APPENDIX

E

EJB Quick Reference Guide

This appendix is a quick reference for programmers to use during EJB development. In the first section, you'll find Figures E.1 through E.17, illustrating what's really going on in an EJB system. Some of these are taken directly from the EJB specification, and some have been created by us; we have condensed the diagrams and commented on them to clarify their meaning. You'll also find summaries and explanations of each method in the EJB API, as well as a transaction reference.

NOTE

Even though session beans and message-driven beans as defined in EJB 2.1 and earlier EJB specifications are supported in EJB 3.0 containers, we will not provide reference to APIs relevant to these. Also, as with the rest of the book, we do not cover the EJB 2.1–defined entity beans in this appendix even though they are part of the EJB 3.0 specification. This appendix focuses solely on EJB 3.0–defined POJO session beans and message driven beans, and Java Persistence API entities. You can refer to the previous editions of this book to get information on session beans, message-driven beans, and entity beans as defined in EJB 2.x and EJB 1.1 specifications.

Figure E.1 The client's view of a session bean object life cycle.

Stateless Session Bean Diagrams

Figure E.2 The life cycle of a stateless session bean. Each method call shown is an invocation from the container to the bean instance.

Figure E.3 Sequence diagram depicting two phases of the life cycle of stateless session bean: addition of a bean instance to the stateless session bean pool and retrieval of a stateless session bean's business interface reference by the client.

Figure E.4 Sequence diagram depicting the other two phases of the life cycle of stateless session bean: servicing a business method and removing a bean instance from the stateless session bean pool.

Figure E.5 The Web services client view of a stateless session bean. The client can be a Java or non-Java client.

Stateful Session Bean Diagrams

Figure E.6 The life cycle of a stateful session bean (does not implement `javax.ejb.SessionSynchronization`). Each method call shown is an invocation from the container to the bean instance.

Figure E.7 The life cycle of a stateful session bean (implements `javax.ejb.SessionSynchronization`). Each method call shown is an invocation from the container to the bean instance.

Figure E.8 Sequence diagram depicting the two phases of the life cycle of a stateful session bean: retrieval of a stateful session bean's business interface reference by the client, and servicing a business method.

Figure E.9 Sequence diagram depicting the other two phases of the life cycle of a stateful session bean: passivation and activation.

Figure E.10 Sequence diagram depicting the execution of a transactional method on a stateful session bean when the `javax.ejb.SessionSynchronization` interface is implemented, as well as the removal of the bean instance.

Message Driven Bean Diagrams

Figure E.11 The life cycle of a message driven bean. Each method call shown is an invocation from the container to the bean instance.

Figure E.12 Sequence diagram for message driven beans. For simplicity, the Container object represents all container subsystems, including home objects, transaction services, and so on.

Java Persistence API Diagrams

Figure E.13 The life cycle of a Java Persistence API defined entity.

NOTE

Given that more often than not container-managed transaction-scoped entity managers will be used in EJB applications, we have decided, for the sake of simplicity, to focus on the life cycle of Java Persistence API entities that use container-managed transaction-scoped entity managers.

Figure E.14 Sequence diagram depicting the deployment of a persistence unit, as well as retrieval of an instance of a container-managed transaction-scoped entity manager.

Figure E.15 Sequence diagram depicting the two phases of the life cycle of a Java Persistence API entity: persisting a new entity, and synchronizing the modified state of a managed entity to the database.

Figure E.16 Sequence diagram for the other two phases of life cycle of a Java Persistence API entity: detaching an entity instance, and merging an entity instance.

Figure E.17 Sequence diagram for the final phase of life cycle of a Java Persistence API entity: removing the entity.

NOTE

Once again, note that the preceding entity sequence diagrams take into consideration only the scenarios pertaining to a container-managed transaction-scoped entity manager.

EJB API Reference

The following sections explain the Enterprise JavaBeans API used in EJB 3.0–style POJO bean development. These APIs are defined in the `javax.ejb` package. Evidently, the APIs involved in development of EJB 3.0 POJO beans are simpler and fewer as compared to those of the earlier versions. Since we have provided a reference to all the standard annotations defined in the EJB specification in Appendix B, this Appendix covers only the interface and class APIs.

EJBContext

An `EJBContext` object is a container-implemented object. Your bean can use an EJB context to enquire about its environment from the container. For instance, the bean can invoke methods on an `EJBContext` object to determine its current transactional status, security status, and more. Each container, therefore, must make an `EJBContext` object available to your enterprise bean at runtime. This interface is extended by the `SessionContext`, `EntityContext`, and `MessageDrivenContext` interfaces to provide additional functionality specific to those bean types.

The EJB 3.0 defined session and message driven beans can get access to `EJBContext` instance through a resource injection mechanism. Source E.1 provides the definition of this interface, and Table E.1 gives a brief explanation of the methods of this interface.

```
public interface javax.ejb.EJBContext
{
     public javax.ejb.EJBHome getEJBHome();

     public javax.ejb.EJBLocalHome getEJBLocalHome();

     public java.util.Properties getEnvironment();

     public java.security.Identity getCallerIdentity();

     public java.security.Principal getCallerPrincipal();

     public boolean isCallerInRole(java.security.Identity role);

     public boolean isCallerInRole(String roleName);

     public javax.transaction.UserTransaction getUserTransaction()
          throws java.lang.IllegalStateException;

     public void setRollbackOnly()
          throws java.lang.IllegalStateException;

     public boolean getRollbackOnly()
          throws java.lang.IllegalStateException;

     public javax.ejb.TimerService getTimerService()
          throws java.lang.IllegalStateException;

     public java.lang.Object lookup(String name);
}
```

Source E.1 The javax.ejb.EJBContext interface.

Table E.1 javax.ejb.EJBContext

METHOD	EXPLANATION
`getEJBHome()`	This method returns the bean's remote home interface object. Since EJB 3.0–style POJO beans do not have a home interface, this method is applicable to enterprise beans that use the previous EJB programming model.
`getEJBLocalHome()`	Same as `getEJBHome()` except this retrieves the local interface version.

