Using Advanced Java Message Service (JMS) Features to Increase the Efficiency and Maintainability of Your Distributed Java Applications

Sabine Heider, Michael Kögel, and Radoslav Nikolov

Sabine Heider, Java Server Technology Group, SAP AG

Michael Kögel, Java Server Technology Group, SAP AG

Radoslav Nikolov, Java Server Technology Group, SAP Labs Bulgaria

(complete bios appear on page 108)

The Java Message Service (JMS) support included with SAP Web Application Server (SAP Web AS) Java[1] 6.20 and higher,[2] delivered by a built-in messaging server called the JMS provider, makes your application development life a whole lot easier. When faced with the task of enabling communication between components running in distributed landscapes, you are no longer limited to synchronously calling specific methods with Java Remote Method Invocation (RMI).[3] Using JMS, you can build applications out of loosely coupled components that exchange data asynchronously via messages. The main advantage of the JMS approach is that it decouples your application components, so unlike with RMI, not all parts have to be available at the same time. JMS also lets your application delegate tasks to be processed asynchronously, freeing it to do other processing or temporarily return control to the user while a lengthy process executes. For these reasons, asynchronous, message-based communication is useful not just for system-to-system applications, but also for building interactive applications. And as a set of standardized, predefined interfaces, JMS enables a platform-independent approach that makes your programs portable across different vendors' messaging servers.

[1] SAP Web AS can be deployed in three ways: with the Java/J2EE runtime (SAP Web AS Java), with the ABAP runtime (SAP Web AS ABAP), and with both the Java/J2EE and ABAP runtimes (SAP Web AS Java+ABAP).

[2] SAP Web AS 6.20 introduced support for the Java programming language and version 1.2 of the J2EE standard, plus support for JMS 1.0.2b. SAP Web AS 6.30/6.40 has even better scalability features, is fully compliant with J2EE 1.3, which includes support for JMS 1.0.2b, and is required for the example detailed in this article.

[3] Remote Method Invocation (RMI) is part of J2SE. For more information, visit **http://java.sun.com/products/jdk/rmi/**.

Key JMS Terms and Concepts in Review

Here we'll briefly recapitulate some of the key Java Message Service (JMS) terms and concepts we introduced in our previous article, "Building Flexible, Reliable, Distributed Java Applications with the Java Message Service (JMS) — An Introduction to JMS Programming" (*SAP Professional Journal*, July/August 2004).

JMS Provider, JMS Clients, and Messages

JMS is a standard service of the Java 2, Enterprise Edition (J2EE) platform, and provides a way for business applications to exchange data asynchronously (without being directly connected to each other) via **messages**. JMS defines a set of interfaces in the *javax.jms* package (included as part of the J2EE standard) that applications can use to send and receive messages to and from a **destination** on a **JMS messaging server** (also called a **JMS provider**). Java programs that send or receive messages using JMS are called **JMS clients**.* JMS messages can contain any kind of data, including plain text, HTML, XML, and images.

JMS Objects

In order to send or receive messages via JMS, a JMS client instantiates several **JMS objects**, including a connection, session, producer, or consumer object, for example. **Connection objects** and **session objects** represent just what their names imply. **Producer objects** are used to send messages to a destination on the JMS provider; **consumer objects** are used to receive messages from a destination.** To help you instantiate JMS objects in a vendor-neutral way, JMS defines a series of **factory** methods to instantiate the objects without having to specify their (vendor-specific) class names.*** The factory

* Programs written in other languages, like C++, can also send and receive messages via a JMS provider if the JMS server vendor offers them a (proprietary) programming interface. These clients are called "non-JMS clients."

** It is common to refer to the entire JMS client as a producer or a consumer, although it is actually a producer or consumer object that sends or receives the messages.

*** It may surprise you to learn that, under the hood, each of these objects are instances of classes that are specific to the vendor of the JMS provider (in our case, SAP). Your code stays vendor-neutral and portable, however, because these objects implement the standardized interfaces from the *javax.jms* package, so you can address the objects via the interface regardless of which vendor implemented them and of which type they actually are. Thus your JMS applications will work with any JMS server.

✔ ***Note!***

For a detailed introduction to JMS, see the article "Building Flexible, Reliable, Distributed Java Applications with the Java Message Service (JMS) — An Introduction to JMS Programming," published in the July/August 2004 issue of SAP Professional Journal. A brief review of the key JMS terms and concepts introduced in that article is available in the sidebar above.

methods are chained, so you use the connection object to create a session object, the session object to create a producer object, and so on.

Administered Objects

Some JMS objects — the connection factory used to start the instantiation of other objects, and the destination to which you send and from which you receive messages — must be instantiated in advance by an administrator and placed centrally in the application server so that the JMS clients can access them. Such objects are called **administered objects**. The clients use the standard Java Naming and Directory Interface (JNDI) service to retrieve references to administered objects from a directory service called the **JNDI provider** (the JNDI provider is part of every J2EE-compliant application server).

JMS Messaging Models

JMS provides two **messaging models** (also called **messaging domains**): **point-to-point (ptp)** and **publish-subscribe (pubsub)**. The ptp model is used for “one-to-one” message delivery; the pubsub model is used for “one-to-many” message delivery. Depending on whether you use ptp or pubsub, destinations, producers, and consumers each have more specific names. With ptp, destinations are called **queues**, producers are called **senders**, and consumers are called **receivers**. With pubsub, destinations are called **topics**, producers are called **publishers**, and consumers are called **subscribers**. These names are actually very logical given the scenario each model supports.

Temporary Destinations (Queues or Topics)

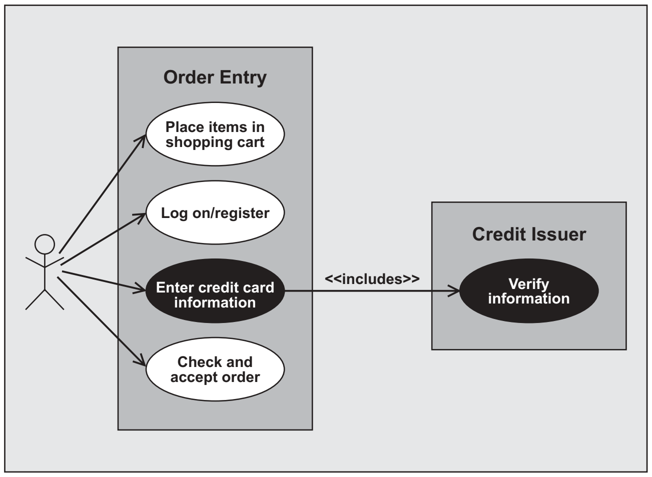
Just like regular destinations, **temporary destinations** represent a logical place to which a producer can send messages (and from which a consumer can receive messages). Unlike regular destinations, however, temporary destinations are not set up as administered objects; rather, they are created by the JMS client at runtime using a factory method of the session object. Temporary destinations exist only for the lifetime of the connection in which they are created. When the connection is closed, the temporary destination is deleted, and all messages inside the destination are lost. Also, only consumers created in the same connection as the destination can receive messages from it, although all producers can send messages to it. Temporary destinations are intended to facilitate a simple request/reply mechanism, like the one used in the example application outlined in our previous article.

In a previous article,[4] we developed a practical example using the JMS API to introduce you to the fundamentals of JMS programming and to show you how to get started building JMS-based applications quickly on your own. In this article, we will reuse the example to build on that foundational knowledge by extending it using some advanced JMS programming features. We will show you how to implement asynchronous message delivery to streamline your code, and how to use JMS message selectors to reduce application memory consumption.

Before we immerse ourselves in the programming details, however, let’s briefly get reacquainted with

[4] “Building Flexible, Reliable, Distributed Java Applications with the Java Message Service (JMS) — An Introduction to JMS Programming” (*SAP Professional Journal*, July/August 2004).

Figure 1 Use Case for the Sample Application



the example scenario introduced in the previous article,[5] and then take a closer look at where the coding can be improved, how we are going to improve it, and the benefits such improvements will provide to the example, and to your own JMS-based applications.

An Overview of the Sample Scenario

To demonstrate the use of the JMS API, in our previous article we developed the credit card validation process of an existing online shopping application.[6] The example consists of a frontend JSP/servlet application that lets a user enter a credit card number, and a backend Java application that simulates the credit card validation normally provided by an external service. **Figure 1** illustrates the major steps and components of the application scenario (the parts that are covered by our example are highlighted in black);

Figure 2 shows the interaction between the application components and the JMS provider within SAP Web AS. To summarize, the frontend payment processing servlet accepts the data entered by the user (the credit card number) and sends it to the backend validation component in a JMS message. The validation component checks the data and sends a new JMS message to the servlet with a positive or negative result, which the servlet then reports to the user.

✓ ***Note!***

Technically, the payment module sends a JMS message with the credit card data to a predefined queue (step ❷ in Figure 2).[7] Since the payment module expects a reply, it creates a temporary queue[8] (step ❶) and adds a reference to it in the reply-to destination of the message sent in step ❷. The credit card validation module receives the message using a synchronous "pull" approach (step ❸), validates the credit card information, and sends the result to the temporary queue in a new JMS message (step ❹). The servlet periodically checks[9] the temporary queue for new messages (step ❺), and deletes the queue upon successful receipt of the reply (step ❻).

