



**J A D E**™

## JADE Exception Handling

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2009 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

---

# Contents

Contents ii

<b>JADE Exception Handling</b>	<b>3</b>
Introduction .....	3
The Basics .....	3
What IS an Exception? .....	3
When Should You Raise Exceptions? .....	4
Exception Classes .....	4
Creating and Raising Exceptions .....	5
Exception Handlers .....	6
Writing a Simple Exception Handler .....	6
Local Exception Handlers .....	7
Global Exception Handlers .....	7
Exception Handler Scope .....	8
Exception Handling In-Depth .....	8
Exception Handler Stack .....	8
Exception Handler Return Values .....	8
Ex_Pass_Back .....	8
Ex_Abort_Action .....	9
Ex_Resume_Next .....	9
Ex_Continue .....	9
Summary of Exception Handling Behaviour .....	10
Building Exception Handlers .....	12
File Exception Handling .....	12
Global File Exception Handler .....	12
Local File Exception Handler .....	13
Connection Exception Handling .....	14
Lock Exception Handling .....	15
If-All-Else-Fails Exception Handling .....	16
Context of Exceptions .....	16
Exception Handling Strategies .....	17
Summary .....	18

---

# JADE Exception Handling

---

## Introduction

This paper is intended to provide the reader with a sound understanding of the use of JADE exception handlers, and presents a number of ways in which exception handlers may be used to help produce robust, high-performance JADE applications which properly encapsulate functionality within the application's classes.

The proper use of the exception handling mechanisms provided by JADE is fundamental to the building of industrial-strength applications, and to the creation of reusable, highly-encapsulated classes. To achieve the objectives of robustness, high performance and encapsulation requires a sound understanding of JADE's exception handling features.

For the best results, the application project leader needs to establish an application exception handling strategy at the outset of a project, and needs to ensure that all the members of the development team understand and adhere to the strategy. Later in this paper, key points for your exception handling strategy are discussed.

If you are experienced in using exceptions in other languages (e.g. C++, Java, C#) then the concepts associated with the material covered in this paper will be already familiar to you, as will the implications for the design of object-oriented systems. However, you should still find the implementation aspects of using JADE exceptions to be useful.

## The Basics

This topic introduces the basics of using JADE exceptions and exception handlers.

### What IS an Exception?

When errors occur in a JADE application, an *exception object* is created (either by JADE or by the application developer's code) and control is automatically passed, along with the exception object, to a predefined method called an *exception handler*. (Exception handlers are discussed in the next topic.) In effect, this is a sort of structured go-to. However, an exception doesn't merely transfer control from one part of your program to another: it also transmits information.

Because the exception is an object, it usually contains information about the condition that resulted in the exception being raised. Depending on the type of exception, the exception object may contain a direct reference to the object that caused or was involved in the error. For example, **FileExceptions** contain a reference to the file object in use at the time of the error, and **ConnectionExceptions** contain a reference to the connection object that encountered the error.

Examples of situations that can cause JADE to raise exceptions are:

- |                   |      |   |
|-------------------|------|---|
| Fatal errors      | e.g. | <ul style="list-style-type: none"> <li>• Wrong number of parameters in a method call.</li> </ul>  |
| Normal exceptions | e.g. | <ul style="list-style-type: none"> <li>• JADE system exceptions such as lock exceptions and integrity violations.</li> <li>• File handling exceptions.</li> <li>• Connection exceptions.</li> <li>• User interface exceptions.</li> </ul> |

When JADE detects an error (for example, a TCP/IP connection failure), it automatically creates an exception object of the appropriate class, sets up the property values in the exception object that describe the nature of the error, then passes control to the first appropriate exception handler method in the exception handler stack.

In addition to the above, the application developer can **create** exception objects (subclasses of RootSchema exception classes, with additional application-specific properties, if required) and use the exception object to **raise** an exception.

As we will see later, this feature provides a very elegant way to handle error conditions (for example, when a persistent class is editing input which originated from the user interface or a TCP/IP message). It also allows the developer to completely encapsulate editing and other class-specific functionality in the affected class methods. This enhances the portability and reusability of classes, and the integrity of the data held in class objects.

