

## **Oracle® Fusion Middleware**

Using Logging Services for Application Logging for Oracle  
WebLogic Server

12c Release 1 (12.1.1)

**E24419-02**

January 2012

Documentation for developers that describes how to use WebLogic Server logging services to monitor application events. It describes WebLogic support for internationalization and localization of log messages, and shows you how to use the templates and tools provided with WebLogic Server to create or edit message catalogs that are locale-specific.

Oracle Fusion Middleware Using Logging Services for Application Logging for Oracle WebLogic Server, 12c Release 1 (12.1.1)

E24419-02

Copyright © 2007, 2012, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

---

---

# Contents

<b>Preface</b> .....	vii
Documentation Accessibility .....	vii
Conventions .....	vii
<b>1 Introduction and Roadmap</b>	
1.1 Document Scope and Audience.....	1-1
1.2 Guide to This Document.....	1-1
1.3 Related Documentation.....	1-2
1.4 Logging Samples and Tutorials .....	1-2
1.4.1 Avitek Medical Records Application (MedRec) and Tutorials.....	1-2
1.4.2 Logging Examples in the WebLogic Server Distribution .....	1-3
1.4.3 Internationalizing Applications Using Simple Message Catalogs Example.....	1-3
1.5 New and Changed Features in This Release.....	1-3
<b>2 Application Logging and WebLogic Logging Services</b>	
2.1 About WebLogic Logging Services .....	2-1
2.2 Integrating Application Logging with WebLogic Logging Services: Main Steps.....	2-2
2.3 Accessing the WebLogic Server Logger .....	2-2
<b>3 Internationalization and Localization for WebLogic Server</b>	
3.1 About Internationalization and Localization Standards.....	3-1
3.2 Understanding Internationalization and Localization for WebLogic Server.....	3-1
3.3 Understanding Message Catalogs.....	3-2
3.4 Understanding Java Interfaces for Internationalization.....	3-2
3.5 Main Steps for Creating an Internationalized Message .....	3-3
<b>4 Using Message Catalogs with WebLogic Server</b>	
4.1 Overview of Message Catalogs.....	4-1
4.2 Message Catalog Hierarchy.....	4-2
4.3 Guidelines for Naming Message Catalogs.....	4-2
4.4 Using Message Arguments.....	4-3
4.5 Message Catalog Formats .....	4-4
4.5.1 Example Log Message Catalog.....	4-4
4.5.2 Elements of a Log Message Catalog.....	4-5

4.5.2.1	message_catalog Element.....	4-5
4.5.2.2	log_message Element.....	4-7
4.5.2.3	Child Elements of log_message Element .....	4-8
4.5.3	Example Simple Text Catalog .....	4-9
4.5.4	Elements of a Simple Text Catalog.....	4-9
4.5.4.1	message_catalog Element.....	4-9
4.5.4.2	message Element .....	4-10
4.5.4.3	messagebody Element .....	4-11
4.5.5	Example Locale-Specific Catalog.....	4-11
4.5.6	Elements of a Locale-Specific Catalog .....	4-12
4.5.6.1	locale_message_catalog Element.....	4-12
4.5.6.2	log_message Element.....	4-12
4.5.6.3	Other locale_message_catalog Elements.....	4-12

## 5 Writing Messages to the WebLogic Server Log

5.1	Using the I18N Message Catalog Framework: Main Steps.....	5-1
5.1.1	Create Message Catalogs .....	5-1
5.1.2	Compile Message Catalogs .....	5-2
5.1.3	Use Messages from Compiled Message Catalogs.....	5-5
5.2	Using the NonCatalogLogger APIs.....	5-6
5.3	Using ServletContext.....	5-8
5.4	Configuring Servlet and Resource Adapter Logging .....	5-8
5.5	Writing Messages from a Client Application .....	5-10
5.6	Writing Debug Messages.....	5-10

## 6 Using the WebLogic Server Message Editor

6.1	About the Message Editor .....	6-1
6.2	Starting the Message Editor.....	6-2
6.3	Working with Catalogs .....	6-3
6.3.1	Browsing to an Existing Catalog .....	6-3
6.3.2	Creating a New Catalog.....	6-6
6.4	Adding Messages to Catalogs.....	6-7
6.4.1	Entering a New Log Message .....	6-7
6.4.2	Entering a New Simple Text Message .....	6-9
6.5	Finding Messages.....	6-10
6.5.1	Finding a Log Message .....	6-10
6.5.2	Finding a Simple Text Message .....	6-10
6.6	Using the Message Viewer .....	6-11
6.6.1	Viewing All Messages in a Catalog.....	6-11
6.6.2	Viewing All Messages in Several Catalogs .....	6-12
6.6.3	Selecting a Message to Edit from the Message Viewer .....	6-12
6.7	Editing an Existing Message .....	6-13
6.8	Retiring and Unretiring Messages.....	6-13

## 7 Using the WebLogic Server Internationalization Utilities

7.1	WebLogic Server Internationalization Utilities .....	7-1
-----	--	-----

7.2	WebLogic Server Internationalization and Localization.....	7-2
7.3	weblogic.i18ngen Utility .....	7-3
7.3.1	Syntax .....	7-3
7.3.2	Options .....	7-3
7.4	weblogic.i10ngen Utility .....	7-4
7.4.1	Syntax .....	7-4
7.4.2	Options .....	7-4
7.4.3	Message Catalog Localization.....	7-5
7.4.4	Examples .....	7-5
7.5	weblogic.GetMessage Utility.....	7-6
7.5.1	Syntax .....	7-6
7.5.2	Options .....	7-6

## **A Localizer Class Reference for WebLogic Server**

A.1	About Localizer Classes .....	A-1
A.2	Localizer Methods.....	A-1
A.3	Localizer Lookup Class .....	A-2

## **B Loggable Object Reference for WebLogic Server**

B.1	About Loggable Objects.....	B-1
B.2	How To Use Loggable Objects.....	B-1

## **C TextFormatter Class Reference for WebLogic Server**

C.1	About TextFormatter Classes.....	C-1
C.2	Example of an Application Using a TextFormatter Class.....	C-1

## **D Logger Class Reference for WebLogic Server**

D.1	About Logger Classes.....	D-1
D.2	Example of a Generated Logger Class.....	D-1



---

---

# Preface

This preface describes the document accessibility features and conventions used in this guide—*Using Oracle WebLogic Logging Services for Application Logging*.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

### Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

## Conventions

The following text conventions are used in this document:

Convention	Meaning
<b>boldface</b>	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.





---

---

# Introduction and Roadmap

This document describes how you use WebLogic Server logging services to monitor application events. It describes WebLogic support for internationalization and localization of log messages, and shows you how to use the templates and tools provided with WebLogic Server to create or edit message catalogs that are locale-specific.

This section describes the contents and organization of this guide—*Using Oracle WebLogic Logging Services for Application Logging*.

- [Document Scope and Audience](#)
- [Guide to This Document](#)
- [Related Documentation](#)
- [Logging Samples and Tutorials](#)
- [New and Changed Features in This Release](#)

## 1.1 Document Scope and Audience

This document is a resource for Java Platform, Enterprise Edition (Java EE) application developers who want to use WebLogic message catalogs and logging services as a way for their applications to produce log messages and want to integrate their application logs with WebLogic Server logs. This document is relevant to all phases of a software project, from development through test and production phases.

This document does not address how you configure logging, subscribe to and filter log messages. For links to information on these topics, see [Section 1.3, "Related Documentation."](#)

It is assumed that the reader is familiar with Java EE and Web technologies, object-oriented programming techniques, and the Java programming language.

## 1.2 Guide to This Document

The document is organized as follows:

- [Chapter 1, "Introduction and Roadmap,"](#) describes the scope of this guide and lists related documentation.
- [Chapter 2, "Application Logging and WebLogic Logging Services,"](#) discusses how to use WebLogic logging services to monitor application events.
- [Chapter 3, "Internationalization and Localization for WebLogic Server,"](#) summarizes the processes required for internationalization and localization.

- [Chapter 4, "Using Message Catalogs with WebLogic Server,"](#) describes message catalog types, message definitions, elements, and arguments.
- [Chapter 5, "Writing Messages to the WebLogic Server Log,"](#) describes how to create and use message catalogs and how to use the `NonCatalogLogger` class to write log messages.
- [Chapter 6, "Using the WebLogic Server Message Editor,"](#) explains how to use the Message Editor that is included with WebLogic Server.
- [Chapter 7, "Using the WebLogic Server Internationalization Utilities,"](#) explains how to use the internationalization utilities included with WebLogic Server.
- [Appendix A, "Localizer Class Reference for WebLogic Server,"](#) describes Localizer classes, Localizer methods, key values for Localizers, and lookup properties for Localizers.
- [Appendix B, "Loggable Object Reference for WebLogic Server,"](#) describes loggable objects and how they are used.
- [Appendix C, "TextFormatter Class Reference for WebLogic Server,"](#) provides an example of an application that uses a TextFormatter class.
- [Appendix D, "Logger Class Reference for WebLogic Server,"](#) describes Logger classes and provides an example of a message catalog and its corresponding Logger class.

## 1.3 Related Documentation

The corporate Web site provides all documentation for WebLogic Server. Specifically, "View and configure logs" in the *Oracle WebLogic Server Administration Console Online Help* describes configuring log files and log messages that a WebLogic Server instance generates, and *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server* describes configuring WebLogic Server to write messages to log files, filtering message output, and listening for the log messages that WebLogic Server broadcasts.

For general information about internationalization and localization, refer to the following sources:

- The Java Developer Connection at <http://www.oracle.com/technetwork/java/index.html>
- The Internationalization section of the World Wide Web Consortium (W3C) Web Site at <http://www.w3.org>

## 1.4 Logging Samples and Tutorials

In addition to this document, we provide a variety of logging code samples and tutorials that show logging configuration and API use.

### 1.4.1 Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other

platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Server.

## 1.4.2 Logging Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in `WL_HOME\samples\server\examples\src\examples`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server Start menu.

## 1.4.3 Internationalizing Applications Using Simple Message Catalogs Example

This example shows various methods for displaying localized text using simple message catalogs. Using any of the languages supported by the example requires the appropriate operating system localization software and character encoding. The package that contains this example is:

```
java examples.i18n.simple>HelloWorld [lang [country]]
```

where `lang` is a two-character ISO language code (for example, `en` for English) and `country` is a two-character ISO country code (for example, `US` for the United States).

The files are located in `WL_HOME\samples\server\examples\src\examples\i18n\simple`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. The default path is `c:\Oracle\Middleware\wlserver_12.1`, however, you are not required to install this directory in the Oracle Middleware Home directory, `MW_HOME`.

## 1.5 New and Changed Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.



---

---

# Application Logging and WebLogic Logging Services

This chapter describes how to use WebLogic logging services for your application logging. You can use WebLogic logging services to keep a record of which user invokes specific application components, to report error conditions, or to help debug your application before releasing it to a production environment.

This chapter includes the following sections:

- [About WebLogic Logging Services](#)
- [Integrating Application Logging with WebLogic Logging Services: Main Steps](#)
- [Accessing the WebLogic Server Logger](#)

## 2.1 About WebLogic Logging Services

WebLogic logging services provide information about server and application events. Your application can also use WebLogic logging services to communicate its status and respond to specific events. See "Understanding WebLogic Logging Services" in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

Two features of WebLogic logging services from which your application can benefit are its extensibility and support for internationalization.

You can create your own catalog of log messages and use WebLogic utilities to generate Java classes that you can use in your application code. The log messages generated from your applications will be integrated with and treated in the same way as log messages generated by the server. See [Chapter 5, "Writing Messages to the WebLogic Server Log."](#)

Log message catalogs you create can be written in any language and can be accompanied by translations for different locales. WebLogic support for internationalization ensures that the log messages are present in the appropriate language for the current locale under which WebLogic Server is running. See [Chapter 3, "Internationalization and Localization for WebLogic Server."](#)

A major advantage of integrating your application logging with WebLogic logging framework is ease of management. The Administration Console lets you manage all of the log files and related options. See "View and configure logs" in the *Oracle WebLogic Server Administration Console Online Help*.

## 2.2 Integrating Application Logging with WebLogic Logging Services: Main Steps

There are several ways to generate log messages from your applications and integrate them with WebLogic logging:

- Use WebLogic tools to build custom log message catalogs and their associated Java APIs. Applications can invoke the log methods exposed by these interfaces to generate log messages. The message catalogs can be easily internationalized. See [Chapter 4, "Using Message Catalogs with WebLogic Server."](#)
- Use the WebLogic non-catalog logger to generate log messages. With `NonCatalogLogger`, instead of calling messages from a catalog, you place the message text directly in your application code. See [Section 5.2, "Using the NonCatalogLogger APIs."](#)
- Use a `log()` method available to servlets and JSPs in `javax.servlet.ServletContext`. See [Section 5.3, "Using ServletContext."](#)

Application developers who do not use WebLogic message catalogs, `NonCatalogLogger`, or servlet logging can do the following:

- Use the Java Logging APIs to produce and distribute messages.
- Use Log4J to produce messages and configure the server to use Log4J or the default, Java Logging, to distribute messages.
- Use the Commons API to produce messages.

For more information, see `org.apache.commons.logging` at <http://jakarta.apache.org/commons/logging/api/index.html>.

- Use the Server Logging Bridge handler (for Java Logging), or appender (for Log4J), which redirects application log messages to WebLogic logging services.

## 2.3 Accessing the WebLogic Server Logger

The WebLogic logging infrastructure supports a logger on each server that collects the log events generated by your own applications and subsystems. WebLogic Server provides direct access to the logger on each server, as well as to the domain logger on the Administration server.

By default, WebLogic logging services use an implementation based on the Java Logging APIs. The `LoggingHelper` class provides access to the `java.util.logging.Logger` object used for server logging. See the `LoggingHelper` Javadoc.

Alternatively, you can direct WebLogic logging services to use Log4j instead of Java Logging. When Log4j is enabled, you get a reference to the `org.apache.log4j.Logger` that the server is using from the `weblogic.logging.log4j.Log4jLoggingHelper` class. With a `Log4jLogger` reference, you can attach your own custom appender (handler) to receive the log events or you can use the `Logger` reference to issue log requests to WebLogic logging services. See the `Log4jLoggingHelper` Javadoc.

In addition, WebLogic logging services provide an implementation of the Jakarta Commons `LogFactory` and `Log` interface, so you can program to the Commons API and direct log messages to the server log file or any of the registered destinations. This API provides you with an abstraction that insulates you from the underlying logging implementation, which could be Log4j or Java Logging.

Applications that do not currently use WebLogic logging services can use the Server Logging Bridge. The Server Logging Bridge provides a lightweight means for applications to have their messages redirected to WebLogic logging services without reconfiguring the application or making any code changes, such as using the WebLogic Logging APIs.

The Server Logging Bridge is available as follows:

- For applications that use Java Logging, the Server Logging Bridge is made available as a `java.util.logging.Handler` object that can be specified in a `logging.properties` file that is passed in an argument in the server's `weblogic.Server` startup command.
- For applications that use Log4J Logging, the Server Logging Bridge is made available as an `org.apache.log4j.Appender` object, which can be specified in a `log4j.properties` file that is included in the application classpath.

