
Core Java 8 and
Development Tools
Lesson 19 : Logging with Log4J
Capgemini

Lesson Objectives

After completing this lesson, participants will be able to
- Understand the necessity of Logging
- Work with log4j components
- Configure java application for logging

Every Programmer is familiar with the process of inserting calls to System.out.println() into troublesome code to gain insight into program behavior. Of course, once you have figured out the cause of trouble, you remove the print statements, only to put them back in when the next problem surfaces. The logging API is designed to overcome the problem.
This lesson talks about Log4j, a package used to output log statements to a variety of output targets

Lesson outline:

19.1: Introduction

Overview of Logging

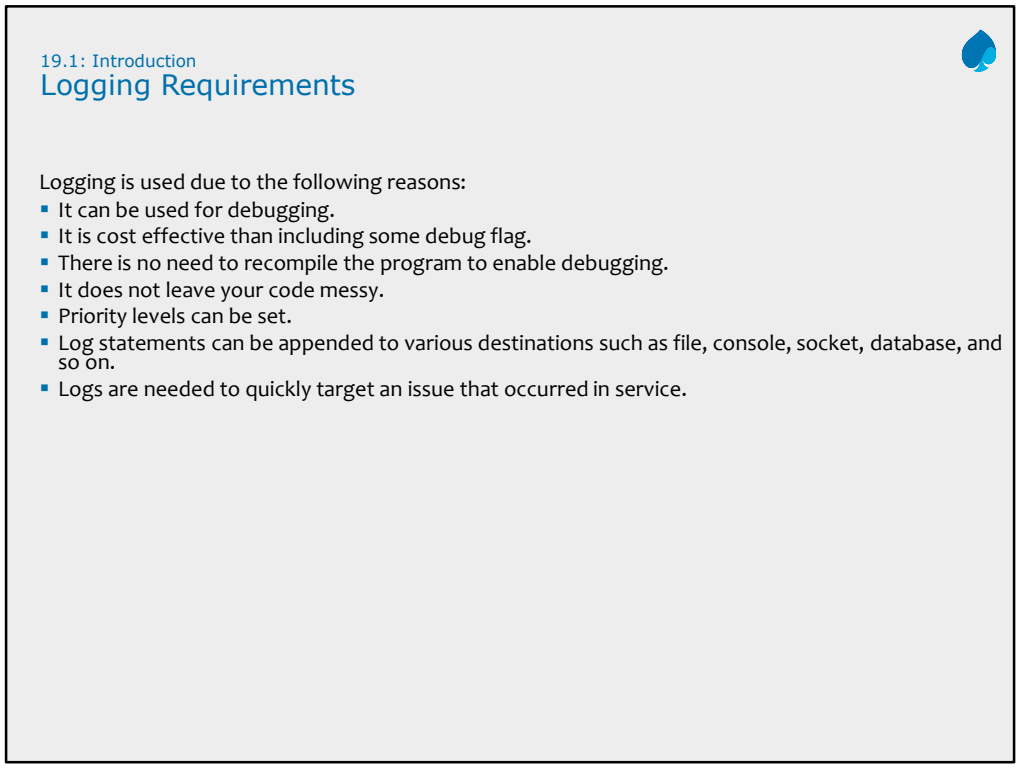
Logging is writing the state of a program at various stages of its execution to some repository such as a log file.

By logging, simple yet explanatory statements can be sent to text file, console, or any other repository.

Using logging, a reliable monitoring and debugging solution can be achieved.

What is Logging?

Logging, or writing the state of a program at various stages of its execution to some repository such as a log file, is an age-old method used for debugging and monitoring applications.