## When Should You Raise Exceptions?

JADE raises exceptions whenever a method encounters an abnormal condition. Similarly, as the application developer, you should raise exceptions whenever one of your methods encounters an error that your method can't handle. For example, if a class method validates data passed as input to the method, it should raise an exception if it finds an error that the method itself can't handle.

However, generally you shouldn't use exceptions where the method could handle the error. For example, when a method of a form is validating user input and an error is detected, the method should handle the error directly rather than raise an exception.

By using exceptions rather than clumsy mechanisms such as method return codes, you can ensure the integrity of the data in your classes without impacting on system efficiency. By using exceptions, you also keep your mainline application code comparatively free of error handling code, thus making it easier to understand and maintain.

## Exception Classes

The JADE RootSchema defines a number of exception classes and associated methods. These are fully described in the JADE *Encyclopaedia of Classes and Types*. The following is a summary of the **Exception** class hierarchy and functions:

<b>Exception</b>	Superclass for all exceptions
<b>FatalError</b>	Serious internal errors
<b>NormalException</b>	Superclass for all non-fatal exceptions
<b>ConnectionException</b>	Exceptions relating to connecting to external systems

<b>FileException</b>	Exceptions relating to file handling errors
<b>JadeMessagingException</b>	Exceptions relating to the JADE messaging framework
<b>JadeSOAPException</b>	Exceptions relating to Web service handling
<b>JadeXMLException</b>	Exceptions relating to XML processing
<b>ODBCException</b>	Exceptions from ODBC connection to external databases
<b>SystemException</b>	Superclass for exceptions relating to errors detected by JADE kernel
<b>DeadlockException</b>	Deadlock errors
<b>LockException</b>	Lock-related errors
<b>NotificationException</b>	Exceptions relating to notification event delivery
<b>IntegrityViolation</b>	Reserved for future use
<b>UserInterfaceException</b>	Superclass for exceptions relating to the handling of windows
<b>ActiveXInvokeException</b>	Exceptions relating to ActiveX properties and methods
<b>JadeDotNetInvokeException</b>	Exceptions relating to .NET component properties and methods

## Creating and Raising Exceptions

The JADE application developer can create exception objects (typically of type **NormalException** or a subclass of), and having created an exception object can then **raise** that exception so that control is passed to the appropriate exception handler. A typical example of a use for this feature is for handling errors arising from edits of data passed as input to a method.

Here are a couple of code fragments showing how a user exception is created and raised:

```

raiseEditException(text : String); // my method of Application
vars
    exObj : EditException; // my subclass of NormalException
begin
    create exObj;
    exObj.errorCode := Edit_Error;
    exObj.extendedErrorText := text;
    raise exObj; // using JADE 'raise' instruction
end;

setName(sName : String); // method of Customer
vars
begin
    if sName.trimBlanks = null then
        app.raiseEditException("Customer name must be specified");
    else
        self.name := sName;
    endif;
end;

```

For the example above, the RootSchema **NormalException** class was subclassed as **EditException**, to allow additional properties to be added to the exception object or to allow reimplementations of superclass methods.

## Exception Handlers

An exception handler is a method to which control is passed, along with the exception object, when an exception is raised. JADE provides a default exception handler, part of which is the Unhandled Exception dialog that you see when errors occur in your application. However, a well-designed and implemented application should seldom, if ever, display the default JADE dialog. The application developer must write exception handler methods to catch most exceptions, and arm and disarm these handlers as needed.

### Writing a Simple Exception Handler

To demonstrate a simple use of exception handlers, the following is a handler written for the form whose **bOK\_click** method is shown in the next section of this paper, in **Local Exception Handlers**. This handler is designed for use with the GUI, so you would normally code it as a method of your form, or form superclass.

```
editExceptionHandler(exObj : EditException) : Integer updating;  
  
vars  
begin  
    abortTransaction; // if persistent database is in  
                      // transaction state, then abort. This  
                      // also releases any other transaction  
                      // locks, e.g. an explicit exclusiveLock.  
                      // It's important to do this before  
                      // displaying the message box/  
    app.msgBox(exObj.extendedErrorText,  
               "Application Error",  
               Msg_Box_OK + Msg_Box_Exclamation_Mark_Icon);  
    return Ex_Abort_Action; // cut back stack  
end;
```

