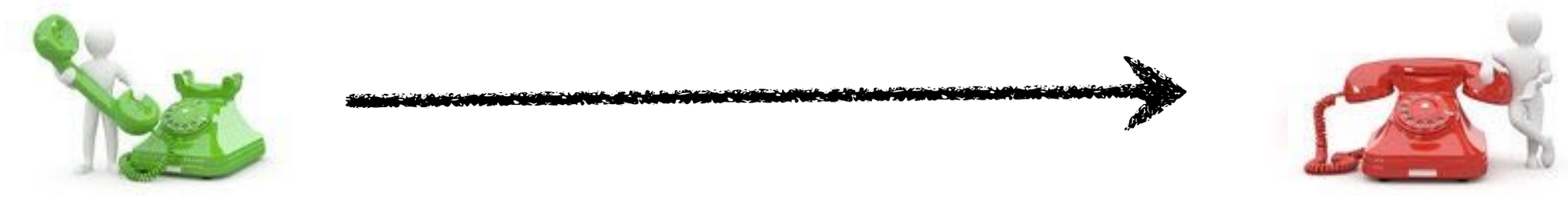
Message Driven Beans

Appel méthodes



Messaging

- moyen de communication léger
  - faible couplage
  - communication asynchrone
  - support émetteurs/récepteurs multiples

Message-Driven Beans =
beans accessibles par messaging asynchrone

Messaging vs RMI

Performance

- un client synchrone (RMI ou appel méthode) doit attendre la réponse du serveur

Fiabilité

- le client est couplé au serveur : pannes potentielles liées au serveur ou au réseau

Pas de broadcasting

- un seul client parle à un seul serveur

Message Oriented Middleware (MOM)

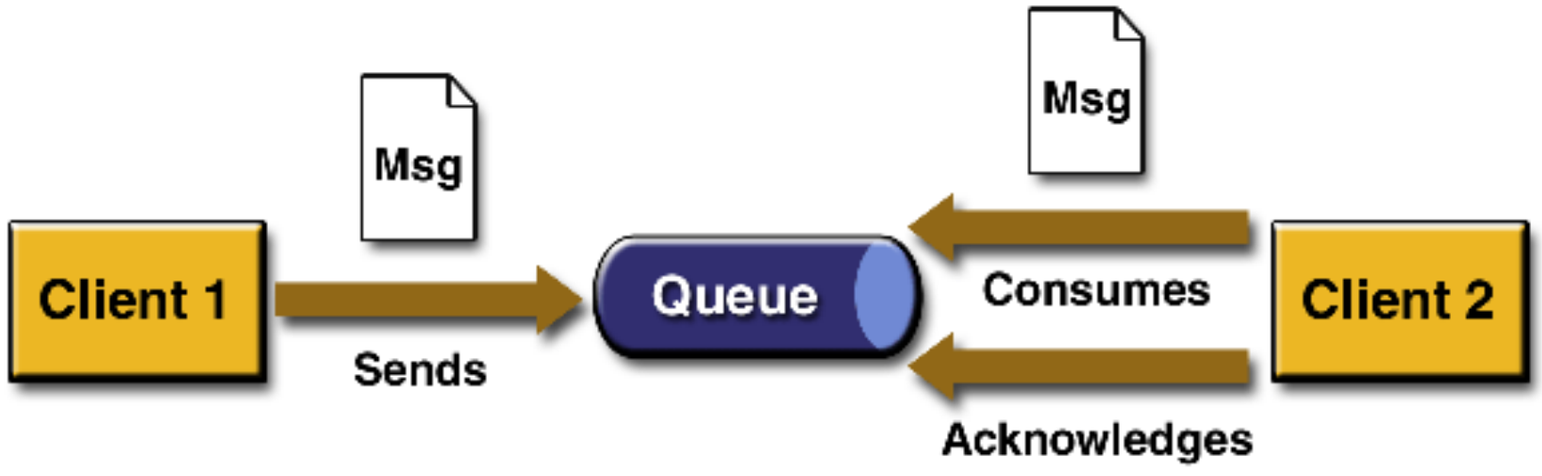
- **middlewares qui supportent le messaging**
- IBM WebSphere MQSeries, Tibco Rendezvous, SonicMQ, ActiveMQ, Oracle Advanced Queuing, ...
- **messages avec garantie de livraison, tolérance aux fautes, load-balancing des destinations, etc…**

JMS

- spécification d'un service de messagerie en Java (standard pour normaliser les échanges entre composant et serveur MOM)
  - JMS Provider : mise en oeuvre du système
  - client JMS : application consommatrice ou productrice de messages
  - message JMS : objet véhiculant les données entre applications

