Community Experience Distilled
Develop, debug, test, and troubleshoot Java EE 7 applications rapidly with Eclipse

Java EE Development with Eclipse
Second Edition
Ram Kulkarni

This guide provides a complete overview of developing JEE applications using Eclipse. The most prominent features of the Eclipse IDE are explained. These enable the rapid development, debugging, testing, and deployment of JEE applications. You’ll explore not just different JEE technologies and how to use them (JSP, Servlet, JSF, JPA, JDBC, EJB, web services and so on), but also suitable technologies for different scenarios.

The book starts with how to set up the development environment for JEE applications and then goes on to describe many JEE specifications in detail, with an emphasis on examples. You’ll learn how to deploy an example application in Tomcat and GlassFish Application Server.

You will create sample applications to process form data, save data to database, and publish and subscribe to queues and topics to process data asynchronously. You will also debug and profile a sample application.

Who this book is written for
If you are a Java developer who has little or no experience in JEE application development, or you have experience in JEE technology but are looking for tips to simplify and accelerate your development process, then this book is for you.

What you will learn from this book
- Set up Eclipse, Tomcat, and GlassFish server for JEE application development
- Use JSP, Servlet, JSF, and EJBs to create a user interface and write business logic
- Create JEE database applications using JDBC and JPA
- Extract support for Source Code Management in Eclipse – Git and SVN
- Handle asynchronous messages using JMS and MDBs for better scalability
- Deploy and debug JEE applications and create SOAP and RESTful web services
- Write unit tests using JUnit from Eclipse and calculate code coverage
- Develop Spring Web MVC applications using JDBC and JPA
- Troubleshoot application performance and memory issues with the help of Memory Analysis Tool

PUBLISHING


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 10 'Asynchronous Programming with JMS'
- A synopsis of the book’s content
- More information on Java EE Development with Eclipse Second Edition
About the Author

Ram Kulkarni has more than two decades of experience in developing software. He has architected and developed many enterprise web applications, client-server and desktop applications, application servers, IDE, and mobile applications. Also, he is the author of Eclipse 4 RCP Development How-to published by Packt Publishing. He blogs at ramkulkarni.com.
Java 2 Enterprise Edition (J2EE) has been used to develop enterprise applications for many years. It provides a standard technique to implement the many aspects of an enterprise application, such as handling web requests, accessing database, connecting to other enterprise systems, and implementing web services. Over the years, it has evolved and made enterprise application development easier than before. Its name has changed as well, from J2EE to JEE, after the J2EE version 1.4. Currently, it is in version 7.

Eclipse is a popular Integrated Development Environment (IDE) for developing Java applications. It has a version specific to the JEE development too, which makes it faster to write code and easier to deploy JEE applications on a server. It provides excellent debugging and unit testing support. Eclipse has a modular architecture, and many plugins are available today to extend its functionality for performing many different tasks.

This book provides you with all the information that you will need to use Eclipse to develop, deploy, debug, and test JEE applications. The focus of this book is to provide you with practical examples of how to develop applications using JEE and Eclipse. The scope of this book is not limited to JEE technologies, but covers other technologies used in the different phases of application development as well, such as source control, unit testing, and profiling.

JEE is a collection of many technologies and specifications. Some of the technologies are so vast that separate books will have to be written on them and many have been already written. This book takes the approach of providing you with a brief introduction to each technology in JEE and provides links for detailed information. Then it moves on to develop sample applications using specific technologies under discussion and explains the finer aspects of the technologies in the context of the sample applications.
This book could be useful to you if you are new to JEE and want to get started with developing JEE applications quickly. You will also find this book useful if you are familiar with JEE but looking for hands-on approach to use some of the technologies in JEE.

What this book covers

Chapter 1, Introducing JEE and Eclipse, explains in brief the different technologies in JEE and where they fit in a typical multitier JEE application. This chapter describes installing Eclipse JEE, Tomcat, GlassFish, and MySQL, which are used to develop sample applications in the later chapters.

Chapter 2, Creating a Simple JEE Web Application, describes the development of web applications using JSP, Servlet, JSTL, and JSF. It also explains how to use Maven for project management.

Chapter 3, Source Control Management in Eclipse, explains how to use the SVN and Git plugins of Eclipse for source code management.

Chapter 4, Creating a JEE Database Application, explains the creation of database applications using JDBC and JPA. You will learn how to execute SQL statements directly using JDBC, map Java classes to database tables, and set relationships between classes using the JPA and database connection pool.

Chapter 5, Unit Testing, describes how to write and run unit tests for Java applications, mock external dependencies in unit tests, and calculate the code coverage.

Chapter 6, Debugging a JEE Application, shows the techniques used to debug JEE applications and covers the debugging support of Eclipse.

Chapter 7, Creating JEE Applications with EJB, describes the use of EJBs to code business logic in the JEE applications. Also, it explains how to connect to remote EJBs using JNDI and inject EJBs into container-managed beans.

Chapter 8, Creating Web Applications with Spring MVC, describes the creation of web applications using Spring MVC and how some of the JEE technologies can be used in a Spring MVC application.

