JMS 2.0: A simplified API

This document proposes how the JMS API 1.1 might be simplified.

Version 2
16th January 2012

Goals

The simplified API has the following goals:

- To reduce the number of objects needed to send and receive messages, and in particular to combine the JMS 1.1 Connection, Session, MessageProducer and MessageConsumer objects as much as possible.
- To take advantage of the fact that this is a new API to simplify method signatures and make other simplifications which cannot be made to the old API because it would break backwards compatibility.
- To maintain a consistent style with the existing API where possible so that users of the old API feel it to be an evolution which that can learn quickly.
- To support, and offer benefits to, both Java EE and Java SE applications.
- To allow resource injection to be exploited in those environment which support it, whilst still offering significant improvements for those environments which do not.
- To provide the option to send and receive message payloads to be sent and received directly without the need to use `javax.jms.Message` objects.
- To remove as much as possible the need to catch `JMSException` on method calls
- To be functionally complete. The old API will remain to provide backwards compatibility. However the new API is intended to be functionally as complete as the old JMS 1.1 API. Users should not need to switch back to the old API to perform an operation that is unavailable in the new API.

Key features of the new API

Introducing MessagingContext

The main object in the new API is `javax.jms.MessagingContext`. This combines the functionality of several JMS 1.1 objects into one: a `Connection`, a `Session` and an anonymous `MessageProducer` (one with no destination specified). It also combines the functionality of `MessageConsumer`, but only for aynchronous message delivery.

For synchronous delivery a separate object is still needed. This is `SyncMessageConsumer` which provides the functionality of `MessageConsumer` for synchronous message delivery.

In terms of the old API a `MessagingContext` should be thought of as representing both a `Connection` and a `Session`. These concepts remain relevant in the new API. As described in JMS 1.1, a connection represents a physical link to the JMS server, and a session represents a single-threaded context for sending and receiving messages.

Java EE allows only one session to be created on each connection, so combining them in a single method takes advantage of this restriction to offer a simpler API. Java EE applications will create `MessagingContext` objects using new factory methods on the `ConnectionFactory` interface.

Java SE applications allow multiple sessions on the same connection. This allows the same physical connection to be used in multiple threads simultaneously. Java SE applications which require multiple sessions to be created on the same connection may call `createMessagingContext` on an existing `MessagingContext` object to create a new `MessagingContext` using the same connection as the existing MessagingContext.

Sending messages

A `MessagingContext` works like an anonymous message producer (one with no destination specified) and offers send methods which allow a message to be delivered to the specified destination:

```
MessagingContext context = connectionFactory.createMessagingContext();
TextMesage textMessage = context.createTextMessage(payload);
context.send(inboundQueue,payload);
```

Complete examples of using the new API in both Java EE and Java SE applications are given in the "Examples" section below.

Consuming messages

Applications that consume messages *asynchronously* no longer need to create a `MessageConsumer`. Instead `MessagingContext` offers methods to allow a `MessageListener` to be set for a specified destination.

```
MessagingContext context =
   connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);
MessageListener messageListener = new MyListener();
context.setMessageListener(inboundQueue,messageListener);
```

Applications that consume messages *synchronously* will continue to need to create a separate consumer object. A new object `SyncMessageConsumer` is like a stripped-down `MessageConsumer` for synchronous delivery only.

```
MessagingContext context = connectionFactory.createMessagingContext();
SyncMessageConsumer syncMessageConsumer =
context.createSyncConsumer(inboundQueue);
TextMessage textMessage = (TextMessage)syncMessageConsumer.receivePayload();
```

The reason that `SyncMessageConsumer` needs to be a separate object rather than be combined with `MessagingContext` is to allow JMS providers to pre-cache messages in the consumer when it is first created and before the first call to `receive().` Although this might be thought of as a implementation detail, many JMS providers currently do this and providing `receive()` methods directly on the `MessagingContext` would prevent it.

New methods to allow a payload to be sent directly

Two new methods have been added to `MessagingContext` which allow a `TextMessage` or `ObjectMessage` to be sent by supplying the payload directly.

```
void send(Destination destination, String payload) throws JMSException;

void send(Destination destination, Serializable payload) throws JMSException;
```

The current proposal is to support `TextMessage` and `ObjectMessage` only, but it may be possible to extend this to other message types. It may also be possible to allow conversion code to be configured by the application.

New methods to allow a payload to be received directly

Three new methods have been added to the new interface `SyncMessageConsumer` which allow a message payload to be returned directly

```
<T> T receivePayload(Class<T>  c);

<T> T receivePayload(Class<T> c, long timeout);

<T> T receivePayloadNoWait(Class<T> c);
```

If the next message is a `TextMessage` when this should be set to `String.class`. If the next message is a `ObjectMessage` this should be set to `Serializable.class.` If the next message is not of the expected type a `ClassCastException` will be thrown and the message will not be delivered..

