WebSphere™ Application Server

IBM

Using the JRas Message Logging and Trace Facility

Version 4.0

GC09-3952-00

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 17.

First Edition (March 2001)

Order publications through your IBM representative or through the IBM branch office serving your locality.

Contents

Figures

Tables

Using the JRas Message Logging and Trace Facility

Introduction

The IBM® JRas toolkit is a set of Java™ packages that enables developers to incorporate message logging and trace facilities into Java applications. Although JRas is a standalone product, it has been customized for use with the Standard, Advanced, and Enterprise (Component Broker) Editions of WebSphere™ Application Server. The WebSphere implementation of JRas integrates with the WebSphere run-time environment and system-management utilities (for instance, Advanced Edition’s Administrative Console and Component Broker’s System Manager). This document discusses the WebSphere implementation of JRas and using it to write WebSphere applications that log and manage application-specific messages and trace. Use of the non-WebSphere implementation of JRas is not discussed in this document.

Note: The non-WebSphere (base) implementation of JRas is not supported for use with WebSphere Application Server. The use of JRas with WebSphere is supported only with the WebSphere-specific JRas implementation and programming model discussed in this document.

Overview of messages and trace

Applications often need to provide information about their internal operations to users, system administrators, programmers, and other interested parties. This information is typically provided as text that can be sent to a console or terminal, written to a log file, directed to a standard output or error device, or all three. The JRas toolkit divides informational text into the following two categories:

- *Messages*, consisting of information about the application that is brief, clear, and meaningful to an end user. An example of a message is a string indicating that the application started successfully. Messages are generated by default; they are not normally suppressed. Messages can be localized; that is, the message catalogs can be translated into various national language versions, and messages can be displayed in the user’s preferred language.
- *Trace*, consisting of detailed technical information about the current state of one or more of the application’s internal data structures, including summaries of all objects in those data structures. Trace information is meant for use by developers and support personnel when debugging applications; it is not generally intended for use by end users. An example of trace information is a string listing an error, the time at which the error occurred, the thread in which the error occurred, the method that was being executed

when the error occurred, and a description of the error. Trace information is not normally generated by applications and is enabled only to help resolve specific problems, because the creation of trace information consumes system resources beyond the application's normal requirements. Trace is not localizable; that is, it cannot be translated into national language versions.

The JRas packages implement objects called *loggers*, *handlers*, *formatters*, and *managers* to provide messaging and trace capabilities. These objects are described in the following list.

- *Loggers* are the primary objects with which the application code interacts.
- *Handlers* receive data that is to be logged from a logger.
- *Formatters* are objects invoked by handlers to format data.
- *Managers* provide methods to predefine and manage logger, handler, and formatter configurations. These configurations can be kept in a persistent data store. Using managers simplifies programming with JRas; when a manager is used to obtain a logger, the manager retrieves the logger's configuration data, creates the logger and populates it with the correct handlers, performs any other needed tasks, and returns the configured logger to the caller. The Manager class provided with WebSphere is WebSphere specific and cannot be used with generic JRas implementations. Using this class to create and manage WebSphere JRas objects ensures that all derived objects (loggers, handlers, and formatters) conform to the requirements of the WebSphere JRas implementation.

To view message and trace text, you must read the appropriate log files. WebSphere currently logs all messages to single-level log files; that is, application messages and run-time messages are written to the same log file. It is recommended that you monitor the size of the log files and increase the allowable size of the files depending on the number of messages written to the log. WebSphere also logs all trace events, whether application trace or run-time trace, to the same trace log file. All editions of WebSphere Application Server provide facilities to view message and trace logs; see the documentation for your edition of WebSphere for more information.

The WebSphere JRas programming model

This section discusses the supported model for programming with JRas in WebSphere Application Server.

In WebSphere, you create and manage JRas loggers and managers by using the Manager class of the com.ibm.websphere.ras package. The Manager class provides mechanisms to obtain JRas message and trace loggers that are integrated with WebSphere; it also provides the ability to group trace loggers into logical groups. The basic process for creating JRas objects is to retrieve a reference to the JRas manager by using the getManager method of the com.ibm.websphere.ras.Manager class, then to retrieve message and trace loggers by using methods on the returned manager. See “Creating manager and logger instances” on page 8 for sample code illustrating this process.

