BEA WebLogic Server™

Using WebLogic Logging Services

Release 8.1 Beta
Document Revised: September 20, 2002

Using WebLogic Logging Services

Part Number	Document Revised	Software Version
N/A	September 20, 2002	BEA WebLogic Server Version 8.1 Beta

Contents

About This Document

This document describes how your application can write messages to the BEA WebLogic Server™ log files and listen for the log messages that WebLogic Server broadcasts. The document also outlines how you can use the WebLogic Server Administration Console to view log messages.

The document is organized as follows:

- Chapter 1, Writing Messages to the WebLogic Server Log
- Chapter 2, Viewing the WebLogic Server Logs
- Chapter 3, Listening for Messages from the WebLogic Server Log

Audience

This document is written for application developers who want to build Web applications or other Java 2 Platform, Enterprise Edition (J2EE) components that run on WebLogic Server. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. Specifically, "Logging" in the Administration Console Online Help describes how to configure log files that a WebLogic Server generates, and the Internationalization Guide describes how to set up message catalogs that your application can use.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA

WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

Convention	Item

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

Convention	Item
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Overview of WebLogic Logging Services

The WebLogic Server logging services include facilities for writing, viewing, and listening for log messages. While WebLogic Server subsystems use these services to provide information about events such as the deployment of new applications or the failure of one or more subsystems, your application can also use them to communicate its status and respond to specific events. For example, 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. In addition, you can configure your application to listen for a log message from a specific subsystem and to respond appropriately.

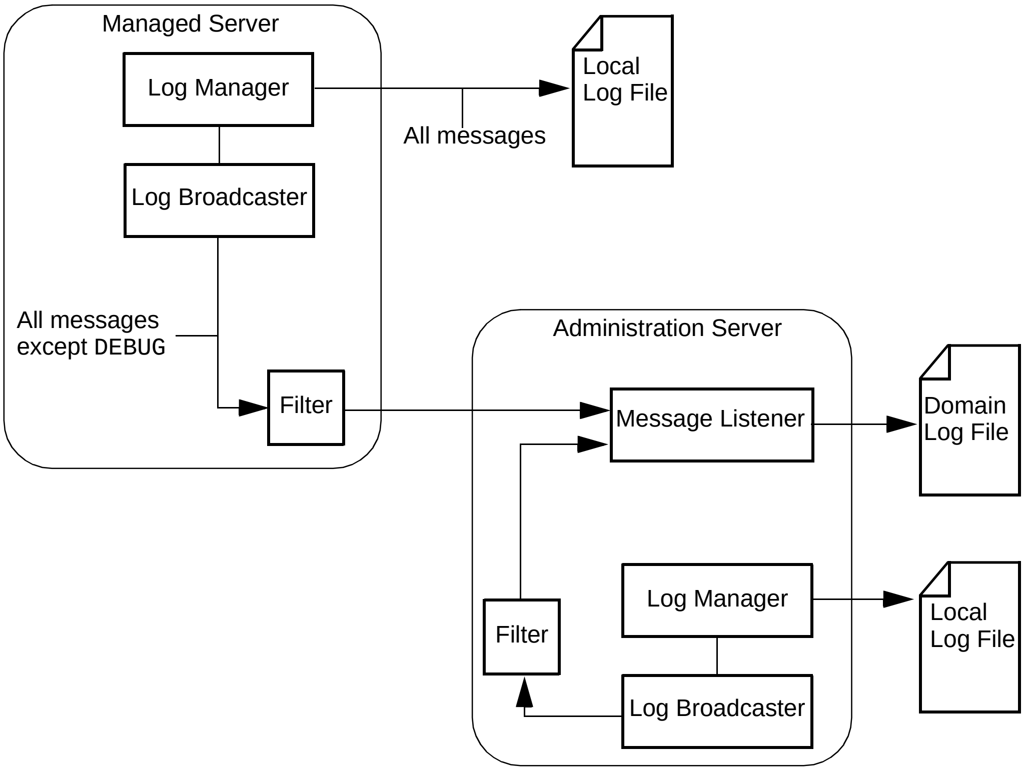
Because each WebLogic Server administration domain can run concurrent, multiple instances of WebLogic Server, the logging services collect messages that are generated on multiple server instances into a single, domain-wide message log. You can use this domain-wide message log to see the overall status of the domain.

To provide this overview of a domain's status, each server instance broadcasts its log messages as Java Management Extensions (JMX) notifications. A server broadcasts all messages and message text except for the following:

- Messages of the `DEBUG` severity level.
- Any stack traces that are included in a message.

The Administration Server listens for a subset of these messages and writes them to the domain log file. To listen for these messages, the Administration Server registers a JMX listener with each Managed Server. By default, the listener includes a filter that allows only messages of severity level `ERROR` and higher to be forwarded to the Administration Server. (See Figure 1-1.)