Logging Requirements:
One of the challenges in any programming environment is to be able to debug the code effectively. Suppose an application that is running in unattended mode on the server is not behaving properly. Everything seems to work fine in your development and test environment; yet in the production environment, something seems not to be "working" correctly. If you decide to print to standard output or to a log file, as an application developer, then you need to worry about commenting out the code in production to reduce the overhead associated with the calls.
Another approach is to define a Boolean variable, say debug, and if the value of the variable is true, then the application prints a whole set of debug messages. You compile by switching the flag one way or the other to get the necessary behavior. This is computationally expensive in addition to being cumbersome.
With the logging API, however, you do not need to recompile your program every time you want to enable debugging. Furthermore you can set different levels for logging messages without incurring too much computational expense. You can even specify the kind of messages you want to log. Using a configuration file, you can change the runtime level of the logging information. This information can be written to a file, a screen console, a socket, a database, or any combination. It can be very detailed or very sparse, based on the level set at runtime, and may differ for various consumers of the information. For detailed analysis, examine the log file to discover when and where a problem occurs.

19.2: Log4J Concepts

Concept of Log4J

Log4j is an open source logging API for Java.

- It handles inserting log statements in application code, and manages them externally without touching application code, by using external configuration files.
- It categorizes log statements according to user-specified criteria and assigns different priority levels to these log statements.
- It lets users choose from several destinations for log statements, such as console, file, database, SMTP servers, GUI components.
- It facilitates creation of customized formats for log output and provides default formats.

What is Log4J?

Log4j is an opensource logging API for Java. This logging API became so popular that it has been ported to other languages such as C, C++, Python, and even C# to provide logging framework for these languages.

Log4j categorizes log statements according to user-specified criteria and assigns different priority levels to these log statements. These priority levels decide which log statements are important enough to be logged to the log repository.

Log4j lets users choose from several destinations for log statements, such as console, file, database, SMTP servers, GUI components; with option of assigning different destinations to different categories of log statements.

These log destinations can be changed anytime by simply changing log4j configuration files.

Advantages of Log4J are as follows:

It organizes the log output in separate categories, makes it easy to pinpoint the source of an error.

Log4j facilitates external configuration at runtime, makes the management and modification of log statements very simple.

Log4j assigns priority levels to loggers and log requests, helps weed out unnecessary log output, and allows only important log statements to be logged.

19.2: Log4J Concepts

Components

Log4j comprises of three main components:

- Logger
- Appender
- Layout

Users can extend these basic classes to create their own loggers, appenders, and layouts.

Log4J Concepts:
Logically, log4j can be viewed as being comprised of three main components: logger, appender, and layout.
The functionalities of each of these components are accessible through Java classes of the same name. Users can extend these basic classes to create their own loggers, appenders, and layouts.

19.2: Log4J Concepts

Logger

The component logger accepts log requests generated by log statements.

Logger class provides a static method getLogger(name).

- This method:
  - Retrieves an existing logger object by the given name (or)
  - Creates a new logger of given name if none exists.
- It then sends their output to appropriate destination called appenders.

Logger:
The component logger accepts or enables log requests generated by log statements (or printing methods) during application execution. Subsequently, it sends their output to appropriate destination, that is appender(s), specified by user. The logger component is accessible through the Logger class of the log4j API.

19.2: Log4J Concepts

Logger

The logger object is then used to ….

- set properties of logger component
- invoke methods which generate log requests, namely:
  - debug(), info(), warn(), error(), fatal(), and log()

Each class in the Java application being logged can have an individual logger assigned to it or share a common logger with other classes.

Any number of loggers can be created for the application to suit specific logging needs.

Logger:

One can create any number of loggers for the application to suit specific logging needs.

It is a common practice to create one logger for each class, with a name same as the fully-qualified class name. This practice helps to:

organize the log outputs in groups by the classes they originate from,

identify origin of log output, which is useful for debugging

19.2: Log4J Concepts

Logger

Log4j provides a default root logger that all user-defined loggers inherit from.

- Root logger is at the top of the logger hierarchy of all logger objects that are created.
- If an application class does not have a logger assigned to it, it can still be logged using the root logger.
- Root logger always exists and cannot be retrieved by name.