The retrieved loggers are implementations of the RASIMessageLogger and RASITraceLogger interfaces. You then program to these interfaces, both of which are derived from the RASILogger interface. The loggers are stateful objects with their states tied to an existing Java Virtual Machine (JVM) and run-time instance. These interfaces are discussed in “Using loggers” on page 9.

Note: Although loggers implement the Java java.io.Serializable interface, they must not be serialized.

Naming and managing loggers

This section discusses considerations for naming and managing loggers.

WebSphere JRas loggers have no predefined granularity or scope. An application consisting of many different classes can be instrumented by using a single logger, can be subdivided into several components with a logger for each component, or can have a logger for each class.

Loggers are named objects; the manager maintains a hierarchical name space of loggers, with separate name spaces for message loggers and trace loggers. For each unique logger name, the logger instance is created on the first request to the manager and the same instance is returned on subsequent calls. The following recommendations apply to naming loggers:

- To prevent name-space conflicts, it is recommended that a dot-separated, fully qualified class name be used to name each logger.
- It is recommended that the full logger name reflect the name of the class that retrieves the logger from the manager.
- Application developers are responsible for ensuring that the logger names used by an application do not conflict with names in use by the WebSphere run time; using full logger names based on retrieval class names automatically provides this assurance.
- Because of potential name-space conflicts and limitations in the size of the name space, it is recommended that any given class have no more than one message logger or trace logger associated with it.
- The name `ORBRas` is reserved for use by the WebSphere run time. Do not use this name in WebSphere applications that use JRas.

The WebSphere run time and system-management utilities enable you to enable and disable trace at any level of the name-space hierarchy. Changing the trace state at any level of the hierarchy automatically makes the same state change for all child levels. For instance, enabling trace at the middle level of a hierarchy automatically enables trace for all levels below the middle level.

Trace loggers can be combined into logical sets called *groups* to track events across various components of an application. For example, if an application contains three different components, you can create a group that includes trace loggers from each component, thereby providing a way to trace the flow of a particular function across all three components. Application developers must provide group names that are unique to the application and that do not conflict with other group names in the name space, including names used by the WebSphere run time.

JRas objects are managed by the WebSphere run time. When a logger is created, the JRas manager queries the WebSphere system-management utility to determine the initial state for the logger's mask. The state of the mask is updated dynamically in accordance with settings provided to the system-management utility. The default initial states for the different types of loggers are as follows:

- For message loggers, the default initial state is always for logging to be enabled to the logger's specified state. There is currently no way to specify an initial state of disabled. For a list of possible initial states, see Table 1 on page 5.
- For trace loggers, the default initial state is for logging to be disabled; however, an initial state of enabled can be specified by using the appropriate WebSphere system-management utility. The trace logger's mask is set as specified in the system-management utility. For a list of possible initial states, see Table 2 on page 6 and Table 3 on page 6. Some editions of WebSphere Application Server enable you to change the state of the mask dynamically by enabling tracing for one or more trace loggers; refer to the documentation for your WebSphere system-management utility for more information.

  All enabling and disabling of trace must be performed through the appropriate WebSphere system-management utility.

Message and trace event types

This section discusses the message and trace types that are available through the WebSphere implementation of JRas. Message types are provided by the RASIMessageEvent interface, and trace types are provided by the RASITraceEvent interface.

Message types and usage

Message types are provided by the RASIMessageEvent interface. Types include the following:

- `TYPE_INFORMATIONAL` for informational messages. This type can be abbreviated as `TYPE_INFO`.
- `TYPE_WARNING` for warning messages. This type can be abbreviated as `TYPE_WARN`.
- `TYPE_ERROR` for error messages. This type can be abbreviated as `TYPE_ERR`.

These types, which are provided by JRas, do not correspond exactly to the message types supported by the different editions of the WebSphere run time. The following table shows the mappings between the JRas message types and their WebSphere equivalents. Note that the Enterprise Edition types apply to Component Broker on workstations.

Table 1. JRas message types and their WebSphere equivalents

JRas message type	Equivalent WebSphere Standard/Advanced Edition type	Equivalent WebSphere Enterprise Edition (Component Broker for workstations) type
`TYPE_INFO`, `TYPE_INFORMATION`	`Audit`	`Informational`
`TYPE_WARN`, `TYPE_WARNING`	`Warning`	`Warning`
`TYPE_ERR`, `TYPE_ERROR`	`Error`	`Error`

Trace types and usage

Trace types are provided by the RASITraceEvent interface. This interface defines two sets of JRas trace types: a basic set of leveled types for simple trace implementations and a more complex set of nonleveled types that can be logically combined to create precise information about any given trace event. It is recommended that only one of these sets be used in any given application.