`getEnvironment()`	This is a deprecated method, and its use is heavily discouraged. In order to gain access to the bean's environment, use the JNDI naming context `java:comp/env` instead.
`getCallerIdentity()`	This is a deprecated method, and its use is heavily discouraged. Use `getCallerPrincipal()` method instead.
`getCallerPrincipal()`	Retrieves the current logged-in user's security principal. You can use this principal to query a database or perform other operations.
`isCallerInRole(Identity)`	This is a deprecated method, and its use is heavily discouraged. Use `isCallerInRole(String)` instead.
`isCallerInRole(String)`	Asks the container if the current logged-in user is in a particular security role. Useful for programmatic security.
`getUserTransaction()`	Retrieves the JTA (Java Transaction API) `UserTransaction` interface to perform programmatic transactions. Only enterprise beans with bean-managed transaction demarcation can use this method.
`setRollbackOnly()`	If something goes horribly wrong inside your bean, you can call this method to force the current transaction to roll back. Only enterprise beans with container-managed transaction demarcation can use this method.
getRollbackOnly()	Asks the container if the transaction is doomed to rollback. If it's doomed, you can avoid performing computer-intensive operations.
`getTimerService()`	Returns `TimerService` object to access the EJB timer service from the enterprise bean.

SessionContext

A session context is a specific EJB context used only for session beans. Source E.2 provides the definition of this interface, and Table E.2 gives a brief explanation of the methods of this interface.

```
public interface javax.ejb.SessionContext
     extends javax.ejb.EJBContext
{
     public javax.ejb.EJBLocalObject getEJBLocalObject()
          throws java.lang.IllegalStateException;

     public javax.ejb.EJBObject getEJBObject()
          throws java.lang.IllegalStateException;

     public javax.xml.rpc.handler.MessageContext getMessageContext()
          throws java.lang.IllegalStateException;

     public <T> T getBusinessObject(Class<T> businessInterface)
          throws java.lang.IllegalStateException;

     public java.lang.Class getInvokedBusinessInterface()
          throws java.lang.IllegalStateException;
}
```

Source E.2 The javax.ejb.SessionContext interface.

Table E.2 javax.ejb.SessionContext

METHOD	EXPLANATION
`getEJBLocalObject()`	Returns a reference to the EJB local object associated with this context instance. This method is useful if your bean wants to call another local bean and while doing so, wants to pass a reference to itself. Since EJB 3.0–style POJO beans do not have an object interface, this method is applicable to enterprise beans that use the previous EJB programming model.
`getEJBObject()`	Same as `getEJBLocalObject()` except that this retrieves the remote object interface version.
`getMessageContext()`	Returns the `MessageContext` object associated with a stateless session bean instance that implements a JAX-RPC Web service endpoint. This `MessageContext` object can then be used to retrieve information about the Web service (SOAP) message via its properties. Only stateless session beans with a Web service endpoint can use this method.

`getBusinessObject(Class <T>)`	This method returns the session bean's business interface object, local or remote, as specified in the argument. Only EJB 3.0–style POJO session beans can call this method.
`getInvokedBusinessInterface()`	This method returns the session bean business interface through which the bean was invoked. You should use this method if you have supported both remote and local business interfaces, and want to check through which interface the invocation on the session bean was made, during runtime. This method is disallowed if the session bean does not define an EJB 3.0 business interface or was not invoked through a business interface.

MessageDrivenContext

A message-driven context is a specific EJB context used only for message-driven beans. This interface serves as a marker interface. There are no specific additional methods that message-driven beans have on their context objects.

SessionSynchronization

If your stateful session bean is caching database data in memory or needs to roll back in-memory conversational state upon a transaction abort, you should implement this interface. The container will call each of the methods in this interface automatically at the appropriate times during transactions, alerting you to important transactional events. Each of these methods can throw a `java.rmi.RemoteException` or `javax.ejb.EJBException`. Source E.3 provides the definition of this interface and Table E.3 gives a brief explanation of the methods of this interface.

```
public interface javax.ejb.SessionSynchronization
{
     public void afterBegin()
          throws javax.ejb.EJBException, java.rmi.RemoteException;

     public void beforeCompletion()
          throws javax.ejb.EJBException, java.rmi.RemoteException;

     public void afterCompletion(boolean committed)
          throws javax.ejb.EJBException, java.rmi.RemoteException;
}
```

Source E.3 The javax.ejb.SessionSynchronization interface.

Table E.3 javax.ejb.SessionSynchronization

METHOD	DESCRIPTION
`afterBegin()`	Called by the container directly after a transaction begins. You should read in any database data you want to cache in your stateful session bean during the transaction. You should also create a backup copy of your state in case the transaction rolls back.
`beforeCompletion()`	Called by the container right before a transaction completes. Write out any database data you've cached during the transaction.
`afterCompletion(boolean)`	Called by the container when a transaction completes either in a commit *or* an abort. True indicates a successful commit; false indicates an abort. If an abort happened, revert to the backup copy of your state to preserve your session bean's conversation.

TimedObject

If your session or message-driven beans want to implement timer expiration notification methods so that containers can call back `ejbTimeout()` on `javax.ejb.TimedObject` after a certain period has elapsed, you should implement this interface. Source E.4 provides the definition of this interface, and Table E.4 gives a brief explanation of the method of this interface.

```
public interface javax.ejb.TimedObject
{
     public void ejbTimeout (javax.ejb.Timer timer);
}
```

Source E.4 The javax.ejb.TimedObject interface.

Table E.4 javax.ejb.TimedObject

METHOD	DESCRIPTION
`ejbTimeout(Timer)`	Called by container upon timer expiration. You should implement the bean logic that you want to execute periodically in this method.

