Sun Microsystems Inc.

Java Transaction API (JTA)

This is the Java Transaction API (JTA) specification. JTA specifies high-level interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the resource manager, and the application server. This document also provides general usage scenarios and implementation considerations to support JTA in a component-based enterprise application server environment.

Please send technical comments on this specification to:

jta@eng.sun.com

Susan Cheung & Vlada Matena

Version 1.0

February 25, 1999

Table of Contents

1 Introduction

This document describes the Java Transaction API (JTA). JTA specifies local Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the resource manager, and the application server.

The JTA package consists of three parts:

- A high-level application interface that allows a transactional application to demarcate transaction boundaries.
- A Java mapping of the industry standard X/Open XA protocol that allows a transactional resource manager to participate in a global transaction controlled by an external transaction manager.
- A high-level transaction manager interface that allows an application server to control transaction boundary demarcation for an application being managed by the application server.

  Note: The JTA interfaces are presented as high-level from the transaction manager's perspective. In contrast, a low-level API for the transaction manager consists of interfaces that are used to implement the transaction manager. For example, the Java mapping of the OTS are low-level interfaces used internally by a transaction manager.

1.1 Background

Distributed transaction services in Enterprise Java middleware involves five players: the transaction manager, the application server, the resource manager, the application program, and the communication resource manager. Each of these players contributes to the distributed transaction processing system by implementing different sets of transaction APIs and functionalities.

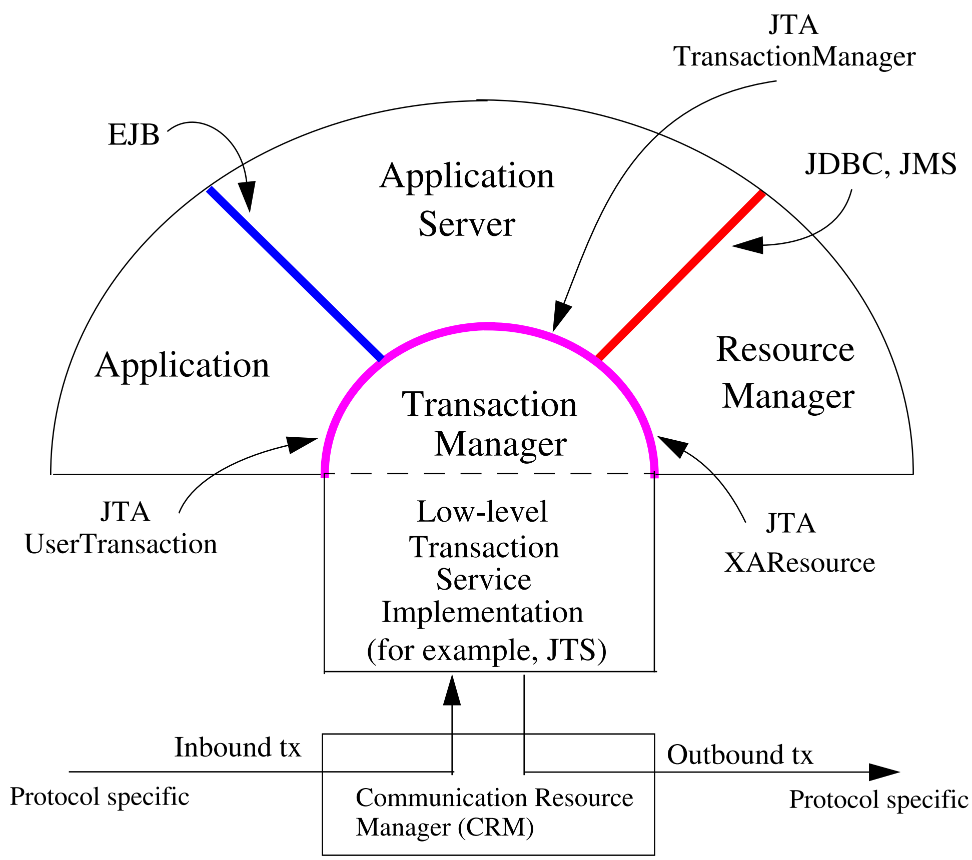
- A transaction manager provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.
- An application server (or TP monitor) provides the infrastructure required to support the application run-time environment which includes transaction state management. An example of such an application server is an EJB server.
- A resource manager (through a resource adapter[1]) provides the application access to resources. The resource manager participates in distributed transactions by implementing a transaction resource interface used by the

1.A Resource Adapter is a system level software library that is used by an application server or client to connect to a Resource Manager. A Resource Adapter is typically specific to a Resource Manager. It is available as a library and is used within the address space of the client using it. Examples of resource adapters are: JDBC driver to connect to a relational database, ODMG driver to connect to an object database, JRFC library to connect to SAP R/3 system. A resource adapter may provide additional services besides the connection API.

transaction manager to communicate transaction association, transaction completion and recovery work. An example of such a resource manager is a relational database server.

- A component-based transactional application that is developed to operate in a modern application server environment relies on the application server to provide transaction management support through declarative transaction attribute settings. An example of this type of applications is an application developed using the industry standard Enterprise JavaBeans (EJB) component architecture. In addition, some other stand-alone Java client programs may wish to control their transaction boundaries using a high-level interface provided by the application server or the transaction manager.
- A communication resource manager (CRM) supports transaction context propagation and access to the transaction service for incoming and outgoing requests. The JTA document does not specify requirements pertained to communication. Refer to the JTS Specification [2] for more details on interoperability between Transaction Managers.

From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed; only high-level interfaces need to be defined to allow transaction demarcation, resource enlistment, synchronization and recovery process to be driven from the users of the transaction services. The purpose of JTA is to define the local Java interfaces required for the transaction manager to support transaction management in the Java enterprise distributed computing environment. In the diagram shown below, the small half-circle represents the JTA specification. Chapter 3 of the document describes each portion of the specification in details.



1.2 Target Audience

This document is intended for implementors of:

- Transaction managers such as JTS.
- Resource adapters such as JDBC drivers and JMS providers.
- Transactional resource managers such as RDBMS.
- Application servers such as EJB Servers.
- Advanced transactional applications written in the Java™ programming language.

2 Relationship to Other Java APIs

2.1 Enterprise JavaBeans

The Enterprise JavaBeans architecture requires that an EJB Container support application-level transaction demarcation by implementing the `javax.transaction.UserTransaction` interface. The `UserTransaction` interface is intended to be used by both the EJB Bean implementor (for TX_BEAN_MANAGED Beans) and by the client programmer who wants to explicitly demarcate transaction boundaries within programs that are written in the Java programming language.

Refer to *http://java.sun.com/products/ejb* for further details on EJB.

Note: The EJB Spec and related Java files will be updated to reflect the current JTA package naming. This work is planned for the next EJB spec release.

2.2 JDBC 2.0 Standard Extension API

One of the new features included in the JDBC 2.0 Extension Specification is support for distributed transactions. Two new JDBC interfaces have been created for JDBC drivers to support distributed transactions using the Java Transaction API's `XAResource` interface. The new JDBC 2.0 interfaces are `javax.sql.XAConnection` and `javax.sql.XADataSource`.

A JDBC driver that supports distributed transactions implements the `javax.transaction.xa.XAResource` interface, the `javax.sql.XAConnection` interface, and the `javax.sql.XADataSource` interface. Refer to the JDBC 2.0 Standard Extension Specification for further details (*http://java.sun.com/products/jdbc)*.

2.3 Java Message Service

The Java Transaction API may be used by a Java Message Service provider to support distributed transactions. A JMS provider that supports the XAResource interface is able to participate as a resource manager in a distributed transaction processing system that uses a two-phase commit transaction protocol. In particular, a JMS provider implements the `javax.transaction.xa.XAResource` interface, the `javax.jms.XAConnection` and the `javax.jms.XASession` interface. Refer to the JMS 1.0 Specification for further details (*http://java.sun.com/products/jms)*.

2.4 Java Transaction Service

Java Transaction Service (JTS) is a specification for building a transaction manager which supports the JTA interfaces at the high-level and the standard Java mapping of the CORBA Object Transaction Service 1.1 specification at the low-level. JTS provides transaction interoperability using the CORBA standard IIOP protocol for transaction propagation between servers. JTS is intended for vendors who provide the transaction system infrastructure for enterprise middleware.

3 Java Transaction API

The Java Transaction API consists of three elements: a high-level application transaction demarcation interface, a high-level transaction manager interface intended for an application server, and a standard Java mapping of the X/Open XA protocol intended for a transactional resource manager. This chapter specifies each of these elements in details.

3.1 UserTransaction Interface

The `javax.transaction.UserTransaction` interface provides the application the ability to control transaction boundaries programmatically. This interface may be used by Java client programs or EJB beans.

The `UserTransaction.begin` method starts a global transaction and associates the transaction with the calling thread. The transaction-to-thread association is managed transparently by the Transaction Manager.

Support for nested tranactions is not required. The `UserTransaction.begin` method throws the `NotSupportedException` when the calling thread is already associated with a transaction and the transaction manager implementation does not support nested transactions.

Transaction context propagation between application programs is provided by the underlying transaction manager implementations on the client and server machines. The transaction context format used for propagation is protocol dependent and must be negotiated between the client and server hosts. For example, if the transaction manager is an implementation of the JTS specification, it will use the transaction context propagation format as specified in the CORBA OTS 1.1 specification. Transaction propagation is transparent to application programs.

