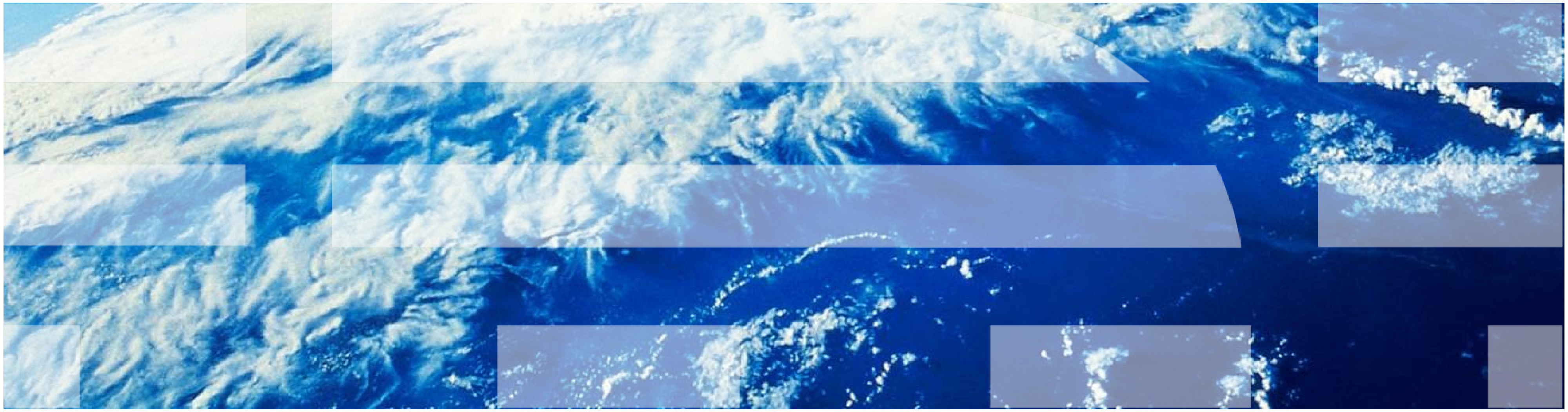
The Java Message Service API
Message Driven Beans (MDBs)