The logging properties file can also specify how the message severity level can be propagated to WebLogic logging services, as well as the specific destination to which messages should be published.

For more information, including examples, see the following sections in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*:

- "How to Use Log4j with WebLogic Logging Services"
- "How to Use the Commons API with WebLogic Logging Services"
- "Best Practices: Integrating Java Logging or Log4j with WebLogic Logging Services"
- "Server Logging Bridge"





---

---

# Internationalization and Localization for WebLogic Server

This chapter describes the processes required for internationalization and localization.

This chapter includes the following sections:

- [About Internationalization and Localization Standards](#)
- [Understanding Internationalization and Localization for WebLogic Server](#)
- [Understanding Message Catalogs](#)
- [Understanding Java Interfaces for Internationalization](#)
- [Main Steps for Creating an Internationalized Message](#)

## 3.1 About Internationalization and Localization Standards

Oracle has adopted the World Wide Web Consortium's (W3C) recommendations for standard formats and protocols that are usable worldwide in all languages and in all writing systems. These standards are part of the Java internationalization APIs that are used by WebLogic Server. Internationalization (I18N) refers to the process of designing software so that it can be adapted to various languages and regions easily, cost-effectively, and, in particular, without engineering changes to the software. Localization (L10N) is the use of locale-specific language and constructs at run time.

## 3.2 Understanding Internationalization and Localization for WebLogic Server

Localization covers not only language, but collation, date and time formats, monetary formats, and character encoding. Messages that are logged to the WebLogic server log can be localized to meet your particular requirements.

WebLogic Server internationalization supports localization of two types of data:

- **Log messages** - Log messages are informational messages that are written to the server log, and may also contain error messages if the appropriate message arguments are included in the message definition. See [Section 4.5.2, "Elements of a Log Message Catalog."](#)
- **Simple text** - Simple text is any text other than log messages and exceptions that the server must display, such as the output from a utility. Examples of simple text include usage messages, graphical user interface (GUI) labels, and error messages. See [Section 4.5.4, "Elements of a Simple Text Catalog."](#)

### 3.3 Understanding Message Catalogs

All internationalized text is defined in message catalogs, each of which defines a collection of log messages or simple text. Log messages contain data that is written to the log file. This data is predominantly dynamic and contains information that is specific to the current state of the application and system. When merged with text in a localized log message catalog, this data results in well-formatted, localized messages that describe the error condition in the language of the user. The output sent to the WebLogic Server Administration Console is simple text. As with log messages, simple text can be merged with dynamic data.

To create an internationalized message, you externalize all message strings in a message catalog so that the strings can be converted to multiple locales without changing or recompiling the code. The application code supplies run-time values to the logging methods. The logging methods merge the code with the message string in the catalog according to the current locale. The application code then prints a localized message in the log files.

There are three types of message catalogs:

- **Log message catalogs** - Collections of log messages. See [Section 4.5.2, "Elements of a Log Message Catalog."](#)
- **Simple text message catalogs** - Collections of simple text messages. See [Section 4.5.4, "Elements of a Simple Text Catalog."](#)
- **Locale message catalogs** - Collections of locale-specific messages corresponding to a top-level log message or simple text catalog. See [Section 4.5.6, "Elements of a Locale-Specific Catalog."](#)

Message IDs in log message catalogs or locale message catalogs are unique across all log message or locale message catalogs. Within the message catalog file, each localized version of the message is assigned a unique message ID and message text specific to the error. Ideally, a message is logged from only one location within the system so that a support team can easily find it. Message IDs in simple text catalogs are unique within each simple text catalog.

See [Chapter 4, "Using Message Catalogs with WebLogic Server,"](#) for more detailed information about message catalogs.

To view the WebLogic Server message catalogs, see *Oracle WebLogic Server Error Message Reference*.

### 3.4 Understanding Java Interfaces for Internationalization

WebLogic Server uses the Java internationalization interfaces to provide internationalization and localization. In addition to understanding how WebLogic Server handles internationalization, you should be familiar with the Java internationalization interfaces and the following classes included in the Java Development Kit (JDK).

Class	Description
<code>java.util.Locale</code>	Represents a specific geographical, political, or cultural region.
<code>java.util.ResourceBundle</code>	Provides containers for locale-specific objects.
<code>java.text.MessageFormat</code>	Produces concatenated messages in a language-neutral way.

## 3.5 Main Steps for Creating an Internationalized Message

The following steps summarize how you create an internationalized message to use with WebLogic Server. Later sections of this guide describe these steps in more detail.

1. Create or edit a top-level log message catalog or simple text message catalog by defining the messages in the catalog.

In addition to message text, include information about the type and placement of any run-time values that the message contains. For details, see [Chapter 6, "Using the WebLogic Server Message Editor."](#)

2. Run `weblogic.i18ngen` to validate the catalog you created or edited in Step 1 and generate run-time classes.

The generated classes contain a method for each message. The class is defined according to information specified in the message catalog entry. The classes include methods for logging or getting message text, depending on the type of catalog. The class name ends with `Logger` or `TextFormatter`. For details, see [Section 7.3, "weblogic.i18ngen Utility."](#)

3. Create locale-specific catalogs as required for the message catalog you created in Step 1. See [Section 4.5.5, "Example Locale-Specific Catalog."](#)
4. Run `weblogic.l10ngen` to process the locale-specific catalogs. For details, see [Section 7.4, "weblogic.l10ngen Utility."](#)
5. Configure your application to use the `Logger` or `TextFormatter` classes you generated in Step 2. When the application logs or returns a message, the message is written using the localized version of the text according to the `Logger` or `TextFormatter` classes used.

For more detailed information, see [Chapter 5, "Writing Messages to the WebLogic Server Log."](#)



---

# Using Message Catalogs with WebLogic Server

This chapter describes message catalogs and how to use them. A message catalog is a single XML file that contains a collection of text messages, with each message indexed by a unique identifier. You compile these XML files into classes that contain methods, which are the objects used to log messages at run time.

This chapter includes the following sections:

- [Overview of Message Catalogs](#)
- [Message Catalog Hierarchy](#)
- [Guidelines for Naming Message Catalogs](#)
- [Using Message Arguments](#)
- [Message Catalog Formats](#)

## 4.1 Overview of Message Catalogs

Message catalogs support multiple locales or languages. For a specific message catalog there is exactly one default version, known as the top-level catalog, which contains the English version of the messages. Then there are corresponding locale-specific catalogs, one for each additional supported locale. You use the `weblogic.i18ngen` utility during the build process to compile the XML message catalog files. For more information, see [Section 7.3, "weblogic.i18ngen Utility"](#).

The top-level catalog (English version) includes all the information necessary to define the message. The locale-specific catalogs contain only the message ID, the date changed, and the translation of the message for the specific locale.

The message catalog files are defined by an XML document type definition (DTD). The DTDs are stored in the `weblogic\msgcat` directory of `WL_HOME\server\lib\weblogic.jar`, where `WL_HOME` represents the top-level installation directory for WebLogic Server. The default path is `c:\Oracle\Middleware\wlserver_12.1`, however, you are not required to install this directory in the Oracle Middleware Home directory, `MW_HOME`.

Two DTDs are included in the WebLogic Server installation:

- `msgcat.dtd` - Describes the syntax of top-level, default catalogs.
- `l10n_msgcat.dtd` - Describes the syntax of locale-specific catalogs.

The `weblogic\msgcat` directory of `WL_HOME\server\lib\weblogic.jar` contains templates that you can use to create top-level and locale-specific message catalogs.

You can create a single log message catalog for all logging requirements, or create smaller catalogs based on a subsystem or Java package. Oracle recommends using multiple subsystem catalogs so you can focus on specific portions of the log during viewing.

For simple text catalogs, we recommend creating a single catalog for each utility being internationalized. You create message catalogs using the Message Editor as described in [Chapter 6, "Using the WebLogic Server Message Editor."](#)

## 4.2 Message Catalog Hierarchy

All messages must be defined in the default, top-level catalog. The WebLogic Server installation includes a collection of sample catalogs in the `SAMPLES_HOME\server\examples\src\examples\i18n\msgcat` directory, where `SAMPLES_HOME` represents the WebLogic Server examples home directory at `WL_HOME\samples`.

Catalogs that provide different localizations of the base catalogs are defined in `msgcat` subdirectories named for the locale (for example, `msgcat/de` for Germany). You might have a top-level catalog named `mycat.xml`, and a German translation of it called `..de/mycat.xml`. Typically the top-level catalog is English. However, English is not required for any catalogs, except for those in the `SAMPLES_HOME\server\examples\src\examples\i18n\msgcat` directory.

Locale designations (for example, `de`) also have a hierarchy as defined in the `java.util.Locale` documentation. A locale can include a language, country, and variant. Language is the most common locale designation. Language can be extended with a country code. For instance, `en\US`, indicates American English. The name of the associated catalog is `..en\US\mycat.xml`. Variants are vendor or browser-specific and are used to introduce minor differences (for example, collation sequences) between two or more locales defined by either language or country.

## 4.3 Guidelines for Naming Message Catalogs

Because the name of a message catalog file (without the `.xml` extension) is used to generate run-time class and property names, you should choose the name carefully.

Follow these guidelines for naming message catalogs:

- Do not choose a message catalog name that conflicts with the names of existing classes in the target package for which you are creating the message catalog.
- The message catalog name should only contain characters that are allowed in class names.
- Follow class naming standards.

For example, the resulting class names for a catalog named `Xyz.xml` are `XyzLogLocalizer` and `XyzLogger`.

The following considerations also apply to message catalog files:

- Message IDs are generally six-character strings with leading zeros. Some interfaces also support integer representations.

---



---

**Note:** This only applies to log message catalogs. Simple text catalogs can have any string value.

---



---

- Java lets you group classes into a collection called a package. A package name should be consistent with the name of the subsystem in which a particular catalog resides.
- The log Localizer "classes" are actually `ResourceBundle` property files.

## 4.4 Using Message Arguments

The message body, message detail, cause, and action sections of a log message can include message arguments, as described by `java.text.MessageFormat`. Make sure your message contents conform to the patterns specified by `java.text.MessageFormat`. Only the message body section in a simple text message can include arguments. Arguments are values that can be dynamically set at run time. These values are passed to routines, such as printing out a message. A message can support up to 10 arguments, numbered 0-9. You can include any subset of these arguments in any text section of the message definition (Message Body, Message Detail, Probable Cause), although the message body must include all of the arguments. You insert message arguments into a message definition during development, and these arguments are replaced by the appropriate message content at run time when the message is logged.

The following excerpt from an XML log message definition shows how you can use message arguments. The argument number must correspond to one of the arguments specified in the `method` attribute. Specifically, `{0}` with the first argument, `{1}` with the second, and so on. In [Example 4-1](#), `{0}` represents the file that cannot be opened, while `{1}` represents the file that will be opened in its place.

### Example 4-1 Example of Message Arguments

```
<messagebody>Unable to open file, {0}. Opening {1}. All arguments must be in
body.</messagebody>
```

```
<messagedetail> File, {0} does not exist. The server will restore the file
contents from {1}, resulting in the use of default values for all future
requests. </messagedetail>
```

```
<cause>The file was deleted</cause>
```

```
<action>If this error repeats then investigate unauthorized access to the
file system.</action>
```

An example of a `method` attribute is as follows:

```
-method="logNoFile(String name, String path)"
```

The message example in [Example 4-1](#) expects two arguments, `{0}` and `{1}`:

- Both are used in the `<messagebody>`
- Both are used in the `<messagedetail>`
- Neither is used in `<cause>` or `<action>`

---

---

**Note:** A message can support up to 10 arguments, numbered 0-9. You can include any subset of these arguments in any text section of the message definition (message detail, cause, action), although the message body must include all of the arguments.

---

---

In addition, the arguments are expected to be strings, or representable as strings. Numeric data is represented as {n,number}. Dates are supported as {n,date}. You must assign a severity level for log messages. Log messages are generated through the generated `Logger` methods, as defined by the `method` attribute.

## 4.5 Message Catalog Formats

The catalog format for top-level and locale-specific catalog files is slightly different. The top-level catalogs define the textual messages for the base locale (by default, this is the English language). Locale-specific catalogs (for example, those translated to Spanish) only provide translations of text defined in the top-level version. Log message catalogs are defined differently from simple text catalogs.

Examples and elements of each type of message catalog are described in the following sections:

- [Example Log Message Catalog](#)
- [Elements of a Log Message Catalog](#)
- [Example Simple Text Catalog](#)
- [Elements of a Simple Text Catalog](#)
- [Example Locale-Specific Catalog](#)
- [Elements of a Locale-Specific Catalog](#)

### 4.5.1 Example Log Message Catalog

The following example shows a log message catalog, `MyUtilLog.xml`, containing one log message. This log message illustrates the usage of the `messagebody`, `messagedetail`, `cause`, and `action` elements.

**Example 4-2 Example of a Log Message Catalog**

```
<?xml version="1.0"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog
  l10n_package="programs.utils"
  i18n_package="programs.utils"
  subsystem="MYUTIL"
  version="1.0"
  baseid="600000"
  endid="600100"
  <log_message
    messageid="600001"
    severity="warning"
    method="logNoAuthorization(String arg0, java.util.Date arg1,
      int arg2)"
    <messagebody>
      Could not open file, {0} on {1,date} after {2,number} attempts.
    </messagebody>
```



```

<messagedetail>
  The configuration for this application will be defaulted to
  factory settings. Custom configuration information resides
  in file, {0}, created on {1,date}, but is not readable.
</messagedetail>
<cause>
  The user is not authorized to use custom configurations. Custom
  configuration information resides in file, {0}, created on
  {1,date}, but is not readable.The attempt has been logged to
  the security log.
</cause>
<action>
  The user needs to gain appropriate authorization or learn to
  live with the default settings.
</action>
</log_message>
</message_catalog>

```

## 4.5.2 Elements of a Log Message Catalog

This section provides reference information for the following elements of a log message catalog:

- [message\\_catalog Element](#)
- [log\\_message Element](#)
- [Child Elements of log\\_message Element](#)

### 4.5.2.1 message\_catalog Element

The `message_catalog` element represents the log message catalog. The following table describes the attributes that you can define for the `message_catalog` element.