The current proposal is to support `TextMessage` and `ObjectMessage` only, but it may be possible to extend this to other message types. It may also be possible to allow conversion code to be configured by the application.

Closing the MessagingContext

A `MessagingContext` needs to be closed after use using the `close()` method. `MessagingContext` will implement `java.lang.AutoCloseable` to make this simpler.

In the following example `close()` is automatically called when the try block is completed:

```
try (MessagingContext context = connectionFactory.createMessagingContext();){
   context.send(inboundQueue,payload);
}
```

If the `MessagingContext` was created directly from a `ConnectionFactory` then calling `close()` will close both the underlying session and the underlying connection. If the `MessagingContext` was created from a `Connection` object then calling `close()` will close only the session. The application will need to close the connection explicitly.

No need to call connection.start()

Developers new to JMS often fail to call `connection.start()` and are surprised when they don't receive any messages. The JMS 1.1 specification explains that this method is needed to allow a client to be fully initialized before the delivery of messages to an asynchronous `MessageListener` is started:

> Section 4.3.3 "Connection Setup" of the JMS 1.1 specification states "It is typical to leave the Connection in stopped mode until setup is complete. At that point the Connection's start() method is called and messages begin arriving at the Connection's consumers. This setup convention minimizes any client confusion that may result from **asynchronous** message delivery while the client is still in the process of setting itself up.
>
> A Connection can be started immediately and the setup can be done afterwards. Clients that do this must be prepared to handle **asynchronous** message delivery while they are still in the process of setting up.
>
> Section 9.1.7, which describes how to synchronously receive messages, explains that connection.start() is needed "so that the… setup could be done without being interrupted with **asynchronously** delivered messages."

This makes it clear that although the JMS 1.1 specification requires `connection.start()` to be called to begin both synchronous and asynchronous delivery, this requirement was added to address an issue which only applies for asynchronous consumers. There is definitely no need for such a requirement when message delivery is synchronous.

Furthermore, the need to call `connection.start()` is unnecessary for asynchronous delivery as well. The specification could have stated that asynchronous message delivery would begin as soon as the client has called the `setMessageListener()` method. The application would then be expected to ensure that the supplied `MessageListener` object was fully initialized before the call to `setMessageListener()` was performed.

The new API will automatically start the underlying connection (if it has not already been started) when either `setMessageListener` or `createSyncMessageConsumer` are called on the `MessageContext` object. There will be `stop()` and `start()` methods on the `MessagingContext` which can be used to suspend and resume delivery of message.. However there will be no need to call `start()` when the consumer is first established.

A consequence of starting the connection as soon as `setMessageListener` is called means that calling `setMessageListener` a second time, perhaps for a second destination, will break the threading restrictions on a `Session`. This restriction is explicitly stated in section 4.4.6 of the JMS 1.1 spec. To avoid the need for applications to temporarily stop the connection before setting the second listener, the specification will explicitly permit this. The JMS provider will be responsible for supporting this usage, either by temporarily stopping the connection automatically or by providing an implementation which avoids the need to do this.

Static constants for session type

New static integer constants have been defined for use with the new API.

`MessagingContext.AUTO_ACKNOWLEDGE`,
`MessagingContext. CLIENT_ACKNOWLEDGE`,
`MessagingContext.DUPS_OK_ACKNOWLEDGE` and
`MessagingContext. SESSION_TRANSACTED`

These have the same values, and the same meaning, as the equivalent constants on `Session`. This avoids applications using the new API needing to be dependent on the `Session` interface.

A deliberate decision was made to use integer constants rather than enums to maintain consistency (and interchangeability) with the equivalent constants on Session.

Fewer Checked Exceptions

The new API does not throw checked exceptions such as JMSException. Instead, equivalent unchecked exceptions are thrown instead.

Most methods in the existing JMS 1.1 API were declared to throw checked exceptions (especially JMSException) even though the circumstances that might cause such an exception were not defined. Typically the javadoc would state that an exception was thrown in the event of some "internal error" or because an input value was invalid in some unspecified way. In most cases the application could not recover from such an exception: all it could do was to log the exception and rethrow it. In such cases it is considered good practice to throw an unchecked exception rather than a checked exception.

Although there may be a few special cases where an exception was recoverable, this API has been designed to throw no checked exceptions at all. This is consistent with modern Java EE APIs such as the javax.persistence API which never throws checked exceptions.