5 The complete code for the example is available for download at **www.SAPpro.com**.

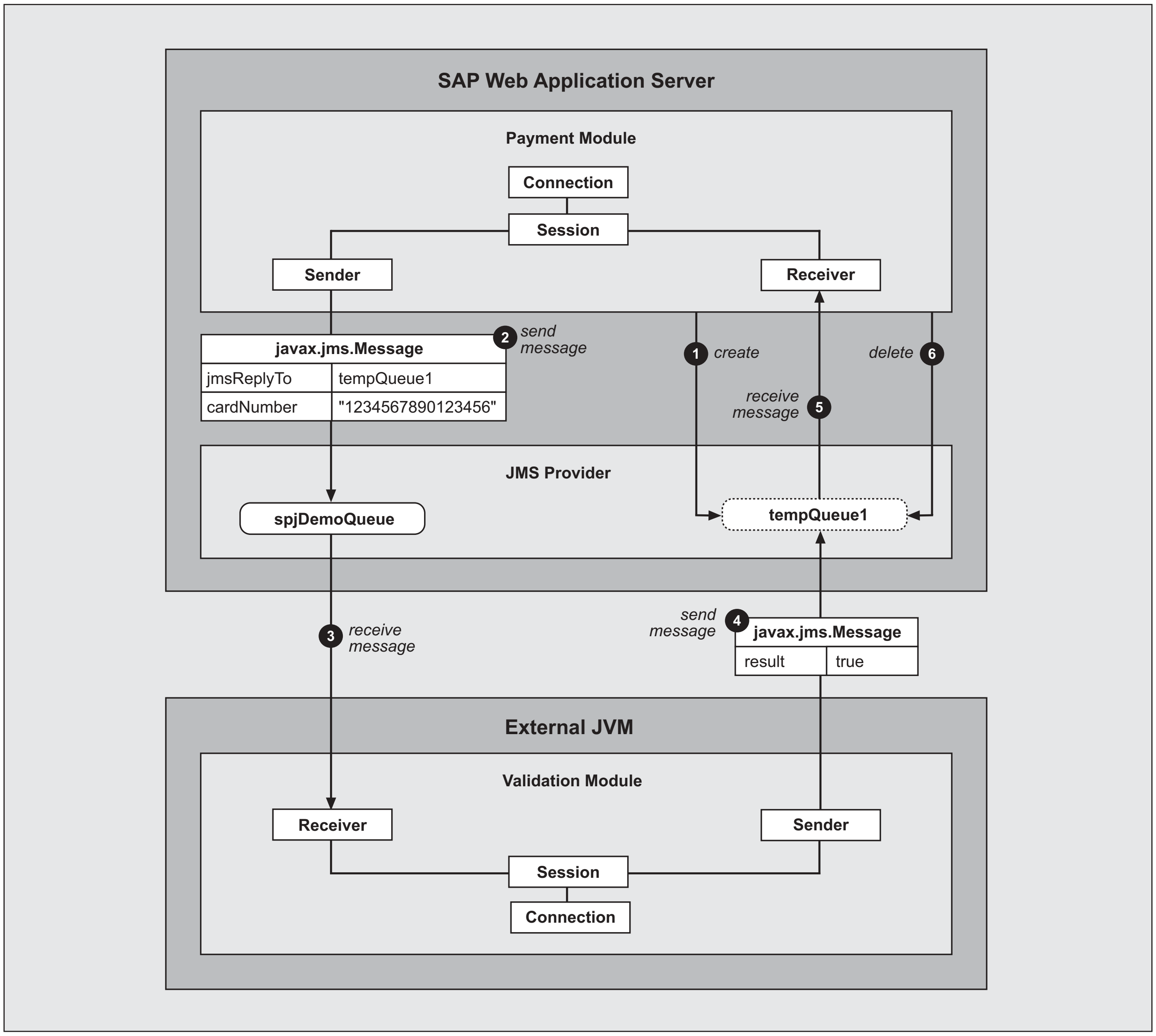
6 The example discussed in the article and included in the download does not include the online shop itself; it includes only the credit card verification component, which is where the JMS logic is used. The online shop referenced in the article is only intended to illustrate a scenario in which the demo could be embedded.

7 *spjDemoQueue* is an example of an object that has to be created in advance (e.g., by an administrator) for later use. At runtime, we retrieve a reference to the existing queue object using standard Java Naming and Directory Interface (JNDI) methods (rather than instantiating the object directly in the application). Objects defined in this way are called "administered objects."

8 Temporary queues are dynamically instantiated destinations to which messages can be sent and from which messages can be retrieved. For more details, refer to our previous article in the July/August 2004 issue of *SAP Professional Journal*, and also to the sidebar "Key JMS Terms and Concepts in Review" on page 78.

9 We coded the servlet to return a temporary "please wait" HTML page after sending the validation request, to give users a visual indicator of activity and to free resources by preventing the servlet from stalling as it waits for a reply. Before exiting, the servlet stores the temporary queue in the user's session memory. A metatag in the wait page causes the browser to periodically submit a "status check," which prompts the servlet to check the temporary queue for a reply message. If there is no reply, the servlet generates another wait page. Note that this page is available as part of the download at **www.SAPpro.com**.

Figure 2 JMS Design of the Application's Credit Card Verification Process



In the interest of simplicity, some components of the example application were implemented in a suboptimal way, however. Using some advanced JMS programming features, we can:

- Increase maintainability by streamlining the message exchange.
- Improve performance and scalability by reducing the number of temporary message queues.
- Improve scalability and leverage additional services by implementing the validation module as an Enterprise JavaBean (EJB).

Streamlining the Message Exchange

We implemented the validation module (which is essentially a server, since its only task is to process external messages sent to *spjDemoQueue*) in a simple, but not very elegant way: we use an endless loop to constantly ask the JMS provider to check for messages in *spjDemoQueue*. Inside the loop, we call a special version of the receive method of the JMS API that blocks execution — i.e., the calling program doesn't continue — until there is a message to deliver. A major disadvantage of this approach is that we cannot shut down the program properly. Unless it is processing a message, the program lies dormant waiting for a new message, so to shut it down we have to kill the entire Java process using operating system commands (or, alternatively, write code to spawn an additional thread for shutdown handling, which greatly increases the complexity of the program).

Fortunately, JMS provides a more elegant, easy-to-implement way to check for and receive incoming messages — **asynchronous message delivery**. Essentially, we create and register a "message listener" with the JMS provider that is notified as soon as a message arrives. As you will see later in the updated example, this will free our main program to do other tasks instead of just waiting for messages, and usually leads to much cleaner and more maintainable code.

Reducing the Number of Temporary Message Queues

The payment module creates a new temporary reply queue for each credit card number to be validated, and the validation module will send exactly one JMS message — the result of the particular validation — to each of these queues. Creating a large number of temporary queues instead of using a centralized queue wastes valuable memory. While it is a feasible approach for low-volume applications, the scalability impact will become apparent with the high loads experienced by most online shops.

A better approach is to use a single queue for all credit card validation results, and locate the corresponding reply message using some type of unique identifier. **JMS message selectors** are just what we need to perform this logical separation — and we'll show you how to use them in this article.

Implementing the Validation Module As an Enterprise JavaBean (EJB)

In our current implementation of the example application, all requests for credit card validation are served by a single-threaded validation module that processes incoming requests sequentially, one after the other. We chose this approach for its simplicity — after all, our goal was to explain the JMS fundamentals, not to delve into the details of high-performance application design — but realistically, under high loads, such an implementation will cause major bottlenecks.

So how can we improve the validation module so that it can handle high loads as well? Clearly we need to give up the single-threaded approach and allow multiple receivers to process the incoming requests in parallel. We could add this capability ourselves, of course, but there is an easier and better solution: We can leverage the infrastructure of the J2EE Engine in SAP Web AS and implement the validation module as an Enterprise JavaBean (EJB) — i.e., as an application component that is executed within the J2EE Engine. In the EJB programming model, we delegate the scalability issue completely to the underlying application server (SAP Web AS), which will ensure that the application components work in parallel if required by the load. While the "classic" EJB types (session beans and entity beans) are not capable of receiving JMS messages asynchronously, a new type of EJB, called a **message-driven bean**, can. Message-driven beans were recently added to the EJB standard to fill the gap between the JMS and EJB programming models, and are designed to react to messages arriving at a particular destination (i.e., "queue" or "topic," depending on the messaging model in use[10]).

It is easy to convert our validation module into a message-driven bean, and doing so will not only solve our scalability problem, it will also allow us to benefit

[10] JMS provides two messaging models: point-to-point (ptp) for one-to-one message delivery, and publish-subscribe (pubsub) for one-to-many message delivery. With ptp, destinations are called queues; with pubsub, destinations are called topics. For more details, refer to our previous article in the July/August 2004 issue, and also to the sidebar "Key JMS Terms and Concepts in Review" on page 78.

from other J2EE Engine features, such as security, user handling, and transaction management, for example. Finally, since an EJB runs within the J2EE Engine, rather than as a standalone Java program, we eliminate the need to separately monitor and administer an additional process.

This article will address the first two issues — improving maintainability by receiving messages asynchronously and improving scalability by using message selectors instead of temporary queues. Since it is such a large topic, we will cover message-driven beans, plus a few more advanced topics, in a future *SAP Professional Journal* article.

Streamlining Your JMS Applications with Asynchronous Message Delivery

A JMS application can receive messages either by actively checking for new messages (synchronous delivery), or by requesting that the JMS provider notify it when new messages arrive (asynchronous delivery). Asynchronous delivery requires a bit more expertise to implement, but results in shorter and more maintainable code in most cases. In this section, we'll modify the validation module of our example application to use asynchronous message delivery.

Let's review a few fundamentals before we start coding.

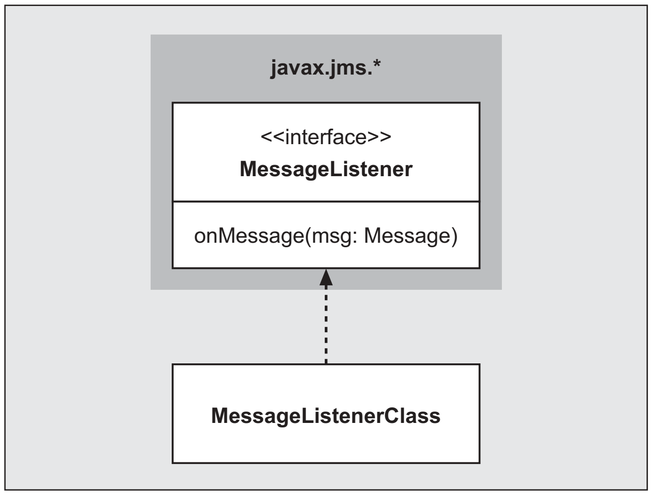
The Fundamentals of Asynchronous JMS Message Delivery

Synchronous message delivery is a pull-based mechanism — the JMS client calls a receive method on the JMS consumer object,[11] thus initiating the message delivery process. Recall from our previous article that there are three versions of the *receive* method: a method that blocks execution until a message is available (a "blocking" method), a non-blocking method that immediately returns a message if there is one (or *null* otherwise), and a blocking method with a timeout. We chose the blocking version for our validation module since we wanted it to wait indefinitely for a new message to arrive. In contrast, we chose the non-blocking version for our frontend servlet since we wanted it to quickly check whether a reply had arrived from the validation module, and to issue a "please wait" page if not.