Attribute	Default Value	Required/ Optional	Description
<code>i18n_package</code>	<code>weblogic.i18n</code>	Optional	Java package containing generated Logger classes for this catalog. The classes are named after the catalog file name. For example, for a catalog using <code>mycat.xml</code> , a generated Logger class would be called <code>&lt;i18n_package&gt;.mycatLogger.class</code> .  Syntax: standard Java package syntax  Example: <code>i18n_package="programs.utils"</code>
<code>l10n_package</code>	<code>weblogic.i18n</code>	Optional	A Java package containing generated <code>LogLocalizer</code> properties for the catalog. For example, for a catalog called <code>mycat.xml</code> , the following property files would be generated: <code>&lt;l10n_package&gt;.mycatLogLocalizer.properties</code> and <code>&lt;l10n_package&gt;mycatLogLocalizerDetail.properties</code> .  Syntax: standard Java package syntax  Example: <code>l10n_package="programs.utils"</code>

Attribute	Default Value	Required/Optional	Description
subsystem	None	Required	<p>An acronym identifying the subsystem associated with this catalog. The name of the subsystem is included in the server log and is used for message isolation purposes.</p> <p>Syntax: a String</p> <p>Example: subsystem="MYUTIL"</p>
version	None	Required	<p>Specifies the version of the msgcat.dtd being used.</p> <p>Use: Must be "1.0"</p> <p>Syntax: x.y where x and y are numeric.</p> <p>Example: version="1.0"</p>
baseid	<p>000000 for WebLogic Server catalogs</p> <p>500000 for user-defined catalogs</p>	Optional	<p>Specifies the lowest message ID used in this catalog.</p> <p>Syntax: one to six decimal digits.</p> <p>Example: baseid="600000"</p>
endid	<p>499999 for WebLogic Server catalogs</p> <p>999999 for user-defined catalogs</p>	Optional	<p>Specifies the highest message ID used in this catalog.</p> <p>Syntax: one to six decimal digits.</p> <p>Example: endid="600100"</p>
loggable	false	Optional	<p>Indicates whether to generate additional methods that return loggable objects.</p> <p>Syntax: "true" or "false"</p> <p>Example: loggable="true"</p>
prefix	<p>Null for user-defined catalogs</p> <p>"BEA" for WebLogic Server catalogs</p>	Optional	<p>Specifies a String to be prepended to message IDs when logged. Server messages default to "BEA" as the prefix and may not specify a different prefix. User messages can specify any prefix. A prefixed message ID is presented in a log entry as follows:</p> <p>&lt;[prefix-]id&gt;</p> <p>where prefix is this attribute and id is the six-digit message ID associated with a specific message.</p> <p>For example, if prefix is "XYZ", then message 987654 would be shown in a log entry as &lt;XYZ-987654&gt;. If the prefix is not defined, then the log entry would be &lt;987654&gt;.</p> <p>Syntax: any String (should be limited to five characters)</p> <p>Example: prefix="BEA"</p>
description	Null (no description)	Optional	<p>An optional attribute that serves to document the catalog content.</p> <p>Syntax: any String</p> <p>Example: description="Contains messages logged by the foobar application"</p>

### 4.5.2.2 log\_message Element

The following table describes the attributes that you can define for the `log_message` element.

Attribute	Default Value	Required/Optional	Description
<code>messageid</code>	None	Required	<p>Unique identifier for this log message. Uniqueness should extend across all catalogs. Value must be in range defined by <code>baseid</code> and <code>endid</code> attributes.</p> <p>Use: Value must be in the range defined by the <code>baseid</code> and <code>endid</code> attributes defined in the <code>message_catalog</code> attribute.</p> <p>Syntax: one to six decimal digits.</p> <p>Example: <code>messageid="600001"</code></p>
<code>datelastchanged</code>	None	Optional	<p>Date/time stamp used for managing modifications to this message. The date is supplied by utilities that run on the catalogs.</p> <p>The syntax is:</p> <pre>Long.toString(new Date().getTime());</pre> <p>Use: The date is supplied by utilities (such as <code>MessageEditor</code>), that run on the catalogs.</p> <p>Syntax: <code>Long.toString(new Date().getTime());</code></p>
<code>severity</code>	None	Required	<p>Indicates the severity of the log message. Must be one of the following: <code>debug</code>, <code>info</code>, <code>warning</code>, <code>error</code>, <code>notice</code>, <code>critical</code>, <code>alert</code>, or <code>emergency</code>. User-defined catalogs may only use <code>debug</code>, <code>info</code>, <code>warning</code>, and <code>error</code>.</p> <p>Example: <code>severity="warning"</code></p>
<code>method</code>	None	Required	<p>Method signature for logging this message.</p> <p>The syntax is the standard Java method signature, without the qualifiers, semicolon, and extensions. Argument types can be any Java primitive or class. Classes must be fully qualified if not in <code>java.lang</code>. Classes must also conform to <code>java.text.MessageFormat</code> conventions. In general, class arguments should have a useful <code>toString()</code> method.</p> <p>Arguments can be any valid name, but should follow the convention of <code>argn</code> where <code>n</code> is 0 through 9. There can be no more than 10 arguments. For each <code>argn</code> there should be at least one corresponding placeholder in the text elements described in <a href="#">Section 4.5.2.3, "Child Elements of log_message Element."</a> Placeholders are of the form <code>{n}</code>, <code>{n,number}</code> or <code>{n,date}</code>.</p>
<code>methodtype</code>	<code>logger</code>	Optional	<p>Specifies type of method to generate. Methods can be loggers or getters. Logger methods format the message body into the default locale and log the results. Getter methods return the message body prefixed by the subsystem and <code>messageid</code>, as follows: <code>[subsystem:msgid]text</code></p> <p>Syntax: values are "logger" and "getter"</p>

Attribute	Default Value	Required/Optional	Description
stacktrace	true	Optional	Indicates whether to generate a stack trace for Throwable arguments. Possible values are true or false. When the value is true, a trace is generated. Syntax: stacktrace="true"
retired	false	Optional	Indicates whether message is retired. A retired message is one that was used in a previous release but is now obsolete and not used in the current version. Retired messages are not represented in any generated classes or resource bundles. Syntax: values are "true" and "false" Example: retired="true"

### 4.5.2.3 Child Elements of log\_message Element

The following table describes the child elements of the log\_message element.

Element	Parent Element	Required/Optional	Description
messagebody	log_message	Required	<p>A short description for this message.</p> <p>The messagebody element can contain a 0 to 10 placeholder as {n}, to be replaced by the appropriate argument when the log message is localized.</p> <p>The message body must include placeholders for all arguments listed in the corresponding method attribute, unless the last argument is throwable or a subclass.</p> <p>Be careful when using single quotes, because these are specially parsed by java.text.MessageFormat. If it is appropriate to quote a message argument, use double quotes (as in the example below). If a message has one or more placeholders, in order for a single quote to appear correctly (for example, as an apostrophe), it must be followed by a second single quote.</p> <p>Syntax: a String</p> <p>Example:</p> <pre>&lt;messagebody&gt;Could not open file "{0}" created on {1,date}.&lt;/messagebody&gt;</pre>
messagedetail	log_message	Optional	<p>A detailed description of the event. This element may contain any argument place holders.</p> <p>Syntax: a String</p> <p>Example:</p> <pre>&lt;messagedetail&gt;The configuration for this application will be defaulted to factory settings.&lt;/messagedetail&gt;</pre>

Element	Parent Element	Required/Optional	Description
cause	log_message	Optional	The root cause of the problem. This element can contain any argument place holders. Syntax: a String Example: <cause>The user is not authorized to use custom configurations. The attempt has been logged to the security log.</cause>
action	log_message	Optional	The recommended resolution. This element can contain any argument place holders. Syntax: a String Example: <action>The user needs to gain appropriate authorization or learn to live with the default settings.</action>

### 4.5.3 Example Simple Text Catalog

The following example shows a simple text catalog, `MyUtilLabels.xml`, with one simple text definition:

```
<messagebody>
  File
</messagebody>
```

#### **Example 4-3 Example of a Simple Text Catalog**

```
<?xml version="1.0"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
  "http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog>
  l10n_package="programs.utils"
  i18n_package="programs.utils"
  subsystem="MYUTIL"
  version="1.0"
  <message>
    messageid="FileMenuTitle"
    <messagebody>
      File
    </messagebody>
  </message>
</message_catalog>
```

### 4.5.4 Elements of a Simple Text Catalog

This section provides reference information for the following simple text catalog elements:

- [message\\_catalog Element](#)
- [message Element](#)
- [messagebody Element](#)

#### 4.5.4.1 message\_catalog Element

The following table describes the attributes that you can define for the `message_catalog` element.

Attribute	Default Value	Required/Optional	Description
l10n_package	weblogic.i18n	Optional	Java package containing generated TextFormatter classes and TextLocalizer properties for this catalog. The classes are named after the catalog file name. mycat.xml would have the properties file, <l10n_package>.mycatLogLocalizer.properties generated.  Syntax: standard Java package syntax Example: l10n_package="programs.utils"
subsystem	None	Required	An acronym identifying the subsystem associated with this catalog. The name of the subsystem is included in the server log and is used for message isolation purposes.  Syntax: a String Example: subsystem="MYUTIL"
version	None	Required	Specifies the version of the msgcat.dtd being used. The format is n.n, for example, version="1.0". Must be at least "1.0".  Example: version="1.0"
description	Null	Optional	An optional attribute that documents the catalog content.  Syntax: a String Example: description="Contains labels used in the foobar GUI"

#### 4.5.4.2 message Element

The following table describes the attributes that you can define for the message element.

Attribute	Default Value	Required/Optional	Description
messageid	None	Required	Unique identifier for this log message in alpha-numeric string format. Uniqueness is required only within the context of this catalog. message is a child element of message_catalog.
datelastchanged	None	Optional	Date/time stamp useful for managing modifications to this message.

Attribute	Default Value	Required/Optional	Description
method	None	Optional	<p>Method signature for formatting this message.</p> <p>The syntax is a standard Java method signature, less return type, qualifiers, semicolon, and extensions. The return type is always <code>String</code>. Argument types can be any Java primitive or class. Classes must be fully qualified if not in <code>java.lang</code>. Classes must also conform to <code>java.text.MessageFormat</code> conventions. In general, class arguments should have a useful <code>toString()</code> method, and the corresponding <code>MessageFormat</code> placeholders must be strings; they must be of the form <code>{n}</code>. Argument names can be any valid name. There can be no more than 10 arguments.</p> <p>For each argument there must be at least one corresponding placeholder in the <code>messagebody</code> element described below. Placeholders are of the form <code>{n}</code>, <code>{n,number}</code> or <code>{n,date}</code>.</p> <p>Example:</p> <pre>method="getNoAuthorization  (String filename, java.util.Date  creDate) "</pre> <p>This example would result in a method in the <code>TextFormatter</code> class as follows:</p> <pre>public String getNoAuthorization  (String filename, java.util.Date  creDate)</pre>

#### 4.5.4.3 messagebody Element

The following table describes the child element of the message element.

Element	Parent Element	Required/Optional	Description
messagebody	message	Required	<p>The text associated with the message.</p> <p>This element may contain zero or more placeholders <code>{n}</code> that will be replaced by the appropriate arguments when the log message is localized.</p>

### 4.5.5 Example Locale-Specific Catalog

The following example shows a French translation of a message that is available in `... \msgcat\fr\MyUtilLabels.xml`.

The translated message appears as shown in [Example 4-4](#).

#### Example 4-4 Example of a Message Translated to French

```
<?xml version="1.0"?>
<!DOCTYPE message_catalog PUBLIC
 "weblogic-locale-message-catalog-dtd"
 "http://www.bea.com/servers/wls90/dtd/110n_msgcat.dtd">
<locale_message_catalog
 110n_package="programs.utils"
 subsystem="MYUTIL">
```

```

version="1.0">
<message>
  <messageid="FileMenuTitle">
    <messagebody> Fichier </messagebody>
  </message>
</locale_message_catalog>

```

When entering text in the `messagebody`, `messagedetail`, `cause`, and `action` elements, you must use a tool that generates valid Unicode Transformation Format-8 (UTF-8) characters, and have appropriate keyboard mappings installed. UTF-8 is an efficient encoding of Unicode character-strings that optimizes the encoding ASCII characters. Message catalogs always use UTF-8 encoding. The `MessageLocalizer` utility that is installed with WebLogic Server is a tool that can be used to generate valid UTF-8 characters.

## 4.5.6 Elements of a Locale-Specific Catalog

The locale-specific catalogs are subsets of top-level catalogs. They are maintained in subdirectories named for the locales they represent. The elements and attributes described in the following sections are valid for locale-specific catalogs.

- [locale\\_message\\_catalog Element](#)
- [log\\_message Element](#)
- [Other locale\\_message\\_catalog Elements](#)

### 4.5.6.1 locale\_message\_catalog Element

The following table describes the attributes that you can define for the `locale_message_catalog` element.

Attribute	Default Value	Required/Optional	Description
<code>l10n_package</code>	<code>weblogic.i18n</code>	Optional	Java package containing generated <code>LogLocalizer</code> or <code>TextLocalizer</code> properties for this catalog. Properties file are named after the catalog file name.  For example, for a French log message catalog called <code>mycat.xml</code> , a properties file called <code>&lt;l10n_package&gt;.mycatLogLocalizer_fr_FR.properties</code> is generated.
<code>version</code>	None	Required	Specifies the version of the <code>msgcat.dtd</code> being used. The format is <code>n.n</code> , for example, <code>version="1.0"</code> . Must be at least "1.0".

### 4.5.6.2 log\_message Element

The locale-specific catalog uses the attributes defined for the `log_message` element in the top-level log message catalog so this element does not need to be defined.

### 4.5.6.3 Other locale\_message\_catalog Elements

The locale-specific catalog uses the `messagebody`, `messagedetail`, `cause`, and `action` catalog elements defined for the top-level log message catalog so these elements do not need to be defined.



---

# Writing Messages to the WebLogic Server Log

This chapter describes how to facilitate the management of your application by writing log messages to the WebLogic server log file.

This chapter includes the following sections:

- [Using the I18N Message Catalog Framework: Main Steps](#)
- [Using the NonCatalogLogger APIs](#)
- [Using ServletContext](#)
- [Configuring Servlet and Resource Adapter Logging](#)
- [Writing Messages from a Client Application](#)
- [Writing Debug Messages](#)

## 5.1 Using the I18N Message Catalog Framework: Main Steps

The internationalization (I18N) message catalog framework provides a set of utilities and APIs that your application can use to send its own set of messages to the WebLogic server log.

To write log messages using the I18N message catalog framework, complete the following tasks:

- [Create Message Catalogs](#)
- [Compile Message Catalogs](#)
- [Use Messages from Compiled Message Catalogs](#)

### 5.1.1 Create Message Catalogs

A message catalog is an XML file that contains a collection of text messages. Usually, an application uses one message catalog to contain a set of messages in a default language and optionally, additional catalogs to contain messages in other languages.

To create and edit a properly formatted message catalog, use the WebLogic Message Editor utility, which is a graphical user interface (GUI) that is installed with WebLogic Server. To create corresponding messages in local languages, use the Message Localizer, which is also a GUI that WebLogic Server installs.

To access the Message Editor, do the following from a WebLogic Server host:

1. Set the classpath by entering `WL_HOME\server\bin\setWLSEnv.cmd` (`setWLSEnv.sh` on UNIX), where `WL_HOME` is the directory in which you installed WebLogic Server.
2. Enter the following command: `java weblogic.MsgEditor`
3. To create a new catalog, choose File > New Catalog.  
For information on using the Message Editor, see [Chapter 6, "Using the WebLogic Server Message Editor."](#)
4. When you finish adding messages in the Message Editor, select File > Save Catalog.
5. Then select File > Exit.