Messaging models

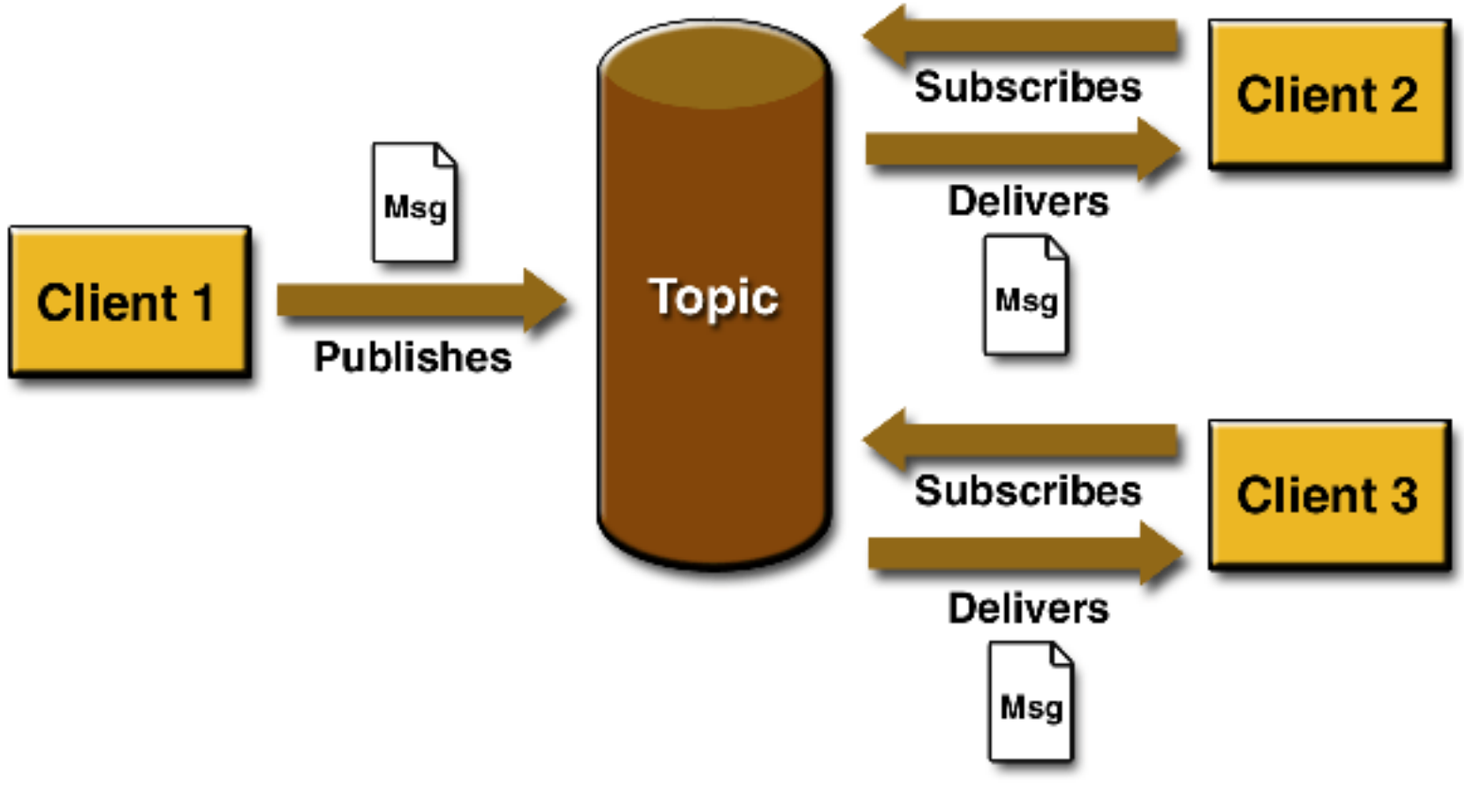
- **Point-to-point**
  - un message circule d'un *producer* à un seul *consumer* (choisi au hasard)
  - l'ordre dans la **queue** n'est pas garanti