19.2: Log4J Concepts

Ways to create a Logger

Retrieve the root logger:

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

Create a new logger:

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

Instantiate a static logger globally, based on the name of the class:

Set the level with:

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

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

19.2: Log4J Concepts

Logger Priority Levels

Loggers can be assigned different levels of priorities.
Priority levels in ascending order of priority are as follows:

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

Logger Priority Levels:
Loggers can be assigned different levels of priorities.
These priority levels decide which log statement is going to be logged.
There are five different priority levels: DEBUG, INFO, WARN, ERROR, and FATAL; in ascending order of priority.

19.2: Log4J Concepts

Logger Priority Levels

There are printing methods for each of these priority levels.

mylogger.info("logstatement1");

- generates log request of priority level INFO for logstatement1

Logger Priority Levels (contd.):

As we can see, log4j has corresponding printing methods for each of these priority levels. These printing methods are used to generate log requests of corresponding priority level for log statements.

For example: mylogger.info("logstatement-1"); generates log request of priority level INFO for logstatement-1.

19.2: Log4J Concepts

Logger Priority Levels

In addition, there are two special levels of logging available:

- ALL: It has lowest possible rank. It is intended to turn on all logging.
- OFF: It has highest possible rank. It is intended to turn off logging.

The behavior of loggers is hierarchical.

The root logger is assigned the default priority level DEBUG.

19.2: Log4J Concepts

Logger Priority Levels

- The behavior of loggers is hierarchical. The following table illustrates this situation.
  - Logger Output Hierarchy:

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

Logger Priority Levels (contd.):

In log4j, there are five normal levels of logger available:

DEBUG: It prints the debugging information and is helpful in the development stage.

INFO: It prints informational messages that highlight the progress of the application.

ERROR: It prints error messages that might still allow the application to continue running.

WARN: It prints information related to some faulty and unexpected behavior of the system, which needs attention in near future or else can cause malfunctioning of the application.

FATAL: It prints system critical information, which are causing the application to crash.

ALL: It has the lowest possible rank and is intended to turn on all logging.

OFF: It has the highest possible rank and is intended to turn off all the logging.

19.2: Log4J Concepts

Logger Priority Levels

A logger will only output messages that are of a level greater than or equal to it.

If the level of a logger is not set, it will inherit the level of the closest ancestor.

If a logger is created in the package com.foo.bar and no level is set for it, then it will inherit the level of the logger created in com.foo.

19.2: Log4J Concepts

Logger Priority Levels

If no logger was created in com.foo, then the logger created in com.foo.bar will inherit the level of the root logger.

The root logger is always instantiated and available. The root logger is assigned the level DEBUG.

19.2: Log4J Concepts

Examples on Priority Levels

```
/* Instantiate a logger named MyLogger */
   Logger mylogger = Logger.getLogger("MyLogger");
/* Set logger priority l*/
     mylogger.setLevel(Level.INFO);
/* statement logged, since INFO = INFO*/
     mylogger.info(" values ");
/* statement not logged, since DEBUG < INFO*/ mylogger.debug(“not logged");
/* statement logged, since   ERROR >INFO*/  mylogger.error(“logged");
```

19.2: Log4J Concepts

Appender

Appender component is interface to the destination of log statements, a repository where the log statements are written/recorded.

A logger receives log request from log statements being executed, enables appropriate ones, and sends their output to the appender(s) assigned to it.

The appender writes this output to repository associated with it.

- Examples: ConsoleAppender, FileAppender, WriterAppender, RollingFileAppender, DailyRollingFileAppender

Appender:

There are various appenders available such as:

ConsoleAppender: It appends log events to System.out or System.err using a layout specified by the user

FileAppender: It appends log events to a file.

WriterAppender: It appends log events to a Writer or an OutputStream depending on the user's choice.

RollingFileAppender: It extends FileAppender to backup the log files when they reach a certain size.

DailyRollingFileAppender: It extends FileAppender so that the underlying file is rolled over at a user chosen frequency.

SMTPAppender: It sends an e-mail when a specific logging event occurs, typically on errors or fatal errors

SyslogAppender: It sends messages to a remote syslog daemon.

TelnetAppender: It specializes in writing to a read-only socket.

SocketAppender: It sends LoggingEvent objects to a remote a log server, usually a SocketNode.

