Audit Access White Paper

VERSION 2018

jade

Copyright©2018 Jade Software Corporation Limited. All rights reserved.

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 © 2018 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.

For details about other licensing agreements for third-party products, you must read the JADE **ReadMe.txt** file.

Contents

Contents iii

JADE Audit Access 4

Looking Back 4
Why Would You Do This? 5
JadeAuditAccess Class 6
Setting Up in Advance 7
Reading Journals 8
Filtering 10
Classes and Collections 10
Date/Time Range 11
Collection Blocks, Blobs, and Slobs 11
Accessing Description Information 11
Reading the Journal 12
Processing a Journal Record 13
Miscellaneous 16
Accessing Property Attributes 16
Description File Did Not Load 17
Warning 17
Why Do I Get 3125 Exceptions Following a 3036 Exception? 17
Is It Safe to Access the Current Write Journal? 17
How Does DB_FILE_EOF Differ From an AuditSwitchRecord? 18
Putting Things Together 18
JADE Objects 18
Transactions 19
Sample Journal Dump Method 19
Journal Files 24
Journal 25
Description 26
Journal File Structure 26
Glossary 28

JADE Audit Access

JADE maintains details of all changes made to the database, and all significant events that occur to the database, in a journal. This journal is used to:

- Recover the database to a consistent state after a crash (crash or restart recovery)
- Undo the effects of failed or incomplete transactions (abort recovery)

For practical purposes, the journal is written as a sequence of files. The set of files containing any information about current database activity is known as the *current* journal set, and the others as the *archive* journal set.

The archive journal set can be moved to another location for backup, security, or just to conserve disk space; however, the files may still be required by JADE to:

- Roll the database forward from a backup image (copy) to recover subsequent changes (roll-forward recovery)
- Pass change information to a remote location to maintain a duplicate copy of the database (Synchronized Database Service and Relational Population Service)

Information about what has changed in the database may also provide valuable data relating to:

- Security – who changed the data
- Audit – what data was changed and what it was changed to
- Extracting change information
- Performance – analysis of session and transaction times
- Data life times – identification of static and dynamic data

The **JadeAuditAccess** class is a tool to extract information from the JADE journal. This paper discusses some aspects of developing an application to realize this potential.

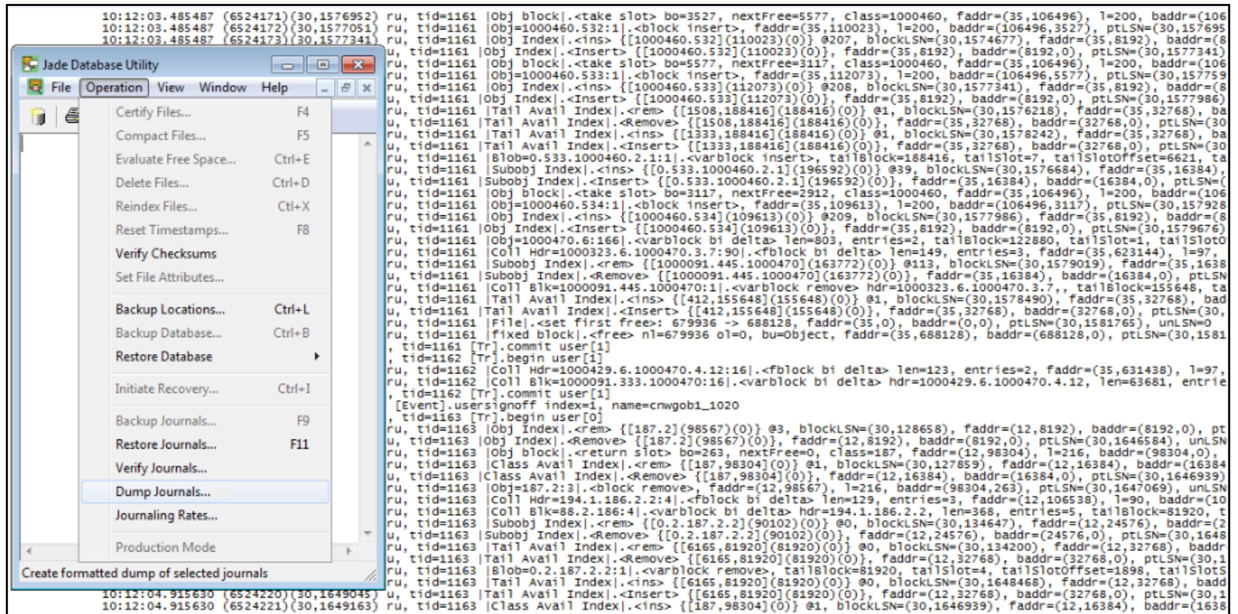
Looking Back

The JADE journal is written in an optimized format to provide an efficient balance between conservation of disk usage and processing requirements. The result of this is that you are not able to just open any of the journal files and read them as you would a text file. This is because the journal uses numbers almost exclusively to identify anything of significance, such as classes, objects, transactions, and record types. These numbers, along with dates, times, and most other items, are stored in a binary format with an implicit length. There are no visible separators to show where one number ends and another starts, and there is no index to show what a number may actually mean.

To understand what is in the journal, you need to know two collections of information:

- The journal and journal record structure
- The JADE object specification

The JADE Database utility can read and understand the structure of a journal and of the journal record. The **Dump Journal** command of the Operation menu analyzes a journal file and produces a disk file displaying the journal record control information. The object data is not displayed.



An examination of the different records and the relationships between them provides useful background knowledge for when you start accessing the journal to retrieve user information.

A user-developed Change Tracker module (in the JADE 5.2.8 release) retrieved very basic information about changes to the database. As the journal was read, records were extracted to build a list of:

- Transaction state changes: Begin/Commit/Abort-Transaction
- Object/collection changes: OID, timestamp, transaction-id, and so on, of creates, updates, and deletes

With the OID, you could access the database to retrieve the current database object – not the value of the object at the time of the audit entry. There was no indication of what properties were changed, or how they changed.

In JADE 6.0.19, the **JadeAuditAccess** class was provided to enable direct JADE access to the journal. In addition to control and state information, the before-image and after-image of objects were retrieved for you to process as required. A dynamic object was constructed, containing only the properties that had changed between the before-image and after-image. This provided an improved view of the journal, but it was still not ideal. Blob and Slob properties were not populated, and it was difficult to match the actual format (the *old* version) of the object audited with a description of the class in the current database specification as changes were made.

Beginning with the JADE 6.1 release, the **JadeAuditAccess** class enables you to extract information from a selected range of the JADE journal and to correlate it with the JADE object specification at the time the journal was written.

Beginning with the JADE 2018 release, access to **Set**, **ObjectArray**, and **MemberKeyDictionary** collection block changes is provided.

Why Would You Do This?

JADE maintains details of all changes made and significant events that occur to the database in a journal.

