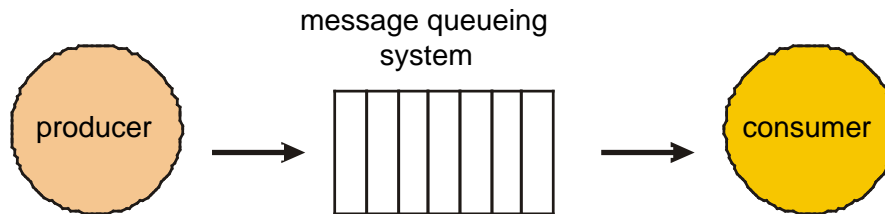


JMS and Message-Driven EJBs

- What is Messaging?
- JMS Fundamentals
- Message-Driven EJBs

What is Messaging?

- Messaging lets you create loosely coupled systems
- This is especially useful in situations where the "client" and "server" run at different processing rates (e.g. GUI client and batch processing server) or to communicate with legacy systems

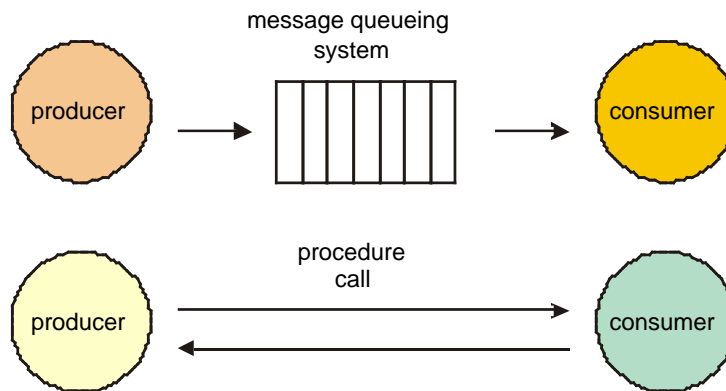


22 - 2

You can think of messaging as "email for applications" (as opposed to human readable email).

Messaging vs Method Calls

- Normal Java method calls are synchronous - the caller can't do anything until the method returns
- Messaging is inherently asynchronous, but you can simulate synchronous operation if necessary with a response message



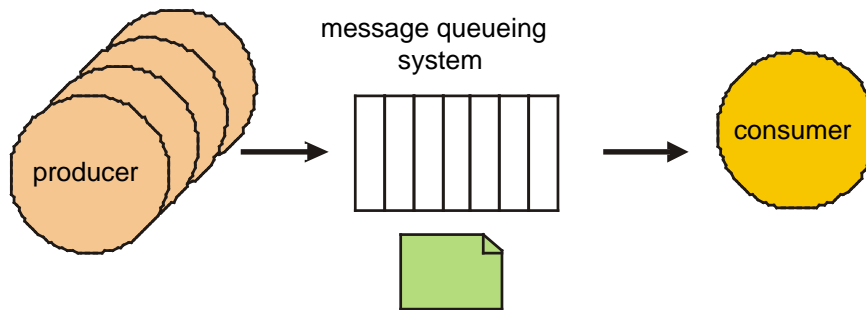
22 - 3

Procedure (method) calls are a natural part of programming languages such as Java. In addition, programmers have created infrastructures for remote procedure calls (RPC) including Java RMI, CORBA and SOAP.

However, even RPCs appear to be synchronous to the caller and thus are best suited for tightly coupled systems where the caller has detailed knowledge about the server (e.g. method and argument names).

Point to Point Messaging

- In a point-to-point setup, each message is delivered to exactly **one** consumer

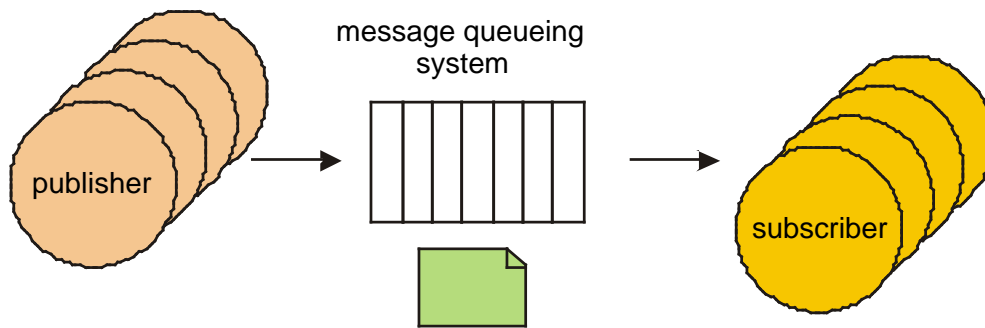


22 - 4

P2P messaging is similar to simple email, where each message goes to exactly one recipient.