Chapter 9, Creating Web Services, explains the creation of SOAP-based and RESTful web services in JEE applications. You will learn how to consume these web services from JEE applications as well.
Chapter 10, Asynchronous Programming with JMS, shows explains how to write applications to process messages asynchronously. It describes how to program queues and topics of messaging systems using JMS and MDBs.

Chapter 11, Java CPU Profiling and Memory Tracking, describes the techniques for profiling CPU and memory in Java applications to find performance bottlenecks.
10

Asynchronous Programming
with JMS

Thus far, we have seen examples of clients making requests to the JEE server and waiting till the server sends a response back. This is a synchronous model of programming. This model of programming may not be suitable when the server takes a long time to process requests. In such cases, a client might want to send a request to the server and return it immediately without waiting for the response. The server would process the request and somehow make the result available to the client. Requests and responses in such scenarios are sent through messages. Further, there is a message broker that makes sure that messages are sent to the appropriate recipients. This is also known as message-oriented architecture. The following are some of the advantages of adopting the message-oriented architecture:

- It can greatly improve the scalability of an application. Requests are put in a queue at one end, and at the other end, there could be many handlers listening to the queue and processing the requests. As the load increases, more handlers can be added, and when the load reduces, some of the handlers can be taken off.

- Messaging systems can act as the glue between disparate software applications. An application developed using PHP can put a JSON or XML message in a messaging system, which can be processed by a JEE application.

- It can be used to implement an event-driven program. Events can be put as messages in a messaging system, and any number of listeners can process events at the other end.

- It can reduce the impact of system outages in your application because messages are persisted till they are processed.
There are many enterprise messaging systems, such as Apache ActiveMQ (http://activemq.apache.org/), RabbitMQ (https://www.rabbitmq.com/), and MSMQ (https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx). Further, the JMS (which stands for Java messaging service) specification provides a uniform interface to work with many different messaging systems. JMS is also a part of the overall Java EE specifications. Refer to http://docs.oracle.com/javaee/7/tutorial/jms-concepts.htm#BNCDQ for an overview of JMS APIs.

There are two types of message containers in messaging systems:

- **Queue**: This is used for point-to-point messaging. One message producer puts a message in a queue, and only one message consumer receives the message. There can be multiple listeners for a queue, but only one listener receives the message. However, it is not necessary that the same listener gets all the messages.

- **Topic**: This is used in a publish–subscribe type of scenario. One message producer puts messages in a topic, and many subscribers receive the message. Topics are useful for broadcasting messages.

In this chapter, we will see how to use JMS APIs for sending and receiving messages. We will use a GlassFish server, which also has a built-in JMS provider. We will use JMS APIs to implement a use case in the Course Management application, the same application that we have been building in the other chapters of this book.

**Steps to send and receive messages using JMS**

However, before we start using JMS APIs, let's take a look at the generic steps involved in using them. The following steps show how to send a message to a queue and receive it; however, the steps for topic are similar but with appropriate topic-related classes.

1. Look up ConnectionFactory using JNDI:

   ```java
   InitialContext ctx = new InitialContext();
   QueueConnectionFactory connectionFactory = (QueueConnectionFactory)initCtx.lookup("jndi_name_of_connection_factory");
   ```
2. Create JMS connection and start it:
   QueueConnection con = connectionFactory.createQueueConnection();
   con.start();

3. Create JMS session:
   QueueSession session = con.createQueueSession(false,
   Session.AUTO_ACKNOWLEDGE);

4. Look up JMS Queue/Topic:
   Queue queue = (Queue)initCtx.lookup("jndi_queue_name");

5. For sending messages:
   - Create a sender:
     QueueSender sender = session.createSender(queue);
   - Create a message. The message could be of any of the following types: TextMessage/ObjectMessage/MapMessage/BytesMessage/StreamMessage:
     TextMessage textMessage = session.createTextMessage("Test Message");
   - Send the message:
     sender.send(textMessage);
   - Close the connection when no longer needed:
     con.close();

6. For receiving messages:
   - Create a receiver:
     //create a new session before creating the receiver.
     QueueReceiver receiver = session.createReceiver(queue);
   - Register a message listener or call a receive method:
     receiver.setMessageListener(new MessageListener() {
       @Override
       public void onMessage(Message message) {
         try {
           String messageTxt = ((TextMessage)message).getText();
         }
     });
Asynchronous Programming with JMS

```java
//process message
} catch (JMSException e) {
    //handle exception

}));

```

- Alternatively, you can use any variation of the receive method:

```java
Message message = receiver.receive(); //this blocks the thread till message is received
Or:
Message message = receiver.receive(timeout);
Or:
Message message = receiver.receiveNoWait(); //returns null if no message is available.
```

- In a JEE application that uses EJB, it is recommended to use MDBs (which stands for message-driven beans). We will see an example of MDBs later in this chapter.

7. When done, close connection. This stops message listeners too:

```java
con.close();
```

Some of the steps can be skipped when JMS annotations are used or when MDBs are used to receive messages. We will see examples of these later.

