Log4J

Mr. Ashok

Ashok IT School

Facebook group Name: Ashok IT School

Email: ashok.javatraining@gmail.com

While developing Java/J2EE applications, as a programmer we need to understand flow of the applications. That is to know the status of a java application at its execution time in general we will use System.out.println () statements in the application.

But we have some disadvantages with SOPL (System.out.println) statements.

- Generally SOPL statements are printed on console, so they are temporary messages and whenever the console is closed then automatically the messages are removed from the console
- It is not possible to store the SOPL messages in a permanent place and these are single threaded model, means these will prints only one by one message on the console screen
- SOPL statements will decrease performance of the application

In order to overcome the problems of SOPL statements **Log4j** came into picture, with **Log4j** we can store the flow details of our Java/J2EE projects in separate persistency medium like file or databases.

Log data is a definitive record of what's happening in every business, organization or agency and it's often an untapped resource when it comes to troubleshooting and supporting broader business objectives.

In real time, after a project is released and it is installed in a client location then we call the location as on-site, when executing the program at on-site location, if we get any problems occurred then these problems will be reported to the off showered engineers, in this time we used to mail these Log files, so that developers can check the problems easily.

Overview of log4J

Log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License.

Log4J available in 2 versions, they are

Log4j 1.x

Log4J 2.x

Log4j Features

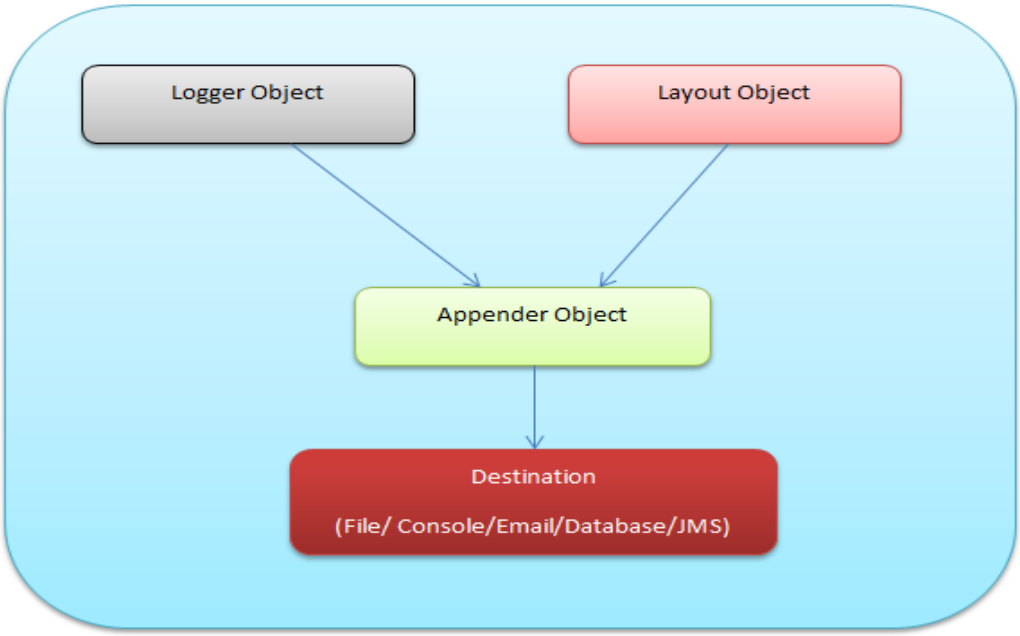
- It is thread-safe.
- It supports multiple output appenders per logger.
- It supports internationalization.
- It is not restricted to a predefined set of facilities.
- Logging behavior can be set at runtime using a configuration file.
- It is designed to handle Java Exceptions from the start.
- The 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.
- Log4j is used for large as well as small projects

 ashok.javatraining@gmail.com

Main Components of log4J

- Logger
- Appender
- Layout

Log4j Architecture



Logger

It is responsible for logging information.

