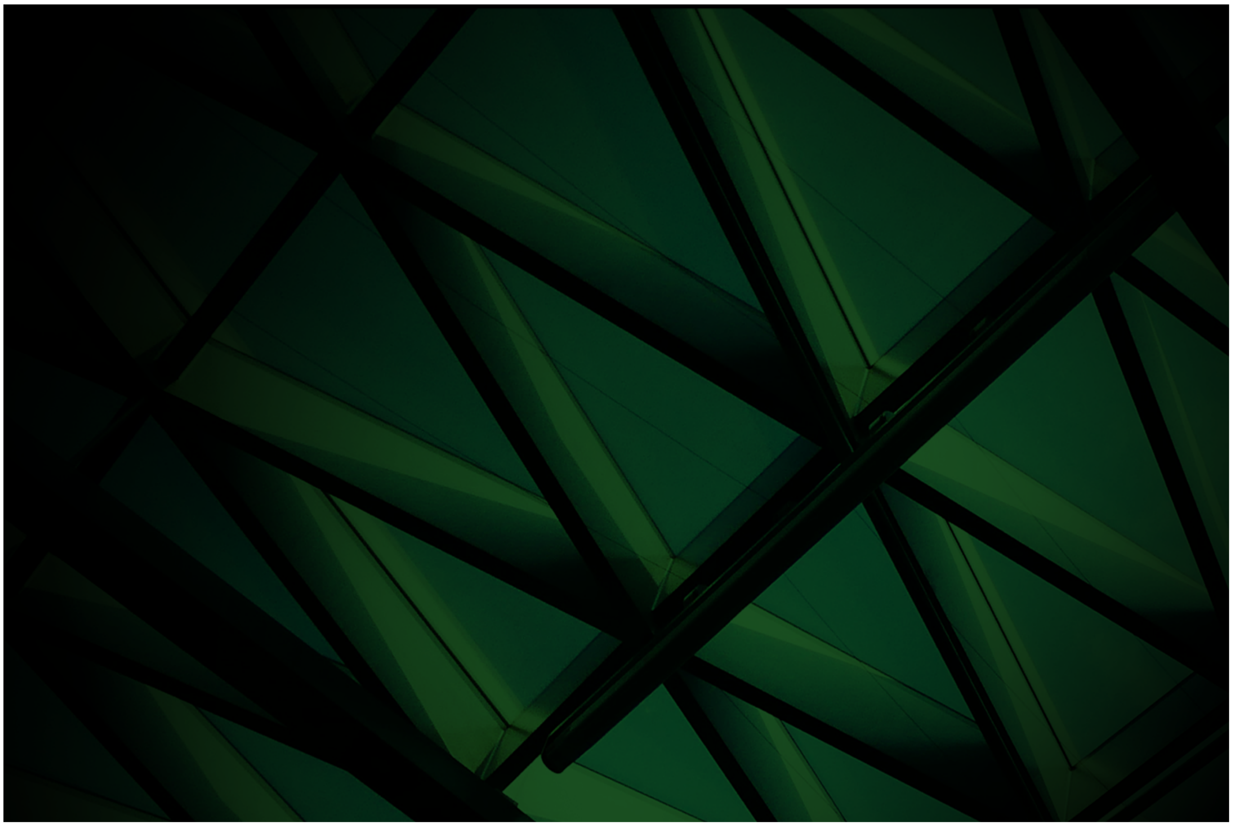


Exception Handling White Paper

VERSION 2022

Contents

Exception Handling

This white paper is intended to provide you with a sound understanding of the use of Jade exception handlers, and it presents a number of ways in which exception handlers can be used to help produce robust, high-performance Jade applications that 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 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 (for example, C++ or C#), 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; that is:

- What is an Exception?
- When Should You Raise Exceptions?
- Exception Classes
- Creating and Raising Exceptions
- 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.)

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 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, for example:
    - Wrong number of parameters in a method call
- Normal exceptions, for example:
    - Jade system exceptions such as lock exceptions and integrity violations
    - File-handling exceptions
    - Connection exceptions
    - User-interface exceptions

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, you 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 an elegant way to handle error conditions (for example, when a persistent class is editing input that originated from the user interface or a TCP/IP message). It also allows you 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, which are fully described in Volume 1 of the *Encyclopaedia of Classes*.

The following table is a summary of the **Exception** class hierarchy and functions.

