

---

# *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:**

[users@jta-spec.java.net](mailto:users@jta-spec.java.net)

Copyright © 2102 by Oracle Corporation  
500 Oracle Parkway, Redwood City, California 94065  
U.S.A. All rights reserved.

## *License*

Specification: JSR-907 Java Transaction API 1.2 ("Specification") Version: 1.2

Status: Public Draft Review

Release: 19 December 2012

Copyright 2012 Oracle America, Inc.

500 Oracle Parkway, Redwood City, California 94065, U.S.A.

All rights reserved.

### NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Oracle America, Inc. ("Oracle") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Oracle's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;

(ii) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and

(iii) Includes the following notice:

"This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Oracle intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Oracle if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof

### TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

### DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY ORACLE. ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON- INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product. THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. ORACLE MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

### LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE

### License

SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Oracle (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

### RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set

forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

### REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Oracle with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

### GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

# *Table of Contents*

1. Introduction .....	7
1.1 Background .....	7
1.2 Target Audience .....	9
2. Relationship to Other Java APIs .....	10
2.1 Enterprise JavaBeans .....	10
2.2 JDBC 4.1 Standard Extension .....	10
2.3 Java Message Service .....	10
2.4 Java Transaction Service .....	10
3. Java Transaction API .....	11
3.1 UserTransaction Interface .....	11
3.1.1 UserTransaction Support in EJB Server .....	11
3.1.2 UserTransaction Support for Transactional Clients .....	12
3.2 TransactionManager Interface .....	12
3.2.1 Starting a Transaction .....	13
3.2.2 Completing a Transaction .....	13
3.2.3 Suspending and Resuming a Transaction .....	13
3.3 Transaction Interface .....	14
3.3.1 Resource Enlistment .....	15
3.3.2 Transaction Synchronization.....	15
3.3.3 Transaction Completion .....	17
3.3.4 Transaction Equality and Hash Code .....	17
3.4 XAResource Interface .....	17
3.4.1 Opening a Resource Manager .....	19
3.4.2 Closing a Resource Manager .....	19
3.4.3 Thread of Control .....	19
3.4.4 Transaction Association .....	20
3.4.5 Externally Controlled Connections .....	21
3.4.6 Resource Sharing .....	21
3.4.7 Local and Global Transactions .....	22
3.4.8 Failures Recovery .....	22
3.4.9 Identifying The Resource Manager Instance .....	23
3.4.10 Dynamic Registration .....	24
3.5 Xid Interface .....	24
3.6 TransactionSynchronizationRegistry Interface .....	24
3.7 Transactional Annotation .....	25
3.8 TransactionScoped Annotation .....	27
4. JTA Support in Application Server .....	30
4.1 Connection-Based Resource Usage Scenario .....	30
4.2 Transaction Association and Connection Request Flow .....	32
5. Java Transaction API Reference .....	34

6. Related Documents.....	67
---------------------------	----

# 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

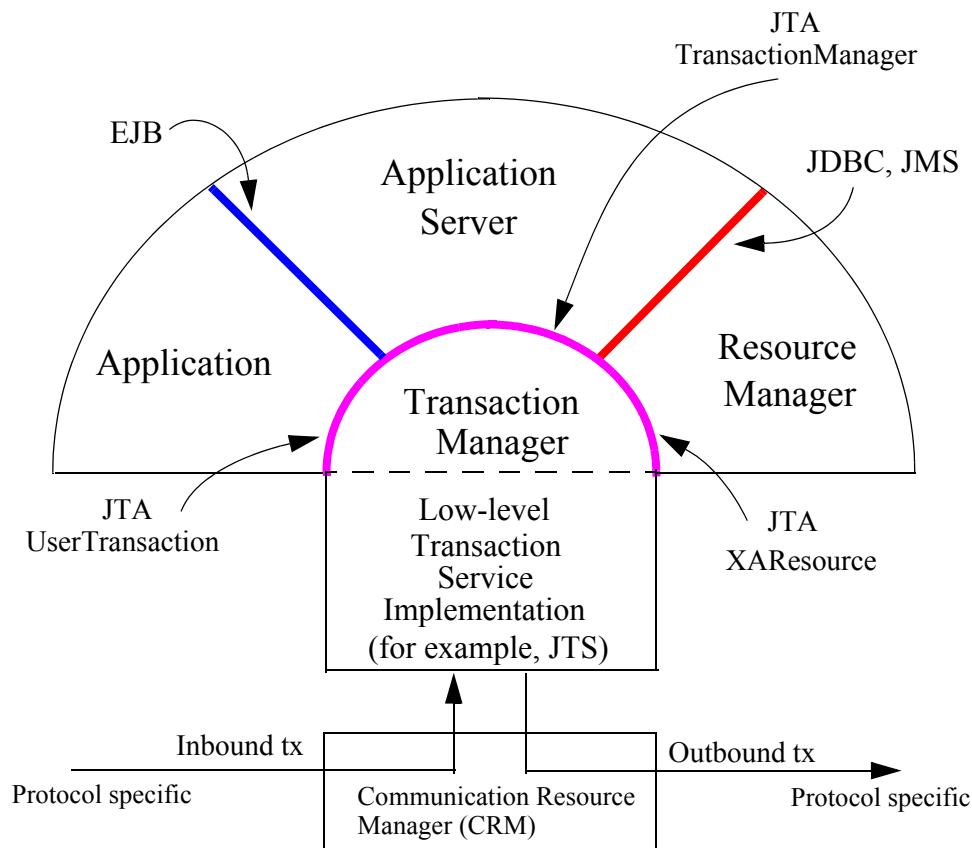
Distributed transaction services in Enterprise Java middleware involves five players: the transaction manager, the application server, the resource manager, the application program, and the communication resource manager. Each of these players contributes to the distributed transaction processing system 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) provides the application access to resources. The resource manager participates in distributed transactions by implementing a transaction resource interface used by the 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 beans with bean-managed transactions) and by the client programmer who wants to explicitly demarcate transaction boundaries within programs that are written in the Java programming language.