Figure 1-1 WebLogic Server Logging Services



The remainder of this document describes how your application can write and listen for messages, and how you can view them through the WebLogic Server Administration Console.

CHAPTER

2 Writing Messages to the WebLogic Server Log

The following sections describe how you can facilitate the management of your application by writing log messages to the WebLogic Server log files:

- "Using the I18N Message Catalog Framework: Main Steps" on page 2-2
- "Using the NonCatalogLogger APIs" on page 2-9
- "Using GenericServlet" on page 2-13

In addition, this section includes the following sections:

- "Writing Messages from a Remote Application" on page 2-14
- "Writing Debug Messages" on page 2-15

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. The framework is ideal for applications that need to localize the language in their log messages, but even for those applications that do not need to localize, it provides a rich, flexible set of tools for communicating status and output.

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

- Step 1: Create Message Catalogs
- Step 2: Compile Message Catalogs
- Step 3: Use Messages from Compiled Message Catalogs

Step 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 optional, 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, refer to the following:

- Using the BEA WebLogic Server Message Editor in the *BEA WebLogic Server Internationalization Guide*.
- Using Message Catalogs with BEA WebLogic Server in the *BEA WebLogic Server Internationalization Guide*.

4. When you finish adding messages in the Message Editor, select File→Save Catalog. 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.

Step 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.

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
   ```

   where:

   - *`targetdirectory`* is 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`* specifies 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 Step 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 l10ngen -d` *`targetdirectory source-files`*

   where:

- *`targetdirectory`* is the root directory in which you want the `l10ngen` 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 `l10n_package` values in the message catalog.

- *`source-files`* 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 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, refer to Using the BEA WebLogic Server Internationalization Utilities in the *BEA WebLogic Server Internationalization Guide*.

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`.

Listing 2-1 shows the contents of the message catalog.

Listing 2-1 Sample Message Catalog

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls710/dtd/msgcat.dtd">

<message_catalog
   i18n_package="com.xyz.msgcat"
   l10n_package="com.xyz.msgcat.l10n"
   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`.

Listing 2-2 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/wls710/dtd/l10n_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:
   `set CLASSPATH=%CLASSPATH%;c:\MessageOutput`
2. Enter
   `java 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`

Step 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 Step 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 Listing 2-1 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 the display the message. For example, the following segment of the message catalog in Listing 2-1 shows the name of the method:

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

Listing 2-3 illustrates a simple class that calls this `startup` method.

Listing 2-3 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.

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. We do not recommended using 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 in this topic, "Writing Messages from a Remote Application" on page 2-14, 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/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/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, refer to “Writing Debug Messages” on page 2-15.

- `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 error log. For information on the `NonCatalogLogger` APIs, refer to the `weblogic.logging.NonCatalogLogger` Javadoc.

Listing 2-4 illustrates a servlet that uses `NonCatalogLogger` APIs to write messages of various severity levels to the WebLogic Server log.

Listing 2-4 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 (< >).

Listing 2-5 NonCatalogLogger Output

```
####<Jun 26, 2002 12:04:21 PM EDT> <Info> <MyApplication> <MyHost>
<examplesServer> <ExecuteThread: '10' for queue: 'default'> <kernel identity> <>
<000000> <Application started.>

####<Jun 26, 2002 12:04:23 PM EDT> <Warning> <MyApplication> <MyHost>
<examplesServer> <ExecuteThread: '10' for queue: 'default'> <kernel identity> <>
<000000> <Can't establish connections. >

javax.naming.CommunicationException.  Root exception is
java.net.ConnectException: t3://localhost:8000: Destination unreachable; nested
exception is:

...
```

Table 2-1 describes all of the fields that `NonCatalogLogger` log messages can contain.

Table 2-1 NonCatalogLogger Log Message Format

Field	Description
Localized Timestamp	Date and time when message originated, including the year, month, day of month, hours, minutes and seconds. For example, `<Jun 26, 2002 12:04:21 PM EDT>`.
Severity	One of the following severity values, which corresponds to the type of method that you used to generate the message: `Info`, `Warning`, `Error`, `Debug`
Subsystem	Indicates the source of the message. This is the string that you supply for the `NonCatalogLogger` constructor.
Server Name Machine Name Thread ID	Identify the origins of the message. Log messages that are generated within a client JVM client do not include these fields. For example, if your application runs in a client JVM and it uses the WebLogic logging services, the messages that it generates and sends to the WebLogic Server log files will not include these fields.
User Id	User on behalf of whom the system was executing when the error was reported. Log messages that are generated within a client JVM client do not include this field.
TransactionId	Present only for messages logged within the context of a transaction.

Table 2-1 NonCatalogLogger Log Message Format

Field	Description
Message Id	A six-digit identifier for the message. The message ID for `NonCatalogLogger` messages is always `000000`.
Message text	The text that you supply for the `NonCatalogLogger` method.
ExceptionName	If the message is logging an Exception, this field contains the name of the Exception.

Using GenericServlet

The `javax.servlet.GenericServlet` servlet specification provides the following APIs that your servlets 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, refer to the J2EE Javadoc for `javax.servlet.GenericServlet` at http://java.sun.com/products/servlet/2.3/javadoc/javax/servlet/GenericServlet.html.

JSPs do not extend from `GenericServlet` and cannot use these APIs. If you want your JSPs to send messages to a log file, consider using the I18N message catalog services or `NonCatalogLogger` APIs.

Writing Messages from a Remote 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 a WebLogic Server log. Instead, the application's messages are written to the remote JVM's standard out.

If you want the WebLogic logging service to send these messages to a log file that the remote JVM maintains, include the following argument in the command that starts the remote 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 standard out as well as the remote JVM log file, include the following additional startup arguments:

```
-Dweblogic.StdoutEnabled=true
```

```
-Dweblogic.StdoutDebugEnabled=boolean
```

```
-Dweblogic.StdoutSeverityLevel = [64 | 32 | 16 | 8 | 4 | 2 | 1 ]
```

where *`boolean`* is either `true` or `false` and the numeric values for `StdoutSeverityLevel` correspond to the following severity levels:

INFO(64) WARNING(32), ERROR(16), NOTICE(8), CRITICAL(4), ALERT(2) and EMERGENCY(1).

Writing Messages from a Remote JVM to a File

A remote JVM can generate its own set of messages that communicate information about the state of the JVM itself. For example, you can configure a JVM to generate messages about garbage collection. By default, the JVM sends these messages to standard out. You cannot redirect these messages to the JVM's log file, but you can save them to a separate file. For more information, refer to "Redirecting JVM Messages to a File" in the Administration Console Online Help.

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 the WebLogic Server log file. (Unlike Log4j, which is a third-party logging service that enables you to dynamically exclude log messages based on level of severity, the WebLogic Server log includes all levels of messages that your application generates.)

You also can configure the WebLogic Server to send `DEBUG` messages to standard out. For more information refer to "Specifying Which Messages a Server Sends to Standard Out" in the *Administration Console Online Help*.

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 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
```

CHAPTER

3 Viewing the WebLogic Server Logs

The WebLogic Server Administration Console provides separate but similar log viewers for the local server log and the domain-wide message log. The log viewer can search for messages based on fields within the message. For example, it can find and display messages based on the severity, time of occurrence, user ID, subsystem, or the short description. It can also display messages as they are logged, or search for past log messages. (See Figure 3-1.)

Figure 3-1 Log Viewer

In addition to viewing messages from the Administration Console, you can specify which messages are sent to standard out. By default, only messages of `WARNING` or higher are sent to standard out.

For information about viewing, configuring, and searching message logs, refer to the following topics:

- Viewing Server Logs in the Administration Console Online Help
- Specifying Which Messages a Server Sends to Standard Out in the Administration Console Online Help
- Viewing the Domain Log in the Administration Console Online Help

CHAPTER

4 Listening for Messages from the WebLogic Server Log

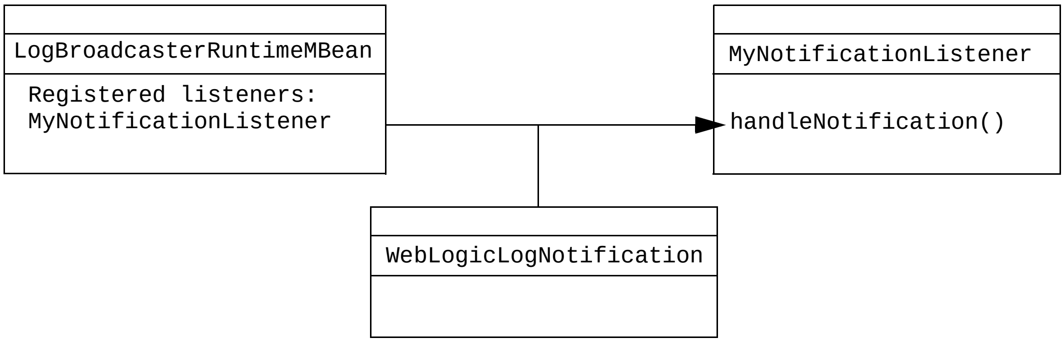
Each WebLogic Server instance broadcasts its log messages in the form of JMX notifications. The broadcast includes all messages (except those of the DEBUG severity level) that the WebLogic Server instance, its subsystems, and any applications write to the WebLogic Server log. An Administration Server listens for these notifications and places a subset of them in the domain-wide message log. (See Figure 1-1, “WebLogic Server Logging Services,” on page 1-6.)