To implement loggers into a project following steps need to be performed

Create an instance for logger class: Logger class is a Java-based utility that has got all the generic methods already implemented to use log4j

Define the Log4j level: Primarily there are five kinds of log levels

- **All** - This level of logging will log everything (it turns all the logs on)
- **DEBUG** – print the debugging information and is helpful in development stage
- **INFO** – print informational message that highlights the progress of the application
- **WARN** – print information regarding faulty and unexpected system behavior.
- **ERROR** – print error message that might allow system to continue
- **FATAL** – print system critical information which are causing the application to crash(Can't continue further)
- **OFF** – No logging

 ashok.javatraining@gmail.com

Syntax to get Logger object

```
static Logger log = Logger.getLogger(YourClassName.class.getName())
```

Note: while creating a Logger object we need to pass either fully qualified class name or class object as a parameter, class means current class for which we are going to use Log4j like below

```
public class StockMarket {

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

	public double getStockPrice(String companyName) throws Exception {

		double price = 0.0;

		if (companyName.equalsIgnoreCase("Amazon")) {
			price = 540.00;
		} else if (companyName.equalsIgnoreCase("Wellsforgo")) {
			price = 450.00;
		} else if (companyName.equalsIgnoreCase("JPMorgan")) {
			price = 480.00;
		} else {
			throw new Exception("");
		}
		return price;
	}
}
```

StockMarket.java

Logger object has some methods, we will use these methods to print the status of our application behavior.

We have totally 6 methods in Logger class

1. trace (Object message): It is used to print the message with the level **Level.TRACE**.

   Syntax: public void trace (Object message)

2. debug (Object message): It is used to print the message with the level **Level.DEBUG**.

   Syntax: public void debug (Object message)

3. info (Object message): It is used to print the message with the level **Level.INFO**.

   Syntax: public void info (Object message)

4. warn (Object message): It is used to print the message with the level **Level.WARN**.

   Syntax: public void warn (Object message)

5. error (Object message): It is used to print the message with the level **Level.ERROR**.

   Syntax: public void error (Object message)

6. fatal (Object message): It is used to print the message with the level **Level.FATAL**.

   Syntax: public void fatal (Object message)

 ashok.javatraining@gmail.com

As a programmer it is our responsibility to know where we need to use what method. Method names are different, but all methods are used for same purpose(to print log messages).

Priority Order

ALL < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < OFF

When a logger is created, generally we will assign a level. The logger outputs all those messages equal to that level and **also all greater levels** than it.

Order for the standard log4j levels are

Log4J Levels	*TRACE Level*	*DEBUG Level*	*INFO Level*	*WARN Level*	*ERROR Level*	*FATAL Level*
TRACE Level	Y	Y	Y	Y	Y	Y
DEBUG Level	N	Y	Y	Y	Y	Y
INFO Level	N	N	Y	Y	Y	Y
WARN Level	N	N	N	Y	Y	Y
ERROR Level	N	N	N	N	Y	Y
FATAL Level	N	N	N	N	N	Y
ALL Level	Y	Y	Y	Y	Y	Y
OFF Level	N	N	N	N	N	N

Appender

It is used to deliver LogEvents to their destination. It decides what will happen with log information. In simple words, it is used to write the logs in file. Following are few types of Appenders

Note: At a time we can use multiple Appenders also

In log4j we have different Appender implementation classes

- FileAppender [writing into a file]
- ConsoleAppender [Writing into console]
- JDBCAppender [For Databases]
- SMTPAppender [Mails]
- SocketAppender [For remote storage]
- SocketHubAppender
- SyslogAppendersends
- TelnetAppender

 ashok.javatraining@gmail.com

Again in FileAppender we have 2 more

- RollingFileAppender
- DailyRollingFileAppender

Layout

It is responsible for formatting logging information in different styles.

We have different types of layout classes in log4j.