The basic set of types consists of the `TYPE_LEVEL1`, `TYPE_LEVEL2`, and `TYPE_LEVEL3` trace levels. These levels are hierarchical; enabling a higher level of trace automatically enables all levels beneath it (for instance, enabling `TYPE_LEVEL2` automatically enables `TYPE_LEVEL1`).

The complex set of types consists of the following trace values:

```
TYPE_API
TYPE_CALLBACK
TYPE_ENTRY_EXIT
TYPE_ERROR_EXC
TYPE_MISC_DATA
TYPE_OBJ_CREATE
TYPE_OBJ_DELETE
TYPE_PRIVATE
TYPE_PUBLIC
TYPE_STATIC
TYPE_SVC
```

These values can be combined logically (that is, by using operators such as AND, OR, and NOR) to provide detailed information about any given trace event.

As with the message types, the JRas trace types do not correspond exactly to the types used by the WebSphere run time. The following tables show the mappings between the JRas trace types and their WebSphere equivalents. Note that the WebSphere equivalents apply to Standard Edition, Advanced Edition, and, for Enterprise Edition, Component Broker on workstations.

Table 2. Leveled JRas trace types and their WebSphere equivalents

JRas level event type	**WebSphere equivalent**
`TYPE_LEVEL1`	`Event`
`TYPE_LEVEL2`	`Entry/Exit`
`TYPE_LEVEL3`	`Debug`

Table 3. Nonleveled JRas trace types and their WebSphere equivalents

JRas nonleveled event types	**WebSphere equivalent**
`TYPE_ERROR_EXC`, `TYPE_OBJ_CREATE`, `TYPE_OBJ_DELETE`, `TYPE_SVC`	`Event`
`TYPE_API`, `TYPE_CALLBACK`, `TYPE_ENTRY_EXIT`, `TYPE_PRIVATE`, `TYPE_PUBLIC`, `TYPE_STATIC`	`Entry/Exit`
`TYPE_MISC_DATA`	`Debug`

Using JRas loggers

This section discusses how to use JRas loggers in WebSphere applications. “Creating resource bundles and message files” provides an overview of creating resource bundles to provide localized (translated) messages. “Creating manager and logger instances” on page 8 discusses how to obtain a JRas manager, and subsequently how to obtain message and trace loggers. “Using loggers” on page 9 describes the logger interfaces and shows how to use them.

Creating resource bundles and message files

This section provides an overview of how to create resource bundles that can be translated to provide localized messages in WebSphere applications. The Java programming language provides the java.util.ResourceBundle class and its subclasses, java.util.ListResourceBundle and java.util.PropertyResourceBundle, to enable national language support for applications. The ResourceBundle class is used in conjunction with the java.text.MessageFormat class to provide localized (translated) text support. See the Java documentation for a full discussion of the ResourceBundle and MessageFormat classes.

ResourceBundle is a class that encapsulates the retrieval of text. Entries in a resource bundle consist of message keys and their corresponding message text. When a resource bundle is translated, only the message text is translated into the national language. The translated resource bundles are packaged together and shipped with the application to provide localized messages.

This section discusses how to create resource bundles in the form of text properties files that can be accessed by PropertyResourceBundle. You can also create resource bundles by using a Java class that extends ListResourceBundle. The class encapsulates the mapping of keys to values by using arrays. For information on creating resource bundles by using ListResourceBundle, see the Java documentation.

The simplest way to create a resource bundle is to create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:

- Each property in the file is terminated with a line-termination character.
- If a line contains only white space, or if the first non-white space character of the line is the symbol # (pound sign) or `!` (exclamation mark), the line is ignored. The # and `!` characters can therefore be used to put comments into the file.
- Each line in the file, unless it is a comment or consists only of white space, denotes a single property. A backslash (\) is treated as the line-continuation character.
- The syntax for a property line consists of a key, a separator, and an element. Valid separators include the equal sign (=), colon (`:`), and white space ( ).
- The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but doing this is not recommended, because escaping characters is error prone and confusing. It is instead recommended that you use a valid separator character that does not appear in any keys in the properties file.
- White space after the key and separator is ignored until the first non-white space character is encountered. All characters remaining before the line-termination character define the element.

See the Java documentation for the java.util.Properties class for a full description of the syntax and construction of properties files.