### 2.2 JDBC 4.1 Standard Extension API

One of the new features included in the JDBC 4.1 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 4.1 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 4.1 Standard Extension Specification for further details.

### 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 2.0 Specification for further details.

### 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 transactions 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 bean-managed transactions. 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 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 transactions 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 resource 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<sup>1</sup> and

ensures that 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.

A container only needs to call `delistResource` to explicitly disassociate a resource from a transaction and it is not a mandatory container requirement to do so as a precondition to transaction completion. A transaction manager is, however, required to implicitly ensure the association of any associated `XAResource` is ended, via the appropriate `XAResource.end` call, immediately prior to completion; that is before prepare (or commit/rollback in the onephase-optimized case).

### 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:

---

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.”



- The `Synchronization.beforeCompletion` method is called prior to the start of the two-phase transaction commit process. This call is executed with the transaction context of the transaction that is being committed.
- The `Synchronization.afterCompletion` method is called after the transaction has completed. The status of the transaction is supplied in the parameter.

### 3.3.3 Transaction Completion

The `Transaction.commit` and `Transaction.rollback` methods allow the target object to be committed 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);
```

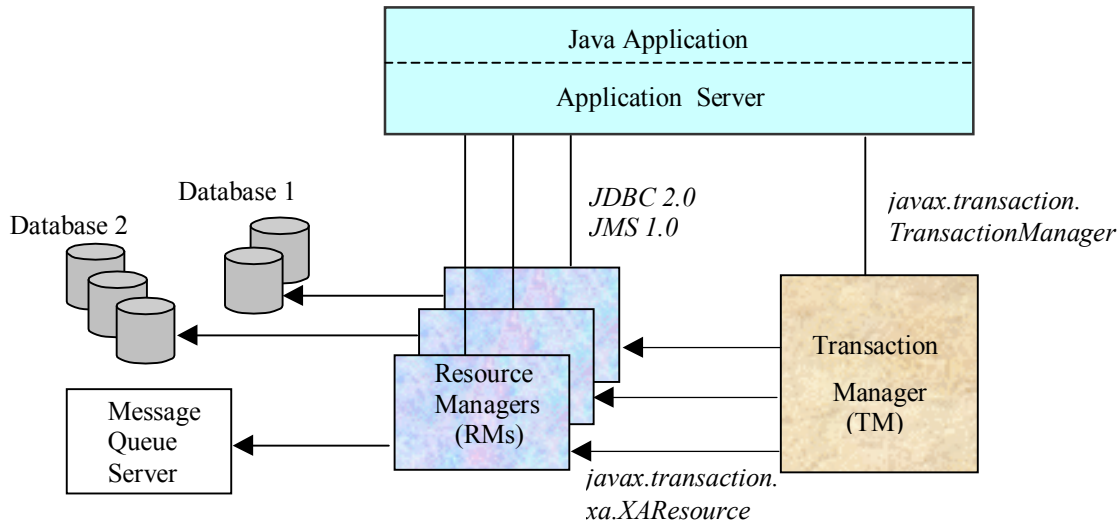
In addition, the transaction manager must implement the `Transaction` object's `hashCode` method so that if two `Transaction` objects are equal, they have the same hash code. However, the converse is not necessarily true. Two `Transaction` objects with the same hash code are not necessarily equal.

## 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 transaction resource 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 <sub>0</sub>	Associated T <sub>1</sub>	Association Suspended T <sub>2</sub>
<i>start()</i>	T <sub>1</sub>		
<i>start</i> (TMRESUME)			T <sub>1</sub>
<i>start</i> (TMJOIN)	T <sub>1</sub>		
<i>end</i> (TMSUSPEND)		T <sub>2</sub>	
<i>end</i> (TMFAIL)		T <sub>0</sub>	T <sub>0</sub>
<i>end</i> (TMSUCCESS)		T <sub>0</sub>	T <sub>0</sub>
<i>recover()</i>	T <sub>0</sub>	T <sub>1</sub>	T <sub>2</sub>

### 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 xidl is the target transaction and
    // xidl 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(xidl, TMJOIN);
    } else {
        //
    }
}
```

```

        // This is a different RM instance,
        // make a new transaction branch for xidl
        //
        xidlNewBranch = makeNewBranch(xidl);
        xares.start(xidlNewBranch, 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.

## 3.6 TransactionSynchronizationRegistry Interface

The `javax.transaction.TransactionSynchronizationRegistry` interface is intended for use by system level application server components such as persistence managers. This provides the ability to register synchronization objects with special ordering semantics, associate resource objects with the current transaction, get the transaction context of the current transaction, get current transaction status, and mark the current transaction for rollback.

This interface is implemented by the application server as a stateless service object. The same object can be used by any number of components with complete thread safety. In standard application server environments, an instance implementing this interface can be looked up via JNDI using a standard name.

The user of `getResource()` and `putResource()` methods is a library component that manages transaction-specific data on behalf of a caller. The transaction-specific data provided by the caller is not immediately flushed to a transaction-enlisted resource, but



instead is cached. The cached data is stored in a transaction-related data structure that is in a zero-or-one-to-one relationship with the transactional context of the caller.

An efficient way to manage such a transaction-related data structure is for the implementation of the `TransactionSynchronizationRegistry` to manage a `Map` for each transaction as part of the transaction state.

The keys of this `Map` are objects that are provided by the library components (users of the API). The values of the `Map` are any values that the library components are interested in storing, for example the transaction-related data structures. This `Map` has no concurrency issues since it is a dedicated instance for the transaction. When the transaction completes, the `Map` is cleared, releasing resources for garbage collection.

The scalability of the library code is significantly enhanced by the addition of the `getResource` and `putResource` methods to the `TransactionSynchronizationRegistry`.

“Java Transaction API Reference” on page 34 has a full description of this interface.

### 3.7 Transactional Annotation

The `javax.transaction.cdi.Transactional` annotation provides the application the ability to control transaction boundaries on CDI managed beans, as well as classes defined as managed beans by the Java EE specification such as servlets, JAX-RS resource classes, and JAX-WS service endpoints, declaratively. This support is provided via an implementation of CDI interceptors that conduct the necessary suspending, resuming, etc. A `javax.transaction.cdi.TransactionException` with a nested exception is thrown from the interceptor as appropriate. The `TxType` element of the annotation indicates whether a bean method is to be executed within a transaction context where the values provide the following corresponding behavior:

`TxType.REQUIRED`

If called outside a transaction context, a new JTA transaction will begin, the managed bean method execution will then continue inside this transaction context, and the transaction will be committed.

If called inside a transaction context, the managed bean method execution will then continue inside this transaction context.

`TxType.REQUIRES_NEW`

If called outside a transaction context, a new JTA transaction will begin, the managed bean method execution will then continue inside this transaction context, and the transaction will be committed.

If called inside a transaction context, the current transaction context will be suspended, a new JTA transaction will begin, the managed bean

method execution will then continue inside this transaction context, the transaction will be committed, and the previously suspended transaction will be resumed.

#### `TxType.MANDATORY`

If called outside a transaction context, a `TransactionException` with a nested `TransactionRequiredException` must be thrown.

If called inside a transaction context, managed bean method execution will then continue under that context.

#### `TxType.SUPPORTS`

If called outside a transaction context, managed bean method execution will then continue outside a transaction context.

If called inside a transaction context, the managed bean method execution will then continue inside this transaction context.

#### `TxType.NOT_SUPPORTED`

If called outside a transaction context, managed bean method execution will then continue outside a transaction context.

If called inside a transaction context, the current transaction context will be suspended, the managed bean method execution will then continue outside a transaction context, and the previously suspended transaction will be resumed.

#### `TxType.NEVER`

If called outside a transaction context, managed bean method execution will then continue outside a transaction context.

If called inside a transaction context, a `TransactionException` with a nested `InvalidException` must be thrown

By default checked exceptions do not result in the transactional interceptor marking the transaction for rollback and instances of `RuntimeException` and its subclasses do.

This default behavior can be overridden by specifying which exceptions result in the interceptor marking the transaction for rollback. The `rollbackOn` element can be set to indicate which exceptions should cause the interceptor to mark the transaction for rollback.

Conversely, the `dontRollbackOn` element can be set to indicate which exceptions should not cause the interceptor to mark the transaction for rollback. When a class is specified for either of these elements, the designated behavior applies to subclasses of that class as well. If both elements are specified, `dontRollbackOn` takes precedence.

The following are some example usages of `rollbackOn` and `dontRollbackOn` elements.

The following will override behavior for application exceptions, causing the transaction to be marked for rollback for all application exceptions.

