phloc-logging 1.2

A generic Java Logging System

Philip Helger / phloc systems / ph@phloc.com

June 12, 2008

Abstract

Logging techniques are becoming more important in distributed environments in which communication is performed by message passing, or remote invocations, through unreliable networks like the Internet. As far as I know no general-purpose logging system exists because of the divergence of system requirements. This document introduces a new logging system called *phloc-logging* which is introduced and described in detail. Its internal operating and usage is also explained in detail. This paper finishes with a short outlook containing ideas for possible extensions and other uses.

1 Logging System

1.1 Introduction

Logging systems were originally used for auditing, keeping histories and reliability maintenance. Today, logging techniques are also popular in file system structuring, profiling and performance analysis, debugging, checkpointing and recovery (fault tolerance), analysis and some garbage collection systems.

1.2 Basic concepts

1.2.1 Log

Basically, a log is a piece of non-volatile storage that can be separated into records. It might be a plain text file, a special structured file, or a database [2].

1.2.2 Record

A log storage unit is the *record*. A record often consists of a descriptor and a body. The *descriptor* may contain information for interpreting the body semantics, to find the location of the record on the log and its origin, e.g. time stamp, logging client identifier, etc. The *body* is the information that the logging client wishes to store [2].

1.2.3 Logging semantics

A typical log often has the following semantics:

- append-only - *writes* always append records to the log
- Obsolete records may be *cleaned* or *removed*
- Logs can be *read* randomly or sequentially, in *backward* or *forward* order [2]

1.2.4 Logical and physical logs

Ruffin [2] distinguished two kinds of logs: logical and physical logs. A *physical log* is a real storage medium that store records from different logging clients, like files in a file system or databases. A *logical log* is a logical view of the client into physical logs. It only consists of records that are relevant to a particular client and can be compared to a view in a relational database.

1.2.5 Log level

To separate log records according to their importance *log levels* exist. A log level is meta-information to a log record and can be of any type as long as an order criteria can be defined on that type. A common log level system may contain the following items[3] (in order of ascending priority):

- *TRACE* designates finer-grained informational events than *DEBUG*
- *DEBUG* designates informational messages that highlight the progress of the application at fine-grained level
- *INFO* designates informational messages that highlight the progress of the application at coarse-grained level
- *WARN* designates potentially harmful situations
- *ERROR* error events that may still allow the application to continue running
- *FATAL* severe error events that will presumably lead the application to abort

1.2.6 Input and Output

When logging a record it needs to be taken into account, that the message source of a single record is not necessarily a string, but can also be a reference into a resource file that stores the message in a particular language. Normally the output of a log record is either discarded, according to the log level, or send to one or more receiver(s). Consider having a test machine which runs an application and logs each and everything into a database for later evaluation. Nevertheless, a user of the application expects error messages to occur on his user interface and a developer wants to see debug output without explicitly querying a database. The element that forwards input messages to output devices is called the *dispatcher*.

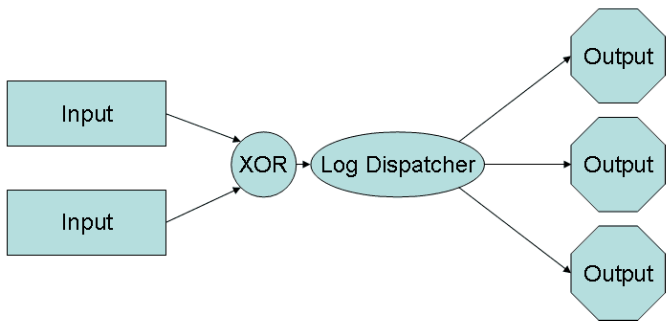

Figure 1: Input and output handling of a logging system

The following list exemplary summarizes the possibilities that Log4J[1] offers for output devices.

- JDBC[2] logging
- JMS[3] logging
- LogFactor5[4] logging
- Windows NT event logger
- Null logger that logs nothing
- SMTP[5] logger to log records via email
- Socket logger to log to a remote device
- Syslog logger to log to a remote syslog daemon
- Telnet logger to log to a Telnet socket
- Console logger to log to `stdout`[6] or `stderr`[7]
- File logger to log to a local file
- Asynchronous logging

In any large software system multilinguality is an issue since software can easily be spread over the Internet and used in many different countries. This can be an issue for a logging system as well in case it is used for end-user messages. In this case the input should not be a hard coded string but a language-independent identifier that can be used to resolve the real language-dependent string. It depends on the purpose of the logging whether a string needs to be translated or not. A rule of thumb is to make information (if visible to the user),

[1]A well known Java-based logging system maintained by the Apache foundation.
[2]JDBC: Java Database Connectivity
[3]JMS: Java Messaging Service
[4]A GUI for the Log4j framework
[5]SMTP: Simple Mail Transfer Protocol
[6]stdout: Standard output device
[7]stderr: Standard error device

warnings, errors and fatal errors language independent. Trace and debug statements normally don't need translation, since they should never be visible to the end user but only to developers for debugging purposes or to system managers for maintenance.