Timer

This interface provides information about the timer created with the help of EJB timer service, such as the next point when the timer expiration is scheduled to occur, the number of milliseconds that will elapse before the next scheduled timer expiration occurs, and much more. An instance of `javax.ejb.Timer` is initialized by the container and passed to your bean as an argument when it calls `ejbTimeout()` on the `javax.ejb.TimedObject` interface. Source E.5 provides the definition of this interface, and Table E.5 gives a brief explanation of the methods of this interface.

```
public interface javax.ejb.Timer
{
     public void cancel()
          throws java.lang.IllegalStateException,
          javax.ejb.NoSuchObjectLocalException,
          javax.ejb.EJBException;

     public long getTimeRemaining()
          throws java.lang.IllegalStateException,
          javax.ejb.NoSuchObjectLocalException,
          javax.ejb.EJBException;

     public java.util.Date getNextTimeout()
          throws java.lang.IllegalStateException,
          javax.ejb.NoSuchObjectLocalException,
          javax.ejb.EJBException;

     public java.io.Serializable getInfo()
          throws java.lang.IllegalStateException,
          javax.ejb.NoSuchObjectLocalException,
          javax.ejb.EJBException;

     public javax.ejb.TimerHandle getHandle()
          throws java.lang.IllegalStateException,
          javax.ejb.NoSuchObjectLocalException,
          javax.ejb.EJBException;
}
```

Source E.5 The javax.ejb.Timer interface.

Table E.5 javax.ejb.Timer

METHOD	DESCRIPTION
`cancel()`	Call this method if you want to cancel all the expiration notifications associated with the given timer instance.
`getTimeRemaining()`	Returns the number of milliseconds that will elapse before the next scheduled timer expiration.
`getNextTimeout()`	Returns the future point at which the next timer expiration is scheduled to occur.
`getInfo()`	Returns the information associated with the given timer at the time of its creation. If no `Serializable` information object was provided at the time of creation, then this method returns null.
`getHandle()`	Returns the `Serializable` timer handle that can be persisted by your bean for later reuse.

TimerHandle

A timer handle is a persistent reference to an EJB timer object. All EJB timers implement this interface. Timer handles allow your beans to persist the timer object for later reuse so that the bean does not have to create a new timer object. Source E.6 provides the definition of this interface, and Table E.6 gives a brief explanation of the methods of this interface.

```
public interface javax.ejb.TimerHandle
     extends java.io.Serializable
{
     public javax.ejb.Timer getTimer()
          throws java.lang.IllegalStateException,
          javax.ejb.NoSuchObjectLocalException,
          javax.ejb.EJBException;
}
```

Source E.6 The javax.ejb.TimerHandle interface.

Table E.6 javax.ejb.TimerHandle

METHOD	DESCRIPTION
`getTimer()`	Returns the reference to the timer object represented by the given handle. Use this method to retrieve the timer object reference from a persisted timer object.

TimerService

This interface provides your enterprise beans with access to the EJB timer service implemented by the EJB container. Source E.7 provides the definition of this interface, and Table E.7 gives a brief explanation of the methods of this interface.

```
public interface javax.ejb.TimerService
{
     public javax.ejb.Timer createTimer (long duration,
          java.io.Serializable info)
          throws java.lang.IllegalArgumentException,
         java.lang.IllegalStateException,
         javax.ejb.EJBException;

     public javax.ejb.Timer createTimer (long initialDuration,
          long intervalDuration, java.io.Serializable info)
          throws java.lang.IllegalArgumentException,
          java.lang.IllegalStateException,
          javax.ejb.EJBException;

     public javax.ejb.Timer createTimer (java.util.Date expiration,
          java.io.Serializable info)
          throws java.lang.IllegalArgumentException,
          java.lang.IllegalStateException,
          javax.ejb.EJBException;

     public javax.ejb.Timer createTimer (java.util.Date
          initialExpiration, long intervalDuration,
          java.io.Serializable info)
          throws java.lang.IllegalArgumentException,
          java.lang.IllegalStateException,
          javax.ejb.EJBException;

     public java.util.Collection getTimers()
          throws java.lang.IllegalStateException,
          javax.ejb.EJBException;
}
```

Source E.7 The javax.ejb.TimerService interface.

Table E.7 javax.ejb.TimerService

METHOD	DESCRIPTION
`createTimer(long, Serializable)`	Creates a one-time expiration timer, which becomes inactive after the first (and last) expiration. The second argument represents a `Serializable` object containing application-specific information. If you do not have any information to associate with the given timer, then pass null as the second argument.
`createTimer(long, long, Serializable)`	Creates a recurrently expiring timer whose first expiration occurs after a given duration (in milliseconds) specified in the first argument has elapsed, and subsequent expirations occur after the duration (in milliseconds) specified in the second argument elapses. Subsequent expirations are scheduled relative to the time of the first expiration. If expiration is delayed for some reason, two or more expiration notifications may occur in close succession.
`createTimer(Date, Serializable)`	Creates a one-time expiration timer that expires at a given time.
`createTimer(Date, long, Serializable)`	Creates a recurrently expiring timer whose first expiration occurs at the time specified by the first argument, and subsequent expirations occur after the duration (in milliseconds) specified in the second argument elapses. Subsequent expirations are scheduled relative to the time of the first expiration. If expiration is delayed for some reason, two or more expiration notifications may occur in close succession.
`getTimers()`	Retrieves a collection of all the timers associated with the given bean.

EJB Exception Reference

Table E.8 describes the purpose of each exception class relevant to the new EJB 3.0 programming model.

Table E.8 EJB Exception Explantations

EXCEPTION	DESCRIPTION
EJBException	Your enterprise bean class should throw this exception to indicate an unexpected error, such as a failure to open a database connection, or a JNDI exception. Your container treats this exception as a serious problem and may take action such as logging the event or paging a system administrator, depending upon your container's policy. For an EJB 3.0 bean, if the problem is encountered at the protocol level, then an `EJBException` instance wrapping `RemoteException` is thrown by the container. Since `EJBException` is a `RuntimeException`, it does not need to be declared in `throws` clauses.
`ConcurrentAccessException`	Concurrent calls from clients to a stateful session bean are disallowed. If a client invokes a business method on an instance of a stateful session bean when another method invocation from the same or a different client is already in progress, the container will throw this exception to the client of the subsequent invocation. Under some deployment scenarios, such as when the application server is clustered, concurrent requests to a stateful session bean instance might be queued. However, this behavior is not guaranteed, and hence, clients should not rely on this. The `javax.ejb.ConcurrentAccessException` is a subclass of `javax.ejb.EJBException`. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `java.rmi.RemoteException` instead.