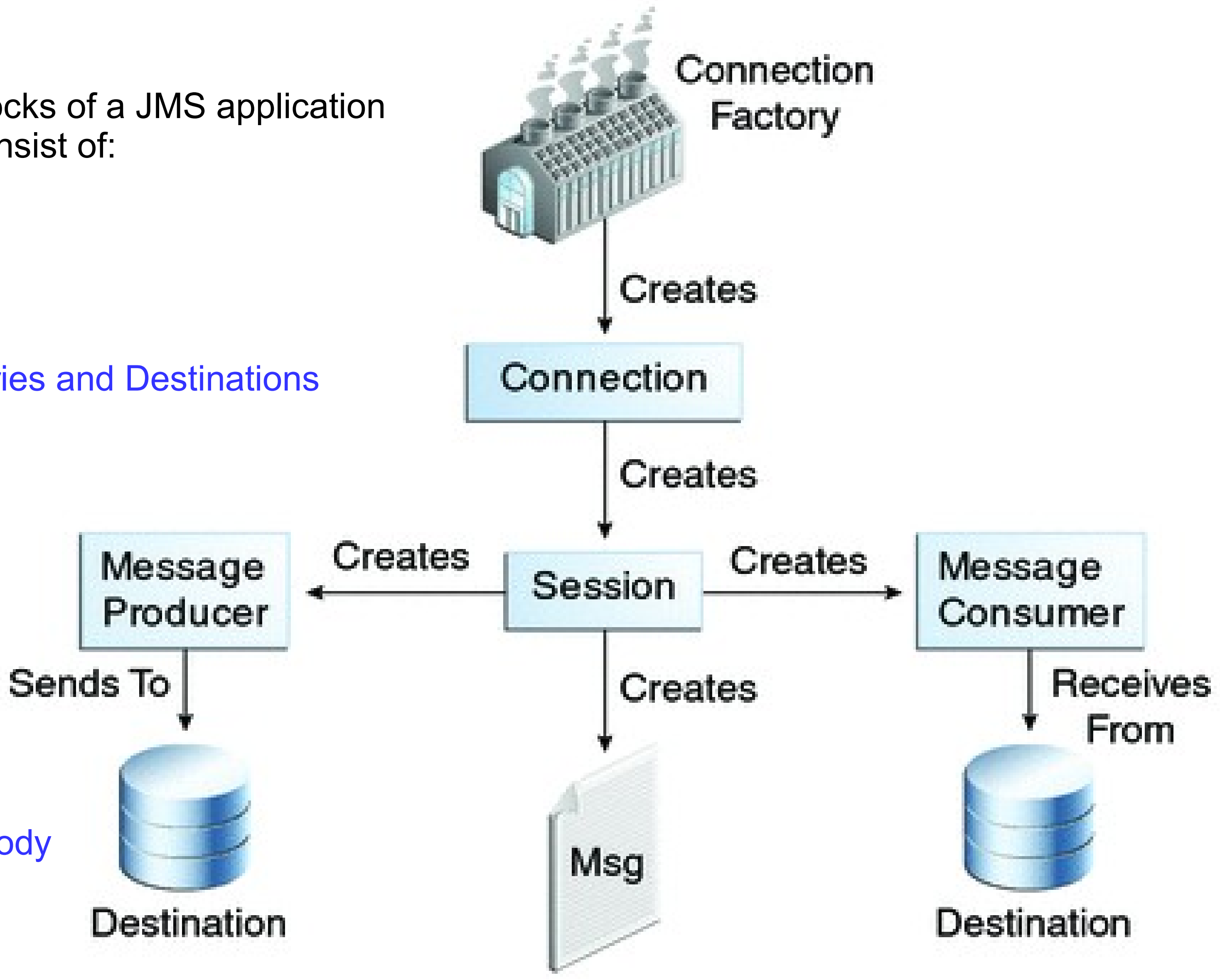
Agenda

- The JMS API Programming Model
- The Connection Factory & Connection
- JMS Destinations
- JMS Sessions
- The JMS Message:
- Producers
- Consumers
- Listeners
- Selectors
- Messages & Headers
- Bodies
-
- The Message Driven Bean (MDBs) API
- Structure of an MDB
- What Makes Message Driven Beans Different from Session Beans?
- Lifecycle of an MDB
- When to Use Message Driven Beans
- Writing an MDB
- JMS example (Producer)
- JMS example (Consumer)
- Pros and Cons of MDBs

The JMS API Programming Model

The basic building blocks of a JMS application consist of:

1) JMS Connection Factories and Destinations
2) Connections
3) Sessions
4) Message Producers
5) Message Consumers
6) Message Listeners
7) Message Selectors
8) Messages, Header & Body



The Connection Factory & Connection

The JMS Connections