Now, let's create a working example of sending and receiving messages using JMS. Make sure that you have installed the GlassFish application server (refer to the Installing the GlassFish section in Chapter 1, Introducing JEE and Eclipse) and configured it in Eclipse JEE (refer to the Configuring the GlassFish server in Eclipse section in Chapter 7, Creating JEE Applications with EJB). The use case that we will implement in this example is adding a new course. Although this is not a strong use case for asynchronous processing, we will assume that this operation takes a long time and needs to be handled asynchronously.
Creating queues and topics in GlassFish

Let's create one queue and one topic in GlassFish. Make sure that the GlassFish server is running. Open the GlassFish admin console. You can right-click the GlassFish server instance configured in Eclipse (in the Servers view) and select GlassFish | View Admin Console. This opens the admin console in the built-in Eclipse browser. If you want to open it outside Eclipse in a browser, then browse to http://localhost:4848/ (assuming a default GlassFish installation).

We will first create a JMS connection factory. In the admin console, go to the Resources | JMS Resources | Connection Factories page. Click the New button to create a new connection factory.

![Figure 10.1 Create JMS Connection Factory](image-url)
Enter JNDI Name of the factory as jms/CourseManagementCF and select javax.jms.ConnectionFactory as Resource Type. Leave the default values for Pool Settings. Click OK.

To create queues and topics, go to the Resources | JMS Resources | Destination Resources page. Click the New button.

Enter JNDI Name of the queue as jms/courseManagementQueue and Physical Destination Name as CourseManagementQueue, and select javax.jms.Queue as Resource Type. Click OK to create the queue.
Similarly, create a topic by entering JNDI Name as jms/courseManagementTopic and Physical Destination Name as CourseManagementTopic, and select javax.jms.Topic as Resource Type.

You should now have one queue and one topic configured in the Destination Resources page.

![JMS Destination Resources](image)

**JMS Destination Resources**
JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify its properties.

![Destination Resources Table](image)

<table>
<thead>
<tr>
<th>Select</th>
<th>JNDI Name</th>
<th>Enabled</th>
<th>Resource Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jms/courseManagementQueue</td>
<td>✔</td>
<td>javax.jms.Queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jms/courseManagementTopic</td>
<td>✔</td>
<td>javax.jms.Topic</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.3 Queue and topic Created in GlassFish

**Creating a JEE project for a JMS application**

We will see examples of using JMS APIs in three different ways.

In the first example, we will create a simple addCourse.jsp page, one JSP bean, and one Service class that actually perform JMS tasks.

In the second example, we will use JSF and managed beans. We will use JMS APIs in the managed beans. We will also see how to use JMS annotations in JSF managed beans.

In the last example, we will use MDB to consume JMS messages.
Let's start with the first example that uses JSP, bean, and JMS APIs. Create a web project by selecting **File | New | Dynamic Web Project**.

Enter **Project name** as `CourseManagementJMSWeb`. Make sure that **Target runtime** is `Glassfish 4`. Click **Next**, and accept all default options. Click **Finish** to create the project.

---

[364]
Creating a JMS application using JSP and JSP bean

Let's first create JSP that displays a form to enter course details and a Submit button. We will have a JSP bean to process the form data. Right-click on the WebContent folder under the project in the Project Explorer view and select New | JSP File. Create a JSP file named addCourse.jsp.

We will now create CourseDTO and JSP bean called CourseJSPBean. Create the CourseDTO class in the packt.jee.eclipse.jms.dto package. Add the id, name, and credits properties, and the getters and setters for them:

```java
import java.io.Serializable;
public class CourseDTO implements Serializable {
    private static final long serialVersionUID = 1L;
    private int id;
    private String name;
    private int credits;

    //getters and setters follow
}
```

Create CourseJSPBean in the packt.jee.eclipse.jms.jsp.beans package:

```java
import packt.jee.eclipse.jms.dto.CourseDTO;

public class CourseJSPBean {
    private CourseDTO course = new CourseDTO();

    public void setId(int id) {
        course.setId(id);
    }
    public String getName() {
        return course.getName();
    }
    public void setName(String name) {
        course.setName(name);
    }
    public int getCredits() {
        return course.getCredits();
    }
}
```
public void setCredits(int credits) {
    course.setCredits(credits);
}
public void addCourse() {
    //TODO: send CourseDTO object to a JMS queue
}
\[\text{courseService.addCourse()}\]
\begin{quote}
\texttt{Course detailed are sent to a JMS Queue. It will be processed later}<\texttt{}/b>
\end{quote}
</c:if>

<h2>New Course:</h2>

\begin{verbatim}
<!-- Course data input form -->
<form method="post">
<table>
<tr>
<td>Name:</td>
<td>
<input type="text" name="course_name">
</td>
</tr>
<tr>
<td>Credits:</td>
<td>
<input type="text" name="course_credits">
</td>
</tr>
<tr>
<td colspan="2">
<button type="submit" name="submit">Add</button>
</td>
</tr>
</table>
</form>
\end{verbatim}

At the top of the JSP file, we check whether the form is submitted. If yes, then we create an instance of \texttt{CourseJSPBean} and set its properties with values from the form submission. Then, we call the \texttt{addCourse} method of the bean.
Executing addCourse.jsp

We still haven’t added any code to put the Course object in the JMS queue. However, if you want to test the JSP and bean, add the project to the GlassFish server configured in Eclipse. To do this, right-click on the configured server in the Servers view of Eclipse and select the Add Remove … option. Select the web project that we created above and click Finish. Make sure that the server is started and the status is [Started, Synchronized].