The new unchecked exceptions, and the checked exceptions to which they correspond (and which still remain in the old API) are listed below:

Old checked exception	**Corresponding new unchecked exception**
`JMSException` (checked exception)	`JMSRuntimeException` (unchecked exception)

`TransactionRolledBackException`	`TransactionRolledBackRuntimeException`
`IllegalStateException`	`IllegalStateRuntimeException`
`InvalidDestinationException`	`InvalidDestinationRuntimeException`
`InvalidSelectorException`	`InvalidSelectorRuntimeException`
`MessageFormatException`	`MessageFormatRuntimeException`
`JMSSecurityException`	`JMSSecurityRuntimeException`
`InvalidClientIDException`	`InvalidClientIDRuntimeException`

The javadocs for the new API list the unchecked exceptions which are expected to be thrown. These exceptions do not form part of the contract of these methods.

The old API continues to throw the same exceptions as before. This is to maintain backwards compatibility. Note that since applications using the new API may still need to use methods from the old API (especially methods on `javax.jms.Message` and its subtypes) the need to handle checked exceptions has not been eliminated entirely.

What state does a MessagingContext hold?

A `MessagingContext` holds the following state:

- A `Connection.` This can be created when the MessagingContext is created (Java EE or Java SE), or it can be created separately and passed in when the MessagingContext is created (Java SE only).
- A `Session`, including its `sessionMode` attribute which is passed in when the MessagingContext is created and cannot be changed.
- An anonymous `MessageProducer`, including its `deliveryMode`, `priority` and `timeToLive` attributes which are set using setter methods. These attribute may be overridden when a message is sent.
- Zero or more `MessageConsumer` objects used for asynchronous message delivery

A `MessagingContext` does *not* hold a destination object as state. When an API method needs a destination this is always passed in as an argument.

A `MessagingContext` is a factory for (but does not hold as state)

- `SyncMessageConsumer` objects
- `Message` objects

Unchanged interfaces

In addition to the modified `ConnectionFactory` and `Connection` objects, and the new `MessagingContext` and `SyncMessageConsumer` objects, the new API uses the following interfaces in the javax.jms package which are unchanged from JMS 1.1:

- `Message` (and its subtypes `BytesMessage`, `StreamMessage`, `MapMessage`, `TextMessage` and `ObjectMessage`)
- `Destination`, `Queue`, `Topic`, `TemporaryQueue`, `TemporaryTopic`, `DeliveryMode`
- `ExceptionListener`, `MessageListener`
- `QueueBrowser`

The above interfaces will continue to throw checked exceptions from the old API:

- `JMSException` (and its subtypes `IllegalStateException`, `InvalidClientIDException`, `InvalidDestinationException`, `InvalidPropertyException`, `InvalidSelectorException`, `JMSSecurityException`, `MessageEOFException`, `MessageFormatException`, `MessageNotReadableException`, `MessageNotWriteableException`, `ReadOnlyPropertyException`, `ResourceAllocationException`, `TransactionInProgressException`, `TransactionRollbackException`).

The new API does not throw these exceptions but will throw equivalent unchecked exceptions instead.

Other issues

The old API will remain, for ever

The new API described in this document will be additional to the existing API, which will remain a mandatory part of JMS 2.0. There is no intention to remove the existing API from future versions of JMS.

Relationship to Java Connector API

The new API should remove the need for Java EE applications to explicitly create `Connection` objects, though they will still be able to do so if needed.

It is not expected that Java EE containers will need to manage pools of `MessagingContext` objects separately from pools of `Connection` objects. Instead the `MessagingContext` objects should be thought of as lightweight wrappers around an existing `Connection` which continues to be pooled as now.

No additional API is needed to supported integration with application servers or resource adapters.

Injection of MessagingContext objects

This document does not describe how to use CDI to inject `MessagingContext` objects. One of the goals of this new API is to provide a simplified API to all applications, including those which are not using CDI or where a CDI environment is not available.

However it is intended that the simpler API, and in particular the reduced number of objects that it requires, should make it possible to use CDI to simplify the API still further. A future revision of this document may address this topic.

Examples

Sending a message (Java EE)

This example compares the old and new API for sending a `TextMessage` in a Java EE (EJB or web container) environment.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/ connectionFactory ")
ConnectionFactory connFact;

@Resource(lookup="jms/inboundQueue")
Destination destination;

public void sendMessageOld(String payload) throws JMSException{

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageProducer messageProducer = session.createProducer(inboundQueue);
   TextMessage textMessage = session.createTextMessage(payload);
   messageProducer.send(textMessage);
   connection.close();
}
```

Here's how you might do this using the new API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageNew(String payload) {

   try (MessagingContext context = connectionFactory.createMessagingContext();){
      context.send(inboundQueue,payload);
   }
}
```

Note that `sendMessageNew` does not need to throw `JMSException`.

Sending a message (Java SE)

This example compares the old and new API for sending a `TextMessage` in a Java SE environment.

