

Arjuna CLF 2.0

Programmer's Guide

CLF-PG-9/17/08





Legal Notices

The information contained in this documentation is subject to change without notice.

Arjuna Technologies Limited makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Arjuna Technologies Limited shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Arjuna is a trademark of Hewlett-Packard Company.

Software Version

Arjuna CLF 2.0

Restricted Rights Legend

Use, duplication, or disclosure is subject to restrictions as set forth in contract subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause 52.227-FAR14.

Arjuna Technologies Limited
Nanotechnology Centre
Herschel Building
Newcastle Upon Tyne
NE1 7RU
United Kingdom

© Copyright 2008 Arjuna Technologies Limited

Content

Table Of Contents

About This Guide.....	4	Getting Started.....	8
What This Guide Contains.....	4	Log Interface.....	10
Audience.....	4	Dependencies.....	11
Organization.....	4	Default File Logging.....	12
Documentation Conventions.....	4	Overview.....	12
Overview.....	6	Setup.....	12
CLF 2.0 Architecture.....	6	Fine-Grained Logging.....	13
Package Overview:		Overview.....	13
com.arjuna.common.util.logging	6	Usage.....	14
LogFactory.....	7	Index.....	16
Setup of Log subsystem.....	7		

About This Guide

What This Guide Contains

The Programmer's Guide contains information on how to use Arjuna CLF 2.0.

Audience

This guide is most relevant to engineers who are responsible for using Arjuna CLF 2.0 installations.

Organization

This guide contains the following chapters:

1. **Chapter 1, Overview**
2. **Chapter 2, Migration to CLF 2.0**
3. **Chapter 3, Helper Classes**
4. **Chapter 4, The Log Interface**

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
<i>Italic</i>	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
Code	Text that represents programming code.
Function Function	A path to a function or dialog box within an interface. For example, " Select File Open. " indicates that you should select the Open function from the File menu.
() and	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the

	<p>following three items can be entered in this syntax:</p> <p><code>persistPolicy (Never OnTimer OnUpdate NoMoreOftenThan)</code></p>
Note: and	A note highlights important supplemental information.
Caution:	A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

The diagram illustrates the CLF 2.0 architecture. At the top is a box labeled "CLF 2.0". Below it, on the left, is a box labeled "Jakarta Commons Logging". To its right is a box labeled "Core Services Framework (CSF) Logging". Below "Jakarta Commons Logging" are four boxes: "log4j", "JDK 1.4", "JDK 1.1", and "Avalon". Below "Core Services Framework (CSF) Logging" is a box labeled "Pure JDK 1.1 logging (for compilation to MS .net)". Below these are two rows of boxes representing loggers and appenders. The first row contains "File (rolling)", "JDBC", "Tcp/ip Socket", "Win NT syslog", and "JMX". The second row contains "console", "JMS", "Chain saw", "Chain saw", and "XML Log". Dashed lines show the flow: from "log4j" to "File (rolling)", "JDBC", "Tcp/ip Socket", and "Win NT syslog"; from "JDK 1.4" to "JDBC"; from "JDK 1.1" to "Win NT syslog"; from "Avalon" to "Win NT syslog"; and from "console" to "Console-Appender can also write to CSF log viewer".

```

graph TD
    CLF20[CLF 2.0]
    JCL[Jakarta Commons Logging]
    CSFL[Core Services Framework (CSF) Logging]
    log4j[log4j]
    JDK14[JDK 1.4]
    JDK11[JDK 1.1]
    Avalon[Avalon]
    PureJDK11[Pure JDK 1.1 logging (for compilation to MS .net)]
    File[File (rolling)]
    JDBC[JDBC]
    TcpSocket[Tcp/ip Socket]
    WinNT[Win NT syslog]
    JMX[JMX]
    console[console]
    JMS[JMS]
    ChainSaw1[Chain saw]
    ChainSaw2[Chain saw]
    XMLLog[XML Log]

    JCL --- log4j
    JCL --- JDK14
    JCL --- JDK11
    JCL --- Avalon
    CSFL --- PureJDK11

    log4j -.-> File
    log4j -.-> JDBC
    log4j -.-> TcpSocket
    log4j -.-> WinNT
    JDK14 -.-> JDBC
    JDK11 -.-> WinNT
    Avalon -.-> WinNT
    console -.-> ConsoleAppender[Console-Appender can also write to CSF log viewer]
  