In contrast, **asynchronous message delivery** is a push-based mechanism. Instead of the client application asking for a message, the JMS provider itself actively delivers an incoming message to the message consumer. The main implementation difference is that, instead of calling the receive method explicitly from the main application code, the application registers a **message listener** object with the consumer object. Since it receives messages asynchronously, the message listener object is now called an **asynchronous consumer**. The message listener implements the JMS interface *javax.jms.MessageListener* (see **Figure 3**), and contains the appropriate business logic to process a message. When a message comes into

[11] A JMS client instantiates producer objects to send messages and consumer objects to receive messages. For more on this, see our previous article and also the sidebar on page 78.

Figure 3 The Message Listener Implements the Interface and Calls onMessage to Process the Request



Critical Issue: Avoiding Collisions in Session Objects

Asynchronous delivery applications are inherently multi-threaded, which means that a sequence of tasks or instructions (a part of the programming code) can be executed in parallel to other sequences. While a detailed discussion of multi-threaded application development is beyond the scope of this discussion, there are a few critical dangers related to multi-threading that you need to steer clear of when using asynchronous delivery in your applications.

Consider our original example application, detailed in our previous article in the July/August 2004 issue. The application uses synchronous delivery, and at all times we have full control over the processing of the (single) thread in which the application code executes. Since we are dealing with just one thread of execution, we do not have to worry about the interaction between different threads or potential concurrency issues.* With asynchronous message delivery, however, there are always at least two threads executing: one in which the client program executes, and a second (the delivery thread) run by the JMS provider's client-side components that invokes the listener's *onMessage* method when a message arrives.

✓ ***Note!***

Technically, we could have developed the validation module to spawn multiple threads, but for simplicity we chose not to. If you choose to spawn multiple threads when using synchronous delivery, be sure to follow the guidelines discussed here for asynchronous delivery.

Understanding how this works is critical because with the exception of destination objects, connection factory objects, and connection objects, JMS objects are not coded to be thread-safe.** You need to pay special attention to session objects, which are involved whenever you send or receive a message. Let's say that you set up an asynchronous consumer and start the connection so that the consumer receives incoming messages. The consumer is associated with a particular session object, and behind the scenes

* When multiple threads access a single object concurrently (e.g., by calling a method or changing a variable), and the object is not coded to handle this concurrent access, the access can cause unpredictable side effects. This is similar to what would happen if two users were allowed to modify the same Microsoft Excel file on a shared drive at the same time: one user's changes might be lost, or worse, a combination of both users' changes might be saved.

** A thread-safe object is designed to handle concurrent use by more than one thread. In this case, the application developer doesn't have to anticipate potential problems because two threads can safely access the object at the same time. Essentially, the critical code sections (e.g., those where data inside the object is modified) have been coded to execute sequentially instead of concurrently.

the destination (the "queue" or "topic," depending on the messaging model in use) for which the listener object is registered, the JMS provider's client-side components[12] call the listener's *onMessage* method, which contains the business logic for processing the request. In our example application, we will place the logic that validates the credit card number and returns a result to the sender inside this *onMessage* method. (See the sidebars above and on the next page for some important considerations to keep in mind when using asynchronous message delivery in your applications.)

[12] Recall that each vendor-specific JMS implementation actually consists of two parts: a server component (the messaging server or JMS provider) plus a client-side class library that is included in the application's class path. Remember that JMS only defines a set of interfaces — the actual classes that implement them are provided by the vendor and are specific to that vendor (SAP, in our case).

Incorporating Asynchronous Message Delivery into the Example

The steps a JMS client must take in order to receive the JMS provider accesses this session whenever it delivers a message to the consumer.*** There are two threads executing in parallel: thread #1, in which your main application runs, and thread #2, which the JMS provider uses to deliver messages (by calling the asynchronous receiver's *onMessage* method). Suppose the application running in thread #1 decides to do something that involves processing by a session object (e.g., send a message, create a new producer object, or create a temporary queue), and uses the same session object to which the asynchronous consumer is assigned. Suppose further that at the same time, in thread #2, the JMS provider delivers an incoming message to the asynchronous consumer. The JMS provider uses the session to track and control the message delivery, so two threads are accessing the same session object at the same time: thread #1 while processing the application code, and thread #2 while delivering the message. Concurrent access to the session object is forbidden, though, and will very likely cause a malfunction. You cannot predict what exactly will happen. You might receive a *JMSException* or a *RuntimeException*, you could lose some messages, or maybe everything will run without incident — there is no way to know in advance. To make matters worse, application errors caused by thread collisions are hard to track down because they appear randomly.

Now the good news! The key to avoiding these collisions is simple: except when closing a session, do not touch the message listener's session in any way after enabling the listener to receive messages (i.e., after calling the start method on the connection object). This includes instantiating new consumer or producer objects, calling other methods on the session object, or even sending messages with existing producers. Each of these actions involves processing by the session object, which can potentially collide with an incoming message, as you just saw. Instead, if you need to send messages, for example, create a new session and producer object to do so in parallel — i.e., dedicate a separate session object for the asynchronous consumer exclusively to the message delivery thread, which the application cannot control (see the section "Tips for Successful JMS Programming" at the end of this article).

In summary, while the multi-threaded nature of the asynchronous delivery mechanism requires you to think a bit more carefully about the design of your application, don't let this discourage you from using it! The approach is very convenient and easy to use once you're used to it, and often yields compact and elegant application code.

*** The session object is the most important JMS object. It is a factory for several objects that use it internally: temporary destinations, producers, consumers, and all types of messages. Session objects also offer methods and options for delivering messages in transactional units. In addition, the JMS provider performs all of its internal message tracking and sequencing at the session level.

Handling Runtime Exceptions with Asynchronous Delivery

An important requirement you need to be aware of with asynchronous delivery is that you must catch runtime exceptions within your message listener. According to the JMS specification, a message listener must not throw a runtime exception, which is considered a client programming error. The specification advises you instead to catch all exceptions, including runtime exceptions, inside the *onMessage* method, and to forward the message that causes the problem to a dedicated destination for "unproccessable" messages. While the question of whether it is good programming style to catch runtime exceptions is

(continued on next page)

(continued from previous page)

arguable (the original purpose of runtime exceptions was to relieve programmers from having to deal with errors), it is what JMS expects from you, so it must be done.*

It is beyond the scope of this discussion to demonstrate JMS exception handling in depth (we will provide a detailed example in an upcoming *SAP Professional Journal* article, so that you can see how it works and what you have to do). For now, you just need to understand how the JMS provider treats a runtime exception and how the application behaves if your message listener throws one. It will depend on the acknowledgement mode that you chose for your JMS session (this was discussed in our previous article):

- **AUTO_ACKNOWLEDGE:** If you chose this mode for your session, you will immediately receive the same message again that caused the problem. To understand the behavior, remember that this mode hides the message acknowledgement process from the application (the provider's client-side objects handle it implicitly). A runtime exception indicates a severe error, so instead of sending an acknowledgement, the provider recovers the session and resends the (last) message.

- **DUPS_OK_ACKNOWLEDGE:** This mode is similar to AUTO_ACKNOWLEDGE, but messages other than the failing one may be resent as well — this mode acknowledges messages as a group or after a predefined time period has passed, instead of immediately as each message arrives.

- **CLIENT_ACKNOWLEDGE:** This mode requires you to handle message acknowledgement yourself, including what happens to a message that causes an exception — JMS does not carry out any implicit actions (like session recovery) for you. When a message cannot be processed successfully, JMS continues to send the messages that follow the failing one until you either acknowledge message delivery or recover the session. If you acknowledge message delivery, the JMS provider will mark all messages — including the failing one — as successfully received, in which case the failing message is lost. If you recover the session (via method *Session.recover*), the JMS provider will redeliver all messages since the last acknowledged one — including the one that caused the error.

* Runtime exceptions (i.e., exceptions derived from the base class *java.lang.RuntimeException*) are a special class of exception that indicate a severe, unexpected error in the program flow, such as a *NullPointerException*, which is thrown when you access a reference that has a value of *null*. Unlike other exceptions, which you have to "catch" (handle the error situation immediately) or "throw" (declare the exception in the method signature and pass it to the caller), runtime exceptions are dealt with automatically. You can catch and handle them if you wish, but you don't have to: they will simply be passed to the caller without being declared in the method signature. Runtime exceptions are also called "unchecked" exceptions, as opposed to regular "checked" exceptions. Since runtime exceptions don't have to be declared, they are the only type of exception a message listener can throw. The listener's *onMessage* method is part of the predefined *javax.jms.MessageListener* interface and, according to its definition, does not throw any checked exceptions.

messages asynchronously are very similar to synchronous message delivery, except for two key differences:

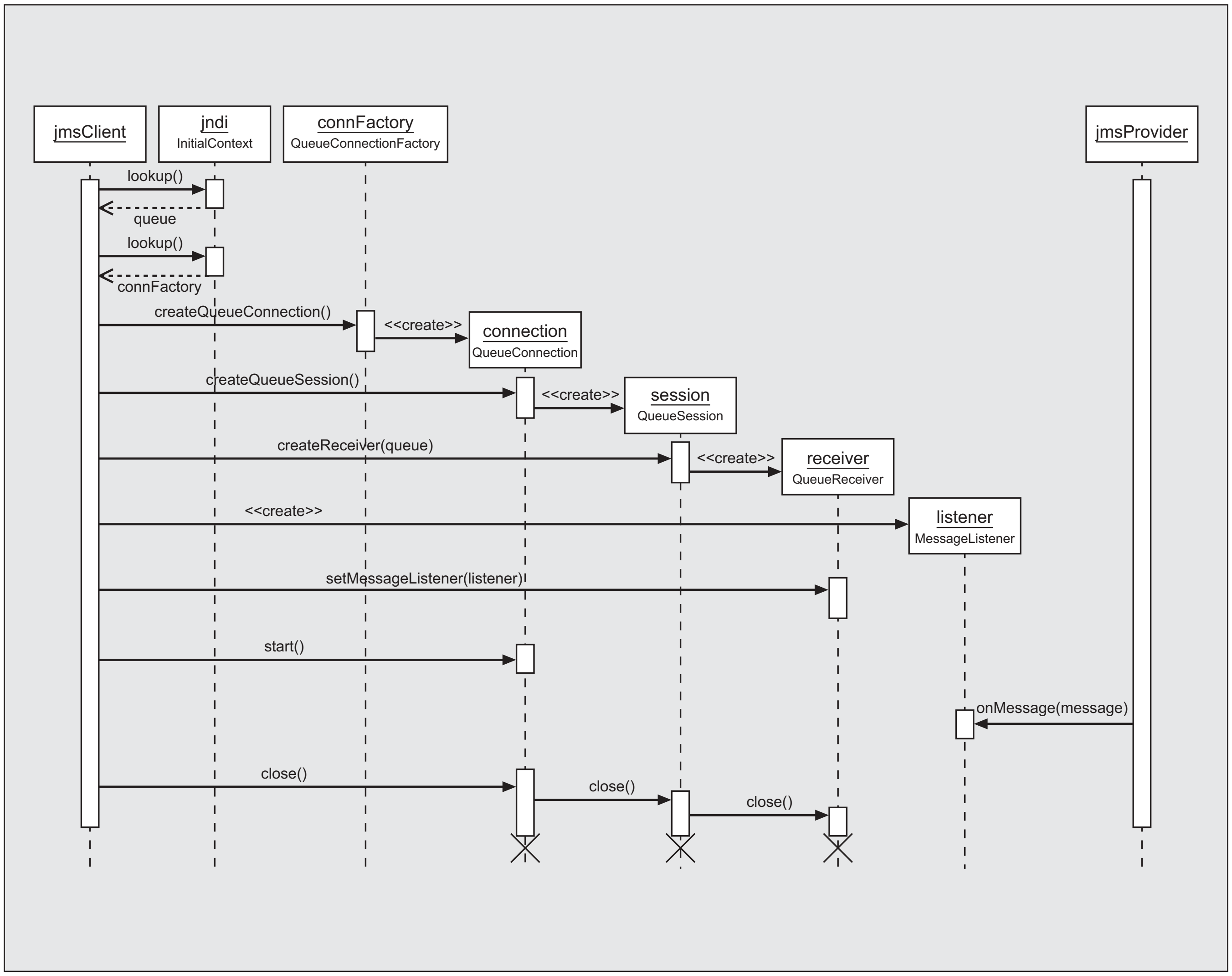
- You need to implement an additional message listener class, and register it with the JMS provider by calling the *setMessageListener* method of the consumer object.

- You don't have to call the consumer's receive method.

With synchronous delivery, you call the consumer's receive method and receive the message as a return value. With asynchronous delivery, the JMS provider automatically calls our registered message listener's *onMessage* method when a message arrives.

Figure 4 provides a graphical summary of the steps for receiving a message asynchronously using the ptp messaging model. In a pubsub messaging

Figure 4 Sequence Diagram for Receiving a Message Asynchronously Using ptp



scenario, the ptp-specific interfaces and methods are simply replaced with their pubsub counterparts.[13]

Let's now modify our example to use asynchronous delivery, which involves two overall steps:

1. Develop the message listener class.

2. Modify the validation module to register the listener with the JMS provider.

[13] Recall that we provided a similar diagram for ptp synchronous delivery in our previous article.

Step 1: Develop the Message Listener Class

The first step in porting our validation module from synchronous to asynchronous delivery is to define a message listener class to handle incoming messages. There are two ways to do this:

- Adjust our existing *CardValidator* class *com.sap.jms.examples.spj.CardValidator* to implement the *javax.jms.MessageListener* interface.

- Create an entirely new, separate listener class.

Figure 5 *Constructor of the ValidatorMessageListener Class*

```
/**
 * This class contains the business logic for the credit card
 * validation. It is a JMS MessageListener, which means it
 * asynchronously receives messages for processing.
 */
public class ValidatorMessageListener implements MessageListener {

  private QueueSession session;
  private QueueSender sender;


  /**
   * Constructs a ValidatorMessageListener object.
   * @param connection The QueueConnection to be used
   * @throws JMSException If a JMS error occurs
   */
  public ValidatorMessageListener(QueueConnection connection)
      throws JMSException {

    // Create a non-transacted session
    session = connection.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);

    // Create a sender with no specific queue
    sender = session.createSender(null);
  }
```

The only requirement is that our Java message listener class must implement the *javax.jms.MessageListener* interface, so from the perspective of JMS, the approaches are equivalent — the one you use is entirely up to you. For clarity, we decided to develop a separate listener class *com.sap.jms.examples.spj.ValidatorMessageListener*, as shown in **Figure 5**.

The constructor (method *ValidatorMessageListener*) expects the queue connection as an input parameter. This lets us reuse an existing connection instead of creating a new one (an expensive task from a processing perspective). We use the connection to create a non-transacted session, and then use that session to create a sender without specifying a queue for the sender to send to. For now (in keeping with our original design), we will send the result of the card validation to the temporary queue specified in the validation request's *JMSReplyTo* header field.

Next, let's add the code to process an incoming message. The interface *javax.jms.MessageListener* defines just a single method, *onMessage*, that is executed when a message arrives in the queue the asynchronous consumer is listening to (see **Figure 6**).

Notice that we have simply moved the logic for processing a validation request from the class *CardValidator*, where it used to be, to our new

Figure 6 *Implementation of the ValidatorMessageListener Class*

```
/**
 * This method contains the message listener's business logic,
 * which is the code to be executed on the arrival of a message.
 * In our case, it simulates the credit card validation.
 * @param msg The message to be processed
 */
public void onMessage(Message msg) {
  String cardNumber;
[…]
    // Get the card number from the message
    cardNumber = msg.getStringProperty(
        CardValidator.MSG_PROPERTY_CARD_NUMBER);
    if (cardNumber == null) {
      System.out.println(
         "Not a valid request. Message property missing: "
          + CardValidator.MSG_PROPERTY_CARD_NUMBER);
    } else {
      // Validate the card number,
      // and keep the result for the response
      boolean result = validateCard(cardNumber);
      System.out.println("Validated card number:" + cardNumber);

      // Create a new message and set the result as a property
      Message replyMsg = session.createMessage();
      replyMsg.setBooleanProperty(
          CardValidator.MSG_PROPERTY_CHECK_RESULT, result);

      // Send the reply message to the destination
      // given in the received message
      Queue replyQueue = (Queue)msg.getJMSReplyTo();
      if (replyQueue == null) {
        System.out.println("JMSReplyTo missing. Request ignored");
      } else {
        sender.send(replyQueue, replyMsg);
        System.out.println("Response sent: " + result);
      }
    }
[…]
/**
 * Validates the cardNumber.
 * <p>
 * In a real world application, this would probably generate
 * a request to an external service provider.
 * We are just checking the length of the number, and simulate
 * the long external request by sleeping for 10 seconds.
 * @param cardNumber The credit card number to validate
 * @return <code>true</code> if the credit card number is okay,
 *         <code>false</code> otherwise.
```

(continued on next page)

Figure 6 (continued)

```
   */
  public boolean validateCard(String cardNumber) {
    boolean result = (cardNumber.length() == 16);
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      // Hey, this is just an example, so we don't care
    }
    return result;
  }

}
```

message listener class *ValidatorMessageListener*. Nothing else has changed: we retrieve the credit card number to be validated from the incoming message,[14] validate the number, and return the result as a reply message to the temporary queue.[15]

There's one more thing to note. As you can see in **Figure 7**, we catch all possible exceptions that might occur (refer back to the sidebar "Handling Runtime Exceptions with Asynchronous Delivery" on page 85). For simplicity, we coded the handler to simply print the stack trace of the exception. In a real application, you should implement a more robust exception-handling routine.

For example, you might log the exception in your environment's system log for later analysis, and send a JMS message containing an error message to the reply queue, which the payment module then displays to the user. The message could contain a string property with a more descriptive message like: *An internal error occurred while validating your payment information. If the problem persists, please inform the system administrator*.

[14] The constant *MSG_PROPERTY_CARD_NUMBER*, which we use in Figure 6 to get the message property, is defined in the class *CardValidator*. It has the value *cardNumber*.

[15] The result is stored as a message property named *result* (see the constant *MSG_PROPERTY_CHECK_RESULT* defined in the class *CardValidator* in Figure 6).

Step 2: Modify the Validation Code to Register the Listener with the JMS Provider

Now that we've moved the business logic for the example to the message listener class, the main class *CardValidator* becomes much shorter. Let's compare the new version with the original implementation from the previous article, section by section, and insert the code that will register our new *ValidatorMessageListener* class with the JMS provider.

The easiest place to start is with the *CardValidator* class's constructor method, because it remains unchanged. As before, the code first creates a *javax.naming.InitialContext* object in order to access the Java Naming and Directory Interface (JNDI) service. We use it to look up the administered JMS objects — that is, the queue connection factory[16] *QueueConnFactory* and the queue *spjDemoQueue*. Finally, we use the connection factory to create a queue connection that will interact with the JMS provider. The queue and the connection are stored in instance variables for later use (see **Figure 8**).

[16] Remember from the previous article that instead of instantiating an object from a vendor-specific class, JMS defines a series of "factory" methods that instantiate each vendor-specific object for you and return a reference to that object (see the sidebar "Key JMS Terms and Concepts in Review" on page 78).

