



MDB



Plan MDB

- Introduction
- JMS
- Exemple de JMS
- Exemple de JMS consommé par MDB

Message-Driven Beans

- Nouveauté apparue avec EJB 2.0,
- *Messaging* = moyen de communication léger.
- Pratique dans de nombreux cas, vérification de CB en envoyant un message en cas de prob.
- Message-Driven beans = beans accessibles par *messaging* asynchrone.
- Couplage faible : autorise les messages entre des systèmes hétérogènes.

MDB: motivation

■ Performance

- Un client RMI-IIOP *attend* pendant que le serveur effectue le traitement d'une requête,

■ Fiabilité

- Lorsqu'un client RMI-IIOP parle avec un serveur, ce dernier doit être en train de fonctionner. S'il crashe, ou si le réseau crashe, le client est coincé.

■ Pas de broadcasting !

- RMI-IIOP limite les liaisons 1 client vers 1 serveur

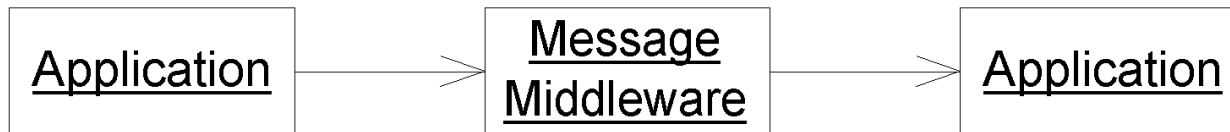
Messaging MOM

- C'est comme le mail ! Ou comme si on avait une troisième personne entre le client et le serveur !

Remote method invocations:



Messaging:



Messaging

- A cause de ce "troisième homme" les performances ne sont pas toujours au rendez-vous !
- Message Oriented Middleware (MOM) est le nom donné aux middlewares qui supportent le *messaging*.
 - Tibco Rendezvous, IBM MQSeries, BEA Tuxedo/Q, Microsoft MSMQ, Talarian SmartSockets, Progress SonicMQ, Fiorano FioranoMQ, ...
 - Ces produits fournissent : messages avec garantie de livraison, tolérance aux fautes, load-balancing des destinations, etc...



Appels asynchrones avec EJB

- Depuis Java EE 6 les beans sessions peuvent avoir des méthodes que l'on peut appeler de façon asynchrones
- Des cas où il fallait utiliser des MDB peuvent maintenant être codés plus simplement avec ces méthodes asynchrones

Exemple

```
@Asynchronous
public Future<String> payer(facture facture)
    throws PaiementException {
    ...
    if (SessionContext.wasCancelled()) {
        // la requête a été annulée
        ...
    } else {
        // Effectuer le paiement
        ...
    }
    ...
}
```

retourne le statut
du paiement sous
la forme d'une
String

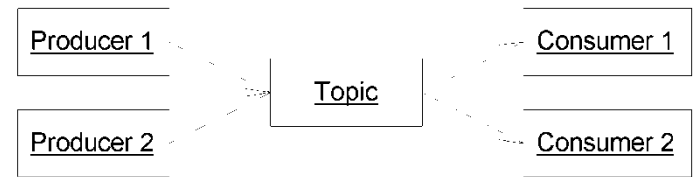
Java Message Service (JMS)

- Les serveurs MOM sont pour la plupart propriétaires : pas de portabilité des applications !
- JMS = un standard pour normaliser les échanges entre composant et serveur MOM,
 - Une API pour le développeur,
 - Un Service Provider Interface (SPI), pour connecter l'API et les serveurs MOM, via les drivers JMS

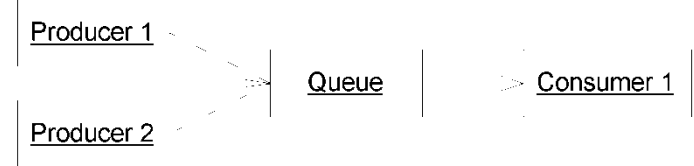
JMS : Messaging Domains

- Avant de faire du messaging, il faut choisir un *domaine*
 - Domaine = type de messaging
- Domaines possibles
 - Publish/Subscribe (pub/sub) : n producteurs, n consommateurs (tv)
 - Point To Point (PTP) : n producteurs, 1 consommateur