```

Package Overview: com.arjuna.common.util.logging

Logi18n	A simple logging interface abstracting the various logging APIs supported by CLF and providing an internationalization layer based on resource bundles.
LogNoi18n	A simple logging interface abstracting the various logging APIs supported by CLF without internationalization support

CommonDebugLevel	The CommonDebugLevel class provides default finer debugging value to determine if finer debugging is allowed or not.
CommonFacilityCode	The CommonFacilityCode class provides default finer facilitycode

	value to determine if finer debugging is allowed or not.
CommonVisibilityLevel	The CommonVisibilityLevel class provides default finer visibility value to determine if finer debugging is allowed or not.
LogFactory	Factory for Log objects.

LogFactory

Factory for Log objects. LogFactory returns different subclasses of logger according to which logging subsystem is chosen. The log system is selected through the property `com.arjuna.common.utils.logger`. Supported log systems are:

- **jakarta** Jakarta Commons Logging (JCL). JCL can delegate to various other logging subsystems, such as:
 - log4j
 - JDK 1.4 logging
 - JDK 1.1 based logging (for compilation to Microsoft .net)
 - Avalon
- **dotnet** .net logging. (must be JDK 1.1 compliant for compilation by the Microsoft compiler)

Note that rather than implementing CSF and .net logging as additional loggers for JCL they have been anchored at this level to maximise code reuse and guarantee that all .net dependent code is 1.1 compliant. Log subsystems are not configured through CLF but instead rely on their own configuration files for the setup of eg. debug level, appenders, etc...

Setup of Log subsystem

The underlying log system can be selected in two ways:

- Through the `commonPropertyManager`:
`com.arjuna.common.internal.util.logging.commonPropertyManager.
propertyManager.setProperty("com.arjuna.common.util.logger", "csf");`
- As a System property (see following table)

Property Name	Description
<code>com.arjuna.common.util.logger</code>	This property selects the log subsystem to use. Note that this can only be set as a System property, e.g. as a parameter to start up the client application:
property name is defined as the public constant: <code>LogFactory.LOGGER_PROPERTY</code>	<code>java -Dcom.arjuna.common.util.logger=log4j ..</code>

Table 2 System property to select the underlying log system to use.

Note: The properties of the underlying log system are configured in a manner specific to that log system, e.g., a `log4j.properties` file in the case that log4j logging is used.

Property Value	Description
----------------	-------------

log4j	Log4j logging (log4j classes must be available in the classpath); configuration through the log4j.properties file, which is picked up from the CLASSPATH or given through a System property: log4j.configuration
jdk14	JDK 1.4 logging API (only supported on JVMs of version 1.4 or higher). Configuration is done through a file logging.properties in the jre/lib directory.
simple	Selects the simple JDK 1.1 compatible console-based logger provided by Jakarta Commons Logging
jakarta	Uses the default log system selection algorithm of the Jakarta Commons Logging framework
dotnet	Selects a .net logging implementation Since a dotnet logger is not currently implemented, this is currently identical to simple. Simple is a purely JDK1.1 console-based log implementation.
noop	Disables all logging

Table 3 Possible values for selecting the client-side logging system.