The original purpose for which the JADE journal was created is to enable JADE to:

- Recover the database to a consistent state after an abnormal termination (a crash or restart recovery)
- Undo the effects of failed or incomplete transactions (abort recovery)

The purpose of the journal was then extended to enable JADE to:

- Roll the database forward from a backup image (copy) to recover changes made after the backup copy was captured (archival or roll-forward recovery)
- Pass change information to a remote location to maintain a duplicate copy of the database (Synchronized Database Service and Relational Population Service)

Users have been discovering that the information in the JADE journal is useful for more than the purposes for which it was originally intended. Indeed, the journal is a valuable information source in several different ways, as follows:

Feature	Enables you to…
Audit	Discover which properties of which objects have been changed, what they changed from, and what they changed to
Data life times	Determine which properties of a class are static, and which are modified more frequently than a specified threshold rate
Diagnostics	Look at what data has been changed, to determine exactly what a transaction did
Extraction	Identify which objects have been modified, to assist in populating or updating an external database
Independence	Find out what is changing in your database without relying upon the application code
Performance	Analyze what is included in a session, measure how long a transaction takes to process, and calculate the transaction-processing rate
Security	Find who initiated a transaction that changed a specific property of a selected object
Understanding	Examine how the data is used, to provide insights into data relationships and business procedures

If you have found another use for the journal information, drop us a note and tell us about it. You can e-mail us at jadesupport@jadeworld.com.

JadeAuditAccess Class

The **JadeAuditAccess** class provides methods to open and read JADE journal files. As a file is read, the relevant database description required to interpret the version of the class objects in the journal file is loaded automatically.

A filtering mechanism is provided to reduce the number of records that need to be processed by the user logic. The data from user class objects is fetched only when it is needed, and it is returned to user logic using a set of tailored methods.

For more details, see the following subsections.

Setting Up in Advance

The format of the JADE journal has changed and been enhanced over time with the release of new capabilities. The **JadeAuditAccess** class requires the most-recent journal format, which uses what is known as *delta logging*. An object update is represented using a *BI-delta* comprised of a before image, and delta information for the parts of the object that have changed. When an after-image is required, it is computed by applying the change information to a copy of the before-image. This differs from what may be regarded as the more traditional approach in which the before-images and after-images are written to the journal.

Note Delta logging has been enabled by default since JADE release 7.0.

A description file of the JADE data structure is required to interpret the structure of the user data. Whenever the format of a persistent object being audited differs from a previous format, a new description file is required. Generally, the format change causes a reorganization of part of the database, and the required description file is created automatically at the completion of a reorganization.

Note The description files can be automatically created at the completion of each reorganization. This is selected in your JADE initialization file, as follows.

```
[PersistentDb]
UseJournalDescriptions = true
```

When a new class is created or when an existing class with no persistent objects is modified, the change is accepted without a reorganization. A new description file is required before instances of the class are created if the journal is to be fully interpreted. In this case, the description file must be created manually, using the **JadeAuditAccess** class method, **generateDescription**; for example:

```
myJadeAuditAccess.generateDescription();
```

When you first want to start accessing your journal files, you may not have any description file.

Note If you have not completed a reorganization, you will need to use the above manual process to create your initial description file.

The description file is written into the journal's root directory with the name:

```
descriptionyyyymmddhhmmss.txt
```

The *yyyymmdd* value is the date and the *hhmmss* value is the time. The creation event is audited with the description file identification timestamp. This timestamp is also saved in the database control file. When a new journal file is first used, the most-recent description file timestamp is written in the journal file header record. The result of this procedure is that each journal file contains references to any description files required to interpret the object data audited in that journal file.

The **JadeAuditAccess** class instance recognizes the description file timestamp in an audit file header record and in any subsequent record announcing the creation of a new description file, and it automatically loads (or reloads) the identified description file if it is available.

Reading Journals

Journals are read using an instance of the **JadeAuditAccess** class.

The first journal is opened using the **getJournal** method. Subsequent journals are opened using the **getNextJournal** method. A JERR_DbFileNotFound (3036) exception is raised when a journal open fails.

The following example processing loop illustrates this method of processing journals.

```
getJournalExceptionHandler(e: Exception;
                           gotJournal: Boolean io;
                           gotJournalMsg: String io): Integer;
vars
begin
    if e.errorCode = JErr_DbFileNotFound then
        gotJournal:= false;
        gotJournalMsg:= 'no next journal';
        return Ex_Resume_Next;
    else
        return Ex_Pass_Back;
    endif;
end;

dump_journals();
vars
    jaa                 : JadeAuditAccess;
    offset              : Integer;
    journalNumber       : Integer;
    startTime           : TimeStamp;
    endTime             : TimeStamp;
    recordType          : Integer;
    recordObjectType    : Integer;
    utcRecordTimestamp  : TimeStamp;
    utcBias             : Integer;
    recordTimestamp     : TimeStamp;
    recordSerialNumber  : Decimal[12,0];
    recordTranId        : Decimal[12,0];
    recordOid           : String;
    recordClassNumber   : Integer;
    recordEdition       : Integer;
    journalsPath        : String;
    descriptionsPath    : String;
    propertyNames       : JadeIdentifierArray;
    valueBefore         : String;
    valueAfter          : String;
    property            : String;
    beforeLen           : Integer;
    beforeBuff          : Binary;
    afterLen            : Integer;
    afterBuff           : Binary;
    gotJournal          : Boolean;
    gotJournalMsg       : String;
    log                 : File;
    jda                 : JadeDatabaseAdmin;
    s                   : String;
    rslt                : Integer;
```

```
begin
    create jda transient;
    journalsPath:= jda.getCurrentJournalDirectory();

    create log transient;
    log.usePresentationFileSystem:= false;
    log.fileName:= "E:\temp\jaa_log.txt";
    log.allowReplace := true;
    log.kind := File.Kind_ANSI;
    log.mode := File.Mode_Output;
    log.open;

    s:= "journal file path is " & journalsPath;
    log.writeLine(s);
    create jaa transient;
    create changedPropertyNames transient;

    // starting with this journal
    journalNumber:= 30;

    gotJournalMsg:= "got next journal";
    gotJournal:= true;
    on Exception do getJournalExceptionHandler(exception, gotJournal,
gotJournalMsg);
    jaa.getJournal(journalsPath, journalNumber, offset);
    if not gotJournal then
        gotJournalMsg:= "no journal file " & journalNumber.String;
        write gotJournalMsg;
        log.writeLine(gotJournalMsg);
    endif;

    while gotJournal do
        log.writeLine("Processing journal " & jaa.getJournalNumber().String);
        if jaa.descriptionTS.isValid then
            s:= "description file = " & jaa.descriptionPath &
jaa.descriptionFilename;
        else
            s:= "description file is invalid";
        endif;
        log.writeLine(s);

        while jaa.getNextRecordUTC(recordType, recordObjectType, offset,
                                   utcRecordTimestamp, utcBias,
                                   recordTimestamp, recordSerialNumber,
                                   recordTranId, recordOid,
                                   recordClassNumber, recordEdition)
        do
            // something
        endwhile; // getNextRecord

        jaa.getNextJournal();
        log.writeLine(" ");
        log.writeLine(gotJournalMsg);
    endwhile; // gotJournal
```

```
        log.close;

    epilog
        delete log;
        delete jaa;
        delete changedPropertyNames;
        delete jda;
    end;
```

Filtering

It is fairly common to want to access only some of the information in the journal file. A set of methods allows you to specify some common filtering conditions. More-detailed conditions must by coded in your logic to be tested after journal records have been retrieved.