oracle.com

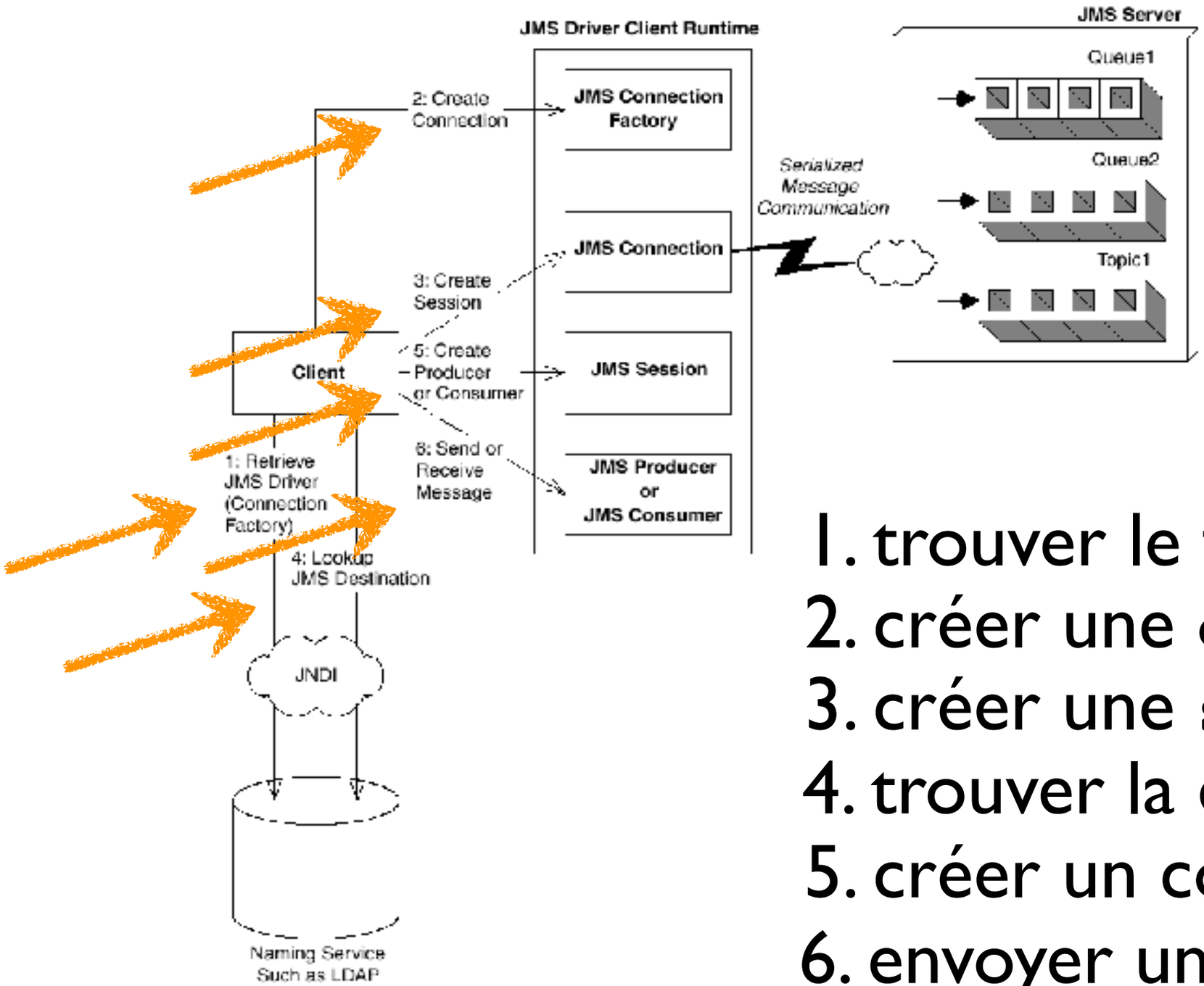
Messaging models

- **Publish-subscribe**
  - messages postés dans un **topic**
  - un message reçu potentiellement par plusieurs *subscribers*



