JLog2 User Guide

Version 1.8

[A guide on how to use the jlog2 logging package]

Kieran Greer, Email: kierangreer@sourceforge.net.
http://jlog2.sourceforge.net/

Table of Contents

1 Introduction

The jlog2 software package is a logging application that allows you to log information relating to the operation of your software program. The package is written in Java under the subset J2ME platform and so might be of use with restricted mobile devices. It also contains some other useful utility classes that are not part of this subset of full Java. It is used by the licas system [http://licas.sourceforge.net/], for example.

The package provides functionality for logging or debugging general information, to different sets of streams. It is based on creating channels to write to, and then allocating these to each logger that is used. A channel could be anything and so could include something like a text area GUI component, which would be written to instead of one of the standard outputs. The logging can be controlled by the logging level and you can also add new custom channels to log to. The package is provided as a jar file that you include in your library and then use the relevant classes that you need.

The rest of the document is organised as follows: Section 2 describes some of the most important classes and methods in the logging package. Section 3 describes how you can create a logger to use in your code. Section 4 describes how you can write your own logger to log information to, while section 5 describes how to configure the loggers that are created. Section 7 provides information on some other useful utility classes.

2 Main Logger Classes

The logger classes are mainly static, meaning that there is only one instance of them. Before you use the logger however you need to initialise it. This is done by making a single call:

```
LoggerFactory.setLoggerFactory(new CustomLoggerFactory());
```

This only needs to be called once in your whole program and initialises the logger for use by reading the configuration file and setting the appropriate properties. There are a number of standard channels that can then be logged to, as specified in the configuration file. Simply put, the default channels can log to a file, or to the standard output and error streams. These streams are all stored in a `LoggerFactory` object. This is an abstract class, where a default `CustomLoggerFactory` is used if you do not supply any other one.

Each class then creates its own instance of a `Logger` class, as described in section 3. The logger for each class is recognised by the classpath description of the class that it belongs to.

This logger is then provided with references to the standard log channels that relate to it. The default configuration provided at the moment is for all loggers to log to only one set of channels and so it is not possible to provide loggers for different classes with different sets of channels. An extension to the `CustomLoggerFactory` would be required to provide this functionality. So, for example, if you declare default logging to a file and standard output, all created loggers will log to both of these streams automatically.

2.1 Dynamic Configurations

The current code has been changed slightly to make sure that each logger instance references a single central source of the default channels. That is, the standard outputs or the file. This means however that you can change the logging properties dynamically and updates to this central source will automatically be passed to every logger. To do that, you need to create a new set of properties, save them to the default properties file and load them in again. The following code should work:

```
propsParser = new PropertiesParser();
props = new Properties();

...
add new property values
...

propsParser.writeToFile(props);
LoggerFactory.getInstance().resetChannels();
```

The call to `resetChannels` will try to update rather than create from new.

2.2 Logging Messages

Any logging requests should then be done through the static `LoggerHandler` object. It can also dynamically change the logging level and so it makes sense to log through this. There are four different methods that can be called to log a message. These are:

```
public static void logMessage(Logger logger, String level, String
methodName, String message)
```

This method is called to log a message to the log channels. You need to include the logger to log to, the level the message is at, the name of the method the message relates to and the message itself. The message is then written to all of the channels related to the logger and also allowed by the configuration file for the specified logging level. A description of the different logging levels is given in section 5.

```
public static void debugMessage(Logger logger, String level, String methodName, String message)
```

This method is called to log a message to the debug channels. You need to include the logger to log to, the level the message is at, the name of the method the message relates to and the message itself. The message is then written to the appropriate channels as described previously.

```
public static void logError(Logger logger, String level, Exception error)
```

This method is called to log an error to the log channels. You need to include the logger to log to, the level the message is at and the error. The message is then written to the appropriate channels as described previously.

```
public static void debugError(Logger logger, String level, Exception error)
```

This method is called to log an error to the debug channels. You need to include the logger to log to, the level the message is at and the error. The message is then written to the appropriate channels as described previously.