Your application also can listen for log messages that are broadcast from a WebLogic Server instance. For example, your application can listen for a log message that signals the failure of a specific subsystem. Then your application can perform actions such as:

- E-mail the log message to the WebLogic Server administrator.
- Shut down or restart itself or its subcomponents.

To listen for these notifications, you create a notification listener and register it with the WebLogic Server broadcast MBean, `LogBroadcasterRuntimeMBean`. A **notification listener** is an implementation of the JMX `NotificationListener` interface. When `LogBroadcasterRuntimeMBean` emits a notification, it uses the registered listener’s `handleNotification` method to pass a `WebLogicLogNotification` object. (See Figure 4-1.)

Figure 4-1 WebLogic Broadcaster and Your Listener


LogBroadcasterRuntimeMBean
Registered listeners: MyNotificationListener
MyNotificationListener
handleNotification()
WebLogicLogNotification

A subsequent subsection, "WebLogicLogNotification Objects" on page 4-13, provides more information about `WebLogicLogNotification` objects

To enable your application to listen for notifications from a WebLogic Server log, complete the following tasks:

- Step 1: Create a Notification Listener
- Step 2: Register the Notification Listener
- Step 3: Create and Register a Notification Filter

Note: If your application runs outside a WebLogic Server JVM, it can listen for WebLogic Server log notifications, but it cannot use WebLogic logging services to broadcast messages.

Step 1: Create a Notification Listener

The steps that you follow to create a notification listener differ depending on whether your application runs within a WebLogic Server JVM.

This section contains the following subsections:

- Creating a Notification Listener for an Application that Runs Within a WebLogic Server JVM

- Creating a Notification Listener for a Remote Application

Creating a Notification Listener for an Application that Runs Within a WebLogic Server JVM

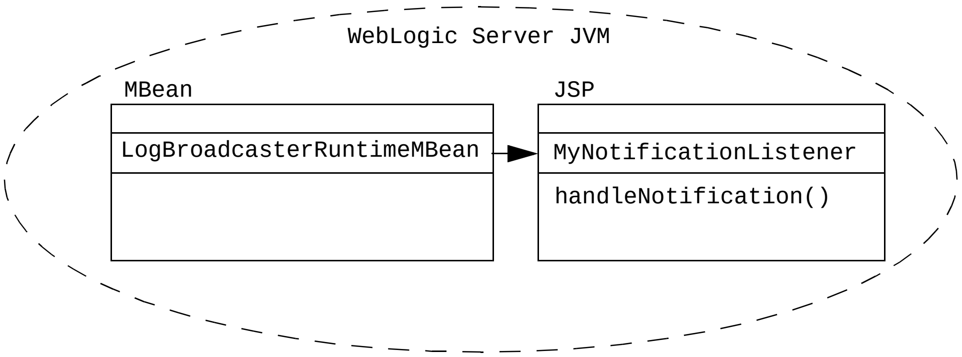
If your application runs within a WebLogic Server JVM, do the following:

1. Import the `javax.management.Notification.*` interfaces. Because WebLogic Server already includes these interfaces and requires them to be on the classpath, you need only to include an import statement in your class.
2. Create a class that implements `NotificationListener`. Your implementation must include the `NotificationListener.handleNotification()` method.

   For more information on `NotificationListener`, refer to the `javax.management.Notification` Javadoc, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html.

Figure 4-2 shows a system in which a JSP is running within a WebLogic Server JVM. The JSP listens for notifications from `LogBroadcasterRuntimeMBean`.

Figure 4-2 Listener for a Local JSP



Listing 4-1 provides an example notification listener for a local client. The listener uses `WebLogicLogNotification` getter methods to print all messages that it receives. For more information, refer to “WebLogicLogNotification Objects” on page 4-13.

Listing 4-1 Example Notification Listener for a Local Client

```
import javax.management.Notification;
import javax.management.NotificationListener;

...

public class MyNotificationListener implements
    NotificationListener {

...

public void handleNotification(Notification notification, Object handback) {
    WebLogicLogNotification wln = (WebLogicLogNotification)notification;
    System.out.println("WebLogicLogNotification");
    System.out.println(" type = " + wln.getType());
    System.out.println(" message id = " + wln.getMessageId());
    System.out.println(" server name = " + wln.getServername());
    System.out.println(" timestamp = " + wln.getTimeStamp());
    System.out.println(" message = " + wln.getMessage() + "\n");
}
```