```
@Transactional(rollbackOn={Exception.class})
```

The following will prevent transactions from being marked for rollback by the interceptor when an `IllegalStateException` or any of its subclasses reaches the interceptor.

```
@Transactional(dontRollbackOn={IllegalStateException.class})
```

The following will cause the transaction to be marked for rollback for all runtime exceptions and all `SQLException` types except for `SQLWarning`.

```
@Transactional(rollbackOn={SQLException.class}, dontRollbackOn={SQLWarning.class})
```

EJB application exceptions (i.e., runtime exceptions annotated with `@ApplicationException`) are treated just as any other runtime exceptions unless otherwise specified.

The `TransactionalException` thrown from the `Transactional` interceptors implementation is a `RuntimeException` and therefore the default behavior is to mark the transaction for rollback.

When `Transactional` annotated managed beans are used in conjunction with EJB container managed transactions the EJB container behavior must be applied before the bean is called. When the bean is called the CDI behavior is applied before calling the bean's methods. It is best practice to avoid such use of `Transactional` annotations in conjunction with EJB container managed transactions in order to avoid possible confusion.

“Java Transaction API Reference” on page 34 has a full description of this annotation.

### 3.8 TransactionScoped Annotation

The `javax.transaction.cdi.TransactionScoped` annotation provides the ability to specify a standard scope to define beans whose lifecycle are scoped to the currently active JTA transaction. The transaction scope is active when the return from a call to `UserTransaction.getStatus` or `TransactionManager.getStatus` is one of the following states:

```
Status.STATUS_ACTIVE  
Status.STATUS_MARKED_ROLLBACK  
Status.STATUS_PREPARED  
Status.STATUS_UNKNOWN  
Status.STATUS_PREPARING  
Status.STATUS_COMMITTING  
Status.STATUS_ROLLING_BACK
```

It is not intended that the term "active" as defined here in relation to TransactionScoped should also apply to its use in relation to transaction context, lifecycle, etc. mentioned elsewhere in this specification. The transaction context must be destroyed after any Synchronization.beforeCompletion methods are called and after completion calls have been made on enlisted resources but before any Synchronization.afterCompletion methods are called. A javax.enterprise.context.ContextNotActiveException must be thrown if an object with this annotation is used when the transaction context is not active. The object with this annotation is associated with the JTA transaction where it is first used and this association is retained through any transaction suspend or resume calls as well as any beforeCompletion Synchronization calls until the transaction is completed. The way in which the JTA transaction is begun and completed (eg BMT, CMT, etc.) is of no consequence. The contextual references used across different JTA transactions are distinct.

The following example test case illustrates the expected behavior:

TransactionScoped annotated CDI managed bean:

```
@TransactionScoped
public class TestCDITransactionScopeBean {
    public void test()
    {
        //...
    }
}
```

Test Class:

```
UserTransaction userTransaction;
TransactionManager transactionManager;
@Inject
TestCDITransactionScopeBean testTxAssociationChangeBean;

public void testTxAssociationChange() throws Exception {
    userTransaction.begin(); //tx1 begun
    testTxAssociationChangeBean.test();
    //assert testTxAssociationChangeBean instance has tx1 association
    Transaction transaction = transactionManager.suspend(); //tx1 suspended
    //assert testTxAssociationChangeBean still associated with tx1 and
    // that no transaction scope is active.
    userTransaction.begin(); //tx2 begun
    testTxAssociationChangeBean.test();
    //assert newtestTxAssociationChangeBean instance has tx2 association
    userTransaction.commit(); //tx2 committed, assert no transaction scope is active
    transactionManager.resume(tx); //tx1 resumed testTxAssociationChangeBean.test();
    //assert testTxAssociationChangeBean is original tx1 instance and not still
    // referencing committed/tx2 tx
}
```