Publish and Subscribe Messaging

- In a publish/subscribe setup, **all** registered consumers receive each message



22 - 5

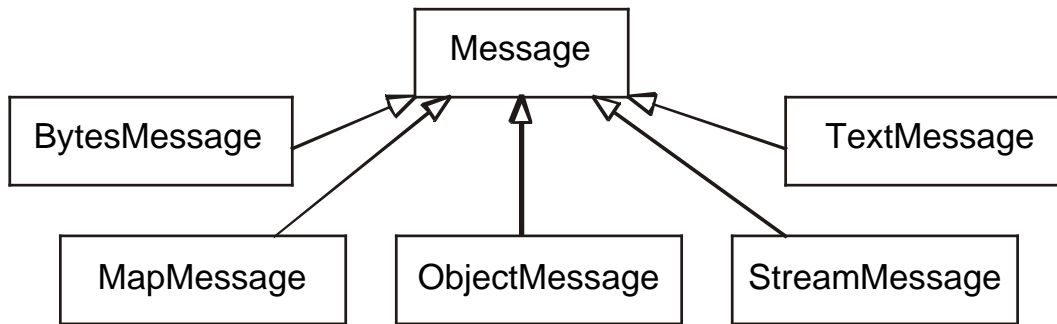
Pub/Sub messaging is similar to an Internet email "listserve", where each message posted to the list is passed to all subscribers.

What is the Java Message Service?

- The Java Message Service (JMS) provides a system-neutral interface so that Java programs can interact with messaging systems
- JMS is not itself a messaging service - like JDBC, it simply provides a standard way to communicate with any messaging service that can act as a JMS provider
- All J2EE 1.3-compliant platforms must support JMS, including message-driven EJBs

JMS Message Types

- To provide flexibility, JMS supports various message types



22 - 7

A `BytesMessage` object is used to send a message containing a stream of uninterpreted bytes.

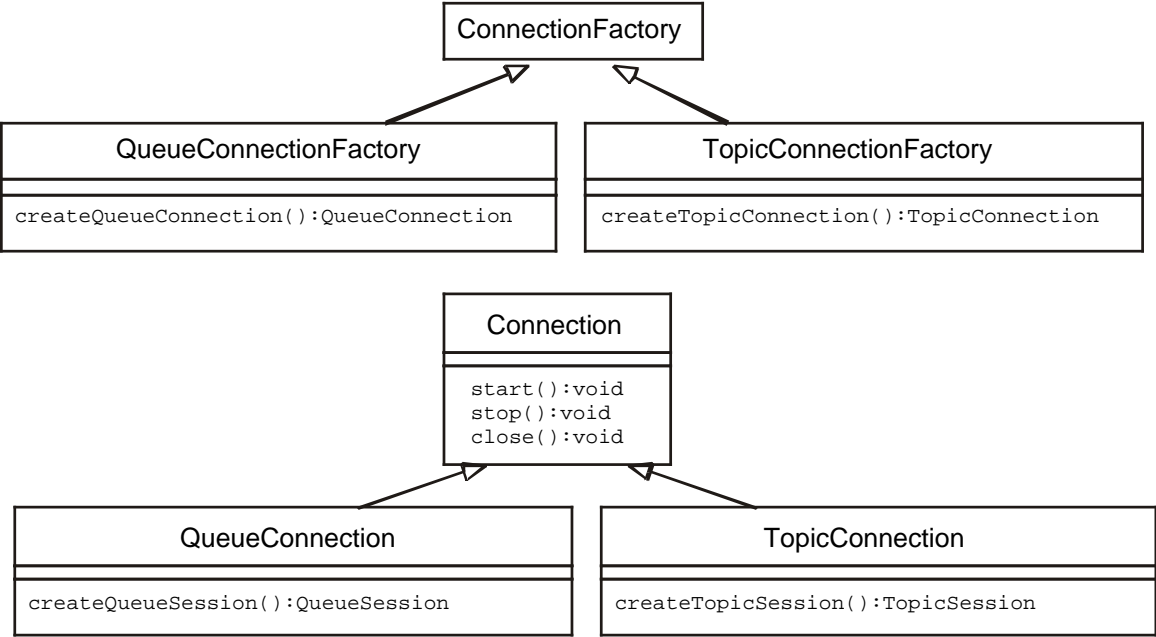
A `MapMessage` object is used to send a set of name-value pairs.

