JavaOne™

ORACLE®

50 JMS 2.0 Best Practices in 50 Minutes

CON3153

Nigel Deakin
Oracle
1 Oct 2014

Email: nigel.deakin@oracle.com
Twitter: @jms_spec

Getting the best out of JMS 2.0

About this session

- Introductory-level session
- Assumes little or no JMS knowledge
- Discusses only JMS 2.0
  - Though most of it applies to JMS 1.1 as well
- If you know JMS 1.1 this session will introduce you to JMS 2.0

JMS Best Practices

- Is JMS the best messaging API for your application?
- Use the best JMS API
- Use the right JMS features
- Use JMS in the easiest way
- Get the most out of JMS: advanced features

1. When to use JMS

- When you want to allow the sending and receiving of the message to occur at different times
- When you don't want sender and receiver to have to know about each other
- When sender and receiver are usually in separate running programs
- When you want pub/sub or point-to-point messaging
- When you don't want messages to be lost if there's a failure
- When you want your application to work with different messaging products

2. When not to use JMS

Other Java EE technologies may match your needs better

- Asynchronous EJB calls
  - A simpler alternative to sending a message to a MDB in the same app server
  - Less scalability, failure-tolerance etc
- CDI events
  - Simple Java EE observer/observable mechanism
  - Sender and receiver must be the same JVM
  - Sending method blocks whilst event is delivered to all listeners
  - If a listener throws an exception,
    - it is thrown directly to the sender
    - no more listeners receive the message (event)

JMS Best Practices

- Is JMS the best messaging API for your application?
- **Use the best JMS API**
- Use the right JMS features
- Use JMS in the easiest way
- Get the most out of JMS: advanced features

3. Use the right JMS API

ConnectionFactory
JMSContext
JMSProducer
JMSConsumer
Message

Simplified API

JMS 2.0+

ConnectionFactory
JMSContext
JMSProducer
JMSConsumer
Message

Classic API

JMS 1.1+

QueueConnectionFactory
QueueConnection
QueueSession
QueueSender
QueueReceiver
Message

Legacy API for point-to-point

JMS 1.0+

TopicConnectionFactory
TopicConnection
TopicSession
TopicPublisher
TopicSubscriber
Message

Legacy API for publish-and-subscribe

JMS 1.0+

4. Using a JMSContext to send a message to a queue (Java SE)

```
public void sendMessage(String text) throws NamingException {

    InitialContext namingContext = new InitialContext(props);

    ConnectionFactory cf = (ConnectionFactory) namingContext.lookup("jms/myCF");

    Queue messageQueue = (Queue)namingContext.lookup("jms/myQueue");

    try (JMSContext context=connectionFactory.createContext();){
        TextMessage message = context.createTextMessage(text);
        context.createProducer().send(dataQueue,message);
    }
}
```

5. Using a JMSContext to send a message to a queue (Java EE)

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

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

public void sendMessage (String text) {
    try (JMSContext context = connectionFactory.createContext();){
        TextMessage message = context.createTextMessage(text);
        context.createProducer().send(messageQueue,message);
    }
}
```

6. Getting your ConnectionFactory

Use JNDI to keep configuration details out of your application

- A ConnectionFactory is the starting object which contains all the information needed to connect to the JMS server
- The way you create and configure these objects is not standard, so create them separately, bind them in JNDI and look them up from your application
- In Java SE, use JNDI API to look up the connection factory

```
InitialContext ic = new InitialContext(props);
ConnectionFactory cf = (ConnectionFactory)ic.lookup("jms/myCF")
```

- In Java EE, inject the connection factory using @Resource

```
@Resource(lookup="jms/myCF") ConnectionFactory cf1;
@Resource ConnectionFactory cf2; // uses the platform default connection factory
```

7. Getting your Queue or Topic object

Use JNDI to keep configuration details out of your application

- A Queue or Topic object defines the queue or topic in the server where you want to send or receive messages
- The way you create and configure these objects is not standard, so create them separately, bind them in JNDI and look them up from your application
- In Java SE, use JNDI API to look up the queue or topic

```
InitialContext ic = new InitialContext(props);
javax.jms.Queue myQueue = (Queue)ic.lookup("jms/myQueue")
```

- In Java EE, inject the queue or topic using @Resource

```
@Resource(lookup="jms/myQueue") javax.jms.Queue myQueue;
```

8. Getting your JMSContext

Create it from the connection factory - and close it after use

- A JMSContext represents an active connection to the JMS server
- Create it from a ConnectionFactory...and close after use

```
JMSContext context = connectionFactory.createContext(sessionMode);
...
context.close
```

- For convenience, use a try-with-resources block

```
try (JMSContext context = connectionFactory.createContext();){
  ...
}
// close() called automatically at end of block or if exception thrown
```

9. Using a JMSContext to send a message

```
// start with a JMSContext
JMSContext context = ...