Creating a Notification Listener for a Remote Application

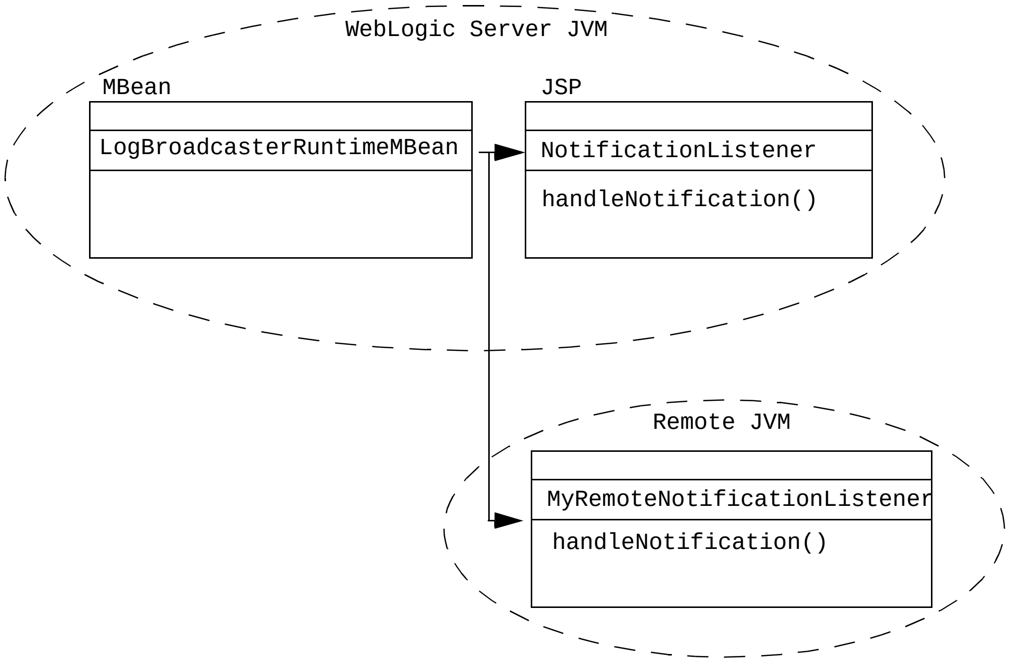
If your application resides outside of the WebLogic Server JVM, do the following:

1. Make sure that `WL_HOME/server/lib/weblogic_sp.jar` and `WL_HOME/server/lib/weblogic.jar` are in the application’s classpath.
2. Import the `javax.management.Notification.*` interfaces.
3. Create a class that implements `weblogic.management.RemoteNotificationListener`. `RemoteNotificationListener` MBean makes notifications available to remote applications via RMI by extending `javax.management.NotificationListener` and `java.rmi`.

   Your implementation must include the `RemoteNotificationListener.handleNotification()` method. For more information, refer to the `weblogic.management.RemoteNotificationListener` Javadoc.

Figure 4-3 shows a system in which a JSP runs in the WebLogic Server JVM and an application runs in a remote JVM. To listen for notifications, the JSP implements `NotificationListener` and the remote application implements `RemoteNotificationListener`.

Figure 4-3 Local JSP and Remote Application



Listing 4-2 provides an example notification listener for a remote client.

Listing 4-2 Example Notification Listener for a Remote Client

```
import javax.management.Notification;
import javax.management.NotificationListener;

import weblogic.management.RemoteNotificationListener;
import weblogic.management.logging.WebLogicLogNotification;

...

public class MyRemoteNotificationListener implements
    RemoteNotificationListener {
```

```
...

public void handleNotification(Notification notification, Object handback) {
    WebLogicLogNotification wln = (WebLogicLogNotification)notification;
}
```

Step 2: Register the Notification Listener

After you implement your notification listener, you must register it with `LogBroadcasterRuntimeMBean` on a WebLogic Server instance. Because each instance broadcasts its own notifications, you must register your notification listener on each WebLogic Server instance from which you want to receive notifications.

This section describes the code fragment that you use to register a listener. You can add this fragment to a class that runs when your client application starts, when a WebLogic Server instance starts, or whenever you want your application to receive notifications.

To register with the `LogBroadcasterRuntimeMBean` on a WebLogic Server instance, the code must do the following:

1. Import the following interfaces:

   ```
   javax.naming.Context
   javax.naming.InitialContext
   javax.naming.AuthenticationException
   javax.naming.CommunicationException
   javax.naming.NamingException
   weblogic.jndi.Environment
   weblogic.management.MBeanHome
   ```

2. Obtain the `MBeanServer` from `MBeanHome`. For more information, refer to Accessing WebLogic Server MBeans in the *Using WebLogic JMX Services* Guide.

3. Use the `addNotificationListener()` method of the `MBeanServer` to register your notification listener with `LogBroadcasterRuntimeMBean`.