3.1.1 UserTransaction Support in EJB Server

EJB servers are required to support the `UserTransaction` interface for use by EJB beans with the TX_BEAN_MANAGED transaction attribute. The `UserTransaction` interface is exposed to EJB components through the `EJBContext` interface using the `getUserTransaction` method. Thus, an EJB application does not interface with the Transaction Manager directly for transaction demarcation; instead, the EJB bean relies on the EJB Server to provide support for all of its transaction work as defined in the Enterprise JavaBeans Specification [5]. *(The underlying interaction between the EJB Server and the TM is transparent to the application.)*

The code sample below illustrates the usage of `UserTransaction` by a TX_BEAN_MANAGED EJB session bean:

```
// In the session bean’s setSessionContext method,
// store the bean context in an instance variable
SessionContext ctx = sessionContext;

..
```

```
// somewhere else in the bean’s business logic
UserTransaction utx = ctx.getUserTransaction();

// start a transaction
utx.begin();

.. do work

// commit the work
utx.commit();
```

3.1.2 UserTransaction Support for Transactional Clients

The `UserTransaction` interface may be used by Java client programs either through support from the application server or support from the transaction manager on the client host.

The application server vendor is expected to provide tools for an administrator to configure the `UserTransaction` object binding in the JNDI namespace. The implementation of the `UserTransaction` object must be both `javax.naming.Referenceable` and `java.io.Serializable`, so that the object can be stored in all JNDI naming contexts.

If an application server supports transaction demarcation performed by transactional clients, the application server must support the client program’s ability to use the JNDI lookup mechanism for obtaining the `UserTransaction` object reference. As JTA does not define the JNDI name for `UserTransaction`, the client program should use an appropriate configuration mechanism to pass the name string to the JNDI `lookup` method.

An example of such an implementation is through the use of a system property. The following sample code is provided for illustrative purposes:

```
// get the system property value configured by administrator
String utxPropVal = System.getProperty(“jta.UserTransaction”);

// use JNDI to locate the UserTransaction object
Context ctx = new InitialContext();
UserTransaction utx = (UserTransaction)ctx.lookup(utxPropVal);

// start transaction work..
utx.begin();
.. do work
utx.commit();
```

3.2 TransactionManager Interface

The `javax.transaction.TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed. For example, the EJB container manages the transaction states for transactional EJB components; the container uses the `TransactionManager` interface mainly to demarcate transaction boundaries where operations affect the calling thread's transaction context. The Transaction Manager maintains the transaction context association with threads as part of its internal data structure. A thread's transaction context is either *null* or it refers to a specific global transaction. Multiple threads may concurrently be associated with the same global transaction.

Support for nested tranactions is not required.

Each transaction context is encapsulated by a `Transaction` object, which can be used to perform operations which are specific to the target transaction, regardless of the calling thread's transaction context. The following sections provide more details.

3.2.1 Starting a Transaction

The `TransactionManager.begin` method starts a global transaction and associates the transaction context with the calling thread.

If the Transaction Manager implementation does not support nested transactions, the `TransactionManager.begin` method throws the `NotSupportedException` when the calling thread is already associated with a transaction.

The `TransactionManager.getTransaction` method returns the `Transaction` object that represents the transaction context currently associated with the calling thread. This `Transaction` object can be used to perform various operations on the target transaction. Examples of `Transaction` object operations are resource enlistment and synchronization registration. The `Transaction` interface is described in section 3.3 below.

3.2.2 Completing a Transaction

The `TransactionManager.commit` method completes the transaction currently associated with the calling thread. After the `commit` method returns, the calling thread is not associated with a transaction. If the `commit` method is called when the thread is not associated with any transaction context, the TM throws an exception. In some implementations, the commit operation is restricted to the transaction originator only. If the calling thread is not allowed to commit the transaction, the TM throws an exception.

The `TransactionManager.rollback` method rolls back the transaction associated with the current thread. After the `rollback` method completes, the thread is associated with no transaction.

3.2.3 Suspending and Resuming a Transaction

A call to the `TransactionManager.suspend` method temporarily suspends the transaction that is currently associated with the calling thread. If the thread is not associated with any transaction, a *null* object reference is returned; otherwise, a valid `Transaction` object is returned. The `Transaction` object can later be passed to the `resume` method to reinstate the transaction context association with the calling thread.

The `TransactionManager.resume` method re-associates the specified transaction context with the calling thread. If the transaction specified is a valid transaction, the transaction context is associated with the calling thread; otherwise, the thread is associated with no transaction.

```
Transaction tobj = TransactionManager.suspend();

..
TransactionManager.resume(tobj);
```

If `TransactionManager.resume` is invoked when the calling thread is already associated with another transaction, the Transaction Manager throws the `IllegalStateException` exception.

Note that some transaction manager implementations allow a suspended transaction to be resumed by a different thread. This feature is not required by JTA.

The application server is responsible for ensuring that the resources in use by the application are properly delisted from the suspended transaction. A resource delist operation triggers the Transaction Manager to inform the resource manager to disassociate the transaction from the specified resource object (`XAResource.end(TMSUSPEND)`).

When the application's transaction context is resumed, the application server ensures that the resource in use by the application is again enlisted with the transaction. Enlisting a resource as a result of resuming a transaction triggers the Transaction Manager to inform the resource manager to re-associate the resource object with the resumed transaction (`XAResource.start(TMRESUME)`). Refer to Sections 3.3.1 and 3.4.4 for more details on resource enlistment and transaction association.

In the EJB environment, the EJB server typically manages the transactional resources in use by the applications (The EJB bean's resource requests are tracked and maintained in the bean's instance context). When suspending a transaction currently associated with an EJB instance, the application server examines the list of resources in use by the bean instance to determine whether any resources need to be delisted. For each resource that is currently enlisted with the suspended transaction, the application server calls the `Transaction.delistResource` method to disassociate the resource from the transaction. When the transaction is resumed for the EJB instance, the application server examines the list of resources in use and enlists the resources with the transaction manager before giving control to the bean's business method. Refer to Chapter 4 for further discussion on JTA support in an application server.

3.3 Transaction Interface

The `Transaction` interface allows operations to be performed on the transaction associated with the target object. Every global transaction is associated with one `Transaction` object when the transaction is created. The `Transaction` object can be used to:

- Enlist the transactional resources in use by the application.
- Register for transaction synchronization callbacks.
- Commit or rollback the transaction.

- Obtain the status of the transaction.

These functions are described in the sections below.

3.3.1 Resource Enlistment

An application server provides the application run-time infrastructure that includes transactional resource management. Transactional resources such as database connections are typically managed by the application server in conjunction with some resource adapter and optionally with connection pooling optimization. In order for an external transaction manager to coordinate transactional work performed by the resource managers, the application server must enlist and delist the resources used in the transaction.

Resource enlistment performed by an application server serves two purposes:

- It informs the Transaction Manager about the resouce manager instance that is participating in the global transaction. This allows the Transaction Manager to inform the participating resource manager on transaction association with the work performed through the connection (resource) object.
- It enables the Transaction Manager to group the resource types in use by each transaction. The resource grouping allows the Transaction Manager to conduct the two-phase commit transaction protocol between the TM and the RMs, as defined by the X/Open XA specification.

For each resource in use by the application, the application server invokes the `enlistResource` method and specifies the `XAResource` object that identifies the resource in use.

The `enlistResource` request results in the Transaction Manager informing the resource manager to start associating the transaction with the work performed through the corresponding resource—by invoking the `XAResource.start` method. The Transaction Manager is responsible for passing the appropriate flag in its `XAResource.start` method call to the resource manager. The `XAResource` interface is described in section 3.4.

If the target transaction already has another `XAResource` object participating in the transaction, the Transaction Manager invokes the `XAResource.isSameRM` method to determine if the specified `XAResource` represents the same resource manager instance. This information allows the TM to group the resource managers who are performing work on behalf of the transaction.

- If the `XAResource` object represents a resource manager instance who has seen the global transaction before, the TM groups the newly registered resource together with the previous `XAResource` object and ensures that the same RM only receives one set of prepare-commit calls for completing the target global transaction.

If the `XAResource` object represents a resource manager who has not previously seen the global transaction, the TM establishes a different transaction branch ID[1] and ensures tha this new resource manager is informed about the transaction completion with proper prepare-commit calls.

The `isSameRM` method is discussed in section 3.4.9.

The `Transaction.delistResource` method is used to disassociate the specified resource from the transaction context in the target object. The application server invokes the `delistResource` method with the following two parameters:

- The `XAResource` object that represents the resource.
- A `flag` to indicate whether the delistment was due to:
  - The transaction being suspended (`TMSUSPEND`).
  - A portion of the work has failed (`TMFAIL`).
  - A normal resource release by the application (`TMSUCCESS`).

An example of `TMFAIL` could be the situation where an application receives an exception on its connection operation.

The delist request results in the transaction manager informing the resource manager to end the association of the transaction with the target `XAResource`. The flag value allows the application server to indicate whether it intends to come back to the same resource. The transaction manager passes the appropriate flag value in its `XAResource.end` method call to the underlying resource manager.

3.3.2 Transaction Synchronization

Transaction synchronization allows the application server to get notification from the transaction manager before and after the transaction completes. For each transaction started, the application server may optionally register a `javax.transaction.Synchronization` callback object to be invoked by the transaction manager:

- The `Synchronization.beforeCompletion` method is called prior to the start of the two-phase transaction complete process. This call is executed in the same transaction context of the caller who initiates the `TransactionManager.commit`, or the call is executed with no transaction context if `Transaction.commit` is used.
- The `Synchronization.afterCompletion` method is called after the transaction has completed. The status of the transaction is supplied in the parameter. This method is executed without a transaction context.

1.Transaction Branch is defined in the X/Open XA spec [1] as follows: “A global transaction has one or more transaction branches. A branch is a part of the work in support of a global transaction for which the TM and the RM engage in a separate but coordinated transaction commitment protocol. Each of the RM’s internal units of work in support of a global transaction is part of exactly one branch. .. After the TM begins the transaction commitment protocol, the RM receives no additional work to do on that transaction branch. The RM may receive additional work on behalf of the same transaction, from different branches. The different branches are related in that they must be completed atomically. Each transaction branch identifier (or XID) that the TM gives the RM identifies both a global transaction and a specific branch. The RM may use this information to optimise its use of shared resources and locks.”

3.3.3 Transaction Completion

The `Transaction.commit` and `Transaction.rollback` methods allow the target object to be comitted or rolled back. The calling thread is not required to have the same transaction associated with the thread.