A connection encapsulates a virtual connection with a JMS provider. A connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Connections implement the Connection interface. When you have a ConnectionFactory object, you can use it to create a Connection:

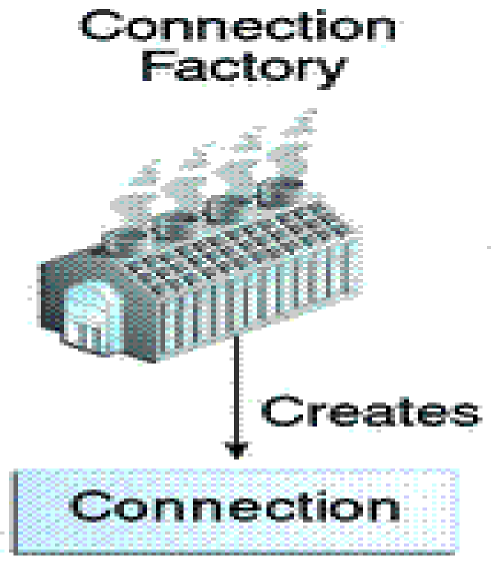
```
Connection connection = connectionFactory.createConnection();
```

Before an application completes, you must close any connections that you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

```
connection.close();
```

Before your application can consume messages, you must call the connection's start method; for details, see JMS Message Consumers. If you want to stop message delivery temporarily without closing the connection, you call the stop method.

The Connecion Factory



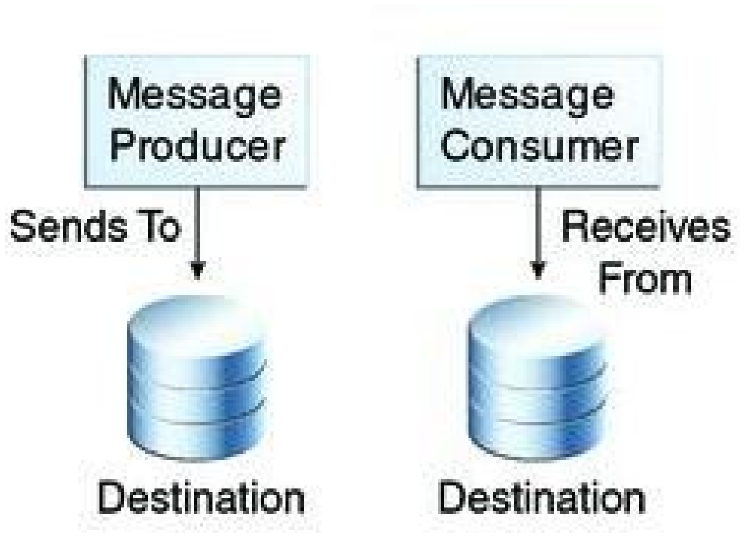
A connection factory is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator.