In this example, we returned **Ex\_Abort\_Action**, which cuts back the stack and essentially leaves it in the state it was in before the method which armed the exception handler started. *However, note that any transient objects (shared or process) that were updated by your method will retain the updates, so you may also need to code a mechanism in your handler to undo those updates if that is important.*

*Also note that returning **Ex\_Abort\_Action** does not abort the database transaction. You must code **abortTransaction** to cause that to happen.*

JADE's **abortTransaction** instruction aborts the current persistent database transaction, if one is in progress, and also releases all transaction duration locks held by the process.

The other values that you can return from an exception handler are **Ex\_Resume\_Next**, **Ex\_Continue** and **Ex\_Pass\_Back**. All these options are discussed later, in the **In-Depth** section of this paper.

## Local Exception Handlers

Local exception handlers only stay armed until the method in which the handler was armed terminates, or until the handler is explicitly disarmed.

The following code fragment for a button control on a form shows how a local exception handler is armed:

```
vars
  cust : Customer;
begin
  on EditException do self.editExceptionHandler(exception);
  beginTransaction;
  cust := app.myRoot.allCustomers[tbName.text.trimBlanks];
  if cust = null then
    create cust;
  endif;

  cust.setName(tbName.text.trimBlanks);
  cust.setAddress(tbAddress.text);
  commitTransaction;
end; // editExceptionHandler is disarmed here
```

If an exception is raised by the **setName** method (see earlier code fragment), control is passed to the form's **editExceptionHandler**.

## Global Exception Handlers

Exception handlers can be designed and armed as **global** exception handlers; that is, once armed they stay active until they are disarmed or until the application terminates. Global exception handlers are useful for handling lock exceptions, connection exceptions, and as 'if all else fails' handlers for unexpected application and system exceptions, such as 'String Too Long' errors.

An example of arming a global exception handler is:

```
on ConnectionException do
  app.connectionExceptionHandler(exception) global;
```

To disarm a previously-armed global exception handler, you write:

```
on ConnectionException do null global;
```

Typically, if you need a global exception handler, it is convenient to arm it in the application's **initialize** method or startup form **load** event. However, you need to be careful that the receiver of the handler doesn't get subsequently deleted. For example, if a global exception handler is armed on a form (with the form as receiver) and that form is subsequently closed and deleted, you will have a handler in the exception handler stack with an invalid receiver object.

As an example, you could always code your global exception handlers as methods of the **Application** subclass, although you could equally well use other classes such as your subclass of **Global**, or **Process**.

## Exception Handler Scope

When your JADE client arms an exception handler, be it global or local, it is armed only for that process. If you start another process within the same **jade.exe** (using **startApplication**, **startApplicationWithParameter**, etc.), you need to arm exception handlers for that process too.

Similarly, if you are using **serverExecution** methods, exception handlers need to be written and armed for the server side of processing. These exception handlers can be armed globally or locally as for **clientExecution** methods, by performing a **serverExecution** method which executes an **on Exception** statement.

If you don't have a suitable exception handler armed for the server and an exception is raised during execution of a server method, the following happens:

A JADE default exception handler is invoked on the server before returning the error to the client. This exception handler is similar to the system Unhandled Exception dialog, but it doesn't display a dialog; it only logs the exception to a file.

JADE raises an exception 1242 "Remote execution aborted" at the client. The **extendedErrorText** property of this exception contains the text that corresponds to the original server method exception.

## Exception Handling In-Depth

This section discusses exception handling in greater detail. The examples and strategies shown here are based on the author's experience so far, and if you have alternative or better strategies for dealing with exceptions then please share them!

## Exception Handler Stack

JADE allows the developer to arm any number of local exception handlers for a given process, and up to 128 global exception handlers per process. As each handler is armed or disarmed it is added to or removed from the exception handler 'stack'. When an exception is raised, JADE passes the exception object to the first exception handler in the stack that is armed to handle that type of exception and control is passed to that handler. The class specified in the exception handler arming statement (**on <exception class> do <exception handler method name>**) is what determines whether a given exception is passed to a particular handler.