If the calling thread is not allowed to commit the transaction, the transaction manager throws an exception.

3.3.4 Transaction Equality and Hash Code

The transaction manager must implement the `Transaction` object's `equals` method to allow comparison between the target object and another `Transaction` object. The `equals` method should return *true* if the target object and the parameter object both refer to the same global transaction.

For example, the application server may need to compare two `Transaction` objects when trying to reuse a resource that is already enlisted with a transaction. This can be done using the `equals` method.

```
Transaction txObj = TransactionManager.getTransaction();
Transaction someOtherTxObj = ..

..
boolean isSame = txObj.equals(someOtherTxObj);
```

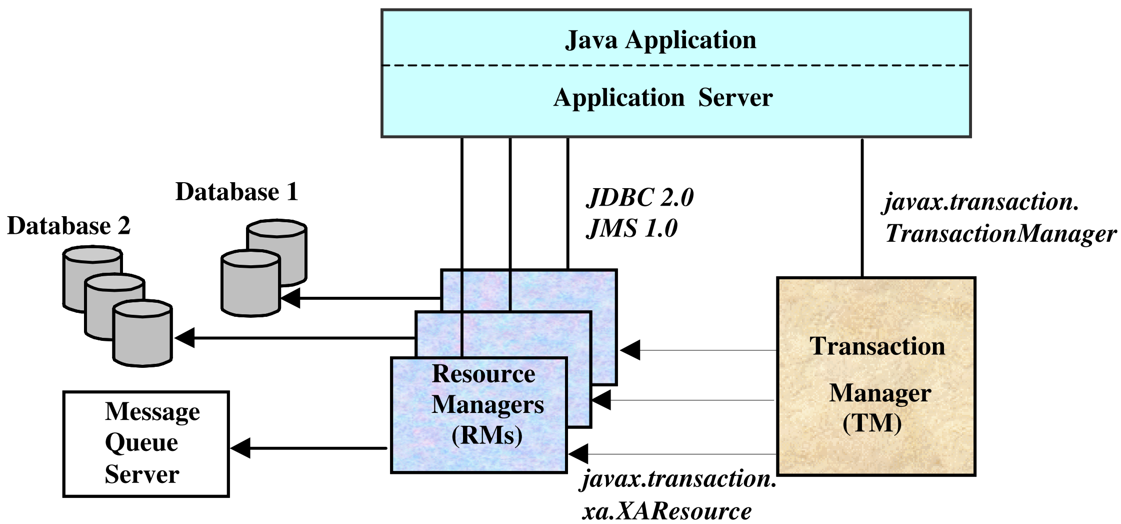
In addition, the transaction manager must implement the `Transaction` object's `hashCode` method so that if two `Transaction` objects are equal, they have the same hash code. However, the converse is not necessarily true. Two `Transaction` objects with the same hash code are not necessarily equal.

3.4 XAResource Interface

The `javax.transaction.xa.XAResource` interface is a Java mapping of the industry standard XA interface based on the X/Open CAE Specification (Distributed Transaction Processing: The XA Specification).

The `XAResource` interface defines the contract between a Resource Manager and a Transaction Manager in a distributed transaction processing (DTP) environment. A resource adapter for a resource manager implements the `XAResource` interface to support association of a global transaction to a transaction resource, such as a connection to a relational database.

A global transaction is a unit of work that is performed by one or more resource managers (RM) in a DTP system. Such a system relies on an external transaction manager, such as Java Transaction Service (JTS), to coordinate transactions.



The `XAResource` interface can be supported by any transactional resource adapter that is intended to be used by application programs in an environment where transactions are controlled by an external transaction manager. An example of such a resource is a database management system. An application may access data through multiple database connections. Each database connection is associated with an `XAResource` object that serves as a proxy object to the underlying resource manager instance. The transaction manager obtains an `XAResource` for each resource manager participating in a global transaction. It uses the `start` method to associate the global transaction with the resource, and it uses the `end` method to disassociate the transaction from the resource. The resource manager is responsible for associating the global transaction with all work performed on its data between the `start` and `end` method invocations.

At transaction commit time, these transactional resource managers are informed by the transaction manager to prepare, commit, or rollback the transaction according to the two-phase commit protocol.

The `XAResource` interface, in order to be better integrated with the Java environment, differs from the standard X/Open XA interface in the following ways:

- The resource manager initialization is done implicitly by the resource adapter when the resource (connection) is acquired. There is no *xa_open* equivalent in the `XAResource` interface. This obviates the need for a resource manager to provide a different syntax to open a resource for use within the distributed transaction environment from the syntax used in the environment without distributed transactions.
- *Rmid* is not passed as an argument. We use an object-oriented approach where each *Rmid* is represented by a separate `XAResource` object.
- Asynchronous operations are not supported. Java supports multi-threaded processing and most databases do not support asynchronous operations.

- Error return values that are caused by the transaction manager's improper handling of the `XAResource` object are mapped to Java exceptions via the `XAException` class.
- The DTP concept of "Thread of Control" maps to all Java threads that are given access to the `XAResource` and `Connection` objects. For example, it is legal (although in practice rarely used) for two different Java threads to perform the `start` and `end` operations on the same `XAResource` object.
- Association migration and dynamic registration (optional X/Open XA features) are not supported. We've omitted these features for a simpler `XAResource` interface and simpler resource adapter implementation.

3.4.1 Opening a Resource Manager

The X/Open XA interface specifies that the transaction manager must initialize a resource manager (*xa_open*) prior to any other *xa_* calls. We believe that the knowledge of initializing a resource manager should be embedded within the resource adapter that represents the resource manager. The transaction manager does not need to know how to initialize a resource manager. The TM is only responsible for informing the resource manager about when to start and end work associated with a global transaction and when to complete the transaction.

The resource adapter is responsible for opening (initializing) the resource manager when the connection to the resource manager is established.

3.4.2 Closing a Resource Manager

A resource manager is closed by the resource adapter as a result of destroying the transactional resource. A transaction resource at the resource adapter level is comprised of two separate objects:

- An `XAResource` object that allows the transaction manager to start and end the transaction association with the resource in use and to coordinate transaction completion process.
- A connection object that allows the application to perform operations on the underlying resource (for example, JDBC operations on an RDBMS).

The resource manager, once opened, is kept open until the resource is released (closed) explicitly. When the application invokes the connection's `close` method, the resource adapter invalidates the connection object reference that was held by the application and notifies the application server about the close. The transaction manager should invoke the `XAResource.end` method to disassociate the transaction from that connection.

The `close` notification allows the application server to perform any necessary cleanup work and to mark the physical XA connection as free for reuse, if connection pooling is in place.

3.4.3 Thread of Control

The X/Open XA interface specifies that the transaction association related `xa` calls must be invoked from the same thread context. This thread-of-control requirement is not applicable to the object-oriented component-based application run-time environment, in which application threads are dispatched dynamically at method invocation time. Different Java threads may be using the same connection resource to access the resource manager if the connection spans multiple method invocation. Depending on the implementation of the application server, different Java threads may be involved with the same `XAResource` object. The resource context and the transaction context may be operated independent of thread context. This means, for example, that it's possible for different threads to be invoking the `XAResource.start` and `XAResource.end` methods.

If the application server allows multiple threads to use a single `XAResource` object and the associated connection to the resource manager, it is the responsibility of the application server to ensure that there is only one transaction context associated with the resource at any point of time.

Thus the `XAResource` interface specified in this document requires that the resource managers be able to support the two-phase commit protocol from any thread context.

3.4.4 Transaction Association

Global transactions are associated with a transactional resource via the `XAResource.start` method, and disassociated from the resource via the `XAResource.end` method. The resource adapter is responsible for internally maintaining an association between the resource connection object and the `XAResource` object. At any given time, a connection is associated with a single transaction or it is not associated with any transaction at all.

Interleaving multiple transaction contexts using the same resource may be done by the transaction manager as long as `XAResource.start` and `XAResource.end` are invoked properly for each transaction context switch. Each time the resource is used with a different transaction, the method `XAResource.end` must be invoked for the previous transaction that was associated with the resource, and `XAResource.start` must be invoked for the current transaction context.

`XAResource` does not support nested transactions. It is an error for the `XAResource.start` method to be invoked on a connection that is currently associated with a different transaction.

Table 1: Transaction Association

XAResource Methods	XAResource Transaction States		
	Not Associated T_0	Associated T_1	Association Suspended T_2
start()	T_1		
start (TMRESUME)			T_1

Table 1: Transaction Association

XAResource Methods	XAResource Transaction States		
	Not Associated T_0	Associated T_1	Association Suspended T_2
start (TMJOIN)			T_1
end (TMSUSPEND)		T_2	
end (TMFAIL)		T_0	T_0
end (TMSUCCESS)		T_0	T_0
recover()	T_0	T_1	T_2

3.4.5 Externally Controlled Connections

Resources for transactional applications, whose transaction states are managed by an application server, must also be managed by the application server so that transaction association is performed properly. If an application is associated with a global transaction, it is an error for the application to perform transactional work through the connection without having the connection's resource object already associated with the global transaction. The application server must ensure that the `XAResource` object in use is associated with the transaction. This is done by invoking the `Transaction.enlistResource` method.

If a server side transactional application retains its database connection across multiple client requests, the application server must ensure, before dispatching a client request to the application thread, that the resource is enlisted with the application's current transaction context. This implies that the application server manages the connection resource usage status across multiple method invocations.

3.4.6 Resource Sharing

When the same transactional resource is used to interleave multiple transactions, it is the responsibility of the application server to ensure that only one transaction is enlisted with the resource at any given time. To initiate the transaction commit process, the transaction manager is allowed to use any of the resource objects connected to the same resource manager instance. The resource object used for the two-phase commit protocol need not have been involved with the transaction being completed.

