Logging with Log4J

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.

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.

- Log4j comprises of three main components:
  - Logger
  - Appender
  - Layout

Logger

- 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.
- 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

- 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.

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:

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

- Set the level with:

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

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

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.

Logger Priority Levels

The behavior of loggers is hierarchical.

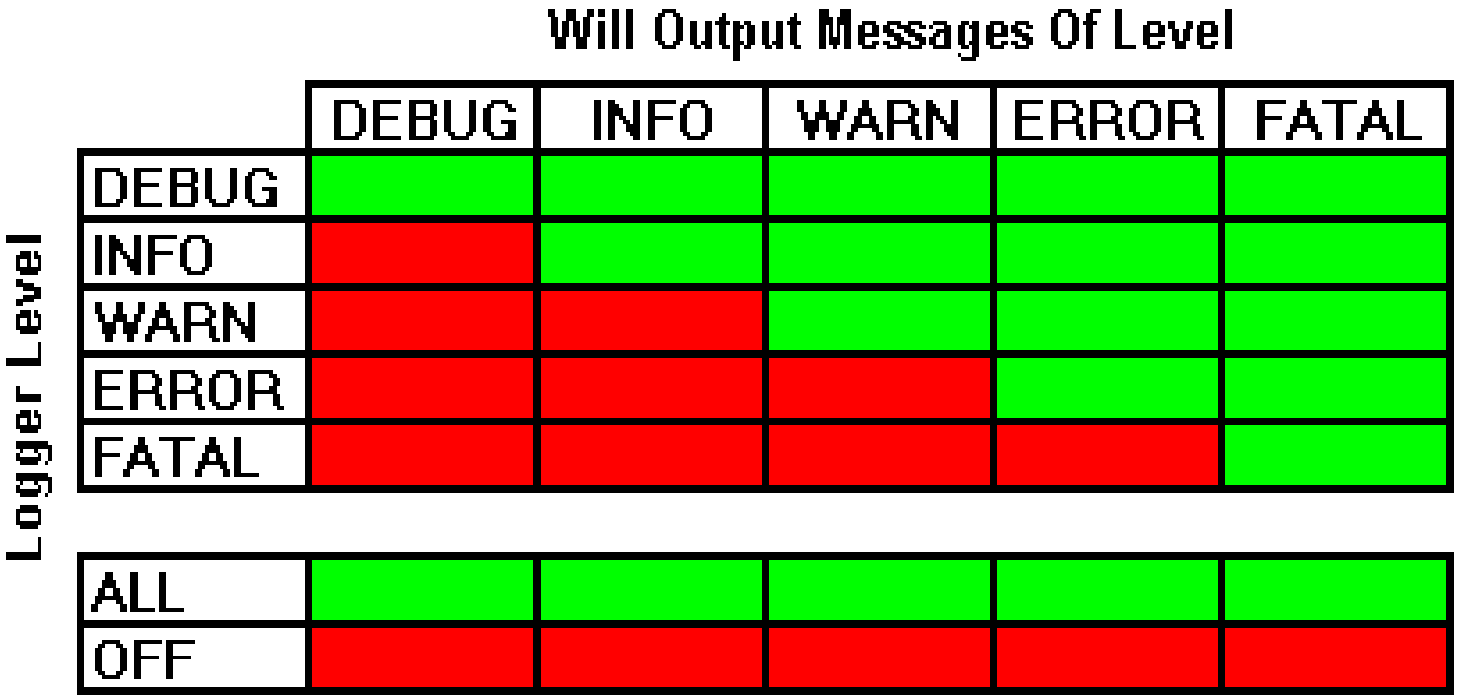
For example, If a logger is created in the package com.foo.bar and no level is set for it, then it will inherit the level of the logger created in com.foo.

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.

Note: The root logger is assigned the default priority level DEBUG.

Logger Priority Levels

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



Examples on Priority Levels

- /* Instantiate a logger named MyLogger */
  Logger mylogger = Logger.getLogger("MyLogger");
- /* Set logger priority I*/
  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");

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

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.

Layout

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

Types of layout:

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.

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.