If an exception handler returns **Ex\_Pass\_Back** (see below), control is passed down to the next lower exception handler that is capable of handling the exception.

---

**Note** You can use the **Process::getExceptionHandlerStack** method to examine the current exception handler stack.

---

## Exception Handler Return Values

A JADE exception handler must return one of four integer values. These values and their significance are discussed below.

### **Ex\_Pass\_Back**

This return value passes control back to any previously armed local exception handler for this type of exception, or if a local exception handler is not found, a global exception

handler for this type of exception. If no exception handler is found, the JADE default exception handler is invoked.

## Ex\_Abort\_Action

This return value causes the currently executing methods to be aborted. The execution stack is cut back, and the application reverts to an idle state in which it is waiting for user input or some other event. If the execution stack contains a method that results in the modal display of a form, the stack is only cut back to the point of awaiting input to the modal form. In this latter case, you need to take care where your **beginTransaction** and **commitTransaction** instructions are coded if your exception handler performs an **abortTransaction**, otherwise input subsequent to an exception could result in an 'update while not in transaction state' error.

Note that if there is a persistent database transaction in progress, an **abortTransaction** instruction must be coded in the exception handler if the database transaction is to be aborted.

The situation is similar for shared transient transactions. You should use an **abortTransientTransaction** instruction to abort the shared transient transaction and to discard any uncommitted updates.

Of course, ordinary process transient objects can also have been updated as a result of an executing method, and returning **Ex\_Abort\_Action** does not undo such updates. If this is important, the application developer must code a mechanism to undo any updates that have been applied.

## Ex\_Resume\_Next

This return value passes control back to the method that armed the exception handler. Execution resumes at the next statement after the method call expression in which the exception occurred.

In order to use **Ex\_Resume\_Next** as the return value, the exception must be **resumable**, otherwise another exception (1238 – Exception handler invalid return code) is raised. For `SystemExceptions`, **resumable** is **true** by default, and **false** for **FatalExceptions**. For exceptions that you raise yourself, you should set the value of **resumable** to meet your application requirements.

**Ex\_Resume\_Next** is generally only useful for local exception handlers. If you were to use this return option with a global exception handler, then since the method that armed the exception handler may no longer be executing, your application thread typically goes idle so this is not usually useful for global exception handlers.

## Ex\_Continue

This return value resumes execution from the next expression following the expression that caused the exception. Apart from any effect arising from execution of code in the exception handler, the execution stack will be as it was before the exception occurred.

In order to use **Ex\_Continue** as the return value, the exception must be **continuable**, otherwise another exception (1238 – Exception handler invalid return code) is raised. For `SystemExceptions` and user exceptions, this property is **false** by default. For exceptions that you raise yourself, you should set the value of **continuable** to meet your application requirements.

An exception to the above is the **LockException** class, where **continuable** is automatically set to **true** when your exception handler successfully retries the lock operation. If you return **Ex\_Continue** from a lock exception handler when you do not have the lock, JADE raises a 1225 exception 'Lock cannot be continued' (or 1224 'Automatic lock ignored' if it was an implicit or internal lock).

---

**Note** The 1146 SystemException 'The object was updated before the lock upgrade completed' is also continuable. This exception is related to the use of **update locks**.

---

Continuable exceptions assume that the cause of the problem has been fixed and the operation retried, so that the net effect of the exception and exception handling is as if the exception never occurred. For example, a lock exception that successfully retries the lock is transparent to the user code that requested the lock. Similarly, continuable user exceptions and their handlers should have the same kind of provision to avoid skipping the execution of important code. Such skipping could result in erratic behavior of the application.