The resource adapter must be able to handle multiple threads invoking the `XAResource` methods concurrently for transaction commit processing. For example, suppose we have a transactional resource `r1`. Global transaction `xid1` was *started* and *ended* with `r1`. Then a different global transaction `xid2` is associated with `r1`. Meanwhile, the transaction manager may start the two phase commit process for `xid1` using `r1` or any other transactional resource connected to the same resource manager. The resource adapter needs to allow the commit process to be executed while the resource is currently associated with a different global transaction.

The sample code below illustrates the above scenario:

```
// Suppose we have some transactional connection-based
// resource r1 that is connected to an enterprise information
// service system.
//
XAResource xares = r1.getXAResource();

xares.start(xid1); // associate xid1 to the connection
..
xares.end(xid1); // dissociate xid1 to the connection
..

xares.start(xid2); // associate xid2 to the connection
..

// While the connection is associated with xid2,
// the TM starts the commit process for xid1

status = xares.prepare(xid1);
..
xares.commit(xid1, false);
```

3.4.7 Local and Global Transactions

The resource adapter is encouraged to support the usage of both local and global transactions within the same transactional connection. Local transactions are transactions that are started and coordinated by the resource manager internally. The `XAResource` interface is not used for local transactions.

When using the same connection to perform both local and global transactions, the following rules apply:

- The local transaction must be committed (or rolled back) before starting a global transaction in the connection.
- The global transaction must be disassociated from the connection before any local transaction is started.

If a resource adapter does not support mixing local and global transactions within the same connection, the resource adapter should throw the resource specific exception. For example, `java.sql.SQLException` is thrown to the application if the resource manager for the underlying RDBMS does not support mixing local and global transactions within the same JDBC connection.

3.4.8 Failures Recovery

During recovery, the Transaction Manager must be able to communicate to all resource managers that are in use by the applications in the system. For each resource manager, the Transaction Manager uses the `XAResource.recover` method to retrieve the list of transactions that are currently in a prepared or heuristically completed state.

Typically, the system administrator configures all transactional resource factories that are used by the applications deployed on the system. An example of such a resource factory is the JDBC `XADataSource` object, which is a factory for the JDBC `XAConnection` objects. The implementation of these transactional resource factory objects are both `javax.naming.Referenceable` and `java.io.Serializable` so that they can be stored in all JNDI naming contexts.

Because `XAResource` objects are not persistent across system failures, the Transaction Manager needs to have some way to acquire the `XAResource` objects that represent the resource managers which might have participated in the transactions prior to the system failure. For example, a Transaction Manager might, through the use of the JNDI lookup mechanism and cooperation from the application server, acquire an `XAResource` object representing each of the Resource Manager configured in the system. The Transaction Manager then invokes the `XAResource.recover` method to ask each resource manager to return any transactions that are currently in a prepared or heuristically completed state. It is the responsibility of the Transaction Manager to ignore transactions that do not belong to it.

3.4.9 Identifying Resource Manager Instance

The `isSameRM` method is invoked by the Transaction Manager to determine if the target `XAResource` object represents the same resource manager instance as that represented by the `XAResource` object in the parameter. The `isSameRM` method returns *true* if the specified target object is connected to the same resource manager instance; otherwise, the method returns *false*. The semi-pseudo code below illustrates the intended usage.

```
public boolean enlistResource(XAResource xares)
{   ..

    // Assuming xid1 is the target transaction and
    // xid1 already has another resource object xaRes1
    // participating in the transaction

    boolean sameRM = xares.isSameRM(xaRes1);
    if (sameRM) {
         //
         // Same underlying resource manager instance,
         // group together with xaRes1 and join the transaction
         //
         xares.start(xid1, TMJOIN);
    } else {
         //
         // This is a different RM instance,
         // make a new transaction branch for xid1
         //
         xid1NewBranch = makeNewBranch(xid1);
         xares.start(xid1NewBranch, TMNOFLAGS);
    }
    ..
```

```
}
```

3.4.10 Dynamic Registration

Dynamic registration is not supported in `XAResource` because of the following reasons:

- In the Java component-based application server environment, connections to the resource manager are acquired dynamically when the application explicitly requests for a connection. These resources are enlisted with the transaction manager on an "as-needed" basis (unlike the static xa_switch table that exists in the C-XA procedural model).
- If a resource manager requires a way to dynamically register its work to the global transaction, then the implementation can be done at the resource adapter level via a private interface between the resource adapter and the underlying resource manager.

3.5 Xid Interface

The javax.transaction.xa.`Xid` interface is a Java mapping of the X/Open transaction identifier XID structure. This interface specifies three accessor methods which are used to retrieve a global transaction's format ID, a global transaction ID, and a branch qualifier. The `Xid` interface is used by the transaction manager and the resource managers. This interface is not visible to the application programs nor the application server.

4 JTA Support in the Application Server

This chapter provides a discussion on implementation and usage considerations for application servers to support the Java Transaction API. Our discussion assumes the application's transactions and resource usage are managed by the application server. We further assume that access to the underlying transactional resource manager is through some Java API implemented by the resource adapter representing the resource manager. For example, a JDBC 2.0 driver may be used to access a relational database, a SAP connector resource adapter may be used to access the SAP R/3 ERP system, and so on. This section focuses on the usage of JTA and assumes a generic connection based transactional resource is in use without being specific about a particular type of resource manager.

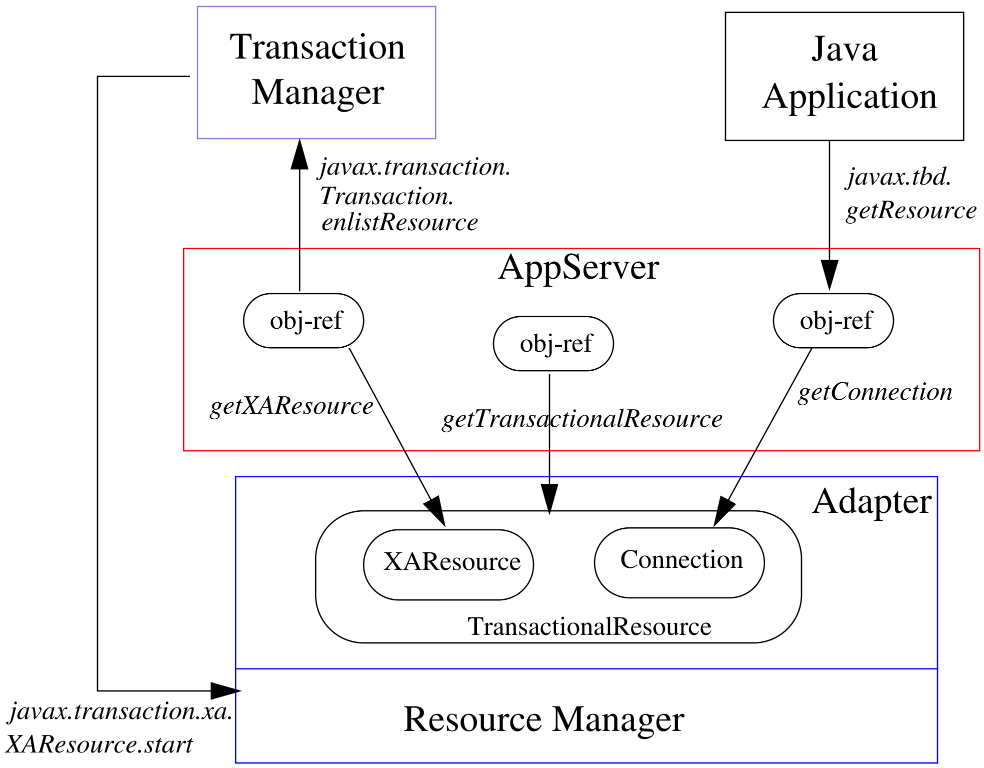
4.1 Connection-Based Resource Usage Scenario

Let's assume that the resource adapter provides a connection-based resource API called *TransactionalResource* to access the underlying resource manager.

In a typical usage scenario, the application server invokes the resource adapter's resource factory to create a *TransactionalResource* object. The resource adapter internally associates the *TransactionalResource* with two other entities: an object that implements the specific resource adapter's connection interface and an object that implements the `javax.transaction.xa.XAResource` interface.

The application server obtains a *TransactionalResource* object and uses it in the following way. The application server obtains the `XAResource` object via a `getXAResource` method. The application server enlists the `XAResource` to the Transaction Manager (TM) using the `Transaction.enlistResource` method. The TM informs the Resource Manager to associate the work performed (through that connection) with the transaction currently associated with the application. The TM does it by invoking the `XAResource.start` method.

The application server then invokes some `getConnection` method to obtain a `Connection` object and returns it to the application. Note that the Connection interface is implemented by the resource adapter and it is specific to the underlying resource supported by the resource manager. The diagram below illustrates a general flow of acquiring resource and enlisting the resource to the Transaction Manager.



In this usage scenario, the `XAResource` interface is transparent to the application program, and the `Connection` interface is transparent to the transaction manager. The application server is the only party that holds a reference to some *TransactionalResource* object.

The code sample below illustrate how the application server obtains the `XAResource` object reference and enlists it with the Transaction Manager.