```
        userTransaction.commit(); //tx1 commit, assert no transactionscope is active
    try {
        testTxAssociationChangeBean.test();
        fail("should have thrownContextNotActiveException");
    }catch (ContextNotActiveException ContextNotActiveException)
    {
        // do nothing intentionally
    }
}
```

“Java Transaction API Reference” on page 30 has a full description of this annotation.

## 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.

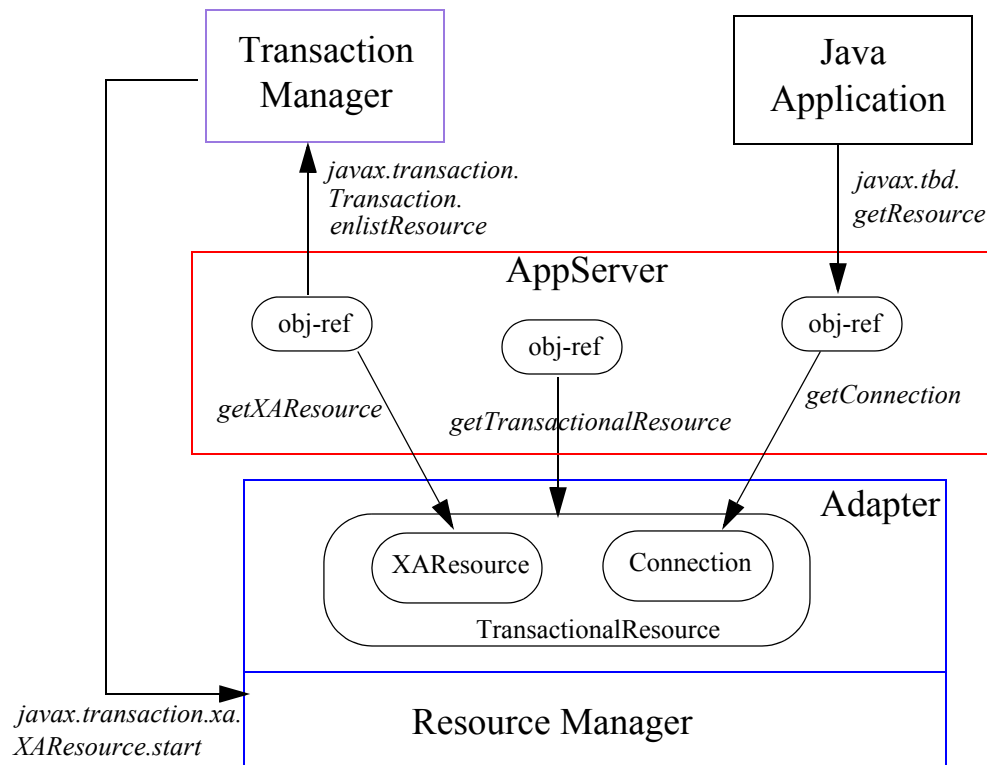
### 4.1 Connection-Based Resource Usage Scenario

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 illustrates 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.getTransactionResource();

// 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
```

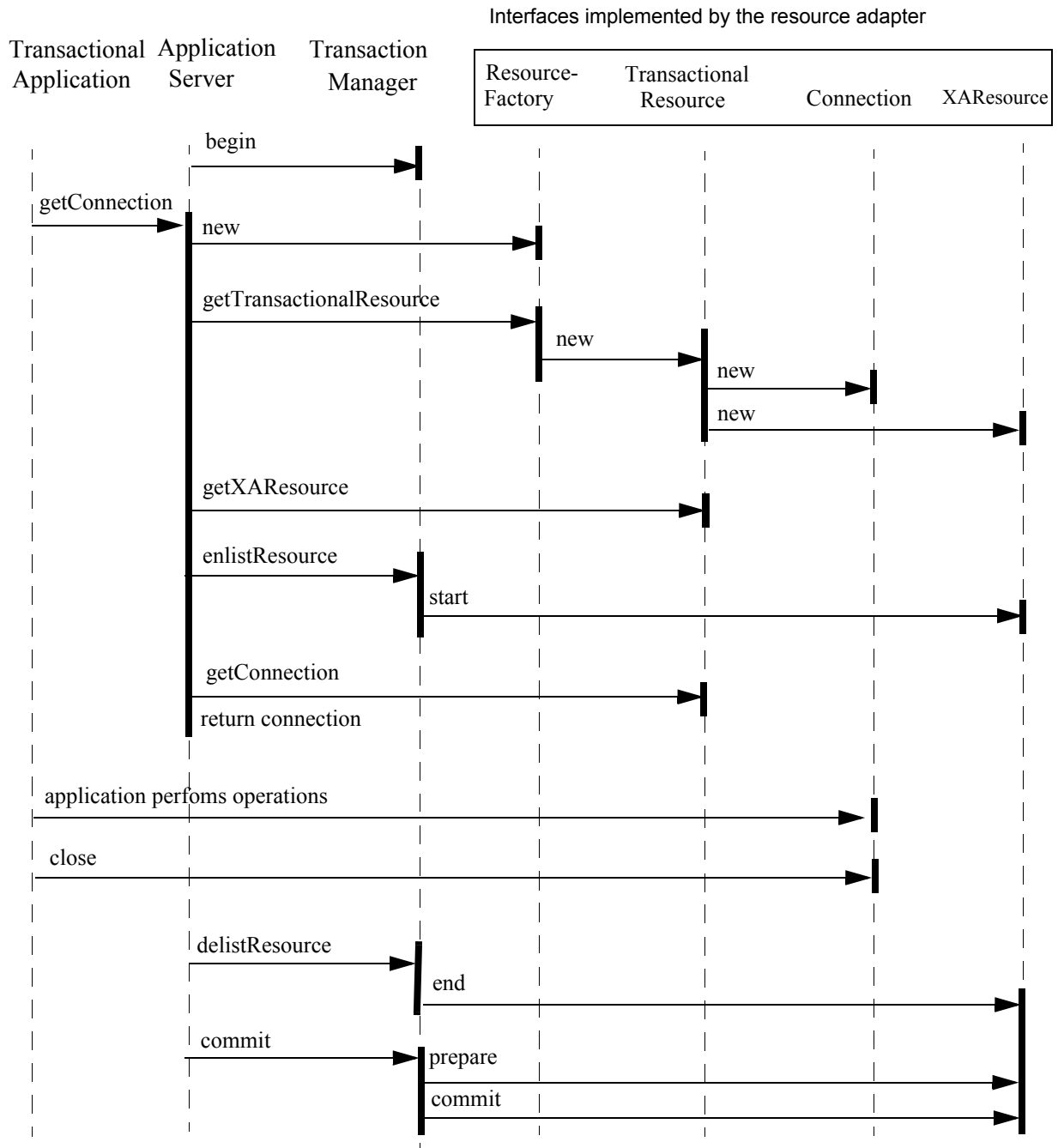
## 4.2 Transaction Association and Connection Request Flow