1.2.7 Log contexts

Each log record occurs in a special context that can be assembled from many different properties. These properties can include e.g. the time when the logging was performed, the name of the file in which the logging action was initiated or the name of the operating system on which the application is executed and many more. These properties can be divided into three different groups:

- Properties provided by the programmer (the message object, and additional information and the log level),
- Properties known at programming-time (the source file, the name of the class where logging is initiated etc.) and
- Properties known at runtime (the time when a message was issued, the name of the operating system etc.).

Each log record receives a copy of each property-value valid at the time the log record was initiated. Some of these property-values stay constant while running an application (e.g. the operating system name) and others change (e.g. the time). Other property-values stay constant for each invocation of the same log statement[8]mainly the property values concerning the source of a log statement) whereas some may change (e.g. the unique sequence number).

1.2.8 Additional information

Sometimes it is required to add additional information to a log record to further specify the logged information. An example for additional information is an exception[9] that was thrown by the application and can to be added to a log record to detail the information. This information should not limited to exceptions because erroneous objects may be passed as well to track down errors. The additional information may contain anything that does not necessarily belong to a log record but improves the usability for the person interpreting the log message.

1.3 Technical issues

1.3.1 Performance

One is of course the performance issue. In case of Java [4], where no pre-processor is available (like in C or C++), each logging statement within the code will a) increase the code size, which implies a higher memory consumption of the Java Virtual Machine (JVM) and b) decrease the execution speed, since each logging statement needs to be evaluated at runtime. According to Log4J[3] the determination whether a statement needs to be logged or not takes approximately

[8]The same log statement means a log statement that is executed more than once while an application is running.

[9]Some logging systems like Apache Commons Logging accept only exception objects as additional information but a general purpose logging system needs to handle all kind of objects.

5 nanoseconds on an AMD Opteron clocked at 800MHz with Sun JDK 1.3.1. According to my personal measurements with Log4J 1.2.8 on an Intel Pentium-M with 1.86GHz, 2GB RAM, Windows XP (SP 2) and Sun JDK 1.4.2 logging a record takes between 150 nanoseconds, if the logging messages is discarded by the first check, and 780 nanoseconds, if the logger object is acquired and logging takes place on a ”NullAppender”[10]. Of course this information depends on the logging system and hardware parameters like the CPU speed, the amount of main memory available and the used Java compiler.

1.3.2 Modularity

Another issue is the modularity of a logging system. An advanced logging system needs to handle a log record with more than one log handler e.g. for both persistent storage in a database and displaying on a console. Another aspect of modularity is the configuration of the logging system. It should be possible to handle log messages of one class different from the rest of the application by changing either the output devices, the log level or selecting different contexts (see section 2.2.2). This modularity can be achieved with a plugin-like system for log handlers that perform different actions depending on the desired output device. Of course each handler needs to have a common interface so that it can be dynamically added.

1.3.3 Flexibility

A logging system needs to be flexible so that it can be queried and modified at runtime. Queries include the currently registered log handlers in a certain context, or querying the available contexts. Modification includes changing the log level, adding or removing log handlers and changing the available log contexts at runtime. It is also a possibility to add or remove log record *filters* at runtime to decrease the amount of output by ignoring all messages that do not match one or more filter criteria.

1.3.4 Multi threading

Multi threading needs to be considered to allow multiple threads within one process to access the logging system in parallel. Multi threading can be an issue for both log dispatching and for sending messages to the output devices. Receiving messages via a socket connection needs to be multi threaded anyway since each client requires a connection for its own which can only be handled by separate threads[11].

1.3.5 Location independence

For distributed applications it may be a requirement to store all log records in a central data storage like a database or a SAN[12]. This means that the logging is performed on each client separately but collected on a central server. That requires the logging system to send and

[10]A NullAppender is a dummy appender that acts like a normal appender but does not persist the message on any device.

[11]Because the network operation accepting a connection is normally a *blocking* operation which means that the function only returns when an incoming connection is detected or a timeout occurs. This is normally implemented by an extra thread waiting for incoming connections and per-client worker threads.

[12]Storage Area Network

receive messages via a network to the central storage. Therefore a network protocol needs to be defined and in case the logged data contains sensitive information the data needs to be encrypted.

2 The generic *phloc-logging* system

In this section, the implementation of the logging system called *phloc-logging* is described and the implementation issues are discussed. It contains as well a quick start guide for a programmer to simplify the process of getting started. The logging system is published as Open Source under the Apache License 2.0 and can be publicly downloaded from the SourceForge web site[13] or via anonymous CVS access. Please refer to the SourceForge notes on how to do an anonymous CVS checkout.