// ...and a queue or topic
Queue myQueue = ...

// create your message and set its payload
TextMessage myMessage = context.createTextMessage("Hello");

// send the message to the queue
context.createProducer().send(myQueue,myMessage);
```

10. Getting a JMSContext object by injection (Java EE only)

```
@Inject @JMSConnectionFactory("jms/connectionFactory") JMSContext context;

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

public void sendMessage (String text) {
    TextMessage message = context.createTextMessage(text);
    context.createProducer().send(messageQueue,message);
}

// context is automatically closed at the end of the Java EE transaction (if active)
// or else at the end of the "request" (e.g. remote EJB call or MDB invocation)
```

11. Receiving a message: sync or async?

Blocking call or asynchronous listener? Best practice depends on your needs.

- Blocking call
  - Fetch the next message from the queue or topic
  - Blocks until a message received
  - Best when you are expecting a particular message (e.g. a reply to a request)
- Asynchronous listener
  - JMS calls your code when the next message is available
  - Event-driven code is generally better
    - Particularly in a Java EE app server when you can process messages in multiple threads
  - Slightly less flexible in a Java EE app server

12. Receiving a message with a blocking call

```
// start with a JMSContext
JMSContext context = ...

// ...and a queue or topic
Queue myQueue = ...

// create a consumer on the queue
JMSConsumer consumer = context.createConsumer(messageQueue);

// fetch the message, blocking for up to 1000ms
Message message = consumer.receive(1000);

// extract the payload from the message
String payload = ((TextMessage)message).getText();

// close the JMSContext (if we've finished using it)
context.close();
```

13. Don't block for ever!

- `receive(timeout)`
  - best, can always repeat the call
- `receive()`
  - danger of blocking for ever
- `receiveNoWait()`
  - returns a message without blocking if one is immediately available
  - don't use, definition is ambiguous

14. Receiving messages asynchronously

Different in Java SE and Java EE

- Java SE:
  - Instantiate an object which implements the javax.jms.MessageListener interface
  - Register it with JMS by calling setMessageListener
- Java EE:
  - Use a message-driven bean instead
  - Calling setMessageListener is discouraged and not portable

15. Receiving messages asynchronously with a MessageListener (Java SE)

- Define the MessageListener class

```
public class MyListener
  implements javax.jms.MessageListener {

  public void onMessage(Message message) {
    TextMessage tm = (TextMessage)message;
    try {
      text = tm.getText();
      System.out.println("Got: "+text);
    } catch (JMSException e) {}

  }

}
```

- Start receiving messages

```
public void startListening() {
  ConnectionFactory cf = ...
  Queue messageQueue = ....
  context = cf.createContext();
  consumer =
    context.createConsumer(messageQueue);
  MyListener listener = new MyListener();
  consumer.setMessageListener(listener);
}
```

- Stop receiving messages

```
public void stopListening(){
  consumer.close();
}
```

16. Receiving messages asynchronously with a message-driven bean (Java EE)

```
@MessageDriven(activationConfig={
  @ActivationConfigProperty(
     propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
  @ActivationConfigProperty(
    propertyName = "destinationLookup", propertyValue = "jms/messageQueue")
})
public class MyMessageListener implements MessageListener {