oracle.com

JMS : les étapes

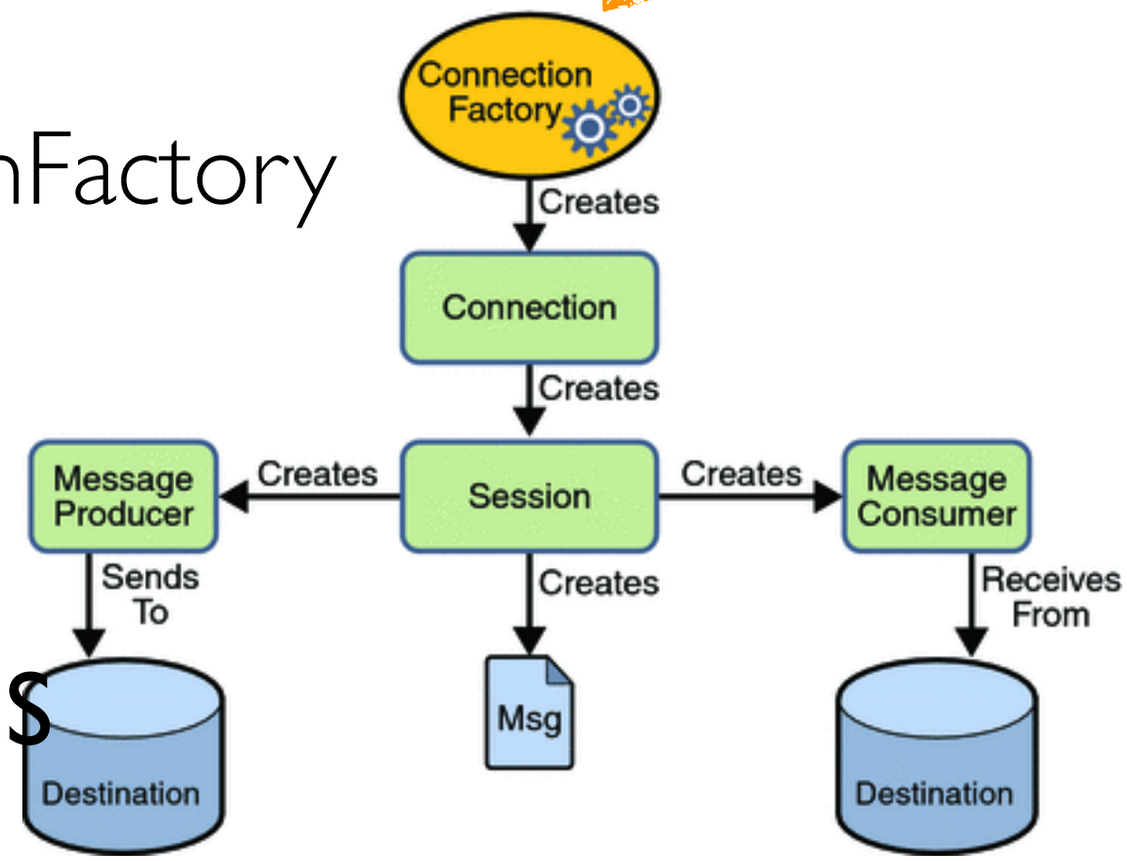


1. trouver le fournisseur
2. créer une `connection`
3. créer une `session`
4. trouver la destination
5. créer un consommateur
6. envoyer un message

JMS : les étapes

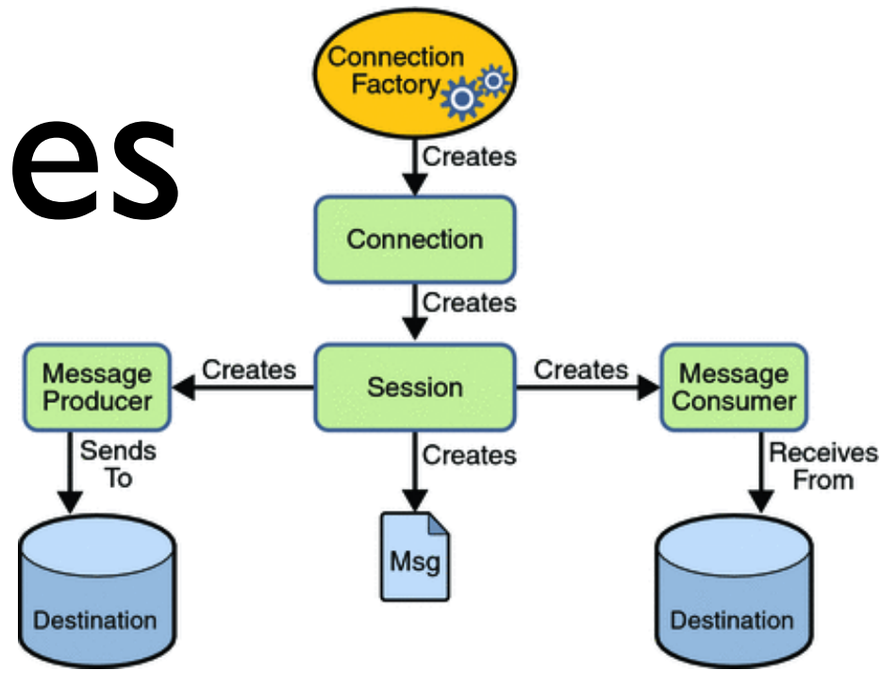
- **Localiser le driver JMS** (DI ou lookup JNDI)
- **Créer une connexion JMS**
  - à partir de la ConnectionFactory
- **Créer une session JMS**
  - à partir de la connexion
- **Localiser la destination JMS**
- **Créer un producteur ou un consommateur JMS**
- **Envoyer ou recevoir un message**

config. info. pour connexion au MOM



oracle.com

JMS : les étapes




```
public void sendOrder(Order order){
   try {
    Context context = new InitialContext();
    // 1. trouver le fournisseur
    ConnectionFactory connectionFactory =
        (ConnectionFactory)
         context.lookup("jms/QueueConnectionFactory");
   // 2. créer une connection
   Connection connection =
                connectionFactory.createConnection();
   // 3. créer une session
   Session session = connection.createSession(false,
                                Session.AUTO_ACKNOWLEDGE);
```