To access the Message Localizer, do the following from a WebLogic Server host:

1. Set the classpath by entering `WL_HOME\server\bin\setWLSEnv.cmd` (`setWLSEnv.sh` on UNIX), where `WL_HOME` is the directory in which you installed WebLogic Server.
2. Enter the following command: `java weblogic.MsgLocalizer`
3. Use the Message Localizer GUI to create locale-specific catalogs.  
For basic command line help, type: `java weblogic.MsgEditor -help`

## 5.1.2 Compile Message Catalogs

After you create message catalogs, you use the `i18ngen` and `l10ngen` command-line utilities to generate properties files and to generate and compile Java class files. The utilities take the message catalog XML files as input and create compiled Java classes. The Java classes contain methods that correspond to the messages in the XML files.

For more information, see [Chapter 7, "Using the WebLogic Server Internationalization Utilities."](#)

To compile the message catalogs, do the following:

1. From a command prompt, use `WL_HOME\server\bin\setWLSEnv.cmd` (`setWLSEnv.sh` on UNIX) to set the classpath, where `WL_HOME` is the directory in which you installed WebLogic Server.
2. Enter the following command:

```
java weblogic.i18ngen -build -d targetdirectory source-files
```

In the preceding command:

- `targetdirectory` represents the root directory in which you want the `i18ngen` utility to locate the generated and compiled files. The Java files are placed in sub-directories based on the `i18n_package` and `l10n_package` values in the message catalog.

The catalog properties file, `i18n_user.properties`, is placed in the `targetdirectory`. The default target directory is the current directory.

- `source-files` represents the message catalog files that you want to compile. If you specify one or more directory names, `i18ngen` processes all XML files in the listed directories. If you specify file names, the names of all files must include an XML suffix. All XML files must conform to the `msgcat.dtd` syntax.

Note that when the `i18ngen` generates the Java files, it appends `Logger` to the name of each message catalog file.

3. If you created locale-specific catalogs in [Section 5.1.1, "Create Message Catalogs,"](#) do the following to generate properties files:

- a. In the current command prompt, add the `targetdirectory` that you specified in step 2 to the `CLASSPATH` environment variable. To generate locale-specific properties files, all of the classes that the `i18ngen` utility generated must be on the classpath.
- b. Enter the following command:

```
java weblogic.i18ngen -d targetdirectory source-files
```

In the preceding command:

- `targetdirectory` represents the root directory in which you want the `i18ngen` utility to locate the generated properties files. Usually this is the same `targetdirectory` that you specified in step 2. The properties files are placed in sub-directories based on the `i18n_package` values in the message catalog.
  - `source-files` represents the message catalogs for which you want to generate properties files. You must specify top-level catalogs that the Message Editor creates; you do not specify locale-specific catalogs that the Message Localizer creates. Usually this is the same set of `source-files` or source directories that you specified in step 2.
4. In most cases, the recommended practice is to include the message class files and properties files in the same package hierarchy as your application.

However, if you do not include the message classes and properties in the application's package hierarchy, you must make sure the classes are in the application's classpath.

For complete documentation of the `i18ngen` commands, see [Chapter 7, "Using the WebLogic Server Internationalization Utilities."](#)

#### Example: Compiling Message Catalogs

In this example, the Message Editor created a message catalog that contains one message of type `loggable`. The Message Editor saves the message catalog as the following file: `c:\MyMsgCat\MyMessages.xml`.

[Example 5-1](#) shows the contents of the message catalog.

#### Example 5-1 Sample Message Catalog

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog
  i18n_package="com.xyz.msgcat"
  i18n_package="com.xyz.msgcat.i18n"
  subsystem="MyClient"
  version="1.0"
  baseid="700000"
  endid="800000"
  loggables="true"
  prefix="XYZ-"
>
<!-- Welcome message to verify that the class has been invoked-->
```

```
<logmessage
  messageid="700000"
  datelastchanged="1039193709347"
  datehash="-1776477005"
  severity="info"
  method="startup()"
>
<messagebody>
  The class has been invoked.
</messagebody>
<messagedetail>
  Verifies that the class has been invoked
  and is generating log messages
</messagedetail>
<cause>
  Someone has invoked the class in a remote JVM.
</cause>
<action> </action>
</logmessage>
</message_catalog>
```

In addition, the Message Localizer creates a Spanish version of the message in `MyMessages.xml`. The Message Localizer saves the Spanish catalog as `c:\MyMsgCat\es\ES\MyMessages.xml`.

#### **Example 5-2 Locale-Specific Catalog for Spanish**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE locale_message_catalog PUBLIC
"weblogic-locale-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/110n_msgcat.dtd">
<locale_message_catalog
version="1.0"
>
<!-- Mensaje agradable para verificar que se haya invocado la clase. -->
<logmessage
  messageid="700000"
  datelastchanged="1039546411623"
  >
  <messagebody>
    La clase se haya invocado.
  </messagebody>
  <messagedetail>
    Verifica que se haya invocado la clase y está
    generando mensajes del registro.
  </messagedetail>
  <cause>Alguien ha invocado la clase en un JVM alejado.</cause>
  <action> </action>
</logmessage>
</locale_message_catalog>
```

To compile the message catalog that the Message Editor created, enter the following command:

```
java weblogic.i18ngen -build -d c:\MessageOutput
c:\MyMsgCat\MyMessages.xml
```

The `i18ngen` utility creates the following files:

- `c:\MessageOutput\i18n_user.properties`

- `c:\MessageOutput\com\xyz\msgcat\MyMessagesLogger.java`
- `c:\MessageOutput\com\xyz\msgcat\MyMessagesLogger.class`
- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizer.properties`
- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizerDetails.properties`

To create properties files for the Spanish catalog, you do the following:

1. Add the `i18n` classes to the command prompt's classpath by entering the following:
2. `set CLASSPATH=%CLASSPATH%;c:\MessageOutput`
3. Enter
4. `java weblogic.l10ngen -d c:\MessageOutput  
c:\MyMsgCat\MyMessages.xml`

The `l10ngen` utility creates the following files:

- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizer_es_ES.properties`
- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizerDetails_es_ES.properties`

### 5.1.3 Use Messages from Compiled Message Catalogs

The classes and properties files generated by `i18ngen` and `l10ngen` provide the interface for sending messages to the WebLogic server log. Within the classes, each log message is represented by a method that your application calls.

To use messages from compiled message catalogs:

1. In the class files for your application, import the `Logger` classes that you compiled in [Section 5.1.2, "Compile Message Catalogs."](#)

To verify the package name, open the message catalog XML file in a text editor and determine the value of the `i18n_package` attribute. For example, the following segment of the message catalog in [Example 5-1](#) indicates the package name:

```
<message_catalog
  i18n_package="com.xyz.msgcat"
```

To import the corresponding class, add the following line:

```
import com.xyz.msgcat.MyMessagesLogger;
```

2. Call the method that is associated with a message name.

Each message in the catalog includes a `method` attribute that specifies the method you call to display the message. For example, the following segment of the message catalog in [Example 5-1](#) shows the name of the method:

```
<logmessage
  messageid="700000"
  datelastchanged="1039193709347"
  datehash="-1776477005"
  severity="info"
  method="startup()"
```

&gt;

[Example 5-3](#) illustrates a simple class that calls this startup method.

**Example 5-3 Example Class That Uses a Message Catalog**

```
import com.xyz.msgcat.MyMessagesLogger;
public class MyClass {
    public static void main (String[] args) {
        MyMessagesLogger.startup();
    }
}
```

If the JVM's system properties specify that the current location is Spain, then the message is printed in Spanish.

## 5.2 Using the NonCatalogLogger APIs

In addition to using the I18N message catalog framework, your application can use the `weblogic.logging.NonCatalogLogger` APIs to send messages to the WebLogic server log. With `NonCatalogLogger`, instead of calling messages from a catalog, you place the message text directly in your application code. Oracle recommends that you do not use this facility as the sole means for logging messages if your application needs to be internationalized.

`NonCatalogLogger` is also intended for use by client code that is running in its own JVM (as opposed to running within a WebLogic Server JVM). A subsequent section, [Section 5.5, "Writing Messages from a Client Application,"](#) provides more information.

To use `NonCatalogLogger` in an application that runs within the WebLogic Server JVM, add code to your application that does the following:

1. Imports the `weblogic.logging.NonCatalogLogger` interface.
2. Uses the following constructor to instantiate a `NonCatalogLogger` object:

```
NonCatalogLogger(java.lang.String myApplication)
```

where `myApplication` is a name that you supply to identify messages that your application sends to the WebLogic server log.

3. Calls any of the `NonCatalogLogger` methods.

Use the following methods to report normal operations:

- `info(java.lang.String msg)`
- `info(java.lang.String msg, java.lang.Throwable t)`

Use the following methods to report a suspicious operation, event, or configuration that does not affect the normal operation of the server or application:

- `warning(java.lang.String msg)`
- `warning(java.lang.String msg, java.lang.Throwable t)`

Use the following methods to report errors that the system or application can handle with no interruption and with limited degradation in service.

- `error(java.lang.String msg)`
- `error(java.lang.String msg, java.lang.Throwable t)`

Use the following methods to provide detailed information about operations or the state of the application. These debug messages are not broadcast as JMX notifications. If you use this severity level, we recommend that you create a "debug mode" for your application. Then, configure your application to output debug messages only when the application is configured to run in the debug mode. For information about using debug messages, see [Section 5.6, "Writing Debug Messages."](#)

- `debug(java.lang.String msg)`
- `debug(java.lang.String msg, java.lang.Throwable t)`

All methods that take a `Throwable` argument can print the stack trace in the server log. For information on the `NonCatalogLogger` APIs, see the `weblogic.logging.NonCatalogLogger` Javadoc.

[Example 5-4](#) illustrates a servlet that uses `NonCatalogLogger` APIs to write messages of various severity levels to the WebLogic server log.

#### **Example 5-4 Example NonCatalogLogger Messages**

```
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.naming.Context;
import weblogic.jndi.Environment;
import weblogic.logging.NonCatalogLogger;
public class MyServlet extends HttpServlet {
    public void service (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        NonCatalogLogger myLogger = null;
        try {
            out.println("Testing NonCatalogLogger. See
                WLS Server log for output message.");
            // Constructing a NonCatalogLogger instance. All messages from this
            // instance will include a <MyApplication> string.
            myLogger = new NonCatalogLogger("MyApplication");
            // Outputting an INFO message to indicate that your application has started.
            mylogger.info("Application started.");
            // For the sake of providing an example exception message, the next
            // lines of code purposefully set an initial context. If you run this
            // servlet on a server that uses the default port number (7001), the
            // servlet will throw an exception.
            Environment env = new Environment();
            env.setProviderUrl("t3://localhost:8000");
            Context ctx = env.getInitialContext();
        }
        catch (Exception e){
            out.println("Can't set initial context: " + e.getMessage());
            // Prints a WARNING message that contains the stack trace.
            mylogger.warning("Can't establish connections. ", e);
        }
    }
}
```

When the servlet illustrated in the previous example runs on a server that specifies a listen port other than 8000, the following messages are printed to the WebLogic server log file. Note that the message consists of a series of strings, or fields, surrounded by angle brackets (< >).

**Example 5-5 NonCatalogLogger Output**

```
####<May 27, 2004 8:45:42 AM EDT> <Error> <MySubsystem> <myhost> <adminServer>
<ExecuteThread: '0' for queue: 'weblogic.kernel.Default (self-tuning)'\> <system>
<> <> <1085661942864> <OBJ-0011> <Test NonCatalogLogger message
java.lang.Exception: Test NonCatalogLogger message
...
>
```

## 5.3 Using ServletContext

The servlet specification provides the following APIs in `javax.servlet.ServletContext` that your servlets and JSPs can use to write a simple message to the WebLogic server log:

- `log(java.lang.String msg)`
- `log(java.lang.String msg, java.lang.Throwable t)`

For more information on using these APIs, see the Javadoc for the `javax.servlet.ServletContext` interface at [http://download.oracle.com/docs/cd/E17802\\_01/products/products/servlet/2.3/javadoc/javax/servlet/ServletContext.html](http://download.oracle.com/docs/cd/E17802_01/products/products/servlet/2.3/javadoc/javax/servlet/ServletContext.html).

[Example 5-6](#) illustrates JSP logging using the `ServletContext`.

**Example 5-6 Example JSP Logging**

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>INDEX</title>
</head>
<body bgcolor="#FFFFFF">
<%
config.getServletContext().log("Invoked ServletContext.log() From a JSP");
out.write("Request param arg0 = " + request.getParameter("arg0"));
%>
</body>
</html>
```

## 5.4 Configuring Servlet and Resource Adapter Logging

In this release of WebLogic Server, you can configure Web application and resource adapter logging behavior using WebLogic specific deployment descriptors. The logging configuration deployment descriptor elements define similar attributes used to configure server logging through the `LogMBean` interface, such as the log file name, location, and rotation policy.



When configured, application events are directed to a Web application specific log file. When the deployment descriptor does not include any logging configuration information, the default behavior is to write these events in the server log file.

Similarly, WebLogic logging services are provided to Java EE resource adapters for `ManagedConnectionFactory` scoped logging. You configure the log file name, location, and rotation policy for resource adapter logs through the `weblogic-ra.xml` deployment descriptor. See "weblogic-ra.xml Schema" in *Programming Resource Adapters for Oracle WebLogic Server*.

### Example: Logging Configuration Deployment Descriptors

[Example 5-7](#) illustrates a snippet of the deployment descriptor for configuring the logging behavior of Web application and resource adapter logging. The elements of logging correspond to attribute definitions on the `LogMBean` interface. All the `LogMBean` attributes are not listed in this example. Logging configuration is defined in the `WEB-INF/weblogic.xml` file for Web applications and in the `META-INF/weblogic-ra.xml` file for resource adapters.

#### Example 5-7 Configuring Web Application and Connector Logging

```
<!DOCTYPE weblogic-web-app PUBLIC "-//DTD Web Application 9.0//EN"
"http://www.bea.com/servers/wls90/dtd/weblogic90-web-jar.dtd">
<weblogic-web-app>
  <logging>
    <log-filename>d:\tmp\mywebapp.log</log-filename>
    <rotation-type>bySize</rotation-type>
    <number-of-files-limited>>true</number-of-files-limited>
    <file-count>3</file-count>
    <file-size-limit>50</file-size-limit>
    <rotate-log-on-startup>true</rotate-log-on-startup>
    <log-file-rotation-dir>config/MedRecDomain/WebApp</log-file-rotation-dir>
  </logging>
</weblogic-web-app>
<weblogic-connector xmlns="http://www.bea.com/ns/weblogic/90">
  <jndi-name>eis/900BlackBoxNoTxConnector</jndi-name>
  <outbound-resource-adapter>
    <connection-definition-group>

<connection-factory-interface>javax.sql.DataSource</connection-factory-interface>
  <connection-instance>
    <jndi-name>eis/900BlackBoxNoTxConnectorJNDIName</jndi-name>
    <connection-properties>
      <pool-params>
        <initial-capacity>5</initial-capacity>
        <max-capacity>10</max-capacity>
        <capacity-increment>1</capacity-increment>
        <shrinking-enabled>true</shrinking-enabled>
        <shrink-frequency-seconds>60</shrink-frequency-seconds>
        <highest-num-waiters>1</highest-num-waiters>
        <highest-num-unavailable>3</highest-num-unavailable>