Using the addNotificationListener API

The syntax for the `addNotificationListener` API is as follows:

```
MBeanServer.addNotificationListener(ObjectName name,
     NotificationListener listener,
     NotificationFilter filter,
     java.lang.Object handback)
```

Provide the following values:

- `name` is the object name of the WebLogic Server instance's `LogBroadcasterRuntimeMBean`. You can obtain the object name by doing one of the following:
  - Creating an instance `weblogic.management.WebLogicObjectName`. For more information, refer to the `WebLogicObjectName` Javadoc.
  - Looking up the `weblogic.management.runtime.LogBroadcasterRuntimeMBean` at runtime and calling `.getObjectName()`. For more information, refer to the `LogBroadcasterRuntimeMBean` Javadoc.
  - Using the `weblogic.Admin GET` command. For more information, refer to the GET command in the *WebLogic Server Command Line Reference*.
- `listener` is the instance of the Notification listener you created in "Step 1: Create a Notification Listener" on page 4-2.
- `filter` is a filter object. If filter is `null`, no filtering will be performed before handling notifications. The next section, "Step 3: Create and Register a Notification Filter" on page 4-11, describes creating and registering a filter object.
- `handback` is the context to be sent to the listener when a notification is broadcast.

Complete documentation for the `addNotificationListener` API is available in the Javadoc for `javax.management.MBeanServer`, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html.

Examples for Registering a Notification Listener

The following examples register the listener defined in Step 1: Create a Notification Listener. The examples in Listing 4-3 and Listing 4-4 do the following:

1. Use the `weblogic.management.Helper` API to obtain the server-specific `MBeanHome` interface for a server named `peach`. For more information about obtaining the `MBeanHome` interface, refer to Accessing WebLogic Server MBeans in the *Programming WebLogic Management Services with JMX* guide.
2. Use the `MBeanHome` interface to retrieve the corresponding `MBeanServer` interface.
3. Use a different method for retrieving the `LogBroadcasterRuntimeMBean` object name.
4. Instantiate the listener object defined in Step 1: Create a Notification Listener.
5. Use the `addNotificationListener` method of the `ServerMBean` interface to register the listener object with the `LogBroadcasterRuntimeMBean`.

Listing 4-3 uses `WebLogicObjectName` to construct the `LogBroadcasterRuntimeMBean` object name.

Listing 4-3 Using WebLogicObjectName

```
public void find(String host, int port, String username, String password,
                 String hostname, String myDomain, String myServer)
{
    String url = "t3://" + host + ":" + port;

    //Get the server’s MBeanHome interface.
     try {
          serverSpecificHome = (MBeanHome)Helper.getMBeanHome(
                                username, password, url, hostname);
     } catch (IllegalArgumentException iae) {
           System.out.println("Illegal Argument Exception: " + iae);
     }

    //Use MBeanHome to get the server’s MBeanServer interface.
    MBeanServer mServer = serverSpecificHome.getMBeanServer();

   //Construct the WebLogicObjectName of the server’s LogBroadcasterRuntimeMBean.
    WebLogicObjectName logBCOname = new WebLogicObjectName(
         "WebLogicLogBroadcaster","LogBroadcasterRuntime",myDomain,myServer);

    //Instantiate a listener object.
   MyRemoteNotificationListener myListener = new MyRemoteNotificationListener();

    //Register the listener.
    mServer.addNotificationListener( logBCOname,myListener,null,null );
}
```

Listing 4-4 uses `MBeanHome.getMBeanByClass` to retrieve the `LogBroadcasterRuntimeMBean` object name.

Listing 4-4 Using getObjectName()

```
public void find(String host, int port, String username, String password,
                 String hostname, String myDomain, String myServer)
{
    String url = "t3://" + host + ":" + port;

    //Get the server’s MBeanHome interface.
     try {
          serverSpecificHome = (MBeanHome)Helper.getMBeanHome(
                                username, password, url, hostname);
     } catch (IllegalArgumentException iae) {
            System.out.println("Illegal Argument Exception: " + iae);
     }

    //Use MBeanHome to get the server’s MBeanServer interface.
    MBeanServer mServer = serverSpecificHome.getMBeanServer();

    //Use getMBeanByClass to retrieve the object name.
    LogBroadcasterRuntimeMBean logBCOname = (LogBroadcasterRuntimeMBean)
       home.getMBeanByClass(
       Class.forName ("weblogic.management.runtime.LogBroadcasterRuntimeMBean")
       );

    //Instantiate a listener object.
   MyRemoteNotificationListener myListener = new MyRemoteNotificationListener();

    //Register the listener.
    mServer.addNotificationListener( logBCOname,myListener,null,null );
}
```

Listing 4-5 assumes that you used `weblogic.Admin GET` to retrieve the `LogBroadcasterRuntimeMBean` object name. It also illustrates the format of object names that `weblogic.Admin GET` returns.