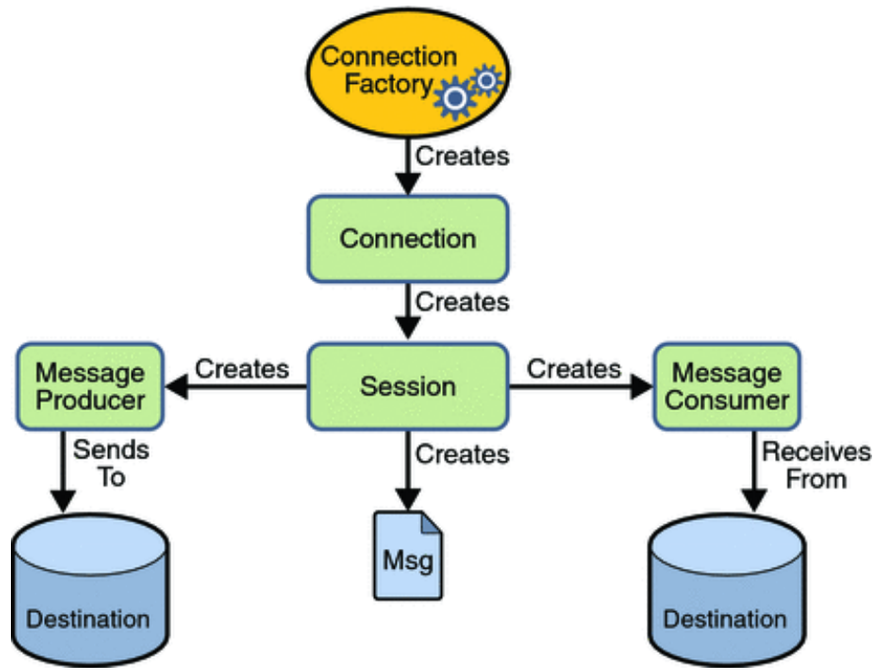
thread-safe, sharable

single threaded

acknowledge mode

(non-)transactional

```
// 4. trouver la destination
Destination orderQueue =
    (Destination)
    context.lookup("jms/OrderQueue");
// 5. créer un producer/consumer
MessageProducer producer =
    session.createProducer(orderQueue);
// 6. envoyer un message
ObjectMessage message =
    session.createObjectMessage();
    message.setObject(order);
    producer.send(message);
    // clean-up
    session.close();
    connection.close();
    connection = null;
 } catch (Throwable ex) {
        ex.printStackTrace();
 }
}
```



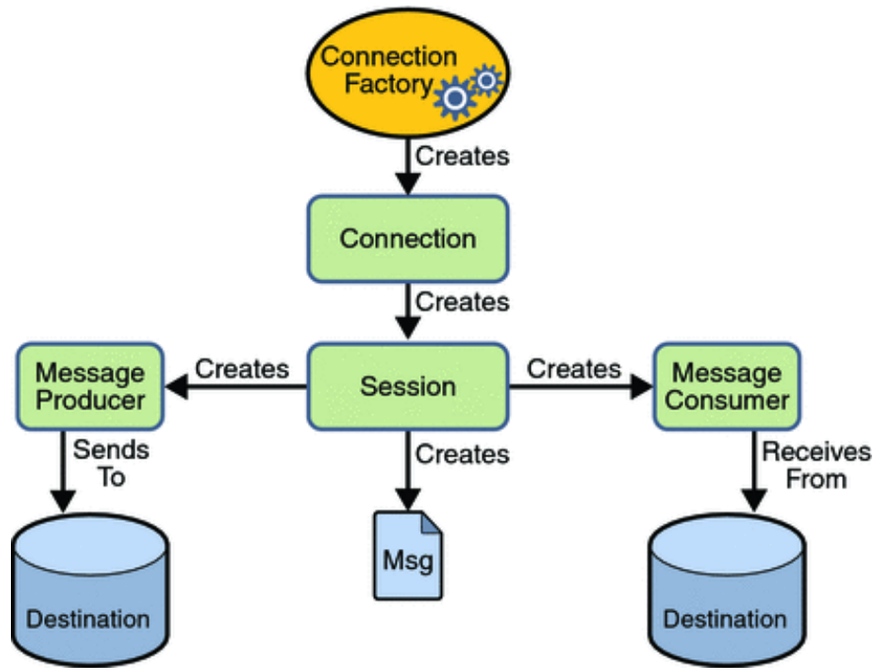
JMS avec DI

```
// 1. trouver le fournisseur
@Resource(lookup = "jms/QueueConnectionFactory")
private ConnectionFactory connectionFactory;
// 4. trouver la destination
@Resource(lookup = "jms/OrderQueue")
private Destination orderQueue;

public void sendOrder(Order order) {
   try {
    // 2. créer une connection
    Connection connection =
                  connectionFactory.createConnection();
    // 3. créer une session
    Session session = connection.createSession(false,
                                  Session.AUTO_ACKNOWLEDGE);
```