2.1 Overview

phloc-logging is designed to be reliable, fast, extensible and easy to use. It consists of three main parts: *input*, a *dispatcher* and *output devices*. The logging system is implemented in Java and was successfully tested with the Sun J2SE 1.4.2 and higher.

2.2 Input

The input consists of a list that contains all values required to create a log message. There are two types of input: *static input* and *dynamic input*. Static input is defined as input provided by the user of the logging system (THE programmer) and is known at development time. It is one of the following: the message (including an optional additional information), the log level (the priority) of the log message and the logger object used to issue the message. Dynamic input in contrast can be any data that is not explicitly provided by the programmer like e.g. the date and time when the message was issued, the name of the computer where the message was issued or the name of the thread issuing the message. When sending a log message, the programmer does not need to know anything about the dynamic input since they are automatically handled by the logging system, he only needs to care about the static input. Each part of input is described by a so called *log context* and is further detailed in chapter 2.2.2.

The input for a single log message is generalized by the `ILogMessage` interface that simply encapsulates the object to be logged (in the method `getMessageObject`) and the optional additional information object (in the method `getAdditionalInformation`).

2.2.1 Log level

The log level designates the severity of a log message. The higher a log level of a message the more severe is a message. The log level is internally represented by instances of class `LogLevel` that is a wrapper around a numeric priority level. This class defines a set of predefined log levels (as static class constants) that should match most real-life requirements. The numeric values and the alias names are closely related to the log level definitions of java.util.logging. The following set of predefined log levels are available, presented in ascending priority:

[13] Get it at http://www.sourceforge.net/projects/phloc-logging

- `LogLevel.ALL` is the very lowest log level and normally not used when issuing a log message. Its numeric representation is -2^{31}.
- `LogLevel.TRACE` is the lowest log level to be used for highly detailed tracing messages. Tracing messages should be used to log an application flow but not for indicating errors and the like. Its numeric representation is 300.
- `LogLevel.FINEST` is an alias for `LogLevel.TRACE`.
- `LogLevel.DEBUG` should be used for fairly detailed tracing messages. Its numeric representation is 400.
- `LogLevel.FINER` is an alias for `LogLevel.DEBUG`.
- `LogLevel.NOTE` should be used for information that will be broadly interesting to developers who do not have a specialized interest in the specific subsystem. Its numeric representation is 500.
- `LogLevel.FINE` is an alias for `LogLevel.NOTE`.
- `LogLevel.CONFIG` is a message level for static configuration messages. They are intended to provide a variety of static configuration information, to assist in debugging problems that may be associated with particular configurations. Its numeric representation is 700.
- `LogLevel.INFORMATION` is a message level for informational messages. Typically these messages will be written to the console or its equivalent. So this level should only be used for reasonably significant messages that will make sense to end users and system administrators. Its numeric representation is 800.
- `LogLevel.INFO` is an alias for `LogLevel.INFORMATION`.
- `LogLevel.WARNING` is a message level indicating a potential problem. In general these messages should describe events that will be of interest to end users or system managers, or which indicate potential problems. Its numeric representation is 900.
- `LogLevel.ERROR` is a message level indicating a serious failure. In general these messages should describe events that are of considerable importance and which will prevent normal program execution. They should be reasonably intelligible to end users and to system administrators. Its numeric representation is 1000.
- `LogLevel.SEVERE` is an alias for `LogLevel.ERROR`.
- `LogLevel.FATAL` is a message level indicating a failure that prevents the application from further execution. Its numeric representation is 1100.
- `LogLevel.OFF` is a special level that can be used to turn off logging. Its numeric representation is 2^{31}-1.

2.2.2 Log contexts

Each log context is represented internally as a class implementing the `ILogContext` interface. Static contexts are implementing the `ILogContextStatic` interface whereas dynamic contexts are implementing the `ILogContextDynamic` interface that also extends the `ILogFormatable` interface[14]. Both specific interfaces inherit declarations from the `ILogContext` interface.

Each log context is identified by a unique ID which is a numeric value different from 0. By convention dynamic contexts have a positive integer value as ID and static contexts have a negative number as ID. The IDs of the dynamic contexts are dynamically created at run-time by incrementing a counter by one. By changing the initialization order of the dynamic contexts (e.g. in the configuration file) the generated IDs are changed as well. The IDs of the static contexts are hard-coded and cannot be altered. They have the same ID no matter how many dynamic contexts are present. Logging contexts can also be enabled or disabled. Disabled contexts are not taken into consideration when a log message is issued and therefore the current values of the context are not stored in the `LogRecord` object (see chapter 2.2.3). Contexts can be enabled or disabled at runtime which affects all log messages from that time on.

The main functionality of a log context is to deliver a value. Therefore the two methods `getContextValue` and `getContextValueClass` exist. `getContextValue` needs to deliver the current value whereas `getContextValueClass` needs to deliver the type (=the Java `Class` object) of the object returned by `getContextValue`. Each context value needs to implement the `java.lang.Comparable` and `java.io.Serializable` interfaces so that log records can be sorted according to a certain log context and serialized to a stream[15].

