When a message-driven bean using bean-managed transaction demarcation uses the `javax.transaction.UserTransaction` interface to demarcate transactions, the message receipt that causes the bean to be invoked is not part of the transaction. If the message receipt is to be part of the transaction, container-managed transaction demarcation with the `REQUIRED` transaction attribute must be used.

The `newInstance` method, the `setMessageDrivenContext` method, the message-driven bean's dependency injection methods, and lifecycle callback methods are called with an unspecified transaction context. Refer to Subsection 8.6.5 for how the container executes methods with an unspecified transaction context.

5.4.13 Security Context of Message-Driven Bean Methods

A caller principal may propagate into a message-driven bean's message listener methods. Whether this occurs is a function of the specific message-listener interface and associated messaging provider, but is not governed by this specification.

The Bean Provider can use the `RunAs` metadata annotation (or corresponding deployment descriptor element) to define a run-as identity for the enterprise bean. The run-as identity applies to the bean's message listener methods and timeout methods. Run-as identity behavior is further defined in section 12.3.4.1.

5.4.14 Association of a Message-Driven Bean with a Destination or Endpoint

A message-driven bean is associated with a destination or endpoint when the bean is deployed in the container. It is the responsibility of the Deployer to associate the message-driven bean with a destination or endpoint.

5.4.15 Activation Configuration Properties

The Bean Provider may provide information to the Deployer about the configuration of the message-driven bean in its operational environment. This may include information about message acknowledgement modes, message selectors, expected destination or endpoint types, etc.

Activation configuration properties are specified by means of the `activationConfig` element of the `MessageDriven` annotation or `activation-config` deployment descriptor element. Activation configuration properties specified in the deployment descriptor are added to those specified by means of the `MessageDriven` annotation. If a property of the same name is specified in both, the deployment descriptor value overrides the value specified in the annotation.

5.4.16 JMS Message-Driven Beans

This section describes activation configuration properties specific to the JMS message-driven beans.

The container remains free to decide whether to support its built-in JMS provider using a resource adapter or not. However it must allow the application to configure a MDB that uses the built-in JMS provider using the activation properties defined here.

Both the container and any JMS resource adapters are free to support activation properties in addition to those listed here. However applications which use non-standard activation properties may not be portable.

5.4.16.1 Message Acknowledgment

JMS message-driven beans should not attempt to use the JMS API for message acknowledgment. Message acknowledgment is automatically handled by the container. If the message-driven bean uses container-managed transaction demarcation, message acknowledgment is handled automatically as a part of the transaction commit. If bean-managed transaction demarcation is used, the message receipt cannot be part of the bean-managed transaction, and, in this case, the receipt is acknowledged by the container. If bean-managed transaction demarcation is used, the Bean Provider can indicate whether JMS `AUTO_ACKNOWLEDGE` semantics or `DUPS_OK_ACKNOWLEDGE` semantics should apply by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the acknowledgment mode is `acknowledgeMode`. If the `acknowledgeMode` property is not specified, JMS `AUTO_ACKNOWLEDGE` semantics are assumed. The value of the `acknowledgeMode` property must be either `Auto-acknowledge` or `Dups-ok-acknowledge` for a JMS message-driven bean.

5.4.16.2 Message Selectors

The Bean Provider may declare the JMS message selector to be used in determining which messages a JMS message-driven bean is to receive. If the Bean Provider wishes to restrict the messages that a JMS message-driven bean receives, the Bean Provider can specify the value of the message selector by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the message selector is `messageSelector`.

For example:

```
@MessageDriven(activationConfig={
     @ActivationConfigProperty(
          propertyName="messageSelector",
          propertyValue="JMSType = ‘car’ AND color = ‘blue’ AND weight
> 2500")})
```

```
<activation-config>
<activation-config-property>
<activation-config-property-name>messageSelector</activation-con-
fig-property-name>
<activation-config-property-value>JMSType = ‘car’ AND color = ‘blue’
AND weight &gt; 2500</activation-config-property-value>
</activation-config-property>
</activation-config>
```

The Application Assembler may further restrict, but not replace, the value of the `messageSelector` property of a JMS message-driven bean.

5.4.16.3 Destination Type

A JMS message-driven bean is associated with a JMS Destination (Queue or Topic) when the bean is deployed in the container. It is the responsibility of the Deployer to associate the message-driven bean with a Queue or Topic.

