticiones de logging envíen sus mensajes a múltiples destinos. En lenguaje de log4j, un destino de salida es llamado un **appender**. Existen **appenders** para la consola, archivos, sockets, JMS, correo electrónico, bases de datos, etc. Además también es posible crear **appenders** propios.

Los **appenders** disponibles son:

- **AsyncAppender**: Permite enviar los mensajes de log de forma asincrónica. Este appender recolectará los mensajes enviados a los **appenders** que estén asociados a él y los enviará de forma asincrónica.
- **ConsoleAppender**: Envía los eventos de log a `System.out` or `System.err`.
- **FileAppender**: Envía los eventos de log a un archivo.
- **DailyRollingFileAppender**: Extiende **FileAppender** de modo que el archivo es creado nuevamente con una frecuencia elegida por el usuario.
- **RollingFileAppender**: Extiende **FileAppender** para respaldar los archivos de log cuando alcanza un tamaño determinado.
- **ExternallyRolledFileAppender**: Escucha en por un socket y cuando recibe un mensaje, lo agrega en un archivo, después envía una confirmación al emisor del mensaje.
- **JDBCAppender**: Proporciona un mecanismo para enviar los mensajes de log a una base de datos.
- **JMSAppender**: Publica mensajes de log como un tópico JMS.
- **LF5Appender**: Envía los mensajes de log a una consola basada en swing.
- **NTEventLogAppender**: Agrega los mensajes a un sistema de eventos NT (solo para Windows).
- **NullAppender**: Un **appender** que no envía los mensajes a ningún lugar.
- **WriterAppender**: Envía los eventos de log a un **Writer** o a un **OutputStream** dependiendo de la elección del usuario.
- **SMTPAppender**: Envía un e-mail cuando ocurre un evento específico de logging, típicamente errores o errores fatales.
- **SocketAppender**: Envía un objeto **LoggingEvent** a un servidor remoto, usualmente un **SocketNode**.
- **SocketHubAppender**: Envía un objeto **LoggingEvent** a un servidor remoto de log.
- **SyslogAppender**: Envía mensajes a un demonio **syslog** remoto.
- **TelnetAppender**: Es un **appender** que se especializa en escribir a un socket de solo lectura.

Un **logger** puede tener asociado más de un appender.

Layouts

Son los responsables de dar formatos a los mensajes de salida, de acuerdo a lo que el usuario quiera.

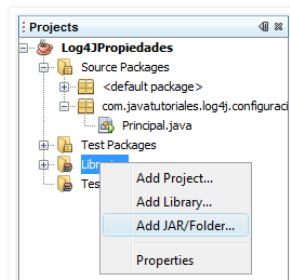
Permiten presentar el mensaje con formato para mostrarlo en la consola o guardarlo en un archivo de texto, en una tabla **HTML**, un archivo **XML**, etc.

Ahora veremos cómo hacer uso de todo esto en un ejemplo.

Lo primero que debemos hacer es descargar la última versión de log4j, que actualmente es la "1.2.16". Podemos descargarla de el [sitio de log4j](#). Una vez que realicemos la descarga extraemos, del archivo comprimido, el jar "log4j-1.2.16.jar" que es el único necesario.

Ahora creamos un nuevo proyecto en **NetBeans**. Para esto vamos al Menú "File->New Project...". En la ventana que se abre seleccionamos la categoría "Java" y en el tipo de proyecto "Java Application". Le damos una ubicación y un nombre al proyecto, en mi caso será "log4j". Nos aseguramos que las opciones "Create Main Class" y "Set as Main Project" estén habilitadas. Presionamos el botón "Finish" y veremos aparecer en el editor nuestra clase "Main". En mi caso la clase "Main" se llamará "Principal".

Lo siguiente que debemos hacer es agregar el jar de log4j a nuestro proyecto, para esto hacemos clic derecho sobre el nodo "Libraries" de nuestro proyecto y hacemos clic sobre la opción "Add JAR/Folder..." y en la ventana que aparece seleccionamos el jar de log4j.



El siguiente paso es crear un **logger**. Hay muchas formas de crear un **logger**, todas ellas a través de métodos estáticos de la clase "Logger". La primera es recuperar el **rootLogger**:

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

También es posible crear un nuevo `logger`, asignándole un nombre:

```
Logger log = Logger.getLogger("MiLogger");
```

Lo más usual es instanciar un `logger` **estático** global para la clase, basado en **el nombre de la clase**. `Logger` proporciona una versión sobrecargada de `getLogger` para esto:

```
private static Logger log = Logger.getLogger(Principal.class);
```

Una vez que ya se tiene un `logger`, lo siguiente es invocar sus **métodos de petición de impresión** para que los mensajes con los niveles correspondientes sean enviados al **appender** que definamos:

```
public static void main(String[] args)
{
    log.trace("mensaje de trace");
    log.debug("mensaje de debug");
    log.info("mensaje de info");
    log.warn("mensaje de warn");
    log.error("mensaje de error");
    log.fatal("mensaje de fatal");
}
```

Unas cuantas notas para un **uso correcto** de estos métodos ^_^:

Existe una versión sobrecargada de cada uno de estos métodos, en los que además de un mensaje (en realidad el **"mensaje"** puede ser cualquier objeto que nosotros queramos) acepta un objeto de tipo **"Throwable"** (o sea cualquier excepción o error que ocurra). Esto es especialmente útil con los métodos **"warn"**, **"error"**, y **"fatal"**, con lo que además del mensaje también se mostrará el stack trace del error que haya ocasionado problemas.

Los métodos **"error"** y **"fatal"** normalmente son colocados dentro de bloques **"catch"** ya que es aquí donde podemos obtener la información sobre los errores que ocurren en nuestra aplicación (claro, si es que los estamos manejando de una forma adecuada ^_^!).

La ejecución de cada uno de los métodos es, en teoría bastante rápida (se supone que entre 1 y 4 milisegundos). Sin embargo, la generación del mensaje (la cual puede incluir concatenación de cadenas, invocación de métodos, o cualquier otra cosa que se nos ocurra) puede tardar más que esto. Dependiendo del nivel de log en el que esté configurada nuestra aplicación, no siempre queremos esta carga adicional de tiempo para la generación de un mensaje que nunca veremos. Por ejemplo, podemos tener muchos métodos **"debug"** en nuestra aplicación, los cuales realizan algunas concatenaciones de cadenas para verificar que todos los valores sean los que esperamos, pero el nivel de la misma puede estar establecido en **"warn"** por lo que se perderá mucho tiempo generando estos mensajes de **"debug"** que finalmente nunca aparecerán en ningún lado.