Here's how you might do this using the existing JMS 1.1 API.

```
public void sendMessageOld(String payload) throws JMSException, NamingException{

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageProducer messageProducer = session.createProducer(inboundQueue);
   TextMessage textMessage = session.createTextMessage(payload);
   messageProducer.send(textMessage);
   connection.close();
}
```

Here's how you might do this using the new API.

```
public void sendMessageNew(String payload) throws NamingException{

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext context = connectionFactory.createMessagingContext();){
      context.send(inboundQueue,payload);
   }
}
```

Note that `receiveMessagesNew` does not need to throw `JMSException`.

Receiving a message synchronously (Java EE)

This example compares the old and new API for synchronously receiving a `TextMessage` in a Java EE (EJB or web container) environment.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Destination destination;

public String receiveMessageOld() throws JMSException {

   Connection connection = connectionFactory.createConnection();
   connection.start();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageConsumer messageConsumer = session.createConsumer(inboundQueue);
   TextMessage textMessage=(TextMessage) messageConsumer.receive();
   String payload = textMessage.getText();
   connection.close();
   return payload;
}
```

Here's how you might do this using the new API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageNew(){

   String payload;
   try (MessagingContext context = connectionFactory.createMessagingContext();) {
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncConsumer(inboundQueue);
      payload = syncMessageConsumer.receivePayload(String.class);
   }
   return payload;
}
```

Note that `receiveMessageNew` does not need to throw `JMSException`.

Receiving a message synchronously (Java SE)

This example compares the old and new API for synchronously receiving a `TextMessage` in a Java SE environment.

Here's how you might do this using the existing JMS 1.1 API.

```
public String receiveMessageOld() throws JMSException, NamingException {
   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageConsumer messageConsumer = session.createConsumer(inboundQueue);
   connection.start();
   TextMessage textMessage=(TextMessage) messageConsumer.receive();
   String payload = textMessage.getText();
   connection.close();
   return payload;
}
```

Here’s how you might do this using the new API.

```
@Resource(lookup = "jms/connectionFactoryWithClientID")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundTopic")
Topic inboundTopic;


public String receiveMessageNew() throws NamingException {

   String payload = null;

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext messagingContext =
          connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);) {
      SyncMessageConsumer syncMessageConsumer =
         messagingContext.createSyncConsumer(inboundQueue);
      payload = syncMessageConsumer.receivePayload(String.class);
   }
   return payload;
}
```

Note that `receiveMessageNew` does not need to throw `JMSException`.

Receiving a message synchronously from a durable subscription (Java EE)

This example compares the old and new API for synchronously receiving a `TextMessage` from a durable topic subscription in a Java EE (EJB or web container) environment.

Here’s how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageOld() throws JMSException {

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageConsumer messageConsumer = session.createDurableSubscriber(
      inboundTopic, "mysub");
   connection.start();
   TextMessage textMessage=(TextMessage) messageConsumer.receive();
   String payload = textMessage.getText();
   connection.close();
   return payload;
}
```

Here's how you might do this using the new API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageNew() {

   String payload = null;
   try (MessagingContext context = connectionFactory.createMessagingContext();){
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncDurableSubscriber(inboundTopic, "mysub");
      payload = syncMessageConsumer.receivePayload(String.class);
   }
   return payload;
}
```

Note that `receiveMessageNew` does not need to throw an exception.

Receiving messages asynchronously (Java SE)

This example compares the old and new API for asynchronously receiving `TextMessage` objects in a Java SE environment.

Here's how you might do this using the existing JMS 1.1 API.

```
public void receiveMessagesOld() throws JMSException, NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageConsumer consumer = session.createConsumer(inboundQueue);
   MessageListener messageListener = new MyListener();
   consumer.setMessageListener(messageListener);
   connection.start();

   // wait for messages to be received
   // details omitted

   connection.close();
}
```

Here's how you might do this using the new API.

```
public void receiveMessagesNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext context =
          connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);){
       MessageListener messageListener = new MyListener();
       context.setMessageListener(inboundQueue,messageListener);

      // wait for messages to be received - details omitted
   }
}
```

Note that `receiveMessagesNew` does not need to throw `JMSException`.

Receiving a message asynchronously from a durable subscription (Java SE)

This example compares the old and new API for asynchronously receiving a `TextMessage` from a durable topic subscription in a Java EE (EJB or web container) environment.

Here's how you might do this using the existing JMS 1.1 API.

```
public void receiveMessagesOld() throws JMSException, NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Topic inboundTopic = (Topic) initialContext.lookup("jms/inboundTopic");

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   session.createDurableSubscriber(inboundTopic, "");
   TopicSubscriber topicSubscriber =
      session.createDurableSubscriber(inboundTopic, "mysub");
   MessageListener messageListener = new MyListener();
   topicSubscriber.setMessageListener(messageListener);

   connection.start();

   // wait for messages to be received - details omitted
   connection.close();
}
```

Here's how you might do this using the new API.

```
public void receiveMessagesNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Topic inboundTopic = (Topic) initialContext.lookup("jms/inboundTopic");

   try (MessagingContext context =
          connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);){
      MessageListener messageListener = new MyListener();
      context.setMessageListener(inboundTopic, "mysub", messageListener);

      // wait for messages to be received - details omitted
   }
}
```

Note that `receiveMessagesNew` does not need to throw `JMSException`.