## Summary of Exception Handling Behaviour

Consider the following method executions:

**method1());**

```
on LockException do lockExceptionHandler(exception);
method2();
```

**method2());**

```
on FileException do fileExceptionHandler(exception);
method3();
```

**method3());**

```
create file;
file.fileName := "Invalid Name";
file.open; // causes file exception
file.writeLine("Test");

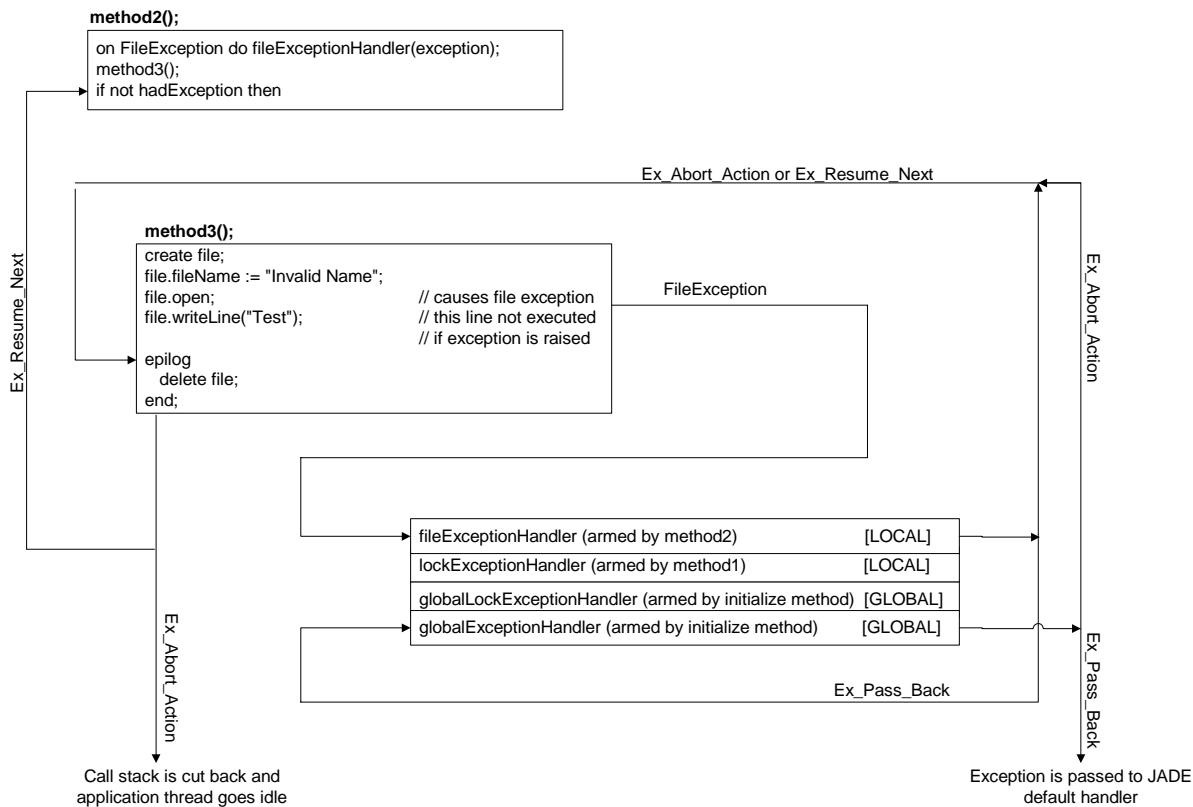
epilog
  delete file;
end;
```

At the time that **method3** is executing, the exception handler stack could look like this:

fileExceptionHandler (armed by method2)	[LOCAL]
lockExceptionHandler (armed by method1)	[LOCAL]
globalLockExceptionHandler (armed by initialize method)	[GLOBAL]
globalExceptionHandler (armed by initialize method)	[GLOBAL]

You will observe that the stack also includes two global exception handlers, in this case armed by the application's **initialize** method. We will now discuss the various options that exist for the handling of an exception in this example environment.

When the **file.open** statement is executed in **method3**, JADE raises a **FileException**. The following diagram shows what happens to that exception, dependent upon the return code returned by the exception handler.



Referring to the above diagram, you will observe that when the **fileExceptionHandler** (armed by **method2**) returns **Ex\_Abort\_Action**, the epilog is executed. When this has been performed, the call stack is cut right back (performing each method's epilog as the stack is cut back), and the application essentially goes idle.