The Bean Provider may provide advice to the Deployer as to whether a message-driven bean is intended to be associated with a queue or a topic by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the destination type associated with the bean is `destinationType`. The value for this property must be either `javax.jms.Queue` or `javax.jms.Topic` for a JMS message-driven bean.

5.4.16.4 Destination Lookup

The bean provider or deployer may specify the JMS queue or topic from which a JMS message-driven bean is to receive messages.

The lookup name of an administratively-defined `Queue` or `Topic` object may be specified by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the lookup name is `destinationLookup`.

5.4.16.5 Connection Factory Lookup

The bean provider or deployer may specify the JMS connection factory that will be used to connect to the JMS provider from which a JMS message-driven bean is to receive messages.

The lookup name of an administratively-defined `ConnectionFactory` object may be specified by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the lookup name is `connectionFactoryLookup`.

5.4.16.6 Subscription Durability

If the message-driven bean is intended to be used with a topic, the Bean Provider may further indicate whether a durable or non-durable subscription should be used by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify whether a durable or non-durable subscription should be used is `subscriptionDurability`. The value for this property must be either `Durable` or `NonDurable` for a JMS message-driven bean. If a topic subscription is specified and `subscriptionDurability` is not specified, a non-durable subscription is assumed.

- Durable topic subscriptions, as well as queues, ensure that messages are not missed even if the EJB server is not running. Reliable applications will typically make use of queues or durable topic subscriptions rather than non-durable topic subscriptions.

- If a non-durable topic subscription is used, it is the container's responsibility to make sure that the message-driven bean subscription is active (i.e., that there is a message-driven bean available to service the message) in order to ensure that messages are not missed as long as the EJB server is running. Messages may be missed, however, when a bean is not available to service them. This will occur, for example, if the EJB server goes down for any period of time.

The Deployer should avoid associating more than one message-driven bean with the same JMS Queue. If there are multiple JMS consumers for a queue, JMS does not define how messages are distribued between the queue receivers.

5.4.16.7 Subscription Name

If the message-driven bean is intended to be used with a Topic, and the bean provider has indicated that a durable subscription should be used by specifying the `subscriptionDurability` property to `Durable`, then the bean provider or deployer may specify the name of the durable subscription.

The name of the durable subscription may be specified by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the name of the durable subscription is `subscriptionName`.

If a durable subscription is specified but `subscriptionName` is not specified then the container will set the name of the durable subscription to be a name which is unique to the deployed MDB (see 5.7.3). If the message-driven bean is deployed into a clustered application server then the `shareSubscriptions` property will be used to determine whether the durable subscription name generated by the container will be the same or different for each instance in the cluster.

5.4.16.8 Durable Subscription Name in Clustered Deployment

If message-driven bean is intended to be used with a Topic and the bean provider or deployer has specified that a *durable* subscription be used but has not specified a durable subscription name then the bean provider or deployer may specify whether the durable subscription name generated by the container will be the same or different for each instance in the cluster.

If message-driven bean is intended to be used with a topic and the bean provider or deployer has specified that a *non-durable* subscription be used then the bean provider or deployer may specify whether the same non-durable subscription should be used for each instance in the cluster.

This may be specified by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used is `shareSubscriptions`.

This property is only used if the message-driven bean is deployed into a clustered application server. The property may have the string values `true` or `false`.

A value of `true` means that the same durable subscription name or non-durable subscription will be used for each instance in the cluster.

A value of `false` means that a different durable subscription name or non-durable subscription will be used for each instance in the cluster.

By default a value of `true` is assumed.

5.4.16.9 Client Identifier

The bean provider or deployer may specify the JMS client identifier that will be used when connecting to the JMS provider from which a JMS message-driven bean is to receive messages.

The client identifier may be specified by using the `activationConfig` element of the `MessageDriven` annotation or by using the `activation-config-property` deployment descriptor element. The property name used to specify the client identifier is `clientId`.

If this property is not specified then the client identifier will be left unset.

5.4.17 Dealing with Exceptions

A message-driven bean's message listener method must not throw the `java.rmi.RemoteException`.

Message-driven beans should not, in general, throw `RuntimeExceptions`.

A `RuntimeException` that is not an application exception thrown from any method of the message-driven bean class (including a message listener method and the callbacks invoked by the container) results in the transition to the "does not exist" state. If a message-driven bean uses bean-managed transaction demarcation and throws a `RuntimeException`, the container should not acknowledge the message. Exception handling is described in detail in Chapter 9. See Section 7.5.1 for the rules pertaining to lifecycle callback interceptor methods when more than one such method applies to the bean class.