A causa de esto, el objeto `Logger` contiene una serie de métodos que nos permite saber si el `logger` está configurado para un nivel de **"TRACE"**, **"DEBUG"**, o **"INFO"** (normalmente los niveles de **WARN** a **FATAL** siempre queremos mostrarlos si es que ocurre algún problema). Estos métodos son:

- `isTraceEnabled()`
- `isDebugEnabled()`
- `isInfoEnabled()`

Normalmente envolvemos las invocaciones a los métodos **"trace"**, **"debug"**, e **"info"** en un **"if"** que verificará si el nivel de `logger` está establecido para aceptar estas llamadas. Modificaremos el código del método **"main"** para hacer uso de estas condiciones:

```
public static void main(String[] args)
{
    if (log.isTraceEnabled())
    {
        log.trace("mensaje de trace");
    }

    if (log.isDebugEnabled())
    {
        log.debug("mensaje de debug");
    }

    if (log.isInfoEnabled())
    {
        log.info("mensaje de info");
    }

    log.warn("mensaje de warn");
    log.error("mensaje de error");
    log.fatal("mensaje de fatal");
}
```

Ahora es necesario configurar `log4j` para realizar la asociación entre el `logger` que acabamos de crear, con los **appenders** a los que enviará los mensajes, y los **layouts** que usarán para mostrar estos mensajes.

Nota: Yo usaré dos proyectos, uno con un archivo de propiedades y otro con un archivo XML, para que el código de ambos no se mezcle.

Nota2: Algunas veces al hacer cambios, ya sea en archivos XML o de propiedades, NetBeans no ve estos cambios y por lo tanto no recarga estos archivos. Para que los cambios tengan efecto algunas veces es necesario ejecutar un "clean and build".

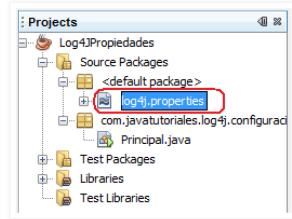
Configuración

El ambiente de **log4j** es completamente configurable de forma programática. Sin embargo, es más flexible configurar **log4j** usando archivos de configuración. Esto permite que una vez que las sentencias de log están colocadas en su lugar, se puedan controlar fácilmente usando el archivo de configuración, sin tener que modificar el código fuente.

Los archivos de configuración pueden ser escritos en XML o en archivos de propiedades. Primero veremos cómo realizar la configuración con archivos de propiedades.

CONFIGURACIÓN DE LOG4J CON UN ARCHIVO DE PROPIEDADES

Si queremos configurar **log4j** haciendo uso de un archivo de propiedades, lo primero que debemos hacer es crear un archivo de propiedades (**properties file**) en el paquete default de la aplicación. Este archivo debe llamarse "**log4j.properties**" (podríamos llamar este archivo de otra forma o colocarlo en otro lugar, pero **log4j** busca por default este archivo en esta ubicación, haciéndolo de esta forma nos evitamos el tener realizar una configuración de forma programática en nuestra aplicación):



Lo primero que hay que hacer es crear un **appender**, al cual le daremos un nombre arbitrario, e indicaremos el tipo de **appender** que será, usando una de las clases que mencionamos hace un momento, cuando hablamos de los **appenders**. En este caso se creará un **appender** llamado "consola" y que será de tipo "**org.apache.log4j.ConsoleAppender**", que permite enviar los mensajes a la salida estándar, además indicaremos que los mensajes deberán mostrarse en "**System.out**", que es su valor por default (también podemos mandar los mensajes a "**System.err**");

```
log4j.appender.consola = org.apache.log4j.ConsoleAppender
log4j.appender.consola.target = System.out
```

Lo que sigue es agregar un **layout** para este **appender**. Existen muchos tipos de **layouts**. Para esta primera prueba usaremos el "**SimpleLayout**", que simplemente nos indica el nivel de log del **logger**, y el mensaje asociado al evento:

```
log4j.appender.consola.layout = org.apache.log4j.SimpleLayout
```

Una vez que se ha creado el **appender**, y que se le ha establecido su **layout**, se debe especificar cuáles **loggers** usarán este **appender**, y establecer su nivel de log. Si se especifica que "**rootLogger**" hará uso de este **appender**, entonces todos los **loggers** enviarán sus mensajes a este **appender** (a menos que se especifique lo contrario de forma explícita). Como **rootLogger** está en la cima de la jerarquía, todos los **loggers** heredarán su nivel de log y su **appender**:

```
log4j.rootLogger = TRACE, consola
```

El archivo de configuración queda de la siguiente forma:

```
log4j.appender.consola = org.apache.log4j.ConsoleAppender
log4j.appender.consola.target = System.out
log4j.appender.consola.layout = org.apache.log4j.SimpleLayout
log4j.rootLogger = TRACE, consola
```

Al ejecutar el código anterior se obtiene la siguiente salida:

```
TRACE - mensaje de trace
DEBUG - mensaje de debug
INFO - mensaje de info
WARN - mensaje de warn
ERROR - mensaje de error
FATAL - mensaje de fatal
```

Ahora, teniendo este archivo de configuración, es posible modificar toda la configuración de la salida. Por ejemplo, modificamos el layout del appender "console" para que en vez de ser de tipo "SimpleLayout" ahora sea de tipo "HTMLLayout":

```
log4j.appender.console.layout = org.apache.log4j.HTMLLayout
```

Obtendremos la siguiente salida:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Log4j Log Messages</title>
<style type="text/css">
<!--
body, table {font-family: arial,sans-serif; font-size: x-small;}
th {background: #336699; color: #FFFFFF; text-align: left;}
-->
</style>
</head>
<body bgcolor="#FFFFFF" topmargin="6" leftmargin="6">
<hr size="1" noshade>
Log session start time Fri Apr 22 01:59:55 CDT 2011<br>
<br>
<table cellspacing="0" cellpadding="4" border="1" bordercolor="#224466" width="100%">
<tr>
<th>Time</th>
<th>Thread</th>
<th>Level</th>
<th>Category</th>
<th>Message</th>
</tr>

<tr>
<td>0</td>
<td title="main thread">main</td>
<td title="Level">TRACE</td>
<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal category</td>
<td title="Message">mensaje de trace</td>
</tr>

<tr>
<td>3</td>
<td title="main thread">main</td>
<td title="Level"><font color="#339933">DEBUG</font></td>
<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal category</td>
<td title="Message">mensaje de debug</td>
</tr>

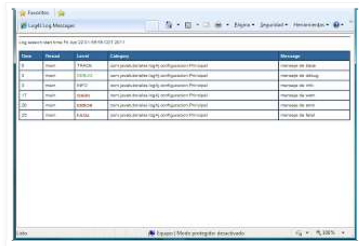
<tr>
<td>3</td>
<td title="main thread">main</td>
<td title="Level">INFO</td>
<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal category</td>
<td title="Message">mensaje de info</td>
</tr>

<tr>
<td>17</td>
<td title="main thread">main</td>
<td title="Level"><font color="#993300"><strong>WARN</strong></font></td>
<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal category</td>
<td title="Message">mensaje de warn</td>
</tr>