If the status is Republish, then right-click on the server and select the Publish option. If the status is Restart, right-click on the server and select the Restart option. You may not have to do this immediately after adding a project, but later when we make changes to the code, you may have to republish or restart the server or both. So, keep a watch on the server status before you execute the code in Eclipse.

To execute addCourse.jsp, right-click on the file in either Project Explorer or the editor, and select the Run As | Run on Server option. This will open the built-in Eclipse browser and open JSP in it. You should see the form for adding the course details. If you click the Submit button, you should see the message that we added in JSP when the form is submitted.

Let’s now add a class to send the course details to the JMS queue.

Implementing a JMS queue sender class

Create the CourseQueueSender class in the packt.jee.eclipse.jms package.

```java
package packt.jee.eclipse.jms;

//skipped imports

public class CourseQueueSender {
    private QueueConnection connection;
    private QueueSession session;
    private Queue queue;

    public CourseQueueSender() throws Exception {
```
//Create JMS Connection, session, and queue objects
InitialContext initCtx = new InitialContext();
QueueConnectionFactory connectionFactory =
    (QueueConnectionFactory)initCtx.lookup("jms/CourseManagementCF");
connection = connectionFactory.createQueueConnection();
connection.start();
session = connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
queue = (Queue)initCtx.lookup("jms/courseManagementQueue");
}

public void close() {
    if (connection != null) {
        try {
            connection.close();
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}

@Override
protected void finalize() throws Throwable {
    close(); //clean up
    super.finalize();
}

public void sendAddCourseMessage (CourseDTO course) throws Exception {
    //Send CourseDTO object to JMS Queue
    QueueSender sender = session.createSender(queue);
    ObjectMessage objMessage = session.createObjectMessage(course);
    sender.send(objMessage);
}

In the constructor, we look up the JMS connection factory and create the connection. We then create a JMS session and look up queue with the JNDI name that we used for creating the queue in a previous section.
Note that we did not specify any configuration properties when constructing InitialContext. This is because the code is executed in the same instance of the GlassFish server that hosts the JMS provider. If you are connecting to a JMS provider hosted in a different GlassFish server, then you will have to specify the configuration properties, particularly for the remote host. For example:

```java
Properties jndiProperties = new Properties();
jndiProperties.setProperty("org.omg.CORBA.ORBInitialHost", "<remote_host>");
//target ORB port. default is 3700 in Glassfish
jndiProperties.setProperty("org.omg.CORBA.ORBInitialPort", "3700");

InitialContext ctx = new InitialContext(jndiProperties);
```

The `CourseQueueSender.sendAddcourseMessage` method creates a `QueueSender` object and `ObjectMessage`. Because the producer and the consumer of the message in this example are in Java, we use `ObjectMessage`. However, if you are to send a message to a messaging system where the message is going to be consumed by a non-Java consumer, then you could create JSON or XML from the Java object and send `TextMessage`. We have already seen how to serialize Java objects to JSON and XML by using JAXB in Chapter 9, Creating Web Services.

Now, let's modify `addCourse` in `CourseJSPBean` to use the preceding class to send JMS messages. Note that we could create an instance of `CourseQueueSender` in the bean class, `CourseJSPBean`, but the bean is created every time a page is requested. So, `CourseQueueSender` will be created frequently and the lookup for the JMS connection factory and the queue will also take place frequently, which is not necessary. Therefore, we will create an instance of `CourseQueueSender` and save it in the HTTP session. Then, we will modify the `addCourse` method to take `HttpServletRequest` as a parameter. We will also get the `HttpSession` object from the request.

```java
public void addCourse(HttpServletRequest request) throws Exception {
    //get HTTP session
    HttpSession session = request.getSession(true);
    //look for instance of CourseQueueSender in Session
    CourseQueueSender courseQueueSender = (CourseQueueSender)session.getAttribute("CourseQueueSender");
    if (courseQueueSender == null) {
        //Create instance of CourseQueueSender and save in Session
        courseQueueSender = new CourseQueueSender();
    }
    //use instance of CourseQueueSender to send message
    courseQueueSender.sendAddcourseMessage(request.getParameter("course")
```

[370]
If we don't find the CourseQueueSender object in the session, then we will create one and save it in the session.

We need to modify the call to the addCourse method from addCourse.jsp. Currently, we do not pass any argument to the method. However, with the preceding changes to the addCourse method, we need to pass the HttpServletRequest object to it. JSP has a build-in property called pageContext that provides access to the HttpServletRequest object. So, modify the code in addCourse.jsp where courseService.addCourse is called as follows:

```jsp
<%! Call addCourse method of the bean -->
${courseService.addCourse(pageContext.request)}
```

We can test our code at this point, but although a message is sent to the queue, we haven't implemented any consumer to receive a message from the queue. So, let's implement a JMS queue consumer for our Course queue.