For example, the following code fragment specifies a resource whose JNDI name is jms/ConnectionFactory and assigns it to a ConnectionFactory object:

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

In a Java EE application, JMS administered objects are normally placed in the jms naming subcontext.

JMS Destinations



A destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes.

In the Point-To-Point (PTP) messaging domain:
Destinations are called queues.
In the pub/sub messaging domain:
Destinations are called topics.
A JMS application can use multiple queues or topics (or both).

To create a destination using a Application Server, you create a JMS destination resource that specifies a JNDI name for the destination. Each destination resource refers to a physical destination.

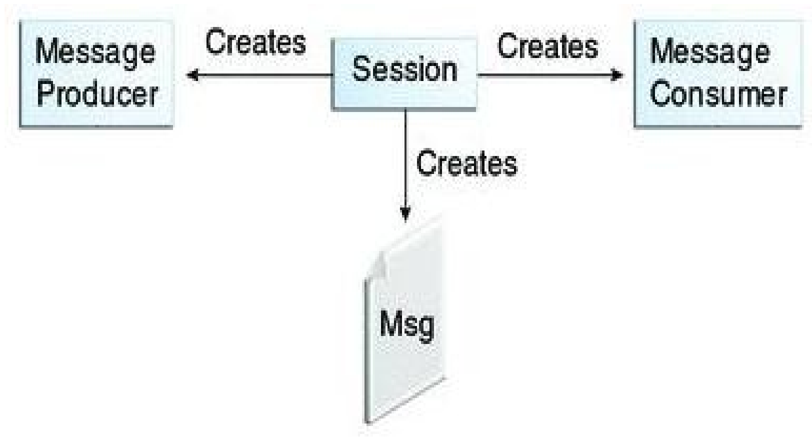
In addition to injecting a connection factory resource into a client program, you usually inject a destination resource. Unlike connection factories, destinations are specific to one domain or the other.
The following code specifies two resources, a queue and a topic.

The resource names are mapped to destination resources created in the JNDI namespace.

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

@Resource(lookup = "jms/Topic")
private static Topic topic;
```

JMS Sessions



A session is a single-threaded context for producing and consuming messages. You use sessions to create the following:

•Message producers
•Message consumers
•Messages
•Queue browsers
•Temporary queues and topics

Sessions serialize the execution of message listeners; for details, see JMS Message Listeners.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work. For details, see Using JMS API Local Transactions.

Sessions implement the Session interface. After you create a Connection object, you use it to create a Session:

```
Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
```

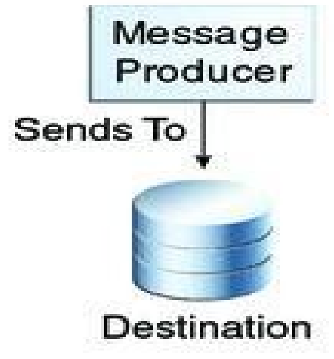
The first argument means that the session is not transacted; the second means that the session automatically acknowledges messages when they have been received successfully.

To create a transacted session, use the following code:

```
Session session = connection.createSession(true, 0);
```

Here, the first argument means that the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions. For more information on transactions, see Using JMS API Local Transactions.

The JMS Message Producers



A message producer is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

You use a Session to create a MessageProducer for a destination. The following examples show that you can create a producer for a Destination object, a Queue object, or a Topic object:

```
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

You can create an unidentified producer by specifying null as the argument to createProducer. With an unidentified producer, you do not specify a destination until you send a message.

After you have created a message producer, you can use it to send messages by using the send method:

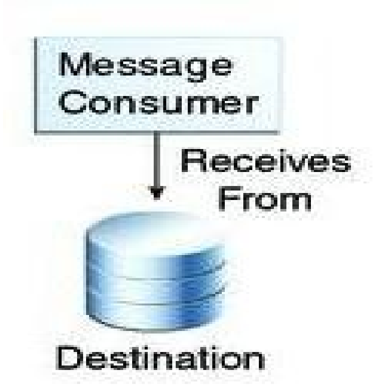
```
producer.send(message);
```

You must first create the messages; see JMS Messages.

If you created an unidentified producer, use an overloaded send method that specifies the destination as the first parameter. For example:

```
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(dest, message);
```

The JMS Message Consumers



A message consumer is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumer interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

For example, you could use a Session to create a MessageConsumer for a Destination object, a Queue object, or a Topic object:

```
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);
```

After you have created a message consumer, it becomes active, and you can use it to receive messages. You can use the close method for a MessageConsumer to make the message consumer inactive. Message delivery does not begin until you start the connection you created by calling its start method. (Remember always to call the start method; forgetting to start the connection is one of the most common JMS programming errors.)