`EJBAccessException`	This subclass of `javax.ejb.EJBException` is thrown to the client when it tries to access an EJB method for which it does not have security permission. A caller is allowed to invoke a method if, and only if, the method is specified as `PermitAll` or the caller is assigned at least one of the security roles that is included in the method's permission list. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `java.rmi.AccessException` instead.
`EJBTransactionRequiredException`	When a bean's method is marked with the `MANDATORY` transaction attribute, the container expects a client to invoke such a method within a client-established transaction context. In the event a client fails to do so, the container will throw this exception. The `javax.ejb.EJBTransactionRequiredException` is subclass of `javax.ejb.EJBException`. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `javax.transaction.TransactionRequiredException` instead.
`EJBTransactionRolledbackException`	A transaction in which the enterprise bean participates can be rolled back at any time. In such cases, any subsequent method calls on the bean will result in the container throwing this exception to the client. This is the container's way of informing the client that the transaction associated with the request has been rolled back or marked for a rollback, and that the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `javax.transaction.TransactionRolledbackException` instead.

`NoSuchEJBException`	If a client tries to call a method on a business interface reference of a stateful session bean that has been removed, the attempted invocation results in this runtime exception. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `java.rmi.NoSuchObjectException` instead.

Transaction Reference

The following section offers reference information on transactions in EJB as outlined in Tables E.9 through E.14.

Table E.9 The Effects of Transaction Attributes

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
`Required`	None T1	T2 T1
`RequiresNew`	None T1	T2 T2
`Supports`	None T1	None T1
`Mandatory`	None T1	Error T1
`NotSupported`	None T1	None None
`Never`	None T1	None Error

Table E.10 Transaction Attributes

CONSTANT	MEANING
`NotSupported`	Your bean *cannot* be involved in a transaction at all. When a bean method is called, any existing transaction is suspended.

`Never`	Your bean *cannot* be involved in a transaction at all. When a bean method is called, if a transaction is in progress, a `javax.ejb.EJBException` is thrown back to the client. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `java.rmi.RemoteException` instead of `javax.ejb.EJBException`.
`Required`	Your bean must *always* run in a transaction. If a transaction is already running, your bean joins that transaction. If no transaction is running, the EJB container starts one for you.
`RequiresNew`	Your bean must always run in a *new* transaction. Any current transaction is suspended.
`Supports`	If a transaction is already under way, your bean joins that transaction. Otherwise, the bean runs with no transaction at all.
`Mandatory`	Mandates that a transaction must be already running when your bean method is called, or a `javax.ejb.EJBTransactionRequiredException` is thrown back to the caller. Note that if the business interface is a remote business interface that extends `java.rmi.Remote`, the client will receive the `javax.transaction.TransactionRequiredException` instead of the `javax.ejb.EJBTransactionRequiredException`.

Table E.11 Permissible Transaction Attributes for Each Bean Type

TRANSACTION ATTRIBUTE	STATELESS SESSION BEAN	STATEFUL SESSION BEAN IMPLEMENTING SESSION SYNCHRONIZATION	MESSAGE-DRIVEN BEAN
`Required`	Yes	Yes	Yes
`RequiresNew`	Yes	Yes	No
`Mandatory`	Yes	Yes	No
`Supports`	Yes	No	No
`NotSupported`	Yes	No	Yes

Never	Yes	No	No

Table E.12 Transaction Isolation Levels

ISOLATION LEVEL	DIRTY READS?	UNREPEATABLE READS?	PHANTOM READS?
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Table E.13 The *javax.transaction.Status* Constants for Transactional Status

CONSTANT	MEANING
STATUS_ACTIVE	A transaction is currently happening and is active.
STATUS_NO_TRANSACTION	No transaction is currently happening.
STATUS_MARKED_ROLLBACK	The current transaction will eventually abort because it's been marked for rollback. This could be because some party called `setRollbackOnly()`.
STATUS_PREPARING	The current transaction is preparing to be committed (during Phase One of the two-phase commit protocol).
STATUS_PREPARED	The current transaction has been prepared to be committed (Phase One is complete).
STATUS_COMMITTING	The current transaction is in the process of being committed right now (during Phase Two).
STATUS_COMMITTED	The current transaction has been committed (Phase Two is complete).
STATUS_ROLLING_BACK	The current transaction is in the process of rolling back.

STATUS_ROLLEDBACK	The current transaction has been rolled back.
STATUS_UNKNOWN	The status of the current transaction cannot be determined.

Table E.14 The javax.transaction.UserTransaction Methods for Transactional Boundary Demarcation

METHOD	DESCRIPTION
begin()	Begin a new transaction. This transaction becomes associated with the current thread.
commit()	Run the two-phase commit protocol on an existing transaction associated with the current thread. Each resource manager will make its updates durable.
getStatus()	Retrieve the status of the transaction associated with this thread.
rollback()	Force a rollback of the transaction associated with the current thread.
setRollbackOnly()	Call this to force the current transaction to roll back. This will eventually force the transaction to abort. One interesting use of this is to test out what your components will do without having them perform any permanent resource updates.
setTransactionTimeout(int)	The transaction timeout is the maximum amount of time that a transaction can run before it's aborted. This is useful to avoid deadlock situations, when precious resources are being held by a transaction that is currently running.

The following sections explain the Java Persistence API used in development of POJO style entities. These APIs are defined in the `javax.persistence` package. Since Appendix B provids a reference to all the standard annotations defined in Java Persistence API specification, this Appendix covers only the interface and class APIs.