  public void onMessage(Message message) {
    TextMessage textMessage = (TextMessage)message;
    try {
      String text = textMessage.getText();
      System.out.println("Received: "+text);
    } catch (JMSException e) {
      e.printStackTrace();
  }
}
```

17. Obey the threading restrictions

Or risk unpredictable behaviour

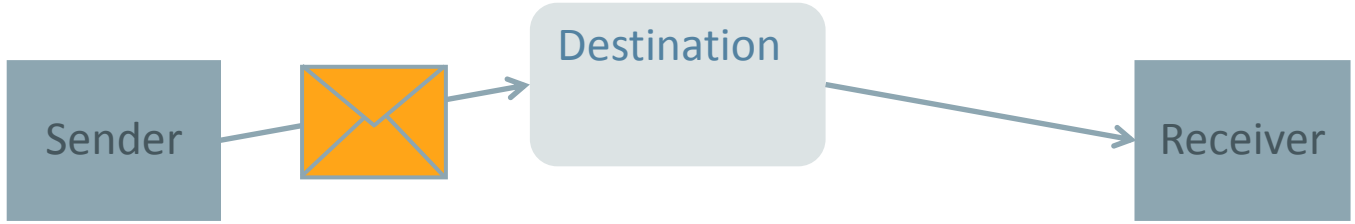
- ConnectionFactory, Queue, Topic (and Connection) support concurrent use
- JMSContext (and Session), *combined with any producer/consumer objects associated with it, and any received messages,* may be used by only one thread at a time
  - E.g. must not create two JMSProducer objects from the same JMSContext and have two concurrent threads using them to send messages at the same time.
  - Setting a MessageListener means the JMSContext is being used by the message delivery thread. No other thread may then use the JMSContext - other than to call setMessageListener again or to call close.
- If you want to use multiple threads, use multiple JMSContexts (or Sessions)