Imagine the example of a dynamic log context that delivers the name of the computer on which logging takes place as its context value[16]. `getContextValue` returns a `String` object with the value ”WS001” because the computer has the name ”WS001”. `getContext-ValueClass` returns `java.lang.String.class` since the value returned by `getContext-Value` is of type `java.lang.String`.

Log contexts are managed by the class `LogContextContainer`. This class maps the ID of a log context to the underlying `ILogContext` object. To access the global `LogContextContainer` instance on which all loggers are based via the `getGlobal` method is used. More details about the global log context container can be found in chapter 2.6.1 about loggers. If a log context is added to a log context container it is checked whether the value class of this log context implements both the `java.lang.Comparable` and the `java.io.Serializable` interface to avoid further incompatibilities.

It is possible to create new dynamic contexts by implementing the `ILogContextDynamic` interface. To simplify the process an abstract helper class `AbstractLogContextDynamic` exists that implements all the functionality required to create a dynamic log context except for the `getContextValueClass` and `getContextValue` methods which is left for the real implementation. The constructor of class `AbstractLogContextDynamic` requires the

[14]The interface `ILogFormatable` is used to format arbitrary objects when representing them as a string.
[15]Standard Java classes like `String`, `Integer` or `Date` all implement these interfaces.
[16]An implementation of this dynamic log context can be found in the class `com.phloc.commons.log.ctx.-LogContextDynamicHostname`

following four parameters[17]:

- The unique ID of the log context.
- The boolean state whether the log context should be enabled or disabled by default.
- The display name of the log context.
- The formatting string that should be used to format the value according to the required formatter. Never pass `null` here.

2.2.3 Logging record

A log record is internally represented as an instance of class `LogRecord`. It consists of the following parts:

- The log level specified when the message was issued,
- a list of `LogRecordField` objects containing references to the log contexts and their current values and
- the optional additional information object passed by the programmer.

Instances of class `LogRecord` are passed to the output handler for further processing. The main content is the list of `LogRecordField` objects that contain references to the original log context objects and their values at the time the `LogRecordField` object was instantiated. Via the `LogRecordField.getAsString` method the context values are converted into a string object and optionally formatted, if a formatting object is present. Whether a value is formatted or not depends on the implementation of the `ILogFormatable` interface within the log context.

2.3 Dispatching

After a log record has been successfully created it needs to be dispatched to the output devices that are interested in this log record. Therefore the log message that is received by a single `ILogger` object is forwarded to a central dispatcher of type `ILogDispatcher`. This interface is implemented in the classes `com.phloc.commons.log.dispatch.LogDispatcherSynchronous` for synchronous dispatching and `com.phloc.commons.log.dispatch.LogDispatcherAsynchronous` for asynchronous dispatching. Both implementations are singletons and there is no need to access them from outside the logging system. Their sole purpose is the forwarding of a message to the registered output handlers. The log dispatcher is just an in-process dispatcher and cannot be used to dispatch messages between different processes or even between different hosts.

The input parameters for the dispatching is the source `ILogger` object (optional), the desired log level (mandatory) and the message itself (mandatory). If no `ILogger` object is passed, the default logger with the name "global" is used. Based on the used logger the output handlers to which the message should be forwarded are determined. Now the message is forwarded to each of these output handlers if they are enabled and if the selected log level of

[17]Another constructor exists that automatically generates an ID and therefore has only three parameters.

the output handler matches[18] the passed log level of the message. For the first output handler that matches the aforementioned criteria the `LogRecord` instance is build from the input parameters and that is the point where the log context values are determined. If no output handler matches the criteria no `LogRecord` is ever build due to performance considerations meaning also that no context values are determined. Before the main forwarding takes place it is checked, whether the output handler has filters defined that may prevent the record from being forwarded. The filter can check any element of the `LogRecord` object for potential filter criteria. If no filters are defined or if the record matches the filter criteria the record is forwarded to the output handler. All of this happens synchronously so the calling thread has to wait until the logging routine is finished. The log dispatcher is instantiated within the `LogFactory` class and this happens only once before the first message is being dispatched.

As an alternative to the synchronous log dispatcher an asynchronous version can be selected. The asynchronous message dispatcher is located in the class `com.phloc.commons.log.-LogDispatcherAsynchronous`. The asynchronous version pools all log requests in a queue and dispatches them in a separate thread. The goal of the asynchronous log dispatching is to speed up the processing of the main application flow by spending less time in the log processing thread. To activate the asynchronous dispatcher, the field `com.phloc.commons.log.-LogFactory.USE_SYNCHRONOUS_DISPATCHER` has to be set to `false`. This needs to happen before any logging takes place because the dispatcher is only instantiated upon the first request.