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 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.getTransactionResource` 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.





## 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
public interface javax.transaction.TransactionSynchronizationRegistry
```

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.cdi*:

Annotations:

```
public interface javax.transaction.cdi.Transactional
public interface javax.transaction.cdi.TransactionScoped
```

Classes:

```
public class javax.transaction.cdi.TransactionException
```

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;
}
```

### Constants

---

- **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 void beforeCompletion();
    public 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 void beforeCompletion()
```

The `beforeCompletion` method is called by the transaction manager prior to the start of the two-phase transaction commit process. This call is executed with the transaction context of the transaction that is being committed. An unchecked exception thrown by a registered `Synchronization` object causes the transaction to be aborted. That is, upon encountering an unchecked exception thrown by a registered synchronization object, the transaction manager must mark the transaction for rollback.

- **afterCompletion**

```
public void afterCompletion(int status)
```

The `afterCompletion` method is called by the transaction manager after the transaction is committed or rolled back.

**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 void commit();
    public boolean delistResource(XAResource xaRes, int flag);
    public boolean enlistResource(XAResource xaRes);
    public int getStatus();
    public void registerSynchronization(Synchronization sync);
    public void rollback();
    public void setRollbackOnly();
}
```

The Transaction interface allows operations to be performed against the transaction in the target Transaction 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 void commit() throws RollbackException,
    HeuristicMixedException, HeuristicRollbackException,
    IllegalStateException, 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 all 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 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 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 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 transaction manager invokes the `beforeCompletion` method prior to starting the two-phase 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 void rollback()
    throws IllegalStateException, SystemException
```

Rollback the transaction associated with the target Transaction object.

**Throws:** `IllegalStateException`

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

**Throws:** `SystemException`

Thrown if the transaction manager encounters an unexpected error condition.

- **setRollbackOnly**

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

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

**Throws:** `IllegalStateException`

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

**Throws:** `SystemException`

Thrown if the transaction manager encounters an unexpected error condition.

## Constants

---

- **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 void begin();
    public void commit();
    public int getStatus();
    public Transaction getTransaction();
    public void resume(Transaction tobj);
    public void rollback();
    public void setRollbackOnly();
    public void setTransactionTimeout(int seconds);
    public 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 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 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 all 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 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 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 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 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 void setTransactionTimeout(int seconds)
           throws SystemException
```

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

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. If the value is negative a `SystemException` is thrown.

**Throws:** `SystemException`

Thrown if the transaction manager encounters an unexpected error condition.

- **suspend**

```
public 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 void begin();
    public void commit();
    public int getStatus();
    public void rollback();
    public void setRollbackOnly();
    public void setTransactionTimeout(int seconds);
}
```

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

### Methods

---

- **begin**

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

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

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. If the value is negative a `SystemException` is thrown.

**Throws:** `SystemException`

Thrown if the transaction manager encounters an unexpected error condition.



## Interface TransactionSynchronizationRegistry

---

```
public interface javax.transaction.TransactionSynchronizationRegistry
{
    public Object getTransactionKey();
    public void putResource(Object key, Object value);
    public Object getResource(Object key);
    public void registerInterposedSynchronization(Synchronization sync);
    public int getTransactionStatus();
    public void setRollbackOnly();
    public boolean getRollbackOnly();
}
```

This interface is intended for use by system level application server components such as persistence managers, resource adapters, as well as EJB and Web application components. This provides the ability to register synchronization objects with special ordering semantics, associate resource objects with the current transaction, get the transaction context of the current transaction, get current transaction status, and mark the current transaction for rollback.

This interface is implemented by the application server as a stateless service object. The same object can be used by any number of components with complete thread safety. In standard application server environments, an instance implementing this interface can be looked up via JNDI using a standard name. The standard name is `java:comp/TransactionSynchronizationRegistry`.

### Methods

---

- **getTransactionKey**

```
public Object getTransactionKey()
```

**Returns:**

An opaque object that represents the transaction bound to the current thread at the time this method is called, or null is returned if a transaction is not associated with the current thread.

The returned object overrides *hashCode* and *equals* methods, to allow its use as the key in a *java.util.HashMap* for use by the caller. The returned object will return the same *hashCode* and compare equal to all other objects returned by calling this method from any component executing in the same transaction context in the same application server.

The *toString* method returns a String that might be usable by a human reader to usefully understand the transaction context. The result of the *toString* method is otherwise not defined. Specifically, there is no forward or backward compatibility guarantee for the result returned by the *toString* method.

The object is not necessarily serializable, and is not useful outside the virtual machine from which it was obtained.