Receiving a message in multiple threads (Java SE)

This example compares the old and new API for asynchronously receiving `TextMessage` objects from a queue using multiple threads in a Java SE environment. In this example two threads are used, which means two sessions are needed. In this example, both sessions use the same connection.

Here's how you might do this using the existing JMS 1.1 API.

```
public void receiveMessagesOld() throws JMSException, NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   Connection connection = connectionFactory.createConnection();

   Session s1 = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageConsumer messageConsumer1 = s1.createConsumer(inboundQueue);
   MyListener messageListener1 = new MyListener("One");
   messageConsumer1.setMessageListener(messageListener1);

   Session s2 = connection.createSession(false,AUTO_ACKNOWLEDGE);
   MessageConsumer messageConsumer2 = s2.createConsumer(inboundQueue);
   MyListener messageListener2 = new MyListener("One");
   messageConsumer2.setMessageListener(messageListener2);
   connection.start();

   // wait for messages to be received - details omitted

   connection.close();
}
```

Here's how you might do this using the new API:

```
public void receiveMessagesNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext context1 =
           connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);
        MessagingContext context2 =
           context1.createMessagingContext(AUTO_ACKNOWLEDGE);){
      MyListener messageListener1 = new MyListener("One");
      context1.setMessageListener(inboundQueue,messageListener1);

      MyListener messageListener2 = new MyListener("Two");
      context2.setMessageListener(inboundQueue,messageListener2);

      // wait for messages to be received - details omitted

   }
}
```

Receiving synchronously and sending a message in the same local transaction (Java SE)

This example considers the use case in which a Java SE application repeatedly consumes a message from one queue and forwards it to another queue. Each message is received and forwarded in the same local transaction. This means that the receiving and sending of the message must be done using the same transacted Session which is then committed.

In this example the application consumes the incoming messages synchronously. However since this is Java SE the message could also be consumed asynchronously using a `MessageListener`.

Here's how you might do this using the existing JMS 1.1 API.

```
public void receiveAndSendMessageOld() throws JMSException, NamingException {
   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");
   Queue outboundQueue = (Queue) initialContext.lookup("jms/outboundQueue");

   Connection connection  = connectionFactory.createConnection();
   Session session = connection.createSession(true,SESSION_TRANSACTED);
   MessageConsumer messageConsumer = session.createConsumer(inboundQueue);
   MessageProducer messageProducer = session.createProducer(outboundQueue);
   connection.start();

   TextMessage textMessage = null;
   do {
      textMessage = (TextMessage) messageConsumer.receive(1000);
      if (textMessage!=null){
         messageProducer.send(textMessage);
         session.commit();
      }
   } while (textMessage!=null);
   connection.close();
}
```

Here's how the same example might look when using the new API:

```
public void receiveAndSendMessageNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");
   Queue outboundQueue = (Queue) initialContext.lookup("jms/outboundQueue");

   try (MessagingContext context =
          connectionFactory.createMessagingContext(SESSION_TRANSACTED);){
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncConsumer(inboundQueue);
      TextMessage textMessage = null;
      do {
         textMessage = (TextMessage) syncMessageConsumer.receive(1000);
         if (textMessage != null) {
            context.send(outboundQueue, textMessage);
            context.commit();
         }
      } while (textMessage != null);
   }
}
```

Note that receiveAndSendMessageNew does not need to throw `JMSException`.

Request/reply pattern using a TemporaryQueue (Java EE)

This example considers how a request/reply pattern might be implemented in Java EE, using the existing and new JMS APIs. In the code below, a method in a session bean (the requestor) sends a request message to some queue (the request queue). The `setJMSReplyTo` property of the request message is set to a TemporaryQueue, to which the reply should be set. After sending the request, the session bean listens on the temporary queue until it receives the reply.

Since the request message won't actually be sent until the transaction is committed, the request message is sent in a separate transaction from that used to receive the reply.

Here's how you might implement the requestor this using the existing JMS 1.1 API.

When implementing this pattern, the following features of JMS must be borne in mind:

- The same `Connection` object that was used to create the `TemporaryQueue` must also be used to consume the response message from it. (This is a restriction of temporary queues).
- If the request message is sent in a transaction then the response message must be consumed in a separate transaction. That's why the message is sent in a separate business which has the transactional attribute `REQUIRES_NEW`.