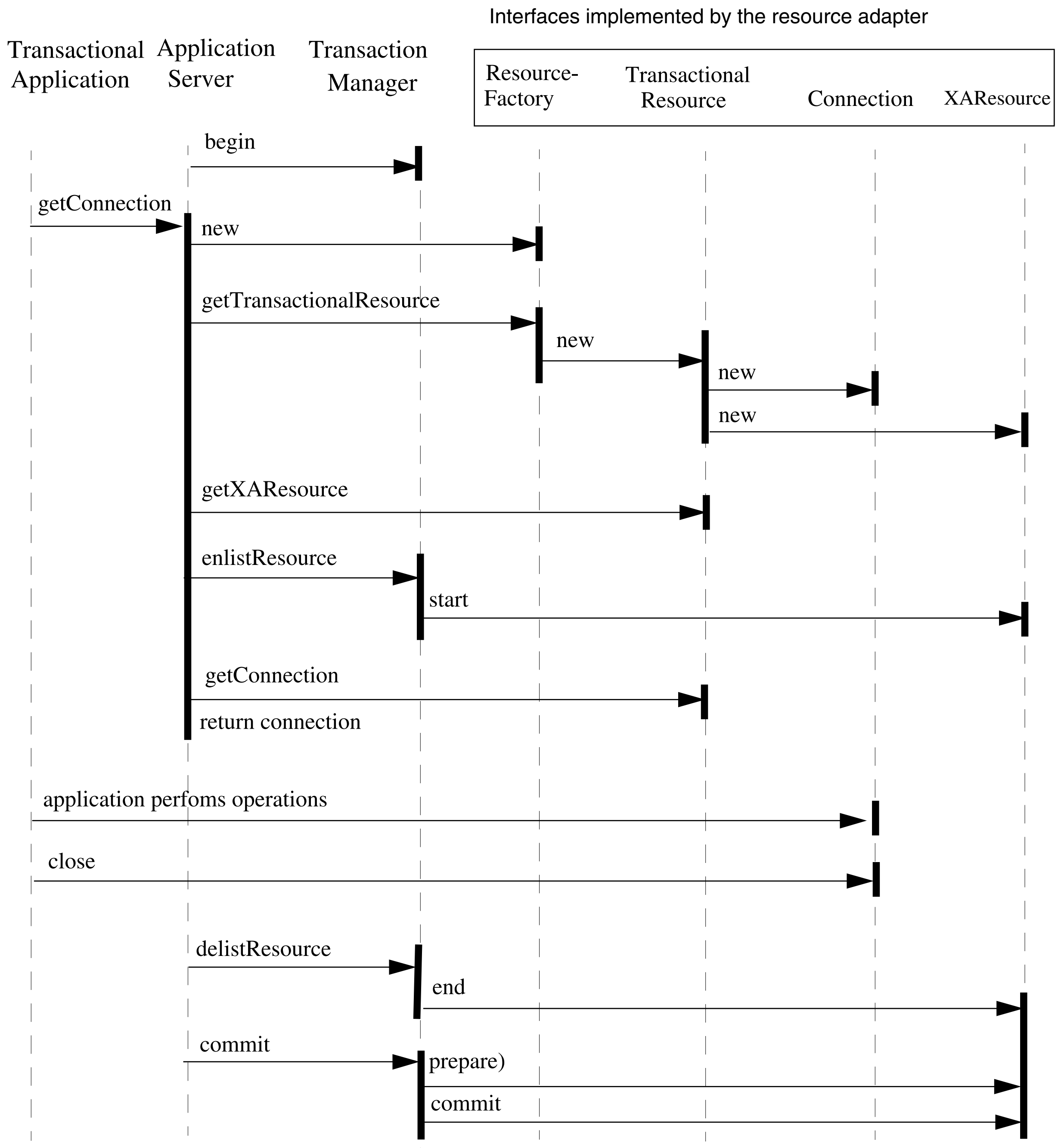
```
// Acquire some connection-based transactional resource to
// access the resource manager

Context ctx = InitialContext();
ResourceFactory rf =(ResourceFactory)ctx.lookup("MyEISResource");
TransactionalResource res = rf.getTransactionalResource();

// Obtain the XAResource part of the connection and
// enlist it with the Transaction Manager

XAResource xaRes = res.getXAResource();
(TransactionManager.getTransaction()).enlistResource(xaRes);

// get the connection part of the transaction resource
Connection con = (Connection)res.getConnection();

.. return the connection to the application
```

4.2 Transaction Association and Connection Request Flow

This session provides a brief walkthrough of how an application server may handle a connection request from the application. The figure that follows illustrates the usage of JTA. The steps shown are for illustrative purposes, they are not prescriptive:

1. Assuming a client invokes an EJB bean with a TX_REQUIRED transaction attribute and the client is not associated with a global transaction, the EJB container starts a global transaction by invoking the `TransactionManager.begin` method.
2. After the the transaction starts, the container invokes the bean method. As part of the business logic, the bean requests for a connection-based resource using the API provided by the resource adapter of interest.
3. The application server obtains a resource from the resource adapter via some *ResourceFactory.getTransactionalResource* method.
4. The resource adapter creates the *TransactionalResource* object and the associated `XAResource` and `Connection` objects.
5. The application server invokes the `getXAResource` method.
6. The application server enlists the resource to the transaction manager.
7. The transaction manager invokes `XAResource.start` to associate the current transaction to the resource.
8. The application server invokes the `getConnection` method.
9. The application server returns the `Connection` object reference to the application.
10. The application performs one or more operations on the connection.
11. The application closes the connection.
12. The application server delist the resource when notified by the resource adapter about the connection close.
13. The transaction manager invokes `XAResource.end` to disassociate the transaction from the `XAResource`.
14. The application server asks the transaction manager to commit the transaction.
15. The transaction manager invokes `XAResource.prepare` to inform the resource manager to prepare the transaction work for commit.
16. The transaction manager invokes `XAResource.commit` to commit the transaction.

This example illustrates the application server's usage of the `TransactionManager` and `XAResource` interfaces as part of the application connection request handling.


Interfaces implemented by the resource adapter
Transactional Application
Application Server
Transaction Manager
Resource-Factory
Transactional Resource
Connection
XAResource
begin
getConnection
new
getTransactionalResource
new
new
new
getXAResource
enlistResource
start
getConnection
return connection
application perfoms operations
close
delistResource
end
commit
prepare)
commit

5 Java Transaction API Reference

This chapter provides the documentation of the interfaces and classes that are part of the Java Transaction API standard extension. The `javax.transaction` package is relevant to the Enterprise JavaBeans, JDBC, JMS, and JTS standard extension APIs.

package ***javax.transaction***:

Interface:

```
public interface javax.transaction.Status
public interface javax.transaction.Synchronization
public interface javax.transaction.Transaction
public interface javax.transaction.TransactionManager
public interface javax.transaction.UserTransaction
```

Classes:

```
public class javax.transaction.HeuristicCommitException
public class javax.transaction.HeuristicMixedException
public class javax.transaction.HeuristicRollbackException
public class javax.transaction.InvalidTransactionException
public class javax.transaction.NotSupportedException
public class javax.transaction.RollbackException
public class javax.transaction.TransactionRequiredException
public class javax.transaction.TransactionRolledbackException
public class javax.transaction.SystemException
```

package ***javax.transaction.xa***:

Interfaces:

```
public interface javax.transaction.xa.XAResource
public interface javax.transaction.xa.Xid
```

Classes:

```
public class javax.transaction.xa.XAException
```

Interface Status

```
interface javax.transaction.Status
{
   public final static int STATUS_ACTIVE;
   public final static int STATUS_COMMITTED;
   public final static int STATUS_COMMITTING;
   public final static int STATUS_MARKED_ROLLBACK;
   public final static int STATUS_NO_TRANSACTION;
   public final static int STATUS_PREPARED;
   public final static int STATUS_PREPARING;
   public final static int STATUS_ROLLEDBACK;
   public final static int STATUS_ROLLING_BACK;
   public final static int STATUS_UNKNOWN;
}
```

Variables

- **STATUS_ACTIVE**

  `public final static int STATUS_ACTIVE`

  A transaction is associated with the target object and it is in the active state. An implementation returns this status after a transaction has been started and prior to a transaction coordinator issuing any prepares unless the transaction has been marked for rollback.

- **STATUS_COMMITTED**

  `public final static int STATUS_COMMITTED`

  A transaction is associated with the target object and it has been committed. It is likely that heuristics exists, otherwise the transaction would have been destroyed and NoTransaction returned.

- **STATUS_COMMITTING**

  `public final static int STATUS_COMMITTING`

  A transaction is associated with the target object and it is in the process of committing. An implementation returns this status if it has decided to commit, but has not yet completed the process, probably because it is waiting for responses from one or more Resources.

- **STATUS_MARKED_ROLLBACK**

  `public final static int STATUS_MARKED_ROLLBACK`

  A transaction is associated with the target object and it has been marked for rollback, perhaps as a result of a setRollbackOnly operation.

- **STATUS_NO_TRANSACTION**

  `public final static int STATUS_NO_TRANSACTION`

  No transaction is currently associated with the target object. This will occur after a transaction has completed.

- **STATUS_PREPARED**

  `public final static int STATUS_PREPARED`

A transaction is associated with the target object and it has been prepared, i.e. all subordinates have responded Vote.Commit. The target object may be waiting for a superior's instruction as how to proceed.

- **STATUS_PREPARING**

  `public final static int STATUS_PREPARING`

  A transaction is associated with the target object and it is in the process of preparing. An implementation returns this status if it has started preparing, but has not yet completed the process, probably because it is waiting for responses to prepare from one or more Resources.

- **STATUS_ROLLEDBACK**

  `public final static int STATUS_ROLLEDBACK`

  A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exist, otherwise the transaction would have been destroyed and NoTransaction returned.

- **STATUS_ROLLING_BACK**

  `public final static int STATUS_ROLLING_BACK`

  A transaction is associated with the target object and it is in the process of rolling back. An implementation returns this status if it has decided to rollback, but has not yet completed the process, probably because it is waiting for responses from one or more Resources.

- **STATUS_UNKNOWN**

  `public final static int STATUS_UNKNOWN`

  A transaction is associated with the target object but its current status cannot be determined. This is a transient condition and a subsequent invocation will ultimately return a different status.

Interface Synchronization

```
interface javax.transaction.Synchronization
{
   public abstract void beforeCompletion();
   public abstract void afterCompletion(int status);
}
```

The transaction manager provides a synchronization protocol that allows the interested party to be notified before and after the transaction completes. Using the registerSynchronization method, the application server registers a Synchronization object for the transaction currently associated with the target Transaction object.

Methods

- **beforeCompletion**

  `public abstract void beforeCompletion()`

  This `beforeCompletion` method is called by the transaction manager prior to the start of the transaction completion process. This call is executed in the same transaction context of the caller who initiates the TransactionManager.commit, or the call is executed with no transaction context if Transaction.commit is used.