SimpleLayout: SimpleLayout consists of the level of the log statement, followed by" – " and then the log message itself.
PatternLayout: A flexible layout configurable with pattern string. This code is known to have synchronization and other issues which are not present in org.apache.log4j.EnhancedPatternLayout.

The goal of this class is to format a LoggingEvent and return the results as a String. The results depend on the conversion pattern.

HTMLLayout: This layout outputs events in a HTML table. Appenders using this layout should have their encoding set to UTF-8 or UTF-16, otherwise events containing non ASCII characters could result in corrupted log files.

XMLLayout : The output of the XMLLayout consists of a series of log4j:event elements as defined in the log4j.dtd. It does not output a complete well-formed XML file. The output is designed to be included as an external entity in a separate file to form a correct XML file.

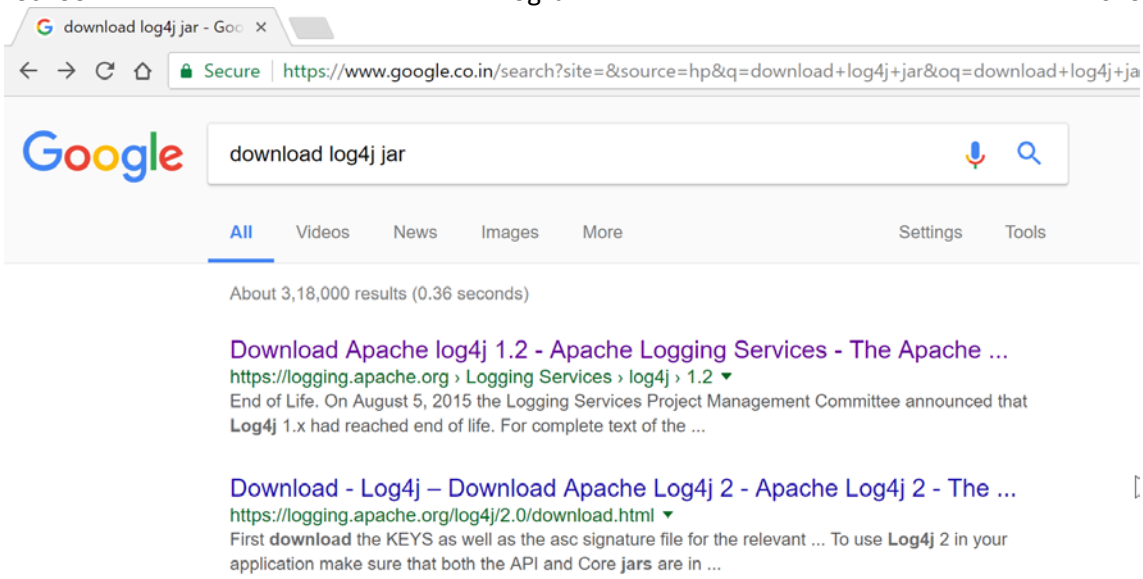
Steps to plug log4j into project

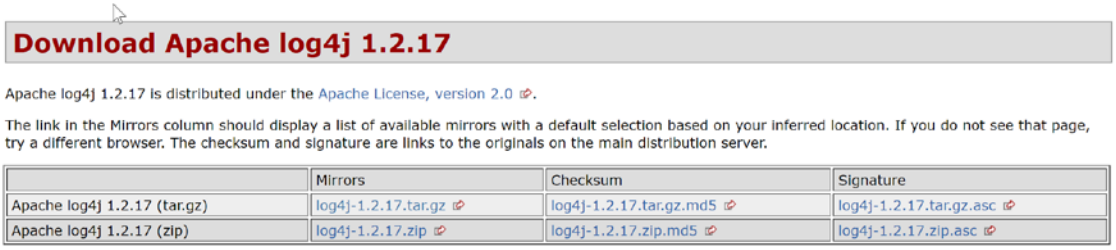
Typically, the steps to use log4j in your Java application are as follows:

- Download latest log4j distribution.
- Add log4j's jar library into your program's class path.
- Create log4j's configuration.
- Initialize log4j with the configuration.
- Create a logger.
- Put logging statements into your code.