NOTE

Towards the very end, the EJB 3.0 Expert Group decided to rename EJB Query Language, or EJB-QL, as Java Persistence Query Language. By the time this change became known (around May 2006), this book was already in print. Since this Appendix was to be published electronically on the book's companion Web site, we were able to reflect this last-minute change here. Hence, while in the book you'll see us referring to EJB Query Language, or EJB-QL, know that we are talking about Java Persistence Query Language.

EntityManager

An `EntityManager` instance is associated with a persistence context. A persistence context is associated with a set of entities such that for each persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their life cycles are managed. The `EntityManager` interface consists of methods to interact with the persistence context so as to create, remove, find, and query persistent entities. Source E.8 provides the definition of this interface, and Table E.15 gives a brief explanation of the methods of this interface.

```
public interface javax.persistence.EntityManager
{
     public void persist(java.lang.Object entity);

     public <T> T merge(T entity);

     public void remove(java.lang.Object entity);

     public <T> T find (java.lang.Class<T> entityClass,
          java.lang.Object primaryKey);

     public <T> T getReference(java.lang.Class<T> entityClass,
          java.lang.Object primaryKey);

     public void flush();
```

```
    public void setFlushMode(
         javax.persistence.FlushModeType flushMode);

    public javax.persistence.FlushModeType getFlushMode();

    public void lock(java.lang.Object entity,
         javax.persistence.LockModeType lockMode);

    public void refresh(java.lang.Object entity);

    public void clear();

    public boolean contains(java.lang.Object entity);

    public javax.persistence.Query createQuery(
         java.lang.String qlString);

    public javax.persistence.Query createNamedQuery(
         java.lang.String name);

    public javax.persistence.Query createNativeQuery(
         java.lang.String sqlString);

    public javax.persistence.Query createNativeQuery(
         java.lang.String sqlString, java.lang.Class resultClass);

    public javax.persistence.Query createNativeQuery(
         java.lang.String sqlString,
         java.lang.String resultSetMapping);

    public void joinTransaction();

    public java.lang.Object getDelegate();

    public void close();

    public boolean isOpen();

    public javax.persistence.EntityTransaction getTransaction();
}
```

Source E.8 The javax.persistence.EntityManager interface.

Table E.15 javax.persistence.EntityManager

METHOD	EXPLANATION

`persist(Object)`	Call this method to make an entity object, passed as the method argument, persistent and managed. The persistence provider will throw `javax.persistence.EntityExistsException` if this operation is invoked on an entity that already exists. The `javax.persistence.TransactionRequiredException` is thrown if this method is invoked outside of an transaction on a container-managed entity manager instance with transaction-scoped persistence context. Also, the `java.lang.IllegalArgumentException` is thrown if the object to be made persistable and managed is not an entity type. Finally, the method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`merge(T entity)`	This method allows for the propagation of state from a detached entity onto the persistent entity managed by the `EntityManager`. The `javax.persistence.TransactionRequiredException` is thrown if this method is invoked outside of an transaction on a container-managed entity manager instance with transaction-scoped persistence context. Also, the `java.lang.IllegalArgumentException` is thrown if the method argument is not an entity type or is a removed entity. Finally, the method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`remove(Object)`	This method removes the entity, which in turn will remove the entity data (state) from the database at or before transaction commit, or as a result of the `flush()` operation. The `javax.persistence.TransactionRequiredException` is thrown if this method is invoked outside of an transaction on a container-managed entity manager instance with transaction-scoped persistence context. Also, the `java.lang.IllegalArgumentException` is thrown if the method argument is not an entity type or is a detached entity. Finally, the method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.

`find(Class<T>, Object)`	Use this method to find an entity by its primary key. This method is not required to be invoked within a transaction context. Thus, if an entity manager with transaction-scope persistence context is in use, the resulting entity will be detached, and if an entity manager with an extended persistence context is used, the resulting entity will be managed. The `java.lang.IllegalArgumentException` is thrown if the first method argument does not denote an entity type or if the second argument is not a valid type for that entity's primary key. The method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`getReference(Class<T>, Object)`	This method can be used to get an instance of an entity; its class is specified in the first method argument, and its primary key is supplied in the second method argument. The instance state may be lazily fetched. If the requested entity does not exist in the database, a `javax.persistence.EntityNotFoundException` is thrown either when the method is called or when the instance state is first accessed. Hence, your application should not assume the availability of the instance state upon detachment, unless you have accessed its state successfully while the entity manager was open. The `java.lang.IllegalArgumentException` is thrown if the first method argument does not denote an entity type or if the second argument is not a valid type for that entity's primary key. The method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.

`flush()`	This method can be used to force synchronization of persistence context with the database. This synchronization involves writing any updates to the persistent entities and their relationships (as dictated by the specified cascading behavior) to the database. Synchronization to the database does not involve refreshing the state of managed entities. The `setFlushMode()` operation on `EntityManager` and `Query` APIs can be used to control synchronization semantics. It throws `javax.persistence.PersistenceException` if the flush operation fails for some reason. Since the flush() operation should always be invoked from within a transaction, a `javax.persistence.TransactionRequiredException` is thrown if this condition is not met. The method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`setFlushMode(FlushModeType)`	Use this method to set the flush mode (`FlushModeType.AUTO` or `FlushModeType.COMMIT`) for all the entities managed within the underlying persistence context. It throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`getFlushMode()`	Use this method to retrieve the current flush mode setting. The method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.

`lock(Object, LockModeType)`	Use this method to set the lock mode (`LockModeType.READ` or `LockModeType.WRITE`) for the supplied entity. The `javax.persistence.TransactionRequiredException` is thrown if the method is executed outside of a transaction.The `javax.persistence.PersistenceException` is thrown if the client cannot support the requested locking mode. For instance, a persistence provider is not required to support `LockModeType.READ` on a nonversioned entity object. In this case, if the read lock mode is still requested, and if the persitsence provider does not support it, the provider will throw the `PersistenceException`. The `java.lang.IllegalArgumentException` is thrown if the first method argument does not denote an entity type or if the second argument is not a valid type for that entity's primary key. The method throws `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`refresh(Object)`	This method should be invoked to refresh the state of the entity instance from the database. The method should always be invoked within an active transaction when using container-managed entity manager instance with transaction-scoped persistence context. Failure to do so will result in a `javax.persistence.TransactionRequiredException`. The method throws a `javax.persistence.EntityNotFoundException` if the requested entity no longer exists in the database. The `java.lang.IllegalArgumentException` is thrown if the method argument is not a valid entity type. Also, it throws a `java.lang.IllegalStateException`, if invoked on a closed entity manager object.
`clear()`	Call this method on entity manager instance to clear the persistence context, which in turn will cause all the entity objects for this persistence context to become detached. In this process, all the changes to entities that have not been flushed to the database will not be persisted. The method throws `java.lang.IllegalStateException` if invoked on a closed entity manager object.