2.4 Output handler

The real actions are taken by so called *output handlers*. An output handler forwards the created `LogRecord` instance to an output device. An output device can be something like a file, a console, a database or a network connection. Each output handler has a its own log level to further detail the logging possibilities of the system and accept only messages with a certain base priority. Output handlers can be disabled so that they receive no log messages at all. This can happen at run-time and affects all forthcoming messages.

Each output handler is based on the interface `ILogOutputHandler` that describes the basic functionality like the log level handling and the enabling and disabling as well as the forwarding of a record to the output device. No default output handler exists because no default output device exists. *phloc-logging* comes with a set of predefined output handlers to log to the following devices:

- Standard error console.

  The implementation resides in the class `com.phloc.commons.log.hdl.LogOutputHandlerCon` and has no specific properties.

- Standard output console.

  The implementation resides in the class `com.phloc.commons.log.hdl.LogOutputHandlerCon` and has no specific properties.

- Standard console.

[18] Log level matching means, that the log level of the output handler is $\leq$ the log level of the message.

The implementation resides in the class `com.phloc.commons.log.hdl.LogOutputHandlerConsol` and has no specific properties. The difference to `com.phloc.commons.log.hdl.-LogOutputHandlerConsoleStderr` and `com.phloc.commons.log.hdl.-LogOutputHandlerConsoleStdout` is that depending on the passed log level, the output is written to `stdout` or to `stderr`.

- Flat file.

  The implementation resides in the class `com.phloc.commons.log.hdl.LogOutput-HandlerFlatFile` and can be parameterized as follows:

  - Should new records be appended to an existing file or should old records be overwritten?
  - A filename prefix that can include a path and a base name of the file to be used.
  - A filename suffix that normally contains the filename extension. The default suffix is ".log".
  - The maximum size of the file to be created. This is used to avoid the infinite growth of the log file. If this value is set to 0 it means that the file can grow infinite. The value for an infinite size is encapsulated in the constant `INDEFINITE_SIZE`. The maximum file size needs to be at least 1024 bytes.
  - The number of files to rotate. Rotation takes place if the maximum file size is reached or if a new logging session starts and appending is disabled. This values needs to be ≥ 1.

- XML file.

  The implementation resides in the class `com.phloc.commons.log.hdl.LogOutput-HandlerXMLFile` and can be parameterized as follows:

  - The absolute filename to be used.

  It is not possible to set all the properties of the flat file because the XML file contains a special header that contains information about the used log contexts and since the context configuration may be different each time the logging system is started the header needs to be written anyway to ensure data consistency.

  Implementation note: currently the XML file handler does not emit valid XML since it emits no XML header and no single root tag. To use the output of the XML output handler the created contents need to be imported in an existing XML file. This is an issue that needs to be solved in the next version. Another problem is that the change of log contexts does not force the XML file handler to re-emit the contexts but only uses the new context list. This is also an issue for the next version.

- In-memory list of log records.

  The implementation resides in the class `com.phloc.commons.log.hdl.LogOutput-HandlerMemory` and can be parameterized as follows:

  - The maximum number of entries to be stored. Once that limit is reached, the oldest entry is discarded before the new entry is added. This is useful to avoid out-of-memory situations if logging is heavily used. By setting the maximum number of entries to `INDEFINITE_ENTRIES` (numerically represented by 0) no storage limits are applied but be careful having too many items in memory since this may lead to an out of memory situation. This output handler also implements the `ILogOutputHandlerModifyable` interface that is an interface that allows modifications on the contained records.

- Network socket.

  The network socket output handler is used to send full log records over a network connection. The transmission includes the full list of log context values and not only the assembled string so that structured storage on the server side is accomplished. The network socket output handler should only be used if an appropriate message server is running somewhere else that accepts the incoming transmissions. The implementation resides in the class `com.phloc.commons.log.hdl.LogOutputHandlerSocket` and can be parameterized as follows:

  - The host name to which the log records should be send to.
  - The port to which the log records should be send to on the destination host. The value needs to be between `LogSettings.NETWORK_PORT_MIN` (=1024) and `LogSettings.NETWORK_PORT_MAX` (=65535) since on Unix operating systems the first 1024 ports (0-1023) are reserved for internal use. The default port is 5657 represented by the constant field `LogSettings.NETWORK_PORT_DEFAULT`.

  phloc-logging ships with an example implementation of the server side part that can be found in the class `com.phloc.commons.log.server.LogServerImpl`. Please note that this implementation is not tested for robustness and may need some adoptions to be used in real-world code.

- Adapters to third-party logging systems.

  As specified in chapter 2.5 *phloc-logging* ships with adapters to third-party logging systems. The following list contains the output handlers that are used to forward log records from *phloc-logging* to the other logging system. None of these output handlers has special properties to be set. All implementation classes can be found in the package `com.phloc.commons.log.adapter`.

  - `AdapterToJavaUtilsLogging` is used to forward messages to the java.util.logging framework that is shipped with J2SE since version 1.4.
  - `AdapterToApacheCommons` is used to forward messages to the Apache commons logging library[19].
  - `AdapterToLog4J` is used to forward messages to the Log4J logging system[20].