- **afterCompletion**

  `public abstract void afterCompletion(int status)`

  The afterCompletion method is called by the transaction manager after the transaction is committed or rolled back. This method executes without a transaction context.

  Parameters:

  `status`

  Status of the transaction that was completed. The value provided is the same as that returned by getStatus.

Interface Transaction

```
interface javax.transaction.Transaction
{
   public abstract void commit();
   public abstract boolean delistResource(XAResource xaRes, int flag);
   public abstract boolean enlistResource(XAResource xaRes);
   public abstract int getStatus();
   public abstract void registerSynchronization(Synchronization sync);
   public abstract void rollback();
   public abstract void setRollbackOnly();
}
```

The `Transaction` interface allows operations to be performed against the transaction in the target Transactioin object. A Transaction object is created corresponding to each global transaction creation.The `Transaction` object can be used for resource enlistment, synchronization registration, transaction completion and status query operations.

Methods

- **commit**

  ```
  public abstract void commit() throws RollbackException,
              HeuristicMixedException, HeuristicRollbackException,
              SecurityException, SystemException
  ```

  Complete the transaction associated with the target Transaction object.

  Throws: RollbackException
  Thrown to indicate that the transaction has been rolled back rather than committed.

  Throws: HeuristicMixedException
  Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.

  Throws: HeuristicRollbackException
  Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.

  Throws: SecurityException
  Thrown to indicate that the thread is not allowed to commit the transaction.

  Throws: IllegalStateException
  Thrown if the transaction in the target object is inactive.

  Throws: SystemException
  Thrown if the transaction manager encounters an unexpected error condition.

- **delistResource**

  ```
  public abstract boolean delistResource(XAResource xaRes, int flag)
                 throws IllegalStateException, SystemException
  ```

  Disassociate the resource specified from the transaction associated with the target Transaction object.

  Parameters:
  `xaRes`

The `XAResource` object associated with the resource (connection).

`flag`

TMSUSPEND - the resource should be dissociated with the suspend mode, the caller intends to come back to the current state.
TMFAIL - the resource is dissociated because part of the work has failed. This typically can be caused by an error exception encountered on the resource in use.
TMSUCCESS - the resource is dissociated as part of the normal work completion.

Returns:
true if the dissociation of the resource is successful; otherwise *false*.

Throws: IllegalStateException
Thrown if the transaction in the target object is inactive.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`enlistResource`**

```
public abstract boolean enlistResource(XAResource xaRes)
        throws RollbackException, IllegalStateException,
        SystemException
```

Enlist the resource specified with the transaction associated with the target Transaction object.

Parameters:
`xaRes`
The XAResource object associated with the resource (connection).

Returns:
true if the enlistment is successful; otherwise *false*.

Throws: IllegalStateException
Thrown if the transaction in the target object is in prepared state or the transaction is inactive.

Throws: RollbackException
Thrown to indicate that the transaction has been marked for rollback only.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`getStatus`**

```
public abstract int getStatus() throws SystemException
```

Obtain the status of the transaction associated with the target object.

Returns:
The transaction status. If no transaction is associated with the target object, this method returns the STATUS_NO_TRANSACTION value.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`registerSynchronization`**

```
public void registerSynchronization(Synchronization sync)
    throws RollbackException, IllegalStateException,
    SystemException
```

Register a synchronization object for the transaction currently associated with the target object. The transc-tion manager invokes the `beforeCompletion` method prior to starting the transaction commit process. After the transaction is completed, the transaction manager invokes the `afterCompletion` method.

Parameters:

`sync`

The Synchronization object for the transaction currently associated with the target object.

Throws: IllegalStateException

Thrown if the transaction in the target object is in prepared state or the transaction is inactive.

Throws: RollbackException

Thrown to indicate that the transaction has been marked for rollback only.

Throws: SystemException

Thrown if the transaction manager encounters an unexpected error condition.

- **`rollback`**

```
public abstract void rollback()
       throws IllegalStateException, SyetemException
```

Rollback the transaction associated with the target Transaction object.

Throws: IllegalStateException

Thrown if the target object is not associtated with any transaction.

Throws: SystemException

Thrown if the transaction manager encounters an unexpected error condition.

- **`setRollbackOnly`**

```
public abstract void setRollbackOnly()
             throws IllegalStateException, SystemException
```

Modify the transaction associated with the target object such that the only possible outcome of the transac-tion is to roll back the transaction.

Throws: IllegalStateException

Thrown if the target object is not associtated with any transaction.

Throws: SystemException

Thrown if the transaction manager encounters an unexpected error condition.

Variables

- **`TMSUCCESS`**

```
public final static int TMSUCCESS =    0x04000000
```

Dissociate caller from transaction branch.

- **`TMSUSPEND`**

```
public final static int TMSUSPEND =    0x02000000
```

Caller is suspending (not ending) association with transaction branch.

- **TMFAIL**

  `public final static int TMFAIL =       0x20000000`

  Dissociates the caller and mark the transaction branch rollback-only.

Interface TransactionManager

```
interface javax.transaction.TransactionManager
{
   public abstract void begin();
   public abstract void commit();
   public abstract int getStatus();
   public abstract Transaction getTransaction();
   public void resume(Transaction tobj);
   public abstract void rollback();
   public abstract void setRollbackOnly();
   public abstract void setTransactionTimeout(int seconds);
   public abstract Transaction suspend() ;
}
```

The TransactionManager interface allows the application server to communicate to the Transaction Manager for transaction boundaries demarcation on behalf of the application. For example, this interface is used by an EJB server to communicate to the transaction manager on behalf of the container-managed EJB components.

Methods

- **begin**

```
public abstract void begin()
            throws NotSupportedException, SystemException
```

Create a new transaction and associate it with the current thread.

Throws: NotSupportedException
Thrown if the thread is already associated with a transaction and the Transaction Manager does not support nested transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **commit**

```
public abstract void commit()
        throws RollbackException, HeuristicMixedException,
            HeuristicRollbackException, SecurityException,
            IllegalStateException, SystemException
```

Complete the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: RollbackException
Thrown to indicate that the transaction has been rolled back rather than committed.

Throws: HeuristicMixedException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.

Throws: HeuristicRollbackException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to commit the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`getStatus`**

```
public abstract int getStatus() throws SystemException
```

Obtain the status of the transaction associated with the current thread.

Returns:
The transaction status. If no transaction is associated with the current thread, this method returns the Status.STATUS_NO_TRANSACTION value.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`getTransaction`**

```
public abstract Transaction getTransaction() throws SystemException
```

Get the transaction object that represents the transaction context of the calling thread.

Returns:
The Transaction object that represents the transaction context of the calling thread. If the calling thread is not associated with a transaction, a null object reference is returned.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`resume`**

```
public void resume(Transaction tobj)
            throws InvalidTransactionException,
               IllegalStateException, SystemException
```

Resume the transaction context association of the calling thread with the transaction represented by the supplied Transaction object. When this method returns, the calling thread is associated with the transaction context specified.

Parameters:
`tobj`
The Transaction object that consists of the transaction to be resumed for the calling thread.

Throws: InvalidTransactionException
Thrown if the parameter *tobj* refers to an invalid transaction.

Throws: IllegalStateException
Thrown if the current thread is already associated with another transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`rollback`**

```
public abstract void rollback()
```

```
throws IllegalStateException, SecurityException, SystemException
```

Roll back the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **setRollbackOnly**

```
public abstract void setRollbackOnly()
                throws IllegalStateException, SystemException
```

Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **setTransactionTimeout**

```
public abstract void setTransactionTimeout(int seconds)
                              throws SystemException
```

Modify the value of the timeout value that is associated with the transactions started by the current thread with the begin method.

If an application has not called this method, the transaction service uses some default value for the transaction timeout.

Parameters:
`seconds`
The value of the timeout in seconds. If the value is zero, the transaction service restores the default value.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **suspend**

```
public abstract Transaction suspend() throws SystemException
```

Suspend the transaction currently associated with the calling thread and return a Transaction object that represents the transaction context being suspended. If the calling thread is not associated with a transaction, the method returns a null object reference. When this method returns, the calling thread is associated with no transaction.

Returns:
The Transaction object that represents the transaction context associated with the calling thread. Null if the calling thread is not associated with a transaction.

Throws: SystemException

Thrown if the transaction manager encounters an unexpected error condition.

Interface UserTransaction

```
public interface javax.transaction.UserTransaction
{
   public abstract void begin();
   public abstract void commit();
   public abstract int getStatus();
   public abstract void rollback();
   public abstract void setRollbackOnly();
   public abstract void setTransactionTimeout(int seconds);
}
```

The UserTransaction interface defines the methods that allow an application to explicitly manage transaction boundaries.

Methods

- **begin**

```
public abstract void begin()
     throws NotSupportedException, SystemException
```

Create a new transaction and associate it with the current thread.

Throws: NotSupportedException
Thrown if the thread is already associated with a transaction and the Transaction Manager implementation does not support nested transactions.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **commit**

```
public abstract void commit()
        throws RollbackException, HeuristicMixedException,
            HeuristicRollbackException, SecurityException,
            IllegalStateException, SystemException
```

Complete the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: RollbackException
Thrown to indicate that the transaction has been rolled back rather than committed.

Throws: HeuristicMixedException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.

Throws: HeuristicRollbackException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to commit the transaction.

Throws: IllegalStateException

Thrown if the current thread is not associated with a transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`getStatus`**

```
public abstract int getStatus() throws SystemException
```

Obtain the status of the transaction associated with the current thread.

Returns:
The transaction status. If no transaction is associated with the current thread, this method returns the STATUS_NO_TRANSACTION value.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`rollback`**

```
public abstract void rollback()
        throws IllegalStateException, SecurityException, SystemException
```

Roll back the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`setRollbackOnly`**

```
public abstract void setRollbackOnly()
                  throws IllegalStateException, SystemException
```

Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

- **`setTransactionTimeout`**

```
public abstract void setTransactionTimeout(int seconds)
                        throws SystemException
```

Modify the value of the timeout value that is associated with the transactions started by the current thread with the begin method.