Class	Function
Exception	Superclass for all exceptions
FatalError	Serious internal errors
NormalException	Superclass for all non-fatal exceptions

Class	Function
ConnectionException	Exceptions relating to connecting to external systems
FileException	Exceptions relating to file handling errors
JadeMessagingException	Exceptions relating to the Jade messaging framework
JadeRegexException	Transient class that defines behavior for exceptions that occur as a result of Jade Regular Expression (JadeRegex) pattern matching
JadeSOAPException	Exceptions relating to web service handling
JadeXMLException	Exceptions relating to XML processing
ODBCException	Exceptions from ODBC connection to external databases
SystemException	Superclass for exceptions for errors detected by Jade kernel
DeadlockException	Deadlock errors
LockException	Lock-related errors
NotificationException	Exceptions relating to notification event delivery
IntegrityViolation	Reserved for future use
UserInterfaceException	Superclass for exceptions relating to the handling of windows
ActiveXInvokeException	Exceptions relating to ActiveX properties and methods
JadeDotNetInvokeException	Exceptions relating to .NET component properties and methods
WebSocketException	Transient class that defines behavior for exceptions that occur as a result of the WebSocket protocol

Creating and Raising Exceptions

As a Jade application developer, you 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 this is handling errors arising from edits of data passed as input to a method. The following methods are examples of creating and raising a user exception.

```
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
begin
    if sName.trimBlanks = null then
        app.raiseEditException("Customer name must be specified");
    else
        self.name := sName;
    endif;
end;
```

In the first of these method examples, the RootSchema **NormalException** class was subclassed as **EditException**, to allow additional properties to be added to the exception object and to allow reimplementation 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. You must write exception handler methods to catch most exceptions, and arm and disarm these handlers as needed.

This section contains the following topics.

- Writing a Simple Exception Handler
- Local Exception Handlers
- Global Exception Handlers
- Exception Handler Scope

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 documented in "Local Exception Handlers", later in this white paper. 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;
begin
    abortTransaction; // if persistent database is in transaction state, then
                      // abort. This also releases any other transaction locks;
                      // for example, an explicit exclusiveLock. It is 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 abort any current persistent transaction then return **Ex_Abort_Action**, which cuts back all current methods, essentially leaving the process in the state it was in before the methods currently on the stack were executed. However, 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. In addition, returning **Ex_Abort_Action** does not abort the database transaction. You must code the **abortTransaction** instruction 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 on persistent objects held by the process.

The other values that you can return from an exception handler are **Ex_Resume_Next**, **Ex_Resume_Method_Epilog**, **Ex_Continue**, and **Ex_Pass_Back**. These options are discussed later, in the "Exception Handling In-Depth" section of this paper.

Local Exception Handlers

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

Caution Care should be taken when disarming a local exception handler. The search for a handler for exceptions of the specified class is not limited to the current method. If a suitable exception handler is not found in the current method, the exception handlers in calling methods are also checked. The **getExceptionHandlerStack** method of the **Process** class can be used to determine what exception handlers are currently armed.

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 (an example of which is shown in "Creating and Raising Exceptions", earlier in this document), 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 handlers of last resort for unexpected application and system exceptions such as String Too Long errors.

The following is an example of arming a global exception handler.

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

To disarm a previously-armed global exception handler, 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 of **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 (using **startApplication**, **startApplicationWithParameter**, and so on), you also need to arm exception handlers for that process.

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 that 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 occurs.

1. 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.

2. Jade raises 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 are based on the author's experience so far. If you have alternative or better strategies for dealing with exceptions, please share them!

- Exception Handler Stack
- Exception Handler Return Values
- Summary of Exception Handling Behavior
- Building Exception Handlers
- Context of Exceptions
- Exception Handling Strategies

Exception Handler Stack

Jade allows the developer to arm up to 128 local exception handlers for each method for a specific process, and up to 128 global exception handlers for each process. As each handler is armed or disarmed it is added to or removed from the local or global exception handler 'stack'. When an exception is raised, Jade passes the exception object to the most-recently armed local 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 specified exception is passed to a particular handler.

If no local exception handler handles a specified exception (or all return **Ex_Pass_Back**), the global exception handler stack is examined to look for a suitable global exception handler, starting with the most-recently armed handler.

If an exception handler returns **Ex_Pass_Back** (documented later in this white paper), control is passed down to the next most-recently armed exception handler that is capable of handling the exception.

Note To examine the current exception handler stack, use the **Process::getExceptionHandlerStack** method.

Exception Handler Return Values

A Jade exception handler must return one of four Integer values. These values and their significance are discussed in the following subsections.