From the client perspective, the message consumer continues to exist. If the client continues sending messages to the destination or endpoint associated with the bean, the container can delegate the client's messages to another instance.

The message listener methods of some messaging types may throw application exceptions. An application exception is propagated by the container to the resource adapter.

5.4.18 Missed PreDestroy Callbacks

The Bean Provider cannot assume that the container will always invoke the `PreDestroy` callback method (or `ejbRemove` method) for a message-driven bean instance. The following scenarios result in the `PreDestroy` callback method not being called on an instance:

- A crash of the EJB container.

- A system exception thrown from the instance's method to the container.

If the message-driven bean instance allocates resources in the `PostConstruct` lifecycle callback method and/or in the message listener method, and releases normally the resources in the `PreDestroy` method, these resources will not be automatically released in the above scenarios. The application using the message-driven bean should provide some clean up mechanism to periodically clean up the unreleased resources.

5.4.19 Replying to a JMS Message

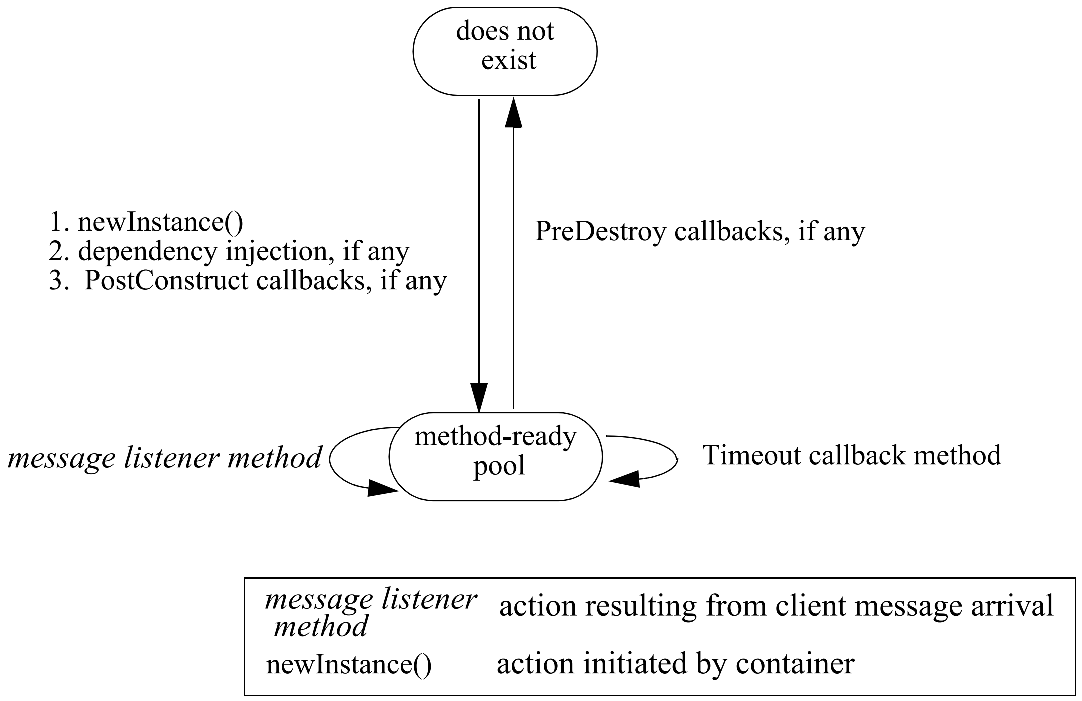
In standard JMS usage scenarios, the messaging mode of a message's `JMSReplyTo` destination (Queue or Topic) is the same as the mode of the destination to which the message has been sent. Although a message-driven bean is not directly dependent on the mode of the JMS destination from which it is consuming messages, it may contain code that depends on the mode of its message's `JMSReplyTo` destination. In particular, if a message-driven bean replies to a message, the mode of the reply's message producer and the mode of the `JMSReplyTo` destination must be the same. In order to implement a message-driven bean that is independent of `JMSReplyTo` mode, the Bean Provider should use `instanceOf` to test whether a `JMSReplyTo` destination is a Queue or Topic, and then use a matching message producer for the reply.

5.5 Message-Driven Bean State Diagram

When a client sends a message to a Destination for which a message-driven bean is the consumer, the container selects one of its method-ready instances and invokes the instance's message listener method.