`contains(Object)`	This method can be used to check if the supplied entity object belongs to the current persistence context or not. It is the responsibility of an application to ascertain that an instance is managed in only a single persistence context. Failure to do so—that is, if an entity instance is managed in more than one persistence context—can result in unexpected behavior. This method can be useful in checking for this condition. The `java.lang.IllegalArgumentException` is thrown if the method argument is not a valid entity type and a `java.lang.IllegalStateException` is thrown if the method is invoked on a closed entity manager object.
`createQuery(String)`	This method will create an instance of `javax.persistence.Query` in order to execute the supplied Java Persistence Query Language string. The method throws `java.lang.IllegalArgumentException` if the supplied string does not follow the Java Persistence Query Language syntax. Also, it throws `java.lang.IllegalStateException` if the method is invoked on a closed entity manager object.
`createNamedQuery(String)`	This method creates an instance of `javax.persistence.Query` in order to execute a named query that might have been defined using the Java Persistence Query Language or in native SQL. The method throws `java.lang.IllegalArgumentException` if a query with the supplied name has not been defined. Also, it throws a `java.lang.IllegalStateException` if the method is invoked on a closed entity manager object.
`createNativeQuery(String)`	Use this method to create an instance of `javax.persistence.Query` in order to execute the native SQL query string supplied as the method argument. The method throws a `java.lang.IllegalStateException` if invoked on a closed entity manager object.

`createNativeQuery(String, Class)`	This method creates an instance of `javax.persistence.Query` to execute the native SQL query string supplied in the first argument and return the resulting instances of the entity whose class type is supplied in the second argument. The method throws `java.lang.IllegalStateException` if invoked on a closed entity manager object.
`createNativeQuery(String, String)`	Use this method to create an instance of `javax.persistence.Query` to execute the native SQL query string passed in the first argument, and populate the entities with the query results as specified by the `SqlResultSetMapping` definition (the `SqlResultSetMapping` name is passed in the second argument). This result set mapping metadata definition can then be used by the persistence provider to map the query results into the expected entity objects. Mostly, if the results of the query are limited to homogeneous entities, the use of `SqlResultSetMapping` is not required; in this case, you can use other simpler variations of this method. The method throws `java.lang.IllegalStateException` if invoked on a closed entity manager object.
`joinTransaction()`	This method is used to associate an application-managed entity manager instance, which was created outside of an active JTA transaction, with the current JTA transaction. The method throws `javax.persistence.TransactionRequiredException` if there is no current transaction, and it throws `java.lang.IllegalStateException` if the method is invoked on a closed entity manager object.
`getDelegate()`	This method returns the underlying (wrapped) provider object for the entity manager. The result of this method varies across the persistence providers. The method throws `java.lang.IllegalStateException` if invoked on a closed entity manager object.

`close()`	This method closes an application-managed `javax.persistence.EntityManager`. After this method is invoked, calls to all the other methods on `javax.persistence.EntityManager` and any `javax.persistence.Query` objects obtained from it, except for `getTransaction()` and `isOpen()`, will throw `java.lang.IllegalStateException`. The method also throws `java.lang.IllegalStateException` if invoked on a container-managed entity manager instance.
`isOpen()`	This method is used to determine whether the entity manager is open.
`getTransaction()`	This method returns an instance of `javax.persistence.EntityTransaction`. This is a resource-level transaction object that is mapped to the resource that underlies the entities managed by the entity manager. The method throws `java.lang.IllegalStateException` when invoked on a JTA entity manager instance.

EntityManagerFactory

An `EntityManagerFactory` is used to create an application-managed entity manager instance. The persistence context associated with an application-managed entity manager is standalone in that it is not propagated automatically along with the underlying JTA transaction across the multiple entity manager references for the given persistence unit. Hence, the application-managed entity manager has an isolated persistence context that is not accessible to other entity managers. Even though the need for application-managed entity managers is less common in Java EE applications, all Java EE EJB and Web containers are required to support it. Source E.9 provides the definition of this interface, and Table E.16 gives a brief explanation of the methods of this interface.

```
public interface javax.persistence.EntityManagerFactory
{
     public javax.persistence.EntityManager createEntityManager();

     public javax.persistence.EntityManager createEntityManager
          (Map map);

     public void close();
```

```
     public boolean isOpen();
}
```

Source E.9 The EntityManagerFactory interface.

Table E.16 javax.persistence.EntityManagerFactory

METHOD	EXPLANATION
`createEntityManager()`	This method creates a new application-managed entity manager instance.
`createEntityManager(Map)`	Use this method to create a new application-managed entity manager instance, thereby utilizing the persistence properties passed in the `java.util.Map` argument.
`close()`	Close the entity manager factory instance. Once an entity manager factory is closed, all the associated entity manager instances are considered to be in a closed state.
`isOpen()`	Returns a boolean indicating whether the factory instance is open or not.

EntityTransaction

An `EntityTransaction` interface represents a resource-level transaction object that is used by an entity manager to control transactions of the resource that underlies the entities managed by the entity manager. This interface is used to demarcate transactions that are local to the resource. The instance of `EntityTransaction` can be obtained by calling the `getTransaction()` on an `EntityManager` object. Source E.10 provides the definition of this interface, and Table E.17 gives a brief explanation of the methods of this interface.

```
public interface javax.persistence.EntityTransaction
{
     public void begin();

     public void commit();

     public void rollback();

     public void setRollback();
```

```
    public boolean getRollback();

    public boolean isActive();
}
```

Source E.10 The javax.persistence.EntityTransaction interface.