Note Any filtering condition (except for the start of time-range) can be changed at any time while the journal is being accessed. The changed conditions are used during subsequent journal reading.

For more details about filtering, see the following subsections.

Classes and Collections

The **JadeAuditAccess** class provides methods that register a set of classes that can be either allowed or disallowed as the journal is read.

The **registerFilterClass** or **registerFilterClassName** method can be used to register a class. The **registerFilterCollection** or **registerFilterCollectionName** method can be used to register a collection.

By default, only objects in the journal of the registered classes or collections are returned, but you can reverse this condition with the **setFilterExcludes** method.

The **registerFilterClass** and **registerFilterCollection** methods use class numbers to specify the classes. These can be used before or after the journal file is opened.

```
// clear any previously registered filters
jaa.clearRegisteredFilters();

// register classes and collections
rslt := jaa.registerFilterClass(2093);
rslt := jaa.registerFilterClass(2087);
rslt := jaa.registerFilterCollection(2813, 0);

// exclude the registered classes and collections
rslt := jaa.setFilterExcludes(true);
```

Note Classes and collections identified by class number can be registered either before or after the journal file is opened.

The **registerFilterClassName** and **registerFilterCollectionName** methods provide similar functions to the **registerFilterClass** and **registerFilterCollection** methods respectively, but accept more meaningful class names rather than class numbers. These methods interrogate the description file to find the actual class numbers associated with the names specified, and so can be used only after the description file has been loaded.

```
// register classes and collections by names
rslt := jaa.registerFilterClassName('ModelSchema', 'ItemLine');
rslt := jaa.registerFilterCollectionName('ModelSchema', 'Invoice', 'items');
```

Note The schema, class, and property names can be used only after the journal file is opened and the description file has been loaded.

Date/Time Range

You can specify an inclusive date/time range as a filter condition, by specifying a start and end timestamp. If either timestamp is null, the date/time range is unbounded at the corresponding end of the range.

The start timestamp is used while the journal file is being opened, so this must be specified prior to the **getJournal** method opening the journal. The end timestamp can be specified at the same time, or at some later time after the journal is open.

```
// if accessing a particular time range
startTime.setDate('27 August 2016'.Date);
startTime.setTime('16:05:00'.Time);
endTime.setDate('27 August 2016'.Date);
endTime.setTime('16:15:00'.Time);
rslt := jaa.registerFilterTimeRange(startTime, endTime);
```

Note The **registerFilterTimeRange** method registers the range as local time. A UTC date and time range can be registered by using the **registerFilterTimeRangeUTC** method.

Collection Blocks, Blobs, and Slobs

Many applications that read the journal just want to access some of the information stored in the record of an object (for example, a unique identifier), so the contents of collection blocks, blobs, or slobs are not required.

The **JadeAuditAccess** class ignores any collection block, blob, or slob records unless you specify that you want this data retrieved, using the **setAccessMode** method.

```
// to ignore collection block, blob, and slob information (default)
jaa.setAccessMode(jaa.Jaa_AccessMode_Standard);

// to retrieve collection block, blob, and slob information
jaa.setAccessMode(jaa.Jaa_AccessMode_Long);
```

Note Collection block information is retrieved only for collections of class **Set**, **ObjectArray**, and **MemberKeyDictionary**, and their subclasses.

Accessing Description Information

Once the description file is loaded, you can use the methods listed in the following table to interrogate information from the JADE system that created the journal and description files.

Method	Returns…
descriptionClassIsSubclass	**True** if the class number specified in the **classNumber** parameter is a subclass of the class number specified in the **superclassNumber** parameter in the loaded description
getBlobProperty	A description of the Blob or Slob property from its OID value
getClassName	The schema and class name of a class from its class number
getClassProperty	The attributes of the property from its class number and property name

Method	Returns…
getClassPropertyNames	A list of the names of all the properties of the specified class
getProperty	The attributes of the property from its property name and the class of the currently accessed journal record

The **JadeAuditAccess** properties listed in the following table provide information about the description file itself. These are updated each time a new description file is read.

Property	Description
descriptionFilename	The file name of the description file
descriptionPath	The description directory path
descriptionTS	The creation timestamp of the description file

Reading the Journal

The objective of the **JadeAuditAccess** class interface is to return any journal information concerning the user classes, objects, and transactions that are permitted by the registered filter conditions. The approach taken is that you request that the next record be read. You get a result that tells you what type of record has been read and some journal and object identification information. You can then use other methods to retrieve particular information from this record, depending upon its type.

The **JadeAuditAccess** class **getNextRecord** method is used where only the local timestamp values are required, as shown in the following code fragment.

```
while jaa.getNextRecord(recordType, recordObjectType, offset, recordTimestamp,
                        recordSerialNumber, recordTranId, recordOid,
                        recordClassNumber, recordEdition) do
    // process Journal record
    // ...
endwhile;
```

When UTC timestamp values are required, the **getNextRecordUTC** method is used, as shown in the following code fragment.

```
while jaa.getNextRecordUTC(recordType, recordObjectType, offset,
                           utcRecordTimestamp, utcBias, recordTimestamp,
                           recordSerialNumber, recordTranId, recordOid,
                           recordClassNumber, recordEdition) do
    // process Journal record
    // ...
endwhile;
```

The **JadeAuditAccess** class **getNextRecord** and **getNextRecordUTC** method read the journal file and find the next record that satisfies the filtering conditions. While locating this record, the read routine may notice a control record recording the creation of a new description file. If this occurs, the new description file is loaded.

The possible journal record types available are listed in the following table.

JadeAuditAccess Class Constant	Description
Jaa_Type_DatabaseOpen	Database open
Jaa_Type_DatabaseClose	Database close

JadeAuditAccess Class Constant	Description
Jaa_Type_NoAuditDiscontinuity	NoAudit discontinuity
Jaa_Type_ReorgDiscontinuity	Reorganization discontinuity
Jaa_Type_UserSignOn	User sign-on
Jaa_Type_UserSignOff	User sign-off
Jaa_Type_BeginTransaction	Transaction begin
Jaa_Type_CommitTransaction	Transaction commit
Jaa_Type_AbortTransaction	Transaction abort
Jaa_Type_Create	Object create
Jaa_Type_Update	Object update
Jaa_Type_AuditSwitch	Audit switch
Jaa_Type_Delete	Object delete

The **Database** and **Discontinuity** record types provide a little state information of what is happening to the database as a whole.

The **User sign-on** and **User sign-off** records provide user identification if you want to track changes by user. Use the **getUserData** method to retrieve the data associated with these records.

The **Transaction** records provide information that enables individual object changes to be grouped into logical units. You *do* need to be aware of transactions as you retrieve information from the journal, as some changes you retrieve may later be reversed if the transaction is aborted. All of the records associated with a specific transaction will have the same transaction-id (**tranId**) value.