If an application has not called this method, the transaction service uses some default value for the transaction timeout.

Parameters:
`seconds`

The value of the timeout in seconds. If the value is zero, the transaction service restores the default value.

Throws: SystemException
Thrown if the transaction manager encounters an unexpected error condition.

Interface XAResource

```
public interface javax.transaction.xa.XAResource
{
   public abstract void commit(Xid xid, boolean onePhase);
   public abstract void end(Xid xid, int flags);
   public abstract void forget(Xid xid);
   public abstract int getTransactionTimeout();
   public abstract boolean isSameRM(XAResource xares);
   public abstract int prepare(Xid xid);
   public abstract Xid[] recover(int flag);
   public abstract void rollback(Xid xid);
   public abstract boolean setTransactionTimeout(int seconds);
   public abstract void start(Xid xid, int flags);
}
```

XAResource interface is a Java mapping of the industry standard XA resource manager interface. Please refer to: X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3.

Methods

- **commit**

  `commit(Xid xid, boolean onePhase) throws XAException`

  This method is called to commit the global transaction specified by `xid`.

  Parameters:

  `xid`
  A global transaction identifier.

  `onePhase`
  If true, the resource manager should use a one-phase commit protocol to commit the work done on behalf of *xid*.

  Throws: XAException
  An error has occurred. Possible XAExceptions are XA_HEURHAZ, XA_HEURCOM, XA_HEURRB, XA_HEURMIX, XAER_RMERR, XAER_RMFAIL, XAER_NOTA, XAER_INVAL, or XAER_PROTO.

  If the resource manager did not commit the transaction and the paramether onePhase is set to *true*, the resource manager may throw one of the XA_RB* exceptions. Upon return, the resource manager has rolled back the branch's work and has released all held resources.

- **end**

  `int end(Xid xid, int flags) throws XAException`

  This method ends the work performed on behalf of a transaction branch. The resource manager dissociates the XA resource from the transaction branch specified and let the transaction be completed.

  If TMSUSPEND is specified in *flags*, the transaction branch is temporarily suspended in incomplete state. The transaction context is in suspened state and must be resumed via *start* with TMRESUME specified.

If TMFAIL is specified, the portion of work has failed. The resource manager may mark the transaction as rollback-only.

If TMSUCCESS is specified, the portion of work has completed successfully.

Parameters:

`xid`

A global transaction identifier that is the same as what was used previously in the *start* method.

`flags`

One of TMSUCCESS, TMFAIL, or TMSUSPEND.

Throws: XAException

An error has occurred. Possible XAException values are XAER_RMERR, XAER_RMFAILED, XAER_NOTA, XAER_INVAL, XAER_PROTO, or XA_RB*.

- **`forget`**

`void forget(Xid xid) throws XAException`

This method is called to tell the resource manager to forget about a heuristically completed transaction branch.

Parameters:

`xid`

A global transaction identifier.

Throws: XAException

An error has occurred. Possible exception values are XAER_RMERR, XAER_RMFAIL, XAER_NOTA, XAER_INVAL, or XAER_PROTO.

- **`getTransactionTimeout`**

`int getTransactionTimeout() throws XAException`

This method returns the transaction timeout value set for this `XAResource` instance. If `XAResource.setTransactionTimeout` was not use prior to invoking this method, the return value is the default timeout set for the resource manager; otherwise, the value used in the previous `setTransactionTimeout` call is returned.

Throws: XAException

An error has occurred. Possible exception values are: XAER_RMERR, XAER_RMFAIL.

Returns:

The transaction timeout values in seconds.

- **`isSameRM`**

`boolean isSameRM(XAResource xares) throws XAException`

This method is called to determine if the resource manager instance represented by the target object is the same as the resource manager instance represented by the parameter *xares* .

Parameters:

`xares`

An XAResource object.

Returns:

true if same RM instance; otherwise *false*.

Throws: XAException

An error has occurred. Possible exception values are: XAER_RMERR, XAER_RMFAIL.

- **`prepare`**

`int prepare(Xid xid) throws XAException`

This method is called to ask the resource manager to prepare for a transaction commit of the transaction specified in *xid*.

`Parameters:`

`xid`

A global transaction identifier.

Throws: XAException

An error has occurred. Possible exception values are: XA_RB*, XAER_RMERR, XAER_RMFAIL, XAER_NOTA, XAER_INVAL, or XAER_PROTO.

`Returns:`

A value indicating the resource manager's vote on the outcome of the transaction. The possible values are: XA_RDONLY or XA_OK. If the resource manager wants to roll back the transaction, it should do so by throwing an appropriate XAException in the `prepare` method.

- **`recover`**

`xid[] recover(int flag) throws XAException`

This method is called to obtain a list of prepared transaction branches from a resource manager. The transaction manager calls this method during recovery to obtain the list of transaction branches that are currently in prepared or heuristically completed states.

The flag parameter indicates where the recover scan should start or end, or start and end. This method may be invoked one or more times during a recovery scan. The resource manager maintains a cursor which marks the current position of the prepared or heuristically completed transaction list. Each invocation of the recover method moves the cursor passed the set of Xids that are returned.

Two consecutive invocation of this method that starts from the beginning of the list must return the same list of transaction branches unless one of the following takes place:

- the transaction manager invokes the commit, forget, prepare, or rollback method for that resource manager, between the two consecutive invocation of the recovery scan.

- the resource manager heuristically completes some transaction branches between the two invocation of the recovery scan.

`Parameters:`

`flag`

One of TMSTARTRSCAN, TMENDRSCAN, TMNOFLAGS. TMNOFLAGS must be used when no other flags are used.

TMSTARTRSCAN - indicates that the recovery scan should be started at the beginning of the prepared or heuristically completed transaction list.

TMENDRSCAN - indicates that the recovery scan should be ended after the method returns the Xid list. If this flag is used in conjunction with the TMSTARTRSCAN, this method invocation starts and ends the recovery scan.

TMNOFLAGS - this flag must be used when no other flags are specified. This flag may be used only if the recovery scan has already been started. The list of Xids are returned

Returns: xid[]

The resource manager returns zero or more Xids for the transaction branches that are currently in a prepared or heuristically completed state. If an error occurs during the operation, the resource manager should throw the appropriate `XAException`.

Throws: XAException

An error has occurred. Possible values are XAER_RMERR, XAER_RMFAIL, XAER_INVAL, and XAER_PROTO.

- **`rollback`**

```
void rollback(Xid xid) throws XAException
```

This method informs the resource manager to roll back work done on behalf of a transaction branch.

`Parameters:`

`xid`

A global transaction identifier.

Throws: XAException

An error has occurred.

- **`setTransactionTimeout`**

```
boolean setTransactionTimeout(int seconds) throws XAException
```

This method sets the transaction timeout value for this `XAResource` instance. Once set, this timeout value is effective until `setTransactionTimeout` is invoked again with a different value. To reset the timeout value to the default value used by the resource manager, set the value to zero.

If the timeout operation is performed successfully, the method returns *true*; otherwise *false*. If a resource manager does not support transaction timeout value to be set explicitly, this method returns *false*.

`Parameters:`

`seconds`

An positive integer specifying the timout value in seconds. Zero resets the transaction timeout value to the default one used by the resource manager. A negative value results in `XAException` to be thrown with XAER_INVAL error code.

`Returns:`

true if transaction timeout value is set successfully; otherwise *false*.

Throws: XAException

An error has occurred. Possible exception values are: XAER_RMERR, XAER_RMFAIL, or XAER_INVAL.

- **`start`**

```
void start(Xid xid, int flags) throws XAException
```

This method starts work on behalf of a transaction branch.

If TMJOIN is specified, the start is for joining an exisiting transaction branch `xid`. If TMRESUME is specified, the start is to resume a suspended transaction branch specified in `xid`.

If neither TMJOIN nor TMRESUME is specified and the transaction branch specified in `xid` already exists, the resource manager throw the XAException with XAER_DUPID error code.

Parameters:

`xid`
A global transaction identifier to be associated with the resource.

`flags`
One of TMNOFLAGS, TMJOIN, or TMRESUME.

Throws: XAException
An error has occurred. Possible exceptions are XA_RB*, XAER_RMERR, XAER_RMFAIL, XAER_DUPID, XAER_OUTSIDE, XAER_NOTA, XAER_INVAL, or XAER_PROTO.

Variables

- **TMENDRSCAN**

  `public final static int TMENDRSCAN =   0x00800000`

  End a recovery scan.

- **TMFAIL**

  `public final static int TMFAIL =       0x20000000`

  Dissociates the caller and mark the transaction branch rollback-only.

- **TMJOIN**

  `public final static int TMJOIN =       0x00200000`

  Caller is joining existing transaction branch.

- **TMNOFLAG**

  `public final static int TMNOFLAG =     0x00000000`

  Use TMNOFLAG to indicate no flags value is selected.

- **TMONEPHASE**

  `public final static int TMONEPHASE =   0x40000000`

  Caller is using one-phase optimization.

- **TMRESUME**

  `public final static int TMRESUME =     0x08000000`

  Caller is resuming association with with suspended transaction branch.

- **TMSTARTRSCAN**

  `public final static int TMSTARTRSCAN = 0x01000000`

  Start a recovery scan.

- **TMSUCCESS**

  `public final static int TMSUCCESS =    0x04000000`

  Dissociate caller from transaction branch.

- **`TMSUSPEND`**

```
public final static int TMSUSPEND =    0x02000000
```

Caller is suspending (not ending) association with transaction branch.

Interface Xid

```
public interface javax.transaction.xa.Xid
{
      int getFormatId();
      byte[] getGlobalTransactionId();
      byte[] getBranchQualifier();
}
```

The `Xid` interface is a Java mapping of the X/Open transaction identifier `Xid` structure. This interface is used by the transaction manager to communicate to the resource manager for associating a transaction to the `XAResource`.

