Arjuna CLF 2.0

Programmer's Guide

CLF-PG-11/10/09

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

Content

Table Of Contents

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
Italic	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

	following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
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

Overview

CLF 2.0 Architecture

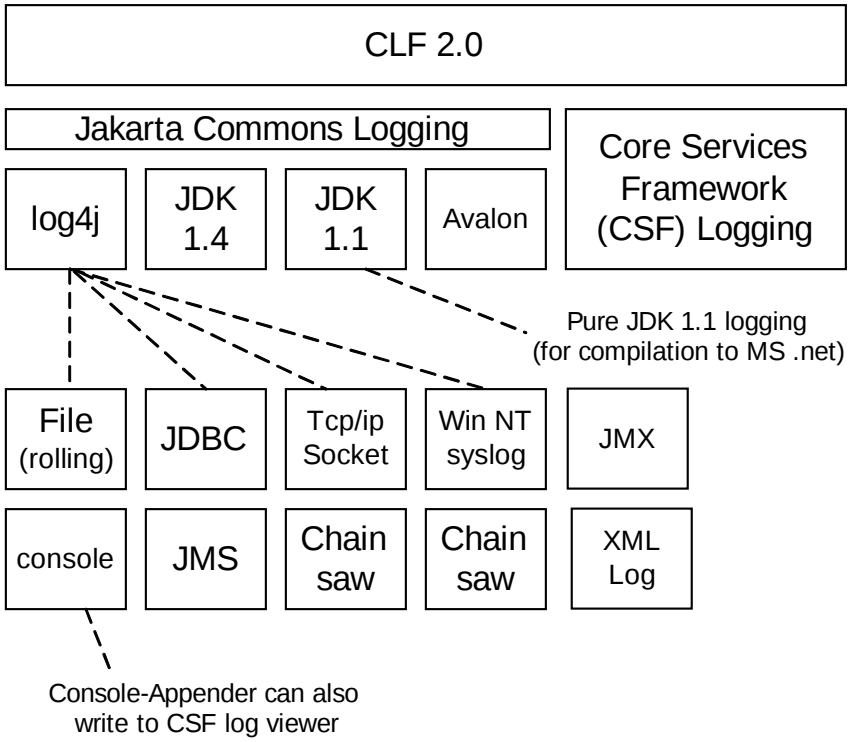

Figure 0-1 CLF 2.0 Architecture

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

Interface Summary

`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

Class Summary

DebugLevel	**The DebugLevel class provides default finer debugging value to determine if finer debugging is allowed or not.**
	The FacilityCode class provides default finer facilitycode value to

FacilityCode	determine if finer debugging is allowed or not.
VisibilityLevel	The VisibilityLevel 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 `LoggingEnvironmentBean.loggingFactory`. Supported log systems are:

- **`jakarta`** Jakarta Commons Logging (JCL). JCL can delegate to various other logging subsystems, such as:
  - log4j
  - JDK 1.4 logging

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: commonPropertyManager.getLoggingEnvironmentBean.setLoggingFactory(value);
- As a System property (deprecated) (see following table)

Property Name	Description
`LoggingEnvironmentBean.loggingFactory`	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: `java –DLoggingEnvironmentBean.loggingFactory=com.arjuna ..`

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.

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

```
java –DLoggingEnvironmentBean.loggingFactory=”com.arjuna.common.internal.util.logging.jakarta.JakartaLogFactory;com.arjuna.common.internal.util.
```

```
logging.jakarta.Log4JLogger”
-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(DebugLevel.OPERATORS,
                  VisibilityLevel.VIS_PUBLIC,
                  FacilityCode.FAC_ALL,
                  "This debug message is enabled since it matches default” +
                  Finer Values");

      mylog.setVisibilityLevel(VisibilityLevel.VIS_PACKAGE);
      mylog.setDebugLevel(DebugLevel.CONSTRUCT_AND_DESTRUCT);
      mylog.setFacilityCode(FacilityCode.FAC_ALL);

      mylog.mergeDebugLevel(DebugLevel.ERROR_MESSAGES);

      if (mylog.debugAllowed(DebugLevel.OPERATORS,
                             VisibilityLevel.VIS_PUBLIC,
                             FacilityCode.FAC_ALL))
      {
         mylog.debug(DebugLevel.OPERATORS,
                     VisibilityLevel.VIS_PUBLIC,
                     FacilityCode.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 implemention 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-1.1.jar`	Jakarta Commons Logging JAR
`log4j-1.2.14.jar`	Log4j Jar file (required when using log4j)

Table 3 Jar file dependencies

Basic File Logging

Overview

Where it is undesirable to have 3rd party dependencies, a simple file based logger may be used.

Setup

Usage of this feature is simple and can be controlled through a set of properties on the BasicLogEnvironmentBean instance obtained via commonPropertyManager, but can also be set using the system properties below.

Property Name	Values	Description
`BasicLogEnvironmentBean.level`	**Info**/error/fatal	Severity level for this log
`BasicLogEnvironmentBean.showLogName`	true/**false**	Record the fully qualified log name
`BasicLogEnvironmentBean.showShortLogName`	**true**/false	Record an abbreviated log name
`BasicLogEnvironmentBean.showDate`	**true**/false	Record the date
`BasicLogEnvironmentBean.logFile`	**error.log** (default)	File to use for default logging. This can be an absolute filename or relative to the working directory
`BasicLogEnvironmentBean.logFileAppend`	**true**/false	Append to the log file above in case that this file already exists

Table 4 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 5 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 7 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 6 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 5 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 visbility levels

Table 6 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 7 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
`com.arjuna.common.util.logging.DebugLevel`	**`NO_DEBUGGING`**
`com.arjuna.common.util.logging.VisibilityLevel`	**`VIS_ALL`**
`com.arjuna.common.util.logging.FacilityCode`	**`FAC_ALL`**

Index