SocketHubAppender: It sends LoggingEvent objects to a set of remote log servers, usually a SocketNodes.

An appender is assigned to a logger using the addAppender( ) method of the Logger class, or through external configuration files. A logger can be assigned one or more appenders that can be different from appenders of another logger. This is useful for sending log outputs of different priority levels to different destinations for better monitoring.

19.2: Log4J Concepts

Layout

The Layout component defines the format in which the log statements are written to the log destination by "appender".

Layout is used to specify the style and content of the log output to be recorded.

- This is accomplished by assigning a layout to the appender concerned.

Layout is an abstract class in log4j API.

- It can be extended to create user-defined layouts.

Layout:
The Layout component defines the format in which the log statements are written to the log destination by appender.
Layout is used to specify the style and content of the log output to be recorded, such as inclusion/exclusion of date and time of log output, priority level, info about the logger, line numbers of application code from where log output originated, and so forth. This is accomplished by assigning a layout to the appender concerned.

19.2: Log4J Concepts

Types of Layout

Let us discuss the different types of layouts:

- HTMLLayout: It formats the output as a HTML table.
- PatternLayout: It formats the output based on a conversion pattern specified. If none is specified, then it uses the default conversion pattern.
- SimpleLayout: It formats the output in a very simple manner, it prints the Level, then a dash "-", and then the log message.
- XMLLayout: It formats the output as a XML.

Note: The sample program's are given later (after installation of log4j).

19.3: Installation of Log4J

Steps

Let us see the steps for installation of Log4J:

- Download log4j-1.2.4.jar from http://logging.apache.org/log4j
- Extract the log4j-1.2.4.jar at any desired location and include its absolute path in the application's CLASSPATH.
- Now, log4j API is accessible to user's application classes and can be used for logging.

Installation of Log4J:
Let us see the steps for installation of Log4J:
Download and unzip logging-log4j-1.2.9.zip.
Set the LOG4J_HOME system variable to the directory where you installed Log4j.
For example: LOG4J_HOME=C:\Tools\logging-log4j-1.2.9
Edit the CLASSPATH system variable:
For example: CLASSPATH=%CLASSPATH%;%LOG4J_HOME%\dist\lib\log4j-1.2.9.jar
Note: The jar file is located in Log4j_HOME\lib\log4j-1.2.9.jar
Create the following directories:
c:\demo
c:\demo\com
c:\demo\com\igate
Create the file c:\demo\com\igate\Log4jDemo.java.
import org.apache.log4j.Logger;

```
public class Log4jDemo {
    static Logger log = Logger.getLogger("Log4jDemo");
    public static void main(String args[]) {
        log.debug("This is my debug message.");
        log.info("This is my info message.");
        log.warn("This is my warn message.");
        log.error("This is my error message.");
        log.fatal("This is my fatal message.");
    }
```

19.4: Configuring Log4J

Process

The log4j can be configured both programmatically and externally using special configuration files.

External configuration is most preferred.

- This is because to take effect it does not require change in application code, recompilation, or redeployment.

Configuration files can be XML files or Java property files that can be created and edited using any text editor or xml editor, respectively.

Configuring Log4J:

Inserting log requests into the application code requires a fair amount of planning and effort. Observation shows that approximately 4 percent of code is dedicated to logging. Consequently, even moderately sized applications will have thousands of logging statements embedded within their code. Given their number, it becomes imperative to manage these log statements without the need to modify them manually.

The log4j environment is fully configurable programmatically.