The following figure illustrates the life cycle of a message-driven bean instance.

Figure 9 Life Cycle of a Message-Driven Bean.


does not exist
1. newInstance()
2. dependency injection, if any
3. PostConstruct callbacks, if any
PreDestroy callbacks, if any
message listener method
method-ready pool
Timeout callback method
message listener method
action resulting from client message arrival
newInstance()
action initiated by container

The following steps describe the life cycle of a message-driven bean instance:

- A message-driven bean instance's life starts when the container invokes `newInstance` on the message-driven bean class to create a new instance. Next, the container injects the bean's `MessageDrivenContext` object, if applicable, and performs any other dependency injection as specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the bean's `PostConstruct` lifecycle callback methods, if any.

- The message-driven bean instance is now ready to be delivered a message sent to its associated destination or endpoint by any client or a call from the container to a timeout callback method.

- When the container no longer needs the instance (which usually happens when the container wants to reduce the number of instances in the method-ready pool), the container invokes the `PreDestroy` lifecycle callback methods for it, if any. This ends the life of the message-driven bean instance.

5.5.1 Operations Allowed in the Methods of a Message-Driven Bean Class

Table 4 defines the methods of a message-driven bean class in which the message-driven bean instances can access the methods of the `javax.ejb.MessageDrivenContext` interface, the `java:comp/env` environment naming context, resource managers, `TimerService` and `Timer` methods, the `EntityManager` and `EntityManagerFactory` methods, and other enterprise beans.

If a message-driven bean instance attempts to invoke a method of the `MessageDrivenContext` interface, and the access is not allowed in Table 4, the container must throw and log the `java.lang.IllegalStateException.`

If a message-driven bean instance attempts to invoke a method of the `TimerService` or `Timer` interface, and the access is not allowed in Table 4, the container must throw the `java.lang.IllegalStateException`.

If a bean instance attempts to access a resource manager, an enterprise bean, or an entity manager or entity manager factory, and the access is not allowed in Table 4, the behavior is undefined by the EJB architecture.

Table 4 Operations Allowed in the Methods of a Message-Driven Bean

Bean method	Bean method can perform the following operations	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
dependency injection methods (e.g., setMessageDrivenContext)	MessageDrivenContext methods: *lookup* JNDI access to java:comp/env	MessageDrivenContext methods: *lookup* JNDI access to java:comp/env
PostConstruct, PreDestroy lifecycle callback methods	MessageDrivenContext methods: *getTimerService, lookup, getContextData* JNDI access to java:comp/env EntityManagerFactory access	MessageDrivenContext methods: *getUserTransaction, getTimerService, lookup, getContextData* JNDI access to java:comp/env EntityManagerFactory access
message listener method, AroundInvoke interceptor method	MessageDrivenContext methods: *getRollbackOnly, setRollbackOnly, getCallerPrincipal, isCallerInRole, getTimerService, lookup, getContextData* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods	MessageDrivenContext methods: *getUserTransaction, getCallerPrincipal, isCallerInRole, getTimerService, lookup, getContextData* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods
timeout callback method	MessageDrivenContext methods: *getRollbackOnly, setRollbackOnly, getCallerPrincipal, getTimerService, lookup, getContextData* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods	MessageDrivenContext methods: *getUserTransaction, getCallerPrincipal, getTimerService, lookup, getContextData* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer service or Timer methods

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `MessageDrivenContext` interface should be used only in the message-driven bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if the methods are invoked while the instance is not associated with a transaction.

The reasons for disallowing operations in Table 4:

- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the message-driven bean methods for which the container does not have a meaningful transaction context, and for all message-driven beans with bean-managed transaction demarcation.

- The `UserTransaction` interface is unavailable to message-driven beans with container-managed transaction demarcation.

- Invoking `getEJBHome` or `getEJBLocalHome` is disallowed in message-driven bean methods because there are no EJBHome or EJBLocalHome objects for message-driven beans. The container must throw and log the `java.lang.IllegalStateException` if these methods are invoked.

5.6 The Responsibilities of the Bean Provider

This section describes the responsibilities of the message-driven Bean Provider to ensure that a message-driven bean can be deployed in any EJB container.

5.6.1 Classes and Interfaces

The message-driven Bean Provider is responsible for providing the following class files:

- Message-driven bean class.

- Interceptor classes, if any.