The globally available list of output handlers is managed within the class `com.phloc.commons.log.LogOutputHandlerContainer`. It allows to add and remove output handlers based on an internal list data structure. It is possible to add the output handler more than once which results in double forwarding to this output handler so be careful when building the list of output handlers.

[19]More information to be found at http://jakarta.apache.org/commons/logging/
[20]More information to be found at http://logging.apache.org/log4j/

All of these output handlers are ready to be used in your applications and new output handlers can easily be created. Creating a new output handler can be simplified by extending from the abstract class `com.phloc.commons.log.hdl.AbstractLogOutputHandler`. This class encapsulates the log level handling, the state handling and the filter management. Everything except the forwarding to the real output device is handled within the abstract class. The only methods that necessarily need to be implemented are `outputRecord` for forwarding a record to an output device and `clone` for creating a deep-copy of the output handler.

2.5 Adapters to other logging systems

Most applications ship with a set of third-party components that incorporate their own logging mechanisms which cannot easily be altered. Therefore *phloc-logging* contains adapters to other frequently used logging systems: java.util.logging, Apache Commons Logging and Log4J. All adapters are bidirectional that means that log messages can be sent to and received from the other logging system. Adapters to send messages to other logging systems are realized as output handlers (see chapter 2.4). There is no generic way to write adapters to receive messages from other logging systems. Normally this is done by implementing a specific interface of the other logging system.

The implementation classes reside in the package `com.phloc.commons.log.adapter` and contain all the necessary code to use. Please note that you still need the additional JAR files that are required to run the other logging system.

2.6 Usage

Now that all the internals are explained the following sections deals with the usage of the logging system. The two important parts to use the logging system are the logger pool and the single logger itself.

2.6.1 ILogger

A logger is the basic entity that forwards log messages to the dispatcher. It is used by the programmer to issue new log messages with a certain log level and an input message. A logger is of interface type `com.phloc.commons.log.ILogger` and only instantiated from within the class `LoggerPool`. A logger has the following properties:

- A read-only name.
- An optional minimum log level.
- An optional list of log contexts that apply to this logger. If no such list is specified, the logger operates on the global list of log contexts.
- An optional list of output handlers this logger wants to log to. If no such list is specified, the logger operates on the global list of output handlers.

The name is used to uniquely identify the object from within the pool. The purpose of the log level is to allow certain loggers to ignore messages below a certain priority independently of the registered output handlers. If no special log level is specified, a logger takes all messages. The

Method name	Logging level
trace()	LogLevel.TRACE
debug()	LogLevel.DEBUG
note()	LogLevel.NOTE
config()	LogLevel.CONFIG
info()	LogLevel.INFORMATION
warning()	LogLevel.WARNING
error()	LogLevel.ERROR
fatalError()	LogLevel.FATAL

Table 1: Shorthand methods for specific log levels

optional lists of log contexts and output handlers are used to specify log contexts and output handlers that should be present only within this logger. If no special log contexts or output handlers are specified, the globally defined elements are used. The class `ILogger` cannot be directly instantiated. Only the class `LoggerPool` instantiates `ILogger` objects to minimize the amount of instantiated objects and therefore save memory.

To issue a new log message the `ILogger` interface offers several possibilities. The most generic way to issue a new message is by invoking the `log` method that takes two or three parameters depending on the amount of input data present. The parameters are separated into two types: the log level and the data to issue. The log level always needs to be specified whereas the data can come in different flavors. The easiest way to log an object is to simply pass it as second parameter to the `log` method with or without an object that encapsulates the optional additional information object. The third version of the `log` method takes an object of type `ILogMessage` as its second parameter which is a wrapper around a data object and the additional information. For sanity reason an `ILogger` object provides shorthand methods that encapsulate a certain log level. So instead of writing `logger.log (LogLevel.DEBUG, "foo")` it is possible to use the shorthand method `logger.debug ("foo")`. The following table summarizes all available shorthand methods:

A special logger type is the hierarchical logger that resides in the interface `com.phloc.commons.log.IHierarchicalLogger`. It is derived from interface `ILogger` and used to build a hierarchy of `ILogger` objects that have shared properties. E.g. if a hierarchical logger has no log level specified, the parent logger is queried. If the parent logger has no log level the parent of the parent needs to be queried etc. The same applies to log contexts and output handlers. If a new hierarchical logger is instantiated it inherits all the properties from its parent logger (except the name). If the new logger does not have a parent logger (a so called ”root logger”) the global list of log contexts and output handlers is used as fallback. If a log context or an output handler is added to or removed from an existing logger this change is also recursively performed in all child logger objects. The differentiation of loggers and hierarchical loggers is to separate the handling of real logging provided by the class `ILogger` and the sanity inheritance functionality provided by the hierarchical logger.