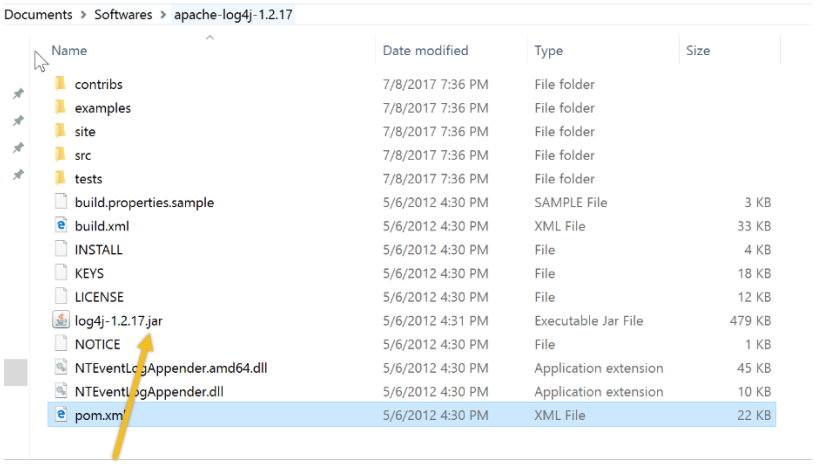
Steps to download log4j framework

 ashok.javatraining@gmail.com





Download above zip file and extract it, we can see below folder structure



To use Log4J in project we need to set the class path for above "log4j-1.2.17.jar" file.

If you are using Maven based project, add below dependency details in pom.xml

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```


 ashok.javatraining@gmail.com

First Program using Log4j: BasicConfiguration

```
package com.aits.apps;

import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Logger;

public class StockMarket {

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

	public double getStockPrice(String companyName) throws Exception {
		BasicConfigurator.configure();
		logger.info("This is info message");
		logger.debug("This is debug message");
		return 1500.00;
	}
}
```

StockMarket.java

Once we run the above program, it will print log messages on the console.

In above program we have used BasicConfiguration, it will print log messages on the console. But in future if we want to store the message in file then again we must open this java file then modifications and recompile bla bla…, so to avoid this problem log4j providing different configuration mechanisms.

Log4J Configurations

- Basic Configuration
- Using properties file
- Using xml file
- Programmatic Configuration

Properties file Configuration

To configures log4j from an external .properties file, invoke the static method configure() of the class PropertyConfigurator:

PropertyConfigurator.configure (String configFilename)

```
log4j.rootLogger=DEBUG, consoleAppender
log4j.appender.consoleAppender=org.apache.log4j.ConsoleAppender
log4j.appender.consoleAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.consoleAppender.layout.ConversionPattern=[%t] %-5p %c %x - %m%n
```


 ashok.javatraining@gmail.com

```
public class PropertiesFileLog4jExample {

	static Logger logger = Logger.getLogger(PropertiesFileLog4jExample.class);

	public static void main(String[] args) {
		String log4jConfigFile = System.getProperty("user.dir")
				+ File.separator + "log4j.properties";
		PropertyConfigurator.configure(log4jConfigFile);
		logger.debug("this is a debug log message");
		logger.info("this is a information log message");
		logger.warn("this is a warning log message");
	}
}
```

PropertiesFileLog4jExample.java

XML file Configuration example

It's also possible to configure log4j with an XML file, by invoking the static method configure() of the class DOMConfigurator:

DOMConfigurator.configure (String configFilename)

```
<! DOCTYPE log4j: configuration SYSTEM "log4j.dtd">
<log4j: configuration debug="true" xmlns:log4j='http://jakarta.apache.org/log4j/'>
 <appender name="console" class="org.apache.log4j.ConsoleAppender">
  <param name="Target" value="System.out"/>
  <layout class="org.apache.log4j.PatternLayout">
  <param ame="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n" />
</layout>
 </appender>

 <appender name="fileAppender" class="org.apache.log4j.RollingFileAppender">
  <param name="File" value="demoApplication.log"/>
  <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n" />
  </layout>
 </appender>

 <root>
  <priority value ="debug"></priority>
  <appender-ref ref="console"></appender>
  <appender-ref ref="fileAppender"></appender>
 </root>
</log4j:configuration>
```


 ashok.javatraining@gmail.com

```
public class XMLFileLog4jExample {

	static Logger logger = Logger.getLogger(XMLFileLog4jExample.class);

	public static void main(String[] args) {
		String log4jConfigFile = System.getProperty("user.dir")
				+ File.separator + "log4j.xml";
		DOMConfigurator.configure(log4jConfigFile);

		logger.debug("this is a debug log message");
		logger.info("this is a information log message");
		logger.warn("this is a warning log message");
	}
}
```

XMLFileLog4jExample.java

Programmatic configuration example

When necessary, all log4j's configuration can be set programmatically. The following example program does the same thing as Properties file configuration example and XML file configuration example, but without creating any external configuration file:

```
import org.apache.log4j.Appender;
import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;

public class ProgrammaticLog4jExample {
    public static void main (String[] args) {
        // creates pattern layout
        PatternLayout layout = new PatternLayout();
        String conversionPattern = "%-7p %d [%t] %c %x - %m%n";
        layout.setConversionPattern(conversionPattern);

        // creates console appender
        ConsoleAppender consoleAppender = new ConsoleAppender();

        consoleAppender.setLayout(layout);
        consoleAppender.activateOptions();

        // creates file appender
        FileAppender fileAppender = new FileAppender();
        fileAppender.setFile("applog3.txt");
        fileAppender.setLayout(layout);
        fileAppender.activateOptions();

        // configures the root logger
        Logger rootLogger = Logger.getRootLogger();
        rootLogger.setLevel(Level.DEBUG);
        rootLogger.addAppender(consoleAppender);
        rootLogger.addAppender(fileAppender);
        // creates a custom logger and log messages
        Logger logger = Logger.getLogger(ProgrammaticLog4jExample.class);
        logger.debug("this is a debug log message");
        logger.info("this is a information log message");
        logger.warn("this is a warning log message");
    }
}
```


 ashok.javatraining@gmail.com

Conversion Patterns in Log4j

A conversion pattern specifies how log messages are formatted, using a combination of literals, conversion characters and format modifiers. Consider the following pattern

Ex : log4j.appender.ConsoleAppender.layout.ConversionPattern=[%-5p] %d %c - %m%n

Where:

The percent sign (%) is a prefix for any conversion characters.
Conversion characters are: p, d, c, m and n. These characters are explained in the table below.

-5p is the format modifier for the conversion character p (priority), which left justifies the priority name with minimum 5 characters.

Other characters are literals: [,], - and spaces

That pattern will format log messages something like this:

```
[DEBUG] 2012-11-02 21:57:53,661 MyClass - this is a debug log message
[INFO] 2012-11-02 21:57:53,662 MyClass - this is a information log message
[WARN] 2012-11-02 21:57:53,662 MyClass - this is a warning log message
```

Briefly summarizes the conversion characters defined by log4j

What to print	Conversion character	Performance
Category name (or logger name)	**c**	Fast
Fully qualified class name	**C**	Slow
Date and time	**d** **d{format}**	*Slow if using JDK's formatters.* *Fast if using log4j's formatters.*
File name of Java class	**F**	**Extremely slow**
Location (class, method and line number)	**l**	**Extremely slow**
Line number only	**L**	**Extremely slow**
Log message	**m**	Fast
Method name	**M**	**Extremely slow**
Priority (level)	**p**	Fast
New line separator	**n**	Fast
Thread name	**t**	Fast
Time elapsed (milliseconds)	**r**	Fast
Thread's nested diagnostic context	**x**	Fast
Thread's mapped diagnostic context	**X**	Fast
Percent sign	**%%**	Fast

 ashok.javatraining@gmail.com

Log4J with FileAppender

```
log4j.rootLogger=DEBUG, fileAppender
log4j.appender.fileAppender=org.apache.log4j.RollingFileAppender
log4j.appender.fileAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.fileAppender.layout.ConversionPattern=[%t] %-5p %c %x - %m%n
log4j.appender.fileAppender.File=demoApplication.log
```