You use the receive method to consume a message synchronously. You can use this method at any time after you call the start method:

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // time out after a second
```

The JMS Message Listeners

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the setMessageListener method. For example, if you define a class named Listener that implements the MessageListener interface, you can register the message listener as follows:

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

After you register the message listener, you call the start method on the Connection to begin message delivery. (If you call start before you register the message listener, you are likely to miss messages.)

When message delivery begins, the JMS provider automatically calls the message listener’s onMessage method whenever a message is delivered. The onMessage method takes one argument of type Message, which your implementation of the method can cast to any of the other message types.

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created.

In the Java EE platform, a message-driven bean is a special kind of message listener. For details, see Using Message-Driven Beans to Receive Messages Asynchronously.

The JMS Message Selectors

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than to the application. For an example of an application that uses a message selector, see An Application That Uses the JMS API with a Session Bean.

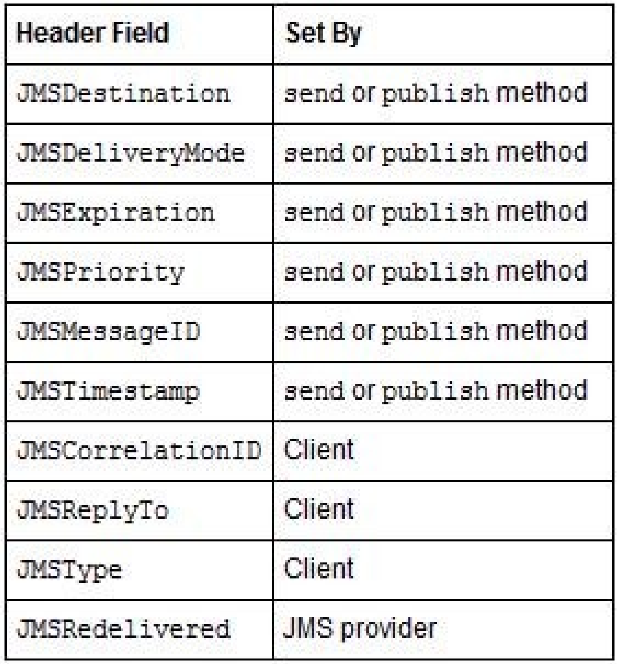
The message selector in the example selects any message that has a NewsType property that is set to the value 'Sports' or 'Opinion':
NewsType = ’Sports’ OR NewsType = ’Opinion’

The createConsumer and createDurableSubscriber methods allow you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector. A message selector cannot select messages on the basis of the content of the message body.

The JMS Messages & Header

Table 47-1



Header Field	Set By
JMSDestination	send or publish method
JMSDeliveryMode	send or publish method
JMSExpiration	send or publish method
JMSPriority	send or publish method
JMSMessageID	send or publish method
JMSTimestamp	send or publish method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

JMS Messages

The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message has three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the Message interface in the API documentation.

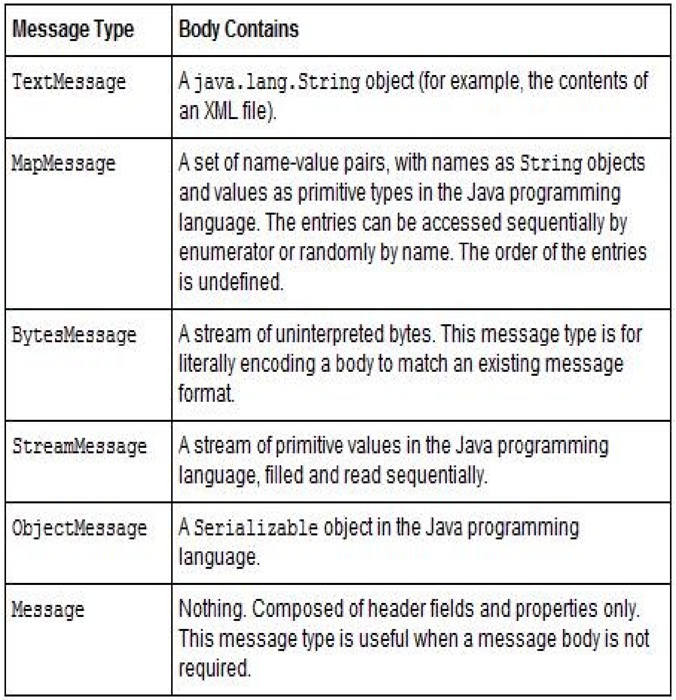
JMS Message Headers

A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages. Table 47-1 lists the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field JMSMessageID. The value of another header field, JMSDestination, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the Message interface. Some header fields are intended to be set by a client, but many are set automatically by the send or the publish method, which overrides any client-set values.

The JMS Message Bodies

Table 47-2



Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an XML file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

The JMS API defines five message body formats, also called message types, which allow you to send and to receive data in many different forms and provide compatibility with existing messaging formats.

Table 47-2 describes these message types.

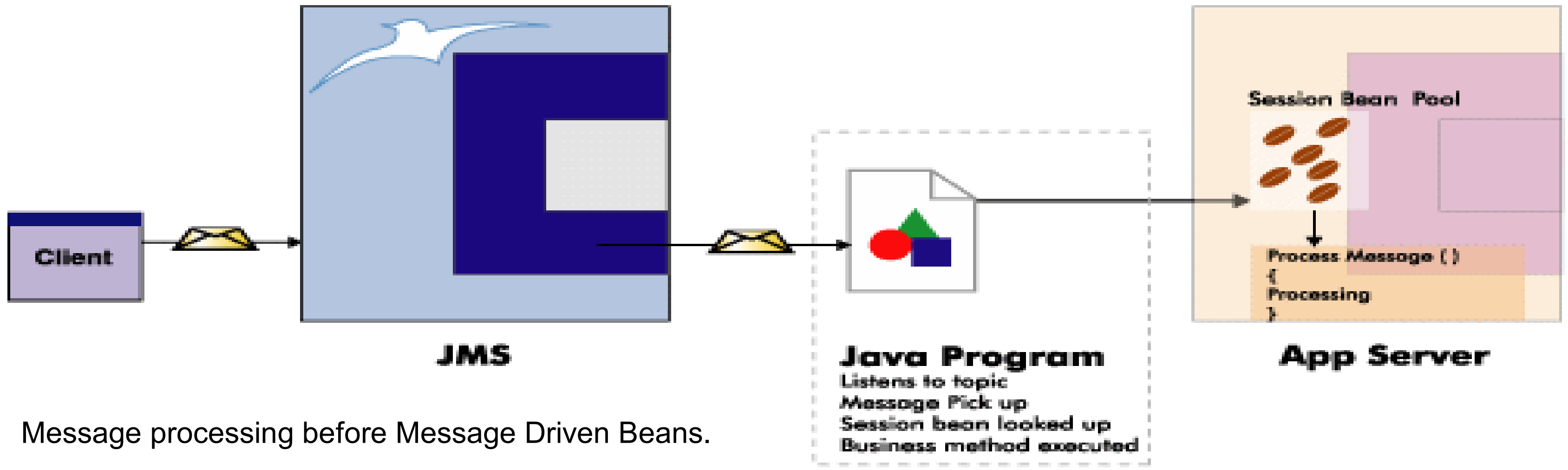
The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a TextMessage, you might use the following statements:

```
TextMessage message =
    session.createTextMessage();
message.setText(msg_text);     // msg_text is a
    String
producer.send(message);
```

At the consuming end, a message arrives as a generic Message object and must be cast to the appropriate message type. You can use one or more getter methods to extract the message contents. The following code fragment uses the getText method:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    TextMessage message = (TextMessage) m;
    System.out.println("Reading message: " + message.getText());
} else {
    // Handle error
}
```

The Message-Driven Bean (MDBs) API

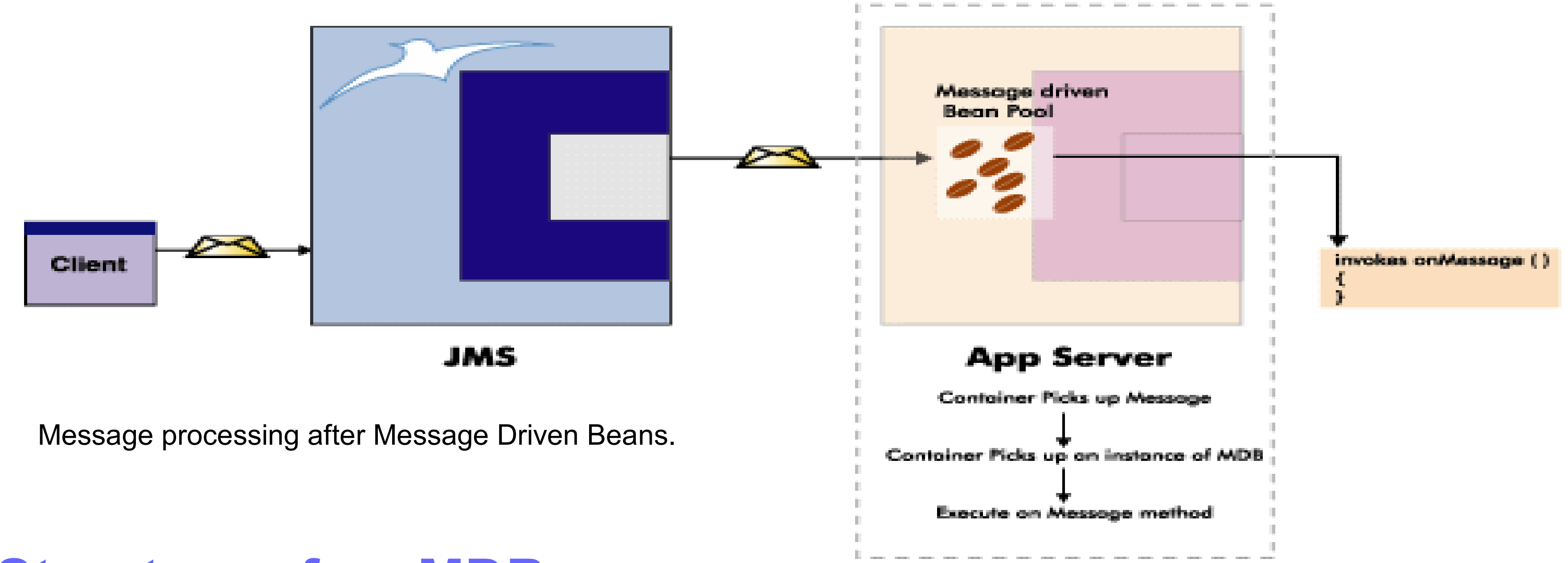


Message processing before Message Driven Beans.

What is a Message Driven Bean?

- A message driven bean is a stateless, server-side, transaction-aware component that is driven by a Java message (javax.jms.message). It is invoked by the EJB Container when a message is received from a JMS Queue or Topic. It acts as a simple message listener.
- A Java client, an enterprise bean, a Java ServerPagesTM (JSP) component, or a non-J2EE application may send the message. The client sending the message to the destination need not be aware of the MDBs deployed in the EJB Container. However, the message must conform to JMS specifications.
- Before MDBs were introduced, JMS described a classical approach to implement asynchronous method invocation. The approach used an external Java program that acted as the listener, and on receiving a message, invoked a session bean method. However, in this approach the message was received outside the application server and was thus not part of a transaction in the EJB Server. MDB solves this problem.

The Message-Driven Bean (MDBs) API


Client
JMS
Message driven Bean Pool
invokes onMessage ( )
{
}
App Server
Container Picks up Message
Container Picks up an instance of MDB
Execute on Message method

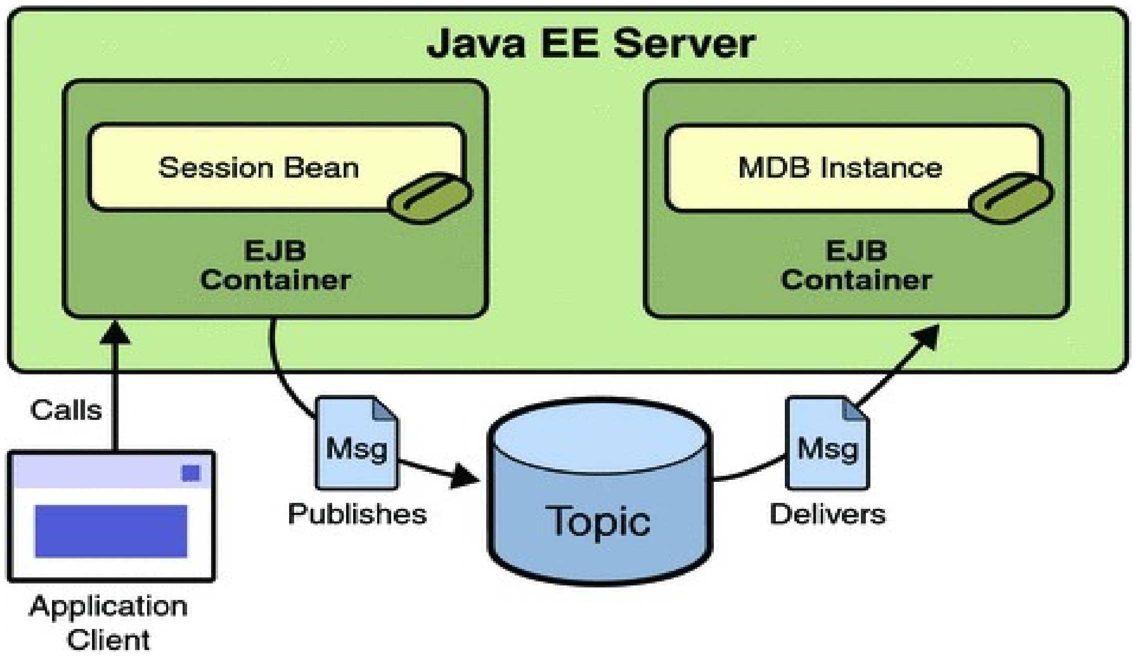
Message processing after Message Driven Beans.