The **Object** records provide details of changes to the contents of the database. The object may be a **Class** instance, **Collection** instance, a collection block instance, or a blob/slob instance. You will get information only about objects of user classes; objects of JADE system classes are automatically suppressed.

Processing a Journal Record

There are two data values returned from the **getNextRecord** and **getNextRecordUTC**methods that will help you structure the processing of the records retrieved; the record type (**recType**) and the object type (**objType**).

The record type tells you what record has just been encountered in the journal. You can use this to select the records you want to process further, or you can use it to decide how a specific record is to be processed.

The object type tells you what sort of data is documented by the journal record. If you are interested (or not interested) in objects, collections, collection blocks, or blobs/slobs, you can use the object type to quickly separate the wanted from the unwanted, as shown in the following example.

```
// process Journal record
if recordObjectType = jaa.Jaa_Object_Object then
    write 'Class ' & jaa.getClassName(recordClassNumber) & '
             (' & recordClassNumber.String & ') object: ' & recordOid;
    if recordType = jaa.Jaa_Type_Delete or recordType = jaa.Jaa_Type_Update then
        // process before-image
        // ...
    elseif recordType = jaa.Jaa_Type_Create then
        // process after-image
        // ...
```

```
        elseif recordType = jaa.Jaa_Type_Update then
            // process Changes
            // ...
        endif;
    elseif recordObjectType = jaa.Jaa_Object_Blob then
        // process a blob or slob
        // ...
    endif;
    elseif recordObjectType = jaa.Jaa_Object_Collection_Block then
        // process a collection block
```

All of the journal records contain a timestamp to document when a change was made. The timestamp is recorded in Universal Time rather than the local time on the computer creating the journal. The journal records also contain the UTC bias, measured in minutes, at the time the record was created. The **getUTCBias** method returns the UTC bias value from the journal header; that is, the bias when the journal was created.

The **getNextRecord** method returns the local timestamp. The **getNextRecordUTC** method returns the local timestamp, the UTC timestamp, and the UTC bias value.

Each **Object** record has (logically) a before-image, an after-image, and a record of changes (delta-image). Similarly, collection block and blob (or slob) records logically contain a before-value and an after-value. A create record has a null before-image/value, and a delete record has a null after-image/value.

Retrieving most information is dependent upon having the description file loaded to provide the knowledge of the structure of the objects, and the names of classes and properties.

If the description file was not able to be loaded for any reason, you can continue to read the journal, but any attempt to access anything except **Set** or **ObjectArray** collection block changes, or **Object** before-images and after-images, and the blob/slob values will not succeed.

The **JadeAuditAccess** methods that retrieve object or blob (or slob) journal record information are listed in the following table.

Method	Description
getAfterImage	Retrieves a raw **Binary** copy of the after-image of the object, which can be accessed even when no description file is available. It may be difficult to identify individual object properties.
getAfterPropertyValue	Retrieves the value of a property from the after-image.
getBeforeImage	Retrieves a raw **Binary** copy of the before-image of the object, which can be accessed even when no description file is available. It may be difficult to identify individual object properties.
getBeforePropertyValue	Retrieves the value of a property from the before-image.
getBlobValue	When the object is a **Jaa_Object_Blob**, this retrieves the entire blob or slob value from both the before-value and after-value.
getChangedPropertyNames	Retrieves a list of the names of properties whose values differ between the before-image and after-images of the object.

Note Use methods to retrieve information from the **Object** records. The value of the **Object** is not returned as a JADE object because the JADE application reading the journal file is unlikely to have a **Class** with properties that match the **Object** being retrieved. Even if the JADE application is reading its own journal, the classes could have been modified and reorganized since the journal was created, and so would no longer match the (old) object being retrieved.

The methods that retrieve collection block journal record information are listed in the following table.

Method	Description
getCollectionBlockKeys	Populates the passed array with dynamic objects describing the additions, removals, and changes in this edition of the **MemberKeyDictionary** collection block
getCollectionBlockOid	Returns the string representation of the oid of the collection block record
getCollectionBlockOids	Populates the passed added and removed oid arrays with the string representations of the oids added and removed in this edition of the collection block

The value of a **Reference** property is the oid of the object referenced. This oid is really only meaningful in the JADE database in which the object resides. When the value of the property is requested, a string representation of the oid value will be generated and returned.

In an object record, the value of a blob or slob property is just the current length and the edition value. When the value of a slob property is requested, a String containing the length of the slob and its edition will be generated and returned. When the value of a blob property is requested, a similar string is generated but it is returned as a Binary value.

```
// get properties
jaa.getClassPropertyNames(recordClassNumber, propertyNames);
// process before-image
beforeBuff := jaa.getBeforeImage;
write 'Before Image, length: ' & beforeBuff.length.String;
write beforeBuff.String.hexDump;
// all properties
foreach property in propertyNames do
    write property & Tab & jaa.getBeforePropertyValue(property).String;
endforeach;
// process after-image
afterBuff := jaa.getAfterImage;
write 'After Image, length: ' & afterBuff.length.String;
write afterBuff.String.hexDump;
// all properties
foreach property in propertyNames do
    write property & Tab & jaa.getAfterPropertyValue(property).String;
endforeach;
// process changes
write 'Changes to ' & recordOid;
// get changed properties
jaa.getChangedPropertyNames(propertyNames);
foreach property in propertyNames do
    write property & Tab & jaa.getAfterPropertyValue(property).String;
endforeach;
```

While you are examining a property of the current object, you may require some knowledge about the property itself. This is available from the **getProperty** method of **JadeAuditAccess**, which interrogates the description file for the information.

```
// get the property's attributes
jaa.getProperty(property, propertyType, propertyLength, propertyPrecision,
                propertyScale, referenceToClassNum);
```

This method does not require any class specification, as the class associated with the currently retrieved audit record is assumed.

When processing a blob or slob record, both the before-image and the after-image are retrieved from a single call to the **getBlobValue** method. The earlier boolean output parameter is always **false** (since the JADE 7.0 release).

While you are processing a blob or slob record, you may want to identify the parent class or actual parent object, and you may need to identify which property of the parent class the blob/slob is. This information is available from the **getBlobProperty** method, as shown in the following example.

```
// get the property's attributes
if jaa.getBlobProperty(jaa.currentOid, parentClassNum, parentOID,
                       propertyName, BlobSlobType) then
    // ... (Blob-type = 11; Slob-type = 1)
else
    // description file is not available, or jaa.currentOid is not a blob or slob
endif;
```

When processing a collection block record, the **getCollectionBlockOids** method can be used to get added and removed oids. In addition, if the collection is a **MemberKeyDictionary**, then added and removed oids and oids with changed keys can be retrieved with keys information using the **getCollectionBlockKeys** method.

See "Sample Journal Dump Method" under the "Putting Things Together" section for an example method that accesses changed property values and changed collection block entry information.

Note As **Set** and **MemberKeyDictionary** are derived from **Btree**, collection block modifications can also result from block reduce or block split operations that move entries.

Miscellaneous

This section contains the following miscellaneous information about accessing audit records.