Figure 7 *Exception Handling in the ValidatorMessageListener Class*

```
public void onMessage(Message msg) {
  String cardNumber;

  try {

[…]
    // business logic of the listener

  } catch (Exception e) {
    // In a serious application, you should handle the exception
    e.printStackTrace();
  }
}
```

Figure 8 *Constructor of the CardValidator Class*

```
/** The class name of the initial context factory to be used. */
public static final String INITIAL_CONTEXT_FACTORY =
    "com.sap.engine.services.jndi.InitialContextFactoryImpl";
/** The URL to the naming service provider's configuration info. */
public static final String PROVIDER_URL = "localhost:50004";

/** The JNDI lookup name for the JMS connection factory to be used.*/
public static final String CONN_FACTORY_LOOKUP_NAME =
    "jmsFactory/QueueConnFactory";
/** The JNDI lookup name for the JMS queue to be used.*/
public static final String QUEUE_LOOKUP_NAME =
    "jmsQueues/spjDemoQueue";
[…]
private Queue queue;
private QueueConnection connection;
[…]
/**
 * Constructs a CardValidator object.
 * @param username Username used to create an InitialContext and
 *        a JMS connection
 * @param password Password used to create an InitialContext and
 *        a JMS connection
 * @throws NamingException If a JNDI error occurs
 * @throws JMSException If a JMS error occurs
 */
public CardValidator(String username, String password)
    throws NamingException, JMSException {

  super();

```

(continued on next page)

Figure 8 (continued)

```
    // Create an initial context for doing JNDI lookup
    Properties env = new Properties();
    env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
    env.put(Context.PROVIDER_URL, PROVIDER_URL);
    env.put(Context.SECURITY_PRINCIPAL, username);
    env.put(Context.SECURITY_CREDENTIALS, password);
    InitialContext jndi = new InitialContext(env);

    // Lookup factory and queue
    QueueConnectionFactory factory =
        (QueueConnectionFactory)jndi.lookup(CONN_FACTORY_LOOKUP_NAME);
    queue = (Queue)jndi.lookup(QUEUE_LOOKUP_NAME);

    // Create the connection
    connection = factory.createQueueConnection(username, password);
  }
```

The class's *main* method (not shown because it is very straightforward) creates a *CardValidator* object and calls the validator's *process* method immediately afterward. We need to modify the *process* method to register our message listener with the JMS provider. As before, we use the connection to create a non-transacted JMS session, and use the session object to create a queue receiver that receives messages from the queue *spjDemoQueue*. Next, we create a message listener by instantiating an object of type *ValidatorMessageListener*, and register the object as a message listener for our receiver by calling a new method on the receiver *setMessageListener*.

Figure 9 shows the updated *process* method. The added code is shown in boldface type.

Next, after everything is set up, we start the connection (see **Figure 10**). From this point on, the JMS provider will deliver messages to our message listener.

✓ ***Note!***

Notice in line 141 of Figure 9 that we pass in our connection object when instantiating the message listener object. As shown earlier in the code for class ValidatorMessageListener (Figure 5), the listener needs to instantiate a sender object to send the reply messages. Normally you would use the connection factory to create a connection, the connection to create a session, and, finally, the session to create a sender. In this case, however, we can reuse the connection object already instantiated by the CardValidator, so we don't have to create another one.

We could even have passed the session object directly, in which case the asynchronous consumer and the sender would share the same session object. But remember that the session object is not thread-safe, so we shouldn't do it unless we are 100% sure that no concurrent access to the session is possible (refer back to the sidebar "Critical Issue: Avoiding Collisions in Session Objects" on page 84). In this particular case it would be okay (see the tips section later in the article for details), but we prefer the defensive approach of creating separate session objects for the sender and the consumer. It is clearer, easier to maintain, and, in the end, less error-prone.

Figure 9 *CardValidator — Create and Set the Message Listener*

```
/**
 * Process the incoming validation requests.
 */
public void process() {

  try {
    // Create a non-transacted session and a receiver for the queue
    QueueSession session = connection.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);
    QueueReceiver receiver = session.createReceiver(queue);

    // Create and set the message listener
    ValidatorMessageListener listener =
        new ValidatorMessageListener(connection);
    receiver.setMessageListener(listener);
```

Figure 10 *CardValidator — Start the Connection*

```
    // Start the connection in order to receive messages
    connection.start();
```

Figure 11 *CardValidator — Implement a Program Shutdown*

```
    // Inform the user
    System.out.println(
        "CardValidator running. Press return to stop");
    System.in.read();
```

This continues until the consumer object the message listener belongs to is closed, or until the connection is explicitly stopped.

Switching to asynchronous message delivery enables us to implement a more graceful shutdown mechanism without much effort. Recall that originally the validation server process ran indefinitely in an endless loop — you had to manually terminate the validation server process via an operating system function. With our new design, we print status information to the console and simply wait for manual input from the administrator to stop the validation program. **Figure 11** shows the required code.

Figure 12 *CardValidator — Close the Connection*

```
    } finally {
      // Make sure we close the connection in any case
      try {
        if (connection != null) {
          connection.close();
        }
      } catch (JMSException e1) {
        e1.printStackTrace();
      }
    }
  } // end process

}
```

Once the administrator presses the *Return* key in the console, the credit card validation program is shut down properly. This includes closing the connection we opened in the constructor, which in turn closes all subordinate objects — i.e., the two sessions, the consumer, and the producer. We place the code in the *finally* class of our *process* method, as shown in **Figure 12**.

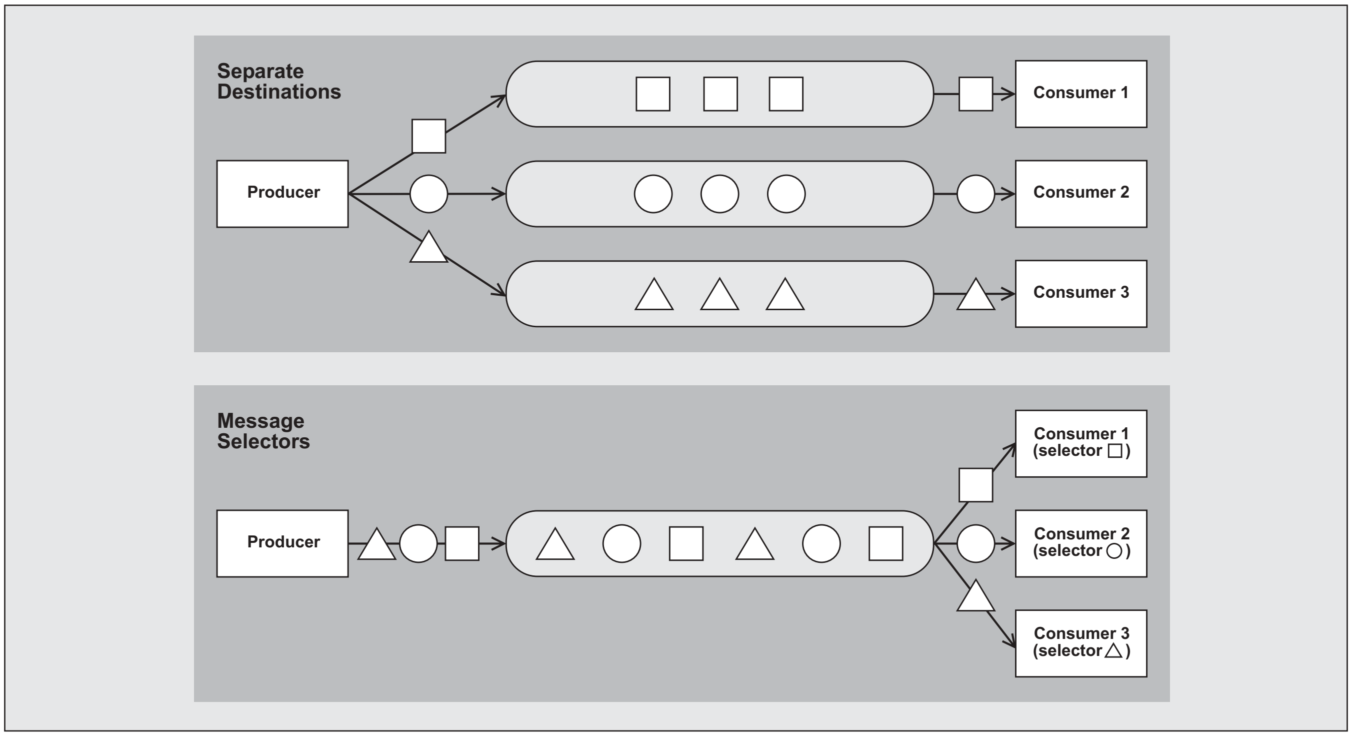
To summarize, we didn't have to change much to use an asynchronous consumer inside our validation module. We created and registered a *MessageListener* object that the JMS provider actively notifies whenever a message arrives. This approach is a great fit for server-like programs like ours (i.e., ones that respond to incoming messages), and leads to more straightforward and maintainable application code. Next, we'll look at how we can improve the performance and scalability of our example using message selectors instead of temporary queues.

✓ ***Tip***

The "message handler" design pattern is quite commonly employed in object-oriented programming languages like Java. If you've seen it before, you may have expected that the switch to using handlers could tremendously improve our application's scalability, perhaps by letting us easily instantiate potentially dozens of message handlers to process requests in parallel. This is possible, as long as message order does not matter. The JMS queue defines a natural order for the messages: the ones that are sent first are received first. When you switch to parallel processing, the order is lost.

Keep in mind that there is a 1:1 relationship between consumer objects and message handlers, so you need to set up multiple consumers as well. JMS does not define how a JMS provider handles multiple consumers on a queue, though! SAP Web AS allows multiple consumers on a queue and serves them with messages in a round-robin fashion, so you can use this technique to distribute your load. However, an application relying on this behavior is not necessarily portable across different providers. Also keep in mind the concurrency issue (refer back to the sidebar on page 84): Each consumer needs its own session object in order to operate in parallel to the others.

Figure 13 Processing Messages Using Separate Destinations vs. Message Selectors



Improving the Efficiency of Your JMS Applications with Message Selectors

Recall that our original example used a separate (temporary) destination[17] for each credit card validation result. This was done for convenience, and to demonstrate how to set up and use temporary destinations. Although JMS message selectors are primarily intended for defining a subset of messages for a consumer to receive, we can also leverage them to more efficiently handle our validation results.

The Fundamentals of JMS Message Selectors

When designing a JMS-based application, you will usually separate different types of messages by using different destinations (queues or topics). In an online shop application, for example, you would send a new customer's registration to a different destination than an existing customer's order. Each is an independent business process, and the requests are handled in completely different ways, so setting up separate destinations makes sense. Sometimes, however, you'll want to further categorize and process a type of message. In our shopping example, for instance, you might want to add an order approval step if the total value of the order exceeds a specified limit. In other words, you might want to categorize the orders according to their purchase value.