Structure of an MDB

- It has no home or remote interfaces, and is only a bean class.
- It resembles a stateless session bean - that is, it has short-lived instances and does not retain state for a client.
- A client interacts with the MDB in the same way it interacts with a JMS application or JMS server.
- Through the MDB, the EJB Container sets itself up as a listener for asynchronous invocation and directly invokes the bean (no interfaces), which then behaves like an enterprise bean.
- All instances of a particular MDB type are equivalent as they are not directly visible to the client and maintain no conversational state. This means that the Container can pool instances to enhance scalability.

The Message-Driven Bean (MDBs) API

What Makes Message-Driven Beans Different from Session Beans?



The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. Unlike a session bean, a message-driven bean has only a bean class.

A message-driven bean's instances retain no data or conversational state for a specific client.

All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.

- Session beans allow you to send JMS messages and to receive them synchronously but not asynchronously.

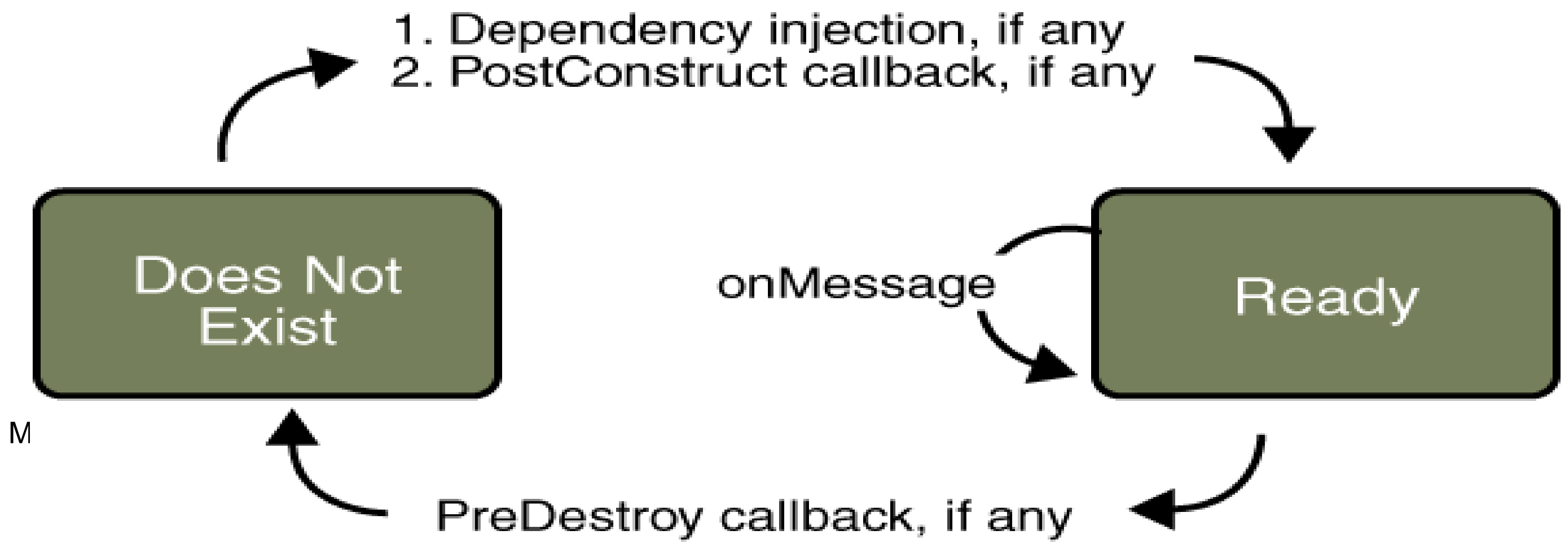
- To avoid tying up server resources, do not to use blocking synchronous receives in a server-side component; in general, JMS messages should not be sent or received synchronously.

- To receive messages asynchronously, use a message-driven bean.

Message-driven beans have the following characteristics

They do not represent directly shared data in the database, but they can access and update this data.
They execute upon receipt of a single client message.
They are invoked asynchronously.
They are relatively short-lived.
They can be transaction-aware.
They are stateless.