- Accessing Property Attributes
- Description File Did Not Load
- Warning
- Why Do I Get 3125 Exceptions Following a 3036 Exception
- Is It Safe to Access the Current Write Journal
- How Does DB_FILE_EOF Differ From an AuditSwitchRecord

Accessing Property Attributes

If you want to be able to make decisions in your logic based upon some attributes of an object's property (for example, formatting its value in particular ways depending upon its type), you can request the property's attributes if the description file is loaded.

```
// for a property of a particular class number
jaa.getClassProperty(classNum, propName, propType, propLength,
                     propPrecision, propScale, propRefClassNum);
// for a property of the current journal record's object
jaa.getProperty(propName, propType, propLength,
                propPrecision, propScale, propRefClassNum);
```

The **propType** parameter returns a **Type** object that specifies what type of property you are examining.

The **propPrecision** and **propScale** parameters are used if the property is a **Decimal** type.

The **propRefClassNum** property is the class number associated with a **Reference** property.

These methods can be used in conjunction with the **getClassPropertyNames** and **getChangedPropertyNames** methods, which return a list of property names associated with a specified class number or of the current journal record's object.

Description File Did Not Load

The description file may change any time you open a journal file or get another record from the file. If the specified file is not able to be automatically loaded, the **descriptionTS** timestamp is set to null. You *may* want to check this and attempt to recover in some way, or just terminate your logic. (If you continue processing, most methods that attempt to access data return unexpected results.)

One way to attempt to recover is to attempt to manually load a description file.

```
if jaa.descriptionTS = null then
    jaa.loadDescription();
endif;
```

The **loadDescription** method presents a file-open dialog for you to specify a file to be used as the description file, or to cancel the file load operation. This method has a side-effect of also re-setting the **autoDescription** property of **JadeAuditAccess**, which will disable subsequent automatic loading of a new description file. Instead, when a subsequent description file is specified in the journal, you are presented with a file-open dialog, pre-filled with the file name of the specified file. You can choose to accept that file to be loaded, select some other file, or cancel the file load operation.

Warning

The internal representations used to store journal information and to maintain class descriptions for accessing a JADE journal are not documented. The actual formats will be modified as required within the JADE product. You should not write any code that makes any assumptions about any format that is not documented.

Why Do I Get 3125 Exceptions Following a 3036 Exception?

Methods such as **getNextJournal** and **getNextRecord** rely on there being an open journal. The journal number reflects the currently open journal and is in an initial state (**0**) when no journal is open. When the journal number is zero (**0**), any attempt to call a method that relies on there being an open journal will raise a 3125 exception (*A required transaction journal was not found*). If a **getNextJournal** method call fails because of a 3036 exception (*The database file being opened is required but not found*), the journal number will be set to zero (**0**) and therefore subsequent method calls may raise a 3125 exception. In this situation, a successful **getJournal** method call is necessary to open a journal.

Is It Safe to Access the Current Write Journal?

Care must be taken when accessing the current write journal. The database journals exist primarily for database recovery. As such, the mechanisms for their generation and access are optimized for performance. A journal file is pre-initialized before it is switched to, with the file length established at **JournalMaxSize** and unused file space set to zeros. The current write journal is a carefully crafted set of records, with the most-recent records being buffered by the database in memory. It is a fluid data stream that is being written to disk, with the most currently written sector possibly being rewritten multiple times. The database engine, as the generator of the data stream, is intimately aware of all of the intricacies that this involves and behaves appropriately. The journal is written to disk in sector-aligned blocks without the involvement of file system cache and the end of the journal (as accessible from disk at the last-written sector) is most often a partial or "torn" audit record. Applications cannot rely on the integrity of records read from the "hot" area being written at the end of the file. Therefore applications reading the current write journal cannot reasonably "chase the tail" of the audit as it is being streamed to disk.

In addition to the **closeCurrentJournal** method, the **JadeDatabaseAdmin** class also provides the **getCurrentJournalNumber** method so that an application can always know the number of the current write journal. An application can also subscribe to the **JournalTransferEvent** event, which is notified every time a journal switch occurs and contains the number of that journal (which is no longer the current write journal).

In-band applications processing the database audit stream should avoid attempting to process the current journal by using these mechanisms. For example, when processing the audit switch at the end of a journal (or end of file), if that journal number is the database current journal number less 1, an application should delay the **getNextJournal** call until it receives the appropriate journal transfer event. Alternatively, an application might maintain a "maximum allowable journal to process" number (that is initialized from the database current journal number and then maintained from journal transfer events) to manage **getNextJournal** method calls appropriately.

The switching behavior of journals can be tailored using the **JournalMaxSize**, **JournalMinSize**, and **JournalSwitchInterval** JADE initialization file parameters in the [PersistentDb] section. The **JournalMaxSize** parameter caters for switching when reaching a size based on transaction activity. The **JournalMinSize** and **JournalSwitchInterval** parameters allow interval-based switching. When a journal switch interval is specified, the database checks the current journal size at the expiration of each interval. If it is larger than **JournalMinSize**, an "interval" switch is forced.

If an application needs up-to-date access to stable journals within a specific amount of time (for example, half-hourly), it can make the journal switch every half hour using a timer and the **closeCurrentJournal** method of the **JadeDatabaseAdmin** class. It can then process up to the end of that known journal. Alternatively, it can wake up half-hourly and process all available journals (using **getNextJournal**) while the open journal number is not equal to **currentJournalNumber**, after which it would go back to sleep.

How Does DB_FILE_EOF Differ From an AuditSwitchRecord?

DB_FILE_EOF means you have reached the end of the journal file and no record values are returned in the call. However, an **AuditSwitchRecord** is a true record, so the call returns with valid values. This is useful for terminating a date/time application read loop, for example. An **AuditSwitchRecord** can also be processed by the application as the equivalent of end of file, as it is always the last valid record in a stable (that is, switched from) journal.

Putting Things Together

When reading the JADE journal, you can access the details of one journal record at a time. This does not accurately reflect how your data exists or is processed, and you may need to accumulate information from multiple records to be able to process the journal information in a way that satisfies your requirements.

For more details, see the following subsections.

JADE Objects

A JADE object comprises an instance of each property defined in the associated JADE class. In general, the following two groups of properties may be required.

- Any simple properties, such as **Integer**, **Boolean**, and references to other objects. This includes **Binary** and **String** properties with a specified length.
- Any maximum-length **Binary** and **String** properties (Blobs and Slobs, respectively).

In the JADE journal, all simple primitive type properties are grouped into the record known as the object's journal record, identified by the OID. To uniquely identify the actual version of the object, the OID is supplemented by an edition number that is incremented each time the object is modified.

The 'parent' object record contains the length and edition of the Blob/Slob. You can retrieve this Blob/Slob information while you are accessing the parent object journal record. You can determine whether any Blob/Slob value has changed by comparing the edition number from the before-image with the edition number from the after-image. The Blob/Slob property value is not written to the journal if it has not changed.

Each of the Blob or Slob property values is written separately to the journal. The Blob/Slob OID contains sufficient information to identify the parent object OID, and the actual property within that object.