**Implementing a JMS queue receiver class**

Create the CourseQueueReceiver class in the packt.jee.eclipse.jms package.

```java
public class CourseQueueReceiver {
    private QueueConnection connection;
    private QueueSession session;
    private Queue queue;
    private String receiverName;

    public CourseQueueReceiver(String name) throws Exception{
        //save receiver name
```
this.receiverName = name;

//look up JMS connection factory
InitialContext initCtx = new InitialContext();
QueueConnectionFactory connectionFactory = (QueueConnectionFactory)initCtx.lookup("jms/CourseManagementCF");

//create JMS connection
connection = connectionFactory.createQueueConnection();
connection.start();

//create JMS session
session = connection.createQueueSession(false,
  Session.AUTO_ACKNOWLEDGE);
//look up queue
queue = (Queue)initCtx.lookup("jms/courseManagementQueue");

topicPublisher = new CourseTopicPublisher();

QueueReceiver receiver = session.createReceiver(queue);
//register message listener
receiver.setMessageListener(new MessageListener() {

@Override
  public void onMessage(Message message) {
    //we expect ObjectMessage here; of type CourseDTO
    //skipping validation
    try {
      CourseDTO course = (CourseDTO)
        ((ObjectMessage)message).getObject();
      //process addCourse action. For example, save it in the
      database

      System.out.println("Received addCourse message for Course
      name - " +
        course.getName() + " in Receiver " + receiverName);
    } catch (Exception e) {
      e.printStackTrace();
      //TODO: handle and log exception
    }
  }
});

public void stop() {
  if (connection != null) {

try {
    connection.close();
} catch (JMSException e) {
    e.printStackTrace();
    //TODO: log exception
}
}

The code to look up the connection factory and the queue is similar to that in CourseQueueSender. Note that the constructor takes a name argument. We don't really need to use a JMS API, but we will use it as an identifier for instances of the CourseQueueReceiver class. We register a message listener in the constructor, and in the onMessage method of the listener class, we get the CourseDTO object from the message and print the message to the console. This message will appear in GlassFish console in Eclipse when we execute the code. To keep the example simple, we have not implemented the code to save the Course information to the database, but you can do so by using JDBC or JDO APIs that we have already learnt in Chapter 4, Creating a JEE Database Application.

We need to instantiate this class at the application startup so that it will start listening for messages. One way to implement this is in a Servlet that loads on startup.

Create the JMSReceiverInitServlet class in the packt.jee.eclipse.jms.servlet package. We will mark this Servlet to load at startup by using annotations and instantiate CourseQueueReceiver in the init method.

```java
package packt.jee.eclipse.jms.servlet;

//@skipped imports

@WebServlet(urlPatterns="/JMSReceiverInitServlet",
loadOnStartup=1)
public class JMSReceiverInitServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private CourseQueueReceiver courseQueueReceiver = null;

    public JMSReceiverInitServlet() {
        super();
    }

    @Override
    public void init(ServletConfig config) throws ServletException {
    
    }
```
super.init(config);  
try {  
courseQueueReceiver = new CourseQueueReceiver("Receiver1");  
} catch (Exception e) {  
log("Error creating CourseQueueReceiver", e);  
}  

@Override  
public void destroy() {  
if (courseQueueReceiver != null)  
courseQueueReceiver.stop();  
super.destroy();  
}  

Publish the project again in the server and execute addCourse.jsp (see the Executing addCourse.jsp section). Switch to the Console view in Eclipse. You should see the message that we printed in the onMessage method in CourseQueueReceiver.

![Console output](image.png)

**Figure 10.6 Example of console message from JMS receiver class**

### Adding multiple queue listeners

Queues are meant for point-to-point communication, but this does not mean that there can't be more than one listener for a queue. However, only one listener gets a message. Further, it is not guaranteed that the same listener will get the message every time. If you want to test this, add one more instance of `CourseQueueReceiver` in `JMSReceiverInitServlet`. Let's add the second instance with a different name, say `Receiver2`.

```java
@WebServlet(urlPatterns="/JMSReceiverInitServlet",
loadOnStartup=1)
public class JMSReceiverInitServlet extends HttpServlet {
    private CourseQueueReceiver courseQueueReceiver = null;
```
private CourseQueueReceiver courseQueueReceiver1 = null;

@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
        //first instance of CourseQueueReceiver
        courseQueueReceiver = new CourseQueueReceiver("Receiver1");
        //create another instance of CourseQueueReceiver with a
different name
        courseQueueReceiver1 = new CourseQueueReceiver("Receiver2");
    } catch (Exception e) {
        log("Error creating CourseQueueReceiver", e);
    }
}

@Override
public void destroy() {
    if (courseQueueReceiver != null)
        courseQueueReceiver.stop();
    if (courseQueueReceiver1 != null)
        courseQueueReceiver1.stop();
    super.destroy();
}

//rest of the code remains the same

Republish the project, execute addCourse.jsp, and add a few courses. Check
the Console messages. You may see that some of the messages were received by
Receiver1 and the others by Receiver2.

Figure 10.7 Console output showing multiple JMS receivers listening to a JMS queue
Implementing the JMS topic publisher

Let’s say that we want to inform a bunch of applications when a new course is added. Such use cases can be best implemented by JMS topic. Topic can have many subscribers. When a message is added to a topic, all subscribers are sent the same message. This is unlike queue where only one queue listener gets a message.

Steps to publish messages to topic and subscribe for messages are very similar to those for queue, except for the different classes, and in some cases, different method names.

Let’s implement a topic publisher, which we will use when a message for adding course is successfully handled in the `onMessage` method of the listener class implemented in `CourseQueueReceiver`.