The details of the responder are omitted here. Typically this will be a MDB which receives the request message, extracts the TemporaryQueue from the `setJMSReplyTo` property and sends the response to it.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/requestQueue")
Queue requestQueue;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public String requestReplyOld(String request) throws JMSException {

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   TemporaryQueue temporaryReplyQueue = session.createTemporaryQueue();

   // send request in a separate transaction
   sendRequestOld(request,temporaryReplyQueue);

   // now receive the reply,
   // using the same connection as was used to create the temporary reply queue
   SyncMessageConsumer syncMessageConsumer =
      session.createSyncConsumer(temporaryReplyQueue);
   connection.start();
   TextMessage reply = (TextMessage) syncMessageConsumer.receive();
   String replyString=reply.getText();
   connection.close();
   return replyString;
}

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void sendRequestOld(
               String requestString, TemporaryQueue temporaryReplyQueue)
               throws JMSException {

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,AUTO_ACKNOWLEDGE);
   TextMessage requestMessage = session.createTextMessage(requestString);
   requestMessage.setJMSReplyTo(temporaryReplyQueue);
   MessageProducer messageProducer = session.createProducer(requestQueue);
   messageProducer.send(requestMessage);
   connection.close();
}
```

Here’s how the same example might look when using the new API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/requestQueue")
Queue requestQueue;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public String requestReplyNew(String request) throws JMSException {

   String replyString = null;
   try (MessagingContext context = connectionFactory.createMessagingContext();) {
      TemporaryQueue temporaryReplyQueue = context.createTemporaryQueue();

      // send request in a separate transaction
      sendRequestNew(request, temporaryReplyQueue);

      // now receive the reply, using the same connection as was used
      // to create the temporary reply queue
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncConsumer(temporaryReplyQueue);
      replyString = syncMessageConsumer.receivePayload(String.class);
   }
   return replyString;
}

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void sendRequestNew(
                requestString, TemporaryQueue temporaryReplyQueue)
                throws JMSException {

   try (MessagingContext context = connectionFactory.createMessagingContext();) {
      TextMessage requestMessage = context.createTextMessage(requestString);
      requestMessage.setJMSReplyTo(temporaryReplyQueue);
      context.send(requestQueue, requestMessage);
   }
}
```

Note that `requestReplyNew` and `sendRequestNew` continue to throw `JMSException` because `sendRequestNew` uses requestMessage.setJMSReplyTo`()`from the old API which continue to throw `JMSException`.

The simplified API in detail

javax.jms.MessagingContext (new)

Methods on MessagingContext	**Details**
New method which creates a second `MessagingContext` object from an existing `MessagingContext`, using the	`MessagingContext createMessagingContext(int sessionMode) throws JMSException;`

Methods on MessagingContext	**Details**
same connection. This is for use in Java SE applications when the application developer wishes to create multiple messaging contexts (i.e. sessions) on a connection.	

All methods have been copied from Connection except for:
 	
• Creating a Session
 	
• Chapter 8 interfaces
	Methods copied from Connection:
	
	`String getClientID() throws JMSException;`
	
	`void setClientID(String clientID) throws JMSException;`
	
	`ConnectionMetaData getMetaData() throws JMSException;`
	
	`ExceptionListener getExceptionListener() throws JMSException;`
	
	`void setExceptionListener(ExceptionListener listener) throws JMSException;`
	
	`void start() throws JMSException;`
	
	`void stop() throws JMSException;`
	
	`void close() throws JMSException;`
	
	Methods NOT copied from Connection:
	
	`Session createSession(boolean transacted, int acknowledgeMode) throws JMSException;`
	
	`ConnectionConsumer createConnectionConsumer(Destination destination, String messageSelector, ServerSessionPool sessionPool, int maxMessages) throws JMSException;`
	
	`ConnectionConsumer createDurableConnectionConsumer(Topic topic, String subscriptionName, String messageSelector, ServerSessionPool sessionPool, int maxMessages) throws JMSException;`

All methods have been copied from Session except for:
 	
• Creating a MessageProducer
 	
• Chapter 8 interfaces
	Methods copied from Session:
	
	`static final int AUTO_ACKNOWLEDGE = 1;`
	`static final int CLIENT_ACKNOWLEDGE = 2;`
	`static final int DUPS_OK_ACKNOWLEDGE = 3;`
	`static final int SESSION_TRANSACTED = 0;`
	
	`BytesMessage createBytesMessage() throws`

<table>
<tr><th>Methods on MessagingContext</th><th>Details</th></tr>
<tr><td>The createConsumer methods which return a MessageConsumer have been renamed createSyncConsumer and return a SyncMessageConsumer

The two createDurableSubscriber methods have been renamed createSyncDurableSubscriber and return a SyncMessageConsumer