If the **fileExceptionHandler** returns **Ex\_Pass\_Back**, the exception is passed to the next appropriate exception handler in the exception handler stack. In this case, this is the **globalExceptionHandler**.

If the **fileExceptionHandler** returns **Ex\_Resume\_Next**, the call stack is cut back to the method that armed the exception handler (in this case **method2**), and execution resumes from the statement following the statement that called the methods that resulted in the exception. Before each method is cut out of the call stack, its epilog is performed.

Note that this epilog execution, while useful, is a potential source of secondary exceptions if the epilog references the object that caused the initial exception. JADE allows multiple (nested) exceptions to occur while handling exceptions, up to a limit of 20. To prevent stack overflows, JADE raises a fatal exception (which no other handler can intercept) when the nested exception limit is exceeded.

## Building Exception Handlers

### File Exception Handling

File exceptions occur when a method attempts an invalid operation on a file; e.g. attempting to open a file that is already opened exclusively by another process. For file exceptions, JADE raises an instance of **FileException**. This class has an additional property, **file**, which is the **File** object that was being handled when the exception was raised.

There are at least two ways to handle **FileExceptions**. Two examples follow, which may be helpful.

### Global File Exception Handler

This approach is mainly useful for user interface methods, where **Ex\_Abort\_Action** may be an acceptable return value from the exception handler; i.e. the stack is cut back and the thread goes idle. Assuming that the following exception handler had been armed globally, a user-friendly error result can be presented in response to a variety of file related errors.

```
initialize() updating; // method of Application
vars
begin
    on FileException do app.globalFileExceptionHandler(exception);
end;

bTest_click(btn : input) updating;
vars
    file : File;
begin
    create file;
    file.fileName := tbFileName.text;
    file.mode      := file.Mode_Output;
    file.kind      := file.Kind_ASCII;
    file.writeLine("Just testing") // possible exception raised here
    file.close;
end;

globalFileExceptionHandler(exObj : FileException) : Integer updating;
vars
begin
    abortTransaction;
    app.msgBox(exObj.text,
               "File Exception in " & app.name,
               Msg_Box_OK + Msg_Box_Exclamation_Mark_Icon);
    return Ex_Abort_Action;
end;
```

## Local File Exception Handler

In my experience, a more common situation is that the application needs to handle file exceptions and allow the thread to continue processing after the exception has been handled. For this situation, you need to use a local exception handler that returns **Ex\_Resume\_Next**. (You will recall that **Ex\_Resume\_Next** returns control to the method that armed the exception handler, following the statement that invoked the current method.)

For example, in the **isDatabaseClosed** method below, I check to see whether a target JADE database is closed, by attempting an exclusive open of the database control file. The transient class singleton **CnKarmaCntrl** is used to return information about the exception back to the method after the exception handler completes and returns control to the method that armed the exception handler.

```
localFileExceptionHandler(exObj : FileException) : Integer updating;
vars
    kc : CnKarmaCntrl;
begin
    kc := app.myCnKarmaCntrl;
    kc.setHadException(true);
    kc.setExceptionErrorCode(exObj.errorCode);
    kc.setExceptionText(exObj.text);
    kc.setExceptionFileName(exObj.file.fileName);
    return Ex_Resume_Next;
end;

isDatabaseClosed(dbPath : String) : Boolean;
vars
    file: File;
    kc : CnKarmaCntrl; // transient singleton
    cc : CnCntrl;      // logging class
begin
    kc := app.mCnKarmaCntrl;
    cc := app.myCnCntrl;
    on FileException do localFileExceptionHandler(exception);
    create file;
    file.kind      := file.Kind_Binary;
    file.mode      := file.Share_Exclusive;
    file.fileName := dbPath & "/_control.dat";
    kc.setHadException(false);
    file.open; // may cause exception
    if kc.hadException then // Ex_Resume_Next returns control to here
        return false;
    else
        file.close;
        return true;
    endif;
epilog
    delete file;
end;
```

## Connection Exception Handling

My experience with Connections is mainly limited to TCP/IP connections which are controlled by a background process which is not part of the application GUI, so the following discussion relates to handling exceptions in that environment.