2.6.2 Logger pool

The purpose of the logger pool is to instantiate `ILogger` or `IHierarchicalLogger` objects. This pool is helpful in minimizing the number of logger objects existing at one moment because only references to already existing `ILogger` objects are returned instead of creating new ones upon each request. This behavior also minimizes the amount of main memory required by the logging system due to the minimization of objects. That's why the constructor of class `ILogger` cannot be instantiated from outside the owning Java package.

The logger pools also keeps track of the hierarchy of loggers. The hierarchy of loggers is build in the same way as Java manages the package hierarchy except that the name of the logger objects are used and not package names. Each dot-separated token of a logger's name forms a hierarchy level where the root level is indicated by the first token on the left. When for example querying the logger with the name ”global.ui.menu” it is first checked whether a root logger with the name ”global” exists. If not it is created, otherwise the existing logger is reused. Afterwards the ”global”-logger is queried for a child-logger with the name ”ui” and so forth. The returned `ILogger` object has the name ”global.ui.menu” and a parent logger with the name ”global.ui” which in turn is a child of the root logger ”global”.

The logger pool resides in the class `com.phloc.commons.log.LoggerPool` and contains only static methods. The method `getLogger` is used to access the predefined global logger with the name ”global”. The other important method is `getLogger (String)` where the parameter designates the fully qualified name of the logger to be retrieved. `getIterator` is only used for diagnostic enumeration of the logger hierarchy. To ensure that the logging system is shut down gracefully the `shutdown` method is provided that closes the complete hierarchy of loggers and invalidates the hierarchy. After a call to `shutdown` no further log processing is possible without new initialization.

2.7 Configuration

The logging system needs to be initialized before it can be used. Initialization means configuring the available log contexts and the output handlers. There are two different ways to initialize the logging system: the first is to manually instantiate the log contexts and add them to the global log context container as well as creating the output handlers manually and inserting them into the global output handler container. The other way is to use an XML based configuration file that handles both global log contexts and global output handlers with their respective properties. The XML file can be read with any `InputStream` and does not necessarily need to reside on the local hard disk even though this is the recommended place.

In case you don't want to use a configuration file the class `DefaultInitializer` contains the code to setup a basic set of log contexts and output handlers. The method `createDefaultContexts` is used to initialize the container of global log contexts and `createDefaultOutputHandler` is used to create output handlers to forward messages to standard output console and to a flat file. The preferred way to initialize *phloc-logging* is the XML configuration file.

The configuration via XML allows the initialization of both static and dynamic log contexts as well as output handlers. The XML file needs to have the following layout: The name of the top-level element is not yet of importance but it's recommended to call it ”logging” in case it becomes relevant for future versions. All child elements of the document element are evaluated in document order.

Source type	corresponding context
loglevel	the log level specified
message	the message object specified
name	the name of the logger issuing a log record

Table 2: Source types and their corresponding static contexts

For using a dynamic log context the element name `dyncontext` is used. It has two mandatory attributes: `class` that contains the fully qualified name of the Java class to instantiate and `enabled` that is used to enable or disable the context by default. The class to instantiate needs to implement the `ILogContextDynamic` interface to be used by *phloc-logging* . The log context is instantiated dynamically via the reflection feature of Java. The only requirement for a dynamic log context to be instantiated is the presence of a constructor that takes a single `Boolean` argument to indicate whether the log context is initially enabled or not.

Listing 1: Example for using a dynamic log context

```
<dyncontext class="com.phloc.commons.log.ctx.↩
   LogContextDynamicTotalSequence" enabled="false" />
```

Static contexts are instantiated similar to dynamic contexts except that there is no class name to select but a source type (attribute `srctype`) since all static log contexts are implemented by the same class `com.phloc.commons.log.ctx.LogContextStatic`. The source type needs be one of the following elements:

Listing 2: Example for using a static log context

```
<staticcontext srctype="loglevel" enabled="true" />
```

Output handlers can be specified via the element name `handler`. They are instantiated like dynamic log contexts via a fully qualified class name specified in the `class` attribute. The class to instantiate needs to implement the `ILogOutputHandler` interface to be used by *phloc-logging* and a constructor without parameters is required. An output handler is always enabled and cannot be disabled via the configuration file. The child elements of an output handler definition are used to dynamically set properties. Internally the name of the element is converted to a method name by prefixing the element name with ”set” and uppercasing the first character of the element name. The method is dynamically invoked on the created output handler and passes the text-content of the element as a `String` parameter. So for example the tag `logLevel` is translated into the method name `setLogLevel` and in the following example invoked with the parameter ”Trace”.