In principle, there are two different ways to process messages according to a logical "category" (see **Figure 13**):

- The application can set up separate destinations for each category.
- The application can use a common destination and let the consumers filter their messages using a message selector.

[17] Technically a queue, since we used point-to-point messaging. Recall that destinations are called "queues" with point-to-point (ptp) messaging and "topics" with publish-subscribe (pubsub) messaging (see the sidebar "Key JMS Terms and Concepts in Review" on page 78).

Figure 14 *Some Valid Message Selectors*

Message Selector	Description
`JMSType = 'ABC'`	Receive only messages where the header field JMSType has the value ABC.
`purchaseValue > 10000 OR customerGroup = 1`	Receive only messages where the value of the property purchaseValue is greater than 10000 or the value of the property customerGroup is 1.
`customerName LIKE 'A%'`	Receive only messages where the property customerName starts with A.

A **message selector** is essentially a provider-side filter for messages. Only messages that match the filter criteria are actually delivered to the consumer. Each consumer can use its own message selector, which is defined when you create the consumer object, and therefore have an individualized view of the available messages in a queue or topic.

The approach that is best for your application will depend on the situation. If you need a large number of message categories, for example, you will probably prefer the message selector approach since it involves less overhead than creating a large number of destinations. Also, if your categories are dynamic (e.g., if categories can be created, removed, or rearranged), message selectors will model the category structure much better than separate destinations because they are more flexible. Correspondingly, if you need just a few categories and don't expect any changes to the category structure, you might prefer a separate destination for each category — this approach is slightly faster because the JMS provider does not have to evaluate the message selectors prior to sending a message.

A message selector is a textual, conditional (boolean or arithmetic) expression that can filter messages based on their properties and some header fields. A selector cannot refer to the body of the message, however! Technically, the message selector is a string that follows a syntax referred to by the JMS specification as "a subset of the SQL92 conditional expression syntax." In practice, this means that message selectors look very similar to the *WHERE* clauses of SQL *SELECT* statements. **Figure 14** lists some examples of valid message selectors.

To use a message selector, you simply have to pass it to the factory method used to create a consumer object. We'll demonstrate how to do this next using our example.

✓ ***Note!***

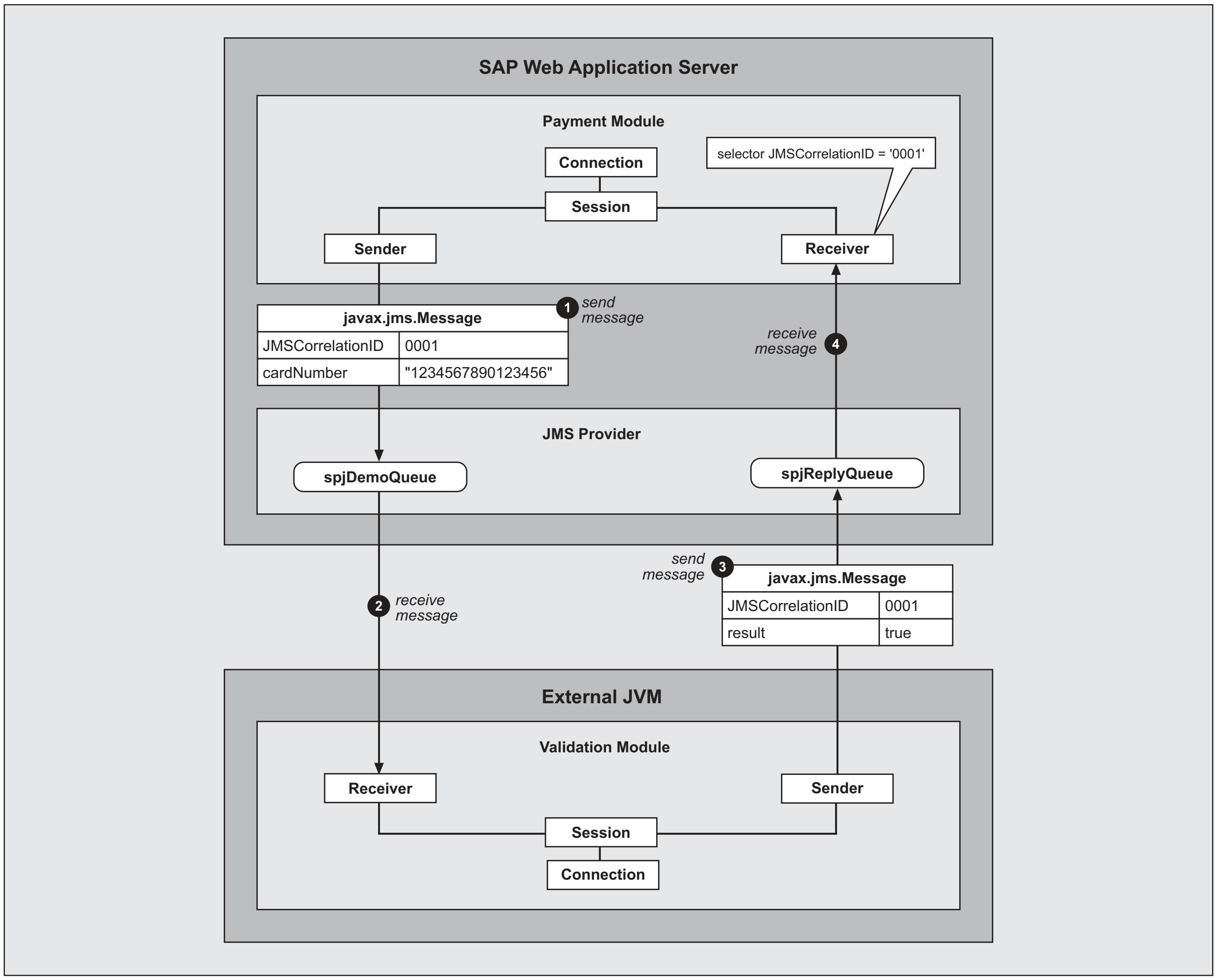
The specifics of the message selector syntax are beyond the scope of this article, but most developers can write them without much trouble. For a reference, consult the JMS specification document available at ***http://java.sun.com/products/jms****.*

Incorporating Message Selectors into the Example

To gain experience with message selectors, let's exchange the temporary-queue-based reply mechanism in our example with one that uses message selectors. **Figure 15** shows how this will affect the interaction between the application components.

Instead of using a separate temporary queue as a reply-to destination for each user session (i.e., each credit card number to validate), we'll now send all reply messages to a single reply queue called *spjReplyQueue*. Since this queue will hold replies for all validation requests, our payment module will need a way to identify its own reply message. This is where message selectors come in: the payment module will place a unique identifier in the header field

Figure 15 Updated JMS Design of the Example Using Message Selectors



JMSCorrelationID of the request message,[18] and the validation module will copy this value into the header field *JMSCorrelationID* of the reply message. The payment module will create a receiver with a message selector that filters on the specific value of the *JMSCorrelationID*, thereby receiving only its own reply message from the generalized queue.

The basic interaction between the payment and validation modules, however, is hardly changed. The payment module sends the credit card verification request (with header field *JMSCorrelationID* set appropriately) to *spjDemoQueue* (step ❶). The validation module receives this message (step ❷) and validates the credit card number. It sends a reply message to *spjReplyQueue*, again setting the value in the header field *JMSCorrelationID* (step ❸). The payment module receives this message (step ❹), because it matches the receiver's message selector, and continues its work according to the result.

[18] Our example implementation uses the HTTP session ID for this identifier.

So what changes in the coding? The version of the validation module that uses message selectors

Figure 16 *Constructor of the ValidatorMessageListener2 Class*

```
public class ValidatorMessageListener2 implements MessageListener {

  private QueueSession session;
  private QueueSender sender;


  /**
   * Constructs a ValidatorMessageListener2 object.
   * @param connection The QueueConnection to be used
   * @param replyQueue The queue to send the reply messages to
   * @throws JMSException If a JMS error occurs
   */
  public ValidatorMessageListener2(
      QueueConnection connection, Queue replyQueue)
      throws JMSException {

    // Create a non-transacted session
    session = connection.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);

    // Create a sender for the reply queue
    sender = session.createSender(replyQueue);
  }
```

is implemented in class *com.sap.jms.examples.spj.CardValidator2*. The code of interest is in the message listener class, implemented in class *com.sap.jms.examples.spj.ValidatorMessageListener2*. Both classes are available for download at **www.SAPpro.com**.

Let's now walk through what we need to do to enable our example to use message selectors:

1. Update the message listener code.

2. Update the payment module servlet.

Step 1: Update the Message Listener Code

Our implementation of class *CardValidator2* will be almost identical to our previous *CardValidator* implementation (Figures 8-12). The queue for the card validation reply messages is now a regular queue (instead of a temporary queue), so we simply add another JNDI lookup statement to the constructor[19] and store the queue's reference in an additional instance variable called *replyQueue*. In the *process* method (called by the *main* method), we instantiate a different message listener — an object of type *ValidatorMessageListener2* — and set this as our receiver's message listener.

Both changes are straightforward, so we'll skip the code example. It is more interesting to look at the message listener class, because this is where we send the reply message for a validation request. **Figures 16-18** show a new version of the message listener class (the modified code is shown in boldface type).

[19] Recall that we previously used JNDI to retrieve a reference to *spjDemoQueue*, which was defined as an administered object on the SAP Web AS system.