<connection-reserve-timeout-seconds>11</connection-reserve-timeout-seconds>
      </pool-params>
    </connection-instance>
  </connection-properties>
  <logging>
    <log-filename>900BlackBoxNoTxOne.log</log-filename>
    <logging-enabled>true</logging-enabled>
    <rotation-type>bySize</rotation-type>
    <number-of-files-limited>true</number-of-files-limited>
    <file-count>3</file-count>
```

```
<file-size-limit>100</file-size-limit>
<rotate-log-on-startup>true</rotate-log-on-startup>
<log-file-rotation-dir>c:/mylogs</log-file-rotation-dir>
<rotation-time>3600</rotation-time>
<file-time-span>7200</file-time-span>
</logging>
<properties>
  <property>
    <name>ConnectionURL</name>
    <value>jdbc:oracle:thin:@bcpdb:1531:bay920</value>
  </property>
  <property>
    <name>unique_ra_id</name>
    <value>blackbox-notx.oracle.810</value>
  </property>
</properties>
</connection-properties>
</connection-instance>
</connection-definition-group>
</outbound-resource-adapter>
</weblogic-connector>
```

## 5.5 Writing Messages from a Client Application

If your application runs in a JVM that is separate from a WebLogic Server, it can use message catalogs and `NonCatalogLogger`, but the messages are not written to the WebLogic server log. Instead, the application's messages are written to the client JVM's standard out.

If you want the WebLogic logging service to send these messages to a log file that the client JVM maintains, include the following argument in the command that starts the client JVM:

```
-Dweblogic.log.FileName=logfilename
```

where *logfilename* is the name that you want to use for the remote log file.

If you want a subset of the message catalog and `NonCatalogLogger` messages to go to standard out as well as the remote JVM log file, include the following additional startup argument:

```
-Dweblogic.log.StdoutSeverityLevel=String
```

where valid values for `StdoutSeverityLevel` are:

Debug, Info, Warning, Error, Notice, Critical, Alert, Emergency, and Off.

See the `weblogic.logging.Severities` class for a description of the supported severity levels.

## 5.6 Writing Debug Messages

While your application is under development, you might find it useful to create and use messages that provide verbose descriptions of low-level activity within the application. You can use the `DEBUG` severity level to categorize these low-level messages. All `DEBUG` messages that your application generates are sent to all WebLogic server logging destinations, depending on the configured minimum threshold severity level.

If you use the `DEBUG` severity level, we recommend that you create a "debug mode" for your application. For example, your application can create an object that contains a Boolean value. To enable or disable the debug mode, you toggle the value of the Boolean. Then, for each `DEBUG` message, you can create a wrapper that outputs the message only if your application's debug mode is enabled.

For example, the following code can produce a debug message:

```
private static boolean debug = Boolean.getBoolean("my.debug.enabled");
if (debug) {
    mylogger.debug("Something debuggy happened");
}
```

You can use this type of wrapper both for `DEBUG` messages that use the message catalog framework and that use the `NonCatalogLogger` API.

To enable your application to print this message, you include the following Java option when you start the application's JVM:

```
-Dmy.debug.enabled=true
```



---

---

## Using the WebLogic Server Message Editor

This chapter describes how to use the Message Editor, which is a graphical interface tool that lets you create, read, and write XML message catalogs.

This chapter includes the following sections:

- [About the Message Editor](#)
- [Starting the Message Editor](#)
- [Working with Catalogs](#)
- [Adding Messages to Catalogs](#)
- [Finding Messages](#)
- [Using the Message Viewer](#)
- [Editing an Existing Message](#)
- [Retiring and Unretiring Messages](#)

### 6.1 About the Message Editor

The Message Editor is installed when you install WebLogic Server. Optionally, you can edit the XML catalogs in a text editor or with any XML editing tool.

---

---

**Note:** WebLogic Server provides its own message catalogs which contain all the messages relating to WebLogic Server subsystems and functionality. You cannot edit these catalogs. For descriptions of WebLogic Server catalogs, see *Oracle WebLogic Server Error Message Reference*.

---

---

You use the Message Editor to perform the following tasks:

- Create XML message catalogs
- Create and edit messages
- View all the messages in one catalog
- View the messages in several catalogs simultaneously
- Search for messages
- Validate the XML in catalog entries
- Retire and unretire messages

---

---

**Note:** The Message Editor does not support the editing of localized catalogs.

---

---

## 6.2 Starting the Message Editor

Before you start the Message Editor, install and configure your WebLogic Server system and set the environment variables, `SAMPLES_HOME\domains\wl_server\setExamplesEnv.cmd`, where `SAMPLES_HOME` represents the WebLogic Server examples home directory at `WL_HOME\samples`). Make sure that your classpath is set correctly. For more information on these topics, see the *Installation Guide for Oracle WebLogic Server*.

Sample message catalog files are located in your `SAMPLES_HOME/server/examples/src/examples/i18n/msgcat` directory.

---

---

**Note:** The location of your `SAMPLES_HOME` directory might be different, depending on where you installed WebLogic Server.

---

---

You can use the sample message catalogs as templates to create your own messages. You simply modify the provided information, such as the package name and class name. Then translate the message text and save the catalog. For more information on this topic, see [Chapter 5, "Writing Messages to the WebLogic Server Log."](#)

To start the Message Editor, type:

```
java weblogic.MsgEditor
```

or

```
java weblogic.i18ntools.gui.MessageEditor
```

To access basic command line help, type:

```
java weblogic.MsgEditor -help
```

The main WebLogic Message Editor window for **Log Messages** appears, as shown in [Figure 6-1](#).

Figure 6–1 WebLogic Message Editor for Log Messages

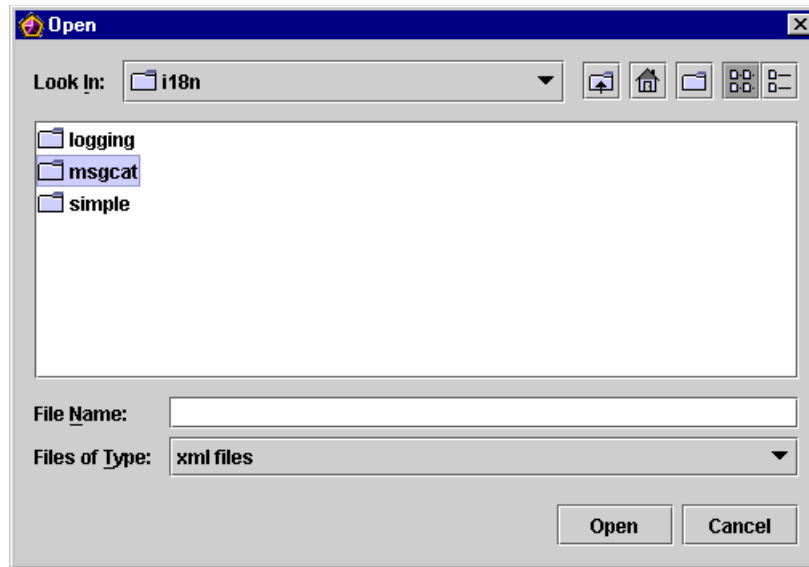
## 6.3 Working with Catalogs

The following sections describe how to use the Message Editor to manage catalogs:

- [Browsing to an Existing Catalog](#)
- [Creating a New Catalog](#)

### 6.3.1 Browsing to an Existing Catalog

To find an existing catalog from the main WebLogic Message Editor window, enter the full pathname in the Message Catalog field, or click Browse and navigate to the existing catalog from the Open dialog.

**Figure 6–2** Navigating to a Catalog

The sample catalogs included with your WebLogic Server installation are in the `SAMPLES_HOME/server/examples/src/examples/i18n/msgcat` directory.

---

---

**Note:** Your directory path might be different, depending on where you installed WebLogic Server.

---

---

You can place your user-defined catalogs in any directory you designate.

Once you locate the catalog, **Packages**, **Subsystem**, **Version**, **Base ID**, and **End ID** (if any) are displayed, and the displayed catalog is the context catalog in which all other actions are performed.

You are now ready to enter new messages, edit existing messages, search for a message, or view all messages in the catalog.

If you select a log message catalog in the **Message catalog** field, the WebLogic Message Editor window for **Log Messages** appears, as shown in [Figure 6–3](#).



Figure 6–3 WebLogic Message Editor for Log Messages

WebLogic Message Editor - UserServerSVExample.xml

File Edit View Options

WebLogic Message Catalog Editor  
Editing non-server message catalogs

Message catalog: 18nmsgcat\UserServerSVExample.xml

I18n Package:	examples.i18n.logging.message	L10n Package:	examples.i18n.logging.message
Subsystem:	UserServlet	Version:	1.0
Base id:	909000	End id:	909009
Loggables:	false	Prefix:	

Log Messages

Last Updated: Thu Oct 03 12:55:53 MDT 2002

Message ID:

Comment:

Method:

Method Type: logger

Severity: error

Message body:

Message detail:

Probable cause:

Action:

Display stack trace:

Retired message:

If you select a simple messages catalog in the **Message catalog** field, the WebLogic Message Editor window for **Simple Messages** appears, as shown in [Figure 6–4](#).

Figure 6–4 WebLogic Message Editor for Simple Messages

WebLogic Message Editor - I18nSimple.xml

File Edit View Options

WebLogic Message Catalog Editor  
Editing non-server message catalogs

Message catalog: examples\i18nmsgcat\I18nSimple.xml

I18n Package:	examples.i18n.logging	L10n Package:	examples.i18n.logging
Subsystem:	I18N	Version:	1.0
Base id:		End id:	
Loggables:	false	Prefix:	

Simple Messages

Last Updated: Thu Oct 03 12:58:32 MDT 2002

Message ID:

Comment:

Method:

Message body:

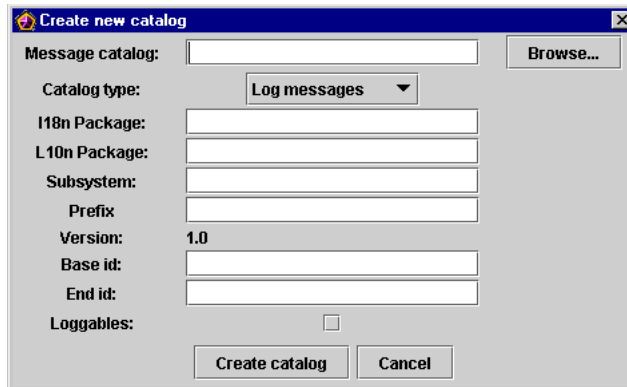
## 6.3.2 Creating a New Catalog

To create a new catalog, complete the following procedure:

1. From the main menu bar of the WebLogic Message Editor window, choose **File > New Catalog**.

The Create New Catalog dialog appears, as shown in [Figure 6–5](#).

**Figure 6–5 Create New Catalog**



1. In the **Message Catalog** field, enter the full pathname and the name of the new catalog, which must include the `xml` extension. Or, click **Browse** and navigate to the appropriate catalog directory. (This would be the `msgcat` directory, if you are using WebLogic Server example messages.)

2. Use the drop-down **Catalog type** list to indicate whether your catalog is a **Log messages** or a **Simple messages** catalog.

If you select a log message catalog, the **Base ID** and **End ID** fields are displayed. These fields indicate the range of ID numbers for messages in the catalog. If you select a simple text message catalog, these fields are not present.

3. Enter the name of the package in which you want to place the generated Logger classes in the **I18n Package** field.

The default is `weblogic.i18n`. If you want to place the logger classes in another package with your application, specify the package name here.

4. Enter the name of the package where you want to place the catalog data in the **L10n Package** field.

The default is `weblogic.l10n`. If you want to place your catalog data in another package with your application, specify the package name here.

---

**Note:** In most cases, the recommended practice is to include the message class files and properties files in the same package hierarchy as your application.

However, if you do not include the message classes and properties in the application's package hierarchy, you must make sure the classes are in the application's classpath.

---

5. Enter a name in the **Subsystem** field to indicate which part of the system will log the message.

This name is logged with the message. For applications, the application name is typically entered in the **Subsystem** field.

6. In the **Prefix** field, enter a prefix to be prepended to the message ID when logged.

The default server message prefix is BEA. You can enter any prefix for user messages. (Oracle recommends that the prefix be less than 10 characters in length. Also, make sure you use standard java naming conventions.)

7. Click **Create Catalog**.

The Create New Catalog dialog closes, and the catalog you just created is displayed as the context catalog in the Message Editor window.

## 6.4 Adding Messages to Catalogs

The following sections describe how to use the Message Editor to add messages to catalogs:

- [Entering a New Log Message](#)
- [Entering a New Simple Text Message](#)

### 6.4.1 Entering a New Log Message

To enter a new message into a log catalog:

1. In the WebLogic Message Editor main dialog ([Figure 6-6](#)), enter the full pathname in the **Message Catalog** field or click **Browse** and navigate to an existing catalog.

Figure 6–6 Log Messages

1. Click **Get next ID** to generate the next unique numerical ID in the context catalog. The ID appears in the Message ID field.
2. Enter any relevant comments about the message in the **Comment** field.
3. Enter the appropriate **Method** for your log message, including parentheses and any arguments. For example, `logErrorSavingTimestamps (Exception ioExcep)`
4. Set the **Method Type** for the log message.  
Your options are `logger` and `getter`. The default method type is `logger`, which is used for messages that will be logged. The `getter` option is for messages that are used for non-logging purposes, such as exceptions.
5. Choose a **Severity** from the list of possible levels.
6. Enter text for the **Message body**.  
Parameters are denoted by `{n}`. For example, "Exception occurred while loading `_WL_TIMESTAMP` FILE."
7. Enter text for the **Message detail**.  
Parameters are denoted by `{n}`. For example, "Exception occurred while loading `_WL_TIMESTAMP` FILE. Forcing recompilation: `{0}`."

**8. Enter text for the Probable Cause.**

Parameters are denoted by {*n*}. For example, "There was an error reading this file."

**9. Enter text for the Action.**

Parameters are denoted by {*n*}. For example, "No action required."

**10. Toggle the Display stacktrace option by selecting or clearing the check box.**

Use this option to print a stacktrace along with the message when a Logger method takes an exception as one of its arguments.

**11. Toggle the Retired message option by selecting or clearing the check box.**

Use this option to retire (hide) obsolete messages. Retired messages are deleted in the sense that they are not represented in the generated classes. However, the message data does remain in the .xml file.

**12. Click Add.**

The message is added and the entire catalog is immediately written to disk.

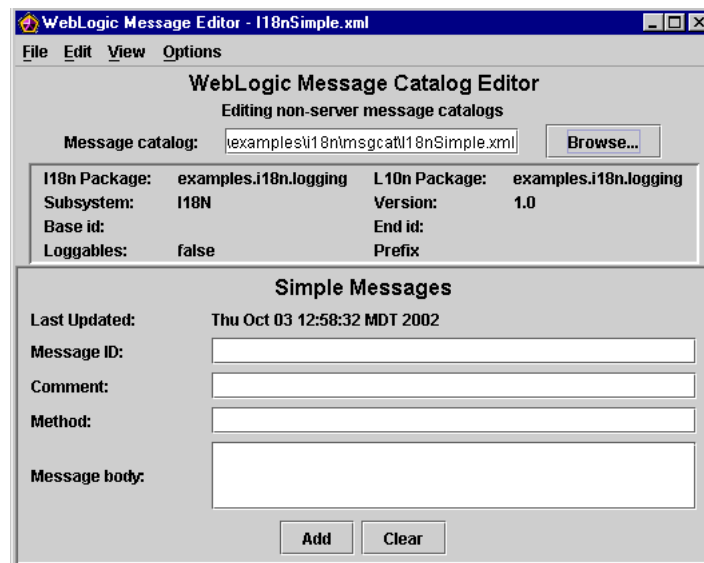
## 6.4.2 Entering a New Simple Text Message

To enter a simple text message into a simple messages catalog:

1. In the WebLogic Message Editor main dialog, enter the full pathname in the **Message Catalog** field or click **Browse** and navigate to the existing catalog.

The WebLogic Message Editor for Simple Messages dialog appears, as shown in [Figure 6-7](#).

**Figure 6-7 Simple Messages**



1. Enter a unique alphanumeric **Message ID**.
2. Enter a **Comment** if required.
3. Enter the appropriate **Method** for your simple message, including parentheses and any arguments. For example, `startingClusterService()`
4. Enter the **Message body** text. For example, `startingClusterService`

5. Click **Add**.

The message is added and the entire catalog is immediately written to disk.

## 6.5 Finding Messages

The following sections describe how to use the Message Editor to find messages:

- [Finding a Log Message](#)
- [Finding a Simple Text Message](#)

### 6.5.1 Finding a Log Message

To find a log message:

1. Make sure that the context catalog is a log message catalog and the WebLogic Message Editor **Log Messages** window appears, as shown in [Figure 6-3](#).
2. Choose **Edit** from the main menu bar.
3. Choose **Search** to display the Search for Log Message dialog, as shown in [Figure 6-8](#).

*Figure 6-8 Search for Log Message*



1. Enter the **Message ID** and the **Method** name.
2. Enter as much information as needed in the **Message text search** field to find the correct message.

The search for text does a partial match in any of the text fields.

3. Click **Find first** or **Find next**.

The fields are strung together to find the message. If a matching message is found, it is displayed in the Message Editor window, as shown in [Figure 6-1](#).

### 6.5.2 Finding a Simple Text Message

To find a simple text message, complete the following procedure:

1. Make sure that the context catalog is a simple text message catalog and the WebLogic Message Editor **Simple Messages** window appears, as shown in [Figure 6-4](#) window appears.
2. Choose **Edit** from the main menu bar.
3. Choose **Search** to display the Search for Simple Message dialog, as shown in [Figure 6-9](#).

**Figure 6–9 Search for Simple Message**

1. Enter the **Message ID**.
2. Enter as much information as needed in the **Message text search** field to find the correct message.

The search for text does a partial match in any of the text fields.

3. Click **Find first** or **Find next**.

The fields are strung together to find the message. If a matching message is found, it is displayed in the Message Editor window, as shown in [Figure 6–4](#).

## 6.6 Using the Message Viewer

The WebLogic Message Editor contains a Message Viewer that lets you view all messages in a catalog, view all messages in multiple catalogs, and select any message to edit.

The following sections describe how to use the Message Viewer to view and select messages to edit:

- [Viewing All Messages in a Catalog](#)
- [Viewing All Messages in Several Catalogs](#)
- [Selecting a Message to Edit from the Message Viewer](#)

### 6.6.1 Viewing All Messages in a Catalog

To view all the messages in a catalog:

1. Open the WebLogic Message Editor.  
The WebLogic Message Editor window displays the catalog for the last message viewed as the current context catalog.
2. **Choose View** from the menu bar.
3. Choose **All messages**.

All the messages for the current context catalog are displayed in the Message Viewer window, as shown in [Figure 6–10](#). The Message Editor window remains open.

**Figure 6–10 Message Viewer**

The screenshot shows a window titled "Message Viewer:UserServerSVExample.xml". Below the title bar, the text "Messages in UserServerSVExample.xml catalog" is displayed. A table with the following columns is shown: Message ID, Method, Method Type, Comment, Severity, Body, Detail, Action, Cause, and Retired. The table contains three rows of data.

Message ID	Method	Method Type	Comment	Severity	Body	Detail	Action	Cause	Retired
909000	logFirstMe...	logger		info	First mes...	Just an ex...			false
909001	logMoreSt...	logger		warning	The servle...				false
909002	logDone()	logger		error	This is not...				false

### 6.6.2 Viewing All Messages in Several Catalogs

If you view the messages from the current context catalog and then change the context by navigating to a new catalog, a second Message Viewer window opens displaying the new catalog. You can view messages for as many catalogs as you require (or can reasonably fit on your screen). Each catalog is displayed in a separate Message Viewer window. See [Section 6.3.1, "Browsing to an Existing Catalog,"](#) for information about browsing to a new catalog.

### 6.6.3 Selecting a Message to Edit from the Message Viewer

You can select any message displayed in the Message Viewer and the selected message becomes the context catalog. The message is displayed in the Message Editor window.



Figure 6–11 Message Viewer and Message Editor for Message 909001

## 6.7 Editing an Existing Message

To edit an existing message:

1. Find the message you want to edit.

You can use the Search dialog, described in [Section 6.5.1, "Finding a Log Message,"](#) and [Section 6.5.2, "Finding a Simple Text Message,"](#) or select the message in the message viewer, described in [Section 6.6.3, "Selecting a Message to Edit from the Message Viewer."](#)

The message appears in the Message Editor window.

2. Edit the fields you want to change.
3. Click **Update**.

The message is updated and the entire catalog is immediately written to disk.

## 6.8 Retiring and Unretiring Messages

You can retire and unretire messages in the Message Editor window. Retiring a message does not mean that the message is deleted from the master catalog; it is simply hidden from user view. This feature is useful for removing obsolete messages. If you need to bring a retired message back into view, you can unretire it.

To retire or unretire a message, complete the following procedure:

1. Find the message you want to retire or unretire.
2. In the Message Editor window, toggle the **Retired message** option by selecting or clearing the check box.
3. Click **Update**.

---

---

## Using the WebLogic Server Internationalization Utilities

This chapter describes the WebLogic Server utilities used for internationalization and localization, including `weblogic.i18ngen`, `weblogic.l10ngen`, and `weblogic.GetMessage`.

This chapter includes the following sections:

- [WebLogic Server Internationalization Utilities](#)
- [WebLogic Server Internationalization and Localization](#)
- [weblogic.i18ngen Utility](#)
- [weblogic.l10ngen Utility](#)
- [weblogic.GetMessage Utility](#)

### 7.1 WebLogic Server Internationalization Utilities

WebLogic Server provides three internationalization utilities:

- [weblogic.i18ngen Utility](#) - Message catalog parser. Use this utility to validate and generate classes used for localizing text in log messages. For more information, see [Section 7.3, "weblogic.i18ngen Utility."](#)
- [weblogic.l10ngen Utility](#) - Locale-specific message catalog parser. Use this utility to process locale-specific catalogs. For more information, see [Section 7.4, "weblogic.l10ngen Utility."](#)
- [weblogic.GetMessage Utility](#) - Utility that lists installed log messages. Use this utility to generate a list of installed log messages or display a message. For more information, see [Section 7.5, "weblogic.GetMessage Utility."](#)

---

---

**Notes:** Text in the catalog definitions may contain formatting characters for readability (for example, end of line characters), but these are not preserved by the parsers. Text data is normalized into a one-line string. All leading and trailing white space is removed. All embedded end of line characters are replaced by spaces as required to preserve word separation. Tabs are left intact.

Use escapes to embed new lines (for example ' \n '). These are stored and result in new lines when printed.

---

---

## 7.2 WebLogic Server Internationalization and Localization

Use the `weblogic.i18ngen` utility to validate message catalogs and create the necessary run-time classes for producing localized messages. The `weblogic.l10ngen` utility validates locale-specific catalogs, creating additional properties files for the different locales defined by the catalogs.

You can internationalize simple text-based utilities that you are running on WebLogic Server by specifying that those utilities use `Localizers` to access text data. You configure the applications with `Logger` and `TextFormatter` classes generated from the `weblogic.i18ngen` utility.

For more information on `Logger` and `TextFormatter` classes, see [Appendix C, "TextFormatter Class Reference for WebLogic Server,"](#) and [Appendix D, "Logger Class Reference for WebLogic Server."](#)

The generated `Logger` classes are used for logging purposes, as opposed to the traditional method of writing English text to a log. For example, `weblogic.i18ngen` generates a class `xyzLogger` in the appropriate package for the catalog `xyz.xml`. For the `MyUtilLog.xml` catalog, the class, `programs.utils.MyUtilLogger.class`, would be generated. For each log message defined in the catalog, this class contains static public methods as defined by the method attributes.

`TextFormatter` classes are generated for each simple message catalog. These classes include methods for accessing localized and formatted text from the catalog. They are convenience classes that handle the interface with the message body, placeholders, and `MessageFormat`. You specify the formatting methods through the method attribute in each message definition. For example, if the definition of a message in a catalog includes the attribute, `method=getErrorNumber(int err)`, the `TextFormatter` class shown in [Example 7-1](#) is generated.

### **Example 7-1 Example of a TextFormatter Class**