Listing 3: Example for using an output handler and dynamically applying properties

```
<handler class="com.phloc.commons.log.hdl.↩
   LogOutputHandlerFlatFile">
  <logLevel>Trace</logLevel>
  <prefix>myApplicationLog</prefix>
  <maxBytesPerFile>1000000</maxBytesPerFile>
  <maxFiles>5</maxFiles>
  <append>true</append>
</handler>
```

The following listing shows the XML representation of the default configuration that is build into *phloc-logging* :

Listing 4: Default XML configuration file

```
<?xml version="1.0" encoding="iso-8859-1"?>
<logging>
  <!-- Log contexts; order is important -->
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicTotalSequence" enabled="true" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicCurThread" enabled="false" />
  <staticcontext srctype="loglevel" enabled="true" />
  <staticcontext srctype="name" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicOSUsername" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicHostname" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicDate" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicTime" enabled="true" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicDateTime" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourcePackage" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourceClass" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourceMethod" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourceFile" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourceLine" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourceFull" enabled="false" />
  <dyncontext class="com.phloc.commons.log.ctx.↩
     LogContextDynamicSourceShort" enabled="true" />
  <staticcontext srctype="message" enabled="true" />

  <!-- output handler; order is recommended -->
  <handler class="com.phloc.commons.log.hdl.↩
     LogOutputHandlerConsole">
```

```
    <logLevel>Debug</logLevel>
  </handler>
  <handler class="com.phloc.commons.log.hdl.↩
      LogOutputHandlerFlatFile">
    <logLevel>Debug</logLevel>
    <prefix>phloc-logging-default</prefix>
    <!-- 0 means infinite size: -->
    <maxBytesPerFile>0</maxBytesPerFile>
    <maxFiles>1</maxFiles>
    <append>true</append>
  </handler>
</logging>
```

3 Quick start guide

This section should give a brief overview how the logging system can be used out of the box. It is assumed that no adapters to other logging systems are required (see chapter 2.5) and that the JAR file containing *phloc-logging* is in your class path.

The following imports are necessary to perform the actions specified in the upcoming example code.

Listing 5: Imports for the logging system

```
import com.phloc.commons.log.*;
```

Before using the logging system it needs to be initialized. By default the *phloc-logging* system tries to look for the file `phloc-logging-config.xml` within the class path. If multiple files with this name exist within the class path the decision is up the JVM. The name of the configuration file can be overridden by a system property called `phloc.logging.configfile`. Example usage from the command line:
`java -Dphloc.logging.configfile= /home/user/logging-config.xml`
Anyway it is preferred to use the default configuration file to decrease coupling between the logging system and the calling application.

Now it's time to send the first log message to the system. Therefore an `ILogger` object is acquired from the `LoggerPool`. Afterwards two requests are send to the logger. One that manually passes the log level as a parameter and the other one that uses the shorthand method `ILogger.debug`.

Listing 6: Using the logging system

```
// get logger object
final ILogger aLogger = LoggerPool.getLogger ("foo");

// Send message with explicit log level "info".
aLogger.log (LogLevel.INFO, "Info␣Message");

// Send message with implicit log level "debug".
```

```
aLogger.debug ("Debug␣Message");

// Send message with implicit log level "error" and an ↩
   additional exception.
// The stacktrace of the exception will be emitted.
try {
  String s = null;
  // cannot work:
  s.split ("x");
} catch (NullPointerException ex) {
  aLogger.error ("An␣error␣occurred␣splitting␣the␣string", ex↩
     );
}
```

If no configuration file is provided, the created output looks like this:

Listing 7: Logger output on the console

```
! Config: Found no configuration file phloc-logging-config.↩
   xml - initializing with defaults
! Config: Initializing default log contexts
! Config: Initializing default output handler with level ↩
   Debug
[1] Info 11:54:39,500 PhlocLoggingTest.main() Info Message
[2] Debug 11:54:39,515 PhlocLoggingTest.main() Debug Message
[3] Error 11:54:39,515 PhlocLoggingTest.main() An error ↩
   occurred splitting the string
java.lang.NullPointerException
1.: PhlocLoggingTest.main(PhlocLoggingTest.java:17)
```

4 Future work

Since no software is ever finished this also applies to *phloc-logging* . The code is quite stable and is used in production in several products. By releasing the code to Open Source I hope the code is tested heavily and maybe further adapters, output handler and contexts are created. Even though *phloc-logging* contains many adapters to other logging systems it may be required to add adapters to other logging systems. Another open point is the creation of unit tests for the logging framework to ensure that code changes do not effect other parts of the system and to maintain integrity. There are already some JUnit 4 unit tests but the test code coverage is very pure and needs to be extended to further improve code quality. There is also a list of known limitations to *phloc-logging* that I plan to resolve in future versions. One point is the extended network logging support and to add some more output handlers.

— Mach's gut, und danke für den Fisch. —

References

[1] Beatrice Weiler and Edgar Nett A Generic Log-Service - The key architectural element to support efficient recovery from node crashes.

[2] Michel Ruffin A Survey of Logging Uses.

[3] Log4J project http://logging.apache.org/log4j/

[4] Sun Java http://java.sun.com