<tr>
<td>20</td>
<td title="main thread">main</td>
<td title="Level"><font color="#993300"><strong>ERROR</strong></font></td>
<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal category</td>
<td title="Message">mensaje de error</td>
</tr>

<tr>
<td>25</td>
<td title="main thread">main</td>
<td title="Level"><font color="#993300"><strong>FATAL</strong></font></td>
<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal category</td>
<td title="Message">mensaje de fatal</td>
</tr>
```

Que genera la siguiente página:



Time	Level	Category	Message
1	TRACE	com.javatutoriales.log4j.configuracion.Principal	Mensaje de trace
2	DEBUG	com.javatutoriales.log4j.configuracion.Principal	Mensaje de debug
3	INFO	com.javatutoriales.log4j.configuracion.Principal	Mensaje de info
4	WARN	com.javatutoriales.log4j.configuracion.Principal	Mensaje de warn
5	ERROR	com.javatutoriales.log4j.configuracion.Principal	Mensaje de error
6	FATAL	com.javatutoriales.log4j.configuracion.Principal	Mensaje de fatal

Como podemos ver, hemos modificado completamente la salida, sin necesidad de mover una sola línea de la aplicación (bueno, solo una línea de un archivo de propiedades, que siempre será un archivo de texto plano).

De la misma forma, si usamos un [XMLLayout](#):

```
log4j.appender.console.layout = org.apache.log4j.xml.XMLLayout
```

Otendremos la siguiente salida:

```
<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881257" level="TRACE">
  <log4j:message><![CDATA[mensaje de trace]]></log4j:message>
</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881259" level="DEBUG">
  <log4j:message><![CDATA[mensaje de debug]]></log4j:message>
</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881259" level="INFO">
  <log4j:message><![CDATA[mensaje de info]]></log4j:message>
</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881259" level="WARN">
  <log4j:message><![CDATA[mensaje de warn]]></log4j:message>
</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881262" level="ERROR">
  <log4j:message><![CDATA[mensaje de error]]></log4j:message>
</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881262" level="FATAL">
  <log4j:message><![CDATA[mensaje de fatal]]></log4j:message>
</log4j:event>
```

Hagamos un último cambio de `layout`. Con los `layout` anteriores, la información que obtenemos podría no cumplir con nuestras expectativas, por ejemplo, no hay forma de obtener información de qué método fue el que envió el mensaje, o usar un formato propio para la fecha, o agregar alguna información constante (alguna marca por ejemplo) en la salida generada. Para estos casos, en los que necesitamos hacer uso de un ["EnhancedPatternLayout"](#) (en versiones anteriores de `log4j` usábamos solo un `"PatternLayout"`). Este `layout` permite definir cómo será la salida de nuestros mensajes, usando una cadena con el patrón conocida como **patrón de conversión**.

El patrón de conversión es similar al de la función `"printf"`, está compuesto de texto literal y expresiones de control de formato llamados **especificadores de conversión**. Cada especificador de conversión inicia con un símbolo de porcentaje (%) y es seguido de uno o más **modificadores de formato** y por un **caracter de conversión**. El caracter de conversión especifica el tipo de dato (por ejemplo, el `logger`, prioridad, fecha, etc.). Los modificadores de formato controlan cosas como el ancho del campo, justificación izquierda o derecha, etc. Al final del tutorial colocaré dos tablas, una con los caracteres de conversión, y otra con los modificadores de formato. Ahora veamos cómo usar este `layout`:

```
log4j.appender.console.layout = org.apache.log4j.EnhancedPatternLayout
```

Para este `layout` debemos indicar su patrón de conversión, en su propiedad `"ConversionPattern"`:

```
log4j.appender.console.layout.ConversionPattern = [%-5p] %c{2} - %m%n
```

Con la configuración anterior se producirá la siguiente salida:

```
[TRACE] configuracion.Principal - mensaje de trace
[DEBUG] configuracion.Principal - mensaje de debug
[INFO ] configuracion.Principal - mensaje de info
```

```
[WARN ] configuracion.Principal - mensaje de warn  
[ERROR] configuracion.Principal - mensaje de error  
[FATAL] configuracion.Principal - mensaje de fatal
```

Podemos notar que ahora tenemos un poco más de información en la consola (como el nombre del `logger`). También noten que agregamos unas cuantas cadenas literales solo para que el la salida sea un poco más clara (como los signos "[" y "]" alrededor del nivel de log, y el "-" antes del mensaje).

Si a la salida anterior quisiéramos agregarle, por ejemplo, la fecha; bastaría con modificar el `"ConversionPattern"` de la siguiente forma:

```
log4j.appender.console.layout.ConversionPattern = %d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n
```

Con lo que obtendríamos la siguiente salida:

```
22 abr 2011 - 10:47:47 [TRACE] configuracion.Principal - mensaje de trace  
22 abr 2011 - 10:47:47 [DEBUG] configuracion.Principal - mensaje de debug  
22 abr 2011 - 10:47:47 [INFO ] configuracion.Principal - mensaje de info  
22 abr 2011 - 10:47:47 [WARN ] configuracion.Principal - mensaje de warn  
22 abr 2011 - 10:47:47 [ERROR] configuracion.Principal - mensaje de error  
22 abr 2011 - 10:47:47 [FATAL] configuracion.Principal - mensaje de fatal
```

De la misma forma que podemos modificar el `layout` de salida, podemos elegir que se muestren solo ciertos mensajes de log de un nivel determinado hacia arriba. Por ejemplo, para mostrar solo nos mensajes de nivel `"WARN"` a `"FATAL"` debemos configurar el `logger` de la siguiente forma:

```
log4j.rootLogger=WARN, console
```

Con lo que los mensajes que obtenemos son los siguientes:

```
22 abr 2011 - 10:50:03 [WARN ] configuracion.Principal - mensaje de warn  
22 abr 2011 - 10:50:03 [ERROR] configuracion.Principal - mensaje de error  
22 abr 2011 - 10:50:03 [FATAL] configuracion.Principal - mensaje de fatal
```

Una última cosa importante que hay que saber, es que dentro del mismo `appender` podemos definir un **umbral inferior** para los mensajes que dicho `appender` mostrará. Por ejemplo, definimos que el `appender "console"` tendrá un umbral inferior (`threshold`) para los mensajes de nivel `"INFO"`, con la siguiente línea:

```
log4j.appender.console.threshold = INFO
```

No importa que definamos el nivel del `logger` como `TRACE`:

```
log4j.rootLogger = TRACE, console
```

Los únicos mensajes que se mostrarán para este `appender` son del nivel `"INFO"` hacia arriba:

```
22 abr 2011 - 10:54:46 [INFO ] configuracion.Principal - mensaje de info  
22 abr 2011 - 10:54:46 [WARN ] configuracion.Principal - mensaje de warn  
22 abr 2011 - 10:54:46 [ERROR] configuracion.Principal - mensaje de error  
22 abr 2011 - 10:54:46 [FATAL] configuracion.Principal - mensaje de fatal
```

Lamentablemente en los archivos de propiedades solo podemos definir un nivel inferior, no uno superior (en los archivos de configuración XML es posible definir ambos niveles) haciendo uso de filtros.

Hasta ahora nuestro archivo de configuración se ve de la siguiente forma:

```
log4j.appender.consola = org.apache.log4j.ConsoleAppender
log4j.appender.consola.threshold = INFO
log4j.appender.consola.target = System.out
log4j.appender.consola.layout = org.apache.log4j.EnhancedPatternLayout
log4j.appender.consola.layout.ConversionPattern = %d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n
log4j.rootLogger = TRACE, consola
```

Como pudimos ver, con solo realizar una modificación al archivo de propiedades fue posible modificar el formato de los mensajes de salida, así como el nivel de log de la aplicación.

Es posible definir múltiples **appenders** en un mismo archivo de configuración, y asociar estos **appenders** a un mismo, o a diferentes **loggers**.

Por ejemplo, para crear un **appender** que envíe los mensajes de log a un archivo, sería necesario crear un **FileAppender** como el siguiente:

```
log4j.appender.archivo = org.apache.log4j.FileAppender
log4j.appender.archivo.file = archivo.log
log4j.appender.archivo.layout = org.apache.log4j.EnhancedPatternLayout
log4j.appender.archivo.layout.ConversionPattern = %d [%-5p] %c{2} - %m%n
```

Y para asociarlo al **rootLogger**, junto con el appender "consola" que ya se tenía configurado, se hace lo siguiente:

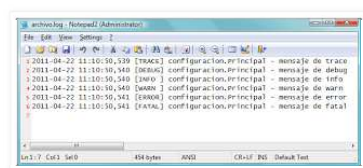
```
log4j.rootLogger = TRACE, consola, archivo
```

En este caso ambos **appenders** tienen el mismo nivel de log.

Cuando ejecutemos nuestra aplicación veremos aparecer un archivo llamado "archivo.log" en el directorio raíz de la aplicación:



Este archivo tiene el siguiente contenido:



Hay que tener cuidado de no cometer el error de tratar de que unos mensajes de log, con cierta prioridad, se envíen a un **appender**, y otros mensajes, con otra prioridad, se envíen a otro **appender**, de la siguiente forma:

```
log4j.rootLogger = TRACE, consola
log4j.rootLogger = ERROR, archivo
```

Con esto, lo único que se consigue es sobre-escribir el **appender** que estaba asociado al **rootLogger**, y que este deje de mostrar mensajes en la consola.

De hecho el objetivo anterior no es posible lograrlo con archivos de propiedades, pero sí con archivos XML.

Es posible asociar **appenders**, no solo al **rootLogger**, sino también a **loggers** particulares, haciendo uso de su nombre. El **appender** que asociemos a un **logger** será heredado por todos sus descendientes. Por ejemplo, para asociar el **appender** "archivo" al logger "com.javatutoriales.log4j.configuracion", se hace lo siguiente:

```
log4j.logger.com.javatutoriales.log4j.configuracion = WARN, archivo
```


Con lo que el archivo de configuración quedaría de la siguiente forma:

```
log4j.appender.consola = org.apache.log4j.ConsoleAppender
log4j.appender.consola.threshold = INFO
log4j.appender.consola.target = System.out
log4j.appender.consola.layout = org.apache.log4j.EnhancedPatternLayout
log4j.appender.consola.layout.ConversionPattern = %d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n

log4j.appender.archivo = org.apache.log4j.FileAppender
log4j.appender.archivo.file = archivo.log
log4j.appender.archivo.layout = org.apache.log4j.PatternLayout
log4j.appender.archivo.layout.ConversionPattern = %d [%-5p] %c{2} - %m%n

log4j.rootLogger=TRACE, consola
log4j.logger.com.javatutoriales.log4j.configuracion=WARN, archivo
```

De la misma forma, es posible definir distintos **appenders** para distintos **loggers**. Suponiendo que la aplicación tiene dos paquetes, un paquete `"com.javatutoriales.log4j.configuracion.datos"` y `"com.javatutoriales.log4j.configuracion.conexiones"`, se pueden asociar distintos **loggers**, así como colocarles distintos niveles de log, a estos paquetes de la siguiente forma:

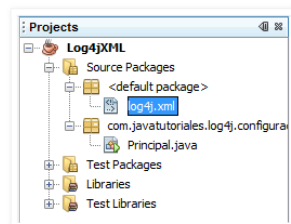
```
log4j.logger.com.javatutoriales.log4j.configuracion.datos = WARN, archivo
log4j.logger.com.javatutoriales.log4j.configuracion.conexiones = INFO, consola
```

Esto es todo lo que podemos decir sobre la configuración de **log4j** usando archivos de propiedades. Ahora veremos cómo realizar la configuración usando archivos XML.

CONFIGURACIÓN DE LOG4J CON UN ARCHIVO XML

Es posible realizar esta misma configuración en un archivo xml. De hecho, es posible incluir una configuración más detallada en este tipo de archivos.

Lo primero que debemos hacer es crear un archivo XML, llamado `"log4j.xml"`, en el paquete raíz de la aplicación. **log4j** busca por default este archivo en esta ubicación. Igual que con los archivos de configuración podríamos llamar a este archivo de otra forma, pero en ese caso tendríamos que realizar un proceso de configuración dentro de nuestra aplicación.



El encabezado del archivo comienza con la declaración estándar de XML seguida de una declaración de un **DOCTYPE**, la cual indica que el **DTD** (Document Type Definition) se encuentra en el mismo sistema:

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

Posteriormente debe incluirse el elemento raíz del archivo, el cual es el elemento `"log4j:configuration"`, dentro de este se indica que se usará el namespace `"log4j"` para definir los elementos de configuración de **log4j**:

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

Dentro de estas etiquetas se hacen las declaraciones de los elementos que conformar la configuración de **log4j** (**loggers**, **appenders**, y **layouts**).

Los **appenders** son agregados usando el elemento `"<appender>"`, al cual se le deben especificar un nombre, usando el atributo `"name"`, y un tipo, usando el atributo `"class"`. En este caso se creará un **appender** llamado `"consola"` que será de tipo `"org.apache.log4j.ConsoleAppender"`:

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

Podemos establecer el valor de las propiedades de los **appenders**, y en general de cualquier elemento, usando la etiqueta "**<param>**". El nombre del parámetro que se establecerá se indica usando el atributo "**name**" y el valor usando el atributo "**value**". En este caso estableceremos el valor de su atributo "**target**":

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />
</appender>
```

Como un **appender** debe tener asociado un **layout**, esto se hace anidando un elemento "**<layout>**" dentro del elemento "**<appender>**" que se acaba de definir. En este caso se usará nuevamente un "**EnhancedPatternLayout**", lo cual se define usando el elemento "**class**". A este **layout** se le pueden definir propiedades, como en este caso su "**ConversionPattern**", usando el elemento "**<param>**":