JMS Best Practices

- Is JMS the best messaging API for your application?
- Use the best JMS API
- **Use the right JMS features**
- Use JMS in the easiest way
- Get the most out of JMS: advanced features

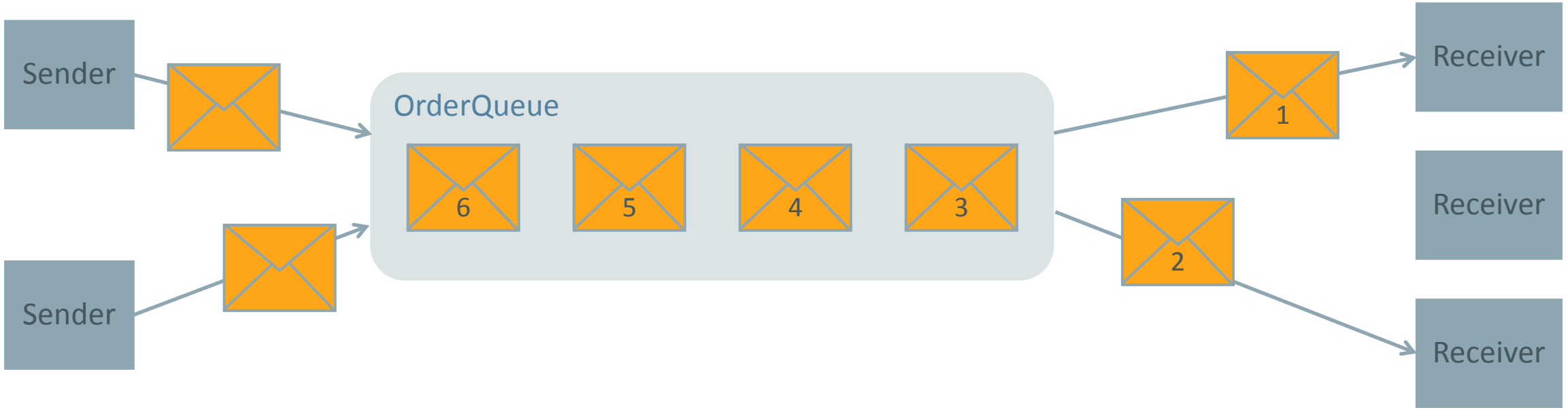
18: What is a JMS destination?

- JMS destinations are abstractions that mean senders and receivers don't need to know about each other
- Messages are sent to a named destination, not to a particular receiver
- Messages are received from a named destination, not from a particular sender
- Two types of JMS destination:
  - Queue
  - Topic
- What is the difference?



19. When to use a Queue

When you want each message to go to *one* of the receivers listening to the queue

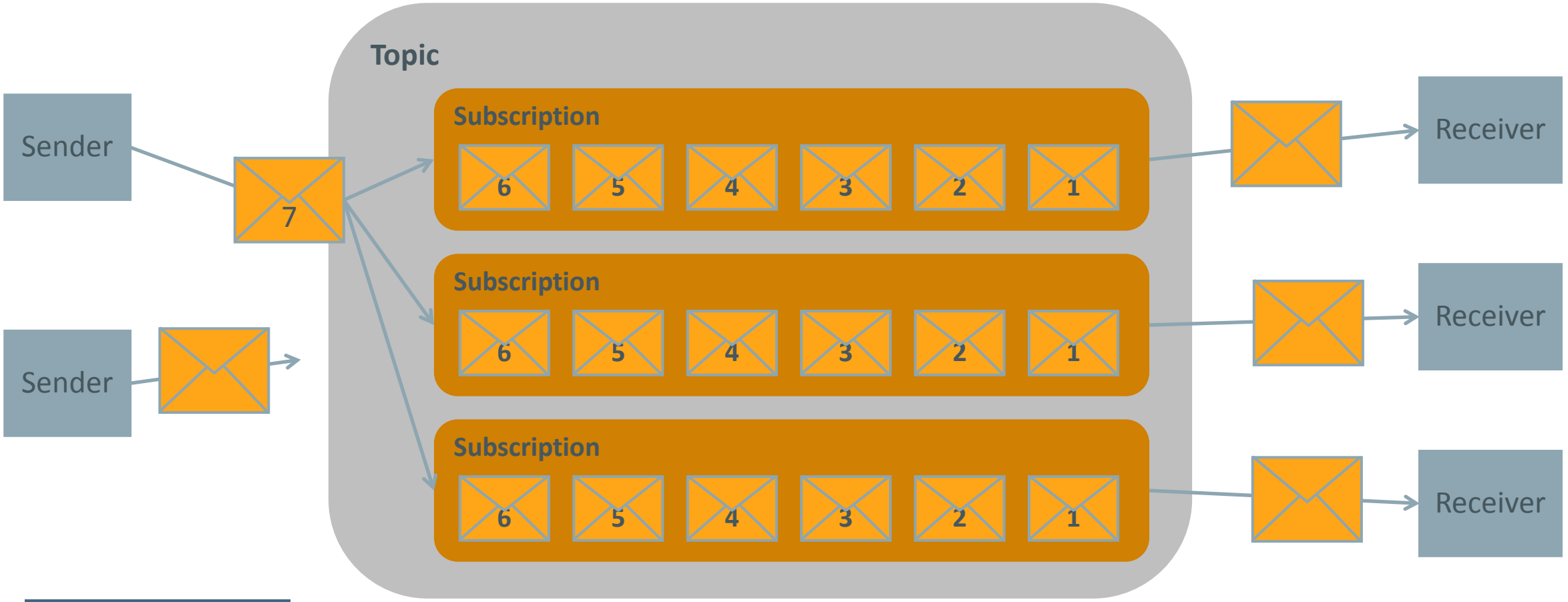


20. When to use a Topic

When you want message to go to *all* the receivers listening to the topic

21. Understanding topics

Messages are held in subscriptions



22. Durable topic subscriptions

- A durable subscription continues to exist even after consumer closes