An `ObjectMessage` object is used to send a message that contains a serializable Java object.

A `StreamMessage` object is used to send a stream of primitive types in the Java programming language.

A `TextMessage` object is used to send a message containing a `java.lang.String`.

Factories and Connections



22 - 8

In JMS, we refer to point-to-point as "queueing" and pub/sub as "topics".

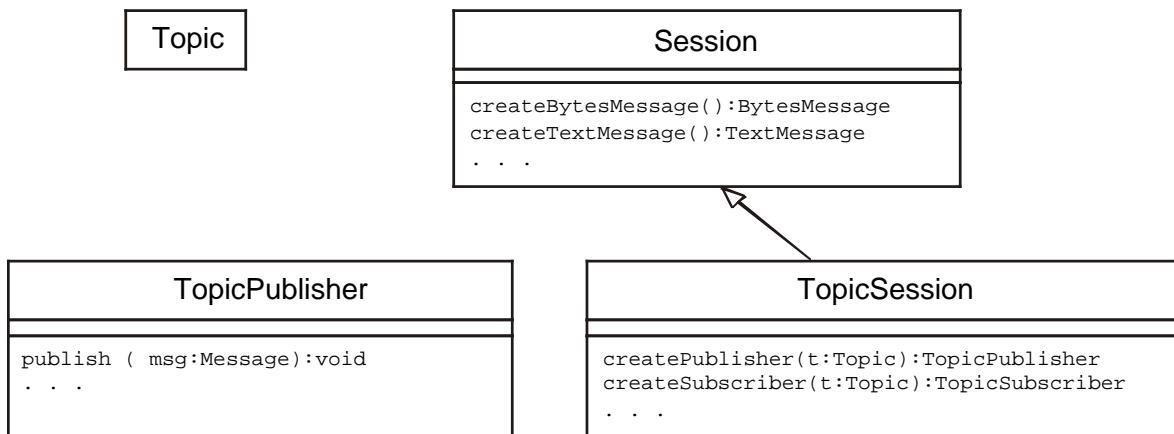
Following standard Java convention, the JMS API requires the use of factories to create the objects.

To bootstrap this whole process, you first typically look up the appropriate factory using JNDI -- you can then use the factory to create a connection.

From the connection, you can create the session.

Pub/Sub Session, Topic and Publisher

- Publish/subscribe messaging applications use **Topic** objects to represent categories of messages



22 - 9

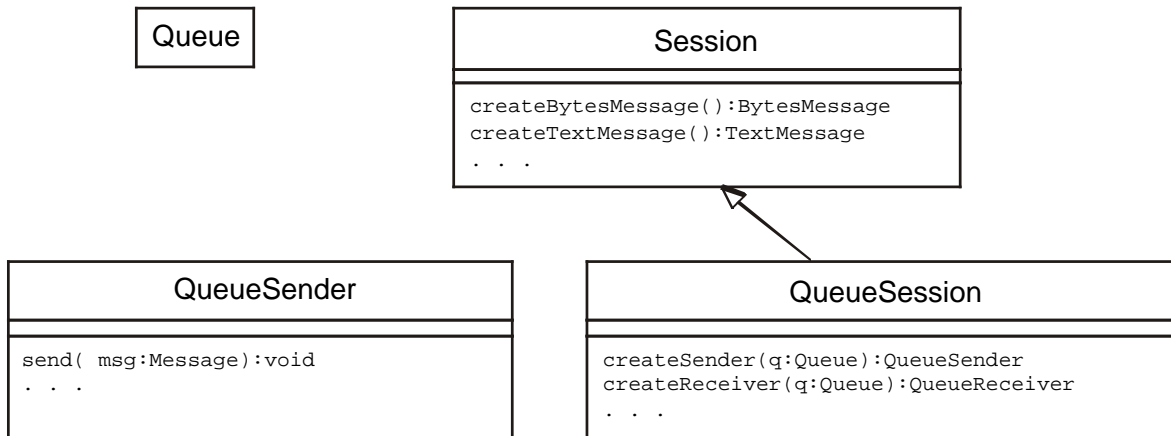
A Topic object represents a name in JNDI -- typically, the JMS (or J2EE) provider will register the name using an administrative facility (e.g a configuration file). Subscribers and publishers can then look up the Topic in JNDI.