Figure 17 Implementation of ValidatorMessageListener2 — Create a Reply Message

```
  /**
   * This method contains the message listener's business logic,
   * that is, the code to be executed on the arrival of a message.
   * In our case, it simulates the credit card validation.
   * @param msg The message to be processed
   */
  public void onMessage(Message msg) {
[…]     // get the card number from the request and validate it

        // Create a new message and set the result as a property
        Message replyMsg = session.createMessage();
        replyMsg.setBooleanProperty(
           CardValidator.MSG_PROPERTY_CHECK_RESULT, result);

        // Copy the original message's correlation ID to allow
        // a mapping between request and reply message
        replyMsg.setJMSCorrelationID(msg.getJMSCorrelationID());
```

Figure 18 Implementation of ValidatorMessageListener2 — Send the Reply Message

```
        // Send the reply message to the reply queue
        sender.send(replyMsg);
        System.out.println("Response sent: " + result);
[…]
  }
[…]
}
```

Note in **Figure 16** how the listener's constructor expects a second reply queue parameter in addition to the queue connection. We pass this reply queue to the factory method that creates the sender object to indicate where the sender should send messages to. Remember that in the original code, we used a temporary queue reply mechanism we called *session.createSender(null)*, because the temporary queue was not available at that point in time and because we had to specify a different queue for each message.

Next, in the listener's *onMessage* method, we have to copy the value of the header field *JMSCorrelationID* from the request message to the reply message. Recall that the payment module will use this value to identify the reply message that corresponds to its validation request. **Figure 17** shows the relevant code, with the modifications shown in boldface type.

Next, when we send the reply message, we don't have to specify the destination (i.e., the reply queue) as we did in the previous implementation. JMS already knows where we want to send the message (to *replyQueue*) since we told it when creating the sender object. **Figure 18** shows the relevant code, with the modifications shown in boldface type.

Figure 19 Implementation of CCVerifyServlet — Send a Validation Request

```
/** Name of the request parameter specifying the action */
public static final String ACTION_PARAM_NAME = "action";
[…]
/** Value of the request parameter ACTION_PARAM_NAME, meaning that
 *  the action to be performed is to send a credit card validation
 *  request */
public static final String ACTION_SEND_VALIDATION_REQUEST =
    "sendVal";
/** Value of the request parameter ACTION_PARAM_NAME, meaning that
 *  the action to be performed is to receive a credit card validation
 *  response */
public static final String ACTION_RECEIVE_VALIDATION_RESPONSE =
  "receiveVal";
[…]
protected void doGet(
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

  // Get the action parameter from the request
  String action = request.getParameter(ACTION_PARAM_NAME);

  try {
    if (ACTION_SEND_VALIDATION_REQUEST.equals(action)) {
      // Executed for requests coming from the form

      // Get the card number from the form
      String cardNumber =
          request.getParameter(CARD_NUMBER_PARAM_NAME);
      // Create a JMS message and set the card number as a property
      Message validationRequest =
          getSession(request).createMessage();
      validationRequest.setStringProperty("cardNumber", cardNumber);
      // Set the session ID as JMSCorrelationID
      validationRequest.setJMSCorrelationID(
          request.getSession().getId());
      // Send the message
      getSender(request).send(validationRequest);

      // Redirect the browser to the wait page
      response.sendRedirect("wait?time=10");
```

Step 2: Update the Payment Module Servlet

Let's now shift our focus to the servlet *com.sap.jms.examples.spj.servlet.CCVerifyServlet*, which is the core of our frontend payment module. *CCVerifyServlet* accepts the credit card validation requests coming from the browser and translates them into JMS requests. The servlet then looks for JMS reply messages containing the validation result and sends the appropriate result page to the browser if

✓ ***Note!***

The code sample in Figure 19 also uses the helper methods getSession(request) and getSender(request), which we used when developing the original example in the previous article. The getSession(request) and getSender(request) helper methods retrieve the session and sender objects that we store as part of the HTTP session object provided automatically by the J2EE runtime. Recall that our design involves showing a wait page while the credit card validation processes, which includes having the browser periodically check (i.e., relaunch our servlet) to see if a reply message has arrived. Since HTTP-based communication is inherently stateless, the servlet that receives the "just checking in" request from the browser would ordinarily be starting from scratch from a memory perspective. Fortunately, the J2EE runtime provides an HTTP session object in which we can store variables and object references, and automatically makes it available to the responding servlet whenever a request arrives from the browser session. This saves us from having to create a new JMS session, sender, and other objects for each new HTTP request from a user. Keep in mind that we cannot share the JMS session and the sender across user sessions, however, because they are not thread-safe — i.e., they are not designed to be accessed by multiple threads at the same time (see the sidebar on page 84) — and each user session (HTTP session) runs in parallel to the others in a separate thread.

Note that we have slightly modified the methods here from how we used them in the original example. We now store "wrapper" objects in the HTTP session that contain the actual JMS objects. On the first call, the respective JMS object and its corresponding wrapper object are created, the wrapper object is stored in the HTTP session context, and the JMS object is returned. Any later call belonging to the same session locates the wrapper object in the session context and simply returns the JMS object stored inside. We explain why we use these wrapper objects in the sidebar "Using Wrapper Objects to Avoid 'Passivation' Errors" on page 103.

one is found.[20] In the listings that follow, the code sections that have changed to enable our new message-selector-based reply mechanism are shown in bold-face type.

Let's start by adding logic to set the unique identifier value we'll use to locate our reply request in the generalized reply queue (see **Figure 19**).

The servlet processes the HTTP GET requests coming from the browser in its *doGet* method. If this request contains a parameter called *action* with the value *sendVal*, it indicates that the request contains a credit card number to validate. We do basically the same thing here that we did in the original example from the previous article: get the credit card number to validate, create a new JMS message, and set the number as a string property *cardNumber*. In order to later identify the response that corresponds to this request, we set the header field *JMSCorrelationID* to a unique value — in this case, the unique server-generated HTTP session identifier — which we get by calling *request.getSession().getId()*. Notice also that we don't have to set the *JMSReplyTo* header field to provide the validation module with a reference to a temporary reply queue, since we're no longer creating or using one — the validation module will be able to look up the regular queue using JNDI. The name of the queue is part of the application "contract" that has to be agreed to by all components beforehand. *CCVerifyServlet* finally uses a queue sender object to send the message to *spjDemoQueue* and redirects the browser to a wait page that informs the user about the ongoing validation process.

The wait page periodically sends a request to *CCVerifyServlet* to check if there is a validation result

[20] Note that the result pages are available as part of the download at **www.SAPpro.com**.

Figure 20   Checking for New Messages and Issuing a Wait Page If None Is Found

```
} else if (ACTION_RECEIVE_VALIDATION_RESPONSE.equals(action)) {
  // Executed for requests coming from the wait.jsp page

  // Try to receive the response from the reply queue
  // without waiting
  Message validationResponse =
    getReplyReceiver(request).receiveNoWait();
  if (validationResponse == null) {
    // Return to the wait page and wait some more
[…]
  } else {
    // Get the check result from the received message
    boolean result = validationResponse.getBooleanProperty(
        CHECK_RESULT_MSG_PROPERTY);

    // Redirect the browser to the appropriate page
[…]
```

available. **Figure 20** shows the part of the *doGet* method that executes.

This part of the code remains unchanged from the original version of the example. We use the receiver object to look for a message in the reply queue (without blocking). If there is one, we retrieve the validation result from the message property *result* and redirect the browser accordingly. Otherwise, we return to the wait page to wait for several seconds

Figure 21 Specifying a Message Selector When Instantiating a Message Receiver Object

```
/** Attribute name used to bind the reply receiver to the
 *  HttpSession */
public static final String REPLY_RECEIVER_ATTRIB_NAME =
    "replyReceiver";
[…]
protected QueueReceiver getReplyReceiver(HttpServletRequest request)
  throws JMSException, NamingException {

    HttpSession session = request.getSession();
    QueueReceiverWrapper wrapper =
        (QueueReceiverWrapper)session.getAttribute(
            REPLY_RECEIVER_ATTRIB_NAME);
    if (wrapper == null) {
      // stop the connection before creating a receiver
      getJMSConnection().stop();
      Queue replyQueue = getQueue(REPLY_QUEUE);
```

Using Wrapper Objects to Avoid “Passivation” Errors

As you can see in lines 183-185 in Figure 21, we store an object of type *com.sap.jms.examples.spj.servlet.QueueReceiverWrapper* in the session context — not the *javax.jms.QueueReceiver* object itself. Why? The reason is that the *QueueReceiver* object is not serializable* — i.e., it is not possible to write the complete state of the object and all the objects it references to an output stream (e.g., a file, a byte array, or a stream associated with a TCP/IP socket) in order to re-create it at some later time by reading its state from the stream. This is actually not a problem on most application servers — we can store non-serializable objects in the HTTP session — unless our application needs some failover capabilities, which includes the ability to migrate a session from one server node to another. To support this capability, the underlying application server “passivates” sessions (i.e., prepares to move them to another server), which implies that all objects stored inside the session are serialized. If the session contains a non-serializable object, the attempt to serialize it will cause a *java.io.NotSerializableException*, and the session will not be passivated.

* To be serializable, an object (as well as all objects it references as member variables, and are not marked with the keyword “transient”) must implement the *java.io.Serializable* or *java.io.Externalizable* interface.

(continued on next page)

before retrying. The key difference is how we set up the receiver for the reply queue in method *getReplyReceiver(request)*, as shown in **Figure 21**.

See how easy this is? We use the HTTP session to store the reply receiver — or, to be precise, the wrapper object for the receiver (see the sidebar above for

Figure 21 (continued)

```
175         String messageSelector =
176             "JMSCorrelationID = '" + request.getSession().getId()
177             + '\'';
178         QueueReceiver qreceiver = getSession(request).createReceiver(
179             replyQueue, messageSelector);
180         // start the connection, so the receiver can work
181         getJMSConnection().start();
182
183         wrapper = new QueueReceiverWrapper(
184             qreceiver, REPLY_RECEIVER_ATTRIB_NAME);
185         session.setAttribute(REPLY_RECEIVER_ATTRIB_NAME, wrapper);
186         System.out.println(
187           "Created reply receiver " + request.getSession().getId());
188       }
189       return wrapper.getQueueReceiver();
190   }
```

(continued from previous page)

The key to avoiding these errors is to remove non-serializable objects (like our *QueueReceiver* object) from the HTTP session just before SAP Web AS tries to passivate the session. But how do we know when this is about to happen? Fortunately, the SAP Web AS J2EE runtime includes a notification mechanism. All our object has to do is implement the *javax.servlet.http.HttpSessionActivationListener* interface. Any object stored in the session context that implements this interface will be notified right before the session is passivated. The application server calls the method *sessionWillPassivate* (defined in the interface), so we can remove the object from the session and close the receiver object by adding code to this method.