The method <code>getAcknowledgeMode()</code> has been renamed <code>getSessionMode()</code></td><td><code>JMSException;</code>

<code>MapMessage createMapMessage() throws JMSException;</code>

<code>Message createMessage() throws JMSException;</code>

<code>ObjectMessage createObjectMessage() throws JMSException;</code>

<code>ObjectMessage createObjectMessage(Serializable object) throws JMSException;</code>

<code>StreamMessage createStreamMessage() throws JMSException;</code>

<code>TextMessage createTextMessage() throws JMSException;</code>

<code>TextMessage createTextMessage(String text) throws JMSException;</code>

<code>boolean getTransacted() throws JMSException;</code>

<code>int getAcknowledgeMode() throws JMSException;</code>

<code>void commit() throws JMSException;</code>

<code>void rollback() throws JMSException;</code>

<code>void recover() throws JMSException;</code>

<code>MessageConsumer createConsumer(Destination destination) throws JMSException;</code>

<code>SyncMessageConsumer createConsumer(Destination destination, java.lang.String messageSelector) throws JMSException;</code>

<code>SyncMessageConsumer createConsumer(Destination destination, java.lang.String messageSelector, boolean NoLocal)   throws JMSException;</code>

<code>Queue createQueue(String queueName) throws JMSException;</code>

<code>Topic createTopic(String topicName) throws JMSException;</code>

<code>TopicSubscriber createDurableSubscriber(Topic topic, String name) throws JMSException;</code>

<code>TopicSubscriber createDurableSubscriber(Topic topic, String name, String messageSelector, boolean noLocal) throws JMSException;</code>

<code>QueueBrowser createBrowser(Queue queue) throws JMSException;</code>

<code>QueueBrowser createBrowser(Queue queue, String</code></td></tr>
</table>

Methods on MessagingContext	Details

	`messageSelector) throws JMSException;`
	
	`TemporaryQueue createTemporaryQueue() throws JMSException;`
	
	`TemporaryTopic createTemporaryTopic() throws JMSException;`
	
	`void unsubscribe(String name) throws JMSException;`
	
	Methods NOT copied from Session:
	
	`MessageListener getMessageListener() throws JMSException;`
	
	`void setMessageListener(MessageListener listener) throws JMSException;`
	
	`public void run();`
	
	`MessageProducer createProducer(Destination destination) throws JMSException;`

All methods have been copied from MessageProducer except those which assume a destination to have been set when the producer was created.	**Methods copied from MessageProducer:**
	
	`void setDisableMessageID(boolean value) throws JMSException;`
	
	`boolean getDisableMessageID() throws JMSException;`
	
	`void setDisableMessageTimestamp(boolean value) throws JMSException;`
	
	`boolean getDisableMessageTimestamp() throws JMSException;`
	
	`void setDeliveryMode(int deliveryMode) throws JMSException;`
	
	`int getDeliveryMode() throws JMSException;`
	
	`void setPriority(int defaultPriority) throws JMSException;`
	
	`int getPriority() throws JMSException;`
	
	`void setTimeToLive(long timeToLive) throws JMSException;`
	
	`long getTimeToLive() throws JMSException;`
	
	`void send(Destination destination, Message message) throws JMSException;`
	
	`void send(Destination destination, Message message, int deliveryMode, int priority, long timeToLive) throws JMSException;`

Methods on MessagingContext	Details

	Methods NOT copied from MessageProducer:
	`Destination getDestination() throws JMSException;`
	`void send(Message message) throws JMSException;`
	`void send(Message message, int deliveryMode, int priority, long timeToLive) throws JMSException;`

Three new methods have been added for consuming messages asynchronously. They are based on the three methods on `Session` for creating a `MessageConsumer`, with an additional argument, the `MessageListener`.	`void setMessageListener(Destination destination, MessageListener listener) throws JMSException;`
	`void setMessageListener(Destination destination, String messageSelector, MessageListener listener) throws JMSException;`
	`void setMessageListener(Destination destination, String messageSelector, boolean NoLocal, MessageListener listener) throws JMSException;`

Two new methods have been added for consuming messages asynchronously from a durable subscription. They are based on the two methods on `Session` for creating a durable `TopicSubscriber`, with an additional argument, the `MessageListener`.	`void setMessageListener (Topic topic, String subscriptionName, MessageListener listener) throws JMSException;` `void setMessageListener (Topic topic, String subscriptionName, String messageSelector, boolean noLocal, MessageListener listener) throws JMSException;`
Two new methods have been added which allow a `TextMessage` or `ObjectMessage` to be sent by supplying the payload directly.	`void send(Destination destination, String payload) throws JMSException;` `void send(Destination destination, Serializable payload) throws JMSException;`
New method `acknowledge to` acknowledge all messages delivered by the session.	`void acknowledge() throws JMSException;`
New method `subscribe to` create a durable subscription but not a consumer on it.	`void subscribe(Topic topic, String name, String messageSelector, boolean noLocal);`

javax.jms.ConnectionFactory (modified)

Methods on ConnectionFactory	Details
Four new methods have been added which create `MessagingContext` objects.	These are for use in Java EE applications and also for Java SE applications when the application developer wishes to create only one messaging context (i.e. session) on a connection.