Connection exceptions are a little different to other types of exceptions, because they often happen on an asynchronous thread. For example, if your connection object performs a **readBinaryAsynch**, JADE forks an additional operating system thread to handle the read, allowing the initiating method to continue processing. The thread that is handling the **readBinaryAsynch** may sit on the read instruction for minutes or hours until the connection breaks (for example), at which time a **ConnectionException** is raised by JADE.

Since the method that performed the original **readBinaryAsynch** would normally have completed execution at this time, a **local** exception handler cannot handle these types of exceptions. For this reason, I use global exception handlers for TCP/IP Connection exceptions in the **CardSchema** class library. This library, in order to provide for diagnostics and reconnection capabilities, subclasses **TcpIpConnection** and adds extra properties which you will see referenced in the following partial example of a **ConnectionException** handler. This doesn't purport to be a fully-fledged handler, but should give you some idea of what is required.

```
globalTcpExceptionHandler(exObj : ConnectionException io) : Integer;
vars
    tcp : CnTcpConnection;
    cc : CnCntrl; // CardSchema logging class
begin
    if not exObj.connection.isKindOf(CnTcpConnection) then
        return Ex_Pass_Back; // can't handle here
    endif;
    cc := app.myCnCntrl;
    tcp := exObj.connection.CnTcpConnection;
    cc.cnWriteLog(cc.CnLogErrors,
        "Tcp connection error to host "
        & tcp.computerName
        & " : error " & exObj.errorCode.String
        & " (" & exObj.text & ") on connection #"
        & tcp.connectionNo.String,
        tcp);
    if tcp.connectionType = Opener then
        tcp.reopenConnection;
    else
        delete tcp; // we use listenContinuousAsynch, so there will
        // already be another listening object
    endif;

    return Ex_Abort_Action; // connection exceptions usually fatal to
        // the current method execution anyway
end;
```

## Lock Exception Handling

Lock exceptions are another special case for exception handling, because you usually want to handle the exception (i.e. retry the lock) and having obtained the lock, continue the method that caused the exception.

Here is some code which you can use as a basis for a lock exception handler. This code retries the lock for a reasonable number of times, but you may wish to code it to retry for a maximum period of, say, 60 seconds. If the lock is not obtained, the handler gives up and returns **Ex\_Pass\_Back**.

```
globalLockExceptionHandler(le : LockException io) : Integer;
vars
  lockedByUserCode : String;
  retryCount : Integer;
begin
  if global.isValidObject(le.targetLockedBy) then
    // locking process could have gone away
    lockedByUserCode := le.targetLockedBy.userCode;
    if lockedByUserCode = null then
      lockedByUserCode := 'not available';
    endif;
  endif;

  // In CardSchema, for JADE clients, we put up a progress
  // dialog here, showing text as follows:
  // "Object " & le.lockTarget.String & " locked by "
  //      & lockedByUserCode
  //      & " : retrying, retry number=" & retryCount.String;

  while not tryLock(le.lockTarget,
                  le.lockType,
                  le.lockDuration,
                  le.lockTimeout) do
    retryCount := retryCount + 1;

    if retryCount > 30 then // 30 secs if lock timeout is 1000 mS
      return Ex_Pass_Back; // let another handler deal with it
    endif;

    // Update progress dialog here
  endwhile;

  return Ex_Continue; // must have the lock if we get to here
epilog
  // Unload progress dialog here
end;
```

---

**Note** `global.isValidObject` doesn't guarantee that the object will be there at the next reference. In this example, there is still a very small window where the **targetLockedBy** process could have gone away. In a highly active system, you might need to cater for this possibility.

---

## If-All-Else-Fails Exception Handling

All applications of any complexity have bugs that have not yet been detected. These can slip past your specific local and global exception handlers, so it is often desirable to have a global exception handler that can catch these errors and capture enough information to permit ready diagnosis of the problem. If your application is a background client running on a server, it is highly desirable to avoid display of the JADE default exception handler dialog, because this effectively halts execution of your application (possibly holding the database in transaction state) until someone notices and takes appropriate action.