Example: To set off log4j (default log system), provide the following System properties:

```
-Dcom.arjuna.common.util.logger=log4j
-Dlog4j.configuration=file:///c:/Projects/common/log4j.properties
```

Getting Started

Simple use example:

```
import com.arjuna.common.util.logging.*;

public class Test
{
    static Log mylog = LogFactory.getLog(Test.class);

    public static void main(String[] args)
    {
        String param0 = "foo";
        String param1 = "bar";

        // different log priorities
        mylog.debug("key1", new Object[]{param0, param1});
        mylog.info("key2", new Object[]{param0, param1});
        mylog.warn("key3", new Object[]{param0, param1});
        mylog.error("key4", new Object[]{param0, param1});
        mylog.fatal("key5", new Object[]{param0, param1});

        // optional throwable
        Throwable throwable = new Throwable();
        mylog.debug("key1", new Object[]{param0, param1}, throwable);
        mylog.info("key2", new Object[]{param0, param1}, throwable);
        mylog.warn("key3", new Object[]{param0, param1}, throwable);
        mylog.error("key4", new Object[]{param0, param1}, throwable);
        mylog.fatal("key5", new Object[]{param0, param1}, throwable);

        // debug guard to avoid an expensive operation if the logger does not
        // log at the given level:
        if (mylog.isDebugEnabled())
        {
```



```

        String x = expensiveOperation();
        mylog.debug("key6", new Object[]{x});
    }

    // *****
    // fine-grained debug extensions
    mylog.debug(CommonDebugLevel.OPERATORS,
        CommonVisibilityLevel.VIS_PUBLIC,
        CommonFacilityCode.FAC_ALL,
        "This debug message is enabled since it matches default" +
        "Finer Values");

    mylog.setVisibilityLevel(CommonVisibilityLevel.VIS_PACKAGE);
    mylog.setDebugLevel(CommonDebugLevel.CONSTRUCT_AND_DESTRUCT);
    mylog.setFacilityCode(CommonFacilityCode.FAC_ALL);

    mylog.mergeDebugLevel(CommonDebugLevel.ERROR_MESSAGES);

    if (mylog.debugAllowed(CommonDebugLevel.OPERATORS,
        CommonVisibilityLevel.VIS_PUBLIC,
        CommonFacilityCode.FAC_ALL))
    {
        mylog.debug(CommonDebugLevel.OPERATORS,
            CommonVisibilityLevel.VIS_PUBLIC,
            CommonFacilityCode.FAC_ALL,
            "key7", new Object[]{"foo", "bar"}, throwable);
    }
}

```

Log Interface

A simple logging interface abstracting the various logging APIs supported by CLF.

The logging levels used by Log are (in order):

1. debug (the least serious)
2. info
3. warn
4. error
5. fatal (the most serious)

The mapping of these log levels to the concepts used by the underlying logging system is implementation dependent. The implementation should ensure, though, that this ordering behaves as expected.

Performance is often a logging concern. By examining the appropriate property, a component can avoid expensive operations (producing information to be logged).

For example,

```

if (log.isDebugEnabled()) {
    ... do something expensive ...
    log.debug(...);
}

```

Configuration of the underlying logging system will generally be done external to the Logging APIs, through whatever mechanism is supported by that system.

Dependencies

Name	Description
commons-logging.jar	Jakarta Commons Logging JAR (v. 1.0.3)
log4j-1.2.8.jar	Log4j Jar file (required when using log4j)
mw-common.jar	(for CSF logging)
csf.jar	(for CSF logging)

Table 4 Jar file dependencies

Note: At runtime, it is important, that `log4j-1.2.8.jar` appears after `common.jar` in the `CLASSPATH`. The reason is the CLF overrides a class in `log4j` that is required to print out correct line number information in the log.

Default File Logging

Overview

Independent of the log system chosen, it is possible to log all messages over a given severity threshold into a file. This is useful to guarantee that e.g., error and fatal level messages are not lost despite a user has not set up a log framework, such as log4j

Setup

Usage of this feature is simple and can be controlled through a set of properties. These can be provided through the Property Manager or as System properties.