The following example shows a properties file named DefaultMessages.properties.

```
# Contents of DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2={1}
MSG_KEY_03=A message with three substitution parameters: parm1={0}, parm2={1}, \
parm3={2}
```

Figure 1. Sample resource bundle

This file can then be translated into localized versions of the file (for example, DefaultMessages_de.properties for German and DefaultMessages_ja.properties for Japanese). When the translated resource bundles are available, they are written to a system-managed persistent storage medium. Resource bundles are then used to convert the messages into the requested national language and locale. When a message logger is obtained from the JRas manager, it can be configured with a default resource bundle. At run time, the user's locale is used to determine the properties file from which to extract the message specified by a message key, thus ensuring that the message is delivered in the correct language. If a default resource bundle is not specified, the msg method of the RASIMessageLogger interface can be used to specify a resource bundle name.

The application locates the resource bundle based on the file's location in the directory structure. For instance, if the resource bundle is located in the *baseDir*/*subDir1*/*subDir2*/resources directory and *baseDir* is in the classpath, the name *subDir1*.*subDir2*.resources.DefaultMessage is passed to the message logger to identify the resource bundle.

Creating manager and logger instances

This section provides sample code in which message loggers and trace loggers are obtained in the main method of a standalone application. To obtain a logger, you first obtain a manager by calling the getManager method on the com.ibm.websphere.ras.Manager class. You then obtain a message logger by calling createRASIMessageLogger on the returned manager object, or a trace logger by calling createRASITraceLogger on the returned manager object. Figure 2 on page 9 demonstrates these methods.

```
// Import the appropriate JRas and WebSphere packages
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;
// Declare the logger attributes and a group name for trace loggers. The storage
// scope used here depends on the application.
static RASITraceLogger trcLogger = null;
static RASIMessageLogger msgLogger = null;
// Define some convenience strings
static String svOrg = "My organization name";
static String svProd = "My product name";
static String svComponent = "My component name";
static String svClassName = "Fully qualified class name";
static java.lang.String groupName = "MyProduct_someGroup";
...
public static void main(String[] argv)
{
// Get a reference to the Manager instance and create the loggers.
// Because "Manager" is a common term, fully qualify it to ensure we
// get the right one.
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASIMessageLogger(svOrg, svProd, svComponent, svClassName);
trcLogger = mgr.createRASITraceLogger(svOrg, svProd, svComponent, svClassName);
// Configure the message logger with the default resource bundle
msgLogger.setMessageFile("subDir1.subDir2.resources.DefaultMessages");
// Add the trace logger to a group
mgr.addLoggerToGroup(trcLogger, groupName);
}
```

Figure 2. Example code: Obtaining a manager, a message logger, and a trace logger

Using loggers

This section discusses the use of JRas loggers in WebSphere applications. “Message and trace parameters” discusses the message and trace parameters used with JRas objects. “The RASILogger interface” on page 10 discusses the RASILogger interface, “The RASIMessageLogger interface” on page 11 discusses the RASIMessageLogger interface, and “The RASITraceLogger interface” on page 13 discusses the RASITraceLogger interface. Figure 3 on page 16 shows examples of using these methods.

Message and trace parameters

The JRas methods accept parameter types of Object, Object[], and Exception. The following is a list of parameter types and how they are handled by the WebSphere implementation of JRas.

- *Primitives*—Primitive data types such as int and long are not recognized as subclasses of the Object class and cannot be directly passed to JRas methods. A primitive value must be transformed to its proper type (for instance, Integer or Long) before being passed as a parameter.

- *Object*—JRas methods accept members of the Object class; the toString method is called on the object and the resulting String is returned. The toString method must therefore be implemented on Objects of traced classes.
- *Object[]*—JRas methods accept members of the Object[] class when two or more Object parameters need to be passed to the method. The toString method is called on each Object in the array. Nested arrays (that is, arrays with elements that are also arrays) are not supported.
- *Throwable*—JRas methods accept members of the Throwable class, returning the stack trace of the Throwable object.
- *Arrays of primitives*—An array of primitives (for example, byte[] or int[]) is considered to be an Object by Java; however, because of potentially inconsistent processing, it is recommended that members of the array be converted to String and then passed to the method. If such conversion is not performed, the results are unpredictable.

The RASILogger interface

The RASILogger interface is the base interface for both the RASIMessageLogger and RASITraceLogger classes. This section discusses topics that are common to both of these classes, including the isLoggable, getName and setName, and isSynchronous and setSynchronous methods. See Figure 3 on page 16 for examples of the classes and methods being used in context.