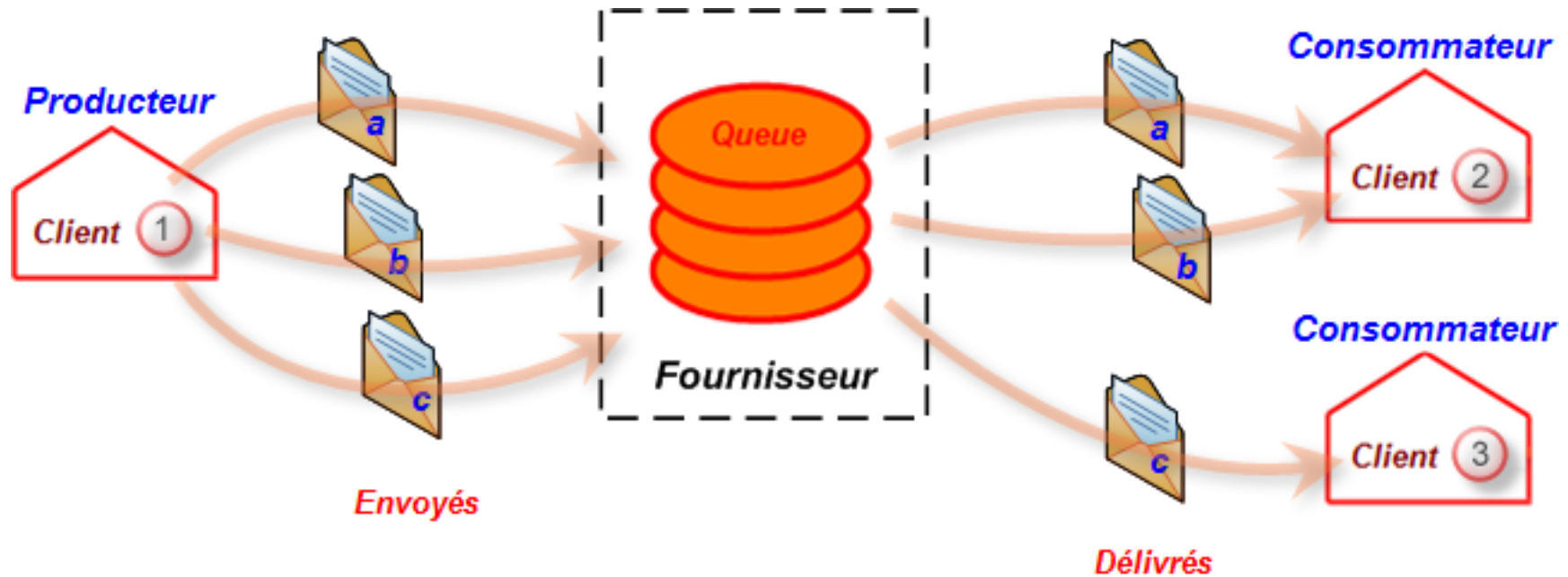
Publish/subscribe:



Point-to-point:

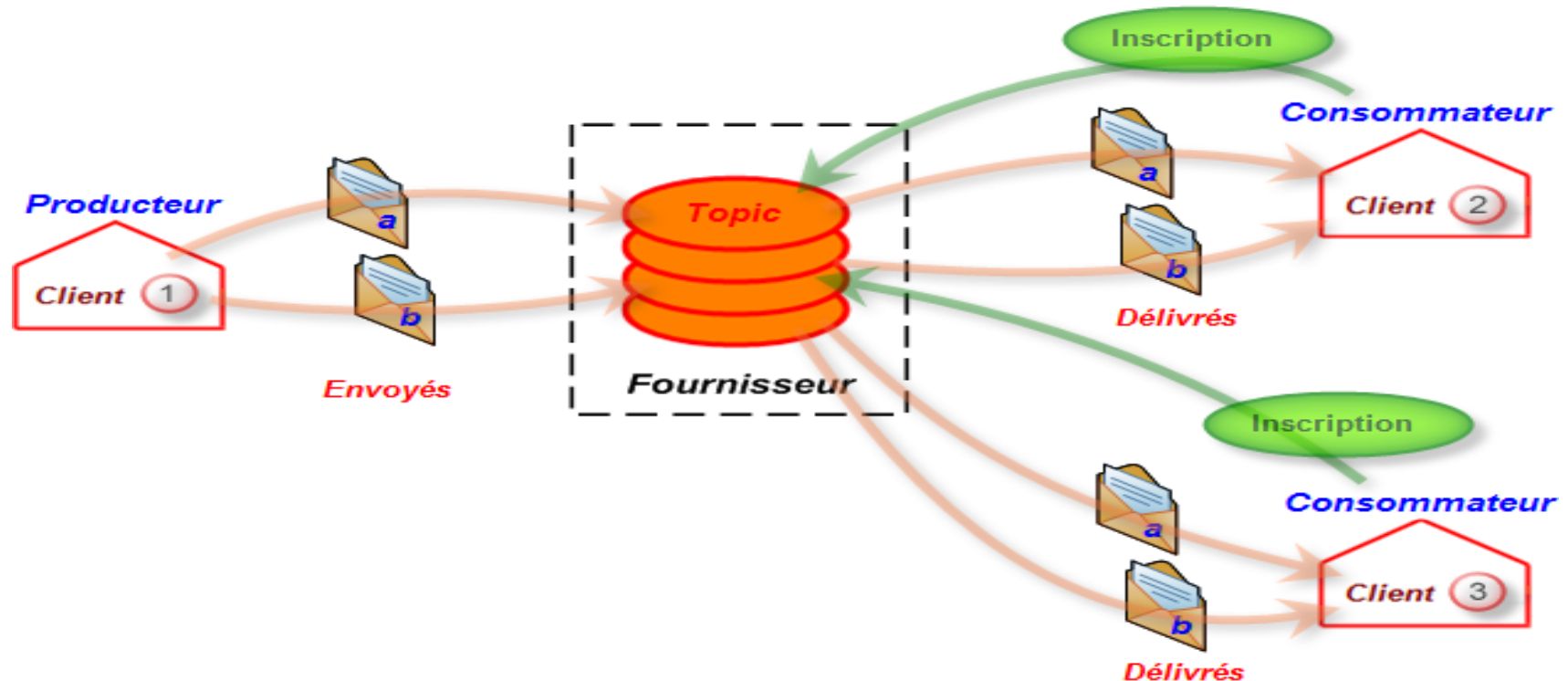


JMS : Queue (point à point)



- *Utilise les files d'attente (`javax.jms.Queue`) pour communiquer.*
- *Tant qu'un message n'est pas consommé, ou qu'il n'a pas expiré, il reste stocké au sein du fournisseur.*
- *Dès que le client devient actif, il peut alors consulter le message qui lui était destiné*

JMS:Topic (*publication/abonnement*)