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />
  <layout class="org.apache.log4j.EnhancedPatternLayout">
    <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
  </layout>
</appender>
```

Para asociar un **appender** al **rootLogger**, se usa el elemento "**<root>**". Dentro de este se establece el nivel de log usando la etiqueta "**<priority>**", con su atributo "**value**", y se indican los **appenders** que tendrá asociado usando el elemento "**<appender-ref>**", cuyo atributo "**ref**" debe indicar el nombre del **appender** que se asociará con este **logger**:

```
<root>
  <priority value="trace" />
  <appender-ref ref="consola"/>
</root>
```

Al final el archivo de configuración queda de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="consola" class="org.apache.log4j.ConsoleAppender">
    <param name="target" value="System.out" />
    <layout class="org.apache.log4j.EnhancedPatternLayout">
      <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
    </layout>
  </appender>

  <root>
    <priority value="trace" />
    <appender-ref ref="consola"/>
  </root>

</log4j:configuration>
```

Al ejecutar la aplicación se obtiene la siguiente salida en la consola:

```
22 abr 2011 - 11:37:50 [TRACE] configuracion.Principal - mensaje de trace
22 abr 2011 - 11:37:50 [DEBUG] configuracion.Principal - mensaje de debug
22 abr 2011 - 11:37:50 [INFO ] configuracion.Principal - mensaje de info
22 abr 2011 - 11:37:50 [WARN ] configuracion.Principal - mensaje de warn
22 abr 2011 - 11:37:50 [ERROR] configuracion.Principal - mensaje de error
22 abr 2011 - 11:37:50 [FATAL] configuracion.Principal - mensaje de fatal
```

Que como puede apreciarse es igual a la obtenida usando el archivo de propiedades.

Igual que con los archivos de propiedades, es posible agregar más de un **appender** por archivo de configuración, agregando más elementos "**<appender>**". También es posible asociar **appenders** a **loggers** particulares, declarando el **logger** usando el elemento "**<logger>**" y definiendo su nombre con el atributo "**name**". El elemento **logger**, es igual que "**root**", por lo que también es posible definir su nivel de log (solo que en este caso usando el elemento "**<level>**") y los **appenders** que tiene asociados:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="consola" class="org.apache.log4j.ConsoleAppender">
```

```

        <layout class="org.apache.log4j.EnhancedPatternLayout">
            <param name="ConversionPattern" value="%d{ABSOLUTE} [%t] %-5p %c %x - %m%n" />
        </layout>
    </appender>

    <appender name="archivo" class="org.apache.log4j.FileAppender">
        <param name="file" value="archivo.log" />
        <layout class="org.apache.log4j.EnhancedPatternLayout">
            <param name="ConversionPattern" value="%d{ABSOLUTE} [%t] %-5p %c %x - %m%n" />
        </layout>
    </appender>

    <logger name="log4j.xml">
        <level value="warn" />
        <appender-ref ref="archivo" />
    </logger>

    <root>
        <priority value="trace" />
        <appender-ref ref="consola"/>
    </root>

</log4j:configuration>

```

También, si queremos agregar más de un **appender** a un **logger** particular, solo basta con agregar otro elemento **<appender-ref>** para indicar el otro **appender**:

```

<logger name="log4j.xml">
    <level value="warn" />
    <appender-ref ref="archivo" />
    <appender-ref ref="consola" />
</logger>

```

Nuevamente, igual que con los archivos de propiedades, es posible definir un umbral mínimo para los mensajes de log, estableciendo el nivel en la propiedad **threshold** del **appender** correspondiente:

```

<appender name="consola" class="org.apache.log4j.ConsoleAppender">
    <param name="threshold" value="INFO" />
    <layout class="org.apache.log4j.EnhancedPatternLayout">
        <param name="ConversionPattern" value="%d{ABSOLUTE} [%t] %-5p %c %x - %m%n" />
    </layout>
</appender>

```

Sin embargo, el los archivos **XML** podemos usar un mecanismo aún más poderoso para limitar los mensajes que se muestran en un **appender**, y esto se hace con el uso de filtros.

Filtros de Mensajes

Una cosa muy interesante que puede hacerse en el archivo de configuración en **XML** es el uso de filtros. Estos filtros permiten indicar que el **appender** solo mostrará, o no, los mensajes de cierto nivel. Los filtros se colocan en forma de cadena dentro del **appender**, y la decisión de si un mensaje debe mostrarse o no se hace tomando en cuenta las decisiones de **todos los filtros**.

log4j viene únicamente con 4 filtros, pero encadenándolos tenemos lo necesario para mostrar cualquier nivel de mensaje que se les pueda ocurrir:

- **"DenyAllFilter"**: Evita que se produzcan mensajes de log.
- **"LevelMatchFilter"**: Muestra solo los mensajes que tengan el nivel indicado.
- **"LevelRangeFilter"**: Muestra únicamente los mensajes que se encuentran entre un nivel inferior y superior de log.
- **"StringMatchFilter"**: Muestra solo los mensajes que en el mensaje de salida tengan la sub-cadena indicada.

Los filtros son colocados dentro de los **appenders** usando el elemento **<filter>**.

Comencemos con el primero y el más sencillo: **"DenyAllFilter"**. Si colocamos este filtro en nuestro **appender** de la siguiente forma:

```

<appender name="consola" class="org.apache.log4j.ConsoleAppender">
    <param name="target" value="System.out" />
    <layout class="org.apache.log4j.EnhancedPatternLayout">
        <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
    </layout>

    <filter class="org.apache.log4j.varia.DenyAllFilter" />

</appender>

```

Evitará que se muestren mensajes en la salida. ¿Para qué puede servirnos esto? Esto es útil en dos situaciones. La primera es cuando queremos que un **appender** deje de enviar mensajes.

La segunda, y creo que la más importante, es cuando lo colocamos **al final de una cadena de filtros**. En ese caso cambia el comportamiento de **"acepta todo a menos que se diga lo contrario"** a **"niega todo a menos que se diga lo contrario"**. Esto quedará más claro cuando veamos los siguientes ejemplos.

"LevelRangeFilter" permite que se muestren solo los mensajes del nivel indicado. Este filtro tiene dos propiedades **"LevelToMatch"**,

que indica el nivel que aceptará el filtro, y "AcceptOnMatch", el cual recibe un valor booleano que indica si el mensaje debe mostrarse o no.

Si queremos que nuestro appender **NO** muestre los mensajes de nivel "WARN", lo configuramos de esta forma:

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />

  <layout class="org.apache.log4j.EnhancedPatternLayout">
    <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
  </layout>

  <filter class="org.apache.log4j.varia.LevelMatchFilter">
    <param name="LevelToMatch" value="warn" />
    <param name="AcceptOnMatch" value="false" />
  </filter>
</appender>
```

En donde podemos ver que el valor de "AcceptOnMatch" es falso. Si queremos que **solo** se muestren los mensaje de nivel "WARN" debemos usar este filtro en combinación con el "DenyAllFilter":

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />
  <layout class="org.apache.log4j.EnhancedPatternLayout">
    <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
  </layout>

  <filter class="org.apache.log4j.varia.LevelMatchFilter">
    <param name="LevelToMatch" value="warn" />
    <param name="AcceptOnMatch" value="true" />
  </filter>
  <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

En este caso es necesario hacer un encadenamiento de los filtros. "LevelMatchFilter" indica que debe aceptar los mensajes de nivel "WARN", y "DenyAllFilter" dice que niegue todo lo que no esté aceptado de forma explícita. En este caso todos los demás niveles. Para determinar qué mensajes son mostrados y cuáles no, en una cadena de filtros, se usa un sistema del que hablaré un poco más adelante.