Variables

- **`MAXGTRIDSIZE`**

  `final static int MAXGTRIDSIZE = 64`

  Maximum number of bytes returned by `getGlobalTransactionId` method.

- **`MAXBQUALSIZE`**

  `final static int MAXBQUALSIZE = 64`

  Maximum number of bytes returned by `getBranchQualifier` method

Methods

- **`getFormatId`**

  `int getFormatID()`

  Obtain the format identifier part of the Xid.

  Returns:
  Format identifier. 0 means the OSI CCR format.

- **`getGlobalTransactionId`**

  `byte[] getGtrid()`

  Obtain the global transaction identifier part of the Xid in a byte array.

  Returns:
  A byte array containing the global transaction identifier.

- **`getBranchQualifier`**

  `byte[] getBqual()`

  Obtain the transaction branch qualifier part of the Xid in a byte array.

  Returns:
  A byte array containing the branch qualifier of the transaction.

Class HeuristicCommitException

```
public class javax.transaction.HeuristicCommitException
      extends java.lang.Exception
{
    public HeuristicCommitException();
    public HeuristicCommitException(String msg);
}
```

This exception is thrown by the rollback operation on a resource to report that a heuristic decision was made and that all relevant updates have been committed.

Constructors

- **HeuristicCommitException**

  `public HeuristicCommitException()`

- **HeuristicCommitException**

  `public HeuristicCommitException(String msg)`

Class HeuristicMixedException

```
public class javax.transaction.HeuristicMixedException
      extends java.lang.Exception
{
    public HeuristicMixedException();
    public HeuristicMixedException(String msg);
}
```

This exception is thrown to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

Constructors

- **HeuristicMixedException**

  `public HeuristicMixedException()`

- **HeuristicMixedException**

  `public HeuristicMixedException(String msg)`

Class HeuristicRollbackException

```
public class javax.transaction.HeuristicRollbackException
      extends java.lang.Exception
{
    public HeuristicRollbackException();
    public HeuristicRollbackException(String msg);
}
```

This exception is thrown by the commit operation to report that a heuristic decision was made and that all relevant updates have been rolled back.

Constructors

- **HeuristicRollbackException**

  `public HeuristicRollbackException()`

- **HeuristicRollbackException**

  `public HeuristicRollbackException(String msg)`

Class InvalidTransactionException

```
public class javax.transaction.InvalidTransactionException
      extends java.rmi.RemoteException
{
    public InvalidTransactionException();
    public InvalidTransactionException(String msg);
}
```

This exception indicates that the request carried an invalid transaction context. This exception is used by any module that needs to indicate the invalid transaction context to the remote client.

Constructors

- **InvalidTransactionException**

  `public InvalidTransactionException()`

- **InvalidTransactionException**

  `public InvalidTransactionException(String msg)`

Class NotSupportedException

```
public class javax.transaction.NotSupportedException
      extends java.lang.Exception
{
    public NotSupportedException();
    public NotSupportedException(String msg);
}
```

This exception is thrown when the requested operation is not supported. For example, this exception can be thrown by the Transaction Manager to indicate that nested transaction is not supported. If *Transaction begin* is called when the calling thread is already associated with a transaction context and the Transaction Manager implementation does not support nested transactions, then this exception is thrown by the Transaction Manager.

Constructors

- **NotSupportedException**

  `public NotSupportedException()`

- **NotSupportedException**

  `public NotSupportedException(String msg)`

Class RollbackException

```
public class javax.transaction.RollbackException
      extends java.lang.Exception
{
    public RollbackException();
    public RollbackException(String msg);
}
```

This exception is thrown when the transaction has been marked for rollback only or the transaction has been rolledback instead of committed. This is a local exception thrown by methods in the `UserTransaction`, `Transaction` and `TransactionManager` interfaces.

Constructors

- **RollbackException**

  `public RollbackException()`

- **RollbackException**

  `public RollbackExcpetion(String msg)`

Class SystemException

```
public class javax.transaction.SystemException extends java.lang.Exception
{
   public SystemException();
   public SystemException(String s);
   public SystemException(int errCode);
}
```

The `SystemException` is thrown by the Transaction Manager to indicate that it has encountered an unexpected error condition that prevents future transaction services from proceeding.

Constructors

- **`SystemException`**

  `public SystemException()`

  Create a `SystemException`.

- **`SystemException`**

  `public SystemException(String s)`

  Create a `SystemException` with the specified string.

- **`SystemException`**

  `public SystemException(int errCode)`

  Create a `SystemException` with the specified error code.

Variables

- **`errorCode`**

  `public int errorCode`

  Error code for the exception

Class TransactionRequiredException

```
public class javax.transaction.TransactionRequiredException
    extends java.rmi.RemoteException
{
    public TransactionRequiredException();
    public TransactionRequiredException(String msg);
}
```

This exception indicates that a request carried a null transaction context, but the target object requires an active transaction. This exception is used by the system module that needs to indicate to the remote client about the error condition.

Constructors

- **TransactionRequiredException**

  `public TransactionRequiredException()`

- **TransactionRequiredException**

  `public TransactionRequiredException(String msg)`

Class TransactionRolledbackException

```
public class javax.transaction.TransactionRolledbackException
    extends java.rmi.RemoteException
{
    public TransactionRolledbackException();
    public TransactionRolledbackException(String msg);
}
```

This exception indicates that the transaction associated with processing of the request has been rolled back, or marked for roll back. Thus the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless. This exception is thrown by a system module to indicate to the remote client about the aborted transaction.

Constructors

- **TransactionRolledbackException**

  `public TransactionRolledbackException()`

- **TransactionRolledbackException**

  `public TransactionRolledbackException(String msg)`

Class XAException

```
public class javax.transactioin.xa.XAException extends java.lang.Exception
{
   public XAException();
   public XAException(String s);
   public XAException(int errCode);
}
```

The XAException is thown by the Resource Manager (RM) to inform the Transaction Manager of error encountered for the transaction involved.

Constructors

- **XAException**

  `public XAException()`

  Create an XAException.

- **XAException**

  `public XAExeption(String s)`

  Create an XAException with the specified string.

- **XAException**

  `public XAException(int errCode)`

  Create an XAException with the specified error code.

Variables

- **errorCode**

  `public int errorCode`

  Error code for the exception

- **XA_RBBASE**

  `public final static int XA_RBBASE = 100`

  The inclusive lower bound of the rollback code.

- **XA_RBROLLBACK**

  `public final static int XA_RBROLLBACK = XA_RBBASE`

  The rollback was caused by an unspecified reason.

- **XA_RBCOMMFAIL**

  `public final static int XA_RBCOMMFAIL = XA_RBBASE + 1`

  The rollback was caused by a communication failure.

- **XA_RBDEADLOCK**
  `public final static int XA_RBDEADLOCK = XA_RBBASE + 2`

  A deadlock was detected.

- **XA_RBINTEGRITY**
  `public final static int XA_RBINTEGRITY = XA_RBBASE + 3`

  A condition that violates the integrity of the resources was detected.

- **XA_RBOTHER**
  `public final static int XA_RBOTHER = XA_RBBASE + 4`

  The resouce manager rolled back the transaction branch for a reason not on this list.

- **XA_RBPROTO**
  `public final static int XA_RBPROTO = XA_RBBASE + 5`

  A protocol error occurred in the resource manager.

- **XA_RBTIMEOUT**
  `public final static int XA_RBRBTIMEOUT = XA_RBBASE + 6`

  A transaction branch took too long.

- **XA_RBTRANSIENT**
  `public final static int XA_RBTRANSIENT = XA_RBBASE + 7`

  May retry the transaction branch

- **XA_RBEND**
  `public final static int XA_RBEND = XA_RBTRANSIENT`

  The inclusive upper bound of the rollback codes.

- **XA_NOMIGRATE**
  `public final static int XA_NOMIGRATE = 9`

  Resumption must occur where suspension occurred.

- **XA_HEURHAZ**
  `public final static int XA_HEURHAZ = 8`

  The transaction branch may have been heuristically completed.

- **XA_HEURCOM**
  `public final static int XA_HEURCOM = 7`

  The transaction branch has been heuristically committed.

- **XA_HEURRB**
  `public final static int XA_HEURRB = 6`

  The transaction branch has been heuristically rolled back.

- **XA_HEURMIX**

  `public final static int XA_HEURMIX = 5`

  The transaction branch has been heuristically committed and rolled back.

- **XA_RDONLY**

  `public final static int XA_RDONLY = 3`

  The transaction branch was read-only and has been committed.

- **XAER_RMERR**

  `public final static int XAER_RMERR = -3`

  A resource manager error occurred in the transaction branch

- **XAER_NOTA**

  `public final static int XAER_NOTA = -4`

  The XID is not valid.

- **XAER_INVAL**

  `public final static int XAER_INVAL = -5`

  Invalid arguments were given.

- **XAER_PROTO**

  `public final static int XAER_PROTO = -6`

  Routine invoked in an improper context.

- **XAER_RMFAIL**

  `public final static int XAER_RMFAIL = -7`

  Resource manager unavailable.

- **XAER_DUPID**

  `public final static int XAER_DUPID = -8`

  The XID already exists.

- **XAER_OUTSIDE**

  `public final static int XAER_OUTSIDE = -9`

  Resource manager doing work outside global transaction.

6 Related documents

[1] X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3

[2] Java Transaction Service (JTS). *http://java.sun.com/products/jts*

[3] OMG Object Transaction Service (OTS 1.1)
http://www.omg.org/corba/sectrans.html#trans.

[4] ORB Portability Submission, OMG document orbos/97-04-14.

[5] Enterprise JavaBeans™. *http://java.sun.com/products/ejb*.

[6] JDBC™ 2.0. *http://java.sun.com/products/jdbc.*

[7] Java Message Service. *http://java.sun.com/products/jms*