When the Blob/Slob record is encountered in the journal, a Blob- or Slob-type journal record is returned from the **getNextRecord** or the **getNextRecordUTC**method call. (This record can be encountered either before or after the parent object record.) This record provides the OID of the Blob/Slob. You can use the **getBlobProperty** method to obtain the OID of the parent object to which the Blob/Slob belongs, and the name of the Blob/Slob property.

Note Remember the following points.

- Blob/Slob records can be either before or after the parent object record.
- Blob/Slob records are written to the journal only if they are changed.
- Parent object records are written to the journal if a Blob or Slob has changed.

Transactions

In your JADE application, your changes to objects are grouped into transactions. At the end of each transaction, you explicitly commit the changes, causing them to become permanent changes to the objects stored in the JADE database.

To save memory usage and time at the end of a transaction, JADE starts writing changes to the journal in anticipation of an eventual commit-transaction occurring. The transaction is completed when the commit-transaction (or an abort-transaction) is processed.

As we probably well know, there are times when no final commit-transaction or abort-transaction occurs, as a result of a premature termination of the application (or computer). The recovery process recognizes the incomplete transactions, and writes any required abort-transaction records to the journal to preserve the integrity of the information. These appear before the database-open audit record.

When you are reading through the JADE journal, you see records of changes but you will not know whether those changes have become permanent database changes until you find the commit-transaction record for the transaction containing those changes.

Each journal record containing a change also contains a transaction-id number. If your application requires you to process records of changes to the database only, you will need to keep track of all the records for each transaction (there may be multiple transactions interleaved in the journal at any time) and only process the records when you access the commit-transaction record.

If you encounter an abort-transaction record, you discard all the journal records you are keeping track of - for that transaction.

Sample Journal Dump Method

The following example of a sample method accesses changed property values and changed collection block entry information.

```
dumpJournals();
vars
    auditAccess             : JadeAuditAccess;
    offset                  : Integer;
```

```
        journalNumber        : Integer;
        recordType           : Integer;
        recordObjectType     : Integer;
        recordTimeStamp      : TimeStamp;
        recordSerialNumber   : Decimal[12,0];
        recordTxnId          : Decimal[12,0];
        recordOid            : String;
        recordClassNumber    : Integer;
        recordEdition        : Integer;
        changedPropertyNames : JadeIdentifierArray;
        changedProperty      : String;
        valueBefore          : String;
        valueAfter           : String;
        journalsPath         : String;
        s                    : String;
        gotJournal           : Boolean;
        gotJournalMsg        : String;
        log                  : File;
        jda                  : JadeDatabaseAdmin;
        addedOids            : StringArray;
        removedOids          : StringArray;
        oidString            : String;
        count                : Integer;
        collType             : String;
        memberKeyDict        : Boolean;
        jdo                  : JadeDynamicObject;
        jdoArray             : JadeDynamicObjectArray;
        keyBytes             : Integer;
        beforeKeys           : Binary;
        afterKeys            : Binary;
        someKeys             : Binary;
        collOwnerClass       : Integer;
        editionString        : String;
    begin
        journalNumber:= 22;

        create jda transient;
        journalsPath:= jda.getCurrentJournalDirectory();

        create log transient;
        log.usePresentationFileSystem:= false;
        log.fileName:= 'c:\temp\jaa_dump_everything.txt';
        log.allowReplace := true;
        log.kind := File.Kind_Unicode;
        log.mode := File.Mode_Output;
        log.open;

        if not log.isAvailable then
            write 'unable to open output log "' & log.fileName & '"';
            return;
        endif;

        s:= "journal file path is " & journalsPath;
        write s;
        log.writeLine(s);
        create changedPropertyNames transient;
        create addedOids transient;
```

```
create removedOids transient;
create jdoArray transient;

create auditAccess transient;
auditAccess.setAccessMode(JadeAuditAccess.Jaa_AccessMode_Long);

offset := 0;
write '';
gotJournalMsg:= 'got next journal';
gotJournal:= true;
on Exception do getJournalExceptionHandler(exception, gotJournal,
            gotJournalMsg);
auditAccess.getJournal(journalsPath, journalNumber, offset);
if not gotJournal then
    gotJournalMsg:= 'no journal file ' & journalNumber.String;
    write gotJournalMsg;
    log.writeLine(gotJournalMsg);
endif;
while gotJournal do
    s:= 'Processing journal ' & auditAccess.getJournalNumber().String;
    write s;
    log.writeLine(s);
    while auditAccess.getNextRecord(recordType, recordObjectType, offset,
           recordTimeStamp, recordSerialNumber,recordTxnId, recordOid,
           recordClassNumber, recordEdition)
    do
        s:= recordSerialNumber.String & ' ' & recordTxnId.String & '
               recordType=' & recordType.String & ' ';
        if recordType = JadeAuditAccess.Jaa_Type_Create then
            s:= s & 'Create ';
        elseif recordType = JadeAuditAccess.Jaa_Type_Delete then
            s:= s & 'Delete ';
        elseif recordType = JadeAuditAccess.Jaa_Type_Update then
            s:= s & 'Update ';
        elseif recordType = JadeAuditAccess.Jaa_Type_AbortTransaction then
            s:= s & 'Abort transaction ';
        elseif recordType = JadeAuditAccess.Jaa_Type_BeginTransaction then
            s:= s & 'Begin transaction ';
        elseif recordType = JadeAuditAccess.Jaa_Type_CommitTransaction then
            s:= s & 'Commit transaction ';
        elseif recordType = JadeAuditAccess.Jaa_Type_UserSignOn then
            s:= s & 'Sign on ';
        elseif recordType = JadeAuditAccess.Jaa_Type_UserSignOff then
            s:= s & 'Sign off ';
        elseif recordType = JadeAuditAccess.Jaa_Type_ChangeUser then
            s:= s & 'Change user ';
        endif;

        if recordObjectType <> JadeAuditAccess.Jaa_Object_Null then
            s:= s & 'recordObjectType=' & recordObjectType.String & ' ';
        if recordObjectType = JadeAuditAccess.Jaa_Object_Object then
            s:= s & 'Object ';
        elseif recordObjectType = JadeAuditAccess.Jaa_Object_Collection then
            s:= s & 'Collection ';
        elseif recordObjectType = JadeAuditAccess.Jaa_Object_CollectionBlock
               then
            s:= s & 'CollectionBlock of Collection ';  // the oid given to us
```

```
                                // in this case is the header oid
    elseif recordObjectType = JadeAuditAccess.Jaa_Object_Blob then
        s:= s & 'Blob/Slob ';
    endif;
endif;

if recordEdition > 0 then
    editionString := ':' & recordEdition.String;
else
    editionString := '';
endif;

log.writeLine('(' & auditAccess.getJournalNumber().String & ',' &
            offset.String & ') ' & s & recordOid.String & editionString);

if recordType = auditAccess.JadeAuditAccess.Jaa_Type_Update then
    if recordObjectType = auditAccess.JadeAuditAccess.Jaa_Object_Object
               then
        log.writeLine('update of ' & auditAccess.currentOid);
        if auditAccess.getChangedPropertyNames(changedPropertyNames) then
            foreach changedProperty in changedPropertyNames do
                valueBefore:= auditAccess.getBeforePropertyValue
                         (changedProperty).String;
                valueAfter:= auditAccess.getAfterPropertyValue
                         (changedProperty).String;
                log.writeLine(Tab & changedProperty & '[' & valueBefore &
                         '] [' & valueAfter & ']');
            endforeach;
        endif;
    endif;
endif;

if recordObjectType = auditAccess.JadeAuditAccess.
            Jaa_Object_CollectionBlock then
    memberKeyDict := false;
    collType := "";
    if auditAccess.descriptionClassIsSubclass(recordClassNumber,
                ObjectArray.number) then
        collType := "ObjectArray";
    elseif auditAccess.descriptionClassIsSubclass(recordClassNumber,
                Set.number) then
        collType := "Set";
    elseif auditAccess.descriptionClassIsSubclass(recordClassNumber,
                MemberKeyDictionary.number) then
        collType := "MemberKeyDictionary";
        memberKeyDict := true;
    endif;
    collOwnerClass := getClassNumberForObject(getOwnerForObject
                (auditAccess.currentOid.asOid()));
    collType := collType & ' (' & auditAccess.getClassName(collOwnerClass)
                & '.';
    collType := collType & auditAccess(getClassName
                auditAccess.currentClassNumber) & ')';
    if recordType = auditAccess.JadeAuditAccess.Jaa_Type_Create then
        log.writeLine('create ' & collType & ' block=' &
                  auditAccess.getCollectionBlockOid());
    elseif recordType = auditAccess.JadeAuditAccess.Jaa_Type_Update then
```

```
                log.writeLine('update ' & collType & ' block=' &
                           auditAccess.getCollectionBlockOid());
            else
                log.writeLine('delete ' & collType & ' block=' &
                           auditAccess.getCollectionBlockOid());
            endif;
            if memberKeyDict then
                auditAccess.getCollectionBlockKeys(jdoArray);
                foreach jdo in jdoArray do
                oidString := jdo.getPropertyValue('entryOid').String;
                keyBytes := jdo.getPropertyValue('keyBytes').Integer;
                beforeKeys := jdo.getPropertyValue('beforeKeys').Binary;
                afterKeys := jdo.getPropertyValue('afterKeys').Binary;
                if beforeKeys.length() > 0 and afterKeys.length() > 0 then
                    if keyBytes > 32 then
                        s := Tab & 'changed ' & oidString & ' keyBytes=' &
                               keyBytes.String & CrLf & Tab & Tab & 'old key = <'
                               & beforeKeys.String & '>' & CrLf & Tab & Tab &
                               '         0x<' & beforeKeys._hex() & '>' & CrLf &
                               Tab & Tab & 'new key = <' & afterKeys.String & '>'
                               & CrLf & Tab & Tab & '         0x<' &
                               afterKeys._hex() & '>';
                    else
                        s := Tab & 'changed ' & oidString & ' keyBytes=' &
                               keyBytes.String & CrLf & Tab & Tab & ' old key = <'
                               & beforeKeys.String & '>  0x<' & beforeKeys._hex()
                               & '>' & CrLf & Tab & Tab & ' new key=  <'
                               & afterKeys.String & '>  0x<' & afterKeys._hex()
                               & '>';
                    endif;
                else
                    if afterKeys.length() = 0 then
                        s := Tab & 'removed ';
                        someKeys := beforeKeys;
                else    // beforeKeys.length() = 0
                        s := Tab & 'added ';
                        someKeys := afterKeys;
                endif;
                if keyBytes > 32 then
                    s := s & oidString & ' keyBytes=' & keyBytes.String & CrLf &
                               Tab & Tab & 'key = <' & someKeys.String & '>' & CrLf
                               & Tab & Tab & '    0x<' & someKeys._hex() & '>';
                    else
                        s := s & oidString & ' keyBytes=' & keyBytes.String & CrLf
                               & Tab & Tab & ' key = <' & someKeys.String & '>
                               0x<' & someKeys._hex() & '>';
                    endif;
                endif;
                log.writeLine(s);
            endforeach;
            jdoArray.purge();
        else
            auditAccess.getCollectionBlockOids(addedOids, removedOids);
            count := 0;
            s := Tab & '  added';
            foreach oidString in addedOids do
```

```
                count := count + 1;
                s := s & '  ' & oidString;
                if count mod 5 = 0 then
                    log.writeLine(s);
                    s := Tab & '        ';
                endif;
            endforeach;
            if s.length() > 10 then     // residual
                log.writeLine(s);
            endif;
            count := 0;
            s := Tab & 'removed';
            foreach oidString in removedOids do
                count := count + 1;
                s := s & '  ' & oidString;
                if count mod 5 = 0 then
                    log.writeLine(s);
                    s := Tab & '        ';
                endif;
            endforeach;
            if s.length() > 10 then     // residual
                log.writeLine(s);
            endif;
        endif;
    endif;

    endwhile;    // getNextRecord

    auditAccess.getNextJournal();
    write ' ';
    write gotJournalMsg;
    log.writeLine(' ');
    log.writeLine(gotJournalMsg);
endwhile;    // gotJournal

log.close;

epilog
    delete log;
    delete auditAccess;
    delete changedPropertyNames;
    delete jda;
    delete addedOids;
    delete removedOids;
    delete jdoArray;
end;
```