5.6.2 Message-Driven Bean Class

The following are the requirements for the message-driven bean class:

- The class must implement, directly or indirectly, the message listener interface required by the messaging type that it supports or the methods of the message listener interface. In the case of JMS, this is the `javax.jms.MessageListener` interface.

- The class must be defined as `public`, must not be `final`, and must not be `abstract`. The class must be a top level class.

- The class must have a `public` constructor that takes no arguments. The container uses this constructor to create instances of the message-driven bean class.

- The class must not define the `finalize` method.

Optionally:

- The class may implement, directly or indirectly, the `javax.ejb.MessageDrivenBean` interface.
- The class may implement, directly or indirectly, the `javax.ejb.TimedObject` interface.
- The class may implement the `ejbCreate` method.

The message-driven bean class may have superclasses and/or superinterfaces. If the message-driven bean has superclasses, the methods of the message listener interface, lifecycle callback interceptor methods, timeout callback methods, the `ejbCreate` method, and the methods of the `MessageDrivenBean` interface may be defined in the message-driven bean class or in any of its superclasses. A message-driven bean class must not have a superclass that is itself a message-driven bean class

The message-driven bean class is allowed to implement other methods (for example, helper methods invoked internally by the message listener method) in addition to the methods required by the EJB specification.

5.6.3 Message-Driven Bean Superclasses

A message-driven bean class is permitted to have superclasses that are themselves message-driven bean classes. However, there are no special rules that apply to the processing of annotations or the deployment descriptor for this case. For the purposes of processing a particular message-driven bean class, all superclass processing is identical regardless of whether the superclasses are themselves message-driven bean classes. In this regard, the use of message-driven bean classes as superclasses merely represents a convenient use of *implementation inheritance*, but does not have *component inheritance* semantics.

5.6.4 Message Listener Method

The message-driven bean class must define the message listener methods. The signature of a message listener method must follow these rules:

The method must be declared as `public`.

The method must not be declared as `final` or `static`.

5.6.5 Lifecycle Callback Interceptor Methods

`PostConstruct` and `PreDestroy` lifecycle callback interceptor methods may be defined for message-driven beans. If `PrePassivate` or `PostActivate` lifecycle callbacks are defined, they are ignored.[31]

[31] This might result from the use of default interceptor classes, for example.

Compatibility Note: If the `PostConstruct` *lifecycle callback interceptor method is the* `ejbCreate` *method, or if the* `PreDestroy` *lifecycle callback interceptor method is the* `ejbRemove` *method, these callback methods must be implemented on the bean class itself (or on its superclasses). Except for these cases, the method names can be arbitrary, but must not start with "ejb" to avoid conflicts with the callback methods defined by the javax.ejb.EnterpriseBean interfaces.*

Lifecycle callback interceptor methods may be defined on the bean class and/or on an interceptor class of the bean. Rules applying to the definition of lifecycle callback interceptor methods are defined in Section 7.5, "Interceptors for LifeCycle Event Callbacks" .

5.7 The Responsibilities of the Container Provider

This section describes the responsibilities of the Container Provider to support a message-driven bean. The Container Provider is responsible for providing the deployment tools, and for managing the message-driven bean instances at runtime.

Because the EJB specification does not define the API between deployment tools and the container, we assume that the deployment tools are provided by the Container Provider. Alternatively, the deployment tools may be provided by a different vendor who uses the container vendor's specific API.

5.7.1 Generation of Implementation Classes

The deployment tools provided by the container are responsible for the generation of additional classes when the message-driven bean is deployed. The tools obtain the information that they need for generation of the additional classes by introspecting the classes and interfaces provided by the Enterprise Bean Provider and by examining the message-driven bean's deployment descriptor.

The deployment tools may generate a class that mixes some container-specific code with the message-driven bean class. This code may, for example, help the container to manage the bean instances at runtime. Subclassing, delegation, and code generation can be used by the tools.

5.7.2 Deployment of JMS Message-Driven Beans

The Container Provider must support the deployment of a JMS message-driven bean as the consumer of a JMS queue or a durable subscription.

5.7.3 Unique Identifier for JMS Message-Driven Bean

The Container Provider must make a name which uniquely identifies the deployed MDB in an application server instance, available in the JNDI naming context under `java:comp/UniqueMDBName` so that it can be looked up by the resource adapter when its `endpointActivation` method is called.

The resource adapter may use this name when constructing a default durable subscription name.