```
// create a durable subscription on a topic
JMSConsumer consumer = context.createDurableConsumer(topic,"mySubscription");

// disconnect from the durable subscription, which continues to exist
consumer.close();
```

- A durable subscription remains in existence even after the consumer has closed. New messages will still be added to the durable subscription.
- If your application shuts down and is restarted later, you can catch all the messages sent to the topic whilst you were away
- You need to provide a name when you first create a durable subscription

23. Durable topic subscriptions with a message-driven bean (Java EE)

```
@MessageDriven(activationConfig={
  @ActivationConfigProperty(
     propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
  @ActivationConfigProperty(
    propertyName = "destinationLookup", propertyValue = "jms/myTopic"),
 @ActivationConfigProperty(
    propertyName = "subscriptionDurability", propertyValue = "Durable"),
 @ActivationConfigProperty(
    propertyName = " subscriptionName", propertyValue = "mySubscription")
})
public class MyMessageListener implements MessageListener {

  public void onMessage(Message message) {
    ...
  }
}
```

24. Non-durable topic subscriptions

- A *non-durable* subscription exists only as long as the consumer exists

```
// create a non-durable subscription
JMSConsumer consumer = context.createConsumer(topic);
consumer.setMessageListener(messageListener);

// receive messages from topic

...

// destroy the non-durable subscription
consumer.close();
```

- So if your application shuts down and is restarted later, you'll miss all the messages sent to the topic whilst you were away

25. Non-durable topic subscriptions with a message-driven bean (Java EE)

```
@MessageDriven(activationConfig={
  @ActivationConfigProperty(
     propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
  @ActivationConfigProperty(
    propertyName = "destinationLookup", propertyValue = "jms/myTopic"),
 @ActivationConfigProperty(
    propertyName = "subscriptionDurability", propertyValue = "NonDurable")
})
public class MyMessageListener implements MessageListener {

  public void onMessage(Message message) {
    ...
  }
}
```

26. Persistent or non-persistent messages?

Queues

- Messages are persistent by default
- Persistent messages sent to a queue will be saved on disk so that it won't be lost if the server is restarted

```
context.createProducer().send(messageQueue,message);
```

- Messages may be optionally configured to be non-persistent
  - usually much faster
  - message will be lost if the server is restarted
  - or if the server becomes short of memory or other resource limits are reached

```
context.createProducer().
  setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(messageQueue,message);
```

27. Persistent or non-persistent messages?

Topics

- For each durable subscription
  - Persistent messages are persisted
  - Non-persistent messages not persisted (and may also be thrown away if consumer disconnects)
- For each non-durable subscription on the topic
  - Messages are not persisted (even messages sent as "persistent").

	Persistent message	**Non-persistent message**
Durable subscription	Message is persisted	Message is not persisted
Non-durable subscription	Message is not persisted	Message is not persisted

28. Persistent or non-persistent messages?

Gotcha

```
// send a non-persistent message using the simplified API
context.createProducer().setDeliveryMode(NON_PERSISTENT).send(messageQueue,message);
```

```
// Warning: this does not do what you might expect!
message.setJMSDeliveryMode(NON_PERSISTENT)
context.createProducer().send(messageQueue,message);
```

29. Which message type to use

Four principal message types

```
// TextMessage
// Contains a single string (e.g. XML)
String stockData = ...
TextMessage message = context.createTextMessage();
message.setText(stockData);

// BytesMessage
// Contains any bytes
byte[] stockData;
BytesMessage message = context.createBytesMessage();
message.writeBytes(stockData);

// ObjectMessage
// Contains a single object that implements java.io.Serializable
ObjectMessage message = context.createObjectMessage();
message.setObject(stockObject);

// Message
// Contains no body (only headers and properties)
Message message = context.createMessage();
```

30. Which message type to use

Two more esoteric message types

```
// StreamMessage
// Contents must be written and read in order
StreamMessage message = context.createStreamMessage();
message.writeString(stockName);  // name of stock
message.writeDouble(stockValue); // current value of stock
message.writeLong(stockTime);    // time stock info was updated
message.writeDouble(stockDiff);  // price change
message.writeString(stockInfo);  // info on this stock

// MapMessage
// Contents may be written and read in any order
MapMessage message = context.createMapMessage();
message.setString("Name", "ORCL");
message.setDouble("Value", stockValue);
message.setLong("Time", stockTime);
message.setDouble("Diff", stockDiff);
message.setString("Info", "Recent server announcement causes market interest");
```

31. Message acknowledgement

- After you've received a message, the server needs to know you've got it
- If the message is not acknowledged, the server will think that something went wrong and will send it again later
- JMS offers several ways of acknowledging messages
- Type of acknowledgement required is specified by calling `createContext(sessionMode)`

32. Automatic acknowledgement

- AUTO_ACKNOWLEDGE - a good choice if you aren't using a transaction
  - message is acknowledged before `receive` returns a message to the application
  - and after the application returns from `onMessage`

```
JMSContext context2 = connectionFactory.createContext(AUTO_ACKNOWLEDGE);
JMSContext context1 = connectionFactory.createContext(); // uses AUTO_ACKNOWLEDGE
```

- DUPS_OK_ACKNOWLEDGE - potentially the fastest mode
  - acknowledgement is automatic but may be deferred
  - may get duplicate messages if your application fails and is restarted

```
JMSContext context3 = connectionFactory.createContext(DUPS_OK_ACKNOWLEDGE);
```

33. Client acknowledgement

- Gives you control over acknowledgement without using a transaction
  - allows you to defer acknowledgement to improve performance
  - may get duplicate messages if your application fails and is restarted
- Specify client-acknowledgement when you create the JMSContext

```
JMSContext context2 = connectionFactory.createContext(CLIENT_ACKNOWLEDGE);
```

- Call acknowledge() to acknowledge all received messages

```
// acknowledge this message (and any other messages received from the same JMSContext)
message.acknowledge()
```

- Use in Java SE applications
- Avoid in Java EE applications - use global (XA) transactions instead

34. Local transactions