Here is the code for the wrapper class that will avoid the potential issue:

```
class QueueReceiverWrapper
    implements HttpSessionActivationListener, Serializable {

  private transient QueueReceiver receiver = null;
  private transient String attributeName = null;
[…]
  public void sessionWillPassivate(HttpSessionEvent arg0) {
    System.out.println("sessionWillPassivate:" + arg0);
    if(receiver != null) {
      arg0.getSession().removeAttribute(attributeName);
      try {
        receiver.close();
        receiver = null;
      } catch (JMSException e) {
        e.printStackTrace();
      }
    }
  }
```

The same mechanism is used inside the *getSession*, *getReceiver*, and *getSender* methods of *CCVerifyServlet*.

more on why we use wrapper objects here). By calling *session.getAttribute(REPLY_RECEIVER_ATTRIB_NAME)* in lines 169-170, we check if the HTTP session already contains a reply receiver, and if not we create one that references the centralized reply queue on the JMS provider. The reference is retrieved using JNDI within *getQueue(REPLY_QUEUE)* — the actual JNDI lookup is performed just once for the servlet and stored in the servlet context. In the highlighted part of the listing, we then create a receiver object with a message selector. Obviously, we are only interested in the reply that matches our validation request, not all the other reply messages residing in *spjReplyQueue*, so we specify the message selector *JMSCorrelationID = '<our session ID>'* as the second parameter when creating the receiver. From the receiver's point of view, it looks as if there is a separate queue holding only its individual reply message (just as in the original temporary queue approach), although in fact all reply messages are sharing the same physical queue.

In summary, we have just modified our payment processing servlet and credit card validation components to use a more efficient, message-selector-based reply mechanism instead of the original temporary-destination-based approach. In this way, all sessions now share a central, administered queue for replies as well as requests.

✓ Note!

The example uses an additional administered object, spjReplyQueue, which needs to be set up before you can run the application. On SAP Web AS 6.30 and 6.40, however, you don't have to create the queue manually — this happens automatically during deployment of the application.

Tips for Successful JMS Programming

Here are some important things to remember in order to avoid common issues with asynchronous delivery and message selectors.

Message Listeners Need Exclusive Access to Their JMS Sessions

As mentioned earlier (see the sidebar on page 84), JMS session objects must not be accessed concurrently from different threads. If you create and register message listeners for asynchronous message delivery, you have to make sure not to subsequently use the JMS sessions associated with these listeners.

Here are some guidelines to avoid collisions:

- ☑ If you have one application component that sends *and* consumes messages asynchronously, use separate sessions for the producer and consumer objects.

✓ Exception!

If you use the message producer only inside the onMessage method of the message listener — for example, to forward a message to another destination — you can use the same session object for the consumer and the producer. Inside the onMessage method, your code is executed within the message delivery thread, so there is no potential concurrency issue.

- ☑ Similarly, use different session objects for a synchronous receiver and an asynchronous receiver, if you need both.

- ☑ Multiple message listeners for the same destination are handled by the JMS provider automatically in order to obey the session's concurrency restriction. That is, if you create multiple asynchronous consumers that share the same session object (i.e., if you set up one session, create multiple consumers on it, and then register a message listener with each consumer), the JMS provider will deliver messages to each consumer sequentially, and wait until its handler has finished processing before delivering the next.

  In contrast, if you provide each consumer with its own session (i.e., if you set up multiple sessions, create a consumer on each of them, and then register a message listener with each consumer), they can receive and process messages in parallel. The thing to remember is that creating additional listeners to a queue or topic will only improve your application's performance and scalability if each is provided its own session to work with.

- ☑ In the rare case that you need more than one message listener for a session, make sure the connection is stopped (or hasn't been started) before you create them. While this is good programming practice anyway, here it is essential in order to avoid potential concurrency problems. The JMS provider starts message delivery as soon as the

Figure 22 Header Fields Available to Clients

Header Field	Setter Method Called by	How Client Can Set the Field*
JMSDestination	Send/publish method	QueueSession.createSender, QueueSender.send
JMSDeliveryMode	Send/publish method	QueueSender.setDeliveryMode, QueueSender.send
JMSExpiration	Send/publish method	QueueSender.setTimeToLive, QueueSender.send
JMSPriority	Send/publish method	QueueSender.setPriority, QueueSender.send
JMSMessageID	Send/publish method	—
JMSTimestamp	Send/publish method	—
JMSCorrelationID	Client	(directly)
JMSReplyTo	Client	(directly)
JMSType	Client	(directly)
JMSRedelivered	Provider	—

* The appropriate API methods are given for the ptp model. The pubsub model offers similar corresponding methods (e.g., TopicPublisher.publish).

connection is started (so it must own the session object exclusively), but in order to create an additional listener on the same session, you have to access the session object as well.

Message Listeners Need Their Consumer Objects

A message listener cannot work without the consumer object it is assigned to. When you close the consumer object, your listener won't receive any more messages. Don't forget that this also happens during garbage collection, so don't lose the reference to the consumer or your listeners won't work!

Not All Message Header Fields Are Intended for Client Use

While the *javax.jms.Message* interface defines a setter method[21] for each header field, not all fields are intended to be set by client applications directly. **Figure 22** summarizes the available fields and how their values are set.

Only fields that are intended for client use should be set directly using the corresponding setter method. For example, don't call the *Message.setJMSDeliveryMode* method to set the delivery mode. This field is set by the producer object's send or publish method automatically, so it will override whatever you specify manually. In this case, use the producer's *setDeliveryMode* method to set its default delivery mode, or use a variant of the send/publish method that allows you to specify the delivery mode for each message individually. There are similar options for the other fields as well.

Use Message Selectors to Filter Messages

With message selectors, you can define a provider-side message filter for a consumer. If your application is not interested in all messages that are sent to a

[21] A *setter* method modifies a member variable (header field in this case) of a class on your behalf since most classes don't let you modify these values directly (to preserve data integrity).

destination, just those in a subset, use a consumer with a message selector to receive only the relevant ones. Keep in mind that filtering on the client side will downgrade your performance.

Use Message Selectors to Replace Multiple Queues

We have showed you how to use message selectors in request/reply scenarios. You can use them more generally, however, to replace multiple queues with a single queue and distribute the messages using message selectors instead. This reduces the resources that are used. Remember, though, that you'll need a value in the message header or properties with which to filter the messages.

Temporary Queues vs. Message Selectors

You have seen two different approaches to implementing a JMS request/reply mechanism: using temporary queues and using regular queues with a message selector.

So when should you use which? What are the differences? Temporary queues are convenient, because you don't have the administrative overhead of a regular, administratively created queue. But they are tied to the connection object under which they are created. When the connection is closed or dies due to an error, the destination is deleted and all messages within it are lost! If this is a problem for you, use a regular destination instead. In addition, keep in mind that you can only receive messages from the temporary destination if the consumer uses the same connection as the temporary destination. This prevents other clients from reading your messages. But if your client application cannot use this connection (for whatever reason), again, use a regular destination instead.

In any case, avoid using too many temporary destinations — they consume processing time and waste resources. Use the same destination for multiple request/reply cycles by using a regular destination with a message selector. Alternatively, combine one temporary destination with a message selector to achieve the same result.

Message Selectors Are Not Optimized

Although the message selector strings resemble SQL *WHERE* clauses, the JMS provider in SAP Web AS doesn't handle them as efficiently as a DBMS. The message selector is not optimized, for example, before being evaluated. So avoid superfluous select conditions. For example, *ID = 1 AND (TYPE = 'X' OR TYPE <> 'X'*) will be slower than simply *ID = 1*.

Summary

This article has explored two important JMS features you will want to leverage in your own distributed Java applications — asynchronous message delivery and message selectors:

- ☑ Asynchronous delivery involves defining and registering a message handler class for the JMS message server (JMS provider) to actively "push" messages to the message consumer for processing, instead of "pulling" messages directly within your main program by calling a receive method. You'll find asynchronous delivery especially useful when writing server-like programs, since it frees the main thread to perform other processing, and can also be used to improve scalability if you spawn multiple threads. The key to avoiding trouble with asynchronous delivery is to avoid concurrent access to the JMS session object, which is best accomplished by providing each handler with its own session.

- ☑ Message selectors offer a powerful way for your applications to selectively identify one or more messages from a generalized queue, and facilitate high-performance architectures in many scenarios, like in our example, where we were able to replace a large number of temporary queues by a single queue plus message selectors. You'll still use temporary queues for smaller applications that you don't want to set up a permanent queue for on the JMS server, however.

The next step is for you to download the example code from **www.SAPpro.com**, if you haven't already, and try your hand at working with each of these features. In an upcoming issue of *SAP Professional Journal*, we'll explore even more ways you can take advantage of JMS programming in your distributed Java applications. In particular, we will look at how to use message-driven beans, which can improve the scalability of your applications, and also enables you to benefit from additional features of the J2EE Engine, including security, user management, and transaction handling, plus a few more advanced JMS topics that will help you on your way.

Sabine Heider is a member of SAP's Java Server Technology group, and since 2002 has worked on the integration of the JMS provider into SAP Web Application Server, as part of the JMS project. Sabine started her career with SAP in 1997 as a developer with the porting team for the DB2 on OS/390 database platform. After three years in that position, she supported strategic development projects as a technical solution specialist. Prior to joining SAP, Sabine studied physics at the University of Bonn, Germany, where she received her diploma in 1995. You can reach her at sabine.heider@sap.com.

Michael Kögel studied technical computer science at FH Konstanz in Germany. After completing his thesis on distributed computing with Java and receiving his diploma in 1998, he worked as a consultant on several major projects in banking, financials, and telecommunications. In early 2003, Michael joined SAP's Java Server Technology group, where he is currently leading the JMS project. He can be reached at michael.koegel@sap.com.

Radoslav Nikolov graduated with a degree in mathematics from Sofia University, Bulgaria, after completing a master's degree thesis on multimedia message services. He joined SAP Labs Bulgaria at the end of 2002 after working for a consulting company. Radoslav is currently a member of the Java Server Technology group, where he leads the JMS development team. He can be reached at radoslav.nikolov@sap.com.