As shown on the previous page, publishers and subscribers also need to establish a connection, from which they can create a session. The session then enables you to create a publisher or subscriber object depending on your role.

The session also lets you create a message which you can then publish. We show only the publisher API here, but the subscriber API is similar.

PTP Session, Topic and Publisher

- Point-to-point messaging applications use **Queue** objects to represent the target of messages



22 - 10

This figure shows the corresponding API for point-to-point messaging applications.

Like a Topic, a Queue object represents a name in JNDI -- typically, the JMS (or J2EE) provider will register the name using an administrative facility (e.g a configuration file). Senders and receivers can then look up the Queue in JNDI.

Sample Topic Client

```
1  public class MyClient
2      public static void main(String[] args)
3      {
4          try
5          {
6              Context ctx = getInitialContext();
7              TopicConnectionFactory factory =
8                  (TopicConnectionFactory)ctx.lookup (
9                  "helloFactory" );
10
11             TopicConnection connection =
12                 factory.createTopicConnection();
13             connection.start();
14
15             TopicSession session =
16                 connection.createTopicSession (
17                 false, Session.AUTO_ACKNOWLEDGE );
```

22 - 11

This listing starts a client program that will act as a pub/sub Topic publisher.

In inlines 6 to 9, we look up the factory in JNDI.

In lins 11 to 17, we create the connection and establish a session.

Sample Topic Client, cont'd

```
18  Topic topic = null;
19      try
20      {
21          topic = (Topic)ctx.lookup ( "hello" );
22      }
23      catch ( NameNotFoundException exc )
24      {
25          System.out.println ( "Topic not found" );
26          System.exit ( 1 );
27      }
28
29      TextMessage msg =
30          session.createTextMessage ( "Hi!!" );
31
32      TopicPublisher sender =
33          session.createPublisher ( topic );
34      sender.publish ( msg );
```

22 - 12

Continuing the listing, we then use JNDI to look up the topic and exit if we can't find it. We are assuming, as is normal, that the topic name was registered in JNDI prior to running this program.

If we find the topic, in lines 29 to 34, we then create a simple text message and a publisher object on which we send the message.

Sample Topic Client, cont'd

```
35     connection.stop();
36         connection.close();
37     }
38     catch ( Exception exc )
39     {
40         exc.printStackTrace();
41     }
42 }
43
44     static public Context getInitialContext()
45         throws Exception
46     {
47         return new InitialContext ();
48     }
49 }
```

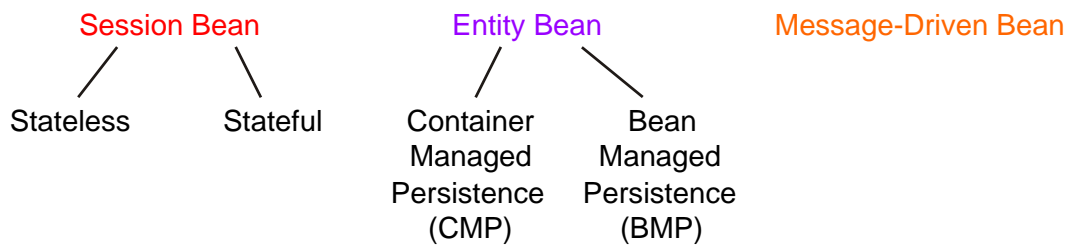
22 - 13

Finishing the pub/sub client, here we show the clean-up after sending the message.

Lines 44 to 48 show the familiar method that returns a JNDI context. And as before, we are assuming that there is a `jndi.properties` file in this program's directory that configures the JNDI provider.