Listing 4-5 Using weblogic.Admin GET

```
MyRemoteNotificationListener myListener = new MyRemoteNotificationListener();
MBeanServer mServer = home.getMBeanServer();

ObjectName logBCOname = new
ObjectName("mydomain:Location=myserver,Name=TheLogBroadcaster,Type=LogBroadcast
erRuntime");

mServer.addNotificationListener( logBCOname,myListener,null,null);
```

Step 3: Create and Register a Notification Filter

By default, the notification listener that you registered in the previous section listens for all notifications from `LogBroadcasterRuntimeMBean` and sends them to your application. You can configure the `LogBroadcasterRuntimeMBean` to send only the notifications that are pertinent to your application by creating and registering a filter. The filter determines whether a notification matches a set of criteria that you create, and the `LogBroadcasterRuntimeMBean` sends the notification only if the filter evaluates as true.

This section contains the following subsections:

- Creating and Registering a Filter
- WebLogicLogNotification Objects
- Example Notification Filter

Creating and Registering a Filter

To create and register a filter, do the following:

1. Import the following interfaces:

   ```
   import javax.management.Notification
   import javax.management.NotificationFilter
   ```

   Optionally import the following interface:
   `import weblogic.management.logging.WebLogicLogNotification`

   `WebLogicLogNotification` provides methods that you can use to get attributes of WebLogic log messages.

2. Create a serializable object that does the following:

   a. Implements `javax.management.NotificationFilter`.

   b. Searches a notification for a string.

To search a notification that has been cast as a `WebLogicLogNotification` object, you can use `WebLogicLogNotification` getter methods. For example, you can use the getter methods to get the message timestamp, severity, user ID, the name of the subsystem that generated the message, the message text, and other data. For more information, refer to WebLogicLogNotification Objects.

c. Uses a boolean to indicate whether the serializable object returns a true value.

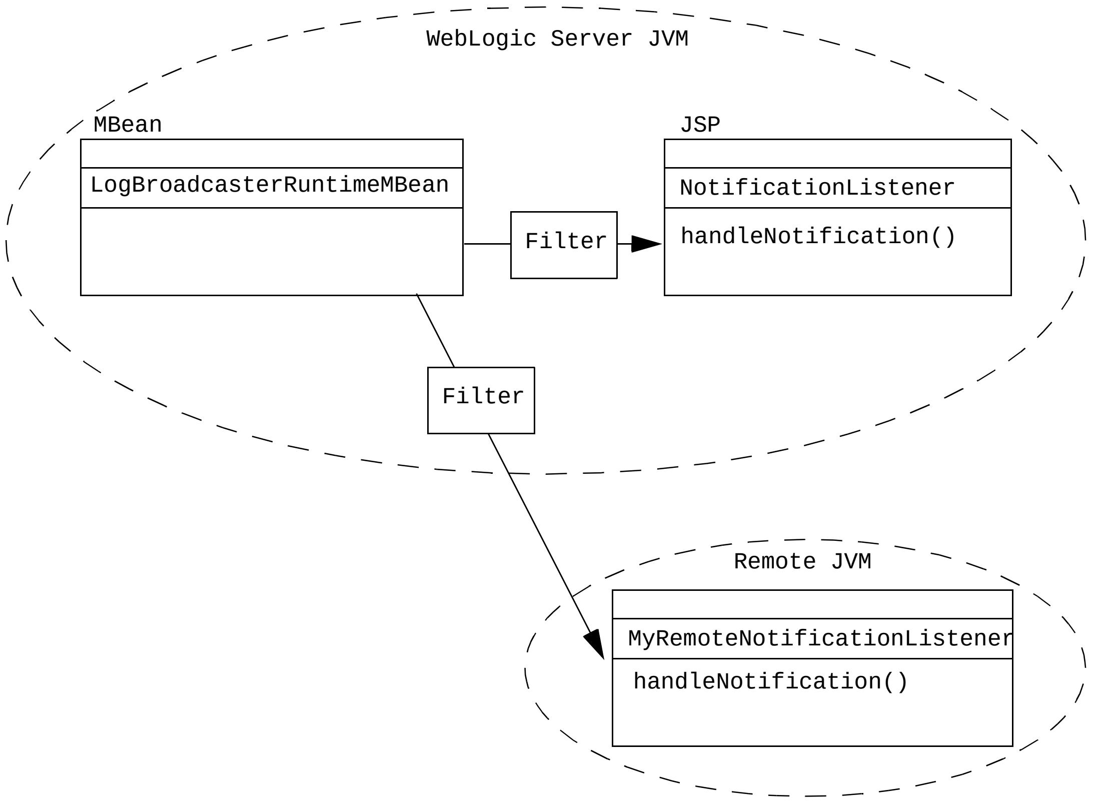
d. (Optional) Includes code that carries out an action depending on the value of the boolean. For example, your filter can use the JavaMail API to send e-mail to an administrator if a message is of severity `WARNING` or higher.