Create `CourseTopicPublisher` in the `packt.jee.eclipse.jms` package.

```java
package packt.jee.eclipse.jms;

//skipped imports

public class CourseTopicPublisher {
    private TopicConnection connection;
    private TopicSession session;
    private Topic topic;

    public CourseTopicPublisher() throws Exception {
        InitialContext initCtx = new InitialContext();
        TopicConnectionFactory connectionFactory =
            (TopicConnectionFactory)initCtx.lookup("jms/CourseManagementCF");
        connection = connectionFactory.createTopicConnection();
        connection.start();
        session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        topic = (Topic)initCtx.lookup("jms/courseManagementTopic");
    }

    public void close() {
        if (connection != null) {
            try {
                connection.close();
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
}
```
public void publishAddCourseMessage (CourseDTO course) throws Exception {
    TopicPublisher sender = session.createPublisher(topic);
    ObjectMessage objMessage =
        session.createObjectMessage(course);
    sender.send(objMessage);
}

The code is quite simple and self-explanatory. Let's now modify the queue receiver class that we implemented, CourseQueueReceiver, to publish a message to the topic from the onMessage method, after the message from the queue is handled successfully.

public class CourseQueueReceiver {

    private CourseTopicPublisher topicPublisher;

    public CourseQueueReceiver(String name) throws Exception{

        //code to lookup connection factory, create session,
        //and look up queue remains unchanged. Skipping this code

        //create topic publisher
        topicPublisher = new CourseTopicPublisher();

        QueueReceiver receiver = session.createReceiver(queue);
        //register message listener
        receiver.setMessageListener(new MessageListener() {

            @Override
            public void onMessage(Message message) {
                //we expect ObjectMessage here; of type CourseDTO
                //Skipping validation
                try {
                    //code to process message is unchanged. Skipping it

                    //publish message to topic
                }
            }
        });
    }
}
Asynchronous Programming with JMS

if (topicPublisher != null)
    topicPublisher.publishAddCourseMessage(course);

} catch (Exception e) {
    e.printStackTrace();
    //TODO: handle and log exception
}
}

//remaining code is unchanged. Skipping it

Implementing the JMS topic subscriber

We will now implement a topic subscriber class. Create the CourseTopicSubscriber class in the packt.jee.eclipse.jms package.

package packt.jee.eclipse.jms;
//skipping imports
public class CourseTopicSubscriber {

    private TopicConnection connection;
    private TopicSession session;
    private Topic topic;
    private String subscriberName;

    public CourseTopicSubscriber(String name) throws Exception{
        thissubscriberName = name;

        InitialContext initCtx = new InitialContext();
        TopicConnectionFactory connectionFactory = (TopicConnectionFactory)initCtx.lookup("jms/CourseManagementCF");
        connection = connectionFactory.createTopicConnection();
        connection.start();
        session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        topic = (Topic)initCtx.lookup("jms/courseManagementTopic");

        TopicSubscriber subscriber = session.createSubscriber(topic);
subscriber.setMessageListener(new MessageListener() {
  
    @Override
    public void onMessage(Message message) {
        // we expect ObjectMessage here; of type CourseDTO
        // skipping validation
        try {
            CourseDTO course = (CourseDTO)
                ((ObjectMessage)message).getObject();
            // process addCourse action. For example, save it in database
            System.out.println("Received addCourse notification for Course name - "
                + course.getName() + " in Subscriber " + subscriberName);
        } catch (JMSException e) {
            e.printStackTrace();
            // TODO: handle and log exception
        }
    }
});

public void stop() {
    if (connection != null) {
        try {
            connection.close();
        } catch (JMSException e) {
            e.printStackTrace();
            // TODO: log exception
        }
    }
}

Again, JMS APIs to subscribe to a topic are similar to those in CourseQueueReceiver, but with different class names and method names. We also identify subscribers with names so that we know which instance of the class receives the messages.
In the preceding example, we created the topic subscriber by calling `TopicSession.createSubscriber`. In this case, the subscriber will receive messages from the topic as long as the subscriber is active. If the subscriber becomes inactive and then active again, it loses messages published by the topic during that period. If you want to make sure that the subscriber receives all messages, you need to create a durable subscription using `TopicSession.createDurableSubscriber`. Along with the topic name, this method takes the subscriber name as the second argument. Refer to https://docs.oracle.com/javaee/7/api/javax/jms/TopicSession.html#createDurableSubscriber-javax.jms.Topic-java.lang.String- for more information.

We will create two instances of this class (so there would be two topic subscribers) in `JMSReceiverInitServlet`, so that subscribers start listening for messages on the application start (the Servlet is loaded on startup).

```java
@WebServlet(urlPatterns="/JMSReceiverInitServlet", loadOnStartup=1)
public class JMSReceiverInitServlet extends HttpServlet {
    private CourseQueueReceiver courseQueueReceiver = null;
    private CourseTopicSubscriber courseTopicSubscriber = null;
    private CourseQueueReceiver courseQueueReceiver1 = null;
    private CourseTopicSubscriber courseTopicSubscriber1 = null;

    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        try {
            courseQueueReceiver = new CourseQueueReceiver("Receiver1");
            courseQueueReceiver1 = new CourseQueueReceiver("Receiver2");
            courseTopicSubscriber = new CourseTopicSubscriber("Subscriber1");
            courseTopicSubscriber1 = new CourseTopicSubscriber("Subscriber2");

        } catch (Exception e) {
            log("Error creating CourseQueueReceiver", e);
        }
    }

    //remaining code is unchanged. Skipping it

```
Therefore, now, we have two queue listeners and two topic listeners ready when the application starts. Republish the project, execute addCourse.jsp, and add a course. Check messages in the **Console** view of Eclipse. You will see that the message published in the topic is received by all subscribers, but the same message published in queue is received by only one receiver.