- Ex_Pass_Back
- Ex_Abort_Action
- Ex_Resume_Next
- Ex_Resume_Method_Epilog
- Ex_Continue

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 cut back only to the point of awaiting input to the modal form. In this latter case, 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 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. Returning **Ex_Abort_Action** does not in itself abort database transactions.

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

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, you 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 executable statement after the method call expression in which the exception occurred. Epilog sections are executed for any methods on the call stack between the method that armed the handler and the method where the exception is raised.

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 **FatalErrors**.

For exceptions that you raise yourself, you should set the value of **resumable** to meet your application requirements.

Ex_Resume_Next is useful only for local exception handlers. Since the method that armed the exception handler may no longer be executing, if **Ex_Resume_Next** is returned from a global exception handler, it acts as if **Ex_Abort_Action** was returned; that is, the execution stack is cut back, and the application reverts to an idle state.

Ex_Resume_Method_Epilog

This return value passes control back to the method that armed the exception handler. Execution resumes at the start of the method epilog or at the end of the method if there is no epilog section. Execution resumes at the next statement in the epilog if the exception was raised while executing the epilog. Epilog sections are executed for any methods on the call stack between the method that armed the handler and the method where the exception is raised.

To use **Ex_Resume_Method_Epilog** 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 **FatalErrors**.

For exceptions that you raise yourself, you should set the value of **resumable** to meet your application requirements.

Ex_Resume_Method_Epilog is generally useful only for local exception handlers. If you were to use this return value with a global exception handler, the method that armed the exception handler may no longer be executing. If **Ex_Resume_Method_Epilog** is returned by a global exception handler, the behavior is as if the return value is **Ex_Abort_Action**; that is, the execution stack is cut back, and the application reverts to an idle state.

Applies to Version: 2022 and higher

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. However, this does not apply to 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 exception 1225 *(Lock cannot be continued)* or 1224 *(Automatic lock ignored)* if it was an implicit or internal lock.

Note System exceptions 1146 *(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 Behavior

Consider the following method executions.

method1();

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

method2();

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

method3();

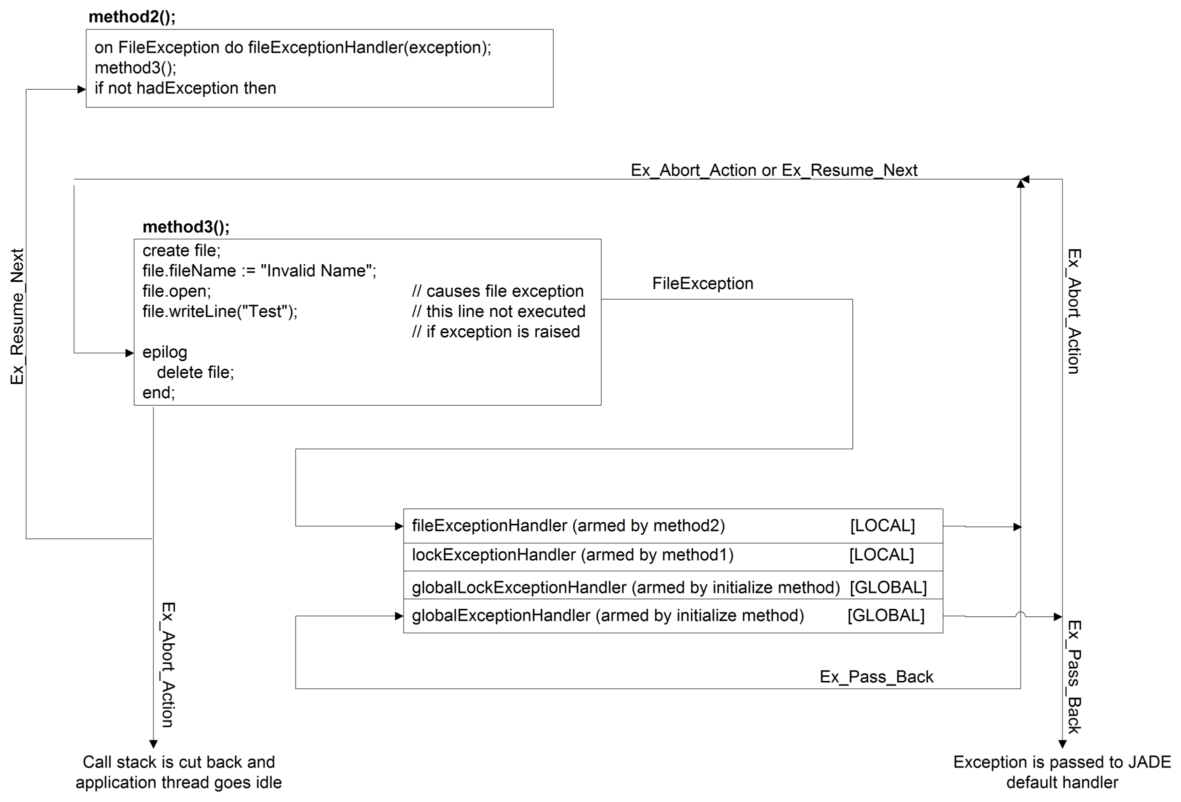
```
create file;
file.fileName := "Invalid Name";
file.open;                              // causes file exception
file.writeLine("Test");

epilog
   delete file;
end;
```