@PostConstruct

```
// 5. créer un producer/consumer
MessageProducer producer =
    session.createProducer(orderQueue);
// 6. envoyer un message
ObjectMessage message =
    session.createObjectMessage();
message.setObject(order);
producer.send(message);
        // clean-up
        session.close();
        connection.close();
        connection = null;
  } catch (Throwable ex) {
            ex.printStackTrace();
  }
}
```



Inconvénients (`JMS 1.1`)

- une `ConnectionFactory` doit être créée
- une destination doit être créée
- des objets intermédiaires doivent être créée `ConnectionFactory - Connection - Session - MessageProducer - Message`
- des exceptions peuvent être levées et donc on doit utiliser un try/catch

JMS 2.x

```
@Inject
JMSContext context;

@Resource(lookup = "jms/OrderQueue")
Destination orderQueue;

@Override
public void sendOrder(Order order) {
    context.createProducer().send(orderQueue, order);
}
```

Avantages JMS 2.x

- L' interface `JMSContext` combine les fonctionnalités de `Connection` et `Session`
- une `ConnectionFactory` (`java:comp/DefaultJMSConnectionFactory`) par défaut est utilisée si pas spécifiée explicitement
- La création d'un `JMSConsumer` démarre la connexion (pas besoin de démarrage explicite)

Avantages JMS 2.x

- `JMSContext`, `Session`, `Connection`, `JMSProducer` et `JMSConsumer` sont `AutoCloseable` (fermées automatiquement avec try-with-resources)
- nouvelle exception `JMSRuntimeException` (pas de try/catch)
- pas besoin de créer un objet `Message` - le corps du message peut être un argument de la méthode `send()`

JMS : les interfaces

PARENT INTERFACE	POINT-TO-POINT	PUB/SUB
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

Message

Message header

`JMSCorrelationID,JMSReplyTo,JMSMessageID`

Message properties

ex: `message.setBooleanProperty("Urgent", true);`

Message body

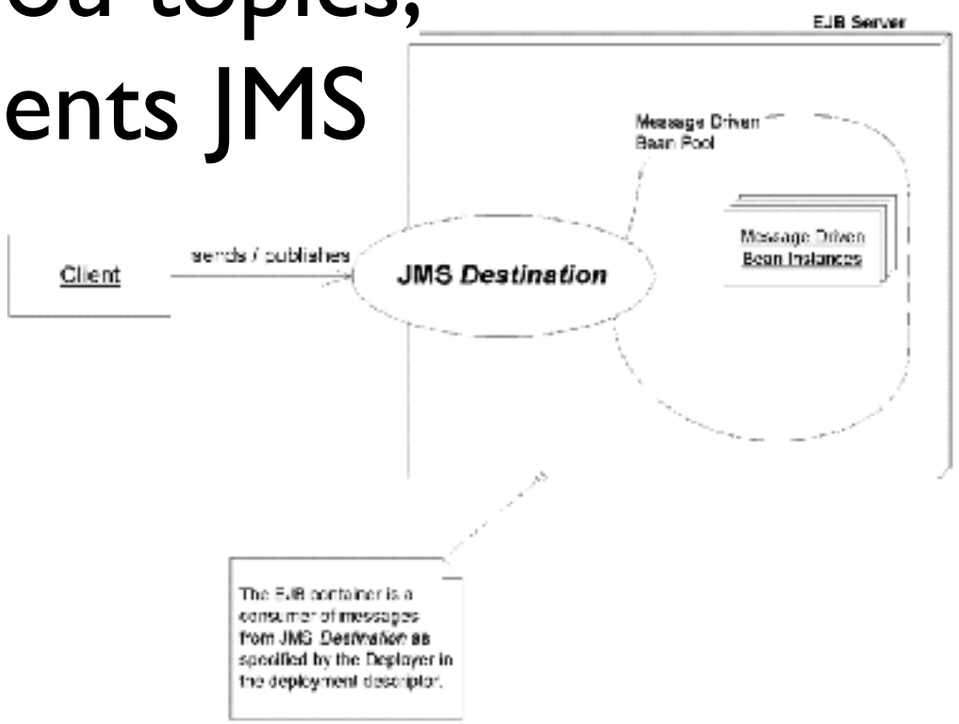
`Message` → `BytesMessage, ObjectMessage, TextMessage, StreamMessage ou MapMessage`