![Console output showing multiple JMS receivers listening to JMS queue and topic](image)

**Figure 10.8** Console output showing multiple JMS receivers listening to JMS queue and topic

### Creating a JMS application using JSF and managed beans

In this section, we will see how to create a JMS application by using JSF and managed beans. With managed beans, we can reduce the code that you write to using JMS APIs, because we can use annotations to inject objects such as the JMS connection factory, queue, and topic. Once we obtain reference to these objects, steps to send or receive data are the same as those discussed in the previous section. Therefore, our examples in this section do not list the entire code. For the complete source code, download the source code for this chapter.

To prepare our project for using JSF, we need to create web.xml and add the JSF Servlet definition and mapping in it. Right-click on the project and select the **Java EE Tools | Generate Deployment Descriptor Stub** option. This creates `web.xml` in the `WebContent/WEB-INF` folder. Add the following Servlet definition and mapping (within the `web-app` tag) in `web.xml`:

```xml
<servlet>
  <servlet-name>JSFServlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>JSFServlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```
Asynchronous Programming with JMS

We will first create three managed beans for JSF. The first one is CourseManagedMsgSenderBean. The second one is CourseManagedMsgReceiverBean, and the last one is CourseJSFBean, which will be referenced from the JSF page.

Create the CourseManagedMsgSenderBean class in the packt.jee.eclipse.jms.jsf_bean package:

```java
package packt.jee.eclipse.jms.jsf_bean;

//@ManagedBean(name="courseMessageSender")
//@SessionScoped
public class CourseManagedMsgSenderBean {

//@Resource(name = "jms/CourseManagementCF")
private QueueConnectionFactory connectionFactory;
//@Resource(lookup = "jms/courseManagementQueue")
private Queue queue;

QueueConnection connection;
QueueSession session;
Exception initException = null;

//@PostConstruct
public void init() {
    try {
        connection = connectionFactory.createQueueConnection();
        connection.start();
        session = connection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
    } catch (Exception e) {
        initException = e;
    }
}

//@PreDestroy
```
public void cleanup() {
    if (connection != null) {
        try {
            connection.close();
        } catch (JMSException e) {
            e.printStackTrace();
            //TODO: log exception
        }
    }
}

public void addCourse(CourseDTO courseDTO) throws Exception {
    if (initException != null)
        throw initException;

    QueueSender sender = session.createSender(queue);
    ObjectMessage objMessage = session.createObjectMessage(courseDTO);
    sender.send(objMessage);
}

Notice that the JMS connection factory and queue objects are injected using the @Resource annotation. We have used the @PostConstruct annotation to create a JMS connection and session and the @PreDestroy annotation for a clean-up operation. The addCourse method is similar to the code that we have already implemented in the CourseQueueSender class in a previous section.

Now, create a JMS message receiver class. Create the CourseManagedMsgReceiverBean class in the packt.jee.eclipse.jms.jsf_bean package.

package packt.jee.eclipse.jms.jsf_bean;

//skipped imports

@ManagedBean(name="courseMessageReceiver")
@ApplicationScoped
public class CourseManagedMsgReceiverBean {

    @Resource(name = "jms/CourseManagedCF")
    private QueueConnectionFactory connectionFactory;
Asynchronous Programming with JMS

```java
@Resource(lookup = "jms/courseManagementQueue")
private Queue queue;

QueueConnection connection;
QueueSession session;
Throwable initException = null;

@PostConstruct
public void init() {
    try {
        connection = connectionFactory.createQueueConnection();
        connection.start();
        session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

        // skipped code to create receiver and add MessageListener
        // the code is same as in the constructor of CourseQueueReceiver
    } catch (Throwable e) {
        initException = e;
    }
}

// skipped @PreDestroy method to close connection
```

In this class also, JMS resources are injected using the @Resource tags. The @ PostConstruct method creates a connection, session, and a receiver. It also registers MessageListener. The code is similar to what we wrote in the constructor of CourseQueueReceiver, so some of the code is skipped in the previous listing. Please download the source code for this chapter to see the complete source code.

We need to create an instance of this class on application startup. We have already created JMSReceiverInitServlet that loads on startup in a previous section. We also instantiated the course and topic listeners that we created earlier in the init method of this Servlet. So, now, let's create an instance of CourseManagedMsgReceiverBean in the init method.