- **putResource**

```
public void putResource(Object key, Object value)
```

Add or replace an object in the map of resources being managed for the transaction bound to the current thread at the time this method is called. The supplied key must be of a caller-defined class so as not to conflict with other users. The class of the key must guarantee that the *hashCode* and *equals* methods are suitable for keys in a map. The key and value are not examined or used by the implementation. The general contract

of this method is that of `java.util.Map#put(Object, Object)` for a Map that supports non-null keys and null values. For example, if there is already an value associated with the key, it is replaced by the value parameter.

**Parameters:**

`key`  
The key for the Map entry.

`value`  
The value for the Map Entry.

**Throws:** `IllegalStateException`

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

**Throws:** `NullPointerException`

Thrown if the parameter key is null.

- **`getResource`**

```
public Object getResource(Object key)
```

Get an object from the Map of resources being managed for the transaction bound to the current thread at the time this method is called. The key should have been supplied earlier by a call to `putResource` in the same transaction. If the key cannot be found in the current resource Map, null is returned. The general contract of this method is that of `java.util.Map#get(Object)` for a Map that supports non-null keys and null values. For example, the returned value is null if there is no entry for the parameter key or if the value associated with the key is actually null.

**Parameters:**

`key`  
The key for looking up the associated value object.

**Returns:**

The value object associated with the key, or null if not found.

**Throws:** `IllegalStateException`

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

**Throws:** `NullPointerException`

Thrown if the parameter key is null.

- **`registerInterposedSynchronization`**

```
public void registerInterposedSynchronization(Synchronization sync)
```

Register a Synchronization instance with special ordering semantics. Its `beforeCompletion` will be called after all `SessionSynchronization` `beforeCompletion` callbacks and callbacks registered directly with the Transaction, but before the 2-phase commit process starts. Similarly, the `afterCompletion` callback will be called after 2-phase commit completes but before any `SessionSynchronization` and Transaction `afterCompletion` callbacks.

The `beforeCompletion` callback will be invoked in the transaction context of the transaction bound to the current thread at the time this method is called. Allowable methods include access to resources, e.g. Connectors. No access is allowed to "user components" (e.g. timer services or bean methods), as these might change the state of data being managed by the caller, and might change the state of data that has already been

flushed by another caller of `registerInterposedSynchronization`. The general context is the component context of the caller of `registerInterposedSynchronization`.

The `afterCompletion` callback will be invoked in an undefined context. No access is permitted to "user components" as defined above. Resources can be closed but no transactional work can be performed with them.

If this method is invoked without an active transaction context, an `IllegalStateException` is thrown.

If this method is invoked after the two-phase commit processing has started, an `IllegalStateException` is thrown.

**Parameters:**

`sync`

The synchronization instance.

**Throws:** `IllegalStateException`

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

- **`getStatus`**

```
public int getTransactionStatus()
```

Return the status of the transaction bound to the current thread at the time this method is called.

This is the result of executing `TransactionManager.getStatus()` in the context of the transaction bound to the current thread at the time this method is called.

**Returns:**

The status of the transaction bound to the current thread at the time this method is called.

- **`setRollbackOnly`**

```
public void setRollbackOnly()
```

Set the *rollbackOnly* status of the transaction bound to the current thread at the time this method is called.

**Throws:** `IllegalStateException`

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

- **`getRollbackOnly`**

```
public boolean getRollbackOnly()  
                throws IllegalStateException
```

Get the `rollbackOnly` status of the transaction bound to the current thread at the time this method is called.

**Returns:**

The `rollbackOnly` status.

**Throws:** `IllegalStateException`

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

## @Interface Transactional

---

```

@Inherited
@InterceptorBinding
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Transactional {
    TxType value() default TxType.REQUIRED;

    public enum TxType {REQUIRED, REQUIRES_NEW, MANDATORY, SUPPORTS, NOT_
        SUPPORTED, NEVER}

    @Nonbinding
    public Class[] rollbackOn() default {};

    @Nonbinding
    public Class[] dontRollbackOn() default {};
}

```

Annotation used by applications to control transaction boundaries on CDI managed beans, as well as classes defined as managed beans by the Java EE specification such as servlets, JAX-RS resource classes, and JAX-WS service endpoints, declaratively. This provides the semantics of EJB transaction attributes in CDI without dependencies such as RMI. This support is implemented via an implementation of a CDI interceptor that conducts the necessary suspending, resuming, etc. The TxType element of the annotation provides the semantic equivalent of the transaction attributes in EJB.

By default checked exceptions do not result in the transactional interceptor marking the transaction for rollback and instances of RuntimeException and its subclasses do. This default behavior can be overridden by specifying which exceptions result in the interceptor marking the transaction for rollback. The rollbackOn element can be set to indicate which exceptions should cause the interceptor to mark the transaction for rollback. Conversely, the dontRollbackOn element can be set to indicate which exceptions should not cause the interceptor to mark the transaction for rollback. When a class is specified for either of these elements, the designated behavior applies to subclasses of that class as well. If both elements are specified, dontRollbackOn takes precedence.

EJB application exceptions (i.e., runtime exceptions annotated with @ApplicationException) are treated just as anyother runtime exceptions unless otherwise specified.

When Transactional annotated managed beans are used in conjunction with EJB container managed transactions the EJB container behavior is applied before the bean is called. When the bean is called the CDI behavior is applied before calling the bean's methods. It is best practice to avoid such use of Transactional annotations in conjunction with EJB container managed transactions in order to avoid possible confusion.

## Elements

---

- **value**

```
public abstract Transactional.TxType value  
Default:  
javax.transaction.cdi.Transactional.TxType.REQUIRED
```

Indicates whether a bean method is to be executed within a transaction context where the values provide the following corresponding behavior.

- **rollbackOn**

```
public abstract java.lang.Class[] rollbackOn  
Default:  
{}
```

Element used to indicate which exceptions should cause the interceptor to mark the transaction for rollback.

- **dontRollbackOn**

```
public abstract java.lang.Class[] dontRollbackOn  
Default:  
{}
```

Element used to indicate which exceptions should not cause the interceptor to mark the transaction for rollback

## Enum Transactional.TxType

---

```
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@NormalScope(passivating=true)
public @interface TransactionScoped {
}
```

Enum of Transactional annotation that indicates whether a bean method is to be executed within a transaction context where the values provide the following corresponding behavior.