Consommer les messages

- Similaire à la production
  - configuration et connexion à faire à la main
  - gestion du cycle de vie
  - gestion du contexte d'exécution

Message-Driven Bean

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



Message-Driven Bean

- **Un MDB n'a pas d'interface**
  - un client n'accède pas à un MDB via une interface, il utilise l'API JMS
- **Les MDB possèdent une seule méthode**

```
public void onMessage(Message m)
```

- **Les MDB n'ont pas de valeur de retour**
  - découplés des producteurs de messages.

Pourquoi MDBs?

- **mutithreading**
  - MDB pooling
- **code simplifié**
  - trouver factory et destination, création connexions, ...
- **démarrage de la consommation de messages**
  - automatique vs. manuel

Règles MDBeans

- implante listener (`implements MessageListener`)
- classe concrète - pas `final`, pas `abstract`
- implante les méthodes de `MessageListener` (`public`, pas `static, final`)
- NE peut pas hériter un autre POJO
- déclarée `public`
- un constructeur sans arguments
- ne pas lever des exceptions

JMS 1.1

```
@MessageDriven(name="OrderProcessor",
    mappedName = "jms/OrderQueue",
    activationConfig = {
     @ActivationConfigProperty(
          propertyName = "acknowledgeMode",
          propertyValue = "Auto-acknowledge"),
     @ActivationConfigProperty(
          propertyName = "destinationType",
          propertyValue = "javax.jms.Topic"),
     @ActivationConfigProperty(
          propertyName = "destinationName",
          propertyValue = "jms/OrderTopic")
})
public class OrderMDB implements MessageListener {
     ...
}
```

glassfish specific

JMS 2.x

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "jms/OrderQueue")
})
public class OrderMDB implements MessageListener {
    ...
}
```

Interface @MessageDriven

messageListenerInterface = "javax.jms.MessageListener"

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface MessageDriven {
  String name() default "";
  Class messageListenerInterface default Object.class;
  ActivationConfigProperty[] activationConfig()
      default {};
  String mappedName();
  String description();
}

public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

JMS 1.1

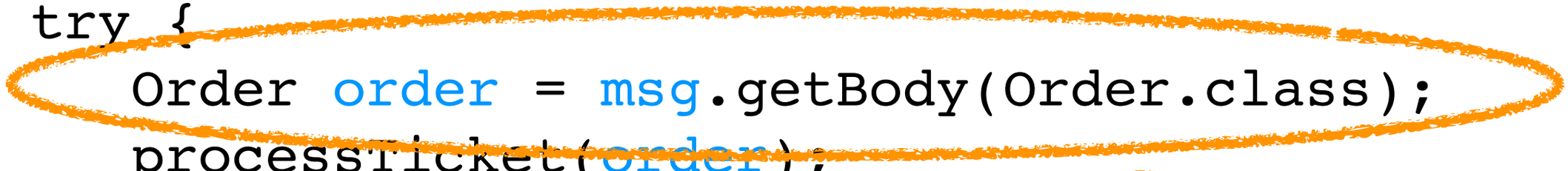
```
@MessageDriven(name="OrderProcessor",...)
public class OrderMDB implements MessageListener {
    @Resource
    private MessageDrivenContext context;

    @Override
    public void onMessage(Message msg) {
      try {
        ObjectMessage objMessage = (ObjectMessage) msg;
        Order order = (Order) objMessage.getObject();
        processTicket(order);
      }catch (JMSException jmse) {
        jmse.printStackTrace();
        context.setRollbackOnly();
      }
    }...
```

JMS 2.x

```
@MessageDriven(...)
public class OrderMDB implements MessageListener {
    @Resource
    private MessageDrivenContext context;

    @Override
    public void onMessage(Message msg) {
      try {
          Order order = msg.getBody(Order.class);
          processTicket(order);
      }catch (JMSException jmse) {
          jmse.printStackTrace();
          context.setRollbackOnly();
      }
}...
```



casting automatique

ActivationConfigProperty

- **acknowledgeMode=AUTO_ACKNOWLEDGE**
  - la session confirme automatiquement la réception quand le message a été traité (message enlevé de la queue)
- **subscriptionDurability (pour Topic)**
  - Durable = reçoit tous les messages, même si l'abonné est inactif (message délivré lorsque l'abonné sera de nouveau actif)
  - Nondurable = messages perdus lorsque abonné inactif

ActivationConfigProperty

- **messageSelector**
  - filtrer les messages reçus
- **exemple:**