Journal Files

This section contains the following information about JADE journal files.

- Journal
- Description

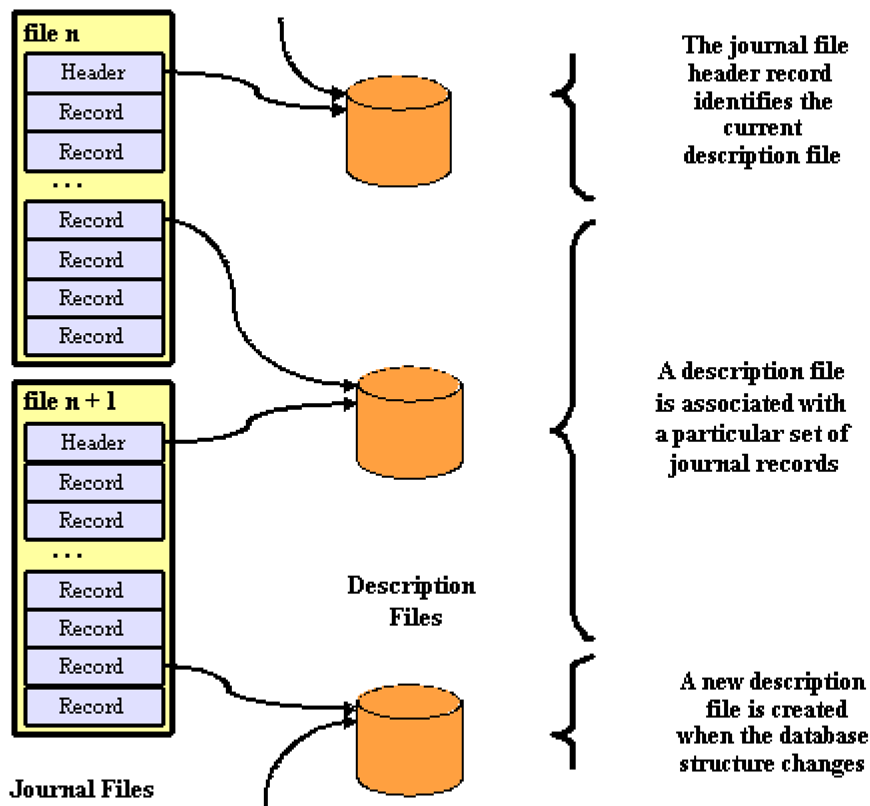
Journal

The JADE journal is also known as the JADE:

- Transaction journal or log
- Audit, audit log, or audit trail
- Recovery log

Note The internal representations used to store journal information and to maintain class descriptions for accessing a JADE journal are not documented. The actual formats will be modified as required within the JADE product. You should not write any code that makes any assumptions about any format that is not documented.