The Message-Driven Bean (MDBs) API

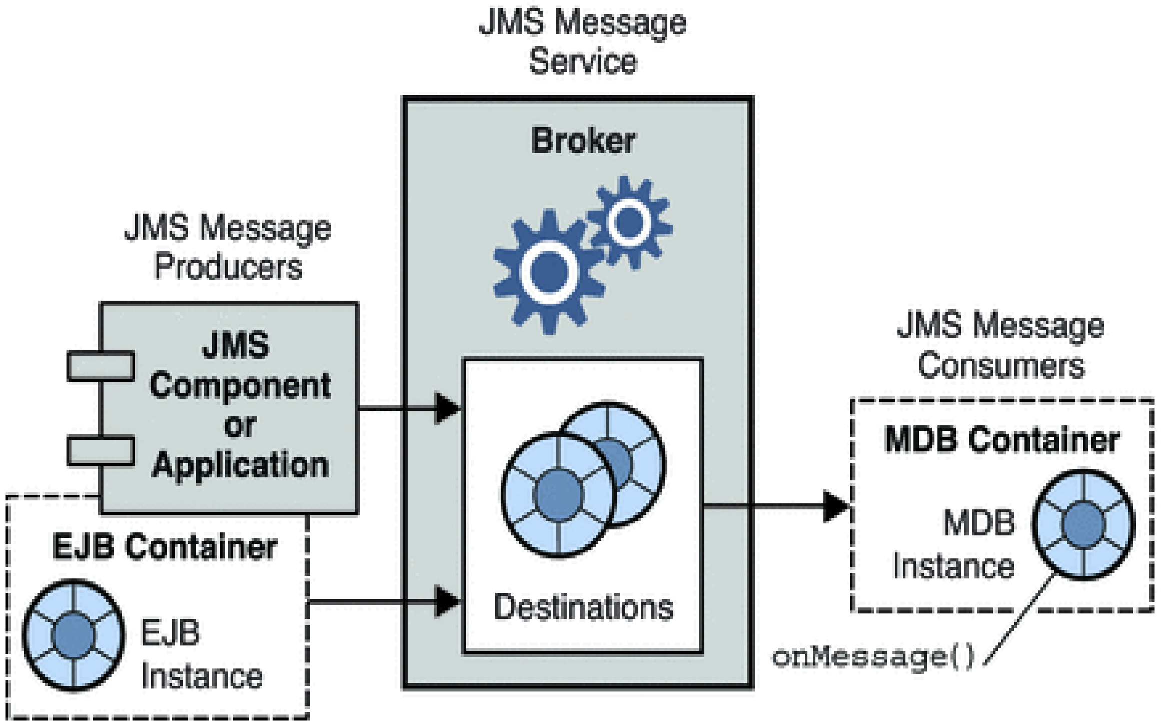

1. Dependency injection, if any
2. PostConstruct callback, if any
Does Not Exist
onMessage
Ready
M
PreDestroy callback, if any

Lifecycle of an MDB

- The EJB Container performs several tasks at the beginning of the life cycle of the MDB:
- Creates a message consumer (a QueueReceiver or TopicSubscriber) to receive the messages
- Associates the bean with a destination and connection factory at deployment
- Registers the message listener and the message acknowledgement mode
- The lifecycle of an MDB depends on the lifespan of the EJB Server in which it is deployed. As MDBs are stateless, bean instances are typically pooled by the EJB Server and retrieved by the Container when a message becomes available on the topic or queue.

The Message-Driven Bean (MDBs) API

When to Use Message-Driven Beans



Writing an MDB

Writing an MDB involves the following tasks:

- Implement the javax.ejb.MessageDrivenBean and javax.jms.MessageListener interfaces in the MDB class.
- Provide an implementation of the business logic inside the onMessage().
- Provide a setMessageDrivenContext() method that associates the bean with its environment.
- Provide an ejbCreate() method that returns void and takes no arguments. This method may be blank.
- Provide an ejbRemove() method implementation. This method may be blank, unless certain resources need to be acquired before the bean goes out of scope.

- Session beans allow you to send JMS messages and to receive them synchronously but not asynchronously.
- To avoid tying up server resources, do not to use blocking synchronous receives in a server-side component; in general, JMS messages should not be sent or received synchronously.
- To receive messages asynchronously, use a message-driven bean.

JMS example (Producer)

```
public class JMSProducerServlet extends HttpServlet {

        @Resource(name = "jms/messageQueueCF")
        private QueueConnectionFactory qcf;

        @Resource(name = "jms/messageQueue")
        private Queue queue;

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        try {
            QueueConnection connection = qcf.createQueueConnection();
            QueueSession session = connection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer = session.createProducer(queue);
            TextMessage message = session.createTextMessage();
            message.setText("#### JMS Example Running #####");
            producer.send(message);
            session.close();
            connection.close();
        } catch (JMSException e) {
            e.printStackTrace();
        }
        }
}
```

JMS example (Consumer)

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


public void onMessage(Message message) {
    if (message instanceof TextMessage) {
        TextMessage text = (TextMessage) message;
        try {
            System.out.println();
            System.out.println("Invoking MDB onMessage() now.");
                    System.out.println("Message Object is: " + message);
                    System.out.println("Text message is: " + text.getText());
                    System.out.println();

        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

Pros and Cons of MDBs

Pros:

EJB Containner provides all the plumbing necessary for Mdbs to process messages concurrently. The result is multithreading environment that provides substantial improvement in performance.

Cons:

MDBs are stateless, so if you need to preserve state between method invocations, you must use statefull Session Seans.

You should not use MDBs when you need:

- Real time response
- The order of the requests are important
- You need a synchronous request/response