3. Use the `addNotificationListener` API to register the filter. For more information, refer to "Using the addNotificationListener API" on page 4-7.

Adding Filter Classes to the Server Classpath

If you create a filter for a listener that runs in a remote JVM, you must add the filter's classes to the classpath of the server instance from which you are listening for notifications. Although the listener runs in the remote JVM, to minimize the transportation of serialized data between the filter and the listener, the filter runs in the JVM of the server instance. (See Figure 4-4.)

Figure 4-4 Filters Run on WebLogic Server



WebLogicLogNotification Objects

All messages that a WebLogic Server generates are cast as `weblogic.management.logging.WebLogicLogNotification` objects. `WebLogicLogNotification` objects contain the following fields:

- Type—identifies the notification as required by the JMX specification. This field has the format:

  `weblogic.log.`*`subSystem.messageID`*

where *`subSystem`* indicates the subsystem or application that issued the log message this notification contains, and *`messageID`* indicates the internal WebLogic Server message ID.

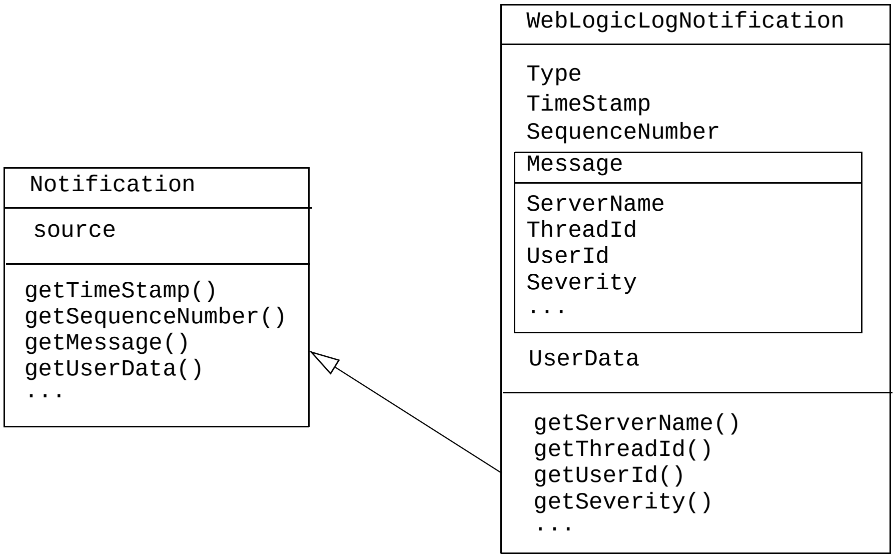
Note: For `NonCatalogLogger` messages, the message ID is always `000000`. All log messages that WebLogic Server subsystems generate have a message ID that begins with the string `BEA-`.

- Time stamp—indicates the time at which the log message causing this notification was generated by the server.
- Sequence number.
- Message—contains the log message.
- User data—the user data field is not currently used.

A `WebLogicLogNotification` inherits getter methods from `javax.management.Notification` and it provides one getter method for each field within the log message. (See Figure 4-5.)

You can use these getter methods to search or print the information within the `WebLogicLogNotification`. For more information, refer to the `weblogic.management.logging.WebLogicLogNotification` Javadoc.

Figure 4-5 WebLogicLogNotification Getter Methods



Example Notification Filter

Listing 4-6 provides an example `NotificationFilter` that uses the `WebLogicLogNotification.getType` method.

Listing 4-6 Example Notification Filter

```
import javax.management.Notification;
import javax.management.NotificationFilter;
import weblogic.management.logging.WebLogicLogNotification;

....

public class MyLogNotificationFilter implements NotificationFilter,
    java.io.Serializable {
```

```
    public MyLogNotificationFilter() {
        subsystem = "";
    }

    public boolean isNotificationEnabled(Notification notification) {
        if (!(notification instanceof WebLogicLogNotification)) {
             return false;
        }

        WebLogicLogNotification wln = (WebLogicLogNotification)notification;

        if (subsystem == null || subsystem.equals("")) {
             return true;
        }

        StringTokenizer tokens = new StringTokenizer(wln.getType(), ".");
        tokens.nextToken();
        tokens.nextToken();
        return (tokens.nextToken().equals(subsystem));
    }

    public void setSubsystemFilter(String newSubsystem) {
        subsystem = newSubsystem;
    }

}
```

Index