"LevelRangeFilter" permite aceptar mensajes de log que se encuentran solo dentro de **ciertos rangos establecidos**. Este filtro tiene tres propiedades: "LevelMin", "LevelMax" y "AcceptOnMatch".

Si queremos mostrar solo los mensajes que se encuentran entre los niveles "INFO" y "ERROR", podemos hacerlo de la siguiente forma:

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />
  <layout class="org.apache.log4j.EnhancedPatternLayout">
    <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
  </layout>

  <filter class="org.apache.log4j.varia.LevelRangeFilter">
    <param name="LevelMin" value="info" />
    <param name="LevelMax" value="error" />
    <param name="AcceptOnMatch" value="true" />
  </filter>
</appender>
```

Con este filtro no hay una forma directa para hacer que se acepten todos los niveles, excepto los que se indican en el, para esto es necesario hacer varias combinaciones de filtros.

Finalmente, el filtro "StringMatchFilter" permite que se muestren o se rechacen solo los mensajes de log que contengan una sub-cadena dada. Este filtro tiene dos tributos: "StringToMatch" que indica la subcadena que será buscada, y "AcceptOnMatch" que indica si el mensaje será aceptado o rechazado.

Por ejemplo, si queremos que **NO** se muestren los mensajes que contengan la cadena "info", debemos configurar el filtro de esta forma:

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />
  <layout class="org.apache.log4j.EnhancedPatternLayout">
    <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
  </layout>

  <filter class="org.apache.log4j.varia.StringMatchFilter">
    <param name="StringToMatch" value="info" />
    <param name="AcceptOnMatch" value="false" />
  </filter>
</appender>
```

Si queremos que **solo** se muestren los mensajes que contengan la cadena "info", debemos usarlo junto con el filtro

"DenyAllFilter":

```
<appender name="consola" class="org.apache.log4j.ConsoleAppender">
  <param name="target" value="System.out" />
  <layout class="org.apache.log4j.EnhancedPatternLayout">
    <param name="ConversionPattern" value="%d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n" />
  </layout>

  <filter class="org.apache.log4j.varia.StringMatchFilter">
    <param name="StringToMatch" value="info" />
    <param name="AcceptOnMatch" value="true" />
  </filter>
  <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

Ahora hablemos sobre el sistema aplicación de los filtros cuando se encuentran en cadenas. Como había dicho, cuando tenemos una cadena de filtros estos filtran los mensajes para decidir si debe mostrarse o no. Los filtros solo pueden aceptar, rechazar, o no tomar decisión sobre un mensaje (ser neutrales). Una vez que un filtro ha aceptado un mensaje, **este no puede ser rechazado posteriormente por otro filtro**.

Para determinar si un filtro aceptará o rechazará un mensaje dependerá del valor que coloquemos en su atributo "AcceptOnMatch" y de si se encuentra o no una coincidencia para el nivel de mensaje (o para el mensaje en el caso del "StringMatchFilter").

Por ejemplo, con un "LevelMatchFilter", si el mensaje no coincide con el nivel especificado, el filtro no acepta ni rechaza el mensaje (es neutral). Si encuentra una coincidencia, y "AcceptOnMatch" es verdadero entonces el filtro **aceptará el mensaje**, pero si "AcceptOnMatch" es falso, el filtro **rechazará el mensaje**.

La siguiente tabla resume el comportamiento de cada uno de los filtros:

Filtro	AcceptOnMatch Verdadero	AcceptOnMarth Falso	Sin Coincidencias
LevelMatchFilter	Aceptado	Rechazado	Neutral
LevelRangeFilter	Aceptado	Neutral	Rechazado
StringMatchFilter	Aceptado	Rechazado	Neutral

Si un mensaje no es rechazado de forma explícita este será mostrado, a menos que se aplique un "DenyAllFilter" al final de la cadena. En este caso todo lo que no sea aceptado de forma explícita será rechazado.

Veamos unos ejemplos para que esto quede más claro. Si declaramos el siguiente filtro de tipo "LevelRangeFilter":

```
<filter class="org.apache.log4j.varia.LevelRangeFilter">
  <param name="LevelMin" value="info" />
  <param name="LevelMax" value="error" />
  <param name="AcceptOnMatch" value="false" />
</filter>
```

Como "AcceptOnMatch" está establecido en "false" los mensajes que coincidan con el rango ("INFO", "WARN" y "ERROR") serán tomados de forma neutral, y los que no coincidan serán rechazados. Por lo que solo veremos los mensajes de nivel "INFO", "WARN", y "ERROR".

Pero si colocamos el filtro de la siguiente forma:

```
<filter class="org.apache.log4j.varia.LevelRangeFilter">
  <param name="LevelMin" value="info" />
  <param name="LevelMax" value="error" />
  <param name="AcceptOnMatch" value="false" />
</filter>
<filter class="org.apache.log4j.varia.DenyAllFilter" />
```

Como ahora estamos agregando un "DenyAllFilter" al final de la cadena, todos los mensajes que no sean aceptados explícitamente serán rechazados. En este caso, como ningún mensaje es aceptado de forma explícita, todos serán rechazados y no veremos ningún mensaje.

Si ahora colocamos los filtros, de esta forma:

```
<filter class="org.apache.log4j.varia.LevelMatchFilter">
  <param name="LevelToMatch" value="warn" />
  <param name="AcceptOnMatch" value="false" />
</filter>
<filter class="org.apache.log4j.varia.LevelRangeFilter">
  <param name="LevelMin" value="info" />
  <param name="LevelMax" value="error" />
  <param name="AcceptOnMatch" value="true" />
</filter>
```

El primer filtro, de tipo "LevelMatchFilter" está configurado para un nivel de "WARN" y con su valor de "AcceptOnMatch" en false. Esto quiere decir que rechazará todos los mensajes de tipo "WARN".

El siguiente filtro, de tipo `LevelRangeFilter`, está configurado para aceptar mensajes de tipo `"INFO"` a `"ERROR"`, y con su valor `"AcceptOnMatch"` en `true`, por lo que aceptará mensajes de tipo `"INFO"`, `"WARN"`, y `"ERROR"`, y rechazará los demás.

Cuando un mensaje entra al primer filtro, este rechaza los mensajes de tipo `"WARN"` y dejará pasar los demás. En segundo filtro solo dejará pasar los mensajes de tipo `"INFO"`, y `"ERROR"`. Por lo que la única salida que obtendremos serán los mensajes de nivel `"INFO"` y `"ERROR"`.

Noten que si invertimos el orden de los filtros el resultado que obtendremos es que se mostrarán los mensajes de tipo `"INFO"`, `"WARN"`, y `"ERROR"`. Esto es porque el primer filtro ya ha aceptado los mensajes de tipo `"WARN"` de forma explícita, y entonces el rechazo del segundo filtro ya no es tomado en cuenta.

Estos son todos los ejemplos que se me ocurren para mostrar el uso de los filtros. Si alguien quiere que hagamos un ejemplo particular, colóquelo en los comentarios y lo agregaré a esta sección `^_^`.

Para terminar este tutorial, veamos las tablas que explican los caracteres de conversión y los modificadores de formato para `"EnhancedPatternLayout"`.

Caracteres de Conversión de Formato de EnhancedPatternLayout

Caracter de Conversión	Efecto
c	<p>Muestra el nombre del <code>logger</code> que generó el mensaje. Se puede especificar un patrón de <code>"NameAbbreviator"</code> dentro de llaves para limitar y modificar la información mostrada.</p> <p>Por ejemplo, para el <code>logger "com.javatutoriales.log4j.configuracion.Principal"</code>, si usamos el patrón <code>%c{2}</code> solo se mostrarán los dos elementos finales del nombre del <code>logger</code>, o sea <code>"configuracion.Principal"</code>. Con <code>%c{-2}</code> se eliminan los dos primeros elementos dejando solo <code>"log4j.configuracion.Principal"</code>. Con <code>%c{1.}</code> solo se mostrará el primer carácter de los elementos no finales del <code>logger</code>, o sea <code>"c.j.l.c.Principal"</code>; con <code>%c{2.}</code> obtendremos <code>"co.ja.lo.co.Principal"</code>. Si hacemos <code>%c{2-.1:}</code> se mostrarán los dos primeros caracteres del primer elemento usando una tilde para indicar los caracteres abreviados y el primer caracter de los siguientes elementos no finales usando dos puntos para indicar los caracteres abreviados, o sea: <code>"co-.j:.l:.c:.Principal"</code></p>
C	<p>Se usa para mostrar el nombre de la clase donde se generó el mensaje de log. En este caso también se especifica un patrón de <code>"NameAbbreviator"</code> que sigue las mismas reglas indicadas anteriormente.</p> <p>ADVERTENCIA: Generar esta información es lento, por lo que debe evitarse el uso a menos que sea estrictamente necesario.</p>
d	<p>Se usa para mostrar la fecha del evento de log. Se puede especificar un patrón para la fecha y hora usando los patrones de <code>"SimpleDateFormat"</code>, <code>ABSOLUTE</code>, <code>DATE</code>, o <code>ISO8601</code>, por ejemplo: <code>%d{HH:mm:ss,SSS}</code>, <code>%d{dd MMM yyyy HH:mm:ss,SSS}</code>, <code>%d{DATE}</code> o <code>%d{HH:mm:ss}{GMT+0}</code></p>
F	<p>Se usa para mostrar el nombre del archivo donde se generó el mensaje de log.</p> <p>ADVERTENCIA: Generar esta información es lento, por lo que debe evitarse el uso a menos que sea estrictamente necesario.</p>
l	<p>Se usa para mostrar la información de la ubicación de quien hizo la petición de evento de log.</p> <p>La información de la ubicación depende de la implementación de la JVM, pero usualmente consiste en el fully qualified name del método, seguido del nombre del archivo, y el número de línea en el que se encuentra la llamada.</p> <p>ADVERTENCIA: Generar esta información es extremadamente lento, por lo que debe evitarse el uso a menos que sea estrictamente necesario.</p>
L	<p>Se usa para mostrar el número de línea dónde se hizo la petición para el evento de log.</p> <p>ADVERTENCIA: Generar esta información es lento, por lo que debe evitarse el uso a menos que sea estrictamente necesario.</p>
m	<p>Se usa para mostrar el mensaje de log que es proporcionado por la aplicación al invocar el método del evento de log.</p>
M	<p>Se usa para mostrar el nombre del método donde se hizo la petición para el evento de log.</p> <p>ADVERTENCIA: Generar esta información es lento, por lo que debe evitarse el uso a menos que sea estrictamente necesario.</p>
n	<p>Sirve para colocar un caracter de salto de línea.</p>
p	<p>Se usa para mostrar el nivel de log del evento.</p>
r	<p>Se usa para mostrar el número de milisegundos transcurridos desde la construcción del layout, hasta la creación del evento de log.</p>
t	<p>Se usa para mostrar el nombre del thread que genera el evento de log.</p>
x	<p>Se usa para mostrar el <code>NDC</code> (Nested Diagnostic Context) asociado con el thread que genera el evento de log. Un <code>NDC</code> es un instrumento para distinguir salidas de log intercaladas que provienen de distintas fuentes. Las salidas de log suelen ser intercaladas cuando un servidor maneja varios clientes al mismo tiempo.</p>
X	<p>Se usa para mostrar el <code>MDC</code> (Mapped Diagnostic Context) asociado con el thread que genera el evento de log. El <code>MDC</code> es lo mismo que el <code>NDC</code> solo que usando un Map como implementación.</p>
properties	<p>Se usa para mostrar las propiedades asociadas con el evento de log. Adicionalmente podemos indicar la llave de la propiedad que queremos obtener, como por ejemplo <code>%properties{aplicacion}</code></p>
throwable	<p>Se usa para mostrar el stack trace asociado con el evento de log, por default se muestra el stack trace completo. Podemos controlar la profundidad del stack mostrado colocando el mismo entre llaves. Por ejemplo <code>%throwable{short}</code> o <code>%throwable{1}</code> mostrarán la primer línea del stack trace. <code>throwable{none}</code> o <code>throwable{0}</code> no mostrarán nada del stack trace. <code>%throwable{n}</code> mostrará las primeras <code>n</code> líneas del stack trace, si es un número positivo; si es un número negativo mostrará las últimas <code>n</code> líneas del stack trace.</p>
%	<p>La secuencia <code>%%</code> muestra un solo signo de porcentaje <code>^_^</code></p>

Modificadores de Formato de EnhancedPatternLayout

Los modificadores de formatos nos permiten controlar el número de caracteres que se reservarán para un dato de salida, así como la justificación del texto dentro de estos caracteres. También permite indicar la longitud máxima de un dato.

En la siguiente tabla se muestran ejemplos de uso de estos modificadores, usando el carácter de conversión de nombre del `logger` (`%c`):

Modificador de Formato	Justificación Izquierda	Longitud mínima	Longitud máxima	Funcionamiento
<code>%20c</code>	No	20	ninguna	Justifica a la derecha dejando espacios en blanco si el nombre del logger tiene menos de 20 caracteres
<code>%-20c</code>	Si	20	ninguna	Justifica a la izquierda dejando espacios en blanco si el nombre del logger tiene menos de 20 caracteres
<code>%.30c</code>	NA	ninguna	30	Trunca a 30 caracteres, de derecha a izquierda, si el nombre del logger tiene más de 30 caracteres (o sea <code>".log4j.configuracion.Principal"</code>)
<code>%20.30c</code>	No	20	30	Justifica a la derecha dejando espacios en blanco si el nombre del logger tiene menos de 20 caracteres. Pero si tiene más de 30 caracteres, lo trunca, de derecha a izquierda.
<code>%-20.30c</code>	Si	30	30	Justifica a la izquierda dejando espacios en blanco si el nombre del logger tiene menos de 20 caracteres. Pero si tiene más de 30 caracteres lo trunca, de derecha a izquierda.

Bueno, esto es todo para este tutorial de `log4j`. Como podemos ver, es un framework muy completo que nos permite controlar prácticamente cada aspecto de la bitácora o logging de nuestra aplicación. De una forma muy completa y sencilla, solo modificando un archivo de configuración.

Espero que les sea de utilidad. No olviden dejar sus dudas, comentarios y sugerencias.

Saludos y gracias.

Descarga los archivos de este tutorial desde aquí:

- [Log4j con Archivo de Propiedades](#)
- [Log4j con Archivo XML](#)


Publicado por Alex en 8:14

Reacciones:	divertido (1)	interesante (1)	increible (3)	no me gusta (0)
-------------	---------------	-----------------	---------------	-----------------

 +15 Recomendar esto en Google


Etiquetas: [bitacoras](#), [configuracion](#), [java](#), [log4j](#), [propiedades](#), [xml](#)

18 comentarios:

- 


gnu.cubo 2 de mayo de 2011, 10:10

super bueno el tutorial.
Mis sinceros agradecimientos. felicidades por el blog super util. me interesan las entradas de Hibernate. a ver como nos va.
Saludos

[Responder](#)
- 


Agustín 9 de mayo de 2011, 15:25

gracias fue el unico tuto completo y sencillo que encuentre

[Responder](#)
- 

Javier 20 de mayo de 2011, 4:16

Hola quería preguntarte que configuracion y como hay que hacer esa configuracion de la que hablas en tu blog, en el caso de que quiera cambiar de nombre el archivo de propiedades. Llamarlo de otra forma en vez de log4j.properties o log4j.xml

[Responder](#)
- 

Alex 20 de mayo de 2011, 7:35

@Javier
Hola Javier;

En ese caso, tendrías que leer el archivo de configuración desde tu aplicación, en un punto en el que esta esté arrancando. Si quieres leerla desde un archivo de propiedades es necesario crear una instancia de la clase [PropertyConfigurator](#) y pasarle la ruta del archivo don de tienes su configuración, en su método configure:


```
new PropertyConfigurator().configure("configuracion.properties");
```

Para los archivos XML se debe usar la clase [DOMConfigurator](#) e igualmente usar su método configure:

```
new DOMConfigurator().configure("archivo.xml");
```

Espero que esto te ayude con lo que quieres hacer :D

Saludos

[Responder](#)
- 

Héctor 4 de noviembre de 2011, 1:03

Magnifico tutorial. Muy completo, desconocía el tema de los NDC y los MDC. Lo he utilizado para un post que he realizado sobre SLF4J Migrator (<http://tuenteperez.es/blog/slf4j-migrator-migra-tus-proyectos-de-log4j-a-slf4j-especial-impacientes>) en el cual te hago referencia.

Muchas Gracias.

Salu2. Héctor.

[Responder](#)**carlos** 18 de noviembre de 2011, 0:06

Hola Alex!!

Como puedo habilitar la salida a un archivo los mensaje pero utilizando slf4j???

[Responder](#)**Juanop** 5 de diciembre de 2011, 7:21

Increible el tutorial!! muchas gracias!

[Responder](#)**GMejia** 7 de enero de 2012, 6:51

Un articulo muy completo, gracias !

[Responder](#)**Athanatos** 13 de enero de 2012, 8:22

muy muy muy completo tutorial, muchas gracias por el aporte

[Responder](#)**shianim** 21 de marzo de 2012, 6:10

Hola a todos, quisiera saber si existe la posibilidad de generar un archivo de log de la siguiente forma:

`archivo_fecha_creacion.log`

Donde fecha_creacion, es el día y la hora en que se genero el archivo.

Es pero me puedan ayudar, muchas gracias de antemano.

[Responder](#)**Julio Gonzalez Rios** 6 de marzo de 2013, 17:45

Muy buen tuto, el mejor que pude encontrar :D

[Responder](#)**Diego** 22 de abril de 2013, 10:43Muchas gracias por el tutorial . De lo mejor que encontré sobre log4j .
Saludos desde Perú[Responder](#)**Hablemos del Balonpie** 26 de noviembre de 2014, 9:31

Buen Día,

Muy buen tutorial completo, como hago para guardar los logs de mi aplicación JSP de la forma .properties que explicabas anteriormente, es decir que guarde todos los logs o mensajes de error de la consola de mi aplicación

Mil Gracias

[Responder](#)**Andrea Rojas** 13 de marzo de 2015, 8:52

Excelente! Gracias por tu aporte. Saludos!

[Responder](#)**Ljudevit Gaj** 16 de mayo de 2015, 23:08

Muchas gracias por compartir tu conocimiento, es oro lo que compartes. Trabajo con Eclipse Indigo y el único problema que tuve fue que no leía los archivos log4j.properties desde el paquete, lo tuve que cortar y pegar en la carpeta bin. Muchas gracias.

[Responder](#)**jose alberto** 25 de octubre de 2015, 3:01Hola solo una pregunta, porque cuando declaro mi logger asi: log4j.rootLogger=ALL, CONSOLA
al momento de correr mi programa me muestra muchos mensajes de trace y debug pero no tengo logguers en mi codigo mas que doa de error.[Responder](#)**ALEJANDRA MORENO HERNANDEZ** 7 de julio de 2016, 13:58

Hola! Qué medidas de toman con esta librería si el archivo del log crece?

Saludos.

[Responder](#)**Parov Hefe** 11 de julio de 2016, 6:46Hola! Si estas interesado en traducir archivos .Properties, yo recomiendo la herramienta en línea para la localización de software [POEditor](#).[Responder](#)

Introduce tu comentario...

Comentar como:

Seleccionar perfil..

Publicar

Vista previa

Enlaces a esta entrada

- [Crear un enlace](#)
- [Entrada más reciente](#)
- [Página principal](#)
- [Entrada antigua](#)
- Suscribirse a: [Enviar comentarios \(Atom\)](#)