Table E.17 javax.persistence.EntityTransaction

METHOD	EXPLANATION
`begin()`	Use this method to start the local transaction on the underlying resource. The method throws `java.lang.IllegalStateException` if the transaction is already active.
`commit()`	Use this method to commit the current transaction. The method throws `java.lang.IllegalStateException` if there is no active transaction to commit. Also, `javax.persistence.RollbackException` is thrown to the client if the commit operation fails for some reason. In this case, the persistence provider will roll back the transaction.
`rollback()`	Use this method to roll back the current transaction. The method throws `java.lang.IllegalStateException` if there is not active transaction to roll back. It also throws a `javax.persistence.PersistenceException` in case a rollback is not successful.
`setRollbackOnly()`	Use this method to mark the current transaction for a rollback. It throws `java.lang.IllegalStateException` if the transaction is not active.
`getRollbackOnly()`	Use this method to determine if the current transaction has been marked for a rollback or not. The method throws a `java.lang.IllegalStateException` if no transaction is active.

`isActive()`	The method indicates whether a transaction is in progress. It throws a `javax.persistence.PersistenceException` in case of an unexpected error.

Query

A `javax.persistence.Query` object represents the Query API, which can be used for the execution of both static and dynamic Java Persistence Query Language as well as EJB Query Language queries. A reference to the `Query` object can be obtained by using the `createQuery()` and `createXxxQuery()` methods of the `javax.persistence.EntityManager` API. The Query API is quite comprehensive in its support of various facilities for query execution. Source E.11 provides the definition of this interface, and Table E.18 gives a brief explanation of the methods of this interface.

```
public interface javax.persistence.Query
{
     public java.util.List getResultList();

     public java.lang.Object getSingleResult();

     public int executeUpdate();

     public javax.persistence.Query setMaxResults(int maxResult);

     public javax.persistence.Query setFirstResult(int startPosition);

     public javax.persistence.Query setHint(java.lang.String hintName,
          java.lang.Object value);

     public javax.persistence.Query setParameter(java.lang.String name,
          java.lang.Object value);

     public javax.persistence.Query setParameter(java.lang.String name,
          java.util.Date value,
          javax.persistence.TemporalType temporalType);

     public javax.persistence.Query setParameter(java.lang.String name,
          java.util.Calendar value,
          javax.persistence.TemporalType temporalType);

     public javax.persistence.Query setParameter(int position,
          javax.lang.Object value);
```

```
     public javax.persistence.Query setParameter(int position,
          java.util.Date value,
          javax.persistence.TemporalType temporalType);

     public javax.persistence.Query setParameter(int position,
          java.util.Calendar value,
          javax.persistence.TemporalType temporalType);

     public javax.persistence.Query setFlushMode(
          javax.persistence.FlushModeType flushMode);
}
```

Source E.11 The javax.persistence.Query interface.

Table E.18 javax.persistence.Query

METHOD	EXPLANATION
`getResultList()`	Calling this method executes the SELECT query associated with the underlying `javax.persistence.Query` instance. The query results are returned as a `java.util.List` of entity objects. If the SELECT query has more than one select expression, the returned `List` will be of type `Object[]`, that is, an object array. The method throws `java.lang.IllegalStateException` if called for a DELETE or an UPDATE Java Persistence query language statement.
`getSingleResult()`	The method executes a SELECT query associated with the underlying `javax.persistence.Query` instance, which returns a single result. The method throws `javax.persistence.NoResultException` if there is no result as a part of executing the query. It throws `javax.persistence.NonUniqueResultException` if more than one result is returned as a part of executing the query. Finally, it returns a `java.lang.IllegalStateException` if called for a DELETE or an UPDATE Java Persistence query language statement.

`executeUpdate()`	This method should be used to execute an UPDATE or a DELETE statement associated with the underlying `javax.persistence.Query` instance. The method throws `javax.persistence.TransactionRequiredException` when executed outside of an active transaction. Also, it throws `java.lang.IllegalStateException` if called for a SELECT Java Persistence query language statement.
`setMaxResults(int)`	Use this method to specify the maximum number of results to be retrieved upon execution of the underlying query. The method throws `java.lang.IllegalArgumentException` if the supplied method argument is a negative integer.
`SetFirstResult(int)`	Specify the position of the first result, of all the results starting with the number 0, to retrieve. The method throws `java.lang.IllegalArgumentException` if the supplied method argument is a negative integer.
`setHint(String, Object)`	This method is used to specify an implementation specific hints in the areas of, say, locking, caching, behavior of various things such as fetch sizes when using joins, and so on. The hint is silently ignored if the hint name passed in the first method argument is not recognized by the persistence provider. However, if the hint name is recognized, then the provider can throw a `java.lang.IllegalArgumentException`, if the second argument is found invalid for the specific provider.

`setParameter(String, Object)`	This variation of the `setParameter()` method binds the object, supplied in the second method argument, to the parameter name, supplied in the first method argument. Note that the use of named parameters is defined only for the Java Persistence query language and not for the native queries. Hence, only positional parameter binding may be used in a portable fashion for the native queries. The method throws `java.lang.IllegalArgumentException` if the parameter name specified in the first method argument does not correspond to a parameter in the query string or if the `Object` argument supplied in the second parameter is of a type different than what is expected.
`setParameter(String, Date, TemporalType)`	This variation lets you bind an object of type `java.util.Date` to a named parameter. The third method argument, `javax.persistence.TemporalType`, is an enum value for identifying the temporal type of the data, viz. date, time, or datetime, represented by the `Date` object. This enum value helps the persistence provider to properly persist the date data to the underlying database. The method throws `java.lang.IllegalArgumentException` if the parameter name specified does not correspond to a parameter in the underlying query string.
`setParameter(String, Calendar, TemporalType)`	Use this method to bind an instance of `java.util.Calendar` to a named parameter. The method throws `java.lang.IllegalArgumentException` if the parameter name specified does not correspond to a parameter in the underlying query string.
`setParameter(int, Object)`	Use this variation of `setParameter()` method to bind an object to the positional parameter. The method throws `java.lang.IllegalArgumentException` if the supplied position does not correspond to a positional parameter of the query, or if the `Object` argument supplied in the second parameter is of a type different than what is expected.