```
@ActivationConfigProperty(
    propertyName="messageSelector",
    propertyValue="Urgent = TRUE"
)
```

Cycle de vie - MDB

container décide
que de nouvelles instances
sont nécessaires

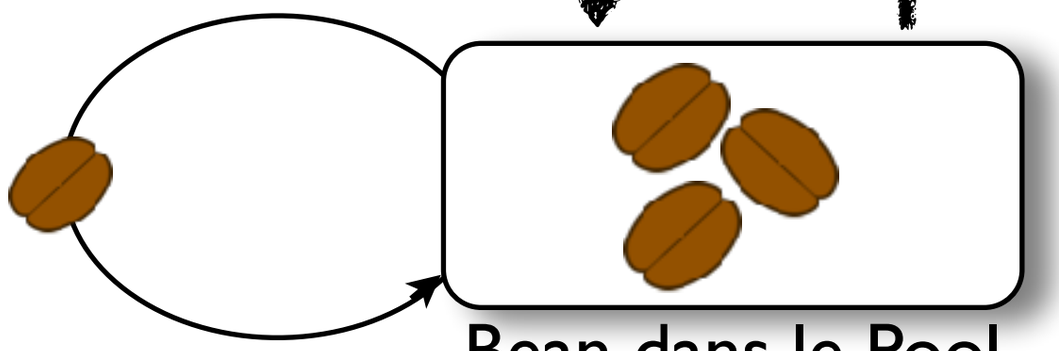
Bean n'existe pas

container décide que
moins d'instances sont
nécessaires

création instance
injection dépendances
@PostConstruct

@PreDestroy
destruction instance

onMessage()



Bean dans le Pool

Best practices

- Choisir avec attention les modèles de messaging
  - écrire du code indépendant du domaine
- Ecrire du code modulaire
  - pas de logique métier dans le listener
- Utiliser les filtres si une seule queue
- Choisir avec attention les types des messages
- Configurer la taille du Pool MDB
- Attention aux messages empoisonnés

Messages empoisonnés

`ClassCastException` → message remis dans la queue (et relu) !!!

```
try {
    ObjectMessage objectMessage =
                    (ObjectMessage) message;
    Order order =
      (Order) objectMessage.getObject();
    processTicket(order);
} catch (JMSException jmse) {
    jmse.printStackTrace();
    context.setRollbackOnly();
}
```

- compteur “redelivery” et queues “dead message”

Beans “Asynchrones”

Beans @Asychronous

- permet l'exécution asynchrone de méthodes dans des threads séparés
- les transactions ne sont pas propagées aux méthodes asynchrones (une nouvelle en est lancée)
- n'offrent pas des garanties de fiabilité
- couplage fort (entre l'appelant et l'appelé)

Utilisation

- fire and forget
  - méthodes `void`
  - qui ne peuvent pas lever des exceptions
- **fire and check back later for an answer**
  - méthodes avec un résultat `Future<V>`
  - qui peuvent lever des exceptions (envelopper par le container dans `ExecutionException`)

```
@RequestScoped
public class RequestBean {
   @EJB
   private MailSenderBean mailSender;
   @EJB
   private RoomReservationBean roomBooker;

   public void processRequest() {
       Future<Integer> roomNumber;
       int room = -1;
       try {
          mailSender.sendMail(this.request);
          roomNumber = roomBooker.bookRoom(this.request);
          ...
          if(tooLate) { roomNumber.cancel(true); }
          room = roomNumber.get();
       } catch (Exception ex) {}
       ...
   }
}
```

appels à des méthodes asynchrones

abandonner

bloqué en attente du résultat

```java
@Target(value = {ElementType.METHOD, ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Asynchronous {
}
```

```java
@Stateless
public class MailSenderBean {

    @Asynchronous
    public void sendMail(String message) {
        ...
    }
    ...
}
```

```
@Stateless
public class RoomReservationBean {

   @Resource
   private SessionContext sessionContext;

   @Asynchronous
   public Future<Integer> bookRoom(String message) {
         int roomNumber;
         ...
         if (sessionContext.wasCancelCalled()) {
               // cancel booking
               roomNumber = -1;
         }
         return new AsyncResult<Integer>(roomNumber);
   }
   ...
}
```

implantation Future

Utilisation

- **impact sur les performances**
  - l'empreinte ressource des threads est importante
- **utiliser seulement si nécessaire**
  - attention aux fuites (des appels non-terminés)
  - utilisation de cancel pour opérations trop longues

Questions