The information that is logged is a follows:

- **Logging date:** the time that the message was written. This is determined automatically.
- **Logging Level:** the level of importance for the message. You need to set this yourself in the log request.
- **Logging class:** the class that is writing the message. You need to set this yourself during the logger initialisation.
- **Class method:** the method in the class that the message relates to. You need to set this yourself in the log request.
- **Log message:** the message to log. You set this yourself.

So, for example, the message below was logged on `18 May 2010 at 20:00`. It is at the `Message` level (see section 5), was logged by the class `org.licas.server.HttpServer`, inside the method `startRequestThread` and the message is '`Http Server started at: http://127.0.0.1:8888/`'.

Log Message Example:

```
Tue May 18 20:00:00 BST 2010: Message: org.licas.server.HttpServer: startRequestThread: Http Server started at: http://127.0.0.1:8888/
```

3 Creating a Standard Logger

To create a logger you need to declare a static object of type `Logger` in your class code and then initialise it through a static constructor. This logger will then be used by all instances of the class that are created. The logger needs to be initialised with the classname of the class it belongs to, for identification purposes. For example:

```
import org.jlog2.*;

public class YourClass
{
   /** The logger */
  private static Logger logger;

   static
   {
       // get the logger
       logger = LoggerHandler.getLogger(YourClass.class.getName());
       logger.setDebug(false);
   }
}
```

This would specify the logger for the `YourClass` class. To also output a debug trace for the specified file, you not only need to specify the configuration properties, but also set the debug variable for the logger to true. This means that you can debug only specific classes - that have the debug variable set to true. If no classes have the debug variable set then there will be no debug output trace. To log information, you then send the message to the `LoggerHandler`, for example:

```
LoggerHandler.logMessage(logger, LoggerHandler.INFO, "method", "message");
```

All logging should be done through a `LoggerHandler` call, which should also be used to choose the logging level.

3.1 Static Inheritance

Because the logger is static, there can be a problem if you declare the logger in a base class and then derive a number of classes from that. For example, if you write a class:

```
Class YourBaseClass ()
{
   protected static Logger logger;
}
```

And then create two derived classes, in the following order:

```
Class YourDerivedClass1 () extends YourBaseClass
{
     static
     {
        logger = LoggerHandler.getLogger(YourDerivedClass1.class.getName());
     }
}

Class YourDerivedClass2 () extends YourBaseClass
{
     static
     {
        logger = LoggerHandler.getLogger(YourDerivedClass2.class.getName());
     }
}
```

If an error from `YourDerivedClass1` is logged, it might be assigned the classname of `YourDerivedClass2`, because they have been assigned the same logger. This can be fixed by declaring the logger in each derived class and not in the base class.

4 Adding your own Custom Logger

You can also write your own logging channels by extending the `LogChannel` class and implementing the abstract methods. For example, you could create a swing `TextArea` component and choose to log to that instead. You then need to add the new channel to the `LoggerHandler` as a default channel. There is now only one set of channels that all loggers write to, so the new method code would be:

```
JTextArea jTextAreaLog;                  //swing text area
TextAreaChannel textAreaChannel;         //extends LogChannel

textAreaChannel = new TextAreaChannel(jTextAreaLog);
LoggerHandler.addLogChannel(LogConst.STDOUTID, textAreaChannel);
LoggerHandler.addDebugChannel(LogConst.STDOUTID, textAreaChannel);
LoggerFactory.getInstance().resetChannels();

logger = LoggerHandler.getLogger(YourClass.class.toString());
logger.setDebug(true);
```

Would initialise the logger handler to include the text area channel with every logger that is created, both for information and debugging, as well as any default ones specified in the configuration file. Note that calling reset will re-read the config file and remove anything not specified there. The text area can then be written to by implementing the `writeMessage` or `writeError` abstract methods of `LogChannel`.

5 Configuring the Logger

The logger can be configured by a configuration file. By default, this file should reside in a folder called `config` and an example folder and file should have been supplied with the package that you downloaded. The config folder should be in the same directory as the executable jar of your program, so that the logger can find it.