`setParameter(int, Date, TemporalType)`	Use this variation to bind an instance of `java.util.Date` to a positional parameter. The method throws a `java.lang.IllegalArgumentException` if the supplied position does not correspond to a positional parameter of the query.
`setParameter(int, Calendar, TemporalType)`	This variation of `setParameter()` binds an instance of `java.util.Calendar` to a positional parameter. The method throws a `java.lang.IllegalArgumentException` if the supplied position does not correspond to a positional parameter of the query.
`setFlushMode(FlushModeType)`	Use this method to set the flush mode for the query execution. The flush mode setting of the `javax.persistence.Query` instance will override that of the `javax.persistence.EntityManager` for purpose of synchronization of results obtained through this query's execution.

Note that query methods, except `executeUpdate()`, are not required to be invoked within a transaction context.

Java Persistence API Exception Reference

Table E.19 describes the purpose of each exception class relevant to the Java Persistence API programming model.

Table E.19 Java Persistence API Exception Explanations

EXCEPTION	DESCRIPTION
`EntityExistsException`	A persistence provider can throw this runtime exception at the time of calling `persist()` or `flush()` operations, or at the time of transaction commit. It indicates that the entity being persisted already exists in the database.

`EntityNotFoundException`	This runtime exception can be thrown either upon invoking the `getReference()` method on `javax.persistence.EntityManager` or upon first accessing the state of an entity, whose reference had been obtained by `getReference()` method. The choice of when to throw this exception has been left to the persistence provider. But an important point to note is that an application should not assume that it will be successful in accessing the state of an entity obtained through `getReference()` method upon detachment, unless it has done so (without an exception) when the entity was being managed. Also, this exception could be thrown by invoking the `refresh()` operation for an entity that no longer exists in the database.
`NonUniqueResultException`	This runtime exception is thrown when execution of a `getSingleResult()` method on `javax.persistence.Query` object returns more than one result.
`NoResultException`	This runtime exception is thrown when execution of a `getSingleResult()` method on `javax.persistence.Query` object returns zero results.
`OptimisticLockException`	The persistence provider throws this runtime exception when an optimistic locking conflict occurs, during an API call such as `merge()` operation or at the time of flush or commit, depending on the provider implementation. Optimistic locking ensures that any updates or deletes to an entity's underlying data take place only when no intervening transaction has updated or deleted this underlying data since the state of the entity was last read from the database. Any changes to an entity's state that would cause violation of the above constraint will make the persistence provider throw this exception and roll back the currently active transaction.

`PersistenceException`	This is the runtime exception that is thrown by the persistence provider whenever a problem occurs. It is the supertype of all other Java Persistence API exceptions. All the instances of `javax.persistence.PersistenceException`, except `javax.persistence.NoResultException` and `javax.persistence.NonUniqueResultException`, will cause any currently active transaction to be marked for a rollback.
`RollbackException`	The persistence provider throws this runtime exception when a call to the `commit()` method on `javax.persistence.EntityTransaction` fails. The `EntityTransaction` object should be used for resource-local transaction demarcation.
`TransactionRequiredException`	This runtime exception is thrown by the persistence provider when a transaction is required but is not active during the invocations of `persist()`, `merge()`, and the like.

The Java Persistence API Miscellaneous Reference

Table E.20 The Types of Entity Managers

TYPE	MEANING
Container-managed	The life cycle of the container-managed entity manager is always managed automatically, transparent to the application, by the container. So also, the persistence context is propagated transparently with the JTA transaction by the container.

Application-managed	The life cycle of an application-managed entity manager is controlled by the application via the `javax.persistence.EntityManagerFactory` instance. The persistence contexts associated with the application-managed entity managers is always extended. Such an application-managed extended persistence context is never propagated with the transaction; in other words, it is a standalone persistence context. When a JTA transaction is used with application-managed extended persistence context, and when the entity manager is created outside of the JTA transaction, it is the responsibility of the application to associate entity manager with the JTA transaction (if so desired) by calling `joinTransaction()` method on `EntityManager`.

Table E.21 The Types of Container-Managed Persistence Contexts

TYPE	MEANING
Transaction-scope	A container-managed persistence context that is defined to have a lifetime scoped to a single transaction. A new container-managed transaction-scoped persistence context is created when any method on an entity manager object is invoked within an active JTA transaction, and there is no current persistence context already associated with that JTA transaction. The container-managed transaction-scoped persistence context ends when the associated JTA transaction is committed or rolled back, and all the entities managed by the entity manager become detached.
Extended	A container-managed persistence context that is defined to have a lifetime spanning multiple transactions. A container-managed extended persistence context can only be initiated within the scope of a stateful session bean. It exists from the point at which the stateful session bean, which declares a dependency on an entity manager of type `javax.persistence.PersistenceContextType.EXTENDED`, is created. The persistence context ends when the `@Remove` method of the stateful session bean completes, or the bean is otherwise destroyed.

Table E.22 The Types of Entity Managers Apropos Transactions Supported

TYPE	MEANING
JTA Entity Manager	A JTA transaction is managed externally to the entity manager by the container. An entity manager whose underlying transactions are controlled through JTA in this fashion is termed a JTA entity manager. Since JTA transactions are supported by all Java EE containers, a container-managed entity manager must be also a JTA entity manager. Note that an application-managed entity manager can also be a JTA entity manager.
Resource-local	An entity manager whose transactions are local to the resource and controlled by the application through the `javax.persistence.EntityTransaction` API is termed a resource-local entity manager. An application-managed entity manager can opt to be a resource-local entity manager.