```
package my.text;
public class xyzTextFormatter
{
    . . .
    public String getErrorNumber(int err)
    {
        . . .
    }
}
```

[Example 7-2](#) shows an example of how the `getErrorNumber` method could be used in code.

### **Example 7-2 Example of getErrorNumber Method**

```
import my.text.xyzTextFormatter
. . .

xyzTextFormatter xyzL10n = new xyzTextFormatter();
System.out.println(xyzL10n.getErrorNumber(someVal));
```

The output prints the message text in the current locale, with the `someVal` argument inserted appropriately.

## 7.3 weblogic.i18ngen Utility

Use the `weblogic.i18ngen` utility to parse message catalogs (XML files) to produce `Logger` and `TextFormatter` classes used for localizing the text in log messages. The utility creates or updates the following properties file, which is used to load the message ID lookup class hashtable `weblogic.i18n.L10nLookup`:

```
targetdirectory\i18n_user.properties
```

Any errors, warnings, or informational messages are sent to `stderr`.

In order for user catalogs to be recognized, the `i18n_user.properties` file must reside in a directory identified in the `WebLogic` classpath.

For example: `targetdirectory\i18n_user.properties`

Oracle recommends that the `i18n_user.properties` file reside in the server classpath. If the `i18n_user.properties` file is in `targetdirectory`, then `targetdirectory` should be in the server classpath.

### 7.3.1 Syntax

```
java weblogic.i18ngen [options] [filelist]
```

---



---

**Note:** Utilities can be run from any directory, but if files are listed on the command line, then their path is relative to the current directory.

---



---

### 7.3.2 Options

Option	Definition
<code>-build</code>	Generates all necessary files and compiles them.  The <code>-build</code> option combines the <code>-i18n</code> , <code>-l10n</code> , <code>-keepgenerated</code> , and <code>-compile</code> options.
<code>-d targetdirectory</code>	Specifies the root directory to which generated Java source files are targeted. User catalog properties are placed in <code>i18n_user.properties</code> , relative to the designated target directory. Files are placed in appropriate directories based on the <code>i18n_package</code> and <code>l10n_package</code> values in the corresponding message catalog. The default target directory is the current directory. This directory is created as necessary.  If this argument is omitted, all classes are generated in the current directory, without regard to any class hierarchy described in the message catalog.
<code>-n</code>	Parse and validate, but do not generate classes.
<code>-keepgenerated</code>	Keep generated Java source (located in the same directory as the class files).
<code>-ignore</code>	Ignore errors.
<code>-i18n</code>	Generates internationalizers (for example, <code>Loggers</code> and <code>TextFormatters</code> ).  <code>i18ngen -i18n</code> creates the internationalizer source (for example, <code>*Logger.java</code> ) that supports the logging of internationalized messages.

Option	Definition
-l10n	Generates localizers (for example, <code>LogLocalizers</code> and <code>TextLocalizers</code> ).  <code>i18ngen -l10n</code> creates the localizer source (resource bundles) that provide access to each message defined in the message catalog. These classes are used by localization utilities to localize messages.
-compile	Compiles generated Java files using the current CLASSPATH. The resulting classes are placed in the directory identified by the <code>-d</code> option. The resulting classes are placed in the same directory as the source.  Errors detected during compilation generally result in no class files or properties file being created. <code>i18ngen</code> exits with a bad exit status.
-nobuild	Parse and validate only.
-debug	Debugging mode.
-dates	Causes <code>weblogic.i18ngen</code> to update message timestamps in the catalog. If the catalog is writable and timestamps have been updated, the catalog is rewritten.
<i>filelist</i>	Process the files and directories in this list of files. If directories are listed, the command processes all XML files in the listed directories. The names of all files must include an XML suffix. All files must conform to the <code>msgcat.dtd</code> syntax. <code>weblogic.i18ngen</code> prints the fully-qualified list of names (Java source) to the <code>stdout</code> log for those files actually generated.

## 7.4 weblogic.l10ngen Utility

The `weblogic.l10ngen` utility generates property resources for localizations of message catalogs named in the file list. The file list identifies the top-level catalogs, not translated catalogs.

Similarly the target directory (`-d` option) identifies the same target directory where the default localizations reside. For example, if the default catalogs are located in `$SRC\weblogic\msgcat` and the generated resources are to be placed in `$CLASSES_DIR`, the appropriate `l10ngen` invocation would be:

```
java weblogic.l10ngen -d $CLASSES_DIR $SRC\weblogic\msgcat
```

This command generates localized resources for all locales defined in the `weblogic\msgcat` subdirectories.

### 7.4.1 Syntax

```
java weblogic.l10ngen [options] [filelist]
```

---



---

**Note:** Utilities can be run from any directory, but if files are listed on the command line, then their path is relative to the current directory.

---



---

### 7.4.2 Options

Option	Definition
<code>-d target</code>	Directory in which to place properties. Default is the current directory.
<code>-language code</code>	Language code. Default is <code>all</code> .

Option	Definition
<code>-country code</code>	Country code. Default is <code>all</code> .
<code>-variant code</code>	Variant code. Default is <code>all</code> .
<code>filelist</code>	Specifies the message catalogs for which you want to generate properties files. You must specify top-level catalogs that the Message Editor creates; you do not specify locale-specific catalogs that the Message Localizer creates. Usually, this is the same set of source files or source directories that you specified in the <code>i18ngen</code> command.

### 7.4.3 Message Catalog Localization

Catalog subdirectories are named after lowercase, two-letter ISO 639 language codes (for example, `ja` for Japanese and `fr` for French). You can find supported language codes in the `java.util.Locale` javadoc.

Variations to language codes are achievable through the use of uppercase, two-letter ISO 3166 country codes and variants, each of which are subordinate to the language code. The generic syntax is `lang\country\variant`.

For example, `zh` is the language code for Chinese. `CN` is a country code for simplified Chinese, whereas `TW` is the country code for traditional Chinese. Therefore `zh\CN` and `zh\TW` are two distinct locales for Chinese.

Variants are of use when, for instance, there is a functional difference in platform vendor handling of specific locales. Examples of vendor variants are `WIN`, `MAC`, and `POSIX`. There may be two variants used to further qualify the locale. In this case, the variants are separated with an underscore (for example, `Traditional_Mac` as opposed to `Modern_MAC`).

---

**Note:** Language, country, and variants are all case sensitive.

---

A fully-qualified locale would look like `zh\TW\WIN`, identifying traditional Chinese on a Win32 platform.

Message catalogs to support the above locale would involve the following files:

- `\*.xml` - default catalogs
- `\zh\*.xml` - Chinese localizations
- `\zh\TW\*.xml` - Traditional Chinese localizations
- `\zh\TW\WIN\*.xml` - Traditional Chinese localizations for Win32 code sets

Specific localizations do not need to cover all messages defined in parent localizations.

### 7.4.4 Examples

1. To generate localization properties for all locales:

```
java weblogic.l10ngen -d $CLASSESEDIR catalogdirectory
```

2. To generate localization properties for all traditional Chinese locales:

```
java weblogic.l10ngen -d $CLASSESEDIR -language zh -country TW catalogdirectory
```

3. To generate localization properties for all Chinese locales:

```
java weblogic.l10ngen -d $CLASSESEDIR -language zh catalogdirectory
```

4. To generate localization properties for the JMS catalog in all locales:

```
java weblogic.l10ngen -d $CLASSESEDIR catalogdirectory
```

---



---

**Note:** Example 2 is a subset of example 3. All Chinese (zh) would include any country designations (for example, TW) and variants.

weblogic.l10ngen does not validate the locale designators (language, country, variant).

---



---

## 7.5 weblogic.GetMessage Utility

The `weblogic.GetMessage` utility replaces the `CatInfo` utility provided with earlier releases of WebLogic Server. This utility displays message content but can also be used to list all or some subset of installed messages. By default (no options), `weblogic.GetMessage` prints a usage statement.

### 7.5.1 Syntax

```
java weblogic.GetMessage [options]
```

### 7.5.2 Options

---



---

**Note:** All options may be abbreviated to a single character except `-verbose`.

---



---

Option	Definition
<code>-id nnnnnn</code>	where <i>nnnnnn</i> represents the message ID. The <code>-id</code> option is used to specify a particular message.
<code>-subsystem identifier</code>	The subsystem identifier. The <code>-subsystem</code> option prints only those messages that match the specified subsystem.
<code>-nodetail</code>	Requests a non-detailed listing, and only outputs the message body of a message. By default, a detailed listing is output, which includes severity, subsystem, message detail, cause, and action information.
<code>-verbose</code>	Requests more detail on the listing. The <code>-verbose</code> option also prints packaging, stacktrace option, severity, subsystem, message detail, cause, and action information.
<code>-lang code</code>	The language to use. For example, <code>en</code> for English.
<code>-country code</code>	The country code to use. For example, <code>US</code> for United States.
<code>-variant code</code>	The variant designator for locale.
<code>-help</code>	Provides usage information.
<code>-retired</code>	Lists all retired messages. Retired messages are not displayed unless this option is used. Only the subsystem and ID's of such messages are listed.

If no arguments are provided, `weblogic.GetMessage` outputs a usage message, equivalent to `-help`.



---

---

# Localizer Class Reference for WebLogic Server

This appendix describes the `Localizer` classes, `Localizer` classes, `Localizer` methods, key values for `Localizers`, and lookup properties for `Localizers`.

This appendix includes the following sections:

- [About Localizer Classes](#)
- [Localizer Methods](#)
- [Localizer Lookup Class](#)

---

---

**Note:** This information on `Localizer` class methods is provided as reference for advanced users. Normally, you do not need to use these interfaces directly. Instead, you would typically use the generated methods in the catalogs.

---

---

## A.1 About Localizer Classes

The `weblogic.i18ngen` utility creates `Localizer` classes based on the content of the message catalog.

One `Localizer` class is generated for each catalog file. The name of the class is the catalog name (without the `.xml` extension, which is stripped by the utility), followed by `LogLocalizer` for log message catalogs and `TextLocalizer` for simple text catalogs. A `Localizer` class for the catalog `ejb.xml` is `ejbLogLocalizer`.

## A.2 Localizer Methods

`Localizers` are `PropertyResourceBundle` objects. Four additional methods are provided to simplify the access of the localization data in the `Localizer`. These methods are described in [Table A-1](#).

These methods are not part of the `Localizer`. Rather, they are part of the `Localizer` class. The `Localizer` class is used by the `Logger` and `TextFormatter` classes to extract data out of the `Localizer`. Each `Localizer` has an associated `Localizer` class that is obtained through `L10nLookup`, the `Localizer` lookup object.

**Table A-1 Methods for Localization Data Access**

Method	Description
<code>public Object getObject(String key, String id)</code>	Returns localization text for the key element for message id.
<code>public Object getObject(String key, int id)</code>	Returns localization text for the key element for message id.
<code>public String getString(String key, String id)</code>	Returns localization text for the key element for message id.
<code>public String getString(String key, int id)</code>	Returns localization text for the key element for message id.

Each of the methods for accessing localization data has a key argument. The following list shows the recognized values for the key argument:

- `Localizer.SEVERITY`
- `Localizer.MESSAGE_ID`
- `Localizer.MESSAGE_BODY`
- `Localizer.MESSAGE_DETAIL`
- `Localizer.CAUSE`
- `Localizer.ACTION`

With the exception of the `Localizer.SEVERITY` key, the localization data returned by Localizers are String objects that return an integer object.

The following list shows the severity values that are returned:

- `weblogic.logging.severities.EMERGENCY`
- `weblogic.logging.severities.ALERT`
- `weblogic.logging.severities.CRITICAL`
- `weblogic.logging.severities.NOTICE`
- `weblogic.logging.severities.ERROR`
- `weblogic.logging.severities.WARNING`
- `weblogic.logging.severities.INFO`
- `weblogic.logging.severities.DEBUG`

The specific strings returned are defined in the message catalogs.

The key argument to the `get*()` methods identify which element of a definition to return. Acceptable values are defined in the `Localizer` class definition. The returned text can be further expanded through `java.text.MessageFormat.format()`. The message body, detail, cause, and action elements are all localizable. The other elements, message ID, severity, and subsystem are not localizable and do not require further processing by `MessageFormat`.

### A.3 Localizer Lookup Class

To obtain the correct `Localizer` for a message, use the `L10nLookup` class, which is a property class extension that is loaded at system startup from the property file:

```
i18n_user.properties
```

This property file is created by `weblogic.i18ngen` and is included in the WebLogic Server installation. When you start up a user application, any `i18n_user.properties` files in its classpath are also loaded into `L10nLookup`.

Properties in the lookup (`i18n_user.properties`) file have the following format:

```
nnnnn=subsystem:Localizer class
```

The arguments on this line are defined as follows:

- *nnnnn* is the message ID
- *subsystem* is the related subsystem
- *Localizer class* is the name of the generated `Localizer` class

For example, message 001234 is identified as an EJB subsystem message ID from the `weblogic.i18n.ejbLogLocalizer` class by the following property in the lookup file:

```
001234=EJB:weblogic.i18n.ejbLogLocalizer
```



---

---

# Loggable Object Reference for WebLogic Server

This appendix describes loggable objects, which are used for generating log messages that are logged at a later time.

This appendix includes the following sections:

- [About Loggable Objects](#)
- [How To Use Loggable Objects](#)

## B.1 About Loggable Objects

By default, all log message catalogs create `Logger` classes with methods that are used to log the messages to the WebLogic server log. The `Logger` classes can optionally include methods that return a loggable object instead of logging the message. Loggable objects are useful when you want to generate the log message but actually log it at a later time. They are also useful if you want to use the message text for other purposes, such as throwing an exception.

## B.2 How To Use Loggable Objects

To create a `Logger` class that provides methods to return loggable objects, you must set the `loggables` attribute in the message catalog.

For example, consider the `test.xml` catalog shown in [Example B-1](#).

### **Example B-1** *test.xml Message Catalog*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog
  subsystem="Examples"
  version="1.0"
  baseid="500000"
  endid="500001"
  loggables="true"
>
  <logmessage
    messageid="500000"
    severity="error"
    method="logIOError(Throwable t)"
  >
    <messagebody>
```

```
        IO failure detected.
    </messagebody>
    <messagedetail>
    </messagedetail>
    <cause>
    </cause>
    <action>
    </action>
    </logmessage>
</message_catalog>
```

When you run this catalog through the `weblogic.i18ngen` utility, a `Logger` class is created for this catalog with the following two methods:

- `logIOException (throwable)` - logs the message
- `logIOExceptionLoggable (throwable)` - returns a loggable object

The loggable object can be used as shown in [Example B-2](#).

**Example B-2 Example of Use of Loggable Object**

```
package test;
import weblogic.logging.Loggable;
import weblogic.i18n.testLogger;
...
try {
    // some IO
} catch (IOException ioe) {
    Loggable l = testLogger.logIOExceptionLoggable(ioe);
    l.log(); // log the error
    throw new Exception(l.getMessage()); // throw new exception with
        same text as logged
}
```

---

---

# TextFormatter Class Reference for WebLogic Server

This appendix describes `TextFormatter` classes, which provide methods for generating localized versions of message text at run time.

This appendix includes the following sections:

- [About TextFormatter Classes](#)
- [Example of an Application Using a TextFormatter Class](#)

## C.1 About TextFormatter Classes

`TextFormatter` classes are generated by `weblogic.i18ngen` from simple message catalogs. These classes provide methods for generating localized versions of message text at run time.

## C.2 Example of an Application Using a TextFormatter Class

The following is an example of a `Hello_World` application, its simple message catalog, and the `TextFormatter` class generated for the catalog.

### **Example C-1 Example of a Simple Message Catalog**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog
  i18n_package="examples.i18n.simple"
  subsystem="I18N"
  version="1.0"
  >
  <message
    messageid="HELLO_WORLD"
    datelastchanged="967575717875"
    method="helloWorld()"
    >
    <messagebody>
      Hello World!
    </messagebody>
  </message>
  <!-- -->
  <message
    messageid="HELLO_AGAIN"
    datelastchanged="967575717804"
```

```

        method="helloAgain()"
    >
    <messagebody>
        Hello again
    </messagebody>
</message>
<!-- -->
<message
    messageid="NTH_HELLO"
    datelastchanged="967575770971"
    method="nthHello(int count)"
    >
    <messagebody>
        This is hello number {0,number}.
    </messagebody>
</message>
<!-- -->
<message
    messageid="VERSION"
    datelastchanged="967578656214"
    method="version(String version)"
    >
    <messagebody>
        Catalog version: {0}
    </messagebody>
</message>
<!-- -->
<message
    messageid="I18N_PACKAGE"
    datelastchanged="967578691394"
    method="i18nPackage(String pkg)"
    >
    <messagebody>
        I18n Package: {0}
    </messagebody>
</message>
<!-- -->
<message
    messageid="L10N_PACKAGE"
    datelastchanged="967578720156"
    method="l10nPackage(String pkg)"
    >
    <messagebody>
        L10n Package: {0}
    </messagebody>
</message>
<!-- -->
<message
    messageid="SUBSYSTEM"
    datelastchanged="967578755587"
    method="subSystem(String sub)"
    >
    <messagebody>
        Catalog subsystem: {0}
    </messagebody>
</message>
</message_catalog>

```



The following is an example of an application using the HelloWorld catalog. The example shows various ways of internationalizing an application using simple message catalogs.

**Example C-2 Example of an Application Using the HelloWorld Catalog**

```
package examples.i18n.simple;
import java.util.Locale;
import java.text.MessageFormat;
import weblogic.i18n.Localizer;
import weblogic.i18ntools.L10nLookup;

/**
 * This example shows various ways of internationalizing an application
 * using simple message catalogs.
 * <p>
 * Usage: java examples.i18n.simple.HelloWorld [lang [country]]
 * <p>
 * lang is a 2 character ISO language code. e.g. "en"
 * country is a 2 character ISO country code. e.g. "US"
 * <p>
 * Usage of any of the languages supported by this example presumes
 * the existence of the appropriate OS localization software and character
 * encodings.
 * <p>
 * The example comes with catalogs for English (the default) and French.
 * The catalog source is in the following files, and were built
 * using the catalog editing utility, weblogic.i18ntools.gui.MessageEditor.
 * <p>
 * <pre>
 * English(base language)      ../msgcat/HelloWorld.xml
 * French                      ../msgcat/fr/FR/HelloWorld.xml
 * </pre>
 * <p>
 * To build this example run the bld.sh(UNIX) or bld.cmd (NT) scripts from
 * the examples/i18n/simple directory. CLIENT_CLASSES must be set up and
 * needs to be in the classpath when running the example.
 */

public final class HelloWorld {

    public static void main(String[] argv) {
        /*
         * The easiest method for displaying localized text is to
         * instantiate the generated formatter class for the HelloWorld catalog.
         * This class contains convenience methods that return localized text for
         * each message defined in the catalog. The class name is
         * the catalog name followed by "TextFormatter".
         *
         * Typically, you would use the default constructor to obtain
         * formatting in the current locale. This example uses a locale
         * based on arguments to construct the TextFormatter.
         */
        Locale lcl;
        if (argv.length == 0) { // default is default locale for JVM
            lcl = Locale.getDefault();
        }
        else {
            String lang = null;
            String country = null;

```

```

        //get the language code
        lang = argv[0];
        if (argv.length >= 2) { // get the country code
            country = argv[1];
        }
        lcl = new Locale(lang, country);
    }
    /*
     * Get formatter in appropriate locale.
     */
    HelloWorldTextFormatter fmt = new HelloWorldTextFormatter(lcl);
    fmt.setExtendedFormat(true);
    /*
     * Print the text in the current locale.
     */
    System.out.println(fmt.helloWorld());
    /*
     * Alternatively, text can be accessed and formatted manually. In this
     * case you must obtain the Localizer class for the catalog. The
     * Localizer class is formed from the l10n_package attribute in the
     * catalog, the catalog name, and the string "TextLocalizer".
     */
    Localizer l10n = L10nLookup.getLocalizer
        (lcl, "examples.i18n.simple.HelloWorldTextLocalizer");
    System.out.println(l10n.get("HELLO_AGAIN"));
    /*
     * If the message accepts arguments, they can be passed to the
     * method defined for the message.
     */
    System.out.println(fmt.nthHello(3));
    /*
     * If using the manual method, you must manually apply the argument to
     * the text using the MessageFormat class.
     */
    String text = l10n.get("NTH_HELLO");
    Object[] args = {new Integer(4)};
    System.out.println(MessageFormat.format(text, args));
    /*
     * The Localizer class also provides methods for accessing catalog
     * information.
     */
    System.out.println(fmt.version(l10n.getVersion()));
    System.out.println(fmt.l10nPackage(l10n.getL10nPackage()));
    System.out.println(fmt.i18nPackage(l10n.getI18nPackage()));
    System.out.println(fmt.subSystem(l10n.getSubSystem()));
    }
}

```

The following listing shows an example of the generated `TextFormatter` for the `HelloWorld` catalog.

**Example C-3 Example of Generated TextFormatter Class for the HelloWorld Catalog**