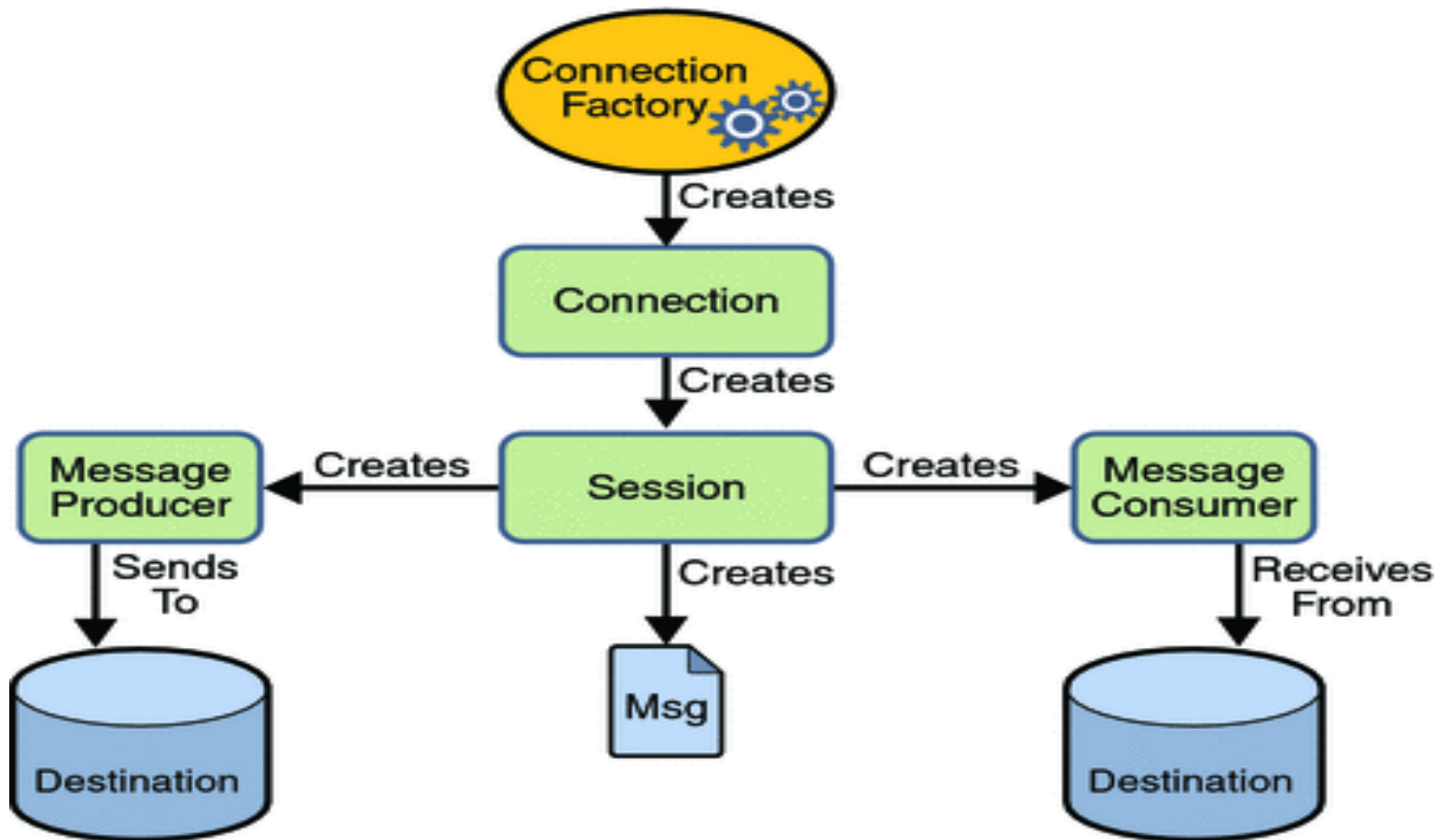


- *un producteur peut envoyer un message à plusieurs consommateurs par le biais d'un sujet (topic)*
- *Chaque consommateur doit cependant être préalablement inscrit à ce sujet*
- *Le message ne disparaît du Topic que lorsque tous les abonnés l'ont lu et acquitté.*

JMS : les étapes

1. *Localiser le driver JMS*
 - lookup JNDI. Le driver est une *connection factory*
2. *Créer une connection JMS*
 - obtenir une *connection* à partir de la *connection factory*
3. *Créer une session JMS*
 - Il s'agit d'un objet qui va servir à recevoir et envoyer des messages. On l'obtient à partir de la *connection*.
4. *Localiser la destination JMS*
 - Il s'agit du canal, de la chaîne télé ! Normalement, c'est réglé par le déployeur. On obtient la *destination* via JNDI.
5. *Créer un producteur ou un consommateur JMS*
 - Utilisés pour écrire ou lire un message. On les obtient à partir de la *destination* ou de la *session*.
6. Envoyer ou recevoir un message

JMS : les étapes



JMS : Etape 1 (1)

Apps airtel Search Certificate Servlet and JSP Tech... Java Regular Express... Google PageRank &... Accessibility validati... Usability report for h...