5.1 The Configuration Handler

If your mobile or other device, does not allow you to read the directory that the app is running from, you can use the `ConfigHandler` to set a different folder path. The default file name will then be appended to the end of the folder path. For example:

```
ConfigHandler.setConfigFolderPath("path to config folder");
```

This will read the config file from the path 'path to config folder' / logger.config.

```
ConfigHandler.setLogFolderPath("path to folder to write to");
```

This will write logging information to the folder 'path to config folder'. The log/debug file names are specified in the config file.

Note: The configuration handler should be called once only, at the very start of your program, before any static declarations for a new logger. Note that the declarations do not have to be made in static sections, in particular, not for the `Main` startup class.

5.2 Config File Structure

The file is a properties file and should be called `logger.config`. An example of what the file looks like is shown next, where the declared values are not case sensitive:

```
log.channels = FO
debug.channels = fe
log.file.path = licas.log
log.file.size = 100000
log.file.backup = 5
log.level = Info
debug.file.path = licas.debug
debug.file.size = 500
```

```
debug.file.backup = 1
debug.level = error
```

The configuration allows the user to specify default channels to log to. These can be standard output (O), standard error (E), or a file (F). To log to these simply specify the combination of these letters, for example:

1. To log to a file and standard output the instruction is: log.channels = FO
2. To debug to a file and standard error the instruction is: debug.channels = FE
3. The file.path property defines the name of the file to log to.

Each log file has a maximum size as specified by the file size property. This is the maximum number of lines allowed in the log file. When this maximum value has been reached, the file is cleared of all entries and then filled with the new log information until the maximum value is reached again, etc. For example:

1. To log to a file with a maximum number of 100000 lines the instruction is: log.file.size = 100000
2. To debug to a file with a maximum number of 10000 lines the instruction is: debug.file.size = 10000

You can also save earlier versions of the log file that would otherwise be overwritten when the maximum size is reached. The number of previous files that you keep is declared by the backup property. Declare the number of backup files to keep as follows:

1. log.file.backup = 5 means to keep 5 earlier versions of the log file. These are always the most recent earlier versions.
2. debug.file.backup = 1 means to keep one earlier version of the debug file. This is always the most recent earlier version.

Logging can be declared at different levels representing different levels of importance.
The levels available in order are: All, Info, Warn, Error, Severe, Message. Declare the level for each logger as follows:

1. To log to a level of info or higher (Info, Warn, Error, Severe): log.level = Info
2. To debug to a level of error or higher (Error, Severe): debug.level = Error

The following additional conditions apply:
1. When specifying the level in the config file, the 'All' level specifies to log everything.
2. When declaring the log level in your code, the `LoggerHandler.MESSAGE` level specifies to log the message no matter what level is set in the config file. The 'Message'

level means to always output the message, even if it is not associated with any specific problem or level, for example.

6 The Exception Handler

The logger is used directly through an `ExceptionHandler` and so these are now packaged together. This logs to both the error and log streams and includes some options for processing the exception. There is a `WriteMessageException` that you can extend with your own specific type. It has a setting to indicate to log the message only once and it will try to log only the message, not the full stack trace. This might lead to more tidy output, if the exception is repeatedly thrown, caught and processed, for example. To make sure that the message is logged only once, it is changed into an `EndOfException` exception in the handler, if the `useOnce` variable is set. It is still passed back however, so you can still catch it and process it outside of the handler. The handler also converts an `InvocationTargetException` into its enclosed exception, so that it can read the type.

7 Other Utility Classes

Although not part of a logger, there are some other utility classes that would be useful in a restricted environment. In J2ME, for example, not all of the String or Collection functionality is available in the available classes. The logger therefore includes some other basic utility classes as follows:

- **StringHandler:** can be used to tokenize a string, perform the replace operation, or test for containment.
- **CollectionHandler:** can be used to clone or merge Vector or Hashtable lists, for example.
- **FileLoader:** can be used to load data from local or remote files, or parse directories to return file or folder names, for example.
- **UuidHandler:** allows the creation of unique ids. An alternative to the Java system version.
- **Monitor:** can be used as part of thread synchronization.