```
JMSContext context = connectionFactory.createContext(SESSION_TRANSACTED);

// now receive a message (could also use a listener)
JMSConsumer consumer = context.createConsumer(someQueue);
Message message = consumer.receive(timeout);

// we can process the received message,
// but message is not acknowledged until the transaction is committed

// now send a message
context.createProducer().send(messageQueue,message);

// message is not actually sent until the transaction is committed

context.commit();
```

- Use in Java SE applications
- Avoid in Java EE applications - use global (XA) transactions instead

35. Global (XA) transactions

- Java EE only
- Allow multiple transactional resources to be committed in the same transaction
- Receive a message and update a database in the same transaction
- Java EE supports two types of transaction management
  - Container-managed transactions
  - Bean-managed transactions

36. Global (XA) transactions (Java EE)

Container-managed transactions with EJBs

```
// EJBs use container-managed transactions by default
@Stateless
public class MyCMTBean {

    @Inject @JMSConnectionFactory("jms.MyXACF1") JMSContext context;
    @Resource(lookup = "TestQueue") Queue queue;

    public String myMethod() throws Exception {
        context.createProducer().send(queue, "Some text");
    } // transaction is committed automatically
}
```

37. Global (XA) transactions (Java EE)

Container-managed transactions with servlets and CDI managed beans

```
@WebServlet(name="myServlet", urlPatterns={"/myservlet"})
public class MyServlet extends HttpServlet {

    @Inject @JMSConnectionFactory("jms.MyXACF1") JMSContext context;
    @Resource(lookup = "TestQueue") Queue queue;

    @Transactional
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String text = request.getParameter("text");
        context.createProducer().send(queue, text);
    } // transaction is committed automatically
}
```

38. Global (XA) transactions (Java EE)

Managing the transaction yourself

```
@WebServlet(name="myServlet", urlPatterns={"/myservlet"})
public class MyServlet extends HttpServlet {

    @Inject @JMSConnectionFactory("jms.MyXACF1") JMSContext context;
    @Resource(lookup = "TestQueue") Queue queue;
    @Resource UserTransaction ut;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String text = request.getParameter("text");
        ut.begin();
        context.createProducer().send(queue, text);
        ut.commit();
    }
}
```

39. Message properties

Adding filterable metadata to your messages

- Properties can be boolean, byte, short, int, long, float, double, and String.

```
setBooleanProperty, setByteProperty, setShortProperty,setLongProperty,
setFloatProperty, setDoubleProperty, setStringProperty
```

- Set a property directly on the message object

```
textMessage.setStringProperty("StockSector","Tech");
context.createProducer().send(topic,textMessage)
```

- or on the JMSProducer object prior to sending

```
context.createProducer().
  setStringProperty("StockSector","Tech").send(topic,textMessage);
```

40. Message selectors

When you don't want every message

- Sender sets properties on the message

```
// create stock info message
String info = ...
TextMessage tm = context.createTextMessage();
tm.setText(info);

// set property to identify stock sector
tm.setStringProperty("StockSector","Tech");


// sent the message
context.send(topic,stockData);
```

- Consumer uses message selector to receive only messages with certain message property values

```
String selector = "(StockSector='Tech')");
JMSConsumer consumer =
  context.createConsumer(topic,selector);
consumer.setMessageListener(listener);
```

- Message selector expressions can filter on property values and on message headers (e.g. JMSPriority)

JMS Best Practices

- Is JMS the best messaging API for your application?
- Use the best JMS API
- Use the right JMS features
- **Use JMS in the easiest way**
- Get the most out of JMS: advanced features

41. Shortcuts for sending messages

Send the message body directly

```
// send a TextMessage
context.createProducer().send(queue,"hello");

// send a MapMessage
Map mapData = .....;
context.createProducer().send(queue,mapData);

// send an ObjectMessage
Serializable object = .....;
context.createProducer().send(queue,object);

// send a BytesMessage
byte[] bytes = .....;
context.createProducer().setStringProperty("foo","bar").send(queue,bytes);
```

42. Using the JMSProducer to configure send parameters

Use the builder pattern

- JMSProducer is a lightweight object used to hold "send configurations"
- Use setter methods to configure
  - persistent/non-persistent (delivery mode), time to live, delivery delay
  - message headers (alternative to setting on message)
  - message properties (alternative to setting on message)
- Setter methods all return the JMSProducer object.
  - allows method calls to be chained together