Home About... Logout Help

User: admin | Domain: domain1 | Server: localhost

GlassFish™ Server Open Source Edition

Common Tasks

- Domain
 - server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Resources
 - JDBC
 - Connectors
 - Resource Adapter Configs
 - JMS Resources
 - Connection Factories
 - Destination Resources
 - JavaMail Sessions
 - JNDI
- Configurations
 - default-config
 - server-config

New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

Pool Name: * myQueueConnectionFactory

Resource Type: * javax.jms.QueueConnectionFactory

Description:

Status: ☒ Enabled

Pool Settings

Initial and Minimum Pool Size: 8 Connections
Minimum and initial number of connections maintained in the pool

Maximum Pool Size: 32 Connections
Maximum number of connections that can be created to satisfy client requests

Pool Resize Quantity: 2 Connections
Number of connections to be removed when pool idle timeout expires


Idle Timeout: 300 Seconds

JMS : Etape 1 (2)

[Home](#) [About...](#) [Logout](#) [Help](#)

User: admin | Domain: domain1 | Server: localhost

GlassFish™ Server Open Source Edition

 Common Tasks

- Domain
 - server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Resources
 - JDBC
 - Connectors
 - Resource Adapter Configs
 - JMS Resources
 - Connection Factories
 - Destination Resources
 - JavaMail Sessions
- JNDI
- Configurations
 - default-config
 - server-config
- Update Tool

New JMS Destination Resource

[OK](#) [Cancel](#)

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

JNDI Name: *
A unique name of up to 255 characters; must contain only alphanumeric, underscore, dash, or dot characters

Physical Destination Name *
Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: *

Description:

Status: ☒ Enabled

Additional Properties (0)

[Add Property](#) [Delete Properties](#)

Name	Value	Description:
No items found.		

[OK](#) [Cancel](#)

JMS : Sender (1)

```
//Create and start connection
InitialContext ctx=new InitialContext();
QueueConnectionFactory f=
(QueueConnectionFactory)ctx.lookup("myQueueConnectionFactory");
QueueConnection con=f.createQueueConnection();
con.start();
//2) create queue session
QueueSession ses=con.createQueueSession(false ,
Session.AUTO_ACKNOWLEDGE);
//3) get the Queue object
Queue t=(Queue)ctx.lookup("myQueue");
//4)create QueueSender object
QueueSender sender=ses.createSender(t);
//5) create TextMessage object
TextMessage msg=ses.createTextMessage();
```

JMS : Sender (2)

```
//6) write message
BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
while (true ) {
    System.out.println("Enter Msg, end to terminate:");
    String s=b.readLine();
    if (s.equals("end")) break ;
    msg.setText(s);
//7) send message
    sender.send(msg);
    System.out.println("Message successfully sent."); }
//8) connection close
    con.close();
```

Note : Dans 3) false = pas de transactions, AUTO_ACKNOWLEDGE = inutile ici puisqu'on envoie des messages.

JMS : Receiver(1)

//1) Create and start connection

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

```
QueueConnectionFactory f=
```

```
(QueueConnectionFactory)ctx.lookup("myQueueConnectionFactory");
```

```
QueueConnection con=f.createQueueConnection();
```

```
con.start();
```

//2) create Queue session

```
QueueSession ses=con.createQueueSession(false ,
```

```
Session.AUTO_ACKNOWLEDGE);
```

//3) get the Queue object

```
Queue t=(Queue)ctx.lookup("myQueue");
```

//4)create QueueReceiver

```
QueueReceiver receiver=ses.createReceiver(t);
```

//5) create listener object

```
MyListener listener=new MyListener();
```

//6) register the listener object with receiver

```
receiver.setMessageListener(listener);
```

```
System.out.println("Receiver1 is ready, waiting for messages...");
```

```
System.out.println("press Ctrl+c to shutdown...");
```

```
while (true ) Thread.sleep(1000);
```