Property Name	Values	Description
<code>com.arjuna.common.logging.default</code>	true/false	Enable/disable default file-based logging
<code>com.arjuna.common.util.logging.default.level</code>	Info/error/fatal	Severity level for this log
<code>com.arjuna.common.util.logging.default.showLogName</code>	true/false	Record the fully qualified log name
<code>com.arjuna.common.util.logging.default.showShortLogName</code>	true/false	Record an abbreviated log name
<code>com.arjuna.common.util.logging.default.showDate</code>	true/false	Record the date
<code>com.arjuna.common.util.logging.default.logFile</code>	error.log (default)	File to use for default logging. This can be an absolute filename or relative to the working directory
<code>com.arjuna.common.util.logging.default.logFileAppend</code>	true/false	Append to the log file above in case that this file already exists

Table 5 Properties to control default file-based logging (default values are highlighted)

Fine-Grained Logging

Overview

Finer-grained logging in CLF is available through a set of debug methods:

```
public void debug(long dl, long vl, long fl, Object message);
public void debug(long dl, long vl, long fl, Throwable throwable);
public void debug(long dl, long vl, long fl, String key, Object[] params);
public void debug(long dl, long vl, long fl, String key, Object[] params,
                  Throwable throwable);
```

All of these methods take the three following parameters in addition to the log messages and possible exception:

dl - The **debug finer level** associated with the log message. That is, the logger object allows to log only if the DEBUG level is allowed and dl is either equals or greater the debug level assigned to the logger Object See Table 6 for possible values.

vl - The **visibility level** associated with the log message. That is, the logger object allows to log only if the DEBUG level is allowed and vl is either equals or greater the visibility level assigned to the logger Object See Table 8 for possible values.

fl - The **facility code level** associated with the log message. That is, the logger object allows to log only if the DEBUG level is allowed and fl is either equals or greater the facility code level assigned to the logger Object See Table 7 for possible values.

The debug message is sent to the output only if the specified debug level, visibility level, and facility code match those allowed by the logger.

Note: The first two methods above do not use i18n. i.e., the messages are directly used for log output.

Usage

Possible values for debug finer level, visibility level and facility code level are declared in the classes `DebugLevel`, `VisibilityLevel` and `FacilityCode` respectively. This is useful for programmatically using fine-grained debugging.

Debug Finer Level	Value	Description
NO_DEBUGGING	0x0000	no debugging
CONSTRUCTORS	0x0001	only output from constructors
DESTRUCTORS	0x0002	only output from finalizers
CONSTRUCT_AND_DESTRUCT	CONSTRUCTORS DESTRUCTORS	
FUNCTIONS	0x0010	only output from methods
OPERATORS	0x0020	only output from methods such as equals, notEquals
FUNCS_AND_OPS	FUNCTIONS OPERATORS	
ALL_NON_TRIVIAL	CONSTRUCT_AND_DESTRUCT FUNCTIONS OPERATORS	
TRIVIAL_FUNCS	0x0100	only output from trivial methods
TRIVIAL_OPERATORS	0x0200	only output from trivial operators
ALL_TRIVIAL	TRIVIAL_FUNCS TRIVIAL_OPERATORS	
ERROR_MESSAGES	0x0400	only output from debugging error/warning messages
FULL_DEBUGGING	0xffff	output all debugging messages

Table 6 Possible settings for finer debug level (class `DebugLevel`)

Visibility Level	Value	Description
VIS_NONE	0x0000	no visibility
VIS_PRIVATE	0x0001	only from private methods
VIS_PROTECTED	0x0002	only from protected methods
VIS_PUBLIC	0x0004	only from public methods
VIS_PACKAGE	0x0008	only from package methods
VIS_ALL	0xffff	output all visibility levels

Table 7 Possible settings for visibility level (class `VisibilityLevel`)

Facility Code Level	Value	Description
FAC_NONE	0x00000000	no facility
FAC_ALL	0xffffffff	output all facility codes

Table 8 Possible settings for facility code level (class `FacilityCode`)

At runtime, the fine-grained debug settings are controlled through a set of properties, listed in the table below:

Property Name	Default Value
<code>com.arjuna.common.util.logging.DebugLevel</code>	NO_DEBUGGING
<code>com.arjuna.common.util.logging.VisibilityLevel</code>	VIS_ALL
<code>com.arjuna.common.util.logging.FacilityCode</code>	FAC_ALL

Index