```
context.createProducer().setProperty("foo","bar").
   setTimeToLive(10000).setDeliveryMode(NON_PERSISTENT).send(dataQueue, body);
```

43. Java EE 7 queue and topic definitions

Non-portable resource definitions for rapid prototyping and development

- Create a queue or topic in JNDI
- **Either** using annotations in your code

```
@JMSDestinationDefinition(
  interfaceName = "javax.jms.Queue",
  name = "java:global/MyTestQueue",
  destinationName ="MyQueue")
```

- Use `@JMSDestinationDefinitions` to wrap multiple `@JMSDestinationDefinitions` elements
- **Or** using XML in your deployment descriptor

```
<jms-destination>
  <interface-name>
    javax.jms.Queue
  </interface-name>
  <name>
    java:global/MyTestQueue
  </name>
  <destination-name>
    MyQueue
  </destination-name>
</jms-destination>
```

44. Java EE 7 connection factory definitions

Non-portable resource definitions for rapid prototyping and development

- Create a JMS connection factory in JNDI using annotations in your code

```
@JMSConnectionfactoryDefinition(
  name = "java:global/MyTestCF",
  properties = {
    "addressList=mq://localhost:7676"
  })
```

- Use `@JMSConnectionFactoryDefinitions` to wrap multiple `@JMSConnectionFactoryDefinition` elements

- Create a JMS connection factory in JNDI using XML in your deployment descriptor

```
<jms-connection-factory>
  <name>
    java:global/MyTestCF
  </name>
  <property>
    <name>addressList</name>
    <value>mq://localhost:7676</value>
  </property>
</jms-connection-factory>
```

JMS Best Practices

- Is JMS the best messaging API for your application?
- Use the best JMS API
- Use the right JMS features
- Use JMS in the easiest way
- Get the most out of JMS: advanced features

45. Message redelivery

In the real world, something things go wrong

- Messages that are delivered but not acknowledged (e.g. if a message listener throws an exception) will be delivered again
- Exactly when a message is redelivered depends on the type of acknowledgment being used
- When a message is redelivered,
  - `message.getJMSRedelivered()` will return true
  - `message.getIntProperty("JMSXDeliveryCount")` will return how many times message has been redelivered
- Design your code to detect messages being redelivered repeatedly

46. Temporary queues and topics

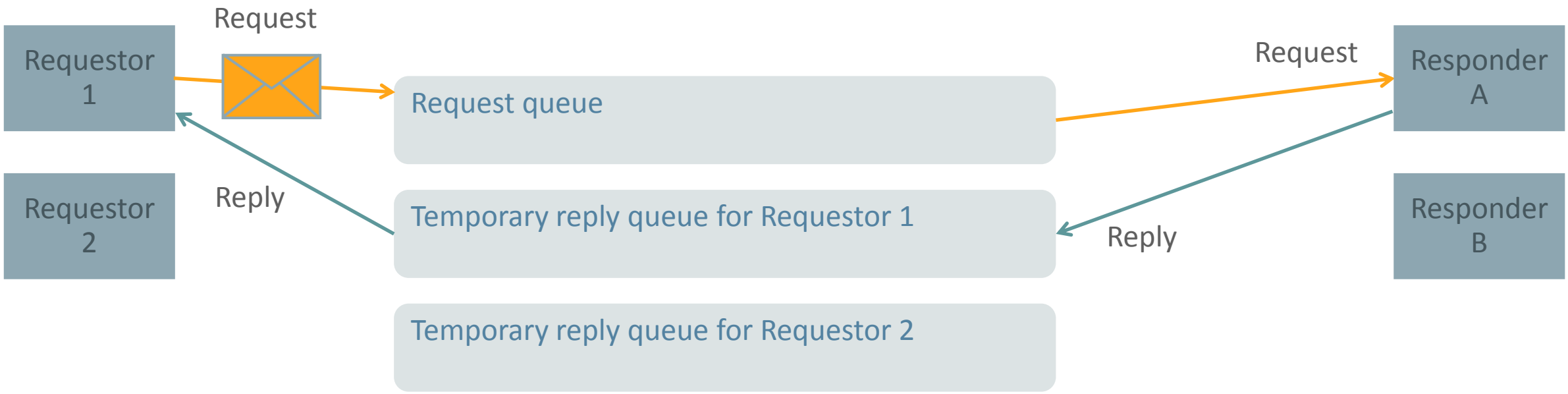
- You can create a temporary queue or topic on-the-fly in your code