The following image is an example of JADE journal files.



For practical purposes, the journal is written as a sequence of files. The set of files containing any information about current database activity (that is, the information required for restart or abort recovery) is known as the *current journal set*. The other journal files are known as the *archived journal set*.

The JADE journal is written to disk as a sequence of journal files named:

```
dbnnnnnnnnnn.log
```

The *nnnnnnnnnn* value is a 10-digit number in the range 1 through 9999999999.

Note You can use parameters in the [PersistentDb] section of your JADE initialization file to specify the location and size of the individual journal files.

The current journal set (of files) is located in the **\current** subdirectory of the root journal directory.

The archive journal set (of files) is located in the **JournalArchiveDirectory** directory (if specified in your JADE initialization file) or in the **\archive** subdirectory of the root journal directory. These archived files can be moved to another location for backup, security or to conserve disk space, but they may later need to be restored.

For more details about the journal, see the *JADE Database Administration Guide*.

Description

A set of description files is also located in the root journal directory. These files (shown in the image under "Journal", earlier in this document) contain the information required to interpret the information in the journal files. Each of these files contains a snapshot of the definitions of the persistent classes of the JADE system as at a start-time recorded in the file name:

```
descriptionyyyymmddhhmmss.txt
```

The *yyyymmdd* value is the date and the *hhmmss* value is the time.

Note These files are created if the **UseJournalDescriptions** parameter in the [PersistentDb] section of your JADE initialization file is specified.

When the JADE initialization file parameter is specified (the default), the termination process of each database reorganization creates a new description file. You can modify the structure of the database without involving a reorganization, by adding or modifying a class with no persistent objects. In this situation, or to initially create a description file, you can manually create a description file, using the **generateDescription** method of the **JadeAuditAccess** class.

Note Although the description file contains readable text, you should not modify it, as the data it contains is relied upon without any verification.

Journal File Structure

Note The internal representations used to store journal information and to maintain class descriptions for accessing a JADE journal are not documented. The actual formats will be modified as required within the JADE product. You should not write any code that makes any assumptions about any format that is not documented.

The first record in any JADE journal file is the header record. This record is used to maintain the integrity of the sequence of journal files and to identify this journal file within the sequence. The header record contains:

- Audit structure version
- File number of this file
- Creation and update timestamps of this file
- File number of the previous file
- Creation timestamp of the previous file
- File number of the next file
- Creation timestamp of the next file
- Universal time bias from audit time to local time
- Block size when the file was written

- Logical end-of-file
- Creation timestamp of the most-recent description file

Following the header record, other records are written as required by events that occur within the JADE system. These records include:

Record	Contains…
Update	Allocate, create, delete, undelete, and update of objects, collections, and files
Control	Database-open, database-close, begin-transaction, commit-transaction, abort-transaction, and audit-switch
Discontinuity	No-audit and reorganization
Compensation	Inverse operations written during recovery, undo, and transaction abort
Reorganization	Start reorganization, add maps, reorganization files, install files, and end reorganization
Event	User sign-on, user sign-off, and cause events
SDS	Change primary and custom records for upgrade
Transaction information	Intermediate transaction table entry snapshot
Custom	Function call and description file creation

The format, structure, and content of these records is designed to suit the primary functions of the journal:

- Low cost to create the journal
- Suitability to support restart recovery
- Suitability to support roll-forward recovery
- Suitability to support Synchronized Database Service (SDS)

Each record begins with standardized control information to assist with navigating around the journal. This heading control information includes:

- Audit record type
- Record length
- Record timestamp and UTC bias
- Journal serial number (or audit serial number)
- Audit flags
- Log Sequence Number (LSN) of this record
- LSN of the next record
- LSN of the prior record of this transaction
- LSN of the next record to be processed during rollback
- Transaction-id

Tip If you want to explore the journal in a little more detail, have a look at a journal dump file. You can create this file from the **Dump Journal** menu item in the JADE Database utility. The dump file is written in the same directory as the journal file you select, and it has a file name suffix of **.dmp**.

Records are written to the journal in the sequence that the associated events occur within the entire JADE system. (Changes made to persistent objects are cached in the database buffers and written to the journal in a sequence determined by the database manager; not in the order that changes were made.)

The JADE system generally has multiple process threads running simultaneously, each doing some activity that contributes audit records to the journal. As a result, the audit records of each thread are written sequentially to the journal, but they are interspersed with the records originating from other threads. Whilst this simplifies writing the journal, it does mean that it is more difficult to read the journal when you want to follow the records of a single activity or thread.

Note All of those details make the JADE journal look very complicated, and in reality, it is. The good news, however, is that you do not need to know all of those details to be able to access some useful information.

Glossary

The terms associated with the JADE audit access are listed in the following table.

Term	Definition
After-image	Contents of an object after a change.
Archive journal set	Set of journal files that are not currently in use. Also known as *offline* or *archived* journal set.
Before-image	Contents of an object before a change.
Blob	Binary long object property. When the maximum length of a Binary property is not specified, the property value is separated from the parent object in the database, to simplify space allocation.
Current journal set	Set of journal files that are currently in use. Also known as *online* or *active* journal set.
Delta-image	Differences between the before-images and after-images.
Journal (1)	The complete set of audit records that capture details of changes to the database or to the database state. Also known as *audit trail*, *audit log*, *recovery log*, *transaction journal*, and *transaction log*.
Journal (2)	A file comprising part of the journal.
Journal file number	Number, sequentially incrementing by 1 for each journal file.
Journal serial number	Number, sequentially incrementing by 1 for each audit record within the entire journal. Also known as *audit serial number* (ASN).
Log Sequence Number (LSN)	Actual location of a journal record, comprising the file number and the byte offset within the file.
OID - Blob/Slob	Unique identifier of Blob or Slob, comprising: *0.parent-instance-number.parent-class-number.subclass-level.property-ordinal-number:edition*; for example, **0.6985.2557.2.1:3**.

Term	Definition
OID - Object	Unique object identifier, comprising: *class-number*.*instance-number*:*edition*; for example, **2557.6985:3**.
Slob	String long object property. When the maximum length of a String property is not specified, the property value is separated from the parent object in the database, to simplify space allocation.
Transaction-id	Number, sequentially incrementing by 1 for each **beginTransaction**.