The RASILogger interface provides the isLoggable method to determine whether a logger is currently enabled to log a particular event type. The event type to be checked is passed to the method. The definition is as follows:

```
public boolean isLoggable(long type);
```

where *type* is a valid message or trace type. See “Message and trace event types” on page 4 for a discussion of message and trace types.

The getName and setName methods provide access to logger names. Because all loggers are assigned an unchangeable name by the manager when they are created, the setName method results in a null operation if used. The getName method can be used at any time to retrieve a logger’s name. The definitions of these methods are as follows:

```
public String getName();
public void setName (String name);
```

where *name* is the logger’s name.

The isSynchronous and setSynchronous methods enable applications to configure loggers to perform synchronous or asynchronous logging, assuming that the logger can accept the configuration. The configuration is set by the

WebSphere run time, so the setSynchronous method is currently implemented as a null operation. The definitions of these methods are as follows:

```
public boolean isSynchronous();
public void setSynchronous(boolean flag);
```

where *flag* is a Boolean value indicating `True` (for synchronous logging) or `False` (for asynchronous logging).

The RASIMessageLogger interface

The RASIMessageLogger interface provides methods that enable localizable message logging. These methods include getMessageFile and setMessageFile, message, msg, and textMessage. When an instance of RASIMessageLogger is obtained from the manager, you must provide nonnull strings that specify the logger's organization name, product name, and component information. These strings are unchangeable for the lifetime of the logger.

The logger interface includes support for an internal mask that identifies which categories of messages are to be logged and which categories are to be disregarded. The mask is set by the WebSphere run time when the logger is created.

The getMessageFile method enables you to specify a resource bundle that the logger uses to localize messages. If the name of the resource bundle is not specified, a default name is assumed. The setMessageFile enables you to configure the message logger with a message file that is used by a message logged by the message interface. There is no default value for the message file; if this value is not specified, using the message interface can have unpredictable results. See "Creating resource bundles and message files" on page 6 for information on resource bundles. The definitions of the methods are as follows:

```
public String getMessageFile();
public void setMessageFile(String file);
```

where *file* is the name of the resource bundle.

The message method provides flexible access to message strings. The definition of the method is as follows:

```
public void message(long type, Object obj, String methodName, String key,
Object parameter);
```

where:

- *type* is a valid message type. See "Message and trace event types" on page 4 for a discussion of trace types.
- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the

class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().
- *methodName* is a valid method name.
- *key* is the message key of the localizable message. The resource bundle that was specified by the setMessageFile method is used to retrieve the message text.
- *parameter* represents an Object that is to be substituted positionally into the message text. More than one *parameter* can be passed. See "Message and trace parameters" on page 9 for more information.

The msg method also provides access to message strings; unlike the message method, it enables you to specify the resource bundle from which message text is to be retrieved. The definition of the method is as follows:

```
public void msg(long type, Object obj, String methodName, String key,
String file, Object parameter);
```

where:
- *type* is a valid message type. See "Message and trace event types" on page 4 for a discussion of message types.
- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().
- *methodName* is a valid method name.
- *key* is the message key of the localizable message.
- *file* is the resource bundle to use when retrieving the message text.
- *parameter* represents an Object that is to be substituted positionally into the message text. More than one *parameter* can be passed. See "Message and trace parameters" on page 9 for more information.

The textMessage method enables applications to send text messages that are not accessed from a resource bundle. This method is intended for use in development environments or environments in which localization support is not required. This method is not intended to be used in production code. The definition of the method is as follows:

```
public void textMessage(long type, Object obj, String methodName,
String text, Object parameter);
```

where:
- *type* is a valid message type. See "Message and trace event types" on page 4 for a discussion of message types.

- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().
- *methodName* is a valid method name.
- *text* is the message text. No resource bundle is accessed to provide the text, and the text cannot be localized.
- *parameter* represents an Object that is to be appended to the message text. More than one *parameter* can be passed. See "Message and trace parameters" on page 9 for more information.

The RASITraceLogger interface

The RASITraceLogger interface provides methods that enable generic tracing mechanisms. These methods include entry, exit, trace, and exception. When an instance of RASITraceLogger is obtained from the manager, you must provide nonnull strings that specify the logger's organization name, product name, and component information. These strings are unchangeable for the lifetime of the logger.