At the time that **method3** is executing, the exception handler stack could look like the following.

```
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 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, depending on the return code returned by the exception handler.


method2();
on FileException do fileExceptionHandler(exception);
method3();
if not hadException then
Ex_Abort_Action or Ex_Resume_Next
Ex_Resume_Next
method3();
create file;
file.fileName := "Invalid Name";
file.open; // causes file exception
file.writeLine("Test"); // this line not executed
// if exception is raised
epilog
delete file;
end;
FileException
Ex_Abort_Action
fileExceptionHandler (armed by method2) [LOCAL]
lockExceptionHandler (armed by method1) [LOCAL]
globalLockExceptionHandler (armed by initialize method) [GLOBAL]
globalExceptionHandler (armed by initialize method) [GLOBAL]
Ex_Abort_Action
Ex_Pass_Back
Ex_Pass_Back
Call stack is cut back and application thread goes idle
Exception is passed to JADE default handler

Referring to the above image, 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 back off 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

This section contains the following topics.

- File Exception Handling
- Connection Exception Handling

- Lock Exception Handling
- Last-Chance Exception Handling

File Exception Handling

File exceptions occur when a method attempts an invalid operation on a file; for example, 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**. The examples in the following subsections 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; that is, 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
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;
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

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, the **isDatabaseClosed** method that follows checks 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

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

Connection exceptions are a little different from other types of exceptions, because they often happen on an asynchronous thread. For example, if your connection object performs a **readBinaryAsynch** method call, 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** call may sit on the **read** instruction for minutes or hours until the connection breaks, for example, at which time Jade raises a **ConnectionException**.

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, 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 the **TcpIpConnection** class 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 (that is, retry the lock) and having obtained the lock, continue the method that caused the exception.

You can use the following code as a basis for a lock exception handler. This code retries the lock for a reasonable number of times, but you may want to code it to retry for a maximum period of 60 seconds, for example. 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 The **global.isValidObject** statement doesn't guarantee that the object will be there at the next reference. In this example, there is still a small window where the **targetLockedBy** process could have gone away. In a highly active system, you might need to cater for this possibility.

Last-Chance 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.

Caution If your application is a background client running on a server, you should avoid the 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.

1. Writes a diagnostic application state dump to log file
2. Optionally advises our monitoring system via SNMP message
3. Aborts the transaction and releases any locks
4. 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
5. 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 are not covered in this white paper. One of these is the issue of context; that is, 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 specific time. Of course, this means that you must maintain the context object, which is a substantial and error-prone overhead.

The Jade Platform provides optional exception handler parameters that make it easier for you to manage exception context. For details, see "Optional Exception handler Parameters", in the following subsection.

Optional Exception Handler Parameters

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.

Optional exception handler parameters do more than just keeping track of context. For example, local exception handlers can have a boolean parameter **exceptionOccurred**, which the handler can set to **true**. If the handler returns **Ex_Resume_Next**, the Jade method that armed the handler can test the value of **exceptionOccurred**, to determine if there was an error.

For more details and examples of contents of the parameter expressions passed to the global and local exception handler methods, see "Creating an Exception Handler" in "Chapter 3 – Exceptions", of the *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, your designer or project leader needs to define and communicate a strategy to your 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; that is, correct system behavior is *assumed*. Errors are dealt with separately, as exceptions.

  This approach enhances system efficiency, because checking for errors (for example, 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; that is, classes should be responsible for precondition checks (that is, 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 needs only to 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 fewer 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 the GUI classes, for example.

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

  If you want to add additional information to the exception objects that you raise, 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 specific type of exception that is to be directed to your handler, rather than all exceptions.

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

  Note When you instantiate **NormalException** or a subclass of it, the 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)*. For more details, see the **Ex_Continue** return value, earlier in this document.

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 you to meet desirable object-oriented development objectives such as maintainability, reusability, and encapsulation, while achieving high levels of system performance and reliability.