The simplest configuration file will contain following specifications that can be modified, both programmatically and externally, to suit specific logging requirements:

priority level and name of appender assigned to root logger

Appender's type

layout assigned to the appender

19.4: Configuring Log4J

Using Property File

The root logger is assigned priority level DEBUG and an appender named myAppender:

- log4j.rootLogger=debug, myAppender

The appender's type specified as ConsoleAppender, that is logs output to console:

- log4j.appender.myAppender=org.apache.log4j.ConsoleAppender

The appender is assigned a layout SimpleLayout:

- log4j.appender.myAppender.layout=org.apache.log4j.SimpleLayout

19.4: Configuring Log4J

Using Configuration Property Files

Example 1:

```
package com.igate.sampleapp;
import org.apache.log4j.*;
public class MyClass
{    static Logger myLogger =
Logger.getLogger(MyClass.class.getName( ));
      //no constructor required
     public void do_something( int a, float b)
     {    myLogger.info("Logged since INFO=INFO"); ...
          myLogger.debug("not enabled, since DEBUG <
INFO");      ...                                          contd.
```

19.4: Configuring Log4J

Using Configuration Property Files

Example 1 (contd.):

```
                if (x == null)
                {myLogger.error("enabled & logged ,since ERROR >
INFO");...}
    }
  }
```

- //Execute the file as
  // java -Dlog4j.configuration=config-sample.properties com.igate.sampleapp.MyClass

19.4: Configuring Log4J

Using Configuration Property Files

Example 2:

```
package com.igate.sampleapp;
import org.apache.log4j.*;
import org.apache.log4j.PropertyConfigurator;
import java.net.*;
public class MyClass
{    static Logger myLogger =
Logger.getLogger(MyClass.class.getName( ));
     public MyClass()
      {
              PropertyConfigurator.configure("config-
sample.properties");
      }
```

contd.

Note: This version of MyClass instructs PropertyConfigurator to parse a configuration file config-sample.properties and set up logging accordingly.

19.4: Configuring Log4J

Using Configuration Property Files

Example 2 (contd.):

```
public void do_something( int a, float b)
{           myLogger.info("Logged since INFO=INFO"); ...
            myLogger.debug("not enabled, since DEBUG < INFO");
...
    if (x == null) {
      myLogger.error("enabled & logged ,since ERROR > INFO");...
    }
}
```

19.4: Configuring Log4J

Demo

MyClass_Property.java

19.5: Pros and Cons

Log4j

Advantages:

- Log4j print methods do not incur heavy process overhead.
- External configuration makes management and modification of log statements simple and convenient.
- Priority level of log statements helps to weed out unwanted logging.

ShortComings:

- Appender additivity may result in log requests being sent to unwanted appenders.

Pros and Cons of Log4J:

The advantages of using log4j are listed below:

The log4j printing methods used for logging can always remain in application code because they do not incur heavy process overhead for the application and assist in ongoing debugging and monitoring of application code, thus proving useful in the long term.

Log4j organizes the log output in separate categories by the name of generating loggers that in turn are same as the names of the classes they log. This approach makes pinpointing the source of an error easy.

Log4j facilitates external configuration at runtime. This makes the management and modification of log statements very simple and convenient as compared to performing the same tasks manually.

Log4j assigns priority levels to loggers and log requests. This approach helps weed out unnecessary log output and allows only important log statements to be logged.

The shortcomings of log4j are listed below:

Appender additivity may result in the log requests being unnecessarily sent to many appenders and useless repetition of log output at an appeneder. Appender additivity is countered by preventing a logger from inheriting appenders from its ancestors by setting the additivity flag to false.

When configuration files are being reloaded after configuration at runtime, a small number of log outputs may be lost in the short time between the closing and reopening of appenders. In this case, Log4j will report an error to the stderr output stream, informing that it was unable send the log outputs to the appender(s) concerned. But the possibility of such a situation is minute. Also, this can be easily patched up by setting a higher priority level for loggers.

Lab : Log4J

Lab 12: Log4J

Summary

In this lesson, you have learnt

- Log 4j and its components
- The method to log messages using Log4J API
- Configuring Log4j applications

Review Question

Question 1: Log4j is a product of ___.

- Option1: RedHat
- Option2: Sun
- Option3: Apache
- Option4: None of the above

Review Question

Question 2: Configuration of Log4j applications can be done through:

- Option1: Property files
- Option2: XML files
- Option 3: Both Option1 and Option2 are right
- Option4: None of the above is true

Review Question

Question 3: If the loglevel is set to INFO, will the following message be logged?
Message: logger.warn(" From warning");

- Yes / No