### Enum Constants

---

- **REQUIRED**

```
public static final Transactional.TxType REQUIRED
```

If called outside a transaction context, a new JTA transaction will begin, the managed bean method execution will then continue inside this transaction context, and the transaction will be committed.

If called inside a transaction context, the managed bean method execution will then continue inside this transaction context.

- **REQUIRES\_NEW**

```
public static final Transactional.TxType REQUIRES_NEW
```

If called outside a transaction context, a new JTA transaction will begin, the managed bean method execution will then continue inside this transaction context, and the transaction will be committed.

If called inside a transaction context, the current transaction context will be suspended, a new JTA transaction will begin, the managed bean method execution will then continue inside this transaction context, the transaction will be committed, and the previously suspended transaction will be resumed.

- **MANDATORY**

```
public static final Transactional.TxType MANDATORY
```

If called inside a transaction context, a `TransactionException` with a nested `TransactionRequiredException` will be thrown.

If called inside a transaction context, managed bean method execution will then continue under that context.

- **SUPPORTS**

```
public static final Transactional.TxType SUPPORTS
```

If called outside a transaction context, managed bean method execution will then continue outside a transaction context.

If called inside a transaction context, the managed bean method execution will then

continue inside this transaction context.

- **NOT\_SUPPORTED**

```
public static final Transactional.TxType NEVER
```

If called outside a transaction context, managed bean method execution will then continue outside a transaction context.

If called inside a transaction context, the current transaction context will be suspended, the managed bean method execution will then continue outside a transaction context, and the previously suspended transaction will be resumed.

- **NEVER**

```
public static final Transactional.TxType NEVER
```

If called outside a transaction context, managed bean method execution will then continue outside a transaction context.

If called inside a transaction context, a `TransactionException` with a nested `InvalidException` will be thrown

## Methods

---

- **values**

```
public static Transactional.TxType[] values()
```

Returns an array containing the constants of this enum type, in the order they are declared.

- **valueOf**

```
public static Transactional.TxType valueOf(java.lang.String name)
```

Returns the enum constant of this type with the specified name.

## **@Interface TransactionScoped**

---

```
@Retention(value=RUNTIME)
@Target(value={TYPE,METHOD,FIELD})
@NormalScope(passivating=true)
public @interface TransactionScoped
```

Annotation used to indicate a bean is to be scoped to the current active JTA transaction.

The transaction scope is active when the return from a call to `UserTransaction.getStatus` or `TransactionManager.getStatus` is one of the following states:

```
Status.STATUS_ACTIVE
Status.STATUS_MARKED_ROLLBACK
Status.STATUS_PREPARED
Status.STATUS_UNKNOWN
Status.STATUS_PREPARING
Status.STATUS_COMMITTING
Status.STATUS_ROLLING_BACK
```

It is not intended that the term "active" as defined here in relation to `TransactionScoped` should also apply to its use in relation to transaction context, lifecycle, etc. mentioned elsewhere in this specification.

The transaction context must be destroyed after any `Synchronization.beforeCompletion` methods are called and after completion calls have been made on enlisted resources. `Synchronization.afterCompletion` calls may occur before the transaction context is destroyed, however, there is no guarantee.

A `javax.enterprise.context.ContextNotActiveException` must be thrown if an object with this annotation is used when the transaction context is not active.

The object with this annotation is associated with the JTA transaction where it is first used and this association is retained through any transaction suspend or resume calls as well as any `beforeCompletion` `Synchronization` calls until the transaction is completed.

The way in which the JTA transaction is begun and completed (eg BMT, CMT, etc.) is of no consequence.

The contextual references used across different JTA transactions are distinct.



## Interface XAResource

---

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

`void 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 parameter *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**

`void 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 suspended 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\_RMFAIL, 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 past 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. Possible `XAExceptions` are `XA_HEURHAZ`, `XA_HEURCOM`, `XA_HEURRB`, `XA_HEURMIX`, `XAER_RMERR`, `XAER_RMFAIL`, `XAER_NOTA`, `XAER_INVAL`, or `XAER_PROTO`.

Upon return, the resource manager has rolled back the branch's work and has released all held resources.

- **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 timeout 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 existing 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_INVALID`, or `XAER_PROTO`.

## Constants

---

- **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.

- **TMNOFLAGS**

```
public final static int TMNOFLAGS = 0x00000000
```

Use `TMNOFLAGS` 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.

- **XA\_OK**

```
public final static int XA_OK = 0
```

The transaction work has been prepared normally.

- **XA\_RDONLY**

```
public final static int XA_RDONLY = 0x00000003
```

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

## 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`.

### Constants

---

- **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[] getGlobalTransactionId()
```

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

**Returns:**

A byte array containing the global transaction identifier.

- **getBranchQualifier**

```
byte[] getBranchQualifier()
```

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 RollbackException(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 TransactionalException

---

```
public class javax.transaction.cdi.TransactionalException extends java.lang.RuntimeException
{
    public TransactionalException(String s, Throwable throwable);
}
```

**RuntimeException** used to rethrow Exceptions thrown from Transactional annotation interceptors implementation.

This results in the transaction being marked for rollback by default.

The original Exception is set as the nested exception and includes **TransactionRequiredException** for the case where a bean with **TxType.MANDATORY** is called outside a transaction context, **InvalidTransactionException** for the case where a bean with **TxType.NEVER** is called inside a transaction context, and various commit-time exceptions such as **RollbackException**, **HeuristicMixedException**, **HeuristicRollbackException**, **SecurityException**, **IllegalStateException**, and **SystemException**.

### Constructors

---

- **TransactionalException**