```

package examples.i18n.simple; import java.text.MessageFormat;
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
import weblogic.i18n.Localizer;
import weblogic.i18ntools.L10nLookup;
public class HelloWorldTextFormatter {

```

```

private Localizer l10n;
private boolean format=false;
// constructors
public HelloWorldTextFormatter() {
    l10n = L10nLookup.getLocalizer(Locale.getDefault(),
"examples.i18n.simple.HelloWorldTextLocalizer");
}
public HelloWorldTextFormatter(Locale l) {
    l10n =
L10nLookup.getLocalizer(l,"examples.i18n.simple.HelloWorldTextLocalizer");
}
public static HelloWorldTextFormatter getInstance() {
    return new HelloWorldTextFormatter();
}
public static HelloWorldTextFormatter getInstance(Locale l) {
    return new HelloWorldTextFormatter(l);
}
public void setExtendedFormat(boolean fmt) {
    format = fmt;
}
public boolean getExtendedFormat() { return format;
/**
 * Hello World!
 */
public String helloWorld() {
    String fmt = "";
    String id = "HELLO_WORLD" ;
    String subsystem = "I18N" ;
    Object [] args = { };
    String output = MessageFormat.format(l10n.get(id) , args);
    if (getExtendedFormat()) {
        DateFormat dformat = DateFormat.getDateInstance(DateFormat.MEDIUM,
DateFormat.LONG);
        fmt = "<"+dformat.format(new Date())+"><"+subsystem+"><"+id+"> ";
    }
    return fmt+output;
}
/**
 * Hello again
 */
public String helloAgain() {
    String fmt = "";
    String id = "HELLO_AGAIN" ;
    String subsystem = "I18N" ;
    Object [] args = { };
    String output = MessageFormat.format(l10n.get(id) , args);
    if (getExtendedFormat()) {
        DateFormat dformat = DateFormat.getDateInstance(DateFormat.MEDIUM,
DateFormat.LONG);
        fmt = "<"+dformat.format(new Date())+"><"+subsystem+"><"+id+">";
    }
    return fmt+output;
}
/**
 * This is hello number {0,number}.
 */
public String nthHello(int arg0) {
    String fmt = "";
    String id = "NTH_HELLO" ;
    String subsystem = "I18N" ;

```

```

        Object [] args = { new Integer(arg0) };
        String output = MessageFormat.format(l10n.get(id) , args);
        if (getExtendedFormat()) {
            DateFormat dformat = DateFormat.getDateInstance(DateFormat.MEDIUM,
DateFormat.LONG);
            fmt = "<" + dformat.format(new Date()) + "><" + subsystem + "><" + id + ">";
        }
        return fmt + output;
    }
}

/**
 * Catalog version: {0}
 */
public String version(String arg0) {
    String fmt = "";
    String id = "VERSION" ;
    String subsystem = "I18N" ;
    Object [] args = { arg0 };
    String output = MessageFormat.format(l10n.get(id) , args);
    if (getExtendedFormat()) {
        DateFormat dformat = DateFormat.getDateInstance(DateFormat.MEDIUM,
DateFormat.LONG);
        fmt = "<" + dformat.format(new Date()) + "><" + subsystem + "><" + id + ">";
    }
    return fmt + output;
}

/**
 * I18n Package: {0}
 */
public String i18nPackage(String arg0) {
    String fmt = "";
    String id = "I18N_PACKAGE" ;
    String subsystem = "I18N" ;
    Object [] args = { arg0 };
    String output = MessageFormat.format(l10n.get(id) , args);
    if (getExtendedFormat()) {
        DateFormat dformat = DateFormat.getDateInstance(DateFormat.MEDIUM,
DateFormat.LONG);
        fmt = "<" + dformat.format(new Date()) + "><" + subsystem + "><" + id + ">";
    }
    return fmt + output;
}

/**
 * L10n Package: {0}
 */
public String l10nPackage(String arg0) {
    String fmt = "";
    String id = "L10N_PACKAGE" ;
    String subsystem = "I18N" ;
    Object [] args = { arg0 };
    String output = MessageFormat.format(l10n.get(id) , args);
    if (getExtendedFormat()) {
        DateFormat dformat = DateFormat.getDateInstance(DateFormat.MEDIUM,
DateFormat.LONG);
        fmt = "<" + dformat.format(new Date()) + "><" + subsystem + "><" + id + ">";
    }
    return fmt + output;
}

/**
 * Catalog subsystem: {0}
 */

```

```
public String subSystem(String arg0) {
    String fmt = "";
    String id = "SUBSYSTEM" ;
    String subsystem = "I18N" ;
    Object [] args = { arg0 };
    String output = MessageFormat.format(l10n.get(id) , args);
    if (getExtendedFormat() {
        DateFormat dformat = DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
DateFormat.LONG);
        fmt = "<" + dformat.format(new Date()) + "><" + subsystem + "><" + id + ">";
    }
    return fmt + output;
}
}
```



---

---

# Logger Class Reference for WebLogic Server

This appendix describes `Logger` classes, which provide the interface to WebLogic Server logging, and provides an example of a message catalog and its corresponding `Logger` class.

This appendix includes the following sections:

- [About Logger Classes](#)
- [Example of a Generated Logger Class](#)

## D.1 About Logger Classes

The classes generated by `i18ngen` are known as `Logger` classes. `Logger` classes provide the interface to WebLogic Server logging. For catalog `Xyz.xml`, a `Logger` class `XyzLogger` is generated.

The `Logger` class provides methods to log all messages defined in a catalog to the WebLogic server log. The methods included are the same as those defined in the associated catalog. If the catalog specifies the `loggables` attribute as `true`, then `Loggable` methods are also generated for each message.

For more information, see [Appendix B, "Loggable Object Reference for WebLogic Server."](#)

## D.2 Example of a Generated Logger Class

[Example D-1](#) contains an example of a generated logger class.

### **Example D-1 Example of Generated Logger Class**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog
  i18n_package="examples.i18n.logging"
  l10n_package="examples.i18n.logging"
  subsystem="I18N"
  version="1.0"
  baseid="600000"
  endid="610000"
  loggables="true"
  >
</logmessage
```

```

        messageid="600000"
        method="logEntry()"
        severity="info"
    >
    <messagebody>Starting I18nLog example...</messagebody>
    <messagedetail></messagedetail>
    <cause></cause>
    <action></action>
</logmessage>
<logmessage
    messageid="600001"
    method="testArgs(String name,int cnt)"
    severity="debug"
    >
    <messagebody>Class {0} started with {1,number} arguments.</messagebody>
    <messagedetail></messagedetail>
    <cause></cause>
    <action></action>
</logmessage>
<logmessage
    messageid="600002"
    method="logTrace(Throwable t)"
    severity="error"
    stacktrace="true"
    >
    <messagebody>This message is followed by a trace</messagebody>
    <messagedetail></messagedetail>
    <cause></cause>
    <action></action>
</logmessage>
<logmessage
    messageid="600003"
    method="logNoTrace(Throwable t)"
    severity="warning"
    stacktrace="false"
    >
    <messagebody>This message is not followed by a trace, but we can insert its
text : {0}</messagebody>
    <messagedetail></messagedetail>
    <cause></cause>
    <action></action>
</logmessage>
<logmessage
    messageid="600004"
    method="getId()"
    severity="info"
    >
    <messagebody>This message's id will be in the next message</messagebody>
    <messagedetail>A message can contain additional detailed
information.</messagedetail>
    <cause>This message is displayed on purpose</cause>
    <action>Nothing to do, the example is working</action>
</logmessage>
<logmessage
    messageid="600005"
    method="showId(String id)"
    severity="info"
    >
    <messagebody>The previous message logged had message id {0}</messagebody>
    <messagedetail></messagedetail>

```



```

        <cause></cause>
        <action></action>
    </logmessage>
</message_catalog>

```

[Example D-2](#) shows the corresponding Java source code generated by `weblogic.i18ngen`.

### **Example D-2 Example of Generated Logger Class**

```

package examples.i18n.logging;

import weblogic.logging.MessageLogger;
import weblogic.logging.Loggable;
import java.util.MissingResourceException;
public class I18nLogLogger
{
    /**
     * Starting I18nLog example...
     * @exclude
     *
     * messageid: 600000
     * severity: info
     */
    public static String logEntry() {
        Object [] args = { };
        MessageLogger.log(
            "600000",
            args,
            "examples.i18n.logging.I18nLogLogLocalizer");
        return "600000";
    }
    public static Loggable logEntryLoggable() throws MissingResourceException {
        Object[] args = { };
        return new Loggable("600000", args);
    }
    /**
     * Class {0} started with {1,number} arguments.
     * @exclude
     *
     * messageid: 600001
     * severity: debug
     */
    public static String testArgs(String arg0, int arg1) {
        Object [] args = { arg0, new Integer(arg1) };
        MessageLogger.log(
            "600001",
            args,
            "examples.i18n.logging.I18nLogLogLocalizer");
        return "600001";
    }
    public static Loggable testArgsLoggable(String arg0, int arg1) throws
MissingResourceException {
        Object[] args = { arg0, new Integer(arg1) };
        return new Loggable("600001", args);
    }
    /**
     * This message is followed by a trace
     * @exclude
     *

```

```

    * messageid: 600002
    * severity: error
    */
    public static String logTrace(Throwable arg0) {
        Object [] args = { arg0 };
        MessageLogger.log(
            "600002",
            args,
            "examples.i18n.logging.I18nLogLogLocalizer");
        return "600002";
    }
    public static Loggable logTraceLoggable(Throwable arg0) throws
MissingResourceException {
        Object[] args = { arg0 };
        return new Loggable("600002", args);
    }
    /**
    * This message is not followed by a trace, but we can insert its text : {0}
    * @exclude
    *
    * messageid: 600003
    * severity: warning
    */
    public static String logNoTrace(Throwable arg0) {
        Object [] args = { arg0 };
        MessageLogger.log(
            "600003",
            args,
            "examples.i18n.logging.I18nLogLogLocalizer");
        return "600003";
    }
    public static Loggable logNoTraceLoggable(Throwable arg0) throws
MissingResourceException {
        Object[] args = { arg0 };
        return new Loggable("600003", args);
    }
    /**
    * This message's id will be in the next message
    * @exclude
    *
    * messageid: 600004
    * severity: info
    */
    public static String getId() {
        Object [] args = { };
        MessageLogger.log(
            "600004",
            args,
            "examples.i18n.logging.I18nLogLogLocalizer");
        return "600004";
    }
    public static Loggable getIdLoggable() throws MissingResourceException {
        Object[] args = { };
        return new Loggable("600004", args);
    }
    /**
    * The previous message logged had message id {0}
    * @exclude
    *
    * messageid: 600005

```

```

    * severity:  info
    */
    public static String showId(String arg0) {
        Object [] args = { arg0 };
        MessageLogger.log(
            "600005",
            args,
            "examples.i18n.logging.I18nLogLogLocalizer");
        return "600005";
    }
    public static Loggable showIdLoggable(String arg0) throws
MissingResourceException {
        Object[] args = { arg0 };
        return new Loggable("600005", args);
    }
}

```

[Example D-3](#) shows an example application that uses the `weblogic.i18nLog` (internationalized (I18n) logging interfaces). The example logs an informational message.

### **Example D-3 Example of Application Using i18nLog**

```

package examples.i18n.logging;

import java.util.Locale;

import weblogic.i18n.Localizer;
import weblogic.i18ntools.L10nLookup;
import weblogic.logging.Loggable;

/**
 * This example shows how to use the internationalized (I18n) logging interfaces.
 * <p>
 * usage: java examples.i18n.logging.I18nLog
 * <p>
 * Build procedure: run bld.sh (UNIX) or bld.cmd (NT). These scripts
 * process the I18nLog.xml catalog, producing the logging class,
 * <tt>examples.i18n.logging.I18nLogLogger</tt>. This class contains static
 * methods for logging messages to the WLS server log. The methods
 * and arguments are defined in the I18nLog.xml catalog. This example also
 * uses a simple message catalog, I18nSimple.xml.
 */

public class I18nLog {

    public I18nLog() {}

    public static void main(String[] argv) {
        /**
         * This call just logs an info message. There are no arguments defined
         * for this method.
         *
         * This also shows how to use the Loggable form of the method.
         */

        Loggable ll = I18nLogLogger.logEntryLoggable();
        ll.log();
    }
}

```

```

System.out.println(l1.getMessage());

/**
 * Here's an example of a message including a variety
 * of arguments.
 */
I18nLogLogger.testArgs(I18nLog.class.getName(), argv.length);
/**
 * If a Throwable is passed then it will result in a stack trace
 * being logged along with the method by default.
 */
Throwable t = new Throwable("Test with stack trace");
I18nLogLogger.logTrace(t);
/**
 * Messages can optionally be defined to not log a stack trace.
 */
I18nLogLogger.logNoTrace(t);
/**
 * The logger methods return the message id for applications
 * that want to do more than just log these messages.
 */
String messageId = I18nLogLogger.getId();
I18nLogLogger.showId(messageId);
/**
 * The message id can be used to obtain the different attributes
 * of a message. The L10nLookup object provides access to the catalogs
 * via Localizer classes. Localizers provide the access to individual
 * messages. Each log message catalog has two Localizers: one for
 * general message information and one for the detailed attributes.
 *
 * The basic Localizer provides access to catalog information:
 *   Version
 *   L10n Package - package for catalog data
 *   I18n Package - package for Logger methods
 *   Subsystem - catalog subsystem
 * For each message it also provides:
 *   Severity: debug, info, warning, error
 *   Message Body - the message text
 *   Stack option - whether to log a stack trace
 *
 * First get to the L10nLookup properties, then use them to get the
 * Localizers for the message.
 */
L10nLookup l10n = L10nLookup.getL10n();
/**
 * This returns the basic Localizer (arg 3 = false)
 */
Localizer lcl = l10n.getLocalizer(messageId, Locale.getDefault(), false);
/**
 * This returns the detailed Localizer (arg 3 = true)
 */
Localizer lclDetail = l10n.getLocalizer(messageId, Locale.getDefault(), true);
/**
 * Use this application's simple message catalog to display the
 * log message catalog information
 */
I18nSimpleTextFormatter fmt = new I18nSimpleTextFormatter();
System.out.println(fmt.version(messageId, lcl.getVersion()));
System.out.println(fmt.l10nPackage(messageId, lcl.getL10nPackage()));
System.out.println(fmt.i18nPackage(messageId, lcl.getI18nPackage()));

```

```
System.out.println(fmt.subsystem(messageId, lcl.getSubSystem()));
System.out.println(fmt.severity(messageId, lcl.getSeverity(messageId)));
System.out.println(fmt.body(messageId, lcl.getBody(messageId)));
System.out.println(fmt.stack(messageId, lcl.getStackTrace(messageId)));
/**
 * Now for the detailed information.
 */
System.out.println(fmt.detail(messageId, lclDetail.getDetail(messageId)));
System.out.println(fmt.cause(messageId, lclDetail.getCause(messageId)));
System.out.println(fmt.action(messageId, lclDetail.getAction(messageId)));
}
}
```