```java
@override
public void init(ServletConfig config) throws ServletException {
    super.init(config);

    // get JSP Managed bean for receiving Course messages
    FacesContext context = FacesContext.getCurrentInstance();
    // Evaluating #{courseMessageReceiver} expression will
```
Chapter 10

// instantiate CourseManagedMsgReceiverBean and start message listener
context.getApplication().evaluateExpressionGet(context, "#{courseMessageReceiver}", CourseManagedMsgReceiverBean.class);

Note that if you want only CourseManagedMsgReceiverBean to receive messages from the course queue, then remove the previously added message receivers from the init method.

Now, let's create the CourseJSFBean class in the packt.jee.eclipse.jms.jsf_bean package.

package packt.jee.eclipse.jms.jsf_bean;

// skipped imports

@ManagedBean(name="course")
@RequestScoped
public class CourseJSFBean {
    private CourseDTO courseDTO = new CourseDTO();

    @ManagedProperty(value="#{courseMessageSender}")
    private CourseManagedMsgSenderBean courseMessageSender;

    // skipped getters and setters

    public void setCourseMessageSender(CourseManagedMsgSenderBean
        courseMessageSender) {
        this.courseMessageSender = courseMessageSender;
    }

    public void addCourse() throws Exception {
        // skipping validation
        // TODO: handle exception properly and show error message
        courseMessageSender.addCourse(courseDTO);
    }
}

CourseJSFBean obtains a reference to CourseManagedMsgSenderBean by using the @ManagedBean annotation. We need to provide the setter for CourseManagedMsgSenderBean so that the container can set its value. The addCourse method simply calls the same named method in CourseManagedMsgSenderBean.
Finally, create addCourse.xhtml in the WebContents folder.

```html
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:h="http://java.sun.com/jsf/html">

<head>
    <title>Add Course</title>
</head>

<body>
    <h2>Course Details</h2>
    <h:form>
        <table>
            <tr>
                <td>Name:</td>
                <td>
                    <h:inputText id="course_name" value="#{course.name}"/>
                </td>
            </tr>
            <tr>
                <td>Credits:</td>
                <td>
                    <h:inputText id="course_credits"
                                  value="#{course.credits}"/>
                </td>
            </tr>
            <tr>
                <td colspan="2">
                    <h:commandButton value="Submit"
                                      action="#{course.addCourse}"/>
                </td>
            </tr>
        </table>
    </h:form>
</body>
</html>
```

Form fields are bound to fields in CourseJSFBean. When the Submit button is clicked, the addCourse method of the same bean is called, which puts a message in the JMS queue.

Republish the project and execute addCourse.xhtml by right-clicking it and selecting Run As | Run on Server. Add a course and see the message printed (from MessageListener in CourseManagedMsgReceiverBean) in the GlassFish Console view of Eclipse.
Consuming JMS messages using MDB

Message-driven beans (MDBs) make consuming JMS messages a lot easier. With just a couple of annotations and implementing the `onMessage` method, you can make any Java object a consumer of the JMS messages. In this section, we will implement an MDB to consume messages from the `Course` queue. To implement MDB, we need to create an EJB project. Select File | New | EJB Project from the main menu.

![Create EJB project to implement MDB](image)
Asyncronous Programming with JMS

Enter Project name as CourseManagementEJB. Click Next. Accept the default values on the subsequent pages, and click Finish on the last page.

Right-click on the project, and select the New | Message-Driven Bean option. This opens the MDB creation wizard.

Figure 10.10 MDB creation wizard: class file information

Enter packt.jee.eclipse.jms.mdb as Java package and CourseMDB as Class name. Keep Destination type as Queue.

Destination name is the physical destination name that we specified when creating the queue and is not the JNDI name.
Enter CourseManagementQueue as **Destination type**. Click **Next**. Accept the default values on the second page and click **Finish**. The wizard creates the following code:

```java
@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType",
                                 propertyValue = "javax.jms.Queue"),
        @ActivationConfigProperty(propertyName = "destination",
                                 propertyValue = "CourseManagementQueue")
    },
    mappedName = "jms/courseManagementQueue"
)
public class CourseMDB implements MessageListener {

    /**
     * Default constructor.
     */
    public CourseMDB() {
        // TODO Auto-generated constructor stub
    }

    /**
     * @see MessageListener#onMessage(Message)
     */
```
Asynchronous Programming with JMS

```java
public void onMessage(Message message) {
    System.out.println("addCourse message received in CourseMDB");
}

The class is annotated with @MessageDriven with activationConfig with the JMS destination parameters specified in the wizard. It also creates the onMessage method. In this method, we just print a message that the MDB received for adding a course. To process ObjectMessage in this class, we will have to refactor the CourseDTO class to a shared .jar between EJB and the web project. This is left to the readers as an exercise.

At runtime, the JEE container creates a pool of MDB objects for a single MDB class. An incoming message can be handled by any one of the instances of MDB in the pool. This can help in building a scalable message processing application.

Summary

Messaging systems can be powerful tools for integrating disparate applications. They provide an asynchronous model of programming. The client does not wait for the response from the server and the server does not necessarily process requests at the same time that the client sends them. Messaging systems can also be useful for building scalable applications and batch processing. JMS provides uniform APIs to access different messaging systems.

In this chapter, we discussed how to send and receive messages from queue and to publish and subscribe messages from topic. There are many different ways to use JMS APIs. We started with the basic JMS APIs and then, discussed how annotations can help reduce some of the code. We also discussed how to use MDBs to consume messages.

In the next chapter, we will see some of the techniques and tools used for profiling the CPU and memory usage in Java applications.
Where to buy this book

You can buy Java EE Development with Eclipse Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.