```
Queue tempQueue = context.createTemporaryQueue();
Topic tempTopic = context.createTemporaryTopic();
```

- Anyone can send to a temporary queue or topic
- Only the JMSContext (Connection) that created it can receive messages
  - so temporary queues are more useful than temporary topics
- When the connection that created it is closed, the temporary queue or topic is automatically destroyed
- Use for request-reply messaging

47a. Request-reply messaging

The main reason for using temporary queues



47b. Request-reply messaging

- Requestor:
  - creates temporary queue using `context.createTemporaryQueue()`
  - creates request message and saves name of temporary queue in it using `message.setJMSReplyTo(queue)`
  - sends message to request queue (which is a normal permanent queue)
  - waits for a reply message from the temporary reply queue
  - Reuse the temporary queue, and the consumer on it, for subsequent requests (optional)
- Responder:
  - receives request message from request queue
  - calls `message.getJMSReplyTo()` to obtain temporary reply queue
  - creates reply message
  - sends reply message to temporary reply queue

48. Understanding message order

- Messages are delivered in the same order in which they are sent
  - for the same sending JMSContext (Session), receiving JMSContext , and queue/topic
  - **no requirement** to preserve relative order of messages sent by different JMSContext s

- Exceptions
  - High priority messages can overtake lower priority messages
  - Non-persistent messages may overtake persistent messages
  - Setting delivery delay may mean messages are delivered in a different order

49. Message priority

When some messages are more urgent than others

- A message can be given an integer priority value in the range 0-9
  - 0 is lowest priority, 4 is default priority, 9 is highest priority
- JMS provider should "do its best" to deliver higher priority messages before lower priority messages - but don't rely on it
- Set message priority on the JMSProducer prior to calling send

```
context.createProducer().setPriority(5).send(destination,message);
```

- Don't try to set the priority directly on the message

```
// This does not do what you might expect
message.setJMSPriority(5);
context.createProducer().send(destination,message)
```

50. Message delivery delay

When you want your message to be delivered later

- Set delivery delay on the JMSProducer prior to calling send

```
long delay = 12*60*60*1000; // 12 hours in milliseconds
context.createProducer().setDeliveryDelay(delay).send(destination,message);
```

- Message will be added to queue or topic immediately
- Message will not be delivered to a consumer until at least 12 hours after time it was sent

Topics we didn't mention

- Shared subscriptions
- Async send
- Exception handling
- Message expiration
- Security
- Using a QueueBrowser to browse queues (sorry, can't browse topics)
- JMSConsumer#receiveBody
- Message#getBody

Any questions?

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

JavaOne™

ORACLE®

Shortcuts for receiving messages

Receive the message body directly

```
// next message is a TextMessage
String text = consumer.receiveBody(String.class);

// next message is a MapMessage, and we want to specify a receive timeout
Map map = consumer.receiveBody(Map.class, 1000);

// next message is an ObjectMessage which holds a MyObject
MyObject object = consumer.receiveBody(MyObject.class);

// next message is a BytesMessage
byte[] bytes = consumer.receiveBody(byte[].class);
```

Only useful if you know the expected message type, and you don't need to access message properties or headers

Shared subscriptions

- Topic subscriptions are normally unshared
  - You can have only one consumer for each subscription
  - Means only one thread can process messages
- Topic subscriptions may alternatively be shared

```
// create multiple consumers on the same shared non-durable subscription
JMSConsumer consumer1 = context1.createSharedConsumer(topic,sharedSubName)
JMSConsumer consumer2 = context2.createSharedConsumer(topic,sharedSubName)

// create multiple consumers on the same shared durable subscription
JMSConsumer consumer3 = context3.createSharedDurableConsumer(topic,sharedSubName)
JMSConsumer consumer4 = context4.createSharedDurableConsumer(topic,sharedSubName)
```

- Advanced feature for Java SE allowing higher scalability and throughput
- In Java EE, MDBs already provide a way to do this

51. Message expiration

When your message soon becomes worthless

- Set timeToLive on the JMSProducer before calling send

```
// stock data message will expire after 500ms
context.createProducer().setTimeToLive(500).send(topic,stockData);
```

- Message will be added to queue or topic immediately
- Message will not be delivered to consumers *after* time to live has expired
- Useful for time-sensitive data such as stock prices