Log4j with ConsoleAppender

```
log4j.rootCategory=debug, console
log4j.logger.com.demo.package=debug, console
log4j.additivity.com.demo.package=false

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.out
log4j.appender.console.immediateFlush=true
log4j.appender.console.encoding=UTF-8
#log4j.appender.console.threshold=warn

log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.conversionPattern=%d [%t] %-5p %c - %m%n
```

Log4j with JDBCAppender

The JDBCAppender provides mechanism for sending log events to a database tables. Each append call adds to an ArrayList buffer. When the buffer is filled each log event is placed in a sql statement (configurable) and executed. BufferSize, db URL, User, & Password are configurable options in the standard log4j ways.

```
# Define the root logger with jdbc appender
log4j.rootLogger = DEBUG, sql

# Define the file appender
log4j.appender.sql=org.apache.log4j.jdbc.JDBCAppender
log4j.appender.sql.URL=jdbc:mysql://localhost/test
# Set Database Driver
log4j.appender.sql.driver=com.mysql.jdbc.Driver
# Set database user name and password
log4j.appender.sql.user=root
log4j.appender.sql.password=password
# Set the SQL statement to be executed.
log4j.appender.sql.sql=INSERT INTO LOGS VALUES ('%x', now() ,'%C','%p','%m')
# Define the xml layout for file appender
log4j.appender.sql.layout=org.apache.log4j.PatternLayout
```


 ashok.javatraining@gmail.com

Create the table in database and test the application

```
CREATE TABLE LOGS
(
        USER_ID VARCHAR(20) NOT NULL,
        DATED   DATETIME NOT NULL,
        LOGGER VARCHAR (50) NOT NULL,
        LEVEL   VARCHAR (10) NOT NULL,
        MESSAGE VARCHAR (1000) NOT NULL
);
```

Now, configure the log4j.properties file using **PropertyConfigurator** and call some log events.

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

public class Log4jJDBCExample{
    static Logger log = Logger.getLogger(Log4jJDBCExample.class);

    public static void main(String[] args){
        PropertyConfigurator.configure("log4j.properties");

        log.debug("Sample debug message");
        log.info("Sample info message");
        log.error("Sample error message");
        log.fatal("Sample fatal message");
    }
 }
```

Log4j with Multiple Appenders

```
log4j.rootLogger=DEBUG, consoleAppender, fileAppender

#Consoler appender details
log4j.appender.consoleAppender=org.apache.log4j.ConsoleAppender
log4j.appender.consoleAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.consoleAppender.layout.ConversionPattern=[%t] %-5p %c %x - %m%n

#File appender details
log4j.appender.fileAppender=org.apache.log4j.RollingFileAppender
log4j.appender.fileAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.fileAppender.layout.ConversionPattern=[%t] %-5p %c %x - %m%n
log4j.appender.fileAppender.File=demoApplication.log
```


 ashok.javatraining@gmail.com

Configuring Log4J in Web Applications

Typically, for Java web application, we can put the log4j.properties file or log4j.xml file under the root of the classpath (WEB-INF\classes directory) to use log4j immediately in a servlet class as follows:

```
Logger logger = Logger.getLogger(MyServlet.class);
logger.debug ("this is a debug log message");
```

However, what if we put log4j configuration file in another location rather than the root of the classpath? In this case, we need to initialize log4j explicitly like this:

```
String log4jConfigFile = "some/path/log4j.properties";
PropertyConfigurator.configure (log4jConfigFile);
```

But this initialization code should be executed only once when the application is being started. To do so, it's recommended to declare an implementation of the ServletContextListener interface to listen to the contextInitialized() event which happens when the application is being started, before serving client's requests. For example:

```
public class ContextListener implements ServletContextListener {
   @Override
   public void contextInitialized(ServletContextEvent event) {
      // code to initialize log4j here...
   }
}
```

Here are the steps to initialize and use log4j in a Java web application:

Step 1: Create a Web application in Eclipse IDE (below screen shot display project structure)

 ashok.javatraining@gmail.com

Step 2: Creating log4j properties file with below content in WEB-INF folder

```
# LOG4J configuration
log4j.rootLogger=DEBUG, Appender1,Appender2

log4j.appender.Appender1=org.apache.log4j.ConsoleAppender
log4j.appender.Appender1.layout=org.apache.log4j.PatternLayout
log4j.appender.Appender1.layout.ConversionPattern=%-7p %d [%t] %c %x - %m%n

log4j.appender.Appender2=org.apache.log4j.FileAppender
log4j.appender.Appender2.File=D:/Logs/Log4jWebDemo.log
log4j.appender.Appender2.layout=org.apache.log4j.PatternLayout
log4j.appender.Appender2.layout.ConversionPattern=%-7p %d [%t] %c %x - %m%n
```

This configuration basically creates two appenders: the first for console logging and the second for file logging (log file will be created under D:/Logs/Log4jWebDemo.log).

Step 3: Configuring location of log4j properties file in web.xml

It's also recommended to have the location of log4j.properties file configurable via web.xml file like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">

    <display-name>Log4jWebDemo1</display-name>

    <context-param>
        <param-name>log4j-config-location</param-name>
        <param-value>WEB-INF/log4j.properties</param-value>
    </context-param>

</web-app>
```


 ashok.javatraining@gmail.com

Step 4: Create a Context Listener class that implements ServletContextListener interface with below code

```
package com.web.apps;

import java.io.File;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

import org.apache.log4j.PropertyConfigurator;

@WebListener
public class ContextLister implements ServletContextListener {

	public void contextDestroyed(ServletContextEvent arg0) {
		//TODO:destruction logic goes here
	}

	public void contextInitialized(ServletContextEvent event) {
		// initialize log4j here
		ServletContext context = event.getServletContext();
		String log4jConfigFile = context.getInitParameter("log4j-config-location");
		String fullPath = context.getRealPath("") + File.separator+ log4jConfigFile;
		PropertyConfigurator.configure(fullPath);
	}
}
```

ContextListener.java

In the contextInitialized () method, we read location of the log4j properties file and construct a complete, absolute path of this file which is then passed to the static configure() method of the PropertyConfigurator class. That's way log4j is initialized with the given properties file.

Note that the @WebListener annotation (Servlet 3.0) is placed before the class declaration to tell the servlet container to register this class as a listener

 ashok.javatraining@gmail.com

Step 5: Create a simple servlet to test log4j. Create a TestLog4jServlet.java class with the following content:

```
package com.web.apps;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.log4j.Logger;

@WebServlet("/TestServlet")
public class TestLog4jServlet extends HttpServlet {

	static final Logger LOGGER = Logger.getLogger(TestLog4jServlet.class);

	protected void doGet(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {

		LOGGER.info("This is a logging statement from log4j");

		String html = "<html><h2>Log4j has been initialized successfully!</h2></html>";
		response.getWriter().println(html);
	}
}
```

TestLog4jServlet.java

This servlet simply writes a log entry to the logger, and sends a short HTML text to the client.
Since we use the @WebServlet annotation (Servlet 3.0), there is no need to configure this servlet in the web.xml file

Step 6: Now deploy the project into Server and send the request to access Servlet, you can see log file will be created in the given path.

Note: In Real time projects, Application infra team/server team will mount application logs into some machine and they will share that machine details and path of the log files. We will connect to that machine using Putty and WinSCP tools to get/access the log files.

 ashok.javatraining@gmail.com