The logger interface includes support for an internal mask that identifies which categories of trace events are to be logged and which categories are to be disregarded. The mask is set by the WebSphere run time when the logger is created.

The entry method provides access to trace entry events. The definition of the method is as follows:

```
public void entry(long type, Object obj, String methodName, Object parameter);
```

where:

- *type* is a valid trace type. See "Message and trace event types" on page 4 for a discussion of trace types.
- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().
- *methodName* is a valid method name.
- *parameter* represents a parameter to be added to the trace text. See "Message and trace parameters" on page 9 for more information.

The exit method provides access to trace exit events. The definition of the method is as follows:

```
public void exit(long type, Object obj, String methodName, Object retValue);
```

where:

- *type* is a valid trace type. See "Message and trace event types" on page 4 for a discussion of trace types.
- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().
- *methodName* is a valid method name.
- *retValue* is a return value for the event. See "Message and trace parameters" on page 9 for more information.

The trace method provides a way to write text strings as trace events. The definition of the method is as follows:

```
public void trace(long type, Object obj, String methodName, String text,
Object parameter);
```

where:

- *type* is a valid trace type. See "Message and trace event types" on page 4 for a discussion of trace types.
- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().
- *methodName* is a valid method name.
- *text* is a text string to be written to the trace event record.
- *parameter* represents a parameter to be added to the trace text. See "Message and trace parameters" on page 9 for more information.

The exception method provides access to exceptions. The definition of the method is as follows:

```
public void exception(long type, Object obj, String methodName,
Exception exc);
```

where:

- *type* is a valid trace type. See "Message and trace event types" on page 4 for a discussion of trace types.
- *obj* is a class name to be passed to the logger. You can pass the class name in the form of either a String or an Object. Passing a String is more efficient and must be used in static methods. For convenience, you can also pass the class name in the form of the `this` object; in this case, the logger retrieves the class name from the `this` reference by calling this.getClass().getName().

- *methodName* is a valid method name.
- *exc* is an exception whose stack trace is to be written to the trace event record.

Figure 3 on page 16 shows an example of using a message logger and a trace logger.

```
private void methodX(int x, String y, Foo z)
{
// Trace an entry point. Use the guard to ensure tracing is enabled. Do this
// checking before we waste cycles gathering parameters to be traced.
if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
// Because we want to trace three parameters, package them into an Object[]
Object[] parms = {new Integer(x), y, z};
trcLogger.entry(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX", parms);
}

// ...additional logic here...

// A debug or verbose trace point
if (trcLogger.isLoggable(RASITraceEvent.TYPE_MISC_DATA)) {
trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA, this, "methodX", "reached here");
}

// ...

// Call methodY on Foo. Assume that Foo is provided by another vendor or user.
// This method throws no Exceptions, so any run-time exceptions such as a
// NullPointerException coming out of it must be logged as errors.
// Although it is not good practice to put stack traces into message,
// it is not explicitly prohibited.
try {
z.methodY(...);
}
catch (Throwable t) {
msgLogger.message(RASIMessageEvent.TYPE_ERR, this, "methodX", "MSG_KEY_01", t);
}
// ...

// Another classification of trace event. An important state change was
// detected, so a different trace type is used.
if (trcLogger.isLoggable(RASITraceEvent.TYPE_SVC)) {
trcLogger.trace(RASITraceEvent.TYPE_SVC, this, "methodX", "an important event");
}

// ...

// Ready to exit method, trace. No return value to trace.
if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
trcLogger.exit(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX");
}

}
```

Figure 3. Example code: Using a message logger and a trace logger

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

For Component Broker:
IBM Corporation
Department LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

For TXSeries:
IBM Corporation
ATTN: Software Licensing
11 Stanwix Street
Pittsburgh, PA 15222
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

Advanced Peer-to-Peer Networking
AFS
AIX
APPN
AS/400
CICS
CICS OS/2
CICS/400
CICS/6000
CICS/ESA
CICS/MVS
CICS/VSE
CICSPlex
DB2
DCE Encina Lightweight Client
DFS
Encina
IBM
IBM System Application Architecture
IMS
IMS/ESA
Language Environment
MQSeries
MVS/ESA
NetView
Open Class
OS/2
OS/390
OS/400
Parallel Sysplex
PowerPC
RACF
RAMAO
RMF
RISC System/6000
RS/6000
S/390
SAA
SecureWay
TeamConnection
Transarc
TXSeries
VSE/ESA
VTAM
VisualAge
WebSphere