JMS : Listener

```
import javax.jms.*;
public class MyListener implements MessageListener {

    public void onMessage(Message m) {
        try{
            TextMessage msg=(TextMessage)m;

            System.out.println("following message is received:"+msg.getText());
        } catch(JMSEException e){System.out.println(e);}
    }
}
```



Jar

Archives à installer

appserv-rt.jar

javaee.jar

appserv-deployment-client.jar

appserv-ext.jar

Archives supplémentaires pour JMS

imqjmsra.jar

appserv-admin.jar

appserv-ws.jar

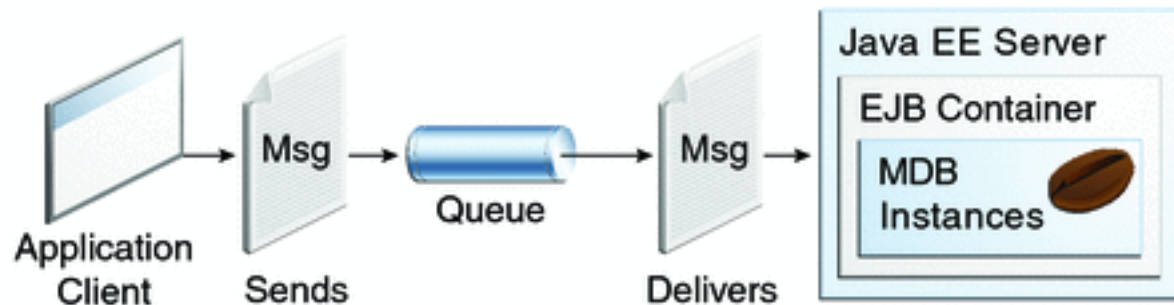


Intégrer JMS et les EJB

- Pourquoi créer un nouveau type d'EJB ?
- Pourquoi ne pas avoir délégué le travail à un objet spécialisé ?
- Pourquoi ne pas avoir augmenté les caractéristiques des session beans ?
- Parce que ainsi on peut bénéficier de tous les avantages déjà rencontrés : cycle de vie, *pooling*, descripteurs spécialisés, code simple...

Qu'est-ce qu'un Message-Driven Bean ?

- Un EJB qui peut recevoir des messages
 - Il consomme des messages depuis les queues ou topics, envoyés par les clients JMS



Qu'est-ce qu'un Message-Driven Bean ?

- Un client n'accède pas à un MDB via une interface, il utilise l'API JMS,
- Un MDB n'a pas d'interface,
- Les MDB possèdent une seule méthode, faiblement typée : **onMessage ()**
 - Elle accepte un message JMS (BytesMessage, ObjectMessage, TextMessage, StreamMessage ou MapMessage)
 - Pas de vérification de types à la compilation.
 - Utiliser `instanceof` au run-time pour connaître le type du message.
- Les MDB n'ont pas de valeur de retour
 - Ils sont découplés des *producteurs* de messages.

Qu'est-ce qu'un Message-Driven Bean ?

- Pour envoyer une réponse à l'expéditeur : plusieurs *design patterns*...
- Les MDB ne renvoient pas d'exceptions au client (mais au container),
- Les MDB sont stateless...
- Les MDB peuvent être des abonnés durables ou non-durables (*durable or nondurable subscribers*) à un *topic*
 - *Durable* = reçoit tous les messages, même si l'abonné est inactif,
 - *Dans ce cas, le message est rendu persistant et sera délivré lorsque l'abonné sera de nouveau actif.*
 - *Nondurable* = messages perdus lorsque abonné inactif.

Qu'est-ce qu'un Message-Driven Bean ?

- Le consommateur (celui qui peut les détruire) des messages est en général le Container
 - C'est lui qui choisit d'être durable ou non-durable,
 - S'il est durable, les messages résistent au crash du serveur d'application.

Exemple : Message-Driven Bean

```
@MessageDriven(mappedName="myQueue")
public class MyListener implements MessageListener{
    @Override
    public void onMessage(Message msg) {
        TextMessage m=(TextMessage)msg;
        try{
            System.out.println("message received: "+m.getText());
        }catch (Exception e){System.out.println(e);}
    }
}
```