Introduction to Message-Driven EJBs

- Message-driven enterprise beans provide an easy and powerful way to write JMS subscriber/receivers.
- Message-driven beans are a part of the EJB 2.0 specification



22 - 14

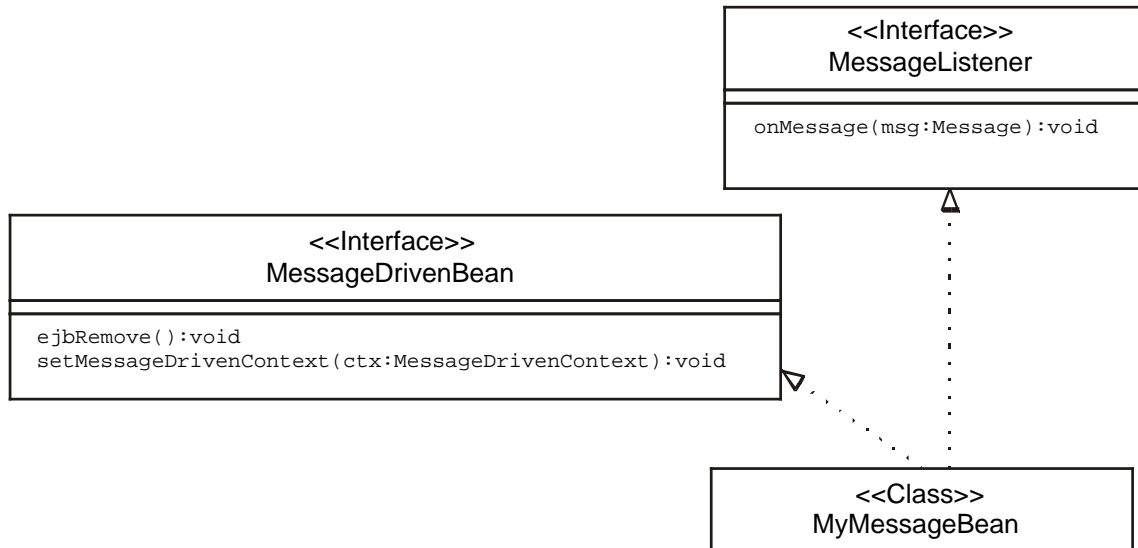
In some ways, message-driven beans are easier to code than any other type of enterprise bean, because you don't have to worry about homes or primary keys.

A container can create a pool of MDBs and assign them to process messages as they come in.

Clients do not directly access MDBs -- instead, they invoke them by sending JMS messages.

Message-Driven Bean Fundamentals

- Like all EJBs, message-driven beans must fit into a particular inheritance hierarchy



22 - 15

The most important method shown here is `onMessage()`, which the container calls when a message arrives for which the bean has been configured.

Configuring the bean for a particular JMS queue or topic is done in a container-specific way.

Sample Message-Driven EJB

```
1  package hello;
2
3  import javax.ejb.*;
4  import javax.jms.*;
5
6  public class HelloWorldBean
7      implements MessageDrivenBean, MessageListener
8  {
9      private MessageDrivenContext ctx;
10
11     public void setMessageDrivenContext(
12         MessageDrivenContext ctx)
13     {
14         this.ctx = ctx;
15     }
```

22 - 16

This page starts a code listing for a very simple message-driven enterprise bean.

In line 6, we define the bean class and specify the required interfaces.

Lines 11 to 14 show a trivial implementation for the required setMessageDrivenContext method.

Sample Message-Driven EJB, cont'd

```
16  public void ejbRemove() {}
17  public void ejbCreate() {}
18
19  public void onMessage ( Message message )
20  {
21      TextMessage msg = (TextMessage)message;
22      try
23      {
24          System.out.println ( "Received message: "
25                              + msg.getText() );
26      }
27      catch (JMSEException e) {}
28  }
29 }
```

22 - 17

Continuing the message-driven bean listing, lines 16 to 19 provide empty implementations of familiar EJB framework methods.

Line 19 starts the `onMessage` method, which the container invokes when a message arrives. This method accepts an argument of type `Message`, which is the base type for all of the supported JMS message types.

In this simple example, we assume that we were sent a text message, so we cast to that type in line 21. Then, in line 25, we retrieve the content of the text message, which we then send to the console.

Deploying an MDB

- Before you can run the client, you typically need to use your container's administrative facility to define the JNDI name
- For example, to run the sample pub/sub publisher and the sample message-driven EJB, you would need to create the JNDI name of **hello** to represent the Topic

22 - 18

On any J2EE platform, you need to perform administrative tasks to configure messaging applications. In the case of WebSphere, you need to use WAS admin tools to define the factory, queues/topics and for MDBs, you need to define a "listener port".

Chapter Summary

In this chapter, you learned:

- The fundamentals of messaging
- About the Java Message Service
- How to code Message-Driven enterprise beans