In the **CardSchema** application, which is the superschema for applications that are managed by Jade's Systems Management service, we provide a global exception handler that is armed on the client and on the server by the application **initialize** method or startup form **load** method. This handler performs the following actions:

- ◆ Writes a diagnostic application state dump to log file
- ◆ Optionally advises our monitoring system via SNMP message
- ◆ Aborts transaction and releases any locks
- ◆ Determines whether the client is a Web client or JADE client, and displays an appropriate user-friendly form (non-modal, self-destructing) advising that a problem occurred.
- ◆ Returns **Ex\_Abort\_Action**

This gives the application a fighting chance of continuing operation without human intervention, while logging enough information for later diagnosis of the problem.

## Context of Exceptions

There are some aspects of exception handling that I have not covered in this paper. One of these is the issue of context. By this, I mean that when an exception is being handled, occasionally the handler has no easy way to find out exactly what the client was doing that resulted in the exception being raised. To address this need, some developers use a context object that is used to keep track of the application context at any particular point in time. Of course, this means that the developer must maintain the context object, which is a substantial and error-prone overhead.

JADE provides optional exception handler parameters that make it easier for developers to manage exception context. When arming an exception handler, following the first parameter (which must always be the reserved word **exception**), you can specify any number of additional parameters that will be passed to the exception handler. These parameters can be input or output, enabling you to pass information into the exception handler and return information from the exception handler. For global exception handlers, the parameter values are evaluated when the exception handler is armed. For local exception handlers, the parameter values are evaluated when the exception is raised. For more details, see the *Exceptions* chapter in the *JADE Developer's Reference*.

## Exception Handling Strategies

Adoption of an exception handling strategy is an essential part of the JADE object-oriented design process. For the best results, the designer / project leader needs to define and communicate a strategy to the development team before implementation begins, not as an add-on later in the project.

Major aspects of the strategy should address the following objectives:

**The overall strategy should be one of Optimistic Error Handling; i.e. correct system behavior is *assumed*. Errors are dealt with separately, as exceptions.**

This approach enhances system efficiency, because checking for errors (e.g. via method return codes) is greatly reduced or eliminated. Error handling code is kept separate from normal application logic, which simplifies maintenance.

**For editing, error detection code should be encapsulated within the class whose data is being edited; i.e. classes should be responsible for precondition checks (i.e. verifying inputs from clients). When errors are detected, the class method should raise an exception.**

For example, editing code for a persistent or model class should reside in that class, not in the client GUI or other client location. Not only does this ensure the integrity of the data in the persistent class, but the code need only exist in one place.

This provides another benefit, where multiple developers work on the same project. The reduced coupling of application classes arising from the use of exceptions allows less dependent, parallel development activities to take place. For example, when the persistent database classes have been designed and their method signatures defined, the implementation of those classes can be worked on reasonably independently of (say) the GUI classes.

**Define application exception classes, if necessary, as subclasses of `NormalException`.**

If you want to add additional information to the exception objects that you raise, then you need your own exception subclasses and attendant properties. It is also useful to define your own exception classes so that your exception arming method (using the **on <Exception Class Name>** syntax) can specify a particular type of exception that is to be directed to your handler, rather than all exceptions.

**Identify which application exceptions are resumable and/or continuable, and set these properties appropriately when raising exceptions.**

Note that when you instantiate **NormalException** or a subclass of it, properties **resumable** and **continuable** are both **false** by default. If you want to allow the client's exception handler to return **Ex\_Resume\_Next** or **Ex\_Continue** after handling your exception, you must set these properties accordingly.

If your exception handler attempts to continue a non-continuable exception or to resume a non-resumable exception, JADE raises a 1238 exception "Invalid exception handler return code". Refer to the earlier discussion on the **Ex\_Continue** return value for more information.

## Summary

JADE's exception classes and exception handling capabilities provide you with a powerful set of features, enhancing your ability to build truly object-oriented systems. Exceptions and exception handlers are a fundamental part of a well-designed JADE application, and need to be considered at the design stage of a project.

The use of this functionality can help developers to meet desirable object-oriented development objectives such as maintainability, reusability, and encapsulation, while achieving high levels of system performance and reliability.