Four new methods have been added which create `MessagingContext` objects.
 	
These are for use in Java EE applications and also for Java SE applications when the application developer wishes to create only one messaging context (i.e. session) on a connection.
	`MessagingContext createMessagingContext() throws JMSException;`
	
	`MessagingContext createMessagingContext(String userName, String password) throws JMSException;`
	
	`MessagingContext createMessagingContext(String userName, String password, int sessionMode) throws JMSException;`
	
	`MessagingContext createMessagingContext(int sessionMode) throws JMSException;`

javax.jms.SyncConsumer (new)

Methods on SyncConsumer	Details

All methods have been copied from `javax.jms.MessageConsumer` **except** for those related specifically to async message consumption:	**Methods copied from MessageConsumer:**
	
	`String getMessageSelector() throws JMSException;`
	
	`Message receive() throws JMSException;`
	
	`Message receive(long timeout) throws JMSException;`
	
	`Message receiveNoWait() throws JMSException;`
	
	`void close() throws JMSException;`
	
	Messages NOT copied from MessageConsumer:
	
	`void setMessageListener(MessageListener listener) throws JMSException;`
	
	`MessageListener getMessageListener() throws JMSException;`

Three new methods have been added which allow a message payload to be returned directly.	`<T> T receivePayload(Class<T>  c);`
	
	`<T> T receivePayload(Class<T> c, long timeout);`

Methods on SyncConsumer	Details
	`<T> T receivePayloadNoWait(Class<T> c);`

javax.jms.JMSRuntimeException

This class is a copy of `javax.jms.JMSException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.JMSException`.

javax.jms.TransactionRolledBackRuntimeException

This class is a copy of `javax.jms.TransactionRolledBackException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.TransactionRolledBackException`.

javax.jms.IllegalStateRuntimeException

This class is a copy of `javax.jms.IllegalStateException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.IllegalStateException`.

javax.jms.InvalidDestinationRuntimeException

This class is a copy of `javax.jms.InvalidDestinationException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.InvalidDestinationException`.

javax.jms.InvalidSelectorRuntimeException

This class is a copy of `javax.jms.InvalidSelectorException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.InvalidSelectorException`

javax.jms.MessageFormatRuntimeException

This class is a copy of `javax.jms.MessageFormatException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.MessageFormatException`.

javax.jms.JMSSecurityRuntimeException

This class is a copy of `javax.jms.JMSSecurityException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.JMSSecurityException`.

javax.jms.InvalidClientException

This class is a copy of `javax.jms.JMSSecurityException` but changed to extend `java.lang.RuntimeException` and with an additional constructor which allows it to wrap a `javax.jms.JMSSecurityException`.

Updates in version 2

Version 1 contains the following changes:

Creating multiple MessagingContext objects with the same Connection

The API used by Java SE applications to create multiple `MessagingContext` objects with the same `Connection` has been changed.

- In version 1 it was proposed that the `Connection` interface should have a method `createMessagingContext(int sessionMode)`.
- In version 2 this method has been moved to the `MessagingContext` interface.

The example "Receiving a message in multiple threads (Java SE)" has been updated accordingly.

New method MessagingContext.acknowledge()

If a client uses the SyncMessageConsumer.receivePayload() methods to consume a message payload directly, they can't perform client acknowledgement because they don't have a Message object on which to call acknowledge().

To resolve this issue, a new method MessagingContext.acknowledge() has been added. Since calling Message.acknowledge() actually acknowledges all messages delivered by the Session this also reflects the actual behaviour better.

Threading restrictions on a Session

It has been pointed out that automatically starting a connection used by a MessagingContext as soon as setMessageListener is called (as explained in "No need to call connection.start()" above) means that if setMessageListener is called a second time, perhaps for a second destination, will break the threading restrictions on a session. That section has now been updated to state that the threading restrictions will be relaxed to permit this.

New method MessagingContext.subscribe()

Version 1 of this document observed the following "infelicity":

> *If you simply want to create a durable subscriber, but not actually consume messages from it, you now need to call* `createSyncDurableSubscriber`*, even if you intend to use async consumption in the future (previously you just called* `createDurableSubscription`*). This works but the method name is confusing when*

used for this purpose rather than to start sync consumption. Perhaps we need a void createDurableSubscription(...) method?

To resolve this a new method `subscribe` has been added to `MessagingContext`, and the note above has been removed.