Steps

- Add the *log4j-1.2.15.jar* to the classpath *(environment variables).*
- Create an instance of Logger class.
- Log4j environment is created by the method,
- *Logger.getLogger (XXX.class)* method. It takes one argument, the fully qualified class name.
- Create configuration file, ***log4j.xml*** or properties file, ***log4j.properties***
- Specify the appender name, type and other details.
  - *Note: The log4j.xml file should be placed in the project's class path. You can place the file in* ***src*** *folder or create a source folder, say resource and place in it.*

Configuration file for Logger can be either by an XML file or properties file

Log4j.xml

•**Right-Click on your project ->new-> folder. folder name: log**

Create Log4j.xml file in src folder
Right-Click on *src* folder->new->xml file (File name : Log4j.xml)

Import *log4j-1.2.12* jar file into your project as external jar file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j='http://jakarta.apache.org/log4j/'>
<appender name="ERROR_LOG_FILE" class="org.apache.log4j.FileAppender">
      <param name="File" value="./log/ErrorLog.log"/>
      <param name="Append" value="true"/>
      <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss,SSS}
            %-5p[%t]%c[%M] - %m%n"/>
      </layout>
</appender>

<root>
      <level value="DEBUG" />
      <appender-ref ref="ERROR_LOG_FILE" />
</root>
</log4j:configuration>
```

Log4j.xml

…….

<layout class=*"org.apache.log4j.PatternLayout"*><param name=*"ConversionPattern"* *value****="%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p[%t]%c[%M] - %m%n"/>***
</layout>…….

https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html

The conversion specifier **%-5p** means the priority of the logging event should be left justified to a width of five characters, **SSS** : milliseconds
c : to output the category of the logging event(packagename.classname) .

M: To output the method name where the logging request was issued.

p: to output the name of the thread that generated the logging event.

m: To output the application supplied message associated with the logging event.
n: generate new line, **t** : To output the name of the thread that generated the logging event

```
2016-06-14 10:54:48,558
ERROR[main]com.wissen.businesstier.VehicleValidator[validate] -
VehicleValidator.INVALID_LICENCEPLATENUMBER
java.lang.Exception: VehicleValidator.INVALID_LICENCEPLATENUMBER
at com.wissen.bussinesstier.VehicleValidator.validate(VehicleValidator.java:38)
at com.wissen.bussinesstier.VehicleManager.getVehicle(VehicleManager.java:17)
at com.wissen.presentationtier.Tester.getVehicle(Tester.java:46)
at com.wissen.presentationtier.Tester.main(Tester.java:58)
```

log4j.properties

log4j.rootLogger=INFO, mylogger

Since rootLogger Level is set to INFO, mylogger LEVEL is also set to INFO

```
#log4j.appender.stdout=org.apache.log4j.ConsoleAppender
#log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
#log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n

#log4j.appender.mylogger=org.apache.log4j.RollingFileAppender
#log4j.appender.mylogger.File=basic.log
log4j.appender.mylogger=org.apache.log4j.FileAppender
#log4j.appender.mylogger.File=./log/basic.log
log4j.appender.mylogger.File=./log/emplog.html
#log4j.appender.mylogger.MaxFileSize=512KB
# Keep three backup files.
#log4j.appender.mylogger.MaxBackupIndex=3
# Pattern to output: date priority [category] - message
#log4j.appender.mylogger.layout=org.apache.log4j.SimpleLayout
log4j.appender.mylogger.layout=org.apache.log4j.HTMLLayout
#log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
#log4j.appender.mylogger.layout.ConversionPattern=%d{yyyy-MM-dd}-%t-%-5p-%-
10c:%m%n
```

Logging exceptions

```
public class EmployeeDemo {
public static Logger myLogger= Logger.getLogger(EmployeeDemo.class.getName());
public static void main(String[] args) {
try {
Employee employee=
new Employee(7934,"Vishal","CLERK",
null,LocalDate.of(2016, 2, 15),1200.0,0.0,20);
int n=addNewEmployee(employee);
if(n==1) {
System.out.println("New Employee Added");
myLogger.info("New employee added to database");
}else {
System.out.println("Unable to add new employee");
}
}catch (EmployeeException e) {
myLogger.error(e.getMessage(), e);
System.out.println("Error, check log file");
}
}
```

```
private static int addNewEmployee(Employee
employee) throws EmployeeException{
EmployeeManager manager=new
EmployeeManager();
int n=manager.addNewEmployee(employee);
return n;
}
}
```

Thank You!