```
public TransactionalException(String s, Throwable throwable)
```

Create an **TransactionalException** with a given string and nested **Throwable**.

## Class XAException

---

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

The XAException is thrown 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 XAException(String s)`  
 Create an XAException with the specified string.
- **XAException**  
`public XAException(int errCode)`  
 Create an XAException with the specified error code.

### Variables and Constants

---

- **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 resource 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_RBTIMEOUT = 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).
- [3] OMG Object Transaction Service (OTS 1.1)
- [4] ORB Portability Submission, OMG document orbos/97-04-14.
- [5] Enterprise JavaBeans™.
- [6] JDBC™ 4.1.
- [7] JMS 2.0.

## 7 Change History for Version 1.0.1B

- Removed the method modifier `abstract` from all interface methods, since interface methods are implicitly abstract.
- Table 1, row 1 (TMJOIN) : move transaction association (T1) from column 3 (association suspended) to column 1 (not associated).
- Interface `javax.transaction.Synchronization`, method `beforeCompletion`, change the following phrase in the description "start of the transaction completion process" to "start of the two-phase transaction commit process".
- Interface `javax.transaction.Transaction`, method `commit`, added `IllegalStateException` to throws clause.
- Interface `javax.transaction.Transaction`, method `commit`, replace the description of `HeuristicRollbackException` with "Thrown to indicate that a heuristic decision was made and that all relevant updates have been rolled back."
- Interface `javax.transaction.Transaction`, change spelling of `Transactioin` to `Transaction` in interface description.
- Interface `javax.transaction.Transaction`, method `registerSynchronization`, first paragraph, line 2, change the phrase "transaction completion process" to "two-phase transaction commit process".
- Interface `javax.transaction.Transaction`, method `rollback`, spelling correction to method signature description, change `SytemException` to `SystemException`.
- Interface `javax.transaction.TransactionManager`, method `commit`, replace the description of `HeuristicRollbackException` with "Thrown to indicate that a heuristic decision was made and that all relevant updates have been rolled back."
- Interface `javax.transaction.TransactionManager`, method `setTransactionTimeout`, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the `begin` method."
- Interface `javax.transaction.TransactionManager`, method `setTransactionTimeout`, replace the description of method parameter `seconds` with "The value of the timeout in seconds. If the value is zero, the transaction service restores the default value. If the value is negative a `SystemException` is thrown."
- Interface `javax.transaction.UserTransaction`, method `commit`, replace the description of `HeuristicRollbackException` with "Thrown to indicate that a heuristic decision was made and that all relevant updates have been rolled back."

- Interface `javax.transaction.UserTransaction`, method `setTransactionTimeout`, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the `begin` method."
- Interface `javax.transaction.UserTransaction`, method `setTransactionTimeout`, replace the description of method parameter `seconds` with "The value of the timeout in seconds. If the value is zero, the transaction service restores the default value. If the value is negative a `SystemException` is thrown."
- Interface `javax.transaction.xa.XAResource`, method `commit`, insert return type `void` to method signature description.
- Interface `javax.transaction.xa.XAResource`, method `commit`, spelling correction to description, change `paramether` to `parameter`.
- Interface `javax.transaction.xa.XAResource`, method `end`, replace return type `int` with `void` in method signature description.
- Interface `javax.transaction.xa.XAResource`, method `end`, corrected spelling of `XAException` errorCode `XAER_RMFAILED` to `XAER_RMFAIL`.
- Interface `javax.transaction.xa.XAResource`, method `recover`, spelling correction to method signature description, replace return type `xid []` with `Xid []`.
- Interface `javax.transaction.xa.XAResource`, method `rollback`, add the following to the description of `XAException`, "Possible `XAException`s are `XA_HEURHAZ`, `XA_HEURCOM`, `XA_HEURRB`, `XA_HEURMIX`, `XAER_RMERR`, `XAER_RMFAIL`, `XAER_NOTA`, `XAER_INVALID`, or `XAER_PROTO`. Upon return, the resource manager has rolled back the branch's work and has released all held resources."
- Interface `javax.transaction.xa.XAResource`, spelling correction to description, replace `TMNOFLAG` with `TMNOFLAGS`.
- Interface `javax.transaction.xa.XAResource`, added constants `XA_OK` and `XA_RDONLY` to be consistent with the actual interface definition.
- Interface `javax.transaction.xa.Xid`, method `getGlobalTransactionId`, spelling correction to method signature description, corrected method name from `getGrid` to `getGlobalTransactionId`.
- Interface `javax.transaction.xa.Xid`, method `getBranchQualifier`, spelling correction to method signature description, corrected method name from `getEqual` to `getBranchQualifier`.

- Class `javax.transaction.xa.XAException`, spelling correction to description of interface definition, replace phrase `javax.transaction.xa.XAException` with `javax.transaction.xa.XAException`.



## 8 Change History for Version 1.1

- Section 3.4 XAResource Interface: The line "The transaction manager obtains an XAResource for each resource manager participating in a global transaction." has been changed to "The transaction manager obtains an XAResource for each transaction resource participating in a global transaction."
- Interface javax.transaction.UserTransaction, method setTransactionTimeout, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the begin method by the current thread."
- Interface javax.transaction.TransactionManager, method setTransactionTimeout, replace the first paragraph of the description with "Modify the timeout value that is associated with transactions started by subsequent invocations of the begin method by the current thread."
- New interface javax.transaction.TransactionSynchronizationRegistry.
- Interface javax.transaction.Synchronization, method beforeCompletion, add the following description: "An unchecked exception thrown by a registered Synchronization object causes the transaction to be aborted. That is, upon encountering an unchecked exception thrown by a registered synchronization object, the transaction manager must mark the transaction for rollback."

## 9 Change History for Version 1.2

- New annotation `javax.transaction.cdi.Transactional` and exception `javax.transaction.cdi.TransactionException`
- New annotation `javax.transaction.cdi.TransactionScoped`
- Added the following description to the end of section 3.3.1 Resource Enlistment: "A container only needs to call `delistResource` to explicitly dissociate a resource from a transaction and it is not a mandatory container requirement to do so as a precondition to transaction completion. A transaction manager is, however, required to implicitly insure the association of any associated `XAResource` is ended, via the appropriate `XAResource.end` call, immediately prior to completion; that is before `prepare` (or `commit/rollback` in the onephase-optimized case)."